

# TDDD37

## Database technology

### SQL

Fang Wei-Kleiner

[fang.wei-kleiner@liu.se](mailto:fang.wei-kleiner@liu.se)

<http://www.ida.liu.se/~TDDD37>



# Announcement

- Course registration: system problems from registration office. Be patient.
- Registration for the lab: possible without being registered to the course → do that now.
- Encourage building a lab group with two.
- Temporary solution for the lab homework without an DB account: install mySQL and download the scripts from the lab website.

# SQL

- SQL: Structured Query Language
  - Pronounced “S-Q-L” or “sequel”
  - The standard query language supported by most commercial DBMS
- A brief history
  - IBM System R
  - ANSI SQL89
  - ANSI SQL92 (SQL2)
  - ANSI SQL99 (SQL3)
  - ANSI SQL 2003 (added OLAP, XML, etc.)
  - ANSI SQL 2006 (added more XML)
  - ANSI SQL 2008, ...

# Create and drop table

```
CREATE TABLE table_name  
    (... , column_namei column_typei , ... );
```

```
DROP TABLE table_name;
```

- Examples

```
CREATE TABLE WORKS_ON (  
    ESSN    integer,  
    PNO     integer,  
    HOURS   decimal(3,1));
```

```
DROP table Student;
```

- -- SQL is insensitive to white space.
- -- SQL is insensitive to case (e.g., ...Hours... is equivalent to HOURS...)

# Basic SFW query

```
SELECT <attribute-list>  
FROM <table-list>  
WHERE <condition>;
```

*attribute-list*:  $R1.A1, \dots, Rk.Ar$

★ Attributes whose values to be required

*table-list*:  $R1, \dots, Rk$

★ Relations to be queried

*condition*: conditional (boolean) expression

★ identifies the tuples that should be retrieved

- comparison operators(=, <>, >, >=, ...)
- logical operators (*and*, *or*, *not*)

# Reading a table

- List all information about the employees of department 5

```
SELECT *  
FROM EMPLOYEE  
WHERE DNO = 5;
```

- \* is short hand for all columns.
- WHERE is optional.

# Selection and projection

- List last name, birth date and address for all employees whose name is 'Alicia J. Zelaya'

```
SELECT LNAME, BDATE, ADDRESS
```

```
FROM EMPLOYEE
```

```
WHERE FNAME = 'Alicia' AND MINIT = 'J' AND LNAME = 'Zeleya';
```

- String literals (case sensitive) are enclosed in single quote

# Pattern matching

- List last name, birth date and address for all employees whose last name contain 'aya'

```
SELECT LNAME, BDATE, ADDRESS  
FROM EMPLOYEE  
WHERE LNAME LIKE '%aya%';
```

- LIKE matches a string against a pattern
  - % matches any sequence of 0 or more characters



# Join -- equijoin

- List all employees and names of their department

SELECT LNAME, DNAME

FROM EMPLOYEE, DEPARTMENT

WHERE DNO = DNUMBER;

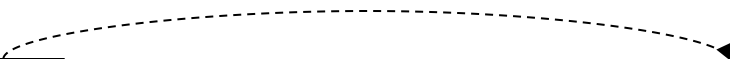
<i>EMPLOYEE</i>	<i>LNAME</i>	<i><u>DNO</u></i>	<i>DEPARTMENT</i>	<i>DNAME</i>	<i><u>DNUM</u></i>
	Smith	5		Research	5
	Wong	5		Administration	4
	Zelaya	4		headquarters	1
	Wallace	4			
	Narayan	5			
	English	5			
	Jabbar	4			
	Borg	1			

# Ambiguous names -- Aliasing

- Same attribute name used in different relations

```
SELECT NAME, NAME  
FROM EMPLOYEE, DEPARTMENT  
WHERE DNO=DNUMBER;
```

<i>EMPLOYEE</i>	<i>NAME</i>	<i>DNO</i>	<i>DEPARTMENT</i>	<i>NAME</i>	<i>DNUM</i>
	Smith	5		Research	5
	Wong	5		Administration	4
	Zelaya	4		headquarters	1
	Wallace	4			
	Narayan	5			
	English	5			
	Jabbar	4			
	Borg	1			



