Chapter 3: Introduction to SQL

Database System Concepts, 6th Ed.

Chapter 3: Introduction to SQL

- Overview of the SQL Query Language
- Data Definition
- Basic Query Structure
- Additional Basic Operations
- Set Operations
- Null Values
- Aggregate Functions
- Nested Subqueries
- Modification of the Database

History

- IBM Sequel language developed as part of System R project at the IBM San Jose Research Laboratory
- Renamed Structured Query Language (SQL)
- Standard SQL:
  - SQL-86, SQL-89, SQL-92
  - SQL:1999, SQL:2003, SQL:2008
- Commercial systems offer most, if not all, SQL-92 features, plus varying feature sets from later standards and special proprietary features
  - Not all examples here may work on your particular system.
  - Some interfaces require each statement to end with a semicolon
  - In some interfaces, SQL is not case-sensitive

SQL

- **Parts**
  - DDL
  - DML
  - DCL

Data Definition Language

The SQL **Data Definition language (DDL)** allows the specification of information about relations, including:

- The schema for each relation
- The domain of values associated with each attribute
- Integrity constraints
- And as we will see later, also other information such as
  - The set of indices to be maintained for each relations
  - Security and authorization information for relation by access rights
  - The physical storage structure of each relation on disk
- Example: CREATE, ALTER, DROP, RENAME, and TRUNCATE

Data Manipulation Language

The SQL **Data Manipulation language (DML)** provides the ability

- To query information from the database
- To insert tuples into
- To delete tuples from
- To modify tuples in the database

- Example: SELECT, INSERT, UPDATE, and DELETE.

Data Control Language

The SQL **Data Control language (DCL)** used for providing security to database objects.

- Example: GRANT and REVOKE.

Domain Types in SQL

- **char(n).** Fixed length character string, with user-specified length *n*
- **varchar(n).** Variable length character strings, with user-specified maximum length *n*
- **int.** Integer (a finite subset of the integers that is machine-dependent)
- **smallint.** Small integer (a machine-dependent subset of the integer domain type)
- **numeric(p,d).** Fixed point number, with user-specified precision of *p* digits, with *n* digits to the right of decimal point
- **real, double precision.** Floating point and double-precision floating point numbers, with machine-dependent precision
- **float(n).** Floating point number, with user-specified precision of at least *n* digits
- Date, Boolean, etc. more are covered in Chapter 4

Special Value - NULL

- Each type may include a special value called the **null value**
- **A null value** indicates an absent value that may exist but be unknown or that may not exist at all.
- In certain cases, we may wish to prohibit null values from being entered for integrity constraints….

Char vs Varchar

- **char data type** stores fixed length strings.
  - For example, an attribute *A of type char(10). If we store a string "Avi" in this attribute, 7 spaces* are appended to the string to make it 10 characters long.
- **varchar data type**
  - if store **"Avi" in attribute *B, no spaces would be*** added.
- When comparing two values of type **char, if they are of different lengths** extra spaces are automatically added to the shorter one to make them the same size, before comparison.
- When comparing a **char type with a varchar type, one may expect extra spaces** to be added to the **varchar type to make the lengths equal, before comparison;**
  - may or may not to be done, depends on the database system.

Create Table Construct

- An SQL relation is defined using the **create table** command:

  create table r (A_1 D_1, A_2 D_2, ..., A_n D_n,
  (integrity-constraint$_1$),
  ...,
  (integrity-constraint$_k$));

  - r is the name of the relation
  - each A_i is an attribute name in the schema of relation r
  - D_i is the data type of values in the domain of attribute A_i

Create Table and Insert

- **Example:** Create table department having department name, building information and budget.
- **create table** ***department***

  (*dept name* ***varchar (20),***

  building ***varchar (15),***

  budget ***numeric (12,2),***

  );

Create Table

- Example: Create Table instructor having ID, Name, Salary and department to which he belongs
- **create table** *instructor* (

```
              ID            char(5),
              name          varchar(20),
              dept_name     varchar(20),
              salary        numeric(8,2));
```

- **insert into** *instructor* **values** (‘10211’, ’Smith’, ’Biology’, 66000);
- **insert into** *instructor* **values** (‘10211’, null, ’Biology’, 66000);

