SQL — Part 1

5DV119 — Introduction to Database Management

Umeå University
Department of Computing Science

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

The SQL "Standard"

- SQL is the "standard" language for access to relational databases.
- There have been many standard versions, beginning with *SQL92*, and currently ending with *SQL:2011*.
- But many of its features have evolved from earlier vendor-specific ones.
- As a result, almost no relational DBMS follows the standard very closely.
- Even the most basic things, such as the datatypes representing dates and times, differ from system to system and greatly limit code portability.
- Most systems implement a "superset of a subset" of the standard specification.
- Nevertheless, the basic features of most systems are very similar, even if not completely compatible.
- In this course, the focus will be upon these basic features.
- The open-source systems *PostgreSQL* and *MySQL* will be the main foci.

The Nature of SQL

- SQL is a very complex language.
- There are many ways of doing the same thing.
- There are many arcane features known only to gurus.
- Many of these work only with certain systems, and/or work differently with different systems.
- It takes many years to master the language completely (if that is possible at all).
- In this course, the emphasis will be on straightforward ways to carry out common tasks.
- "Tricks" which are intended to show how clever the programmer is will be avoided if possible.
- Also, the emphasis will be upon using that part of the language which is most portable across the major dialects.

The Parts of SQL

SQL consists of several parts:

DDL: The *data-definition language* provides commands for defining and altering database schemata, including integrity constraints and virtual relations called *views*.

DML: The *data-manipulation language* provides commands for both querying and updating databases.

Transactions: There are basic SQL commands for specifying transactions.

- These are limited in scope and most systems have their own ways of managing transactions.

Authorization: SQL contains directives for granting and revoking privileges.

Access from a host languages: SQL contains some basic commands for use when the language is embedded in a host programming language.

- These are limited in scope and many approaches to hosting SQL, including ODBC, have their own ways of of doing similar things.

General Notes on Syntax

Case sensitivity:

- Keywords are case insensitive in both PostgreSQL and MySQL.
- Keywords will be written in all caps in these slides.

PostgreSQL: Identifiers are folded to lower case and so are case insensitive.

MySQL: Case sensitivity of identifiers is dependent upon the configuration.

Linux: Case sensitive by default.

Windows: Case insensitive by default.

Clients for Direct Access to SQL via PostgreSQL

- The best way to access PostgreSQL is via the command-line interface:

  `psql -username <username> -hostname <servername> <dbname>`

  or

  `psql -U <username> -h <servername> <dbname>`

- `<username>` and `<dbname>` are usually the same on the systems of the department.
- `<hostname>` is `postgres` on the systems of the department.
- With `ident` authentication, no special password is used.
- However, it is necessary to log into a departmental Linux system first and run these commands from a shell.
- From a Windows machine, use `PuTTY` or something similar to obtain a shell on a Linux machine via ssh.
- `\?` shows a list of system commands.
- Use ctrl-Z and then kill the process if parsing becomes too confused.

Clients for Direct Access to SQL via MySQL

- The command-line interface is invoked with:

```
mysql --user <username> --host <servername> --password <dbname>
```

or

```
mysql -u=<username> -h=<hostname> -p <dbname>
```

- `<username>` and `<dbname>` are usually the same on the systems of the department.
- `<hostname>` is `mysql` on the systems of the department.
- A prompt will appear at which the password must be given.
- There is no `ident` authentication with MySQL.
- `\h` or `help;` shows a list of system commands.
- In addition, there are also some usable graphical interfaces.
- The *MySQL Query Browser* `mysql-query-browser` is one of the more common ones.
- It has been superseded by the *MySQL Workbench* in newer installations.

Remarks on MySQL Database Engines

If you decide to install MySQL on your own computer:

- MySQL is a DBMS *front end*.
- There are a number of *storage engines* which may be used with it.
- Make sure that you use the *InnoDB* engine.
- The older *MyISAM* engine does not support many useful features, such as foreign-key constraints.
  -  It accepts the associated directives but silently ignores them.
- It is the default with some Linux distributions.
- To see which engine your MySQL is running,
  - Connect to the MySQL server.
  - Issue the command `show variables;`.
  - `storage_engine` should be `InnoDB`.
- If you need help to change this, ask.

