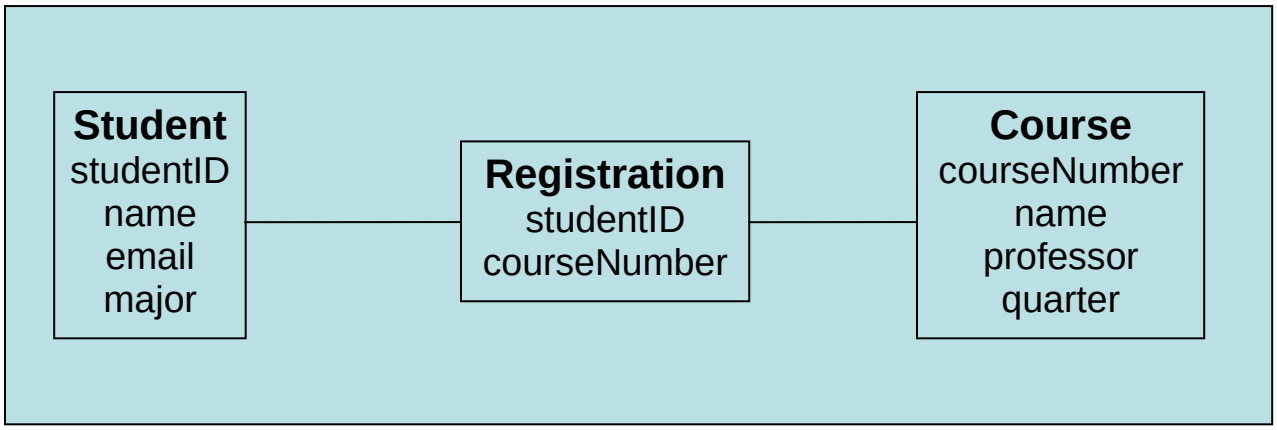
# Java Database Connectivity and Structured Query Language (Part II)
# ECE 155B, Spring 2012
# Discussion 6

## Example MySQL Database

**Design**
In this example we will build a simple three table database. The database has three tables: Student, Registration and Course tables. The following Entity Relationship (ER) diagram shows the basic design of the database:




The records in the Student table have fields (attributes) for studentID, name, email and major. The studentID is the Primary Key, which uniquely identifies each record in the table. The records in the Course table have fields for courseNumber, name, professor, quarter. The courseNumber is the Primary Key. The Registration table links the records in the Student table with the records in the Course table. A student can register for many courses and a course can have many students. The relationship between course and student is stored in the records in the Registration table, which have fields studentID and courseNumber that correspond to the Primary Keys of the related tables. Because a student cannot be registered for a course twice, both fields (studentID and courseNumber) are the joint Primary Key for the records in the Registration table.

**SQL Data Definition**
To create the database in MySQL we need to convert the above design into SQL statements. To create the three tables in the database, we use the CREATE TABLE statement as shown below:

```
CREATE TABLE Student (
  ID int NOT NULL AUTO_INCREMENT PRIMARY KEY,
  name VARCHAR(30) NOT NULL,
  email VARCHAR(45),
  major VARCHAR(15)
);


CREATE TABLE Course (
  courseNumber INT NOT NULL PRIMARY KEY,
  name VARCHAR(25) NOT NULL,
  professor VARCHAR(30),
  quarter VARCHAR(10) NOT NULL
);

CREATE TABLE Registration (
  studentID INT NOT NULL,
  courseNumber INT NOT NULL,
```

```
  PRIMARY KEY (studentID, courseNumber)
);
```

Note:  You should take advantage of the AUTO_INCREMENT command (used in the Student table above) because it will enable you to assign a PRIMARY key without worrying about it.  No counter is needed.

**Data Types**

Each field (column) must have a data type defined for it. As with most database systems, MySQL provides a number of data types. The most common are INT (an integer), FLOAT (a floating point number),

**DOUBLE** (a double precision floating point number),
**DATE**,
**DATETIME**,
**CHAR** (a fixed-length string),
**VARCHAR** (a variable-length string)

We used INT and VARCHAR in the example database. See the MySQL Reference Manual for more information about the data types that MySQL supports.
NOT NULL, AUTO_INCREMENT, PRIMARY KEY Keywords

Besides the data types in the above example, we used the following three key words or phrases.
NOT NULL
AUTO_INCREMENT
PRIMARY KEY

NOT NULL means that the field must have a value for every record.
AUTO_INCREMENT means that the field automatically gets the largest value plus one for each insert where the column value is 0 or NULL. Note that not all SQL database systems use this syntax for auto-numbering.

**PRIMARY KEY** indicates that the field (attribute) is the primary key for that table. Note that in MySQL, for tables where the Primary Key is made up of more than one attribute, you cannot use the keyword after each attribute name. Instead, you must define the Primary Key at the end of the SQL statement with the attributes in parenthesis (see the CREATE TABLE statement for the Registration table above).

**SHOW, DESCRIBE, DROP Commands**

Some other useful MySQL commands are SHOW, DESCRIBE, DROP.

The SHOW command creates a list of tables in the database. The following is an example of how you use the SHOW command in MySQL:

```
mysql> show tables;
+------------------+
| Tables in CRS  |
+------------------+
| Course          |
| Registration    |
| Student         |
+------------------+
```

