+

COMP3311 Database Management Systems

Spring 2022

香港科技大學
THE HONG KONG UNIVERSITY OF SCIENCE AND TECHNOLOGY

SQL

Prof Xiaofang Zhou

+ SQL

- SQL is considered one of the major reasons for the success of relational model in the database industry
- It provides an industry wide standard for database access
  - In practice different product support different dialects of SQL
- It is a declarative language for users to specify what the result of the query should be, DBMS decides operations and order of execution
- SQL is designed for data definition, data manipulation, and data control, powerful enough to retrieve any piece of data from database

+ Three Types of SQL Statements

- Data definition language (DDL)
  - Statements to define the database schema
- Data manipulation language (DML)
  - Statements to manipulate the database instances
- Data control language (DCL)
  - Statements such as GRANT and REVOKE which mainly deal with the rights, permissions, and other controls of the database system

… SQL is used in all commercial RDBMSs
… different RDBMSs may have different syntax/features of SQL

+ History of SQL

- SEQEL (Structured English Query Language) was first introduced from IBM Research for SYSTEM R in 1970s
- SQL:1986 introduced by ANSI and ISO
- SQL:1992 was a revised and much expanded version
- SQL:1999 extends SQL with object-oriented concepts
- SQL:2003 introduced XML features
- SQL:2016 adds JSON

+ SQL Outline

- DDL
  - Database Definition
- DML
  - Database Modification
  - SELECT Statements
- Database Views
- More on Database Constraints

+ Data Definition Language (DDL)

The SQL DDL allows the specification of:

- The schema for each relation and their attributes
- The types of values associated with each attribute
  - char, varchar, int, smallint, numeric, real, double precision, float, date, time, timestamp…
- User-defined types and domains
- Integrity constraints
  - domain, key, foreign key, general
- The physical storage structure of each relation on disk
- The set of indexes to be maintained for each relation

+ DDL - Data Definition Language

- Basic SQL DDL Statements
  - CREATE TABLE
  - DROP TABLE
  - ALTER TABLE
  - CREATE DOMAIN

.. create/dop/modify views and indexes are also part of DDL, to be discussed later

… we don't discuss the specification of physical storage and transactions in this course

+ CREATE TABLE

- It creates a new relation, by specifying its name, attributes and constraints
  - The definitions are recorded the table definition in the system catalog (aka data dictionary)
  - The key, entity and referential integrity constraints are specified within the statement
  - The domain constraint is specified for each attribute by giving a valid (e.g., SQL99) data type and (optionally) excluding NULL from the domain
    - Valid data types include INT, CHAR, DATE, DECIMAL, etc.
    - Data type of an attribute can be specified directly or by declaring a domain (CREATE DOMAIN)

+ CREATE TABLE Syntax

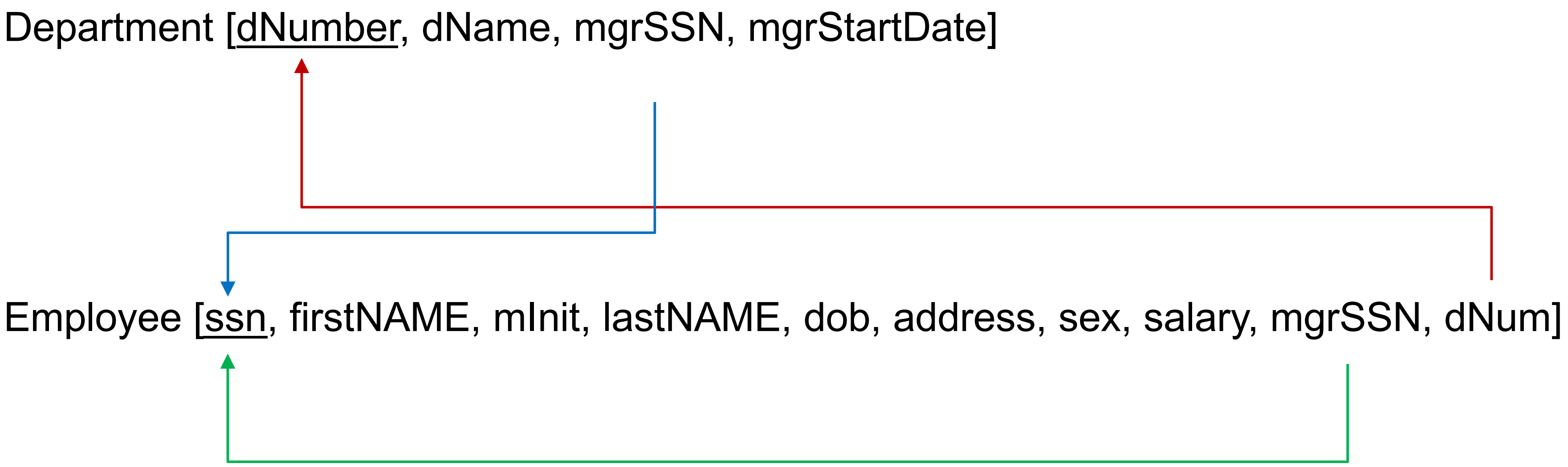
```
CREATE TABLE <table name>
  (<column name> <column type> [<attribute constraint>]
  {, <column name> <column type>  [<attribute constraint>] }
  [<table constraint> {, <table constraint>} ] );
```

Notations:

KEY WORDS, “,” “(” “)” “;”
<name>,
{repeat 0-n times},
[optional]

…you should keep an SQL syntax quick reference handy

+ CREATE TABLE Example



Department [dNumber, dName, mgrSSN, mgrStartDate]

Employee [ssn, firstNAME, mInit, lastNAME, dob, address, sex, salary, mgrSSN, dNum]

☞The domain type of an attribute is enforced by the DBMS whenever tuples are added or modified, together with other constraints

… the domain type of each attribute needs to be specified (not captured in ER diagrams)

+ CREATE TABLE Example (1)

```
CREATE TABLE Employee
  (    firstName          VARCHAR (15)         NOT NULL,
       mInit              CHAR,
       lastName           VARCHAR (15)         NOT NULL,
       ssn                CHAR (9)             NOT NULL,
       dob                DATE,
       address            VARCHAR (30),
       sex                CHAR,
       salary             DECIMAL (10, 2),
       mgrSSN             CHAR (9),
       dNum               INT                  NOT NULL,
  PRIMARY KEY (ssn),
  FOREIGN KEY (mgrSSN) REFERENCES Employee (ssn),
  FOREIGN KEY (dNum) REFERENCES Department(dNumber) );
```

+ CREATE TABLE Example (2)

Constraints can be given a name:

```
CREATE TABLE Employee
  (     firstName          VARCHAR (15)        NOT NULL,
        …….
        ssn                CHAR (9)            NOT NULL,
        mgrSSN             CHAR (9),
        dNum               INT                 NOT NULL,
  CONSTRAINT empPK PRIMARY KEY (ssn),
  CONSTRAINT smpMgrFK FOREIGN KEY (mgrSSN)
             REFERENCES Employee (ssn),
  CONSTRAINT empDNumFK FOREIGN KEY (dNum)
             REFERENCES Department (dNumber)
  );
```