# Ambiguous names -- Aliasing

- No alias (wrong) 

```
SELECT NAME, NAME  
FROM EMPLOYEE, DEPARTMENT  
WHERE DNO=DNUMBER;
```
- Whole name 

```
SELECT EMPLOYEE.NAME, DEPARTMENT.NAME  
FROM EMPLOYEE, DEPARTMENT  
WHERE EMPLOYEE.DNO= DEPARTMENT.DNUMBER;
```
- Alias 

```
SELECT E.NAME, D.NAME  
FROM EMPLOYEE E, DEPARTMENT D  
WHERE E.DNO=D.DNUMBER;
```

# Self join

- List last name for all employees together with last names of their bosses

```
SELECT E.LNAME "Employee", S. LNAME "Boss"  
FROM EMPLOYEE E, EMPLOYEE S  
WHERE E.SUPERSSN = S.SSN;
```

<i>Employee Boss</i>	
Smith	Wong
Wong	Borg
Zelaya	Wallace
Wallace	Borg
Narayan	Wong
English	Wong
Jabbar	Wallace

# Bag vs. set

- List all salaries

```
SELECT SALARY
```

```
FROM EMPLOYEE;
```

## *SALARY*

30000

40000

25000

43000

38000

25000

25000

55000

- SQL considers a table as a multi-set (bag), i.e. tuples can occur more than once in a table
- Why?
  - Removing duplicates is expensive
  - User may want information about duplicates (real distribution)
  - Aggregation operators

# Distinct

- List all salaries

```
SELECT SALARY  
FROM EMPLOYEE;
```

## ***SALARY***

30000  
40000  
25000  
43000  
38000  
25000  
25000  
55000

- List all salaries without duplicates

```
SELECT DISTINCT SALARY  
FROM EMPLOYEE;
```

## ***SALARY***

30000  
40000  
25000  
43000  
38000  
55000

# Set and bag operations

- Queries can be combined by set operations: UNION, INTERSECT, EXCEPT (MySQL only supports UNION)
- Retrieve all first names of all people in our mini world

(Set semantic)

```
SELECT FNAME FROM EMPLOYEE
```

UNION

```
SELECT DEPENDENT_NAME FROM DEPENDENT;
```

(Bag semantic)

```
SELECT FNAME FROM EMPLOYEE
```

UNION ALL

```
SELECT DEPENDENT_NAME FROM DEPENDENT;
```

# Subqueries

- List employees do not have project assignment more than 10 hours.

```
SELECT LNAME  
FROM EMPLOYEE, WORKS_ON  
WHERE SSN = ESSN AND HOURS <= 10.0;
```

Why is the query wrong?

- Employees who do not work in any project:
  - They should be in the answer set, but is not from the above query → their SSN does not occur in WORKS\_ON



# Subqueries

- List employees do not have project assignment more than 10 hours.


```
SELECT LNAME  
FROM EMPLOYEE  
WHERE SSN NOT IN (SELECT ESSN FROM WORKS_ON  
                  WHERE HOURS > 10.0);
```

- $x$  IN ( subquery) checks if  $x$  is in the result of subquery

# Subqueries

- List employees do not have project assignment more than 10 hours. (solution 2 using NOT EXISTS)

```
SELECT LNAME  
FROM EMPLOYEE  
WHERE NOT EXISTS (SELECT * FROM WORKS_ON  
                   WHERE SSN = ESSN AND HOURS > 10.0);
```



- EXISTS (subquery ) checks if the result of subquery is non-empty
- This is a **correlated** subquery -- a subquery that references tuple variables in surrounding queries

# Operational semantics of subquery

- List employees do not have project assignment more than 10 hours.

```
SELECT LNAME  
FROM EMPLOYEE  
WHERE NOT EXISTS (SELECT * FROM WORKS_ON  
                   WHERE SSN = ESSN AND HOURS > 10.0);
```

- For each row E in EMPLOYEE
  - Evaluate the subquery with the appropriate value of E.SSN
  - If the result of the subquery is not empty, output E.LNAME
- The DBMS query optimizer may choose to process the query in an equivalent, but more efficient way