The name of the database is CRS. It consists of three tables: Course, Registration, and Student.
To obtain a description of a table type: SHOW tableName. For example:

```
mysql> describe Course;
+-----------------+--------------+-------+-------+-----------+-------------------+
| Field           | Type         | Null  | Key   | Default |     Extra         |
```

```
+-----------------+--------------+-------+-------+----------+-----------------+
| courseNumber| int(8)       |       | PRI  | 0        |                 |
| name          | varchar(25)|       |       |          |                 |
| professor      | varchar(30)| YES  |       | NULL    |                |
| quarter       | varchar(10)|       |       |          |                 |
+-----------------+--------------+--------+-------+----------+-----------------+

mysql> describe Registration;
+-----------------+---------------+---------+-------+----------+----------------+
| Field           | Type          | Null    | Key  | Default |     Extra       |
+-----------------+---------------+---------+-------+----------+----------------+
| studentID      | int(9)        |         | PRI  |   0     |                 |
| courseNumber| int(8)        |         | PRI  |   0     |                 |
+-----------------+---------------+---------+-------+----------+----------------+

mysql> describe Student;
+---------------+---------------+----------+-------+----------+--------------------+
| Field          | Type          | Null    | Key   | Default |     Extra          |
+---------------+---------------+----------+-------+----------+--------------------+
| studentID     | int(9)        |          | PRI   |    0     | auto_increment |
| name          | varchar(30) |          |       |          |                    |
| email         | varchar(45) | YES     |       | NULL    |                    |
| major         | varchar(15) | YES     |       | NULL    |                    |
+---------------+------------ ---+----------+-------+----------+--------------------+
```

Finally, you might want to delete a table. You can do this with the DROP TABLE command. If tableName is the name of the table you want to delete, you would say:

```
DROP TABLE tableName;
```

Inserting Data

Adding data to a table is done with the INSERT command. It looks like this:

```
INSERT INTO Course VALUES
  (100, 'ECE 155B Network Programming', 'John Smith', 'Spring 2005')
);
```

Creating a Simple Query

The SELECT command is used for retrieving data. The syntax of SELECT is as follows:

```
SELECT column(s) (or '*' for all)
FROM table(s)
[WHERE constraint(s)];
```

An example of a simple query is:

```
mysql> SELECT * FROM Student;
+-------------+-----------------+----------------------------+---------+
| studentID | name           | email                      | major |
+-------------+-----------------+----------------------------+---------+
|123456789 | Bill Lindgren   | lindgren@cs.ucsb.edu |   CE    |
|987654321 | Rajiv Alur      | alur@ece.ucsb.edu    |   CS    |
|654321987 | Yin Yang        | yang@ece.ucsb.edu    |  ECE   |
+-------------+-----------------+----------------------------+---------+
```

The optional WHERE clause allows you to retrieve a limited data set. For example:

```
mysql> SELECT * FROM Course
    -> WHERE quarter = 'Spring 2005';
+------------------+-----------------------------------+---------------+---------------+
| courseNumber | name                              | professor     | quarter       |
```

| | | | |
|---|---|---|---|
| ECE 155B | Network Computing | Moser | Spring 2005 |
| ECE 154 | Computer Architecture | Butner | Spring 2005 |
| CS 174A | Database Systems | Su | Spring 2005 |

**Adapted from http://www.w3schools.com/sql/**

## SQL Database Tables

**Create a Database**
To create a database:

```
CREATE DATABASE database_name
```

**Create a Table**
To create a table in a database:

```
CREATE TABLE table_name
(
column_name1 data_type,
column_name2 data_type,
.......
)
```

Example
This example demonstrates how you can create a table named "Person", with four columns. The column names will be "LastName", "FirstName", "Address", and "Age":

```
CREATE TABLE Person
(
LastName varchar,
FirstName varchar,
Address varchar,
Age int
)
```

This example demonstrates how you can specify a maximum length for some columns:

```
CREATE TABLE Person
(
LastName varchar(30),
FirstName varchar,
Address varchar,
Age int(3)
)
```

The data type specifies what type of data the column can hold. The table below contains the most common data types in SQL:

| Data Type | Description |
|---|---|
| integer(size)<br>int (size)<br>smallint (size)<br>tinyint (size) | Hold integers only. The maximum number of digits is specified in parenthesis. |
| decimal(size, d)<br>numeric(size, d) | Hold numbers with fractions. The maximum number of digits is specified in "size". The maximum number of digits to the right of the decimal is specified in "d". |
| char(size) | Holds a fixed length string (can contain letters, numbers, and special characters). The fixed size is specified in parenthesis. |

| | |
|---|---|
| Varchar (size) | Holds a variable length string (can contain letters, numbers, and special characters). The maximum size is specified in parenthesis. |
| date(yyyymmdd) | Holds a date |

**Delete a Table or Database**
To delete a table (the table structure, attributes, and indexes will also be deleted):

```
DROP TABLE table_name
```

**To delete a database:**

```
DROP DATABASE database_name
```

**Truncate a Table**

What if we only want to get rid of the data inside a table, and not the table itself? Use the TRUNCATE TABLE command (deletes only the data inside the table):

```
TRUNCATE TABLE table_name
```

**ALTER TABLE**
The ALTER TABLE statement is used to add or drop columns in an existing table.

```
ALTER TABLE table_name
ADD column_name datatype
ALTER TABLE table_name
DROP COLUMN column_name
```

Note: Some database systems don't allow the dropping of a column in a database table (DROP COLUMN column_name).

---

Person:

| LastName | FirstName | Address |
|---|---|---|
| Pettersen | Kari | Storgt 20 |

**Example**
To add a column named "City" in the "Person" table:

```
ALTER TABLE Person ADD City varchar(30)
```

Result:

| LastName | FirstName | Address | City |
|---|---|---|---|
| Pettersen | Kari | Storgt 20 | |

Example
To drop the "Address" column in the "Person" table:

```
ALTER TABLE Person DROP COLUMN Address
```

Result:

| LastName | FirstName | City |
|---|---|---|
| Pettersen | Kari | |

**SQL Queries**
With SQL, we can query a database and have a result set returned.

A query like this:

```
SELECT LastName FROM Persons
```

Gives a result set like this:

| LastName |
|---|
| Hansen |
| Svendson |
| Pettersen |

**SQL Data Manipulation Language (DML)**
SQL (Structured Query Language) is syntax for executing queries. But the SQL language also includes syntax to update, insert, and delete records.
These query and update commands together form the Data Manipulation Language (DML) part of SQL:

*SELECT - extracts data from a database table*
*UPDATE - updates data in a database table*
*DELETE - deletes data from a database table*
*INSERT INTO - inserts new data into a database table*

**SQL Data Definition Language (DDL)**
The Data Definition Language (DDL) part of SQL permits database tables to be created or deleted. We can also define indexes (keys), specify links between tables, and impose constraints between database tables.