+ Basic Types

char(n)	Fixed length character string with length n
varchar(n)	Variable-length character string with maximum length n
int	An integer (a finite subset of the integers that is machine-dependent)
smallint	A small integer (a machine-dependent subset of the integer domain type)
numeric(p, d)	A number with a total of p digits and d digits to the right of the decimal point
real	Floating-point and double-precision floating-point
double precision	Numbers with machine-dependent precision
float(n)	Floating point number, with user-specified precision of at least n digits
date	A date containing a (4 digit) year, month and day of month
time	The time of day, in hours, minutes and seconds
timestamp	A combination of date and time

…Null values are allowed in all the domain types

User-defined Types

- Using CREATE TYPE clause

```
create type id_type as object (id numeric(10));
```

```
create type location_type as object (
    address varchar(100),
    country varchar(20)
);
```

```
create table customer (
    id id_type, // Oracle disallows a user-defined type as PK
    add location_type
);
```

☞ Not all relational systems support user-defined types

… this is a privileged operation
… complex user-defined types with operations/methods are possible
… object type inheritance is possible too

+ User-defined Domains

- The CREATE DOMAIN clause is used to define a new domain

```
create domain hourly_wage numeric(5,2) not null default 18;
```

```
create domain age int check (value > 15 and value < 75);
```

- Differences between user-defined types and domains:
  - Domains cannot be a composite type; types can
  - Domains can have constraints specified on them and can have default values defined for variables of the domain; types cannot
  - Domains are not strongly typed; types are strongly typed

☞ Not all relational systems support user-defined domains

... "strongly typed" enforces strict restrictions on intermixing of values with different data types

CREATE TABLE Example (3)

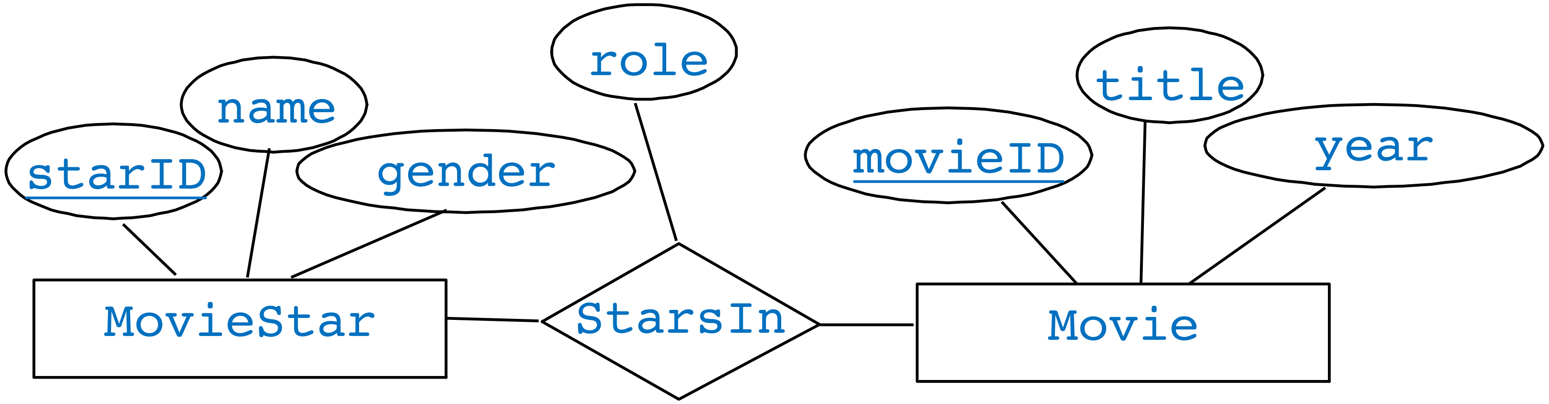
- A referential triggered action clause can be attached to a foreign key constraint, to specify the action to take if a referenced tuple is deleted, or a referenced primary key value is modified

ON DELETE SET NULL | SET DEFAULT | CASCADE

ON UPDATE SET NULL | SET DEFAULT | CASCADE

```
CREATE TABLE Employee
(   .......
  dNum              INT       NOT NULL DEFAULT 100,
  .......
  FOREIGN KEY (dNum) REFERENCES Department (dNumber)
          ON DELETE SET DEFAULT
          ON UPDATE CASCADE);
```

+ Another Example



StarsIn[movieID, starID, role]

StarsIn.starID → MovieStar.starID

StarsIn.movieID → Movies.movieID

```
CREATE TABLE StarsIn (
  starID      INT,
  movieID  INT,
  role          CHAR(20),
  PRIMARY KEY (starID, movieID),
  FOREIGN KEY (starID)  REFERENCES movieStar,
  FOREIGN KEY (movieID)  REFERENCES Movie);
```

+ Enforcing Referential Integrity

- `movieID` in `StarsIn` is a foreign key that references `Movie`
  - `StarsIn.movieID` → `Movie.movieID`
- What should be done if a `Movie` tuple is deleted, and there is a `StarsIn` tuple refers to it?
  1. Delete all roles that refer to it?
  2. Disallow the deletion of the movie?
  3. Set `moveID` in `StartsIn` tuples that refer to it to null?
  4. Set `moveID` in `StartsIn` tuples that refer to it to default value?

By default, no action is taken, and the delete/update is rejected.
Other actions include :

ON DELETE SET NULL | SET DEFAULT | CASCADE
ON UPDATE SET NULL | SET DEFAULT | CASCADE

+ Create Table with FK Actions

```
CREATE TABLE  StarsIn (
  starID      INTEGER,
  movieID   INTEGER,
  role          CHAR(20),

  PRIMARY KEY (starID, movieID),
  FOREIGN KEY (starID)  REFERENCES MovieStar
       ON DELETE CASCADE
       ON UPDATE CASCADE,
  FOREIGN KEY (movieID)  REFERENCES Movie
        ON DELETE SET NULL
        ON UPDATE CASCADE);
```

+ Question:

Consider the following table definition.

```
CREATE TABLE  ParkingPermit  (
    pID      INT,
    staffID  INT,  …

    PRIMARY KEY (pID),
    FOREIGN KEY (staffID) REFERENCES Staff  ON DELETE CASCADE);
```

Assume there is a tuple with `pID` = 1000 and `staffID` = 5678 in the table, choose the best answer

1. If the row for `staffID` value 5678 in `Staff` is deleted, then only the row with `pID = 1000` in `ParkingPermit` is automatically deleted

2. If the row with `staffID` value 5678 in `Staff` is deleted, then all rows with `staffID=5678` in `ParkingPermit` are automatically deleted

3. Both of the above

Based on the instruction given in the table definition, only option 2 is correct

+ ALTER TABLE

- ALTER TABLE command is used for schema evolution, that is the definition of a table created using the CREATE TABLE command, can be changed using the ALTER TABLE command
- Alter table actions include
  - Adding or dropping a column
  - Changing a column definition
  - Adding or dropping constraints

+ ALTER TABLE Syntax

```
ALTER TABLE <table name>
  ADD <column name> <column type> [<attribute constraint>]
        {, <column name> <column type> [<attribute constraint>] }
  | DROP <column name> [CASCADE]
  | ALTER <column name> <column-options>
  | ADD <constraint name> <constraint-options>
  | DROP <constraint name> [CASCADE];
```

☞ Commercial products have variations!

