Review on Database Management System

Anthony

Database

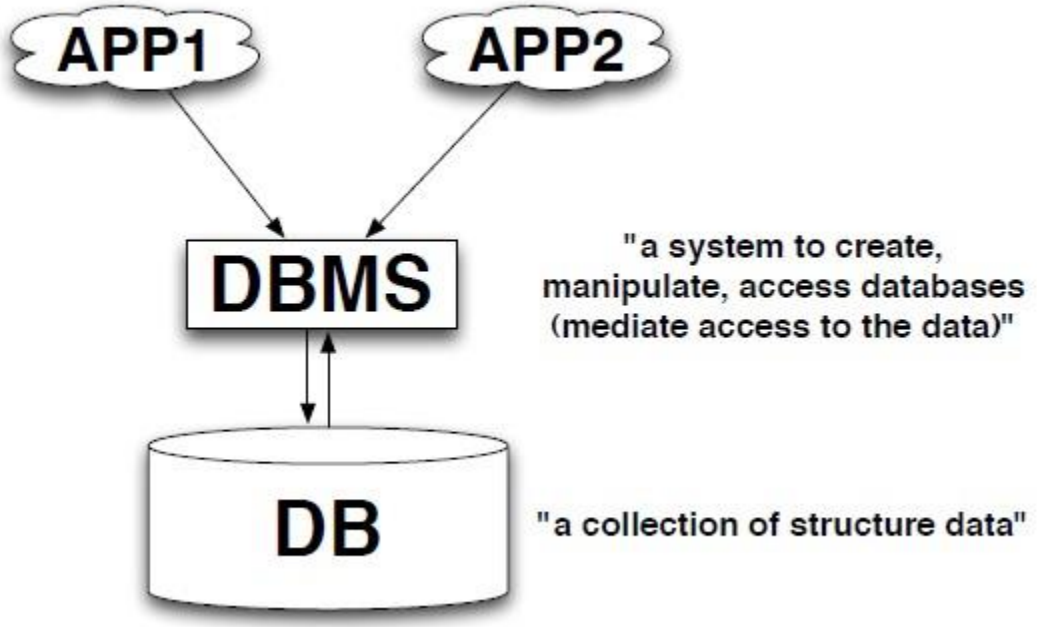
- A collection of data (Structured and Unstructured).
- Organized as records.
- Relationship between records.[1]

[1] With permission from Kaleema

Database Management Systems

DBMS is a collection of interrelated data and a set of programs to access those data.

The primary goal of a DBMS is to provide a way to store and retrieve database information that is both *convenient* and *efficient*



Database-System Applications

DBMS arose in 1960s in response to the computerized management of commercial data. They are :

- Highly valuable,
- Relatively large, and
- Accessed by multiple users and applications, often at the same time.
- Key to the management of complexity is the concept of abstraction.

Abstraction

- Management of complexity.
- Through abstraction DBMS makes it possible for any enterprise to combine data of various types into a unified repository.
- Sales

- Manufacturing
- Banking
- Web-based service
- Document Database
- Navigation systems
- As database systems became more sophisticated, better languages were developed for programmers to use in interacting with the data

Modes

Two modes in which databases are used:

- The first mode is to support *online transaction processing*
- The second mode is to support *data analytics called online analytical processing*

Purpose of Database Systems

- Data redundancy and inconsistency
- Difficulty in accessing data.
- Data isolation.
- Integrity problems.
- Atomicity problems.
- Concurrent-access anomalies.
- Security problems

View of Data

Data Models:

Underlying the structure of a database is the data model

- Relational Model:

  Uses a collection of tables to represent both data and the relationships among those data
- Entity-Relationship Model:

  The entity-relationship (E-R) data model uses a collection of basic objects, called *entities*, and *relationships* among these objects
- Semi-structured Data Model:

  The semi-structured data model permits the specification of data where individual data items of the same type may have different sets of attributes.
- Object-Based Data Model

  Objects Based data model is well integrated into relational databases

Database Languages

- data-definition language (DDL) - to specify the database schema
- data-manipulation language (DML) to express database queries and updates Retrieval of information stored in the database.

Insertion of new information into the database.

Deletion of information from the database.

Modification of information stored in the database

- Procedural DMLs require a user to specify *what* data are needed and *how* to get those data.
- Declarative DMLs (also referred to as nonprocedural DMLs) require a user to specify *what* data are needed *without* specifying how to get those data.