 Remember that final versions of all submissions should be tested for compatibility on the departmental servers.

Remarks on MariaDB

If you decide to install MySQL on your own computer:

- Recently, MySQL was acquired by Oracle corporation.
- Although it is officially open source, there have been and will be inevitable changes.
- *MariaDB* is a fork of MySQL which is completely independent of Oracle.
- The force behind it is Monty Widenius, a Finnish computer scientist who was also an original developer of MySQL.
  - *My* and *Maria* are the names of his daughters.
- It is designed to be a drop-in replacement for MySQL.
  - This means that anything which works with MySQL should work with MariaDB as well.
- It is also said to have better performance a better bug-reporting and bug-fixing system.
- If you can use MariaDB instead of MySQL, it should work fine.

Remember that final versions of all submissions should be tested for compatibility on the departmental servers.

Comments in SQL

```
/* Some simple code
        to illustrate the use of comment delimiters in SQL
                                    Stephen J. Hegner 23.01.13 */
SELECT LName /* Last Name */, FName /* First Name */
FROM Employee    -- Selecting the last and first names of
WHERE Sex='F';    -- females from the Employee relation.
```

- There are two standard ways of inserting comments into SQL code:

  Block comments: As in the programming language C, `/*` is an open marker while `*/` is a close marker for comments.

  - Such comments may span several lines or be inserted within lines of SQL code.

  Line comments: Anything after two consecutive dashes `--` is a comment for the rest of that line.

  - Such comments may begin at any point, but always run to the end of the line and terminate there.
  - It is similar to the `#` comment marker of Python.

Defining Tables

```
CREATE TABLE Employee
   (FName      VARCHAR(15)         NOT NULL,
    MInit      CHAR,
    LName      VARCHAR(15)         NOT NULL,
    SSN        CHAR(9)             NOT NULL,
    BDate      DATE,
    Address    VARCHAR(30),
    Sex        CHAR,
    Salary     DECIMAL(10,2),
    Super_SSN  CHAR(9),
    DNo        INT                 NOT NULL,
    PRIMARY KEY (SSN),
    FOREIGN KEY (Super_SSN) REFERENCES Employee(SSN),
    FOREIGN KEY (DNo)       REFERENCES Department(DNumber)
   );
```

- `VARCHAR(n)` is a type of at most `n` characters.
- `CHAR(n)` is a type of exactly `n` characters.
- `DATE` is an SQL standard but varies from installation to installation.
  PostgreSQL,MySQL: `YYYY-MM-DD` without timestamp.
- `INT` is type integer.
- `DECIMAL(n,m)` fixedpoint, `n` digits total, `m` to right of decimal point.
- `SSN` is not defined as `DECIMAL(9)` since leading zeros would not be displayed.

Defining Tables 2

```
CREATE TABLE Department
   (DName           VARCHAR(15)        NOT NULL,
    DNumber         INT                NOT NULL,
    Mgr_SSN         CHAR(9),
    Mgr_Start_Date  DATE,
    PRIMARY KEY (DNumber),
    UNIQUE (DName),
    FOREIGN KEY (Mgr_SSN)   REFERENCES Employee(SSN),
   );
CREATE TABLE Dept_Locations
   (DNumber         INT                NOT NULL,
    DLocation       VARCHAR(15)        NOT NULL,
    PRIMARY KEY (DNumber,DLocation),
    FOREIGN KEY (DNumber)   REFERENCES Department(DNumber)
   );
```

- `UNIQUE` identifies a candidate key which is not the primary key.
- Note how keys with several attributes are written.
- Note that the same name may be used for an attribute of two distinct relations.