… to alter a constraint, it must be dropped and added again
… you can drop a PK constraint!
… FK doesn’t have to reference to PK but can only to UNIQUE attributes; but in practice they always do

+ ALTER TABLE Examples

- To add an attribute

```
ALTER TABLE Employee ADD job VARCHAR(12);
```

  - Note: values for the added attribute in all tuples will be initially NULL, so NOT NULL cannot be specified

- To drop an attribute

```
ALTER TABLE Employee DROP address;
```

  - Note: drop at attribute which has been used by other tables in their FK references, CASCADE can be used

- To drop a constraint (constraint must have been given a name when it was specified)

```
ALTER TABLE Employee DROP CONSTRAINT
empPK CASCADE;
```

…

+ Questions:

1. When adding a column, what happens to the existing records for their values of the added column?
2. When you drop a constraint, does it delete any data instances?
3. When you drop a PK constraint, do you have to specify cascade options?
4. When you drop an FK constraint, do you have to specify cascade options?
5. When two tables have mutual FKs, how do you insert data?

+ What We Discussed So Far

- Foundations:
  - ER Disarms → Relational schemas
  - RA
- SQL = DDL + DML + DCL
  - DDL: create/modify schema and constraints
    - User-defined types/domains
    - Cascade options
    - DROP TABLE: DROP TABLE <table name> [CASCADE];
      - Drops all constraints defined on the table including constraints in other tables which reference this table if CASCADE option is used
      - Deletes all tuples within the table
      - Removes the table definition from the system catalog
  - DML: manipulating instances
    - INSERT/DELETE/UPDATE
    - SELECT

+ SQL Outline

- DDL
  - Database Definition
- DML
  - Database Modification
  - SELECT Statements
- Database Views
- More on Database Constraints

+ Modifying Database Instances

- Statements to modify database instances
  - Those for modifying database schemas are called DDL
- Basic SQL Statements for modifying records
  - INSERT
  - DELETE
  - UPDATE

+ INSERT Statement

Used to add tuples to an existing relation

- Single Tuple INSERT
  - Specify the relation's name and a list of values for the tuple
  - Values are listed in the same order as the attributes were specified in the CREATE TABLE command
  - User may specify explicit attribute names that correspond to the values provided in the insert statement
    - The attributes not included cannot have the NOT NULL constraint
- Multiple Tuple INSERT
  - By separating each tuple’s list of values with commas
  - By loading the result of a query

+ INSERT Syntax

```
INSERT INTO <table name>
    [(<column name> {, <column name> })]
    VALUES (<constant value> {,<constant value> })
                {(<constant value> {,<constant value> })}
    | <select statement>;
```

…syntax check will be performed

+ INSERT Example: From Values

```
INSERT INTO Employee
VALUES
(‘653298695’, ‘Richard Marini’, ‘30-DEC-1995’,
  ‘6B/98 University Road, Sai Kung, NT’, ‘M’,37000, 4,  ‘987654321’);
```

```
INSERT INTO Employee(name, ssn)
VALUES (‘Jane Chow’, ‘987654321’),
             (‘Emily Li’, NULL);
```

```
Employee[ssn,name,dob,address,sex,salary,dNum,mgrSSN]
```

… syntax and integrity checks will be performed

… values with quotation or not can be flexible (i.e., supporting automatic data type convention)

+ INSERT Example: From Queries

```
INSERT INTO DeptInfo(dName, numOfEmployees, totalSalary)
          SELECT          dName, COUNT(*), SUM(salary)
          FROM            Department, Employee
          WHERE           dNumber = dNum
          GROUP BY        dName ;
```

```
Employee[ssn,name,dob,address,sex,salary,dNum,mgrSSN]
Department[dName, dNumber, mgrSSN, mgrStartDate]
```

…we will discuss SELECT statement later

+ DELETE Statement

Used to remove some existing tuples from a relation

- Tuples are selected to delete from a single table
- Deletion may propagate to other tables if referential triggered actions are specified in the referential integrity constraints

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

…DELETE all tuples doesn’t equivalent to DROP a table

+ DELETE Example

```
DELETE FROM Employee
WHERE dob < ‘1-JAN-1800’;
```

```
DELETE FROM Employee
WHERE dNum = 5;
```

```
DELETE FROM Employee
WHERE salary >= 100000;
```

```
DELETE FROM Employee;
```

```
Employee[ssn,name,dob,address,sex,salary,dNum,mgrSSN]
```

… a single DELETE statement may delete zero, one, several or all tuples from a table

+ UPDATE Statement

Used to modify attribute values of one or more selected tuples in a relation

- Tuples are selected for update from a single table
- Updating a primary key value may propagate to other tables if referential triggered actions are specified in the referential integrity constraints

```
UPDATE <table name>
SET <column name> = <value expression>
    {, <column name> = <value expression>}
[WHERE <select condition>];
```

+ UPDATE Example

```
UPDATE Employee
SET         salary = salary * 1.1
WHERE  lastName = ‘McGowen’;
```

```
Employee[ssn,name,dob,address,sex,salary,dNum,mgrSSN]
```

… a single UPDATE statement may change zero, one, several or all tuples from a table

+ More on DELETE/UPDATE

- Conceptually, deletion/update are done in two steps
  - Find the tuples to delete/update
  - Delete/update the tuples found

- Notes
  - They can only be used to delete/update records in one table, but can cause cascading changes in other tables
  - The where-clause can be as complex as in a SELECT statement, including using multiple relations
  - For UPDATE, only the values before the changes are considered in the find operation

+ SQL Outline

- DDL
  - Database Definition
- DML
  - Database Modification
  - SELECT Statements
- Database Views
- More on Database Constraints

+ SELECT Statement

- SQL has one basic statement for retrieving information from the database
- In the SELECT statement, users specify what the result of the query should be, and the DBMS decides the operations and order of execution, thus SQL queries are declarative

+ SELECT Statements

- Simple SELECT queries
- Join queries
- Ordering your results
- Aggregation and Grouping
- Set operations
- Renaming

+ SELECT Basic Syntax

```
SELECT <attribute list>

FROM <table list>

[WHERE <condition>] ;
```

- `<attribute list>` is a list of attribute names whose values are to be retrieved by the query
- `<table list>` is a list of relation names required to process the query
- `<condition>` is a conditional (Boolean) expression that identifies the tuples to be retrieved by the query

+ Simple SELECT Example

```
SELECT       empAddress
FROM         Employee
WHERE        empName = ‘Joe Bates’;
```

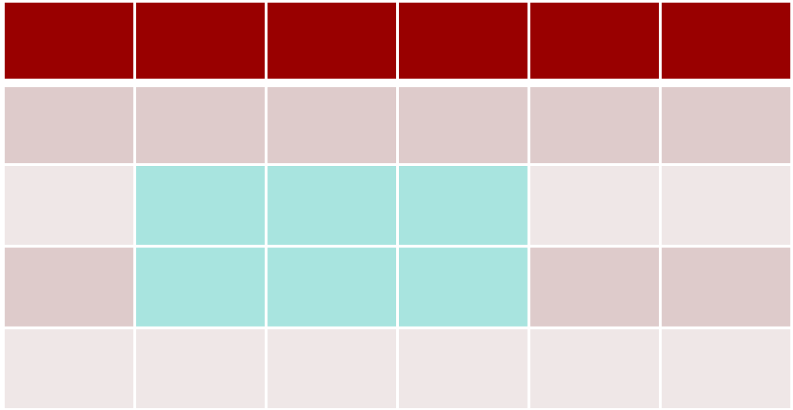
Employee

empName	empAddress	department
Nicole Smith	1 Pine Road	CSE
Joe Bates	32 Chandler Rd	ECE

