SQL — Part 2

5DV119 — Introduction to Database Management

Umeå University
Department of Computing Science

Stephen J. Hegner
hegner@cs.umu.se
http://www.cs.umu.se/~hegner

Embedded Subqueries

- It is often useful, if not essential, to be able to use subqueries in the `WHERE` clause of a query.

Query: Find all employees who work in the same department as Alicia Zeyala (999887777).

- When a subquery returns a set consisting of just one tuple, the result may be regarded as a tuple.

```
SELECT LName, FName, MInit, SSN
FROM Employee
WHERE DNo = (SELECT DNo
             FROM   Employee
             WHERE  SSN='999887777');
```

- Note the lexical scoping of variables!
- An alternative using `IN` (set membership $\in$):

```
SELECT LName, FName, MInit, SSN
FROM Employee
WHERE DNo IN (SELECT DNo
              FROM   Employee
              WHERE  SSN='999887777');
```

Queries with Existential Quantification

Query: Find all employees who work on some project which Alicia Zeyala (999887777) also works on.

```
SELECT LName, FName, MInit, SSN, PNo
FROM Employee JOIN Works_On ON (SSN=ESSN)
WHERE PNo IN (SELECT PNo
              FROM   Works_On
              WHERE  ESSN='999887777');
```

- Exclude Alicia from the list.

```
SELECT LName, FName, MInit, SSN, PNo
FROM Employee JOIN Works_On ON (SSN=ESSN)
WHERE PNo IN (SELECT PNo
              FROM   Works_On
              WHERE  ESSN='999887777')
     AND SSN<>'999887777';
```

Realizing INTERSECT Using Subqueries

- As already noted, MySQL does not support the INTERSECT operation.

Query: Find those employees who work on both the ProductX and ProductY projects.

```
SELECT LName, FName, MInit, SSN
FROM Employee JOIN Works_On ON (SSN=ESSN) JOIN Project ON (PNo=PNumber)
WHERE PName='ProductX'
INTERSECT
SELECT LName, FName, MInit, SSN
FROM Employee JOIN Works_On ON (SSN=ESSN) JOIN Project ON (PNo=PNumber)
WHERE (PName='ProductY');
```

- Fortunately, this operation can be realized using subqueries.

```
SELECT LName, FName, MInit, SSN
FROM Employee JOIN Works_On ON (SSN=ESSN) JOIN Project ON (PNo=PNumber)
WHERE (PName='ProductX') AND
      (SSN IN
       (SELECT SSN
        FROM Employee JOIN Works_On ON (SSN=ESSN)
                      JOIN Project ON (PNo=PNumber)
        WHERE (PName='ProductY')));
```

- Only the key SSN need be used in the subquery.

Realizing EXCEPT Using Subqueries

- As already noted, MySQL does not support the EXCEPT operation.

Query: Find those employees who work on the ProductY project but not the ProductX project.

```
SELECT LName, FName, MInit, SSN
FROM Employee JOIN Works_On ON (SSN=ESSN) JOIN Project ON (PNo=PNumber)
WHERE PName='ProductY'
EXCEPT
SELECT LName, FName, MInit, SSN
FROM Employee JOIN Works_On ON (SSN=ESSN) JOIN Project ON (PNo=PNumber)
WHERE (PName='ProductX');
```

- Fortunately, this operation can be realized using subqueries.

```
SELECT LName, FName, MInit, SSN
FROM Employee JOIN Works_On ON (SSN=ESSN) JOIN Project ON (PNo=PNumber)
WHERE (PName='ProductY') AND
      (NOT (SSN IN
            (SELECT SSN
             FROM Employee JOIN Works_On ON (SSN=ESSN)
                           JOIN Project ON (PNo=PNumber)
             WHERE (PName='ProductX'))));
```

- Only the key SSN need be used in the subquery.

Queries with Universal Quantification

Query: Find all employees who work on every project which Alicia Zeyala (999887777) also works on. Exclude Alicia herself.

