Chapter 3: Introduction to SQL[1]

Abu Raihan Mostofa Kamal

Professor, CSE Department
Islamic University of Technology (IUT)

August 28, 2023

[1]This is based on Textbook, its companion slide and other sources

Chapter Outline

History of SQL

- IBM Sequel language developed as part of System R project at the IBM San Jose Research Laboratory
- Renamed Structured Query Language (SQL)
- A number of standards defined (for industry compatibility)
  - ✓ SQL-86, SQL-89, SQL-92
  - ✓ SQL:1999 (language name became Y2K compliant!)
  - ✓ SQL:2003
- Commercial systems offer most, if not all, SQL-92 features, plus varying feature sets from later standards and special proprietary features.
- **Hence**, not all examples here may work on your particular system.

History of SQL

- IBM Sequel language developed as part of System R project at the IBM San Jose Research Laboratory
- Renamed Structured Query Language (SQL)
- A number of standards defined (for industry compatibility)
  - SQL-86, SQL-89, SQL-92
  - SQL:1999 (language name became Y2K compliant!)
  - SQL:2003
- Commercial systems offer most, if not all, SQL-92 features, plus varying feature sets from later standards and special proprietary features.
- **Hence**, not all examples here may work on your particular system.

History of SQL

- IBM Sequel language developed as part of System R project at the IBM San Jose Research Laboratory
- Renamed Structured Query Language (SQL)
- A number of standards defined (for industry compatibility)
  - ✓ SQL-86,SQL-89, SQL-92
  - ✓ SQL:1999 (language name became Y2K compliant!)
  - ✓ SQL:2003
- Commercial systems offer most, if not all, SQL-92 features, plus varying feature sets from later standards and special proprietary features.
- **Hence**, not all examples here may work on your particular system.

History of SQL

- IBM Sequel language developed as part of System R project at the IBM San Jose Research Laboratory
- Renamed Structured Query Language (SQL)
- A number of standards defined (for industry compatibility)
  - ✓ SQL-86,SQL-89, SQL-92
  - ✓ SQL:1999 (language name became Y2K compliant!)
  - ✓ SQL:2003
- Commercial systems offer most, if not all, SQL-92 features, plus varying feature sets from later standards and special proprietary features.
- **Hence**, not all examples here may work on your particular system.

History of SQL

- IBM Sequel language developed as part of System R project at the IBM San Jose Research Laboratory
- Renamed Structured Query Language (SQL)
- A number of standards defined (for industry compatibility)
  - ✓ SQL-86,SQL-89, SQL-92
  - ✓ SQL:1999 (language name became Y2K compliant!)
  - ✓ SQL:2003
- Commercial systems offer most, if not all, SQL-92 features, plus varying feature sets from later standards and special proprietary features.
- **Hence**, not all examples here may work on your particular system.

History of SQL

- IBM Sequel language developed as part of System R project at the IBM San Jose Research Laboratory
- Renamed Structured Query Language (SQL)
- A number of standards defined (for industry compatibility)
  - ✓ SQL-86,SQL-89, SQL-92
  - ✓ SQL:1999 (language name became Y2K compliant!)
  - ✓ SQL:2003
- Commercial systems offer most, if not all, SQL-92 features, plus varying feature sets from later standards and special proprietary features.
- **Hence**, not all examples here may work on your particular system.

History of SQL

- IBM Sequel language developed as part of System R project at the IBM San Jose Research Laboratory
- Renamed Structured Query Language (SQL)
- A number of standards defined (for industry compatibility)
  - ✓ SQL-86,SQL-89, SQL-92
  - ✓ SQL:1999 (language name became Y2K compliant!)
  - ✓ SQL:2003
- Commercial systems offer most, if not all, SQL-92 features, plus varying feature sets from later standards and special proprietary features.
- **Hence**, not all examples here may work on your particular system.

History of SQL

- IBM Sequel language developed as part of System R project at the IBM San Jose Research Laboratory
- Renamed Structured Query Language (SQL)
- A number of standards defined (for industry compatibility)
  - ✓ SQL-86,SQL-89, SQL-92
  - ✓ SQL:1999 (language name became Y2K compliant!)
  - ✓ SQL:2003
- Commercial systems offer most, if not all, SQL-92 features, plus varying feature sets from later standards and special proprietary features.
- **Hence**, not all examples here may work on your particular system.