Database Languages

The SQL Data-Manipulation Language

- The SQL query language is nonprocedural. A query takes as input several tables (possibly only one) and always returns a single table.

- **Database Access from Application Programs**

- SQL does not support action such as input from users, output to displays, or communication over the network. Such computations and actions must be written in a *host* language, such as C/C++,Java, or Python, with embedded SQL queries that access the data in the database

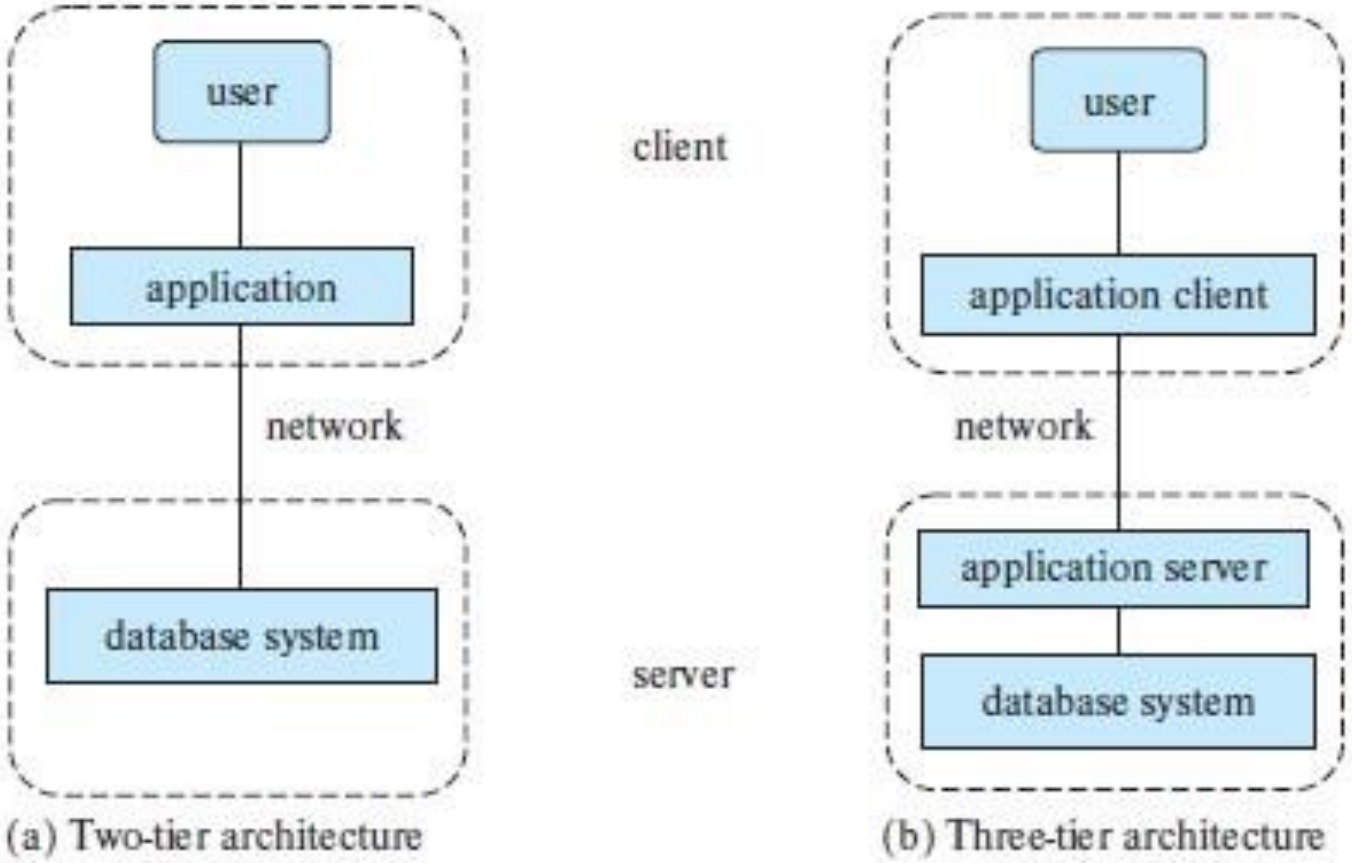
Database Engine

- Storage Manager
- The Query Processor
- Transaction Management

Database and Application Architecture


naive users
(tellers, agents,
web users)
application
programmers
sophisticated
users
(analysts)
database
administrators

Database and Application Architecture



(a) Two-tier architecture

(b) Three-tier architecture

Database Users and Administrators

- Database Users

- Database Administrator : A person who has such central control over the system is called a database administrator (DBA).