Query results

empAddress
32 Chandler Rd

+ SPJ Queries



- Selection (WHERE clause)
  - Horizontally select tuples
- Projection (SELECT clause)
  - Vertically select the attributes
- Join (FROM clause)
  - Combine tuples from different relations for the search purposes

Conceptually, the Cartesian product of all the relations are generated, followed by applying the selection operation and then projection operation to produce the final results

+ Cartesian Product

- R1 X R2: every row in R1 is combined with every row in R2 to form tuples in the result relation
- The schema of R1 X R2 is the concatenation of all the columns from R1 and all the columns from R2

X	Y	Z
x1	y1	z1
x2	y2	z2

X

A	B	C
a1	b1	c1
a2	b2	c2

X	Y	Z	A	B	C
x1	y1	z1	a1	b1	c1
x1	y1	z1	a2	b2	c2
x2	y2	z2	a1	b2	c1
x2	y2	z2	a2	b2	c2

+ Selection in SQL

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

- There are two types of conditions
  - <join condition> links tuples from the tables
  - <search condition> is a conditional (Boolean) expression that identifies the tuples to be retrieved by the query
  - Without any join condition, it means Cartesian Product
  - While joins can occur between any pair of attributes, most joins occur between primary keys and foreign keys

+ Selection Examples

List the names of employees working in department number 5.

```
SELECT     name
FROM       Employee
WHERE      dNum = 5;
```

List the names of employees who work in department 4 and earn over $25000, or work in department 5 and earn over $30000.

```
SELECT     name
FROM       Employee
WHERE      (dNum = 4 AND salary > 25000)  OR
           (dNum = 5 AND salary > 30000);
```

`Employee[ssn,name,dob,address,sex,salary,dNum,mgrSSN]`

+ Join Query Example

List the names of employees working in the "Research" department.

```
SELECT    name
FROM      Employee, Department
WHERE     dNum = dNumber AND dName = 'Research';
```

```
Employee[ssn,name,dob,address,sex,salary,dNum,mgrSSN]
Department[dName, dNumber, mgrSSN, mgrStartDate]
```

+ Complex WHERE Conditions

- Substring comparisons
  - LIKE
    - WHERE Address LIKE ‘%Sai Kung%’
    - WHERE mgrStartDate LIKE ‘_ _ / 0 5 / _ _’
  - IN
    - WHERE lastName IN (‘Jones’, ‘Wong’, ‘Chow’)
  - IS
    - WHERE dNum IS NULL

= null (<> null): not defined (always false)

- Arithmetic operators and functions
  - +, -, *, /, date and time functions, etc.
    - WHERE salary / 7.8 > 50000
    - WHERE datediff(year, getdate(),dob) > 55
  - BETWEEN
    - WHERE salary BETWEEN 10000 AND 30000

% : place holder for 0 or more characters

_ : place holder for a single character

+ Using Complex Conditions

List the names of employees in a research department with a salary between 40 and 60K.

```
SELECT    name
FROM      Employee, Department
WHERE     dNum = dNumber AND dName LIKE ‘%Research%’
           AND salary BETWEEN 40000 AND 60000;
```

```
Employee[ssn,name,dob,address,sex,salary,dNum,mgrSSN]
Department[dName, dNumber, mgrSSN, mgrStartDate]
```

+ Projection Examples

List the names, ssn and date of birth of all employees.

```
SELECT name, ssn, dob
FROM   Employee;
```

List all details of employees and their corresponding departments.

```
SELECT    *
FROM      Employee, Department
WHERE     dNum = dNumber;
```

```
Employee[ssn,name,dob,address,sex,salary,dNum,mgrSSN]
Department[dName, dNumber, mgrSSN, mgrStartDate]
```

+ Projection and Duplicates

- Theoretical, the projection operator produces a relation, which is a set of tuples
  - A set, by definition, cannot contain duplicates
- The SQL Projection can produce multisets (aka bags), which are tables with duplicate tuples
- Set semantics can be enforced in SQL using the DISTINCT option

… the default behavior for an SQL query is to return a bag, unless it is a set query

+ Projection Example With DIST

List the salaries and departments of all employees.

```
SELECT    salary, dDum
FROM      Employee;
```

```
SELECT    DISTINCT salary, dDum
FROM      Employee;
```

salary	dNum
30000	5
40000	5
25000	4
43000	4
25000	5
25000	4
55000	1

`Employee[ssn,name,dob,address,sex,salary,dNum,mgrSSN]`

+ Question:

List the details of all employees.

```
SELECT * FROM Employee;
```

```
SELECT DIST * FROM Employee;
```

Will there be any difference in the results?

+ Projection and Expressions

- SQL queries can also evaluate expressions and return the value of these expressions together with the projected attributes

- Expressions use standard arithmetic operators (+, -, *, /) on numeric values or attributes with numeric domains, including a constant

+ Expression Example

List the names of all employees working in department 6, their salaries and salaries with a 17% increase.

```
SELECT      name, salary, ‘Including super’,1.17 * salary
FROM        Employee
WHERE       dNum = 6;
```

name	salary	Including super	<a unique name>
Peter	10000	Including super	11700

```
SELECT      name, salary, 1.17 * salary AS ‘includingSuper’
FROM        Employee
WHERE       dNum = 6;
```

name	salary	Salary including super
Peter	10000	11700

`Employee[ssn,name,dob,address,sex,salary,dNum,mgrSSN]`

+ Sorting in SQL

```
SELECT [DISTINCT] <target list>
FROM   <table list>
[WHERE <condition>]
[ORDER BY <column>[ASC|DESC]
       {,<column>[ASC|DESC]}];
```

target list: can be a column name, an expression, or *
column: can be a column name or a column position in the SELECT list

… sorting can be done by multiple columns

+ Sorting Examples

List all employee names and salaries, ordered by salary.

```
SELECT          name, salary
FROM            Employee
ORDER BY        salary DESC;
```

List all employee names, department number and salaries, ordered by department and salary.

```
SELECT          dNum, name, salary
FROM            Employee
ORDER BY        1 ASC, 3 DESC;
```

`Employee[ssn,name,dob,address,sex,salary,dNum,mgrSSN]`

+ What is Aggregation ?