The most important DDL statements in SQL are:

*CREATE TABLE - creates a new database table*
*ALTER TABLE - alters (changes) a database table*
*DROP TABLE - deletes a database table*
*CREATE INDEX - creates an index (search key)*
*DROP INDEX - deletes an index*

**The SELECT Statement**
The SELECT statement is used to select data from a table. The tabular result is stored in a result table (called the result-set).
Syntax

```
SELECT column_name(s)
FROM table_name
```

**Select Some Columns**
To select the columns named "LastName" and "FirstName", use a SELECT statement like this:

```
SELECT LastName,FirstName FROM Persons
```

"Persons" table

| LastName | FirstName | Address | City |
|---|---|---|---|
| Hansen | Ola | Timoteivn 10 | Sandnes |
| Svendson | Tove | Borgvn 23 | Sandnes |
| Pettersen | Kari | Storgt 20 | Stavanger |

Result

| LastName | FirstName |
|---|---|
| Hansen | Ola |
| Svendson | Tove |
| Pettersen | Kari |

**Select All Columns**

To select all columns from the "Persons" table, use a * symbol instead of column names, like this:

```
SELECT * FROM Persons
```

Result

| LastName | FirstName | Address | City |
|---|---|---|---|
| Hansen | Ola | Timoteivn 10 | Sandnes |
| Svendson | Tove | Borgvn 23 | Sandnes |
| Pettersen | Kari | Storgt 20 | Stavanger |

**The Result Set**
The result from a SQL query is stored in a result-set. Most database software systems allow navigation of the result set with programming functions, like: Move-To-First-Record, Get-Record-Content, Move-To-Next-Record, etc.
Programming functions like these are not a part of this tutorial. To learn about accessing data with function calls, please visit our ADO tutorial.

**Semicolon after SQL Statements?**
Semicolon is the standard way to separate each SQL statement in database systems that allow more than one SQL statement to be executed in the same call to the server.
Some SQL tutorials end each SQL statement with a semicolon. Is this necessary? We are using MS Access and SQL Server 2000 and we do not have to put a semicolon after each SQL statement, but some database programs force you to use it.

**The SELECT DISTINCT Statement**
The DISTINCT keyword is used to return only distinct (different) values.
The SELECT statement returns information from table columns. But what if we only want to select distinct elements?
With SQL, all we need to do is to add a DISTINCT keyword to the SELECT statement:
Syntax

```
SELECT DISTINCT column_name(s)
FROM table_name
```

Using the DISTINCT keyword
To select ALL values from the column named "Company" we use a SELECT statement like this:

```
SELECT Company FROM Orders
```

"Orders" table

| Company | OrderNumber |
|---|---|
| Sega | 3412 |
| W3Schools | 2312 |
| Trio | 4678 |
| W3Schools | 6798 |

Result

| Company |
|---|
| Sega |
| W3Schools |
| Trio |
| W3Schools |

Note that "W3Schools" is listed twice in the result-set.
To select only DIFFERENT values from the column named "Company" we use a SELECT DISTINCT statement like this:

```
SELECT DISTINCT Company FROM Orders
```

Result:

| Company |
|---|
| Sega |
| W3Schools |
| Trio |

Now "W3Schools" is listed only once in the result-set.
The WHERE clause is used to specify a selection criterion.

**The WHERE Clause**

To conditionally select data from a table, a WHERE clause can be added to the SELECT statement.
Syntax

```
SELECT column FROM table
WHERE column operator value
```

With the WHERE clause, the following operators can be used:

| Operator | Description |
|---|---|
| = | Equal |
| <> | Not equal |
| > | Greater than |
| < | Less than |
| >= | Greater than or equal |
| <= | Less than or equal |
| BETWEEN | Between an inclusive range |
| LIKE | Search for a pattern |

Note: In some versions of SQL the <> operator may be written as !=

**Using the WHERE Clause**
To select only the persons living in the city "Sandnes", we add a WHERE clause to the SELECT statement:

```
SELECT * FROM Persons
WHERE City='Sandnes'
```

"Persons" table

| LastName | FirstName | Address | City | Year |
|---|---|---|---|---|
| Hansen | Ola | Timoteivn 10 | Sandnes | 1951 |
| Svendson | Tove | Borgvn 23 | Sandnes | 1978 |
| Svendson | Stale | Kaivn 18 | Sandnes | 1980 |
| Pettersen | Kari | Storgt 20 | Stavanger | 1960 |

Result

| LastName | FirstName | Address | City | Year |
|---|---|---|---|---|
| Hansen | Ola | Timoteivn 10 | Sandnes | 1951 |
| Svendson | Tove | Borgvn 23 | Sandnes | 1978 |
| Svendson | Stale | Kaivn 18 | Sandnes | 1980 |

**Using Quotes**
Note that we have used single quotes around the conditional values in the examples.
SQL uses single quotes around text values (most database systems will also accept double quotes).
Numeric values should not be enclosed in quotes.

For text values:

```
This is correct:
SELECT * FROM Persons WHERE FirstName='Tove'
This is wrong:
SELECT * FROM Persons WHERE FirstName=Tove
```

For numeric values:

```
This is correct:
SELECT * FROM Persons WHERE Year>1965
This is wrong:
SELECT * FROM Persons WHERE Year>'1965'
```

**The LIKE Condition**
The LIKE condition is used to specify a search for a pattern in a column.
Syntax

```
SELECT column FROM table
WHERE column LIKE pattern
```

A "%" sign can be used to define wildcards (missing letters in the pattern) both before and after the pattern.