SQL

Overview

DML

SQL

DDL

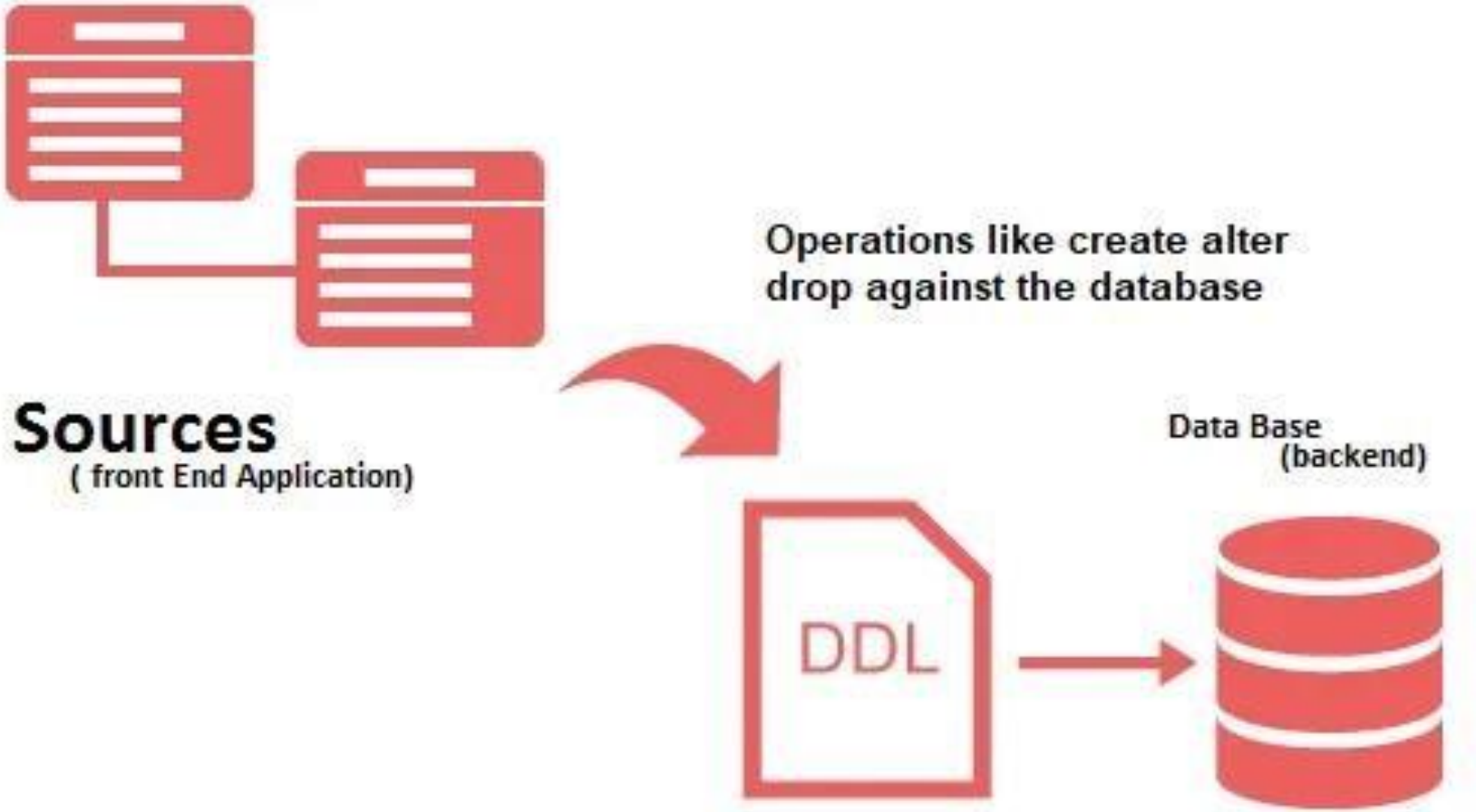
INTEGRITY

TRAN CONTROL

AUTHORIZATION

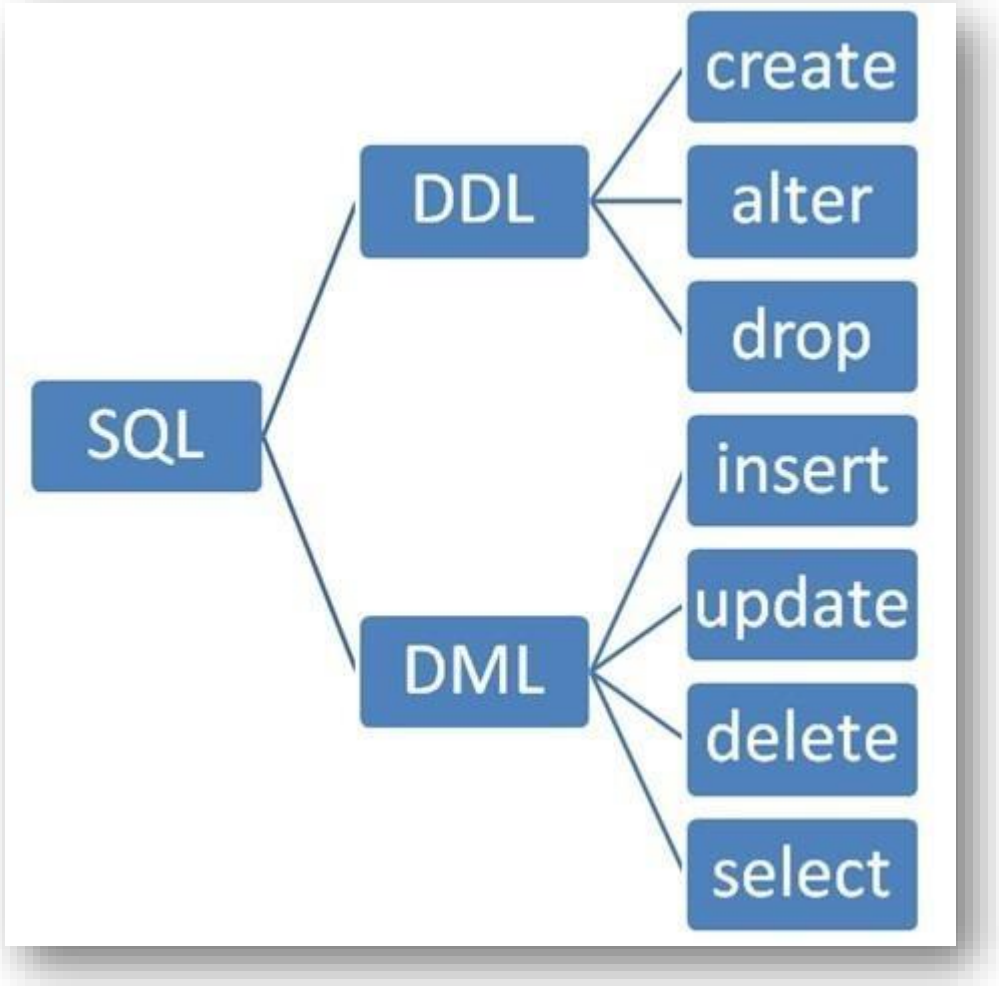
EMBEDDED/DYNAMIC

SQL Data Definition



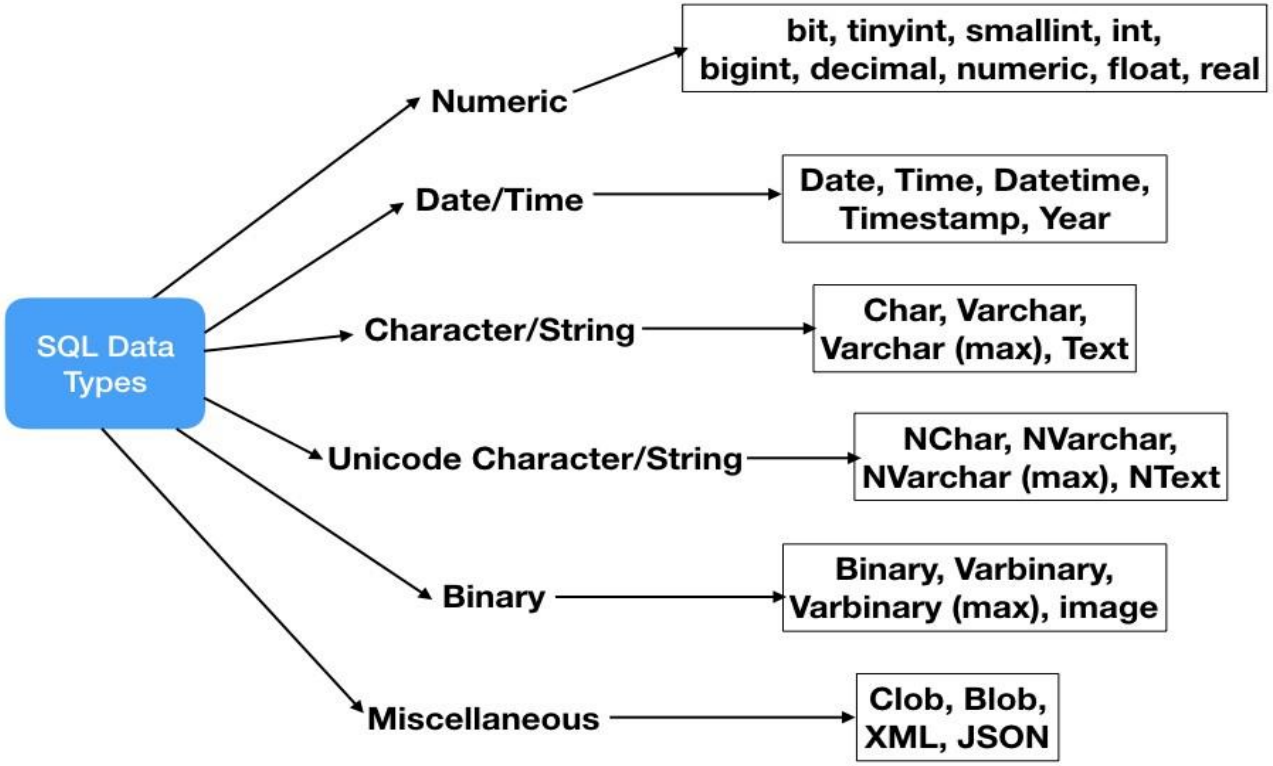
DDL vs DML

DDL	DML
SQL Commands that helps in defining database schemas	SQL Commands that helps to retrieve and manage data in relational databases.
Stands for Data Definition Language	Stands for Data Manipulation Language
Create, drop, alter are some DDL commands	Insert, update, delete and select are some commands.
Commands affect the entire database or the table	Commands affect one or more records in a table