- At first sight, this appears to be impossible with SQL.
- However, it may be rephrased as a double negation:
  - Find all employees E for which there is no project P which Alicia works on but E does not work on.
- This operation is formally known as *division* and will be studied more carefully in connection with the relational algebra.

```
SELECT DISTINCT LName, FName, MInit, SSN
FROM Employee JOIN Works_On ON (SSN=ESSN)
WHERE NOT EXISTS (SELECT PNo
                  FROM   Works_On
                  WHERE  (ESSN='999887777')
                  EXCEPT (SELECT PNo
                          FROM   Works_On
                          WHERE  (SSN=ESSN)))
                AND (SSN<>'999887777');
```

An Alternative Construction for Division

- As already noted, MySQL does not support the `EXCEPT` operation.
- Division may also be realized using `NOT..IN` for negation.

Query: Find all employees who work on every project which Alicia Zeyala (999887777) also works on. Exclude Alicia herself.

```
SELECT DISTINCT LName, FName, MInit, SSN
FROM Employee JOIN Works_On ON (SSN=ESSN)
WHERE NOT EXISTS (SELECT PNo
                  FROM   Works_On AS W1
                  WHERE  (W1.ESSN='999887777')
                         AND NOT PNo IN
                          (SELECT PNo
                           FROM   Works_On
                           WHERE  (ESSN=SSN)))
                AND SSN<>'999887777';
```

Queries which Count (without Aggregation)

Query: Find all employees who work on at least two distinct projects.

```
SELECT DISTINCT LName, FName, MInit, SSN
FROM   Employee JOIN Works_On AS W1 ON (SSN=W1.ESSN)
                JOIN Works_on AS W2 ON (SSN=W2.ESSN)
WHERE  (W1.PNo<>W2.PNo);
```

Query: Find all employees who work on at exactly one project.

```
SELECT DISTINCT LName, FName, MInit, SSN
FROM   Employee
WHERE  EXISTS (SELECT *
               FROM   Works_On
               WHERE  (SSN=ESSN))
       AND
       NOT EXISTS
        (SELECT *
         FROM   Works_On AS W1 JOIN Works_on AS W2
                                 ON (W1.ESSN=W2.ESSN)
         WHERE  (W1.PNo<>W2.PNo) AND (SSN=W1.ESSN));
```

Exercise: Find all employees who work on at exactly two projects.

ALL and ANY

Query: Find the employee(s) with the highest salary.

```
SELECT DISTINCT LName, FName, MInit, SSN, Salary
FROM   Employee
WHERE  Salary >= ALL (SELECT Salary
                      FROM   Employee);
```

Query: Find the employee(s) with salaries which are not the lowest.

```
SELECT DISTINCT LName, FName, MInit, SSN, Salary
FROM   Employee
WHERE  Salary > ANY (SELECT Salary
                     FROM   Employee);
```

Aggregation Operators

Query: For each project find the minimum, maximum, average, and total hours worked by all employees, as well as the number of such employees.

```
SELECT   PName, MIN(Hours), MAX(Hours),
                AVG(Hours), SUM(Hours), COUNT(*)
FROM     Project JOIN Works_On ON (PNumber=PNo)
GROUP BY PName
UNION
SELECT   PName, 0, 0, 0, 0, 0
FROM     Project as P1
WHERE    (NOT EXISTS
          (SELECT *
           FROM   Project JOIN Works_On ON (PNumber=PNo)
           WHERE  P1.PNumber=PNo));
```

An important rule: Every attribute which occurs in the SELECT clause and which is not aggregated must occur in the GROUP BY clause as well.

- Will not work (even though it clearly should):

```
SELECT   PName, PNo, MIN(Hours), MAX(Hours),
                     AVG(Hours), SUM(Hours), COUNT(*)
FROM     Project JOIN Works_On ON (PNumber=PNo)
GROUP BY PName
UNION
...
```

Omission of GROUP BY

- If the GROUP BY clause is omitted, the aggregation is over the entire table.

```
SELECT   MIN(Hours), MAX(Hours),
         AVG(Hours), SUM(Hours), COUNT(*)
FROM     Project JOIN Works_On ON (PNumber=PNo);
```

- In this case, there must be no non-aggregated attributes in the SELECT clause.