**Using LIKE**
The following SQL statement will return persons with first names that start with an 'O':

```
SELECT * FROM Persons
WHERE FirstName LIKE 'O%'
```

The following SQL statement will return persons with first names that end with an 'a':

```
SELECT * FROM Persons
WHERE FirstName LIKE '%a'
```

The following SQL statement will return persons with first names that contain the pattern 'la':

```
SELECT * FROM Persons
WHERE FirstName LIKE '%la%'
```

**The INSERT INTO Statement**
The INSERT INTO statement is used to insert new rows into a table.
Syntax

```
INSERT INTO table_name
VALUES (value1, value2,....)
```

You can also specify the columns for which you want to insert data:

```
INSERT INTO table_name (column1, column2,...)
VALUES (value1, value2,....)
```

**Insert a New Row**
This "Persons" table:

| LastName | FirstName | Address | City |
|---|---|---|---|
| Pettersen | Kari | Storgt 20 | Stavanger |

And this SQL statement:

```
INSERT INTO Persons
VALUES ('Hetland', 'Camilla', 'Hagabakka 24', 'Sandnes')
```

Will give this result:

| LastName | FirstName | Address | City |
|---|---|---|---|
| Pettersen | Kari | Storgt 20 | Stavanger |
| Hetland | Camilla | Hagabakka 24 | Sandnes |

**Insert Data in Specified Columns**
This "Persons" table:

| LastName | FirstName | Address | City |
|---|---|---|---|
| Pettersen | Kari | Storgt 20 | Stavanger |
| Hetland | Camilla | Hagabakka 24 | Sandnes |

And This SQL statement:

```
INSERT INTO Persons (LastName, Address)
VALUES ('Rasmussen', 'Storgt 67')
```

Will give this result:

| LastName | FirstName | Address | City |
|---|---|---|---|
| Pettersen | Kari | Storgt 20 | Stavanger |
| Hetland | Camilla | Hagabakka 24 | Sandnes |
| Rasmussen | | Storgt 67 | |

**The Update Statement**
The UPDATE statement is used to modify the data in a table.
Syntax

```
UPDATE table_name
SET column_name = new_value
WHERE column_name = some_value
```

Person:

| LastName | FirstName | Address | City |
|---|---|---|---|
| Nilsen | Fred | Kirkegt 56 | Stavanger |
| Rasmussen | | Storgt 67 | |

**Update one Column in a Row**
We want to add a first name to the person with a last name of "Rasmussen":

```
UPDATE Person SET FirstName = 'Nina'
WHERE LastName = 'Rasmussen'
```

Result:

| LastName | FirstName | Address | City |
|---|---|---|---|
| Nilsen | Fred | Kirkegt 56 | Stavanger |
| Rasmussen | Nina | Storgt 67 | |

**Update several Columns in a Row**
We want to change the address and add the name of the city:

```
UPDATE Person
SET Address = 'Stien 12', City = 'Stavanger'
WHERE LastName = 'Rasmussen'
```

Result:

| LastName | FirstName | Address | City |
|---|---|---|---|
| Nilsen | Fred | Kirkegt 56 | Stavanger |
| Rasmussen | Nina | Stien 12 | Stavanger |

**The Delete Statement**
The DELETE statement is used to delete rows in a table.
Syntax

```
DELETE FROM table_name
WHERE column_name = some_value
```

Person:

| LastName | FirstName | Address | City |
|---|---|---|---|
| Nilsen | Fred | Kirkegt 56 | Stavanger |
| Rasmussen | Nina | Stien 12 | Stavanger |

**Delete a Row**

"Nina Rasmussen" is going to be deleted:

```
DELETE FROM Person WHERE LastName = 'Rasmussen'
```

Result

| LastName | FirstName | Address | City |
|---|---|---|---|
| Nilsen | Fred | Kirkegt 56 | Stavanger |

**Delete All Rows**

It is possible to delete all rows in a table without deleting the table. This means that the table structure, attributes, and indexes will be intact:

```
DELETE FROM table_name
or
DELETE * FROM table_name
```

**Test your SQL Skills**
On this page you can test your SQL skills.
We will use the Customers table in the Northwind database:

| CompanyName | ContactName | Address | City |
|---|---|---|---|
| Alfreds Futterkiste | Maria Anders | Obere Str. 57 | Berlin |
| Berglunds snabbköp | Christina Berglund | Berguvsvägen 8 | Luleå |
| Centro comercial Moctezuma | Francisco Chang | Sierras de Granada 9993 | México D.F. |
| Ernst Handel | Roland Mendel | Kirchgasse 6 | Graz |
| FISSA Fabrica Inter. Salchichas S.A. | Diego Roel | C/ Moralzarzal, 86 | Madrid |
| Galería del gastrónomo | Eduardo Saavedra | Rambla de Cataluña, 23 | Barcelona |
| Island Trading | Helen Bennett | Garden House Crowther Way | Cowes |
| Königlich Essen | Philip Cramer | Maubelstr. 90 | Brandenburg |
| Laughing Bacchus Wine Cellars | Yoshi Tannamuri | 1900 Oak St. | Vancouver |
| Magazzini Alimentari Riuniti | Giovanni Rovelli | Via Ludovico il Moro 22 | Bergamo |
| North/South | Simon Crowther | South House 300 Queensbridge | London |
| Paris spécialités | Marie Bertrand | 265, boulevard Charonne | Paris |
| Rattlesnake Canyon Grocery | Paula Wilson | 2817 Milton Dr. | Albuquerque |
| Simons bistro | Jytte Petersen | Vinbæltet 34 | København |
| The Big Cheese | Liz Nixon | 89 Jefferson Way Suite 2 | Portland |
| Vaffeljernet | Palle Ibsen | Smagsløget 45 | Århus |
| Wolski Zajazd | Zbyszek Piestrzeniewicz | ul. Filtrowa 68 | Warszawa |

To preserve space, the table above is a subset of the Customers table used in the example below.