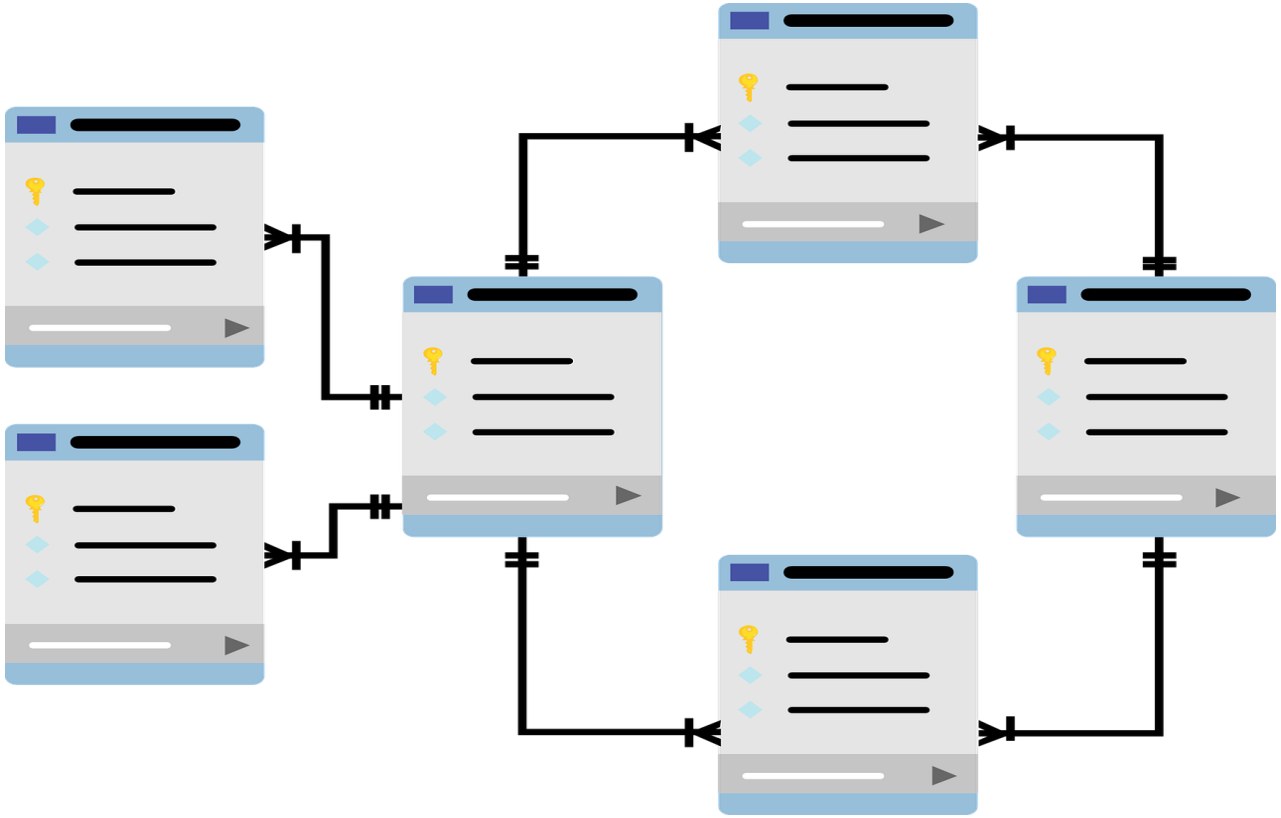


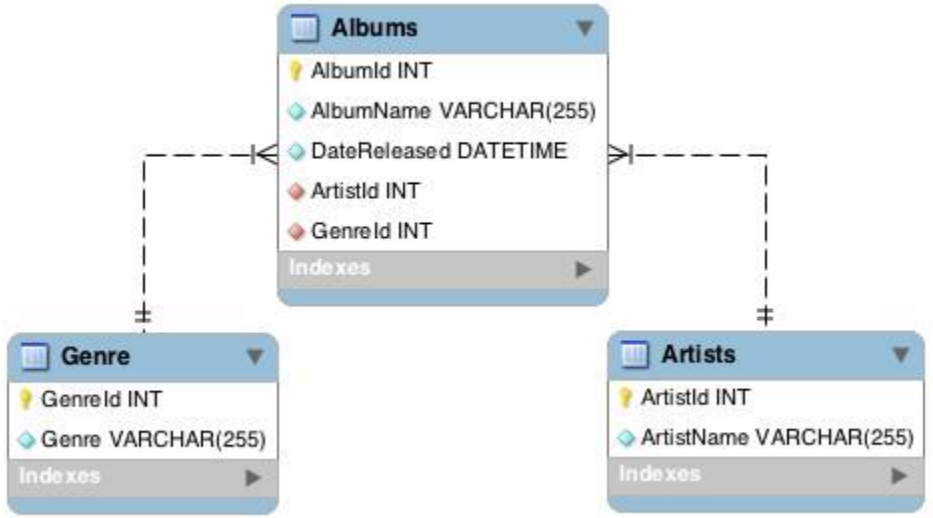
SQL Statements can not be rolled back	SQL statements can be rolled back