Defining Tables – CHECK Constraints

```
CREATE TABLE Employee
   (...
    SSN         CHAR(9)              NOT NULL,
     ...
    Sex         CHAR,
     ...
    Salary      DECIMAL(10,2),
     ...
    PRIMARY KEY (SSN),
    FOREIGN KEY (Super_SSN) REFERENCES Employee(SSN),
    FOREIGN KEY (DNo)       REFERENCES Department(DNumber),
    CHECK (SSN SIMILAR TO
      '[0-9][0-9][0-9][0-9][0-9][0-9][0-9][0-9][0-9]'),
    CHECK (Sex IN ('M','F')),
    CHECK (Salary < 200000)
   );
```

- More complex constraints may be stated in CHECK clauses.
- These constraint may also be named.

MySQL parses such constraints but the InnoDB engine does not enforce them.

Defining Tables – Naming Constraints

```
CREATE TABLE Department
   (DName           VARCHAR(15)        NOT NULL,
    DNumber         INT                NOT NULL,
    Mgr_SSN         CHAR(9),
    Mgr_Start_Date  DATE,
    CONSTRAINT Dept_PK  PRIMARY KEY (DNumber),
    CONSTRAINT Dept_CK1 UNIQUE (DName),
    CONSTRAINT Dept_FK1 FOREIGN KEY (Mgr_SSN) REFERENCES Employee(SSN),
   );
```

- Constraints may be named.
- This facilitates enabling and disabling them dynamically using the `ALTER TABLE` directive.
  - Only named constraints may be enabled and disabled.
  - Only named constraints may be added and removed.
- These are useful when populating tables.
  - Examples later.

```
ALTER TABLE Department DROP CONSTRAINT Dept_FK1;
 ...
ALTER TABLE Department ADD CONSTRAINT Dept_FK1
            FOREIGN KEY (Mgr_SSN) REFERENCES Employee(SSN);
```

Forward References in Table Definitions

```
CREATE TABLE Employee
   (SSN        CHAR(9)             NOT NULL,
    SuperSSN   CHAR(9),
     ...
    PRIMARY KEY (SSN),
    FOREIGN KEY (DNo)         REFERENCES Department(DNumber),
   );
CREATE TABLE Department
   (DNumber          INT                  NOT NULL,
    Mgr_SSN          CHAR(9),
    ...
    PRIMARY KEY (DNumber),
    FOREIGN KEY (Mgr_SSN) REFERENCES Employee(SSN),
   );
```

- Note that there is a forward reference to `Department` in the declaration of `Employee`.
- This is not allowed in most dialects of SQL.
- A table must be created before foreign-key references to its key are possible.
- The solution is to add the constraint after the definition of the second table.

Adding and Dropping Constraints on Tables

```
CREATE TABLE Employee
   (SSN        CHAR(9)             NOT NULL,
    SuperSSN   CHAR(9),
     ...
    -- Foreign key to Department is not declared here.
    -- FOREIGN KEY (DNo)        REFERENCES Department(DNumber),
   );

CREATE TABLE Department
   (DNumber          INT                  NOT NULL,
    Mgr_SSN          CHAR(9),
    ...
    PRIMARY KEY (DNumber),
    FOREIGN KEY (Mgr_SSN) REFERENCES Employee(SSN),
   );

ALTER TABLE Employee ADD CONSTRAINT Emp_FK2
    FOREIGN KEY (DNo) REFERENCES Department(DNumber);

ALTER TABLE Employee DROP CONSTRAINT Emp_FK2;

-- MySQL does not support the standard syntax for DROP; it wants:
ALTER TABLE Employee DROP FOREIGN KEY Emp_FK2;
```

- Added and dropped constraints must be named.
  - `Emp_FK2` is the name of the above constraint.

The Basic From of a Query

- The basic form of an SQL query is as follows.

```
SELECT <attributes>
FROM   <tables>
WHERE  <conditions>
```

- The `WHERE` part is optional but most interesting queries require it.
- A very simple query:

```
SELECT FName, MInit, LName, SSN
FROM   Employee;
```

- Star captures all attributes:

```
SELECT *
FROM   Employee;
```

- A simple condition:

```
SELECT FName, MInit, LName, SSN
FROM   Employee
WHERE  Salary >= 30000;
```

More Complex Conditions

```
SELECT FName, MInit, LName, SSN
FROM   Employee
WHERE  (SALARY >= 30000) OR (SEX = 'F');

SELECT FName, MInit, LName, SSN
FROM   Employee
WHERE  (SALARY >= 30000) AND (NOT (SEX = 'M'));

SELECT FName, MInit, LName, SSN
FROM   Employee
WHERE  (SALARY >= 30000) AND (SEX <> 'M');
```

- Complex logical and arithmetic conditions are also possible.
- Shown above is just a sample.