# Aggregates

- Standard SQL aggregate functions: COUNT , SUM , AVG , MIN , MAX
- List the number of employees and their average salary

```
SELECT COUNT(*), AVG(SALARY)  
FROM EMPLOYEE;
```

- COUNT(\*) counts the number of rows

# Grouping

- Used to apply an aggregate function to subgroups of tuples in a relation

GROUP BY – grouping attributes

- List for each department the department number, the number of employees and the average salary.

```
SELECT DNO, COUNT(*), AVG(SALARY)
```

```
FROM EMPLOYEE
```

```
GROUP BY DNO;
```

- List for each department the department number, the number of employees and the average salary.

```
SELECT DNO, COUNT(*), AVG(SALARY)
FROM EMPLOYEE
GROUP BY DNO;
```

<i>DNO</i>	<i>NAME</i>	<i>SALARY</i>
5	Smith	65210
4	Lee	21000
5	Brin	43250
4	Page	12220
5	Jobs	56750
5	Gates	24670
4	Wills	33250
1	Yang	55000

<i>DNO</i>	<i>COUNT(*)</i>	<i>AVG(SALARY)</i>
5	4	47470
4	3	22156
1	1	55000

# Operational semantics of GROUP BY

SELECT ... FROM ... WHERE ... GROUP BY ... ;

- Compute FROM (join)
- Compute WHERE (selection)
- Compute GROUP BY: group rows according to the values of GROUP BY columns
- Compute SELECT for each group
- For aggregation functions with DISTINCT inputs, first eliminate duplicates within the group

➔ Number of groups = number of rows in the final output



# Example of computing GROUP BY

SELECT DNO, COUNT(\*), AVG(SALARY) FROM EMPLOYEE **GROUP BY** DNO;

<i>DNO</i>	<i>NAME</i>	<i>SALARY</i>
5	Smith	65210
4	Lee	21000
5	Brin	43250
4	Page	12220
5	Jobs	56750
5	Gates	24670
4	Wills	33250
1	Yang	55000

Group rows according to the values of GROUP BY columns

<i>DNO</i>	<i>NAME</i>	<i>SALARY</i>
5	Smith	65210
5	Brin	43250
5	Jobs	56750
5	Gates	24670
4	Page	12220
4	Lee	21000
4	Wills	33250
1	Yang	55000

<i>DNO</i>	<i>COUNT(*)</i>	<i>AVG(SALARY)</i>
5	4	47470
4	3	22156
1	1	55000

← Compute SELECT for each group



# Restriction on SELECT

- If a query uses aggregation/group by, then every column referenced in SELECT must be either
  - Aggregated, or
  - A GROUP BY column
- This restriction ensures that any SELECT expression produces only one value for each group

SELECT ~~NAME~~, COUNT(\*), AVG(SALARY) FROM EMPLOYEE GROUP BY DNO;

- Recall there is one output row per group
  - There can be multiple NAME values per group

# HAVING

- Used to filter groups based on the group properties (e.g., aggregate values, GROUP BY column values)

```
SELECT DNO, COUNT(*), AVG(SALARY)
FROM EMPLOYEE
GROUP BY DNO
HAVING COUNT(*) > 2;
```

<i>DNO COUNT(*) AVG(SALARY)</i>		
5	4	47470
4	3	22156
<del>1</del>	<del>1</del>	<del>55000</del>

# Order of query results

- Select department names and their locations in alphabetical order.

```
SELECT DNAME, DLOCATION  
FROM DEPARTMENT D, DEPT_LOCATIONS DL  
WHERE D.DNUMBER = DL.DNUMBER  
ORDER BY DNAME ASC, DLOCATION DESC;
```

<i>DNAME</i>	<i>DLOCATION</i>
Administration	Stafford
Headquarters	Houston
Research	Sugarland
Research	Houston
Research	Bellaire

# NULL value

- SQL solution for unknown or non-applicable values
  - A special value NULL
  - For every domain
  - Special rules for dealing with NULL's
- Example: EMPLOYEE(LNAME, SSN, SALARY, SUPERSSN)
  - <Borg, 8888888, 55000, NULL>
- When we operate on a NULL and another value (including another NULL ) using +, -, etc., the result is NULL
- Aggregate functions ignore NULL , except COUNT(\*)
  - ➔(since it counts rows)