- Does not work:

```
SELECT   PNo, MIN(Hours), MAX(Hours),
         AVG(Hours), SUM(Hours), COUNT(*)
FROM     Project JOIN Works_On ON (PNumber=PNo);
```

- In the above case, PNo must appear in the GROUP BY clause.

The HAVING Clause

Query: For each project with at least three employees working on it, find the average and total hours worked on it.

- The following does <u>not</u> work:

```
SELECT   PName, AVG(Hours), SUM(Hours)
FROM     Project JOIN Works_On ON (PNumber=PNo)
WHERE    (Count(*) >= 3)
GROUP BY PName;
```

- The problem is that the WHERE clause is evaluated <u>before</u> the aggregation.

- The solution is to use a HAVING clause.

```
SELECT   PName, AVG(Hours), SUM(Hours)
FROM     Project JOIN Works_On ON (PNumber=PNo)
GROUP BY PName
HAVING   (Count(*) >= 3);
```

- The HAVING clause must come <u>after</u> the GROUP BY clause.

Formats on Output Columns

- In the following, PostgreSQL will express `AVG(Hours)` with 16 places to the right of the decimal point (there are "only" five for MySQL):

```
SELECT   PName, MIN(Hours), MAX(Hours),
                AVG(Hours), SUM(Hours), COUNT(*)
FROM     Project JOIN Works_On ON (PNumber=PNo)
GROUP BY PName;
```

- To remedy this, casting may be used:

```
SELECT   PName, MIN(Hours), MAX(Hours),
                CAST(AVG(Hours) AS DECIMAL(8,2)),
                SUM(Hours), COUNT(*)
FROM     Project JOIN Works_On ON (PNumber=PNo)
GROUP BY PName;
```

- MySQL will use the entire expression `CAST(AVG(Hours) AS DECIMAL(8,2))` as the column header, but this can be fixed easily:

```
SELECT   PName, MIN(Hours), MAX(Hours),
                CAST(AVG(Hours) AS DECIMAL(8,2)) AS AVG_Hours,
                SUM(Hours), COUNT(*)
FROM     Project JOIN Works_On ON (PNumber=PNo)
GROUP BY PName;
```

Formats on Output Columns — 2

- To obtain a consistent representation, even constants may be cast:

```
SELECT    PName, MIN(Hours), MAX(Hours),
                 CAST(AVG(Hours) AS DECIMAL(8,2)) AS AVG_Hours,
                 SUM(Hours), COUNT(*)
FROM      Project JOIN Works_On ON (PNumber=PNo)
GROUP BY PName
UNION
SELECT    PName, 0, 0, CAST(0 AS DECIMAL(8,2)), 0, 0
FROM      Project as P1
WHERE     (NOT EXISTS
           (SELECT *
            FROM   Project JOIN Works_On ON (PNumber=PNo)
            WHERE  P1.PNumber=PNo));
```

- Of course, for really “pretty” and consistent output, all of the aggregated columns should be cast in this query.

Embedded Queries in the HAVING Clause

Query: Find the project(s) with the greatest number of hours.

```
SELECT   P1.PName, SUM(W1.Hours)
FROM     Project AS P1 JOIN Works_On AS W1 ON (P1.PNumber=W1.PNo)
GROUP BY P1.PName
HAVING   NOT EXISTS (SELECT   W2.PNo, SUM(W2.Hours)
                     FROM     Works_On AS W2
                     GROUP BY W2.PNo
                     HAVING   (SUM(W2.Hours) > SUM(W1.Hours)) );
```

- It is also possible to do this with an embedded subquery in the FROM clause.

```
SELECT S1.PName, S1.SHrs
FROM   (SELECT   PName, Sum(Hours) AS SHrs
        FROM     Project JOIN Works_On ON (PNumber=PNo)
        GROUP BY PName ) AS S1
WHERE   S1.SHrs >= ALL
         (SELECT SHrs
          FROM (SELECT   PName, Sum(Hours) AS SHrs
                FROM     Project JOIN Works_On ON (PNumber=PNo)
                GROUP BY PName ) AS Pointless);
```