- Aggregates are functions that produce summary values from a set of tuples
- Aggregates can be applied to
  - All tuples
  - A selected set of tuples
  - Multiple groups of tuples specified by the GROUP BY clause (to be discussed)
  - Aggregation functions can be used in the SELECT clause and the HAVING clause (to be discussed)

+ SQL Aggregation Functions

- SQL provides the following built-in functions for aggregates:
  - COUNT: Counts the number of tuples that the query returns
  - SUM/AVG: Calculates the sum (average) of a set of *numeric* values
  - MAX/MIN: Returns the maximum (minimum) value from a set of values which have a *total ordering*. Note that the domain of values can be non-numeric
  - They can be used with DISTINCT

+ Aggregation Examples

Find the total and average salary of employees.

```
SELECT    AVG(salary), SUM(salary)
FROM      Employee;
```

Find the age of the youngest employee in the "Research" department.

```
SELECT    DATEDIFF(YEAR, MAX(dob), GetDate())
FROM      Employee, Department
WHERE     dNum = dNumber AND dName = 'Research';
```

```
Employee[ssn,name,dob,address,sex,salary,dNum,mgrSSN]
Department [dName, dNumber, mgrSSN, mgrStartDate]
```

+ More Aggregation Examples

Find the total number of employees in department 5.

```
SELECT          COUNT (*)
FROM            Employee
WHERE           dNum = 5;
```

Count the distinct salaries of employees in department 5.

```
SELECT          COUNT (DISTINCT salary)
FROM            Employee
WHERE           dNum = 5;
```

Note that COUNT (salary) will give the same result as COUNT (*), duplicate values are not eliminated

`Employee[ssn,name,dob,address,sex,salary,dNum,mgrSSN]`

+ Aggregation and Grouping

- An aggregation function can be applied to all rows of a table as previously shown
- However, aggregation functions are often applied to groups of rows within a table
- The GROUP BY clauses provides this functionality

+ Grouping in SQL

```
SELECT [DISTINCT] <target list>
FROM    <table list>
[WHERE <condition>]
[GROUP BY <grouping attributes> ]
[HAVING      <group conditions> ]
[ORDER BY <column> [ASC|DESC] {, <column> [ASC|DESC]}];
```

When GROUP BY is used in an SQL statement, any attribute appeared in SELECT Clause must appear either in GROUP BY clause or in an aggregation function!

+ Group By Examples

Find the total number of employees in each department.

```
SELECT      COUNT (*)
FROM        Employee;
```

```
SELECT      dNum, COUNT (*)
FROM        Employee
GROUP BY    dNum;
```

```
SELECT      dNum, COUNT (*)
FROM        Employee
GROUP BY    dNum
OODER BY    2 ASC;
```

dNum	count
5	34
4	560
1	120
3	45

dNum	count
5	34
3	45
1	120
4	560

```
Employee[ssn,name,dob,address,sex,salary,dNum,mgrSSN]
Department[dName, dNumber, mgrSSN, mgrStartDate]
```

More Group By Examples

Find the total number of employees with salary > 40000 in each department.

```
SELECT      dNum, COUNT (*)
FROM        Employee
WHERE       salary > 40000
GROUP BY    dNum;
```

Find the total number of employees in each department, for each distinct salary.

```
SELECT      dNum, salary, COUNT (*)
FROM        Employee
GROUP BY    dNum, salary;
```

`Employee[ssn,name,dob,address,sex,salary,dNum,mgrSSN]`

+ Conditions on Groups

- Conditions can be imposed on the selection of groups to be included in the query result
- The HAVING clause (following the GROUP BY clause) is used to specify these conditions, similar to the WHERE clause
- Unlike the WHERE clause, the HAVING clause can also include aggregates

+ Group By Examples with Having

Find the total number of employees in departments with more than 5 employees.

```
SELECT      dNum, COUNT (*)
FROM        Employee
GROUP BY    dNum
HAVING      COUNT (*) > 5;
```

Find the total number of employees who earn more than 40000, in departments with more than 5 employees.

```
SELECT      dNum, COUNT (*)
FROM        Employee
WHERE       salary > 40000
GROUP BY    dNum
HAVING      COUNT (*) > 5;
```

The query is wrong! It returns departments with more than 5 employees earning 40000. We will learn how to write the correct query later, which is non-trivial to write.

`Employee[ssn,name,dob,address,sex,salary,dNum,mgrSSN]`

+ What We Discussed So Far…

- SQL
  - DDL → creating/deleting/modifying schemas & constraints
  - DML part1 → creating/deleting/modifying instances
  - DML part2 → selecting instances

```
SELECT [DISTINCT] <target list>
FROM    <table list>
[WHERE <condition>]
[GROUP BY <grouping attributes> ]
[HAVING      <group conditions> ]
[ORDER BY <column> [ASC|DESC] {, <column> [ASC|DESC]}];
```

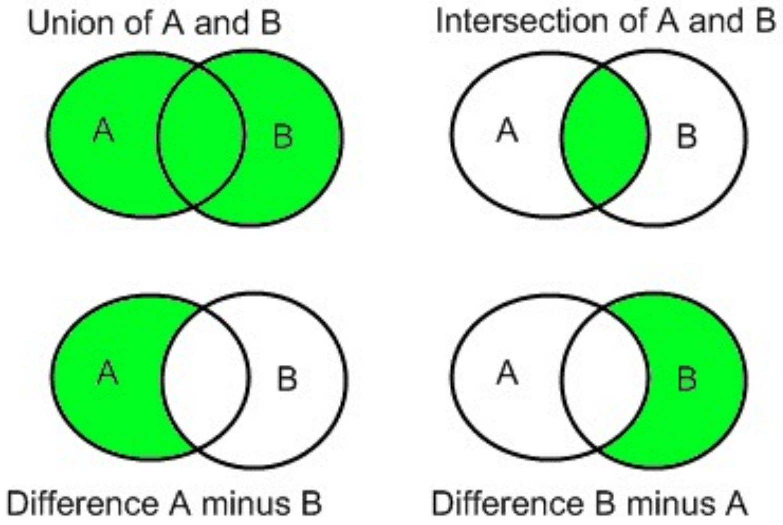
When GROUP BY is used in an SQL statement, any attribute appeared in SELECT-clause must appear either in GROUP BY clause or in an aggregation function!

+ Multiple Relation SQL Queries

- Basic Set Operations in SQL
  - Union
  - Intersection
  - Difference/Minus
- Renaming in SQL
- Joins in SQL

+ Basic Set Operators

- Relation is a set of tuples (no duplicates; no order)
- Basic set operators apply to relations
  - UNION
  - INTERSECTION
  - DIFFERENCE/MINUS
- Union compatibility
  - The same number of columns
  - Pair-wise compatible domains



Union of A and B

Intersection of A and B

Difference A minus B

Difference B minus A

$A \cup B$	commutative	$A \cup B = B \cup A$
	associative	$(A \cup B) \cup C = A \cup (B \cup C)$
$A \cap B$	commutative	$A \cap B = B \cap A$
	associative	$(A \cap B) \cap C = A \cap (B \cap C)$
$A - B$	not commutative	$A - B \neq B - A$
	not associative	$(A - B) - C \neq A - (B - C)$

Set Queries in SQL

```
SELECT eSSN FROM WorksOn
UNION
SELECT ssn FROM WorkedOn;
```

```
SELECT * FROM WorksOn
UNION
SELECT * FROM WorkedOn;
```

```
SELECT eSSN FROM WorksOn
INTERSECT
SELECT ssn FROM WorkedOn;
```

```
SELECT eSSN FROM WorksOn
MINUS
SELECT ssn FROM WorkedOn;
```

- A UNION effectively does SELECT DISTINCT
- UNION ALL: Results can contain duplicate tuples.
- This can also be applied to Intersection and Minus (but all are not supported by Oracle for interaction and minus)
- You can do (SELECT…) UNION (SELECT …) ORDER BY…

```
WorksOn[eSSN, pNo, hours]
WorkedOn[ssn, projNum, duration]
```

+ Renaming in SQL

- SQL provides the facility to rename attribute and/or table names
  - Declaring an alias