SQL: Broad Classification

SQL can be categorized into 2 major classes:

1. Data-definition language (DDL): The SQL DDL provides commands for defining relation schemas, deleting relations, and modifying relation schemas.
   Example: Create table, drop table, modify table
2. Data-manipulation language (DML): The SQL DML provides the ability to query information from the database and to insert tuples into, delete tuples from, and modify tuples in the database.
   Example: update, insert, delete

SQL: Broad Classification

SQL can be categorized into 2 major classes:

1. Data-definition language (DDL): The SQL DDL provides commands for defining relation schemas, deleting relations, and modifying relation schemas.
   Example: Create table, drop table, modify table
2. Data-manipulation language (DML): The SQL DML provides the ability to query information from the database and to insert tuples into, delete tuples from, and modify tuples in the database.
   Example: update, insert, delete

Data Definition Language

The SQL data-definition language (DDL) allows the specification of information about relations, including:

- The schema for each relation.
- The type of values associated with each attribute.
- The Integrity constraints (such as primary constraint and others)
- The set of indices to be maintained for each relation.
- Security and authorization information for each relation.
- The physical storage structure of each relation on disk.

Data Definition Language

The SQL data-definition language (DDL) allows the specification of information about relations, including:

- The schema for each relation.
- The type of values associated with each attribute.
- The Integrity constraints (such as primary constraint and others)
- The set of indices to be maintained for each relation.
- Security and authorization information for each relation.
- The physical storage structure of each relation on disk.

Data Definition Language

The SQL data-definition language (DDL) allows the specification of information about relations, including:

- The schema for each relation.
- The type of values associated with each attribute.
- The Integrity constraints (such as primary constraint and others)
- The set of indices to be maintained for each relation.
- Security and authorization information for each relation.
- The physical storage structure of each relation on disk.

Data Definition Language

The SQL data-definition language (DDL) allows the specification of information about relations, including:

- The schema for each relation.
- The type of values associated with each attribute.
- The Integrity constraints (such as primary constraint and others)
- The set of indices to be maintained for each relation.
- Security and authorization information for each relation.
- The physical storage structure of each relation on disk.

Data Definition Language

The SQL data-definition language (DDL) allows the specification of information about relations, including:

- The schema for each relation.
- The type of values associated with each attribute.
- The Integrity constraints (such as primary constraint and others)
- The set of indices to be maintained for each relation.
- Security and authorization information for each relation.
- The physical storage structure of each relation on disk.

Data Definition Language

The SQL data-definition language (DDL) allows the specification of information about relations, including:

- The schema for each relation.
- The type of values associated with each attribute.
- The Integrity constraints (such as primary constraint and others)
- The set of indices to be maintained for each relation.
- Security and authorization information for each relation.
- The physical storage structure of each relation on disk.

Domain Types in SQL

- **char(n)**. Fixed length character string, with user-specified length n.
- **varchar(n)**. ANSI standard. Variable length character strings, with user-specified maximum length n. Distinguish between null and empty string.
- **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 d digits to the right of decimal point. (ex., numeric(3,1), allows 44.5 to be stores exactly, but not 444.5 or 0.32)
- **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.

Domain Types in SQL

- **char(n)**. Fixed length character string, with user-specified length n.
- **varchar(n)**. ANSI standard. Variable length character strings, with user-specified maximum length n. Distinguish between null and empty string.
- **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 d digits to the right of decimal point. (ex., numeric(3,1), allows 44.5 to be stores exactly, but not 444.5 or 0.32)
- **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.

Domain Types in SQL

- **char(n)**. Fixed length character string, with user-specified length n.
- **varchar(n)**. ANSI standard. Variable length character strings, with user-specified maximum length n. Distinguish between null and empty string.
- **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 d digits to the right of decimal point. (ex., numeric(3,1), allows 44.5 to be stores exactly, but not 444.5 or 0.32)
- **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.

Domain Types in SQL

- **char(n)**. Fixed length character string, with user-specified length n.
- **varchar(n)**. ANSI standard. Variable length character strings, with user-specified maximum length n. Distinguish between null and empty string.
- **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 d digits to the right of decimal point. (ex., numeric(3,1), allows 44.5 to be stores exactly, but not 444.5 or 0.32)
- **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.

Domain Types in SQL

- **char(n)**. Fixed length character string, with user-specified length n.
- **varchar(n)**. ANSI standard. Variable length character strings, with user-specified maximum length n. Distinguish between null and empty string.
- **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 d digits to the right of decimal point. (ex., numeric(3,1), allows 44.5 to be stores exactly, but not 444.5 or 0.32)
- **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.

Domain Types in SQL

- **char(n)**. Fixed length character string, with user-specified length n.
- **varchar(n)**. ANSI standard. Variable length character strings, with user-specified maximum length n. Distinguish between null and empty string.
- **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 d digits to the right of decimal point. (ex., numeric(3,1), allows 44.5 to be stores exactly, but not 444.5 or 0.32)
- **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.

Domain Types in SQL

- **char(n)**. Fixed length character string, with user-specified length n.
- **varchar(n)**. ANSI standard. Variable length character strings, with user-specified maximum length n. Distinguish between null and empty string.
- **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 d digits to the right of decimal point. (ex., numeric(3,1), allows 44.5 to be stores exactly, but not 444.5 or 0.32)
- **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.