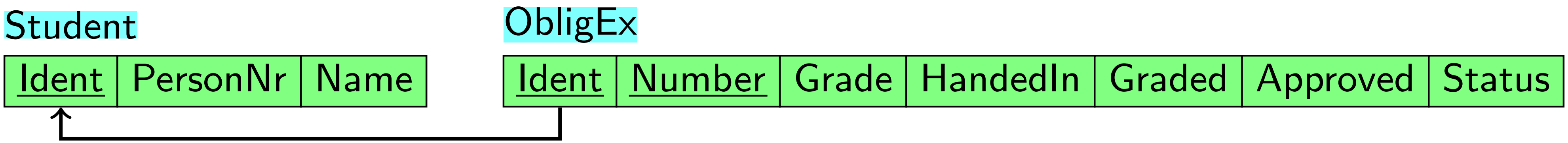
- Note the alias Pointless which is required by SQL rules.

A Schema for a Grading Database

- Shown below are SQL definitions for two of the relations for a grading database similar to that used for this course.

```
CREATE TABLE Student (
  Name        VARCHAR(40)  Not Null,
  Personnr    CHAR(11)     Not Null, -- YYMMDD-XXXX
  Ident       VARCHAR(10)  Not Null, -- @cs.umu.se user ID
  PRIMARY KEY (Ident),
  UNIQUE      (Personnr)  );

CREATE TABLE ObligEx (
  Ident       VARCHAR(10)  Not Null,
  Number      INTEGER      Not Null, -- exercise number (1 or 2)
  Grade       INTEGER              , -- numerical point score
  HandedIn    DATE                 , -- date first submitted
  Graded      DATE                 , -- date first graded
  Approved    DATE                 , -- date approved satisfactory
  Status      CHAR(1)              , -- S or U
  PRIMARY KEY (Ident, Number),
  CONSTRAINT obligex_ident_fkey FOREIGN KEY (Ident)
             REFERENCES Student(Ident) ON UPDATE CASCADE );
```



Outer Joins

Preliminary goal: Define a query with the form of `ObligEx`, for Exercise 1 only, with an entry for every student and nulls for missing values.

- The *(left) outer join* operation delivers the desired structure.
- It is similar to an (inner) join, but it fills in missing matches with nulls.
- It is called *left* because the left table in the construction is the base; the right table is padded with nulls.

```
SELECT S.Ident, E1.Grade, E1.HandedIn, E1.Graded,
                           E1.Approved, E1.Status
FROM   (SELECT Ident FROM Student) AS S LEFT OUTER JOIN
       (SELECT * FROM  ObligEx WHERE Number=1) AS E1
       ON (S.Ident=E1.Ident);
```

- A *right* outer join is defined analogously.
- A *full* outer join is the combination of a left outer join and a right outer join.

Outer Joins —2

- Now add on the second exercise as well by using a second outer join.

ObligExAll

Ident	Gr1	DateH1	DateG1	DateA1	St1	Gr2	DateH2	DateG2	DateA2	St2

```
SELECT S.Ident, E1.Grade AS Gr1, E1.HandedIn AS DateH1,
                E1.Graded AS DateG1, E1.Approved AS DateA1,
                E1.Status St1,
                E2.Grade AS Gr2, E2.HandedIn AS DateH2,
                E2.Graded AS DateG2, E2.Approved AS DateA2,
                E2.Status AS St2
FROM   (SELECT Ident FROM Student) AS S LEFT OUTER JOIN
       (SELECT * FROM ObligEx WHERE Number=1) AS E1
       ON (S.Ident=E1.Ident) LEFT OUTER JOIN
       (SELECT * FROM ObligEx WHERE Number=2) AS E2
       ON (S.Ident=E2.Ident);
```

Outer Joins —3

- Recall the query for computing summary information about projects.

```
SELECT   PName, MIN(Hours), MAX(Hours),
                AVG(Hours), SUM(Hours), COUNT(*)
FROM     Project JOIN Works_On ON (PNumber=PNo)
GROUP BY PName
UNION
SELECT   PName, 0, 0, 0, 0, 0
FROM     Project AS P1
WHERE    (NOT EXISTS
          (SELECT *
           FROM   Project JOIN Works_On ON (PNumber=PNo)
           WHERE  P1.PNumber=PNo));
```

- A query which is almost the same using outer join is much simpler.

```
SELECT   PName, MIN(Hours), MAX(Hours),
                AVG(Hours), SUM(Hours), COUNT(*)
FROM     Project LEFT OUTER JOIN Works_On ON (PNumber=PNo)
GROUP BY PName;
```

- The only difference is that the explicit `0` values of the first query are replaced with NULL.