DM
L

- Check: **insert into** *instructor* **values** (‘’, null, ’XYZ’, 0);

Integrity Constraints in Create Table

- **not null**
- **primary key** (A_1, ..., A_n)
- **foreign key** (A_m, ..., A_n) **references** *r*

Integrity Constraints in Create Table

- **not null**

•The **not null constraint on an attribute specifies that the null value** is not allowed for that attribute

  •Excludes the null value from the domain of that attribute

  •For example, the **not null constraint on the *name attribute of the instructor relation ensures that*** the name of an instructor cannot be null.

  •The following SQL ensures that the "ID", "LastName", and "FirstName" columns will NOT accept NULL values when the "Persons" table is created:

```
•CREATE TABLE Persons (
    ID int NOT NULL,
    LastName varchar(255) NOT NULL,
    FirstName varchar(255) NOT NULL,
    Age int
);
```

Not NULL on ALTER Table:

To create a NOT NULL constraint on the "Age" column when the "Persons" table is already created, use the following SQL:

```
ALTER TABLE Persons
MODIFY Age int NOT NULL;
```

Integrity Constraints in Create Table

- **primary key** (A_1, ..., A_n)

- ***The primary-key specification says that attributes*** A_1, ..., A_n *form the primary key for the relation*
- *The primary key* attributes are required to be ***non null*** *and* ***unique***
  - *No tuple can have* a null value for a primary-key attribute, and
  - No two tuples in the relation can be equal on all the primary-key attributes
- Although the primary-key specification is optional, it is generally a good idea to specify a primary key for each relation
- **primary key** declaration on an attribute automatically ensures **not null**

Create Table and Insert

- **Example:** Create table department having department name, building information and budget.
- **create table** ***department***

  (*dept_name* ***varchar (20),***

  building ***varchar (15),***
  budget ***numeric (12,2),***

  Primary key (dept_name)*);*

Integrity Constraints in Create Table

- **foreign key** (A_m, ..., A_n) **references** *r*

- ***The foreign key specification says*** that the values of attributes (A_m, ..., A_n) *for any tuple in the relation* must correspond to values of the primary key attributes of some tuple in relation *s*

- The definition of the *course table has a declaration "**foreign key** (dept name) **references department**"*
  - ***This foreign-key declaration specifies that*** for each course tuple, the department name specified in the tuple must exist in the primary key attribute (*dept name) of the department relation.*
  - *Without* this constraint, it is possible for a course to specify a nonexistent department name.

Integrity Constraints in Create Table

- **not null**
- **primary key** (A_1, ..., A_n)
- **foreign key** (A_m, ..., A_n) **references** *r*

Example: Declare *ID* as the primary key for *instructor, name requires value and dept_name should belong to department table*

.

```
create table instructor (
                    ID          char(5),
                    name        varchar(20) not null,
                    dept_name   varchar(20),
                    salary      numeric(8,2),
                    primary key (ID),
                    foreign key (dept_name) references department)
```

primary key declaration on an attribute automatically ensures **not null**

Schema of University

1. ***department****(dept name, building, budget)*
2. ***course****(course id, title, dept name, credits)*
3. ***student****(ID, name, dept name, tot cred)*
4. ***classroom****(building, room number, capacity)*
5. ***instructor****(ID, name, dept name, salary)*
6. ***section****(course id, sec id, semester, year, building, room number, time slot id)*
7. ***teaches****(ID, course id, sec id, semester, year)*
8. ***takes****(ID, course id, sec id, semester, year, grade)*
9. ***advisor****(s ID, i ID)*
10. ***time slot****(time slot id, day, start time, end time)*
11. ***prereq****(course id, prereq id)*

And a Few More Relation Definitions

- create table *student* (
```
ID           varchar(5),
name         varchar(20) not null,
dept_name    varchar(20),
tot_cred     numeric(3,0),
primary key (ID),
foreign key (dept_name) references department) );
```
- create table *takes* (
```
ID           varchar(5),
course_id    varchar(8),
sec_id       varchar(8),
semester     varchar(6),
year         numeric(4,0),
grade        varchar(2),
primary key (ID, course_id, sec_id, semester, year),
foreign key (ID) references student,
foreign key (course_id, sec_id, semester, year) references section );
```
  - Note: *sec_id* can be dropped from primary key above, to ensure a student cannot be registered for two sections of the same course in the same semester