Recommendation: Always use parentheses for clear indication of association.

- Do not depend upon default rules of precedence, which may vary amongst implementations.

- Many other types of more complex conditions will be illustrated later.

Renaming the Columns of a Query

- Columns may be given explicit names using the AS directive.

```
SELECT FName AS First_Name, MInit AS Middle_Initial,
       LName AS Last_Name, SSN AS Soc_Sec_Num
FROM   Employee
WHERE  (SALARY >= 30000) OR (SEX = 'F');
```

- These names will appear as the column headers.
- However, such name changes cannot be used as aliases in the WHERE clause.

Does not work:

```
SELECT FName AS First_Name, MInit AS Middle_Initial,
       LName AS Last_Name, SSN AS Soc_Sec_Num
FROM   Employee
WHERE  (First_Name='Alicia');
```

- The use of AS in the FROM clause has different scoping, as will be illustrated in examples to follow.

Duplicates and Order

```
SELECT Salary
FROM   Employee;

SELECT DISTINCT Salary  -- Duplicates removed
FROM   Employee;

SELECT SSN, Salary
FROM   Employee
ORDER BY Salary DESC;  -- DESCending order

SELECT SSN, Salary
FROM   Employee
ORDER BY Salary ASC;  -- ASCending order

SELECT FName, MInit, LName, SSN
FROM   Employee
ORDER BY SSN ASC;

SELECT FName, MInit, LName, SSN
FROM   Employee
ORDER BY LName, FName, MInit;   -- Major-to-minor order;
                                -- default collation
```

Queries on Two Relations

Problem: For each employee, find the names of the dependents. Identify the employee by name.

- This query requires information from both the `Employee` and the `Dependent` relations.
- A first try:

```
SELECT FName, MInit, LName, Dependent_Name
FROM   Employee, Dependent;
```

- Does this work?
- No, it generates the *Cartesian product* of the two relations.
- A *join condition* is required:

```
SELECT FName, MInit, LName, Dependent_Name
FROM   Employee, Dependent
WHERE  (SSN=ESSN);
```

Queries on Many Relations

Problem: For each employee, find the names of the projects on which that employee works. Identify the employee by name.

- This query requires information from the `Employee`, `Project`, `Works_On` relations.
- It requires two join operations:

```
SELECT FName, MInit, LName, PName
FROM   Employee, Project, Works_On
WHERE  (SSN=ESSN) AND (PNo=PNumber);
```

The JOIN Operation of SQL

- The join operation is used so often that SQL has a special notation for it.

```
SELECT FName, MInit, LName, Dependent_Name
FROM   Employee INNER JOIN Dependent ON (SSN=ESSN);
```

  is equivalent to

```
SELECT FName, MInit, LName, Dependent_Name
FROM   Employee, Dependent
WHERE  (SSN=ESSN);
```

  and

```
SELECT FName, MInit, LName, PName
FROM   Employee INNER JOIN Works_On  ON (SSN=ESSN)
                INNER JOIN Project   ON (PNo=PNumber);
```

  is equivalent to

```
SELECT FName, MInit, LName, PName
FROM   Employee, Project, Works_On
WHERE  (SSN=ESSN) AND (PNo=PNumber);
```

- The keyword INNER may be omitted, as it is the default.
- Such queries may include a WHERE clause as well.

```
SELECT FName, MInit, LName, Dependent_Name
FROM   Employee INNER JOIN Dependent ON (SSN=ESSN)
WHERE (Salary > 30000);
```

Queries with Overloaded Attribute Names

Query: Find the locations of each department; identify departments by name.