The COALESCE Operator

- Recall the query of the previous slide.

```
SELECT   PName, Min(Hours), Max(Hours),
                Avg(Hours), Sum(Hours), Count(*)
FROM     Project LEFT OUTER JOIN Works_On ON (PNumber=PNo)
GROUP BY PName;
```

- The COALESCE operator selects its first non-null argument, and may be used to put zeros where they belong.

```
SELECT   PName, COALESCE(Min(Hours),0), COALESCE(Max(Hours),0),
                COALESCE(Avg(Hours),0), COALESCE(Sum(Hours),0),
                COALESCE(Count(*),0)
FROM     Project LEFT OUTER JOIN Works_On ON (PNumber=PNo)
GROUP BY PName;
```

- Coalesced columns may also be cast and renamed:

```
SELECT   PName, COALESCE(Min(Hours),0), COALESCE(Max(Hours),0),
                CAST(COALESCE(Avg(Hours),0) AS DECIMAL(8,2)) AS AVG_Hours,
                COALESCE(Sum(Hours),0), COALESCE(Count(*),0)
FROM     Project LEFT OUTER JOIN Works_On ON (PNumber=PNo)
GROUP BY PName;
```

Views

- A *view* is a virtual table which is constructed using a query.
- It differs from a query in that:
  - It persists in time, just as a true table.
  - Its state reflects updates to the true tables.
  - It has a name and may be used in large part as would any table of the database.

Basic syntax: `CREATE VIEW <name> AS <query>;`

`CREATE OR REPLACE VIEW <name> AS <query>;`

Example:

```
CREATE VIEW Project_Summary_Info AS
SELECT   PName, Min(Hours), Max(Hours),
                Avg(Hours), Sum(Hours), Count(*)
FROM     Project JOIN Works_On ON (PNumber=PNo)
GROUP BY PName
UNION
SELECT   PName, 0, 0, 0, 0, 0
FROM     Project as P1
WHERE    (NOT EXISTS
          (SELECT *
           FROM   Project JOIN Works_On ON (PNumber=PNo)
           WHERE  P1.PNumber=PNo));
```

Naming Columns of Views

- Here is how it is done:

```
CREATE VIEW Project_Summary_Info
(Project_Name, Min_Hours, Max_Hours, Avg_Hours,
                              Total_Hours, Num_Empl)
AS
SELECT    PName, Min(Hours), Max(Hours),
                 Avg(Hours), Sum(Hours), Count(*)
FROM      Project JOIN Works_On ON (PNumber=PNo)
GROUP BY PName
UNION
SELECT    PName, 0, 0, 0, 0, 0
FROM      Project as P1
WHERE     (NOT EXISTS
           (SELECT *
            FROM    Project JOIN Works_On ON (PNumber=PNo)
            WHERE   P1.PNumber=PNo));
```

Views in Queries

- In (read) queries, views may be used just as ordinary declared relations (tables).
- The following query uses the view definition on the previous slide.

```
SELECT Project_Name, Total_Hours, DName, Mgr_SSN
FROM   Project_Summary_Info NATURAL JOIN Department;
```

- Here is the view definition from the previous slide, for completeness.

```
CREATE VIEW Project_Summary_Info
(Project_Name, Min_Hours, Max_Hours, Avg_Hours,
                               Total_Hours, Num_Empl)
AS
SELECT    PName, Min(Hours), Max(Hours),
                 Avg(Hours), Sum(Hours), Count(*)
FROM      Project JOIN Works_On ON (PNumber=PNo)
GROUP BY PName
UNION
SELECT    PName, 0, 0, 0, 0, 0
FROM      Project as P1
WHERE     (NOT EXISTS
           (SELECT *
            FROM   Project JOIN Works_On ON (PNumber=PNo)
            WHERE  P1.PNumber=PNo));
```