And more still

- **create table** *course* (

  ```
      course_id     varchar(8) primary key,
      title         varchar(50),
      dept_name     varchar(20),
      credits       numeric(2,0),
      foreign key (dept_name) references department) );
  ```

  - Primary key declaration can be combined with attribute declaration as shown above

Alter Table Construct

- **alter table**
  - To modify the structure of the table
  - **alter table** *r* **add** *A D*
    - 4 where *A* is the name of the attribute to be added to relation *r* and *D* is the domain of *A*

      ALTER TABLE *table_name*
      ADD *column_name datatype*;

      ALTER TABLE Customers
      ADD Email varchar(255);
    - 4 **All tuples in the relation are assigned *null* as the value for the new attribute**.
  - **alter table** *r* **drop** *A*
    - 4 where *A* is the name of an attribute of relation *r*
    - 4 Dropping of attributes **not supported by many databases**

```
ALTER TABLE table_name
DROP COLUMN column_name;
```

The following SQL deletes the "Email" column from the "Customers" table:

```
ALTER TABLE Customers
DROP COLUMN Email;
```

Drop Table Construct

- **drop table** *student*
  - Deletes the table and its structure
  - If dropped the table, cannot get it back and all the references to the table will not be valid

Truncate Table

- Removes all rows from the table, but the table structure and its columns, constraints, indexes and so on remain
- To remove the table definition with its data, one can use drop table
  - Truncate table <tablename>;

- Restrictions
  - Cannot use truncate on table that is referenced by a foreign key constraint

Rename Table

- Rename Table OLD_NAME to NEW_NAME
- E.g.
    - Rename Table Employees to New_Employees

DML

- Provides the ability to query information
  - Select
- Insert, delete and update tuples
  - Insert, Delete, Update

The Select Clause (Cont.)

- A typical SQL query has the form:

  select $A_1, A_2, ..., A_n$
  from $r_1, r_2, ..., r_m$
  where P

  - A_i represents an attribute
  - R_i represents a relation
  - P is a predicate.
- **The result of an SQL query is a relation**

The Select Clause

- The **select** clause list the attributes desired in the result of a query
  - Corresponds to the projection operation of the relational algebra
- Example: Find the names of all instructors:
  - **select** *name* **from** *instructor*

The Select Clause (Cont.)

- SQL allows duplicates in relations as well as in query results
- To force the elimination of duplicates, insert the keyword **distinct** after select.
- **Syntax:**

  SELECT DISTINCT *column1*, *column2, ...*
  FROM *table_name*;
- Find the names of all departments with instructor, and remove duplicates
  - **select distinct** *dept_name* **from** *instructor*

Difference between following:

SELECT Country FROM Customers;

SELECT DISTINCT Country FROM Customers;

1. SQL statement selects all (including the duplicates) values from the "Country" column in the "Customers" table.

2. SQL statement selects only the DISTINCT values from the "Country" column in the "Customers" table:

The select Clause (Cont.)

- An asterisk in the select clause denotes "all attributes"

  select *
  from *instructor*

- The **select** clause can contain
  - Arithmetic expressions involving the operation, +, –, *, and /, and operating on constants or attributes of tuples
  - Character Functions (uppercase…), Numeric functions (round…), Truncating numeric data, Concatenating character data, etc.
- The query:

  select *ID, name, salary/12* **from** *instructor;*

  would return a relation that is the same as the *instructor* relation, except that the value of the attribute *salary* is divided by 12.