**Try it Yourself**
To see how SQL works, you can copy the SQL statements below and paste them into the textarea, or you can make your own SQL statements.

| SELECT * FROM customers | | | | | | |
|---|---|---|---|---|---|---|
| CustomerID | CompanyName | ContactName | Address | City | PostalCode | Country |
| ALFKI | Alfreds Futterkiste | Maria Anders | Obere Str. 57 | Berlin | 12209 | Germany |
| BERGS | Berglunds snabbköp | Christina Berglund | Berguvsvägen 8 | Luleå | S-958 22 | Sweden |
| CENTC | Centro comercial Moctezuma | Francisco Chang | Sierras de Granada 9993 | México D.F. | 05022 | Mexico |
| ERNSH | Ernst Handel | Roland Mendel | Kirchgasse 6 | Graz | 8010 | Austria |
| FISSA | FISSA Fabrica Inter. Salchichas S.A. | Diego Roel | C/ Moralzarzal, 86 | Madrid | 28034 | Spain |
| GALED | Galería del gastrónomo | Eduardo Saavedra | Rambla de Cataluña, 23 | Barcelona | 08022 | Spain |
| ISLAT | Island Trading | Helen Bennett | Garden House Crowther Way | Cowes | PO31 7PJ | UK |
| KOENE | Königlich Essen | Philip Cramer | Maubelstr. 90 | Brandenburg | 14776 | Germany |

| | | | | | | |
|---|---|---|---|---|---|---|
| LAUGB | Laughing Bacchus Wine Cellars | Yoshi Tannamuri | 1900 Oak St. | Vancouver | V3F 2K1 | Canada |
| MAGAA | Magazzini Alimentari Riuniti | Giovanni Rovelli | Via Ludovico il Moro 22 | Bergamo | 24100 | Italy |
| NORTS | North/South | Simon Crowther | South House 300 Queensbridge | London | SW7 1RZ | UK |
| PARIS | Paris spécialités | Marie Bertrand | 265, boulevard Charonne | Paris | 75012 | France |
| RATTC | Rattlesnake Canyon Grocery | Paula Wilson | 2817 Milton Dr. | Albuquerque | 87110 | USA |
| SIMOB | Simons bistro | Jytte Petersen | Vinbæltet 34 | København | 1734 | Denmark |
| THEBI | The Big Cheese | Liz Nixon | 89 Jefferson Way Suite 2 | Portland | 97201 | USA |
| VAFFE | Vaffeljernet | Palle Ibsen | Smagsløget 45 | Århus | 8200 | Denmark |
| WOLZA | Wolski Zajazd | Zbyszek Piestrzeniewicz | ul. Filtrowa 68 | Warszawa | 01-012 | Poland |

**SELECT CompanyName, ContactName FROM customers**

| CompanyName | ContactName |
|---|---|
| Alfreds Futterkiste | Maria Anders |
| Berglunds snabbköp | Christina Berglund |
| Centro comercial Moctezuma | Francisco Chang |
| Ernst Handel | Roland Mendel |
| FISSA Fabrica Inter. Salchichas S.A. | Diego Roel |
| Galería del gastrónomo | Eduardo Saavedra |
| Island Trading | Helen Bennett |
| Königlich Essen | Philip Cramer |
| Laughing Bacchus Wine Cellars | Yoshi Tannamuri |
| Magazzini Alimentari Riuniti | Giovanni Rovelli |
| North/South | Simon Crowther |
| Paris spécialités | Marie Bertrand |
| Rattlesnake Canyon Grocery | Paula Wilson |
| Simons bistro | Jytte Petersen |
| The Big Cheese | Liz Nixon |
| Vaffeljernet | Palle Ibsen |
| Wolski Zajazd | Zbyszek Piestrzeniewicz |

**SELECT * FROM customers WHERE companyname LIKE 'a%'**

| CustomerID | CompanyName | ContactName | Address | City | PostalCode | Country |
|---|---|---|---|---|---|---|
| ALFKI | Alfreds Futterkiste | Maria Anders | Obere Str. 57 | Berlin | 12209 | Germany |

```
SELECT CompanyName, ContactName
FROM customers
WHERE CompanyName > 'g'
AND ContactName > 'g'
```

| CompanyName | ContactName |
|---|---|

| | |
|---|---|
| Island Trading | Helen Bennett |
| Königlich Essen | Philip Cramer |
| Laughing Bacchus Wine Cellars | Yoshi Tannamuri |
| Magazzini Alimentari Riuniti | Giovanni Rovelli |
| North/South | Simon Crowther |
| Paris spécialités | Marie Bertrand |
| Rattlesnake Canyon Grocery | Paula Wilson |
| Simons bistro | Jytte Petersen |
| The Big Cheese | Liz Nixon |
| Vaffeljernet | Palle Ibsen |
| Wolski Zajazd | Zbyszek Piestrzeniewicz |

**The ORDER BY keyword is used to sort the result.**

---

Sort the Rows
The ORDER BY clause is used to sort the rows.
Orders:

| Company | OrderNumber |
|---|---|
| Sega | 3412 |
| ABC Shop | 5678 |
| W3Schools | 2312 |
| W3Schools | 6798 |

Example
To display the companies in alphabetical order:

```
SELECT Company, OrderNumber FROM Orders
ORDER BY Company
```

Result:

| Company | OrderNumber |
|---|---|
| ABC Shop | 5678 |
| Sega | 3412 |
| W3Schools | 6798 |
| W3Schools | 2312 |

Example
To display the companies in alphabetical order AND the order numbers in numerical order:

```
SELECT Company, OrderNumber FROM Orders
ORDER BY Company, OrderNumber
```

Result:

| Company | OrderNumber |
|---|---|
| ABC Shop | 5678 |

| | |
|---|---|
| Sega | 3412 |
| W3Schools | 2312 |
| W3Schools | 6798 |

Example
To display the companies in reverse alphabetical order:

```
SELECT Company, OrderNumber FROM Orders
ORDER BY Company DESC
```

Result:

| Company | OrderNumber |
|---|---|
| W3Schools | 6798 |
| W3Schools | 2312 |
| Sega | 3412 |
| ABC Shop | 5678 |

Example
To display the companies in reverse alphabetical order AND the ordernumbers in numerical order:

```
SELECT Company, OrderNumber FROM Orders
ORDER BY Company DESC, OrderNumber ASC
```

Result:

| Company | OrderNumber |
|---|---|
| W3Schools | 2312 |
| W3Schools | 6798 |
| Sega | 3412 |
| ABC Shop | 5678 |

**AND & OR**
AND and OR join two or more conditions in a WHERE clause.
The AND operator displays a row if ALL conditions listed are true. The OR operator displays a row if ANY of the conditions listed are true.