# Three-valued logic

- TRUE = 1, FALSE = 0, UNKNOWN = 0.5
- $x \text{ AND } y = \min(x, y)$
- $x \text{ OR } y = \max(x, y)$
- NOT  $x = 1 - x$
- When we compare a NULL with another value (including another NULL) using =, >, etc., the result is UNKNOWN
- WHERE and HAVING clauses only select rows for output if the condition evaluates to TRUE
  - UNKNOWN is not enough

30000  
40000  
43000  
NULL

# NULL values

- SELECT AVG(SALARY) FROM EMPLOYEE;
- SELECT SUM(SALARY)/COUNT(\*) FROM EMPLOYEE;
  - Not equivalent
  - Although  $AVG(SALARY) = SUM(SALARY)/COUNT(SALARY)$  still
- SELECT \* FROM EMPLOYEE;
- SELECT \* FROM EMPLOYEE WHERE SALARY=SALARY;
  - Not equivalent
- List all employees that do not have a boss:  
SELECT LNAME FROM EMPLOYEE WHERE SUPERSSN **IS NULL**;

E

<i>LNAME</i>	<i>SSN</i>	<i>SUPERSSN</i>
Smith	333445555	123456789
Borg	123456789	NULL
Wong	888665555	123456789

S

<i>LNAME</i>	<i>SSN</i>	<i>SUPERSSN</i>
Smith	333445555	123456789
Borg	123456789	NULL
Wong	888665555	123456789

Outer  
join

- List the last name of all employees together with the names of their bosses.
  - Some employees do not have any boss
  - We want to list the bossless employees too – where boss field is noted as NULL

SELECT E.LNAME “Employee”, S.LNAME “Boss”  
 FROM EMPLOYEE E, EMPLOYEE S  
 WHERE E.SUPERSSN = S.SSN

<i>Employee</i>	<i>Boss</i>
Smith	Borg
Wong	Borg

- Returns only ‘Smith’ and ‘Wong’
- Tuple of ‘Borg’ does not have a join partner



E	LNAME	SSN	SUPERSSN
	Smith	333445555	123456789
	Borg	123456789	NULL
	Wong	888665555	123456789

S	LNAME	SSN	SUPERSSN
	Smith	333445555	123456789
	Borg	123456789	NULL
	Wong	888665555	123456789

Dangling row

SELECT E.LNAME "Employee", S.LNAME "Boss"  
 FROM EMPLOYEE E LEFT JOIN EMPLOYEE S  
 ON E.SUPERSSN = S.SSN

Employee	Boss
Smith	Borg
Wong	Borg
Borg	NULL

- A **left outer join** (LEFT JOIN) of R with S includes rows in R join S plus **dangling** R rows padded with NULL
  - Dangling R rows are those that do not join with any S rows
- A **right outer join** (RIGHT JOIN) of R with S includes rows in R join S plus **dangling** S rows padded with NULL
  - Dangling S rows are those that do not join with any R rows



# Add tuples into table

```
INSERT INTO <table> (<attr>,...) VALUES ( <val>, ...);
```

```
INSERT INTO <table> (<attr>, ...) <subquery> ;
```

- Store information about how many hours an employee works for the project '1' into WORKS\_ON

```
INSERT INTO WORKS_ON VALUES (123456789, 1, 32.5);
```

# Update data

```
UPDATE <table> SET <attr> = <val> ,...  
WHERE <condition> ;
```

```
UPDATE <table> SET (<attr>, ....) = ( <subquery> )  
WHERE <condition> ;
```

- Give all employees in the 'Research' department a 10% raise in salary.

```
UPDATE EMPLOYEE  
SET SALARY = SALARY*1.1  
WHERE DNO IN (SELECT DNUMBER  
FROM DEPARTMENT  
WHERE DNAME = 'Research');
```

# Delete data

- DELETE FROM <table> WHERE <condition> ;
- Delete employees having the last name 'Borg' from the EMPLOYEE table

```
DELETE FROM EMPLOYEE  
WHERE LNAME = 'Borg';
```