The where Clause

- The **where** clause specifies conditions that the result must satisfy
  - Corresponds to the selection predicate of the relational algebra
- To find all instructors in Comp. Sci. dept with salary > 80000

  Select *name* **from** *instructor*
  where *dept_name* = 'Comp. Sci.' **and** *salary* > 80000

- Comparison results can be combined using the logical connectives **and, or,** and **not**

- Comparisons can be applied to results of arithmetic expressions

The from Clause

- Display data from more than one table
- The **from** clause lists the relations involved in the query
  - Corresponds to the Cartesian product operation of the relational algebra.
- Find the Cartesian product *instructor X teaches*

  select * **from** *instructor, teaches;*

  - **Generates every possible instructor – teaches pair, with all attributes from both relations**

- Cartesian product
  - Not very useful directly
  - But useful combined with where-clause condition (selection operation in relational algebra)

Cartesian Product: *instructor X teaches*

instructor

ID	*name*	*dept_name*	*salary*
10101	Srinivasan	Comp. Sci.	65000
12121	Wu	Finance	90000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
32343	El Said	History	60000

teaches

ID	*course_id*	*sec_id*	*semester*	*year*
10101	CS-101	1	Fall	2009
10101	CS-315	1	Spring	2010
10101	CS-347	1	Fall	2009
12121	FIN-201	1	Spring	2010
15151	MU-199	1	Spring	2010
22222	PHY-101	1	Fall	2009

select * from *instructor, teaches*

inst.ID	*name*	*dept_name*	*salary*	*teaches.ID*	*course_id*	*sec_id*	*semester*	*year*
10101	Srinivasan	Comp. Sci.	65000	10101	CS-101	1	Fall	2009
10101	Srinivasan	Comp. Sci.	65000	10101	CS-315	1	Spring	2010
10101	Srinivasan	Comp. Sci.	65000	10101	CS-347	1	Fall	2009
10101	Srinivasan	Comp. Sci.	65000	12121	FIN-201	1	Spring	2010
10101	Srinivasan	Comp. Sci.	65000	15151	MU-199	1	Spring	2010
10101	Srinivasan	Comp. Sci.	65000	22222	PHY-101	1	Fall	2009
...	...	...	...	...	...	...	...	...
...	...	...	...	...	...	...	...	...
12121	Wu	Finance	90000	10101	CS-101	1	Fall	2009
12121	Wu	Finance	90000	10101	CS-315	1	Spring	2010
12121	Wu	Finance	90000	10101	CS-347	1	Fall	2009
12121	Wu	Finance	90000	12121	FIN-201	1	Spring	2010
12121	Wu	Finance	90000	15151	MU-199	1	Spring	2010
12121	Wu	Finance	90000	22222	PHY-101	1	Fall	2009
...	...	...	...	...	...	...	...	...
...	...	...	...	...	...	...	...	...

Insert into Table

- To insert the values without the column names
  - **insert into** *instructor* **values** (‘10211’, ’Smith’, ’Biology’, 66000);
- To insert the values with the corresponding column names
  - **insert into** *instructor (Id, name, dept_name, Salary)* **values** (‘10211’, ’Smith’, ’Biology’, 66000);
- To insert values from other table
  - All the columns
    - **insert into** *instructor* **select * from old_instructor;**
  - Specific column
    - **insert into** *instructor (id, name)* **select** id, name **from old_instructor;**
  - Specific rows
    - **insert into** *instructor* **select * from old_instructor where id = ‘10211’;**

Delete From Table

- **delete from <tablename>** *[where condition]*
  - **Retaining the table structure and**
    - **Deletes all the rows if written without where condition**
    - Deletes only the rows satisfying the condition

- Check: delete * from <tablename>;

- The following SQL statement deletes the customer "Alfreds Futterkiste" from the "Customers" table:
- `DELETE FROM Customers WHERE CustomerName='Alfreds Futterkiste’;`

- The following SQL statement deletes all rows in the "Customers" table, without deleting the table:

- `DELETE FROM Customers;`

DELETE :

It is a Data Manipulation Language Command (DML). It is used to delete one or more tuples of a table.
With the help of the “DELETE” command, we can either delete all the rows in one go or can delete rows one by one. i.e., we can use it as per the requirement or the condition using the Where clause.
It is comparatively slower than the TRUNCATE command. The TRUNCATE command does not remove the structure of the table.