- The following will not work:

```
SELECT DName, DLocation
FROM   Department JOIN Dept_Locations ON (DNumber=DNumber);
```

- SQL cannot resolve `DNumber` because it is an attribute of both relations.
- Name resolution follows a familiar notation from programming languages:

```
SELECT DName, DLocation
FROM   Department JOIN Dept_Locations
         ON (Department.DNumber=Dept_Locations.DNumber);
```

- Another possibility is to use *aliases*:

```
SELECT DName, DLocation
FROM   Department AS D JOIN Dept_Locations AS DL
                    ON (D.DNumber=DL.DNumber);
```

or

```
SELECT DName, DLocation
FROM   Department AS D, Dept_Locations AS DL
WHERE  (D.DNumber=DL.DNumber);
```

The Natural Join of SQL

- When the names of the columns to be joined are the same in both relations, instead of using qualifiers or aliases, the *natural join* operation may be used.

Query: Find the locations of each department.

```
SELECT DName, DLocation
FROM   Department NATURAL JOIN Dept_Locations;
```

- Here the match is on all columns with matching names in the relations.
  - In this case, `DNumber`.
- The matching columns are not repeated.
- If there are no matching columns, the Cartesian product is the result (which is almost never what is intended).

Try the following query to see this effect:

```
SELECT LName, PNo, Hours
FROM   Employee NATURAL JOIN Works_On;
```

Joins on Several Columns and Compound Join Conditions

- It is possible to join on several columns.

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));
```

- Compound join conditions including operations other than conjunction are also possible.

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.

- More on this later.

Theta Joins

- Join conditions need not be based only upon equality.
- Joins based upon other comparison operators are often called *theta joins*.

Query: Find those employees who have a dependent who is older.

```
SELECT DISTINCT LName, FName, MInit, SSN
FROM   Employee JOIN Dependent
                ON ((SSN=ESSN) AND
                    (Employee.BDate > Dependent.BDate));
```

Queries Which Use the Same Relation More Than Once

Query: Find the name of the supervisor of each employee.

- Here it is necessary to use aliases:

```
SELECT E.LName, E.FName, E.MInit, S.LName, S.FName, S.MInit
FROM   Employee as E JOIN Employee as S
                       ON (E.Super_SSN=S.SSN);
```

or

```
SELECT E.LName, E.FName, E.MInit, S.LName, S.FName, S.MInit
FROM   Employee as E, Employee as S
WHERE  (E.Super_SSN=S.SSN);
```

- With this query, if an employee has no supervisor, no entry is produced.
- To address this issue properly requires the use of the `UNION` operation.

Set and Multiset Operations in SQL

- Queries return *multisets* of tuples.

Multiset: A "set" in which elements may occur more than once.

- The usual set operations apply to multisets in SQL.

Name	Symbol	SQL
union	$\cup$	`UNION`
intersection	$\cap$	`INTERSECT`
difference	$\setminus$ or $-$	`EXCEPT`

Some implementations of SQL do not support `INTERSECT` and `EXCEPT`.

Example: Mostly, these are small single-user systems such as MS Access.

Example: MySQL is alone amongst "major" systems which do not support these set operations. ☹

- Later, it will be shown how to achieve the same results using embedded subqueries.

A Simple Application of the UNION Operation

Query: Find the name of the supervisor of each employee, with blank entries if the employee has no manager.

- This can be accomplished with the aid of the UNION operation.

```
SELECT E.LName, E.FName, E.MInit, S.LName, S.FName, S.MInit
FROM   Employee as E JOIN Employee as S ON (E.Super_SSN=S.SSN)
UNION
SELECT LName, FName, MInit, '', '', '␣'
FROM   Employee
WHERE  Super_SSN IS NULL;
```

- The same query with custom column headers:

```
SELECT E.LName AS Emp_LNAME, E.FName AS EMP_FName,
       E.MInit AS EMP_MInit,
       S.LName AS Super_LName, S.FName AS Super_FName,
       S.MInit AS Super_MInit
FROM   Employee as E JOIN Employee as S ON (E.Super_SSN=S.SSN)
UNION
SELECT LName, FName, MInit, '', '', '␣'
FROM   Employee
WHERE  Super_SSN IS NULL;
```

Pattern Matching in SQL

- SQL has two features for pattern matching:

  `LIKE`: uses a special syntax.

  `SIMILAR TO`: uses regular expressions.

Query: Find all employees whose last names begin with the letter W.

- % is the wildcard symbol for `LIKE`.
- The following works in both PostgreSQL and MySQL:

```
SELECT LName, FName, MInit
FROM   Employee
WHERE  LName LIKE 'W%';
```

- MySQL uses case-insensitive matching, and so the following works as well.