Domain Types in SQL: Oracle Implementation

In Oracle Database (basic) data-types are classified as follows:

- **varchar2(n)**: Oracle standard. n bytes/(characters!!) are allocated. Null and empty string are interchangable.
- **number(p,s)**: where p is the precision: total number of digits while s is the digits after decimal point. Any number can be generated varying these 2 parameters.
  Example: only fraction: number(3,3)
  integer value: number(3,0)
- **Date:** it is not string anymore, all date related built-ins can be applied here. (Chapter 4 for more details)

Domain Types in SQL: Oracle Implementation

In Oracle Database (basic) data-types are classified as follows:

- **varchar2(n)**: Oracle standard. n bytes/(characters!!) are allocated. Null and empty string are interchangable.
- **number(p,s)**: where p is the precision: total number of digits while s is the digits after decimal point. Any number can be generated varying these 2 parameters.
  Example: only fraction: number(3,3)
  integer value: number(3,0)
- **Date:** it is not string anymore, all date related built-ins can be applied here. (Chapter 4 for more details)

Domain Types in SQL: Oracle Implementation

In Oracle Database (basic) data-types are classified as follows:

- **varchar2(n)**: Oracle standard. n bytes/(characters!!) are allocated. Null and empty string are interchangable.
- **number(p,s)**: where p is the precision: total number of digits while s is the digits after decimal point. Any number can be generated varying these 2 parameters.
  Example: only fraction: number(3,3)
  integer value: number(3,0)
- **Date:** it is not string anymore, all date related built-ins can be applied here. (Chapter 4 for more details)

Create Table Construct

```
create table r
(A1 D1, A2 D2, ..., A n Dn,
(integrity-constraint1),
...,
(integrity-constraintk))
```

- r is the name of the relation, Ai is the attribute
- Di is the data type of values in the domain of attribute Ai
- Types of (basic) integrity constraints:
  1. primary key ($A_1, ..., A_n$)
  2. foreign key ($A_m, ..., A_n$) references r
  3. not null

Create Table Construct

```
create table r
(A1 D1, A2 D2, ..., A n Dn,
(integrity-constraint1),
...,
(integrity-constraintk))
```

- r is the name of the relation, Ai is the attribute
- Di is the data type of values in the domain of attribute Ai
- Types of (basic) integrity constraints:
  1. primary key ($A_1, \ldots, A_n$)
  2. foreign key ($A_m, \ldots, A_n$) references r
  3. not null

Create Table Construct

```
create table r
(A1 D1, A2 D2, ..., A n Dn,
(integrity-constraint1),
...,
(integrity-constraintk))
```

- r is the name of the relation, Ai is the attribute
- Di is the data type of values in the domain of attribute Ai
- Types of (basic) integrity constraints:
  1. primary key (A1, ..., An)
  2. foreign key (Am , ..., An) references r
  3. not null

Create Table Construct

```
create table r
(A1 D1, A2 D2, ..., A n Dn,
(integrity-constraint1),
...,
(integrity-constraintk))
```

- r is the name of the relation, Ai is the attribute
- Di is the data type of values in the domain of attribute Ai
- Types of (basic) integrity constraints:
  1. primary key (A1, ..., An)
  2. foreign key (Am , ..., An) references r
  3. not null

Create Table Construct

```
create table r
(A1 D1, A2 D2, ..., A n Dn,
(integrity-constraint1),
...,
(integrity-constraintk))
```

- r is the name of the relation, Ai is the attribute
- Di is the data type of values in the domain of attribute Ai
- Types of (basic) integrity constraints:
  1. primary key (A1, ..., An)
  2. foreign key (Am , ..., An) references r
  3. not null

Create Table Construct

```
create table r
(A1 D1, A2 D2, ..., A n Dn,
(integrity-constraint1),
...,
(integrity-constraintk))
```

- r is the name of the relation, Ai is the attribute
- Di is the data type of values in the domain of attribute Ai
- Types of (basic) integrity constraints:
  1. primary key (A1, ..., An)
  2. foreign key (Am , ..., An) references r
  3. not null

Create Table Construct: Example

```
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)
```

Create Table Construct: Better Example

```
create table instructors
(ID varchar2(5),
name varchar2(20) not null,
dept_name varchar(20),
salary number(8,2),

constraint pk_instructor primary key (ID),
constraint fk_instructor_dept foreign key (dept_name)
references department)
```

The benefit of naming constraint is mostly felt when errors occurs.

Alter / Drop Table Construct

- **drop table r** : table r is dropped both its structure and data are lost
- alter table r add A D
  - where A is the name of the attribute to be added to relation r and D is the domain of A
  - All existing tuples in the relation are assigned null as the value for the new attribute.