SYNTAX –
If we want to delete all the rows of the table:

```
DELETE from;
```

SYNTAX –
If we want to delete the row of the table as per the condition then we use the WHERE clause,

```
DELETE from  WHERE  ;
```

DROP :
It is a Data Definition Language Command (DDL). It is used to drop the whole table. With the help of the “DROP” command we can drop (delete) the whole structure in one go i.e. it removes the named elements of the schema. By using this command the existence of the whole table is finished or say lost.

SYNTAX –
If we want to drop the table:
DROP table ;

TRUNCATE :
It is also a Data Definition Language Command (DDL). It is used to delete all the rows of a relation (table) in one go. With the help of the "TRUNCATE" command, we can't delete the single row as here WHERE clause is not used. It is comparatively faster than the delete command as it deletes all the rows fastly.

SYNTAX –
If we want to use truncate :
TRUNCATE;

Update

- Update single or multiple columns as per requirement
  - **Update** instructor **set** salary = 20000
    - Single column, for all the Rows
  - **Update** instructor **set** name = 'Sudarshan' **where** name = 'Srinivasan"
    - Only for the specific Row
  - **Update** instructor **set** name = 'Sudarshan'**,** salary=20000 **where** name = 'Srinivasan"
    - Multiple columns for the specific row

The WHERE clause can be combined with AND, OR,and NOT operators
The AND and OR operators are used to filter records based on more than one condition:

- The AND operator displays a record if all the conditions separated by AND are TRUE.
- The OR operator displays a record if any of the conditions separated by OR is TRUE.

The NOT operator displays a record if the condition(s) is NOT TRUE.

AND Syntax:

```
SELECT column1, column2, ...
FROM table_name
WHERE condition1 AND condition2 AND condition3 ...
;
```

```
SELECT * FROM Customers
WHERE Country='Germany' AND City='Berlin';
```

The SQL statement selects all fields from "Customers" where country is "Germany" AND city is "Berlin":

```
OR Syntax:
SELECT column1, column2, ...
FROM table_name
WHERE condition1 OR condition2 OR condition3 ...;

SELECT * FROM Customers
WHERE City='Berlin' OR City='München’;
```

The SQL statement selects all fields from "Customers" where city is "Berlin" OR "München":

```
SELECT * FROM Customers
WHERE Country='Germany' OR Country='Spain';

NOT Syntax:
SELECT column1, column2, ...
FROM table_name
WHERE NOT condition;
```

```
SELECT * FROM Customers
WHERE NOT Country='Germany';
```

Having
Group by
All
As
Like
Cartesian

The Rename Operation

- The SQL allows renaming relations and attributes using the **as** clause:

  old-name **as** *new-name*
- E.g.
  - **select** *ID, name, salary/12* **as** *monthly_salary*
    from *instructor*

String Operations

- SQL includes a string-matching operator for comparisons on character strings. The operator "like" uses patterns that are described using two special characters:
  - percent (%). The % character matches any substring.
  - underscore (_). The _ character matches any character.
- Find the names of all instructors whose name includes the substring "dar".
  - **select** *name*
    from *instructor*
    where *name* **like** '%dar%'

String Operations (Cont.)

- Patters are case sensitive.
- Pattern matching examples:
  - ‘Intro%’
    - matches any string beginning with “Intro”.
  - ‘%Comp%’
    - matches any string containing “Comp” as a substring.
  - ‘_ _ _’
    - matches any string of exactly three characters.
  - ‘_ _ _ %’
    - matches any string of at least three characters.

String Operations (Cont.)

- SQL supports a variety of string operations such as
  - Foncatenation (using "||")
  - Converting from upper to lower case (and vice versa)
  - Finding string length, extracting substrings, etc.