Original Table (used in the examples)

| LastName | FirstName | Address | City |
|---|---|---|---|
| Hansen | Ola | Timoteivn 10 | Sandnes |
| Svendson | Tove | Borgvn 23 | Sandnes |
| Svendson | Stephen | Kaivn 18 | Sandnes |

Example
Use AND to display each person with the first name equal to "Tove", and the last name equal to "Svendson":

```
SELECT * FROM Persons
WHERE FirstName='Tove'
AND LastName='Svendson'
```

Result:

| LastName | FirstName | Address | City |
|---|---|---|---|
| Svendson | Tove | Borgvn 23 | Sandnes |

Example
Use OR to display each person with the first name equal to "Tove", or the last name equal to "Svendson":

```
SELECT * FROM Persons
WHERE firstname='Tove'
OR lastname='Svendson'
```

Result:

| LastName | FirstName | Address | City |
|---|---|---|---|
| Svendson | Tove | Borgvn 23 | Sandnes |
| Svendson | Stephen | Kaivn 18 | Sandnes |

Example
You can also combine AND and OR (use parentheses to form complex expressions):

```
SELECT * FROM Persons WHERE
(FirstName='Tove' OR FirstName='Stephen')
AND LastName='Svendson'
```

Result:

| LastName | FirstName | Address | City |
|---|---|---|---|
| Svendson | Tove | Borgvn 23 | Sandnes |
| Svendson | Stephen | Kaivn 18 | Sandnes |

**IN**
The IN operator may be used if you know the exact value you want to return for at least one of the columns.

```
SELECT column_name FROM table_name
WHERE column_name IN (value1,value2,..)
```

Original Table (used in the examples)

| LastName | FirstName | Address | City |
|---|---|---|---|
| Hansen | Ola | Timoteivn 10 | Sandnes |
| Nordmann | Anna | Neset 18 | Sandnes |
| Pettersen | Kari | Storgt 20 | Stavanger |
| Svendson | Tove | Borgvn 23 | Sandnes |

Example 1
To display the persons with LastName equal to "Hansen" or "Pettersen", use the following SQL:

```
SELECT * FROM Persons
WHERE LastName IN ('Hansen','Pettersen')
```

Result:

| LastName | FirstName | Address | City |
|---|---|---|---|
| Hansen | Ola | Timoteivn 10 | Sandnes |
| Pettersen | Kari | Storgt 20 | Stavanger |

**BETWEEN ... AND**
The BETWEEN ... AND operator selects a range of data between two values. These values can be numbers, text, or dates.

```
SELECT column_name FROM table_name
WHERE column_name
BETWEEN value1 AND value2
```

Original Table (used in the examples)

| LastName | FirstName | Address | City |
|---|---|---|---|
| Hansen | Ola | Timoteivn 10 | Sandnes |
| Nordmann | Anna | Neset 18 | Sandnes |
| Pettersen | Kari | Storgt 20 | Stavanger |
| Svendson | Tove | Borgvn 23 | Sandnes |

Example 1
To display the persons alphabetically between (and including) "Hansen" and exclusive "Pettersen", use the following SQL:

```
SELECT * FROM Persons WHERE LastName
BETWEEN 'Hansen' AND 'Pettersen'
```

Result:

| LastName | FirstName | Address | City |
|---|---|---|---|
| Hansen | Ola | Timoteivn 10 | Sandnes |
| Nordmann | Anna | Neset 18 | Sandnes |

IMPORTANT! The BETWEEN...AND operator is treated differently in different databases. With some databases a person with the LastName of "Hansen" or "Pettersen" will not be listed (BETWEEN..AND only selects fields that are between and excluding the test values). With some databases a person with the last name of "Hansen" or "Pettersen" will be listed (BETWEEN..AND selects fields that are between and including the test values). With other databases a person with the last name of "Hansen" will be listed, but "Pettersen" will not be listed (BETWEEN..AND selects fields between the test values, including the first test value and excluding the last test value). Therefore: Check how your database treats the BETWEEN....AND operator!

Example 2
To display the persons outside the range used in the previous example, use the NOT operator:

```
SELECT * FROM Persons WHERE LastName
NOT BETWEEN 'Hansen' AND 'Pettersen'
```

Result:

| LastName | FirstName | Address | City |
|---|---|---|---|
| Pettersen | Kari | Storgt 20 | Stavanger |
| Svendson | Tove | Borgvn 23 | Sandnes |

With SQL, aliases can be used for column names and table names.

**Alias**
**Column Name Alias**
The syntax is:

```
SELECT column AS column_alias FROM table
```

**Table Name Alias**
The syntax is:

```
SELECT column FROM table AS table_alias
```

**Example: Using a Column Alias**
This table (Persons):

| LastName | FirstName | Address | City |
|---|---|---|---|
| Hansen | Ola | Timoteivn 10 | Sandnes |
| Svendson | Tove | Borgvn 23 | Sandnes |
| Pettersen | Kari | Storgt 20 | Stavanger |

And this SQL:

```
SELECT LastName AS Family, FirstName AS Name
FROM Persons
```

Returns this result:

| Family | Name |
|---|---|
| Hansen | Ola |
| Svendson | Tove |
| Pettersen | Kari |

Example: Using a Table Alias
This table (Persons):

| LastName | FirstName | Address | City |
|---|---|---|---|
| Hansen | Ola | Timoteivn 10 | Sandnes |
| Svendson | Tove | Borgvn 23 | Sandnes |
| Pettersen | Kari | Storgt 20 | Stavanger |