Alter / Drop Table Construct

- **drop table r** : table r is dropped both its structure and data are lost
- **alter table r add A D**
  - ✓ where A is the name of the attribute to be added to relation r and D is the domain of A.
  - ✓ All exiting tuples in the relation are assigned null as the value for the new attribute.

Alter / Drop Table Construct

- **drop table r** : table r is dropped both its structure and data are lost
- **alter table r add A D**
  - ✓ where A is the name of the attribute to be added to relation r and D is the domain of A.
  - ✓ All existing tuples in the relation are assigned null as the value for the new attribute.

Alter / Drop Table Construct

- **drop table r** : table r is dropped both its structure and data are lost
- **alter table r add A D**
  - ✓ where A is the name of the attribute to be added to relation r and D is the domain of A.
  - ✓ All exiting tuples in the relation are assigned null as the value for the new attribute.

Alter / Drop Table Example

```
alter table instructor
 add address varchar2(40);
```

Listing: alter table

```
drop table instructor;
```

Listing: drop table

Basic Structure of SQL Queries: Single Relation

The basic structure of an SQL query consists of three clauses: select, from, and where clause.

- Queries on a **Single Relation**.

```
select id,name, salary
from instructors
where dept_name='CSE' or dept_name='EEE';
```

- where clause `dept_name='CSE' or dept_name='EEE'` ... predicate part is the select operator $\sigma_p(r)$ (sigma)
- select part is the project $\Pi A_1, A_2, A_N(r)$

Basic Structure of SQL Queries: Single Relation

The basic structure of an SQL query consists of three clauses: select, from, and where clause.

- Queries on a **Single Relation**.

```
select id,name, salary
from instructors
where dept_name='CSE' or dept_name='EEE';
```

- where clause `dept_name='CSE' or dept_name='EEE'` ... predicate part is the select operator $\sigma_p(r)$ (sigma)
- select part is the project $\Pi A_1, A_2, A_N(r)$

Basic Structure of SQL Queries: Single Relation

The basic structure of an SQL query consists of three clauses: select, from, and where clause.

- Queries on a **Single Relation**.

```
select id,name, salary
from instructors
where dept_name='CSE' or dept_name='EEE';
```

- where clause `dept_name='CSE' or dept_name='EEE'` ... predicate part is the select operator $\sigma_p(r)$ (sigma)
- select part is the project $\Pi A_1, A_2, A_N(r)$

The select Clause

- Select **all** can be expressed by * sign as follows

```
select *
from stutents;
```

- Keyword **distinct** is used for eliminating duplicate records (not duplicate column)

```
select distinct dept,name
from stutents;
```

The select Clause

- Select **all** can be expressed by * sign as follows

```
select *
from stutents;
```

- Keyword **distinct** is used for eliminating duplicate records (not duplicate column)

```
select distinct dept,name
from stutents;
```

Basic Structure of SQL Queries: Multiple Relations

- Queries on a **Cartesian Product**. Remember it is all possible combinations between 2 relations.

```
create table dept
(dept_name varchar2(10),
location varchar2(10),
year_est number(4,0),
yearly_budget number(10,2),
constraint pk_dept primary key(dept)
);

create table students
( name varchar2(20),
prog varchar2(20),
dob date,
cgpa number(5,4),
dept varchar2(10),
constraint fk_students foreign key(dept)
references dept(dept)
);
```

```
select *
from dept, students;
```

Listing: Cartesian Product, no where clause

The Rename Operation

Renaming in SQL is very useful in a number of cases:

1. Rename attribute name
2. Rename expression
3. Rename relation name
4. Rename entire query
5. Rename to compare tuples in the same relation (Self-reference)

The Rename Operation

Renaming in SQL is very useful in a number of cases:

1. Rename attribute name
2. Rename expression
3. Rename relation name
4. Rename entire query
5. Rename to compare tuples in the same relation (Self-reference)

The Rename Operation

Renaming in SQL is very useful in a number of cases:

1. Rename attribute name
2. Rename expression
3. Rename relation name
4. Rename entire query
5. Rename to compare tuples in the same relation (Self-reference)

The Rename Operation

Renaming in SQL is very useful in a number of cases:

1. Rename attribute name
2. Rename expression
3. Rename relation name
4. Rename entire query
5. Rename to compare tuples in the same relation (Self-reference)

The Rename Operation

Renaming in SQL is very useful in a number of cases:

1. Rename attribute name
2. Rename expression
3. Rename relation name
4. Rename entire query
5. Rename to compare tuples in the same relation (Self-reference)

Rename: Examples

```
select name as instructorname,
courseid
from instructor, teaches
where instructor.ID= teaches.ID;
```