# Constraints

- Restrictions on allowable data in a database
  - In addition to the simple structure and type restrictions imposed by the table definitions
  - Declared as part of the schema
  - Enforced by the DBMS
- Why use constraints?
  - Protect data integrity (catch errors)
  - Tell the DBMS about the data (so it can optimize better)

# Type of SQL constraints

- NOT NULL
- Key
- Referential integrity (foreign key)
- General assertion
- Tuple- and attribute-based CHECK's

# NOT NULL example

```
CREATE TABLE EMPLOYEE  
(SSN INTEGER NOT NULL,  
LNAME VARCHAR(30) NOT NULL,  
ADDRESS VARCHAR(30),  
SALARY INTEGER,  
SUPERSSN INTEGER);
```

# Key declaration

- At most one PRIMARY KEY per table
  - Typically implies a primary index
  - Rows are stored inside the index, typically sorted by the primary key value → best speedup for queries
- Any number of UNIQUE keys per table
  - Typically implies a secondary index
  - Pointers to rows are stored inside the index → less speedup for queries

# Key example

```
CREATE TABLE EMPLOYEE  
(SSN INTEGER NOT NULL PRIMARY KEY,  
LNAME VARCHAR(30) NOT NULL,  
EMAIL VARCHAR(30) UNIQUE,  
SALARY INTEGER,  
SUPERSSN INTEGER);
```



# Referential integrity example

- WORKS\_ON.ESSN references EMPLOYEE.SSN
    - If an ESSN appears in WORKS\_ON, it must appear in EMPLOYEE
  - WORKS\_ON.PNO references PROJECT.PNUMBER
    - If a PNO appears in WORKS\_ON, it must appear in PROJECT
- ➔ That is, no “dangling pointers”
- Referenced column(s) must be PRIMARY KEY
  - Referencing column(s) form a FOREIGN KEY

# COMPANY schema

- **EMPLOYEE** (*FNAME, MINIT, LNAME, SSN, BDATE, ADDRESS, SEX, SALARY, SUPERSSN, DNO*)
  - **DEPT\_LOCATIONS** (*DNUMBER, DLOCATION*)
  - **DEPARTMENT** (*DNAME, DNUMBER, MGRSSN, MGRSTARTDATE*)
  - **WORKS\_ON** (*ESSN, PNO, HOURS*)
  - **PROJECT** (*PNAME, PNUMBER, PLOCATION, DNUM*)
  - **DEPENDENT** (*ESSN, DEPENDENT-NAME, SEX, BDATE, RELATIONSHIP*)
-

# Create tables

```
CREATE TABLE WORKS_ON (
```

```
    ESSN    integer  
           constraint fk_works_emp  
           references EMPLOYEE(SSN),
```

```
    PNO     integer  
           constraint fk_works_proj  
           references PROJECT(PNUMBER),
```

```
    HOURS   decimal(3,1),
```

```
    constraint pk_workson  
    primary key (ESSN, PNO)  
);
```

# Enforcing referential integrity

Delete employees having the last name 'Borg' from the EMPLOYEE table

```
DELETE FROM EMPLOYEE  
WHERE LNAME = 'Borg';
```

referential integrity constraints

Foreign key

EMPLOYEE	FNAME	M	LNAME	<u>SSN</u>
	Ramesh	K	Narayan	666884444
	Joyce	A	English	453453453
	Ahmad	V	Jabbar	987987987
	James	E	Borg	888665555

DEPARTMENT	DNAME	<u>DNUMBER</u>	<u>MGRSSN</u>
	Research	5	333445555
	Administration	4	987654321
	Headquarters	1	888665555

SET NULL ? SET DEFAULT ? CASCADE ?

# Views

- A virtual table derived from other – possible virtual -- tables.

```
CREATE VIEW dept_view  
AS SELECT DNO, COUNT(*), AVG(SALARY)  
FROM EMPLOYEE  
GROUP BY DNO
```

- Why?
  - Simplify query commands
  - Provide data security
  - Enhance programming productivity
- Update problems