Data Types

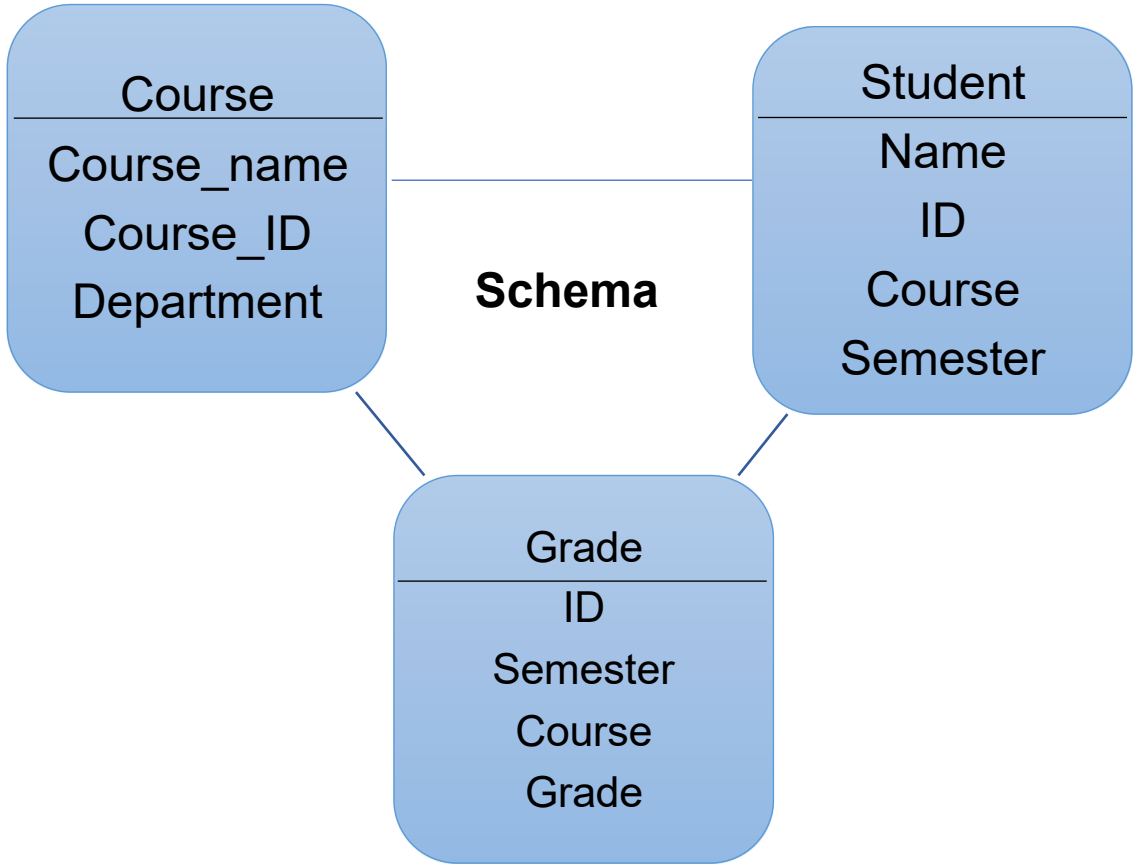

SQL Data Types
Numeric
bit, tinyint, smallint, int, bigint, decimal, numeric, float, real
Date/Time
Date, Time, Datetime, Timestamp, Year
Character/String
Char, Varchar, Varchar (max), Text
Unicode Character/String
NChar, NVarchar, NVarchar (max), NText
Binary
Binary, Varbinary, Varbinary (max), image
Miscellaneous
Clob, Blob, XML, JSON