Ordering the Display of Tuples

- List in alphabetic order the names of all instructors
  - **select distinct** *name*
    from *instructor*
    order by *name*
- Specify **desc** for descending order or **asc** for ascending order, for each attribute; ascending order is the default.
  - Example: **order by** *name* **desc**
- Can sort on multiple attributes
  - Example: **order by** *dept_name, name*

Where Clause Predicates

- SQL includes a **between** comparison operator
- Example: Find the names of all instructors with salary between \$90,000 and \$100,000 (that is, $\geq$ \$90,000 and $\leq$ \$100,000)
  - **select** *name*
    from *instructor*
    where *salary* **between** 90000 **and** 100000

Set Operations

- Union
- Intersect
- Except

Set Operations

- Find courses that ran in Fall 2009 or in Spring 2010 or both

  (**select** *course_id* **from** *section* **where** *sem* = 'Fall' **and** *year* = 2009)
  union
  (**select** *course_id* **from** *section* **where** *sem* = 'Spring' **and** *year* = 2010)

- Find courses that ran in Fall 2009 and in Spring 2010

  (**select** *course_id* **from** *section* **where** *sem* = 'Fall' **and** *year* = 2009)
  intersect
  (**select** *course_id* **from** *section* **where** *sem* = 'Spring' **and** *year* = 2010)

- Find courses that ran in Fall 2009 but not in Spring 2010

  (**select** *course_id* **from** *section* **where** *sem* = 'Fall' **and** *year* = 2009)
  except
  (**select** *course_id* **from** *section* **where** *sem* = 'Spring' **and** *year* = 2010)

Set Operations

- Set operations **union**, **intersect**, and **except**
  - Each of the above operations **automatically eliminates duplicates**
- To retain all duplicates use the corresponding multiset versions **union all, intersect all** and **except all.**

Null Values

- It is possible for tuples to have a null value, denoted by *null*, for some of their attributes
- *null* signifies an **unknown value** or that a value does not exist.
- The result of any arithmetic expression involving *null* is *null*
  - Example: 5 + *null* returns null
- The predicate **is null** can be used to check for null values.
  - Example: Find all instructors whose salary is null.
    - **select** *name*
      from *instructor*
      where *salary* **is null**

Aggregate Functions

- These functions operate on the multiset of values of a column of a relation, and return a value

  avg: average value
  min: minimum value
  max: maximum value
  sum: sum of values
  count: number of values

Aggregate Functions (Cont.)

- Find the average salary of instructors in the Computer Science department
  - **select avg** (*salary*)
    from *instructor*
    where *dept_name*= 'Comp. Sci.';
- Find the total number of instructors who teach a course in the Spring 2010 semester
  - **select count (distinct** *ID*)
    from *teaches*
    where *semester* = 'Spring' **and** *year* = 2010
- Find the number of tuples in the *course* relation
- **select count** (*)
  from *course*;

From □ Where □ Aggregate Function

MIN() Syntax

```
SELECT MIN(column_name)
FROM table_name
WHERE condition;
```

```
SELECT MIN(Price) AS SmallestPrice
FROM Products;
```

MAX() Syntax

```
SELECT MAX(column_name)
FROM table_name
WHERE condition;
```

```
SELECT MAX(Price) AS LargestPrice
FROM Products;
```

SUM() Example
The following SQL statement finds the sum of the "Quantity" fields in the "OrderDetails" table:

```
SELECT SUM(Quantity)
FROM OrderDetails;
```

Aggregate Functions – Group By

- Find the average salary of instructors in each department
  - **select** *dept_name*, **avg** (*salary*)
    from *instructor*
    group by *dept_name*;
  - Note: departments with no instructor will not appear in result

ID	*name*	*dept_name*	*salary*
76766	Crick	Biology	72000
45565	Katz	Comp. Sci.	75000
10101	Srinivasan	Comp. Sci.	65000
83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000
12121	Wu	Finance	90000
76543	Singh	Finance	80000
32343	El Said	History	60000
58583	Califieri	History	62000
15151	Mozart	Music	40000
33456	Gold	Physics	87000
22222	Einstein	Physics	95000

dept_name	*avg_salary*
Biology	72000
Comp. Sci.	77333
Elec. Eng.	80000
Finance	85000
History	61000
Music	40000
Physics	91000

Aggregate Functions – Having Clause

- Useful to state a condition that applies to groups rather than to tuples.
- Find the names and average salaries of all departments whose average salary is greater than 42000

> **select** *dept_name*, **avg** (*salary*)
> **from** *instructor*
> **group by** *dept_name*
> **having avg** (*salary*) > 42000;

Note: predicates in the **having** clause are applied **after the formation of groups** whereas predicates in the **where** clause are applied **before forming groups**

Null Values and Aggregates

- Salary: 20000

  30000

  null

  40000

- Total all salaries

  select sum (*salary*) **from** *instructor*

  - Above statement ignores null amounts

The SQL SELECT TOP Clause

The SELECT TOP clause is used to specify the number of records to return.