Listing: Rename Attribute

```
select ID,name,
(salary/12) MSalary
from emp;
```

Listing: Rename expression

[1] as keyword is optional in oracle database

Rename: Examples (Cont.1)

```
select name as instructorname,
courseid
from instructor I, teaches T
where I.ID= T.ID;
```

Listing: Rename Relation

- Renaming Relation is called **Correlation Name** in the SQL standard, but it is also commonly referred to as a **Table Alias**

Rename: Examples (Cont.2)

```
select A.N as Name,A.CID as Course
from
(select name as N, CourseID as CID
from instructor I, teaches T
where I.ID= T.ID) A
;
```

Listing: Rename a Query

- Such renaming is very useful particularly when we need to sort records not based on the stored values but on the computed values (will be covered in soon in details).

Rename: Examples (Cont.3)

```
select E1.ID,
E1.Name, E1.BID
from Emp E1, Emp E2
where E1.BID=E2.ID;
```

Listing: Rename for self-reference

- Such self-reference works like a for loop in C.

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:
  1. percent (%). The % character matches any substring.
  2. underscore (_). The _ character matches any character.
- An Exact Match is done using = operator. Single quote is used for string matching and backslash \ as the escape character

Example:

```
select *
from depts
where name='CSE';
```

Listing: String Operation

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:
  1. percent (%). The % character matches any substring.
  2. underscore (_). The _ character matches any character.
- An Exact Match is done using = operator. Single quote is used for string matching and backslash \ as the escape character

Example:

```
select *
from depts
where name='CSE';
```

Listing: String Operation

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:
  1. percent (%). The % character matches any substring.
  2. underscore (_). The _ character matches any character.
- An Exact Match is done using = operator. Single quote is used for string matching and backslash \ as the escape character

Example:

```
select *
from depts
where name='CSE';
```

Listing: String Operation

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:
  1. percent (%). The % character matches any substring.
  2. underscore (_). The _ character matches any character.
- An Exact Match is done using = operator. Single quote is used for string matching and backslash \ as the escape character

Example:

```
select *
from depts
where name='CSE';
```

Listing: String Operation

String Operations (Cont.)

- Patterns 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.
  - '%a%' matches any string containing letter a anywhere.
- SQL supports a variety of string operations such as:
  - concatenation (using || sign)
  - Many built-ins are normally used (i.e. UPPER, LOWER, SUBSTR.)

String Operations (Cont.)

- Patterns 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.
  - ✓ '%m%' matches any string containing letter m anywhere.
- SQL supports a variety of string operations such as:
  - ✓ concatenation (using || sign)
  - ✓ Many built-ins are normally used (i.e. UPPER, LOWER, SUBSTR,)

String Operations (Cont.)

- Patterns 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.
  - ✓ '%m%' matches any string containing letter m anywhere.
- SQL supports a variety of string operations such as:
  - ✓ concatenation (using || sign)
  - ✓ Many built-ins are normally used (i.e. UPPER, LOWER, SUBSTR.)

String Operations (Cont.)

- Patterns 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.
  - ✓ '%m%' matches any string containing letter m anywhere.
- SQL supports a variety of string operations such as:
  - ✓ concatenation (using || sign)
  - ✓ Many built-ins are normally used (i.e. UPPER, LOWER, SUBSTR.)

String Operations (Cont.)

- Patterns 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.
  - ✓ '%m%' matches any string containing letter m anywhere.
- SQL supports a variety of string operations such as:
  - ✓ concatenation (using || sign)
  - ✓ Many built-ins are normally used (i.e. UPPER, LOWER, SUBSTR,)

String Operations (Cont.)

- Patterns 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.
  - ✓ `'%m%'` matches any string containing letter m anywhere.
- SQL supports a variety of string operations such as:
  - ✓ concatenation (using || sign)
  - ✓ Many built-ins are normally used (i.e. UPPER, LOWER, SUBSTR,)

String Operations (Cont.)

- Patterns 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.
  - ✓ `'%m%'` matches any string containing letter m anywhere.
- SQL supports a variety of string operations such as:
  - ✓ concatenation (using || sign)
  - ✓ Many built-ins are normally used (i.e. UPPER, LOWER, SUBSTR..)

String Operations (Cont.)

- Patterns 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.
  - ✓ `'%m%'` matches any string containing letter m anywhere.
- SQL supports a variety of string operations such as:
  - ✓ concatenation (using || sign)
  - ✓ Many built-ins are normally used (i.e. UPPER, LOWER, SUBSTR..)

String Operations (Cont.)

- Patterns 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.
  - ✓ `'%m%'` matches any string containing letter m anywhere.
- SQL supports a variety of string operations such as:
  - ✓ concatenation (using || sign)
  - ✓ Many built-ins are normally used (i.e. UPPER, LOWER, SUBSTR..)