- Renaming assists in shorten names, removing ambiguity and specifying self-joins

```
SELECT CustomerName,
            CONCAT(address, ‘, ’, country) AS fullAddress
FROM    Customers;
```

```
SELECT emp.name, mgr.name AS MgrName
FROM    Employee AS emp, Employee AS mgr
WHERE emp.mgrSSN = mgr.ssn;
```

…. Oracle can omit AS for table renaming

+ Joins in SQL

- A Join used to combine related tuples from two relations into a single tuple in a new (result) relation
- Join operation is needed for organizing a search space of data
  - This is needed when information is contained in more than one relation
- Join relations are specified in the FROM Clause. When two relations are combined for a search, we need to know how the relations are combined
  - Based on Cartesian Product (denoted as X)
  - Denoted as ⋈

$$R_1 \bowtie_{\text{<join conditions>}} R_2$$

+ Cartesian Product in SQL

- Cartesian Product of two tables with no common attributes

```
SELECT    *
FROM      Employee, Project;
```

```
Employee[ssn,name,dob,address,sex,salary,dNum,mgrSSN]
Project[pNo, pName, pLocation, dNum]
```

+ Theta-Join

- A join operation is to select a subset of the Cartesian Product
- The most general type of join is called θ-join
- Join condition allows 6 logical operators {=, ≠, <, ≤, >, ≥}
- Be careful about the operators for strings and other data types where “=“ may not do what you expect
- And be careful about NULL values

$$\theta \in \{=, \neq, <, >, \leq, \geq\}$$

+ Equi-Join

- Joins tuples from two relations when they have the same value for some pair(s) of designated attributes
- The specification is termed a “join condition”
- With equi-join, the join condition only has equality comparisons

+ Join Examples

List the names of the managers of each department.

```
SELECT    E.name, D.dName
FROM      Department AS D, Employee AS E
WHERE     D.mgrSSN = E.ssn;
```

For each employee, list the employees who earn more than his/her manager.

```
SELECT    A.name, B.name
FROM      Employee A, Employee B
WHERE     A.salary > B.salary AND A.mgrSNN = B.ssn;
```

```
Employee[ssn,name,dob,address,sex,salary,dNum,mgrSSN]
Department [dName, dNumber, mgrSSN, mgrStartDate]
```

+ Natural Join

- Similar to equi-join except that the attributes that are used for the join are those that have the same name in each relation
  - Join attributes (and join condition) are not explicitly specified
  - Duplicate attributes are eliminated
  - No special SQL syntax for this join (some products can use keyword such as NATURAL JOIN in FROM-clause)
  - Natural join occurs frequently (due to table normalization)

+ Natural Join Example

empID	dNum
1	101
2	210
3	101
4	300

dNum	head
101	Jane
210	Peter
300	Simon
400	Anna

empID	dNum	head
1	101	Jane
2	210	Peter
3	101	Jane
4	300	Simon

Inner and Outer Joins

- Inner Join
  - This is the default join, in which a tuple is included in the result relation only if matching tuples exist in both relations
- Outer Join
  - Outer joins can include the tuples that do not satisfy the join condition, i.e, a matching tuple does not exist, which is indicated by a NULL value
  - Full Outer Join: Includes all rows from both tables
  - Left Outer Join: Includes all rows from first table
  - Right Outer Join: Includes all rows from second table
  - No special SQL syntax for this join (some products can use keyword such as FULL OUTER JOIN etc in FROM-clause)

```
<table> NATURAL [ { LEFT | RIGHT } [ OUTER ] | INNER ] JOIN <table>
```

+ Nested Queries in SQL

- Concept of Nested (sub) queries
- Correlated and non-correlated types
- Joins vs. Sub-queries
- Using the IN Operator
- Using the EXISTS Function
- Rules of sub-query construction

+ Nested SQL Queries

- A nested query (often termed subquery) is a query that appears within a query
  - Inside the WHERE clause of another SELECT statement
  - Inside an INSERT, UPDATE or DELETE statement
  - Nesting can occur at multiple levels
  - The query that contains the nested query is called its outer query
- Nested queries represent an alternative technique for expressing multi-table manipulation
- Useful for expressing queries where data must be fetched and used in a comparison condition

+ Two Types of Sub-queries

- Non-correlated
  - Results are returned from an inner query to an outer clause
  - Sub-queries are evaluated from the “inside out”
  - The outer query takes an action based on the results of the inner query
- Correlated
  - Conditions in subquery WHERE clause have references to some attributes of a relation in the outer query
  - The outer SQL statement provides the values for the inner subquery to use in its evaluation
  - The subquery is evaluated once for each (combination of) tuple in the outer query

+ Non-Correlated Query Example 1

Find the names of employees who work in a department managed by the employee with SSN = 11.

```
SELECT name
FROM   Employee
WHERE dNum IN
            (SELECT dNumber
             FROM   Department
             WHERE  mgrSSN = 11);
```

```
Employee[ssn,name,dob,address,sex,salary,dNum,mgrSSN]
Department[dName, dNumber, mgrSSN, mgrStartDate]
```

+ Non-Correlated Query Example 2

Find the names of employees who work on projects in Dubai.

```
SELECT name
FROM   Employee
WHERE ssn IN (SELECT  ssn
              FROM    WorksOn
              WHERE   pNo IN
                      (SELECT pNum
                       FROM   Project
                       WHERE  pLocation = ‘Dubai’));
```

```
Employee[ssn,name,dob,address,sex,salary,dNum,mgrSSN]
WorksOn[eSSN, pNo, hours]
Project[pNum, pName, pLocation, dNum]
```

+ Correlated Query Example

Find the names of departments that are responsible for a project located in Dubai.

```
SELECT     dName
FROM       Department
WHERE      ‘Dubai’ IN
                (SELECT        pLocation
                 FROM          Project
                 WHERE         dNum = dNumber);
```

```
Department[dName, dNumber, mgrSSN, mgrStartDate]
Project[pNum, pName, pLocation, dNum]
```

+ Naming in Correlated Queries

- There may be ambiguity, if attributes of outer and nested queries have the same name
- Reference to an unqualified attribute refers to the relation in the inner-most nested query
- If an attribute with the same name has to be referred from an outer query, it must be qualified by the relation (alias) name, e.g. `Department.dNum`

+ Correlated/Non-Correlated Queries

Find the name of departments which have projects in Dubai.

```
SELECT dName
FROM    Department
WHERE ‘Dubai’ IN
    (SELECT pLocation
     FROM    Project
     WHERE dNumber = dNum);
```

```
SELECT dName
FROM    Department
WHERE  dNumber IN
    (SELECT dNum
     FROM    Project
     WHERE  pLocation = ‘Dubai’);
```

For m=5 departments and n=100 projects, there are m x n =500 scans in the correlated query but only m+n=105 scans in non-correlated query

```
Department[dName, dNumber, mgrSSN, mgrStartDate]
Project[pNum, pName, pLocation, dNum]
```

+ Joins vs Sub-queries

- Both joins and sub-queries are used for multi-table queries
  - They can often be used interchangeably
- Use sub-queries when you need to compare aggregates to other values
  - The aggregates can be generated form the nested query
- Use joins when you are displaying results from multiple tables
  - Results cannot be displayed from subqueries

+ Join Example

List department names for the departments that are located in the same place as a project.

```
SELECT     DISTINCT name
FROM       Department, Project
WHERE      dLoc = pLocation;
```

```
SELECT     DISTINCT name
FROM       Department D
WHERE      dLoc IN (SELECT   pLocation
                    FROM     Project P
                    WHERE    P.dNum = D.dNumber);
```

```
Department[name,dNumber,dLoc,mgrSSN,mgrStartDate]
Project[pNum, pName, pLocation, dNum]
```

+ Aggregation Example

List all employees with a salary equal to the lowest salary paid.

```
SELECT    MIN(salary)
FROM      Employee;
// Suppose Result = $15,500
```

```
SELECT    ssn, name, salary
FROM      Employee
WHERE     salary = 15500;
```

```
SELECT    ssn, name, salary
FROM      Employee
WHERE     salary =  (SELECT MIN(salary)
                      FROM Employee);
```

`Employee[ssn,name,dob,address,sex,salary,dNum,mgrSSN]`

+ Sub-query Operators

A subquery S can return a set or a single value.