And this SQL:

```
SELECT LastName, FirstName
FROM Persons AS Employees
```

Returns this result:
Table Employees:

| LastName | FirstName |
|---|---|
| Hansen | Ola |
| Svendson | Tove |
| Pettersen | Kari |

**Joins and Keys**

Sometimes we have to select data from two or more tables to make our result complete. We have to perform a join.
Tables in a database can be related to each other with keys. A primary key is a column with a unique value for each row. The purpose is to bind data together, across tables, without repeating all of the data in every table.
In the "Employees" table below, the "Employee_ID" column is the primary key, meaning that no two rows can have the same Employee_ID. The Employee_ID distinguishes two persons even if they have the same name.
When you look at the example tables below, notice that:
The "Employee_ID" column is the primary key of the "Employees" table
The "Prod_ID" column is the primary key of the "Orders" table
The "Employee_ID" column in the "Orders" table is used to refer to the persons in the "Employees" table without using their names

Employees:

| Employee_ID | Name |
|---|---|
| 01 | Hansen, Ola |
| 02 | Svendson, Tove |
| 03 | Svendson, Stephen |
| 04 | Pettersen, Kari |

Orders:

| Prod_ID | Product | Employee_ID |
|---|---|---|
| 234 | Printer | 01 |
| 657 | Table | 03 |
| 865 | Chair | 03 |

**Referring to Two Tables**

We can select data from two tables by referring to two tables, like this:
Example
Who has ordered a product, and what did they order?

```
SELECT Employees.Name, Orders.Product
FROM Employees, Orders
WHERE Employees.Employee_ID=Orders.Employee_ID
```

Result

| Name | Product |
|---|---|
| Hansen, Ola | Printer |
| Svendson, Stephen | Table |
| Svendson, Stephen | Chair |

Example
Who ordered a printer?

```
SELECT Employees.Name
FROM Employees, Orders
WHERE Employees.Employee_ID=Orders.Employee_ID
AND Orders.Product='Printer'
```

Result

| Name |
|---|
| Hansen, Ola |

**Using Joins**

OR we can select data from two tables with the JOIN keyword, like this:
Example INNER JOIN
Syntax

```
SELECT field1, field2, field3
FROM first_table
INNER JOIN second_table
ON first_table.keyfield = second_table.foreign_keyfield
```

Who has ordered a product, and what did they order?

```
SELECT Employees.Name, Orders.Product
FROM Employees
INNER JOIN Orders
ON Employees.Employee_ID=Orders.Employee_ID
```

The INNER JOIN returns all rows from both tables where there is a match. If there are rows in Employees that do not have matches in Orders, those rows will not be listed.
Result

| Name | Product |
|---|---|

| | |
|---|---|
| Hansen, Ola | Printer |
| Svendson, Stephen | Table |
| Svendson, Stephen | Chair |

Example LEFT JOIN
Syntax

```
SELECT field1, field2, field3
FROM first_table
LEFT JOIN second_table
ON first_table.keyfield = second_table.foreign_keyfield
```

List all employees, and their orders - if any.

```
SELECT Employees.Name, Orders.Product
FROM Employees
LEFT JOIN Orders
ON Employees.Employee_ID=Orders.Employee_ID
```

The LEFT JOIN returns all the rows from the first table (Employees), even if there are no matches in the second table (Orders). If there are rows in Employees that do not have matches in Orders, those rows also will be listed.
Result

| Name | Product |
|---|---|
| Hansen, Ola | Printer |
| Svendson, Tove | |
| Svendson, Stephen | Table |
| Svendson, Stephen | Chair |
| Pettersen, Kari | |

Example RIGHT JOIN
Syntax

```
SELECT field1, field2, field3
FROM first_table
RIGHT JOIN second_table
ON first_table.keyfield = second_table.foreign_keyfield
```

List all orders, and who has ordered - if any.

```
SELECT Employees.Name, Orders.Product
FROM Employees
RIGHT JOIN Orders
ON Employees.Employee_ID=Orders.Employee_ID
```

The RIGHT JOIN returns all the rows from the second table (Orders), even if there are no matches in the first table (Employees). If there had been any rows in Orders that did not have matches in Employees, those rows also would have been listed.
Result

| Name | Product |
|---|---|

| | |
|---|---|
| Hansen, Ola | Printer |
| Svendson, Stephen | Table |
| Svendson, Stephen | Chair |

Example
Who ordered a printer?

```
SELECT Employees.Name
FROM Employees
INNER JOIN Orders
ON Employees.Employee_ID=Orders.Employee_ID
WHERE Orders.Product = 'Printer'
```

Result

| Name |
|---|
| Hansen, Ola |

**UNION**
The UNION command is used to select related information from two tables, much like the JOIN command. However, when using the UNION command all selected columns need to be of the same data type.
Note: With UNION, only distinct values are selected.

```
SQL Statement 1
UNION
SQL Statement 2
```

Employees_Norway:

| Employee_ID | E_Name |
|---|---|
| 01 | Hansen, Ola |
| 02 | Svendson, Tove |
| 03 | Svendson, Stephen |
| 04 | Pettersen, Kari |

Employees_USA:

| Employee_ID | E_Name |
|---|---|
| 01 | Turner, Sally |
| 02 | Kent, Clark |
| 03 | Svendson, Stephen |
| 04 | Scott, Stephen |

**Using the UNION Command**

Example
List all different employee names in Norway and USA:

```
SELECT E_Name FROM Employees_Norway
UNION
SELECT E_Name FROM Employees_USA
```

Result

| Name |
|---|
| Hansen, Ola |
| Svendson, Tove |
| Svendson, Stephen |
| Pettersen, Kari |
| Turner, Sally |
| Kent, Clark |
| Scott, Stephen |

Note: This command cannot be used to list all employees in Norway and USA. In the example above we have two employees with equal names, and only one of them is listed. The UNION command only selects distinct values.