Set Operations: When to use

- In practice, set operations are used to extract data from two relations having no references (it may happen if data is collected from a legacy system and merged into new system)
- It is also useful to merge data from different sources where conditions vary in each case.
- In many cases, (specially the SQL shown here) same SQL can be written alternatively (using AND , OR, NOT operators) in much simpler way.

Set Operations: When to use

- In practice, set operations are used to extract data from two relations having no references (it may happen if data is collected from a legacy system and merged into new system)
- It is also useful to merge data from different sources where conditions vary in each case.
- In many cases, (specially the SQL shown here) same SQL can be written alternatively (using AND , OR, NOT operators) in much simpler way.

Set Operations: When to use

- In practice, set operations are used to extract data from two relations having no references (it may happen if data is collected from a legacy system and merged into new system)
- It is also useful to merge data from different sources where conditions vary in each case.
- In many cases, (specially the SQL shown here) same SQL can be written alternatively (using AND , OR, NOT operators) in much simpler way.

Set Operations: Examples (Cont.)

- Find courses that ran in Fall 2017 or in Spring 2018

```
(select course_id from section where sem = Fall and year = 2017)
union
(select course_id from section where sem = Spring and year = 2018)
```

- Find courses that ran in Fall 2017 and in Spring 2018

```
(select course_id from section where sem = Fall and year = 2017)
intersect
(select course_id from section where sem = Spring and year = 2018)
```

Set Operations: Examples (Cont.)

- Find courses that ran in Fall 2017 or in Spring 2018

```
(select course_id from section where sem = Fall and year = 2017)
union
(select course_id from section where sem = Spring and year = 2018)
```

- Find courses that ran in Fall 2017 and in Spring 2018

```
(select course_id from section where sem = Fall and year = 2017)
intersect
(select course_id from section where sem = Spring and year = 2018)
```

Null values

- **Null Values** present special problems in relational operations, including arithmetic operations, comparison operations, and set operations.
- For example:

```
select Name, (salary+bonus) Total
from emp;
```

For employees who have null values for their bonus will return Null as total salary!!

- Solution: use **nvl()** built-in.
  Example:

```
select Name, (salary+ nvl(bonus,0)) Total
from emp;
```

Now the null values found will be treated as 0, so total salary is now meaningful.

Null values

- **Null Values** present special problems in relational operations, including arithmetic operations, comparison operations, and set operations.
- For example:

```
select Name, (salary+bonus) Total
from emp;
```

For employees who have null values for their bonus will return Null as total salary!!

- Solution: use **nvl()** built-in.
  Example:

```
select Name, (salary+ nvl(bonus,0)) Total
from emp;
```

Now the null values found will be treated as 0, so total salary is now meaningful.

Null values

- **Null Values** present special problems in relational operations, including arithmetic operations, comparison operations, and set operations.
- For example:

```
select Name, (salary+bonus) Total
from emp;
```

For employees who have null values for their bonus will return Null as total salary!!

- Solution: use **nvl()** built-in.
  Example:

```
select Name, (salary+ nvl(bonus,0)) Total
from emp;
```

Now the null values found will be treated as 0, so total salary is now meaningful.

Aggregate Functions

- Aggregate functions are functions that take a collection of values as input and return a single value. SQL offers five (5) standard built-in aggregate functions (Note: more are available in many commercial databases):
  - ✓ Average: **avg**
  - ✓ Minimum: **min**
  - ✓ Maximum: **max**
  - ✓ Total: **sum**
  - ✓ Count: **count**

  Note: The input to **sum** and **avg** must be a collection of numbers, but the other operators can operate on collections of nonnumeric data types, such as strings, as well

Aggregate Functions: Examples

- 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 2018 semester

```
select count (distinct ID)
from teaches
where semester = 'Spring' and year = 2018;
```

- Find the number of tuples/records in the course relation

```
select count (*)
from course;
```

Aggregate Functions: Examples

- 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 2018 semester

```
select count (distinct ID)
from teaches
where semester = 'Spring' and year = 2018;
```

- Find the number of tuples/records in the course relation

```
select count (*)
from course;
```

Aggregate Functions: Examples

- 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 2018 semester

```
select count (distinct ID)
from teaches
where semester = 'Spring' and year = 2018;
```

- Find the number of tuples/records in the course relation

```
select count (*)
from course;
```

Aggregation with Grouping

- Need to apply the aggregate function not only to a single set of tuples, but also to a group of sets of tuples; we specify this in SQL using the group by clause.
- Tuples with the same value on all attributes in the **group by** clause are placed in one group.
- Applying avg() of salary in instructor relation will return one single value. But we are more interested for finding average salary for each department.