SET

- <exp> [NOT] IN (S)
  - <exp> is checked for set membership
- <exp> θ {ANY | ALL} (S)
  - <exp> is compared with the set returned
  - ANY: true for one tuple
  - ALL: true for all tuples

SINGLE VALUE

- <exp> θ (S)
  - <exp> is compared with the value returned
  - S must return a single value (otherwise an error will occur)

… θ = {=, <, >, >=, <=, <>}
… Oracle uses SOME and ANY interchangeably

+ 'IN' And '= ANY' Are Equivalent

List department names for departments that have a manager with last name "Smith".

```
SELECT    DIST name
FROM      Department
WHERE     mgrSSN IN (SELECT    ssn
                     FROM      Employee
                     WHERE     name LIKE '%Smith');
```

```
SELECT    DISTINCT name
FROM      Department
WHERE     mgrSSN = ANY
                 (SELECT       ssn
                  FROM         Employee
                  WHERE        name LIKE '%Smith');
```

```
Employee[ssn,name,dob,address,sex,salary,dNum,mgrSSN]
Department[name, dNumber, mgrSSN, mgrStartDate]
```

+ Comparison to ALL Example

List employees whose salary is greater than the salary of all employees in department 5.

```
SELECT    *
FROM      Employee
WHERE     salary > ALL    (SELECT       salary
                           FROM         Employee
                           WHERE        dNum = 5);
```

`Employee[ssn,name,dob,address,sex,salary,dNum,mgrSSN]`

+ Not IN and <>ANY

- IN and = ANY can be equivalent
- However, NOT IN and <> ANY are not!
- Instead, Not IN and <> ALL are equivalent

- Example
  - If a sub-query returns: {$30K, $32K, $37K}
  - NOT IN means: NOT=$30K AND NOT=$32K AND NOT=$37K
  - <> ANY means: NOT=$30K OR NOT=$32K OR NOT=$37K
  - <> ALL means: NOT=$30K AND NOT=$32K AND NOT=$37K

+ NOT IN Example

Find the names of departments that are not responsible for any project located in Perth.

```
SELECT     name FROM Department
WHERE      dNumber NOT IN
      (SELECT dNum FROM Project WHERE pLocation = ‘Perth’);
```

```
SELECT     name FROM Department
WHERE      dNumber <> ALL
      (SELECT dNum FROM Project WHERE pLocation = ‘Perth’);
```

```
SELECT     name FROM Department
WHERE      dNumber <> ANY
      (SELECT dNum FROM Project WHERE pLocation = ‘Perth’);
```

```
Department[name,dNumber,dLoc,mgrSSN,mgrStartDate]
Project[pNum, pName, pLocation, dNum]
```

+ Correlated Subquery Example

Display employee name and department number for employees who earn a salary higher than the average in their department.

```
SELECT    E1.name, E1.dNum
FROM      Employee E1
WHERE     salary > (SELECT     avg(salary)
                    FROM       Employee E2
                    WHERE      E1.dNum = E2.dNum);
```

`Employee[ssn,name,dob,address,sex,salary,dNum,mgrSSN]`

+ Now Let’s Look Back

Find the total number of employees who earn more than 40000, in departments with more than 5 employees.

```
SELECT      dNum, COUNT (*)
FROM        Employee
WHERE       salary > 40000
GROUP BY    dNum
HAVING      COUNT (*) > 5;
```

```
SELECT      dNum, Count(*)
FROM        Employee
WHERE       salary > 40000 AND
            dNum IN  (SELECT dNum FROM Employee
                      GROUP BY    dNum
                      HAVING      count(*) >5)
GROUP BY    dNum;
```

`Employee[ssn,name,dob,address,sex,salary,dNum,mgrSSN]`

+ Using the Exists Function

- The EXISTS function tests for the existence of data that meet the criteria of the sub-query
- WHERE EXISTS (sub-query)
  - Evaluates to true if the result of the correlated sub-query is a non-empty set, i.e. contains 1 or more tuples
  - Evaluates to false if the result of the correlated sub-query returns an empty set, i.e. zero tuples
- Subqueries are used with EXISTS and NOT EXISTS are always correlated

+ EXISTS Examples

Display the names of employees who have dependents.

```
SELECT    DIST name FROM Employee
WHERE     EXISTS (SELECT      *
                 FROM         Dependent
                 WHERE        empSSN = ssn );
```

Display the names of employees who have no dependents.

```
SELECT    DIST name FROM Employee
WHERE     NOT EXISTS (SELECT  *
                      FROM    Dependent
                      WHERE   empSSN = ssn );
```

```
Employee[ssn,name,dob,address,sex,salary,dNum,mgrSSN]
Dependent[empSSN,name,dob,relationship]
```

+ A Sophisticated Example

Display the details of employees who work on ALL projects.

```
SELECT * FROM Employee
WHERE NOT EXISTS
            (SELECT * FROM Project
             WHERE NOT EXISTS
                  (SELECT * FROM WorksOn
                   WHERE eSSN = Employee.ssn AND
                               pNum = Project.pNum));
```

```
Employee[ssn,name,dob,address,sex,salary,dNum,mgrSSN]
WorksOn[eSSN,pNo,hours]
Project[pNum,pName,pLocation,dNum]
```

+ SQL Outline

- DDL
  - Database Definition
- DML
  - Database Modification
  - SELECT Statements
- Database Views
- More on Database Constraints

+ Views

- Provide a way to hide certain data from certain users, or a convinience way for further queries
- Use DDL to create views
  - CREATE VIEW <view-name> AS <query expression>;
  - <query expression> is any legal SQL query

```
create view BranchLoan as
select branchName, loanNumber
from Loan;
```

```
select  loanNumber
from    BranchLoan
where branchName='Perryridge';
```

- Use DML to manipulate views
  - They are virtual tables, and can be used as tables

```
Loan[branchName,loanNumber,amount]
```

+ View Update

- Assume that we allow users who have access to the BranchLoan view to insert records in the view
- Example: to add a new tuple to the BranchLoan view.

```
insert into BranchLoan values ('Perryridge', 'L-307');
```

This insertion must be represented by the insertion of the tuple ('Perryridge', 'L-307', *null*) into the Loan relation

```
Loan[branch_name,loan_number,amount]
Branch_loan[branch_name,loan_number,amount]
```