The SELECT TOP clause is useful on large tables with thousands of records. Returning a large number of records can impact performance.

Note: Not all database systems support the SELECT TOP clause. MySQL supports the LIMIT clause to select a limited number of records.

The following SQL statement selects the first three records from the "Customers" table

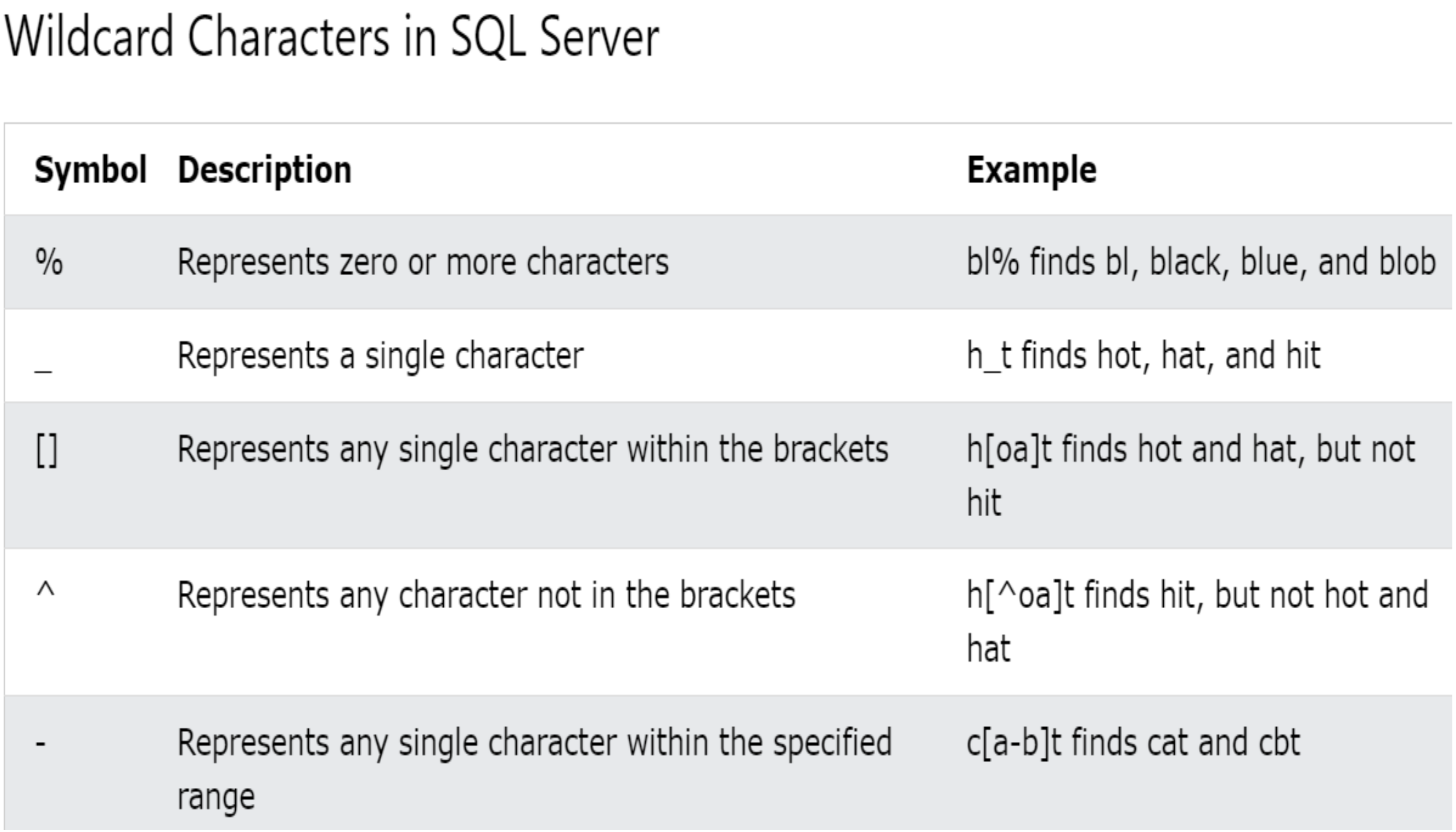
```
SELECT TOP 3 * FROM Customers;

SELECT * FROM Customers
LIMIT 3;
```

SQL Wildcard Characters

A wildcard character is used to substitute one or more characters in a string.