Updates to Views

- Under limited conditions, updates to views are possible in standard SQL.
- There must be an "obvious" way to reflect the update to the true tables.
- Unfortunately, PostgreSQL does not support updates to views.
- Because the rules are complex in any case, they will not be considered further in this course.
- If updates to a view are essential, it is often best to realize the view indirectly, via an application program which interfaces to the database.

The Logic of Null Values

- The value of NULL is treated as *unknown* in truth-valued expressions.

`A`	`B`	`(A OR B)`	`(A AND B)`	`(NOT A)`
false	**false**	**false**	**false**	**true**
false	**true**	**true**	**false**	**true**
true	**false**	**true**	**false**	**false**
true	**true**	**true**	**true**	**false**
true	**unknown**	**true**	**unknown**	**false**
false	**unknown**	**unknown**	**false**	**true**
unknown	**true**	**true**	**unknown**	**unknown**
unknown	**false**	**unknown**	**false**	**unknown**
unknown	**unknown**	**unknown**	**unknown**	**unknown**

- Conditions of the form `(A=B)` also evaluate to **unknown** when at least one of the arguments evaluates to NULL.
- Expressions which evaluate to **unknown** are not considered to be true for the purpose of a query in SQL.
- Recall the queries on the next slide.

Illustration of Unknown Values in Logical Expressions

Query: Find the names and departments of those employees whose supervisor is the same as the manager of the department in which the employee works.

```
SELECT LName, FName, MInit, DName
FROM   Employee JOIN Department
                ON ((DNo=DNumber) AND (Super_SSN=Mgr_SSN));
```

Query: Find the names and departments of those employees whose supervisor is the not the same as the manager of the department in which the employee works.

```
SELECT LName, FName, MInit, DName
FROM   Employee JOIN Department
                ON ((DNo=DNumber) AND (NOT (Super_SSN=Mgr_SSN)));
```

Observation: An employee with NULL as `Super_SSN` is in the result of neither query.

- The conditions `(Super_SSN=Mgr_SSN)` and `(NOT (Super_SSN=Mgr_SSN))` evaluate to **unknown**.

BLOBs and TEXTs

- Special data, including multimedia data, have become very commonplace in recent years.
- There are two types which may be used for such data, defined in the SQL:2003 standard.

  `BLOB`: Binary Large OBject. `TEXT`: Text data object.

Example:

```
CREATE TABLE Employee
 (...
  Photo       BLOB,
  CV          TEXT,
  ...);
```

MySQL: Supports both with variations for the maximum size.

- `SMALLBLOB`, `BLOB`, `MEDIUMBLOB`, `LARGEBLOB`
- `TINYTEXT`, `TEXT`, `MEDIUMTEXT`, `LONGTEXT`

PostgreSQL: Supports `TEXT` and `OID` (its variant of `BLOB`).

- Type `TEXT` has no limit on length

- These types will not be considered further in this course.
- Support is still very nonstandard across dialects.

Further Features of SQL

SQL functions: Just as in other programming languages, it is possible to write functions in SQL and then call them within a query.

Persistent Stored Modules (PSMs) : It is possible to write functions which are written in another, imperative language, and call them from SQL.

- In ODBC, to be studied later in this course, SQL is called from an imperative language.
- PSMs are the other way around — an imperative language is called from SQL.

Triggers: Triggers are special functions, called when an update occurs.

- Triggers are particularly useful in enforcing complex constraints.

Iteration: SQL supports limited iteration ((mis)named *recursion*), for computing closures such as the set of all ancestors of a person.

Advanced aggregation and OLAP: SQL supports more advanced aggregation operators than those covered in this course, including in particular those associated with *OLAP*, on-line analytical processing.

- All of these topics are covered in the followup course 5DV120, *Database System Principles*.