```
SELECT LName, FName, MInit
FROM   Employee
WHERE  LName LIKE 'w%';
```

- `LIKE` in PostgreSQL uses case-sensitive matching,
- Use `ILIKE` for case-insensitive matching.

```
SELECT LName, FName, MInit
FROM   Employee
WHERE  LName ILIKE 'w%';
```

Pattern Matching in SQL Using `SIMILAR TO`

MySQL does not support `SIMILAR TO`, although it is part of the SQL:1999 standard.

- Here are some examples which run under PostgreSQL:

Query: Find all employees whose last names begin with the letter W.

```
SELECT LName, FName, MInit
FROM   Employee
WHERE  LName SIMILAR TO 'W%';
```

```
SELECT LName, FName, MInit
FROM   Employee
WHERE  LName SIMILAR TO 'W[a-z]*';
```

- Underscore matches a single character with both `SIMILAR TO` and `LIKE`.

```
SELECT LName, FName, MInit
FROM   Employee
WHERE  LName SIMILAR TO 'W___';
```

Additional Examples of Pattern Matching

Query: Find the name and SSN of all employees whose SSN has 3 or 8 as the third digit from the left.

```
SELECT LName, FName, MInit, SSN
FROM   Employee
WHERE  (SSN LIKE '__3%') OR (SSN LIKE '__8%');
```

- `SIMILAR TO` has a built-in disjunction operator:

```
SELECT LName, FName, MInit, SSN
FROM   Employee
WHERE  SSN SIMILAR TO '__(3|8)%';
```

- Tilde is the escape character for both operators.

Query: Find all last names containing the character %:

```
SELECT LName, FName, MInit, SSN
FROM   Employee
WHERE  LName LIKE '%~%%';
```

Basic Update Operations in SQL

- There are three basic forms of update in SQL:

  Insertion: Using the `INSERT` directive, new tuples may be added to a relation.

  Deletion: Using the `DELETE` directive, existing tuples may be removed from a relation.

  Modification: Using the `UPDATE` directive, the values in fields of existing tuples may be changed.

A note on terminology:

- The word *update* has to distinct meanings, both of which are entrenched in the database literature and practice.
  1. In the general database literature, an update may refer to any operation which changes of the state of a database.
  2. In SQL, an update refers to a modification of an existing tuple or tuples.
- It is necessary to resolve which of these two meanings is intended by using context.

The SQL INSERT Directive

- There are two basic forms of insertion:
- In the first, all fields for a single tuple are specified from left to right:

```
INSERT INTO Employee
VALUES ('Jane', 'S', 'User', '000112222', '1960-01-01',
        '13␣Mockingbird,␣El␣Paso,␣TX', 'F', 90000.00, NULL, 1);
```

- In the second, fields may be listed in any order and missing fields are taken to be NULL:

```
INSERT INTO Employee (SSN, LName, FName, MInit, Sex, BDate,
                      Address, Salary, DNo)
VALUES ('000112222', 'User', 'Jane', 'S', 'F', '1960-01-01',
        '13␣Mockingbird,␣El␣Paso,␣TX', 90000.00, 1);
```

- In both cases, the insertion will fail if any integrity constraint would be violated by the insertion.
- There is also a third form of insertion which involves views and which will be considered later.

The SQL DELETE Directive

- Deletion involves the use of a WHERE clause to identify the tuples to be deleted.:

```
DELETE FROM Employee
       WHERE SSN='000112222';
```

- In contrast to the INSERT directive, DELETE may remove several tuples at once.