Aggregation with Grouping

- Need to apply the aggregate function not only to a single set of tuples, but also to a group of sets of tuples; we specify this in SQL using the group by clause.
- Tuples with the same value on all attributes in the **group by** clause are placed in one group.
- Applying avg() of salary in instructor relation will return one single value. But we are more interested for finding average salary for each department.

Aggregation with Grouping

- Need to apply the aggregate function not only to a single set of tuples, but also to a group of sets of tuples; we specify this in SQL using the group by clause.
- Tuples with the same value on all attributes in the **group by** clause are placed in one group.
- Applying avg() of salary in instructor relation will return one single value. But we are more interested for finding average salary for each department.

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

Aggregation with Grouping

- Find the average salary in each department

```
select deptname, avg (salary) as avg salary
from instructor
group by deptname;
```

Aggregation: Basic Principle

All selected attributes must be present either in (i) aggregate functions or in (ii) group by clause.

- Following Query has some errors!!!

```
select id,deptname, avg (salary) as avg salary
from instructor
group by deptname;
```

- It is needed specially when we need joining over an aggregated function. For example: Find the department name, its budget and total number of students. **Trick:** use either max or min functions for those extra attributes.

  test the Query

```
select dname,max(budget), count (*) totalStudent
from depts,students
where depts.dname=students.dept
group by dname;
```

Aggregation: Basic Principle

All selected attributes must be present either in (i) aggregate functions or in (ii) group by clause.

- Following Query has some errors!!!

```
select id,deptname, avg (salary) as avg salary
from instructor
group by deptname;
```

- It is needed specially when we need joining over an aggregated function. For example: Find the department name, its budget and total number of students. **Trick:** use either max or min functions for those extra attributes.

  test the Query

```
select dname,max(budget), count (*) totalStudent
from depts,students
where depts.dname=students.dept
group by dname;
```

Aggregation: Having Clause

- It is useful to state a condition that applies to groups (which will be computed at run-time)
- **where** clause can only be applied on the values already stored, but here the values are not stored rather will be computed on-the-fly. Hence we have to use **Having** clause
- Find out the department name, its budget and total number of students for those who have at-least 200 students;

```
select dname,max(budget), count (*) totalStudent
from depts,students
where depts.dname=students.dept
group by dept
having count(*)>=200;
```

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

Aggregation: Having Clause

- It is useful to state a condition that applies to groups (which will be computed at run-time)
- **where** clause can only be applied on the values already stored, but here the values are not stored rather will be computed on-the-fly. Hence we have to use **Having** clause
- Find out the department name, its budget and total number of students for those who have at-least 200 students;

```
select dname,max(budget), count (*) totalStudent
from depts,students
where depts.dname=students.dept
group by dept
having count(*)>=200;
```

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

Aggregation: Having Clause

- It is useful to state a condition that applies to groups (which will be computed at run-time)
- **where** clause can only be applied on the values already stored, but here the values are not stored rather will be computed on-the-fly. Hence we have to use **Having** clause
- Find out the department name, its budget and total number of students for those who have at-least 200 students;

```
select dname,max(budget), count (*) totalStudent
from depts,students
where depts.dname=students.dept
group by dept
having count(*)>=200;
```

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

Aggregation: Having Clause

- It is useful to state a condition that applies to groups (which will be computed at run-time)
- **where** clause can only be applied on the values already stored, but here the values are not stored rather will be computed on-the-fly. Hence we have to use **Having** clause
- Find out the department name, its budget and total number of students for those who have at-least 200 students;

```
select dname,max(budget), count (*) totalStudent
from depts,students
where depts.dname=students.dept
group by dept
having count(*)>=200;
```

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

Sub-query

- SQL provides a mechanism for nesting subqueries. A subquery is a select-from-where expression that is nested within another query.
- It can be placed at 2 places:
  1. A subquery in the WHERE clause of a SELECT statement is also called *a nested subquery*.
  2. A subquery in the FROM clause of a SELECT statement is also called *an inline view*.

Sub-query

- SQL provides a mechanism for nesting subqueries. A subquery is a select-from-where expression that is nested within another query.
- It can be placed at 2 places:
  1. A subquery in the **WHERE** clause of a SELECT statement is also called *a nested subquery*.
  2. A subquery in the **FROM** clause of a SELECT statement is also called *an inline view*.

Sub-query

- SQL provides a mechanism for nesting subqueries. A subquery is a select-from-where expression that is nested within another query.
- It can be placed at 2 places:
  1. A subquery in the **WHERE** clause of a SELECT statement is also called *a nested subquery*.
  2. A subquery in the **FROM** clause of a SELECT statement is also called *an inline view*.