**UNION ALL**
The UNION ALL command is equal to the UNION command, except that UNION ALL selects all values.

```
SQL Statement 1
UNION ALL
SQL Statement 2
```

**Using the UNION ALL Command**
Example
List all employees in Norway and USA:

```
SELECT E_Name FROM Employees_Norway
UNION ALL
SELECT E_Name FROM Employees_USA
```

Result

| Name |
|---|
| Hansen, Ola |
| Svendson, Tove |
| Svendson, Stephen |
| Pettersen, Kari |
| Turner, Sally |
| Kent, Clark |
| Svendson, Stephen |
| Scott, Stephen |

**Function Syntax**
The syntax for built-in SQL functions is:

```
SELECT function(column) FROM table
```

Types of Functions
There are several basic types and categories of functions in SQL. The basic types of functions are:
Aggregate Functions
Scalar functions

---

**Aggregate functions**
Aggregate functions operate against a collection of values, but return a single value.
Note: If used among many other expressions in the item list of a SELECT statement, the SELECT must have a GROUP BY clause!!
"Persons" table (used in most examples)

| Name | Age |
|---|---|
| Hansen, Ola | 34 |
| Svendson, Tove | 45 |
| Pettersen, Kari | 19 |

Aggregate functions in MS Access

| Function | Description |
|---|---|
| AVG(column) | Returns the average value of a column |
| COUNT(column) | Returns the number of rows (without a NULL value) of a column |
| COUNT(*) | Returns the number of selected rows |
| FIRST(column) | Returns the value of the first record in the specified field |
| LAST(column) | Returns the value of the last record in the specified field |
| MAX(column) | Returns the highest value of a column |
| MIN(column) | Returns the lowest value of a column |
| STDEV(column) | |
| STDEVP(column) | |
| SUM(column) | Returns the total sum of a column |
| VAR(column) | |
| VARP(column) | |

Aggregate functions in SQL Server

| Function | Description |
|---|---|
| AVG(column) | Returns the average value of a column |
| BINARY_CHECKSUM | |
| CHECKSUM | |
| CHECKSUM_AGG | |
| COUNT(column) | Returns the number of rows (without a NULL value) of a column |
| COUNT(*) | Returns the number of selected rows |
| COUNT(DISTINCT column) | Returns the number of distinct results |
| FIRST(column) | Returns the value of the first record in the specified field (not supported in SQLServer2K) |
| LAST(column) | Returns the value of the last record in the specified field (not |

| | |
|---|---|
| | supported in SQLServer2K) |
| MAX(column) | Returns the highest value of a column |
| MIN(column) | Returns the lowest value of a column |
| STDEV(column) | |
| STDEVP(column) | |
| SUM(column) | Returns the total sum of a column |
| VAR(column) | |
| VARP(column) | |

**Scalar functions**
Scalar functions operate against a single value, and return a single value based on the input value.
Useful Scalar Functions in MS Access

| Function | Description |
|---|---|
| UCASE(c) | Converts a field to upper case |
| LCASE(c) | Converts a field to lower case |
| MID(c,start[,end]) | Extract characters from a text field |
| LEN(c) | Returns the length of a text field |
| INSTR(c) | Returns the numeric position of a named character within a text field |
| LEFT(c,number_of_char) | Return the left part of a text field requested |
| RIGHT(c,number_of_char) | Return the right part of a text field requested |
| ROUND(c,decimals) | Rounds a numeric field to the number of decimals specified |
| MOD(x,y) | Returns the remainder of a division operation |
| NOW() | Returns the current system date |
| FORMAT(c,format) | Changes the way a field is displayed |
| DATEDIFF(d,date1,date2) | Used to perform date calculations |

Aggregate functions (like SUM) often need an added GROUP BY functionality.

**GROUP BY...**
GROUP BY... was added to SQL because aggregate functions (like SUM) return the aggregate of all column values every time they are called, and without the GROUP BY function it was impossible to find the sum for each individual group of column values.
The syntax for the GROUP BY function is:

```
SELECT column,SUM(column) FROM table GROUP BY column
```

GROUP BY Example
This "Sales" Table:

| Company | Amount |
|---|---|
| W3Schools | 5500 |
| IBM | 4500 |
| W3Schools | 7100 |

And This SQL:

```
SELECT Company, SUM(Amount) FROM Sales
```

Returns this result:

| Company | SUM(Amount) |
|---|---|
| W3Schools | 17100 |
| IBM | 17100 |
| W3Schools | 17100 |

The above code is invalid because the column returned is not part of an aggregate. A GROUP BY clause will solve this problem:

```
SELECT Company,SUM(Amount) FROM Sales
GROUP BY Company
```

Returns this result:

| Company | SUM(Amount) |
|---|---|
| W3Schools | 12600 |
| IBM | 4500 |

**HAVING...**

HAVING... was added to SQL because the WHERE keyword could not be used against aggregate functions (like SUM), and without HAVING... it would be impossible to test for result conditions.
The syntax for the HAVING function is:

```
SELECT column,SUM(column) FROM table
GROUP BY column
HAVING SUM(column) condition value
```

This "Sales" Table:

| Company | Amount |
|---|---|
| W3Schools | 5500 |
| IBM | 4500 |
| W3Schools | 7100 |

This SQL:

```
SELECT Company,SUM(Amount) FROM Sales
GROUP BY Company
HAVING SUM(Amount)>10000
```

Returns this result

| Company | SUM(Amount) |
|---|---|
| W3Schools | 12600 |

A database most often contains one or more tables. Each table is identified by a name (e.g. "Customers" or "Orders"). Tables contain records (rows) with data.
Below is an example of a table called "Persons":

| LastName | FirstName | Address | City |
|---|---|---|---|
| Hansen | Ola | Timoteivn 10 | Sandnes |
| Svendson | Tove | Borgvn 23 | Sandnes |
| Pettersen | Kari | Storgt 20 | Stavanger |

The table above contains three records (one for each person) and four columns (LastName, FirstName, Address, and City).