Wildcard characters are used with the LIKE operator. The LIKE operator is used in a WHERE clause to search for a specified pattern in a column.



Wildcard Characters in SQL Server

Symbol	Description	Example
%	Represents zero or more characters	bl% finds bl, black, blue, and blob
_	Represents a single character	h_t finds hot, hat, and hit
[]	Represents any single character within the brackets	h[oa]t finds hot and hat, but not hit
^	Represents any character not in the brackets	h[^oa]t finds hit, but not hot and hat
-	Represents any single character within the specified range	c[a-b]t finds cat and cbt

```
SELECT * FROM Customers
WHERE City LIKE 'ber%’;
```

This SQL statement selects all customers with a City starting with "ber":

The following SQL statement selects all customers with a City containing the pattern "es":

```
SELECT * FROM Customers
WHERE City LIKE '%es%';
```

The SQL IN Operator

The IN operator allows you to specify multiple values in a WHERE clause.

The IN operator is a shorthand for multiple OR conditions.

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

The following SQL statement selects all customers that are located in "Germany", "France" or "UK":

```
SELECT * FROM Customers
WHERE Country IN ('Germany', 'France', 'UK');
```

OR

SELECT column_name(s)
FROM table_name
WHERE column_name IN (SELECT STATEMENT);

The following SQL statement selects all customers that are from the same countries as the suppliers:

```
SELECT * FROM Customers
WHERE Country IN (SELECT Country FROM Suppliers);
```

SQL Aliases
SQL aliases are used to give a table, or a column in a table, a temporary name.
Aliases are often used to make column names more readable.
An alias only exists for the duration of that query.
An alias is created with the AS keyword.

The following SQL statement creates two aliases, one for the CustomerID column and one for the CustomerName column:

```
SELECT CustomerID AS ID, CustomerName AS Customer
FROM Customers;
```

Modification of the Database

- Deletion of tuples from a given relation
- Insertion of new tuples into a given relation
- Updating values in some tuples in a given relation

Modification of the Database – Deletion

- Delete all instructors

```
delete from instructor
```

- Delete all instructors from the Finance department

```
delete from instructor
            where dept_name= ’Finance’;
```

- Delete all tuples in the *instructor* relation for those instructors associated with a department located in the Watson building.

```
delete from instructor
            where dept_name in (select dept_name
                                from department
                                where building = ’Watson’);
```

Modification of the Database – Insertion

- Add a new tuple to *course*

  insert into *course*

  values (’CS-437’, ’Database Systems’, ’Comp. Sci.’, 4);

- or equivalently

  insert into *course* (*course_id*, *title*, *dept_name*, *credits*)

  values (’CS-437’, ’Database Systems’, ’Comp. Sci.’, 4);

- Add a new tuple to *student* with *tot_creds* set to null

  insert into *student*

  values (’3003’, ’Green’, ’Finance’, *null*);

Modification of the Database – Updates

- Increase salaries of instructors whose salary is over $100,000 by 3%, and all others receive a 5% raise
  - Write two **update** statements:

```
update instructor
    set salary = salary * 1.03
    where salary > 100000;

update instructor
    set salary = salary * 1.05
    where salary <= 100000;
```

  - The order is important
  - Can be done better using the **case** statement (next slide)

Nested Subqueries

- SQL provides a mechanism for the nesting of subqueries.
- A **subquery** is a **select-from-where** expression that is nested within another query.
- A common use of subqueries is to perform tests for set membership, set comparisons, and set cardinality.
- Can use
  - With where clause
  - With from clause