```
DELETE FROM Employee
       WHERE Address LIKE '%El␣Paso%';

DELETE FROM Employee
       WHERE Sex='M';
```

- Deletion can never cause a violation of a PRIMARY KEY or UNIQUE constraint, but it is possible for a FOREIGN KEY to be violated.
- The deletion is not executed if it would cause such a violation.

The SQL UPDATE Directive

- Modification via UPDATE involves the use of a SET clause to identify the changes and a WHERE clause to identify the tuples to be modified:

```
UPDATE Employee
       SET Address '123 Main, El Paso, TX'
       WHERE SSN='000112222';
```

- UPDATE may modify several tuples at once.

```
UPDATE Employee
       SET Salary = Salary + 10000
       WHERE DNo=4;
```

- Modification can cause a violation of a PRIMARY KEY or UNIQUE constraint only if it alters the value of a primary or candidate key.
- Other forms of constraint may be violated, however.
- The update is not executed if it would cause such a violation.

Multiple Updates and Integrity Constraints

- By default, integrity constraints are checked after each `INSERT`, `DELETE`, and `UPDATE` operation.
- This can lead to *intermediate inconsistency* problems.

Example: Add the new employee Jane S. User and have her work in and be head of the newly added Security department.

```
INSERT INTO Employee
VALUES ('Jane', 'S', 'User', '000112222', '1960-01-01',
        '13␣Mockingbird,␣El␣Paso,␣TX', 'F', 90000.00, NULL, 6);

INSERT INTO Department
VALUES ('Security',6,'000112222','2011-09-01');
```

- Regardless of which order these operations are executed, a violation of a foreign-key constraint will result.
- In PostgreSQL and most other major systems, the solution involves two steps:
  - Use `DEFERRABLE` constraints.
  - Use *explicit transactions*.

Explicit Transactions 1

- The foreign key which references the Department relation is made *deferrable* and is then *deferred*.

```
CREATE TABLE Employee
   (<column declarations>
    <constraints other than fkey_emp2>);

CREATE TABLE Department
   (<column declarations>
    <constraints>);

ALTER TABLE Employee ADD CONSTRAINT
    fkey_emp2 FOREIGN KEY (DNo) REFERENCES Department (DNumber)
    DEFERRABLE INITIALLY DEFERRED;
```

- This means that it is not checked until the end of a transaction.

Notes:

- The constraint `fkey_emp2` is added using `ALTER TABLE Employee` after the `CREATE TABLE Department` directive to avoid forward-reference errors during table definition.
- The constraint `fkey_emp2` is made `DEFERRABLE` and set to be `INITIALLY DEFERRED` to allow transactions to defer checking it until commit time during update execution.

Explicit Transactions 2

- In both PostgreSQL and MySQL `BEGIN` and `COMMIT` markers may be used to identify the bounds of the transaction.

```
BEGIN;

INSERT INTO Employee
       VALUES ('Jane', 'S', 'User', '000112222', '1960-01-01',
               '13␣Mockingbird,␣El␣Paso,␣TX', 'F',
               90000.00, NULL, 6);

INSERT INTO Department
       VALUES ('Security',6,'000112222','2011-09-01');

COMMIT;
```

Note: If the constraint `fkey_emp2` already exists but is not `DEFERRABLE` or is `DEFERRABLE` but not `INITIALLY DEFERRED`, this may be changed by dropping the constraint and then adding it again with the desired properties.

```
ALTER TABLE Employee DROP CONSTRAINT fkey_emp2;
ALTER TABLE Employee ADD  CONSTRAINT fkey_emp2
       FOREIGN KEY (DNo) REFERENCES Department (DNumber)
       DEFERRABLE INITIALLY DEFERRED;
```

Multiple Updates and Integrity Constraints in MySQL

- Unfortunately, MySQL does not support deferrable constraints.
- One way to deal with this problem is to drop the constraint and then re-add it after the updates.
  - Not a good practice for ordinary updates.

```
CREATE TABLE Employee
   (...,
    CONSTRAINT FKey_Emp1 FOREIGN KEY (Mgr_SSN)
                         REFERENCES Employee(SSN),
   );

 ...