Sub-query

- SQL provides a mechanism for nesting subqueries. A subquery is a select-from-where expression that is nested within another query.
- It can be placed at 2 places:
  1. A subquery in the **WHERE** clause of a SELECT statement is also called *a nested subquery*.
  2. A subquery in the **FROM** clause of a SELECT statement is also called *an inline view*.

Subquery in WHERE clause: Example

- Find the employees name,address and salary for those who are now getting more than the average salary of the company.

```
select id,name,salary
from emp
where salary>(select avg(salary) from emp);
```

- Here, subquery in the where part is first executed and the value is locked. This is particularly important for update statement. Suppose, consider the Query:
Increase salary of employees by 20% for those who are now getting lower than the average salary of the company.

```
update emp
set salary=salary*1.2
where salary<(select avg(salary) from emp);
```

Subquery in WHERE clause: Example

- Find the employees name,address and salary for those who are now getting more than the average salary of the company.

```
select id,name,salary
from emp
where salary>(select avg(salary) from emp);
```

- Here, subquery in the where part is first executed and the value is locked. This is particularly important for update statement. Suppose, consider the Query:
Increase salary of employees by 20% for those who are now getting lower than the average salary of the company.

```
update emp
set salary=salary*1.2
where salary<(select avg(salary) from emp);
```

Subquery in WHERE clause: Set-membership

- IN and NOT IN are normally used in the where clause as a mechanism of set-membership. It works in iterative for (i.e. loop as long as data exists). In other words, it works as a subset membership.
- Example: Find courses offered in Fall 2017 and in Spring 2018

```
select distinct course_id
from section
where semester = 'Fall' and year= 2017 and
course_id in (select course_id
from section
where semester = 'Spring' and year= 2018);
```

Subquery in WHERE clause: Set-membership

- **IN** and **NOT IN** are normally used in the where clause as a mechanism of set-membership. It works in iterative for (i.e. loop as long as data exists). In other words, it works as a subset membership.
- **Example:** Find courses offered in Fall 2017 and in Spring 2018

```
select distinct course_id
from section
where semester = 'Fall' and year= 2017 and
course_id in (select course_id
from section
where semester = 'Spring' and year= 2018);
```

Subquery in FROM clause: Example

- A subquery can also be found in the FROM clause. These are called inline views.
- **Example:** Display the top five earner names and salaries from the EMPLOYEES table:

```
SELECT ROWNUM as RANK, last_name, salary
FROM (SELECT last_name, salary
FROM employees
ORDER BY salary DESC)
WHERE ROWNUM <= 5;
```

DML Statements

There are 3 DML statements:

1. **INSERT** : new record entry
2. **UPDATE** : existing record modification
3. **DELETE** : existing record remove

General INSERT Statement: Example

- Add a new tuple to course:

```
insert into course
values ('CS-437', 'Database Systems', 'Comp. Sci.', 4);

%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);
```

INSERT Statement: INSERT INTO SELECT...FROM

- More dynamically, we might want to insert tuples on the basis of the result of a query.
- We can follow `INSERT INTO T1 (SELECT ...FROM T2);`
- **Example:** Make each student in the Music department who has earned more than 144 credit hours an instructor in the Music department with a salary of $18,000 (fixed values can be passed as well).

```
insert into instructor
select ID, name, dept_name, 18000
from student
where dept_name = 'Music' and total_cred > 144;
```

UPDATE statement: Example

- Give a 5% salary raise to those instructors who earn less than 70000

```
update instructor
set salary = salary * 1.05
where salary < 70000;
```

- Give a 5% salary raise to instructors whose salary is less than average (already shown!!)

```
update instructor
set salary = salary * 1.05
where salary < (select avg (salary)
from instructor);
```

Compound UPDATE statement

- Sometimes, we need to specify a number of different conditions for updating. Simply we can achieve it by writing different UPDATE statements for each condition.
- **Example:** Increase salaries of instructors whose salary is over $\$100,000$ by 3%, and all others by a 5%
  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. Together they will form a **transaction**.
- Can be done better using the case statement (newly introduced)

Compound UPDATE statement (Cont.)

- **Same Example:** (using case) Increase salaries of instructors whose salary is over $100,000$ by 3%, and all others by a 5%

```
update instructor
set salary = case
when salary <= 100000 then salary * 1.05
else salary * 1.03
end;
```

DELETE statement: Example

- Delete all instructors:

  ```
  delete [from] instructor;
  ```

- 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');
```

DELETE statement: Example (Cont.)

- Delete all instructors whose salary is less than the average salary of instructors

```
delete from instructor
where salary < (select avg (salary)
from instructor);
```

- **Problem:** as we delete tuples from instructor, the average salary changes
- **Solution:** In RDBMS, locking is used to resolve the issue. First the subquery is computed and locked. Then delete statement is executed one by one without recomputing subquery (ie. avg)

End of Chapter 3

Thank You