+ View Update Problems

- Suppose we want to insert the tuple ('Choi Hung', 'Lei Chen') into the `BranchBorrower` view

```
create view BranchBorrower as
select branchName, customerName
from Loan natural join Borrower;
```

- The `Loan` and `Borrower` relations have to be updated accordingly, violating integrity constraints!

```
insert into Loan (loanNumber, amount, branchName)
      values (null, null, 'Choi Hung');
insert into Borrower (customerName, loanNumber)
      values ('Lei Chen', null);
```

```
Loan[branchName,loanNumber,amount]
Borrower[customerName, loanNumber]
```

+ Rules For Updatable Views

- A view is updateable (i.e., tuples can be inserted, updated and deleted) if all the following conditions are satisfied by the query that defines the view
  - The from clause has only one relation, AND
  - The select clause contains only attribute names of the relation, and does not have any expressions, aggregates, or distinct specification, AND
  - Any attribute not listed in the select clause can be set to null (i.e., it does not have a not null constraint and is not part of a primary key), AND
  - The query does not have a group by or having clause
- The rules for view update are often system dependent

+ SQL Outline

- DDL
  - Database Definition
- DML
  - Database Modification
  - SELECT Statements
- Database Views
- More on Database Constraints

+ More on Integrity Constraints (IC)

- Domain constraints define valid values for attributes
- DBMS tests values inserted/updated to the database and check queries to ensure that the comparisons make sense
- The constraints that can be included:
  - not null: specifies that null values are not allowed
  - primary key: specifies a key for a relation (cannot be null)
  - unique: specifies that a set of attributes is a candidate key (can be null)
  - foreign key: specifies that one or more attributes refer to a primary key attribute in another relation
  - check: specifies a predicate that the values in every tuple must satisfy

+ Primary Key vs Unique Constraints

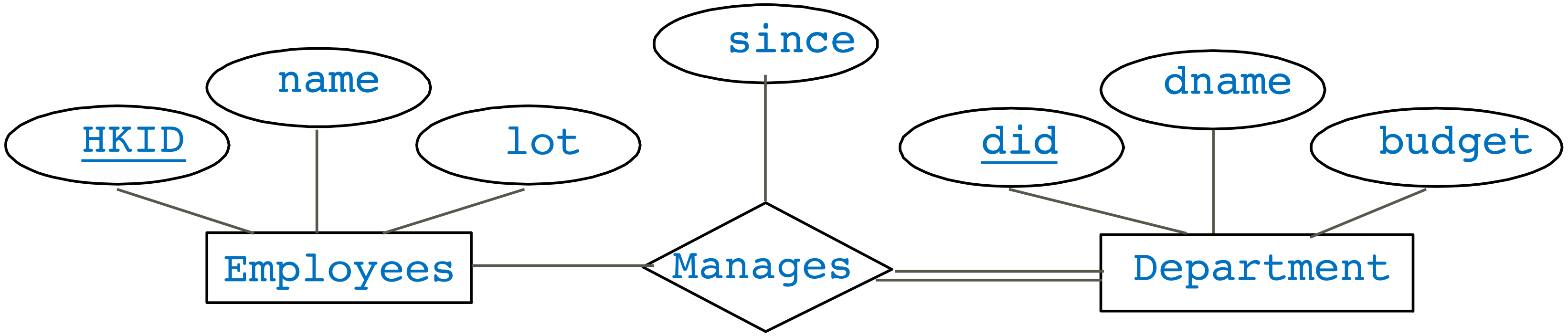
- A relation can possibly have many candidate keys but only one of them is chosen as the primary key

```
create table student (
   student_id  char (8),
   name         char (20),
   login        char (10),
   age          int,
   cga          real,
   primary key (student_id)
   unique (login));
```

```
create table enrolled (
   student_id   char (8),
   course_id    char (10),
   grade        char (2),
   primary key (student_id),
   unique (course_id, grade));
```

… used carelessly, an IC can prevent the storage of database instances that should be allowed!

+ Not Null & Participation Constraints



We can capture total participation constraints using NOT NULL

```
create table department (
   did  int,
   dname  char(20),
   budget  real,
   HKID  char(11) not null,
   primary key (did),
   foreign key (HKID) references employees on delete cascade);
```

+ CHECK CLAUSE: DOMAINS

- The check clause can be used in a create domain clause to add an integrity constraint to the domain.
  - Example: ensure that an hourly-wage domain allows only values greater than a specified value ($25 in this example)

```
create domain hourly_wage numeric(5, 2)
    constraint wage_test check (value >= 25.00);
```

  - The new 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 named wage_test

☞ The constraint name is optional, but useful to indicate which constraint an update violated

+ Check Clause: Attributes

- The check clause can be used to add an integrity constraint for an attribute and can contain an arbitrary predicate
  - It is specified in the definition of a relation and checked whenever there is an update to the relation
  - Example: to ensure that semester can have only specified values and that the course id is between 1000 and 4999).

```
create table section (
    course_id     char(8),
    section_id    char(8),
    semester      varchar(6),
    year          numeric(4,0),
    check (course_id between 1000 and 4999),
    check (semester in ('Fall', 'Winter', 'Spring', 'Summer')));
```

+ Assertions

- An assertion is an arbitrary SQL predicate that the database must always satisfy
- An assertion takes the form:

```
create assertion <assertion-name> check <predicate>;
```

- Difference from previous constraints:
  - A constraint is associated with a single table and checked when there is an update on this specific table
  - An assertion may be associated with several tables and is checked every time when there is an update anywhere
- Assertion testing may introduce a significant amount of overhead and should be used with great care

+ Assertion 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 exist (
    select *
    from branch
    where (select sum(amount)
                from loan
                where loan.branch_name=branch.name)
                >=
                (select sum(amount)
                 from account
                 where account.branch_name=branch.name)));
```

- Since the assertion refers to multiple tables, it cannot be included as a constraint in the definition of `loan` or account

+ Triggers

- A trigger is a statement that is executed automatically by the system as a side effect of a modification to the database

- A trigger must
  - Specify the condition(s) under which it is to be executed
  - Specify the action(s) to be taken when it executes

- A trigger is used to implement integrity constraints that cannot be specified by SQL constraints
  - Example: automatically update the assets value of the branch relation whenever the balance value of the account relation is updated

… the ECA model: event-condition-action

+ 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
  - Hiving this loan a loan number which is 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

+ Trigger Example

```
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.(T is the working version of account and S is the original version of account)

+ SQL Summary

- Structured Query Language (SQL) is a relational data manipulation language that provides facilities to
  - Query relations
    - Select-From-Where Statement
    - Set Operations (Union, Intersect, Except)
    - Nested Subqueries (to test for set membership, comparison, cardinality)
    - Aggregate Functions (avg, min, max, sum, count)
    - Group By with Having clause
  - Create and modify relations
    - Create, Alter, Drop Tables
    - Specify integrity constraints: domain, key, foreign key, general
    - Specify views
    - Insert, Delete, Update Tuples

+ Readings

- Textbook: Ch 3, 4, 5 (ed 6/ed 7)

Further References:

- W3School SQL tutorial
  - https://www.w3schools.com/sql/
- Oracle SQL
  - https://docs.oracle.com/cd/B19306_01/server.102/b14200/toc.htm
- Oracle University
  - https://education.oracle.com/