 -- Unfortunately, MySQL does not use the standard syntax:
    ALTER TABLE Employee DROP CONSTRAINT FKey_Emp1;
 -- It uses this instead:
    ALTER TABLE Employee DROP FOREIGN KEY FKey_Emp1;

 INSERT INTO Employee
        VALUES ('Jane', 'S', 'User', '000112222', '1960-01-01',
                '13 Mockingbird, El Paso, TX', 'F',
                 90000.00, NULL, 6);
 INSERT INTO Department
         VALUES ('Security',6,'000112222','2011-09-01');

 ALTER TABLE Employee ADD CONSTRAINT FKey_Emp1
             FOREIGN KEY (Mgr_SSN) REFERENCES Employee(SSN);
```

Multiple Updates and Integrity Constraints in MySQL — 2

- A second way to deal with the problem of circular integrity constraints is to do the update in stages which preserve integrity.
- It is ugly and creates incorrect intermediate data but it works.
- First insert Jane User into Employee using an existing department.

```
INSERT INTO Employee
       VALUES ('Jane', 'S', 'User', '000112222', '1960-01-01',
              '13␣Mockingbird,␣El␣Paso,␣TX', 'F',
              90000.00, NULL, 1);
```

- Now add the new Security department with Jane as head.

```
INSERT INTO Department VALUES ('Security',6,'000112222','2011-09-01');
```

- Finally, move Jane to work in her new department.

```
UPDATE Employee SET DNo=6 WHERE SSN='000112222';
```

- This problem only arises when there are cycles in the foreign-key graph.
- In all other cases, the updates may be ordered to avoid intermediate constraint violations.

Cascading Updates

- By default, if an update would violate an integrity constraint, that update fails and the database is not changed.
- However, it is possible to *cascade* `DELETE` and `UPDATE` directives with respect to foreign-key constraints.

Example: Suppose that an Employee leaves the company (is deleted from the Employee relation.)

- It then makes sense to delete all information about the dependents of that employee as well.
- This may be accomplished via the `ON DELETE CASCADE` clause:

```
CREATE TABLE Dependent
   (ESSN     CHAR(9)      NOT NULL,
    ...
    FOREIGN KEY (ESSN) REFERENCES Employee (SSN)
                       ON DELETE CASCADE);
```

Cascading Updates — 2

- Cascading may also be applied to `UPDATE` (modification) directives.

Example: Suppose that the bureaucrats decide to change the project numbers.

- It then makes sense to reflect those changes in the `Works_On` relation as well.
- This may be accomplished via the `ON UPDATE CASCADE` clause:

```
CREATE TABLE Works_On
    (...
     PNo      INT      NOT NULL,
     ...
     FOREIGN KEY (PNo) REFERENCES Project (PNumber)
                       ON UPDATE CASCADE);
```

Cascading Updates —

 Cascading deletions can be dangerous.

Example: Suppose that deletions on `SSN` for `Employee` cascade to `Super_SSN`.

```
CREATE TABLE Employee
   (...
    SSN       CHAR(9)          NOT NULL,
    ...
    Super_SSN CHAR(9),
    PRIMARY KEY (SSN),
    FOREIGN KEY (Super_SSN) REFERENCES Employee (SSN)
                ON DELETE CASCADE);
```

- This can lead to long chains of deletions.

Question: What happens if the big boss Borg is deleted?

Answer: All tuples in the `Employee` relation are deleted (ignoring foreign-key constraints on other relations for the moment.)

- Clearly, cascading must be used with great care.
- Some systems, but not all, will detect and not allow recursive cascading.

Avoiding Ambiguity in Query Answers

- Many examples in these slides were of the form "Find the employees who ...".
- For simplicity, only the names of the employees were retrieved.
- However, this is a bit risky, since it is possible for two employees to have the same name.
- It is always advisable to retrieve a key (*e.g.*, `SSN`) as part of the answer.
- Use:

```
SELECT LName, FName, MInit, SSN, <other things>
FROM   Employee <and other relations>
WHERE  <condition>
```

- instead of:

```
SELECT LName, FName, MInit, <other things>
FROM   Employee <and other relations>
WHERE  <condition>
```