Schema

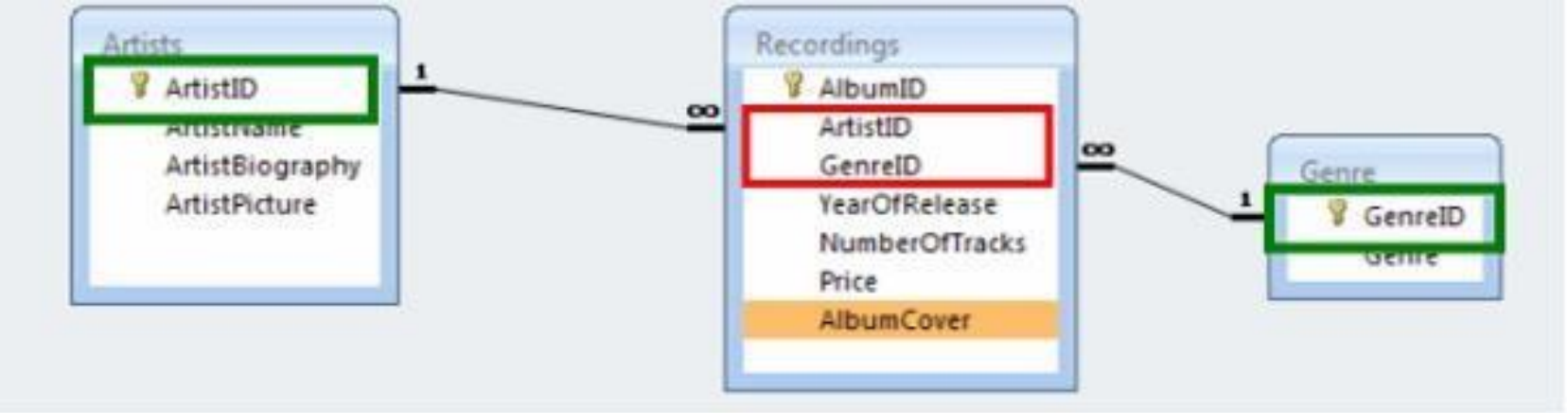



Albums
AlbumId INT
AlbumName VARCHAR(255)
DateReleased DATETIME
ArtistId INT
GenreId INT
Indexes
Genre
GenreId INT
Genre VARCHAR(255)
Indexes
Artists
ArtistId INT
ArtistName VARCHAR(255)
Indexes

Database Vs Schema


Course
Course_name
Course_ID
Department
Schema
Student
Name
ID
Course
Semester
Grade
ID
Semester
Course
Grade
Data Base

Keys


Artists
ArtistID
ArtistBiography
ArtistPicture
1
∞
Recordings
AlbumID
ArtistID
GenreID
YearOfRelease
NumberOfTracks
Price
AlbumCover
∞
1
Genre
GenreID
= primary key
= foreign key

Primary Key Vs Foreign Key

Primary Key	Foreign Key
A specific choice of a minimal set of attributes or columns that uniquely specify a tuple or a row in a table	A field or collection of fields in one table that uniquely identifies a row of another table or the same table
Related to a single table	Related to two tables
Value can not be null	Value can be null
Can not have duplicate values	Can have duplicate values
There can be a single primary key in a table	There can be a multiple foreign keys in a table

Used to identify the records of the table uniquely	Used to link two tables together.

Basic Structure of SQL Queries

- The basic structure of an SQL query consists of three clauses:
- select
- from
- where
- The ***select*** clause is used to list the attributes desired in the result of a query.
- The ***from*** clause is a list of the relations to be accessed in the evaluation of the query. The from clause by itself defines a Cartesian product of the relations listed in the clause.
- The ***where*** clause is a predicate involving attributes of the relation in the from clause.

select $A_1, A_2, \ldots, A_n$
from $r_1, r_2, \ldots, r_m$
where P;

Basic Structure of SQL Queries

SELECT

FROM

WHERE

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
33456	Gold	Physics	87000
45565	Katz	Comp. Sci.	75000
58583	Califieri	History	62000
76543	Singh	Finance	80000
76766	Crick	Biology	72000
83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000

Instructor

name
Srinivasan
Wu
Mozart
Einstein
El Said
Gold
Katz
Califieri
Singh
Crick
Brandt
Kim

Result of "select *name* from *instructor*".

SQL Queries

- **Queries on a Single Relation**

  Queries can be performed on a single relation / table.

```
select name
from instructor
where dept_name = 'Comp. Sci.' and salary > 70000;
```

- **Queries on Multiple Relations**

  Queries often need to access information from multiple relations.

```
select name, instructor.dept_name, building
from instructor, department
where instructor.dept_name= department.dept_name;
```

Queries On Single Relation

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
33456	Gold	Physics	87000
45565	Katz	Comp. Sci.	75000
58583	Califieri	History	62000
76543	Singh	Finance	80000
76766	Crick	Biology	72000
83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000

Instructor

select *dept_name*
from *instructor*;

dept_name
Comp. Sci.
Finance
Music
Physics
History
Physics
Comp. Sci.
History
Finance
Biology
Comp. Sci.
Elec. Eng.

Result of "**select** *dept_name* **from** *instructor*".

select distinct *dept_name*
from *instructor*;

dept_name
Comp. Sci.
Finance
Music
Physics
History
Physics
Comp. Sci.
History
Finance
Biology
Comp. Sci.
Elec. Eng.

Result of "**select** *dept_name* **from** *instructor*".

SQL Queries

```
for each tuple t1 in relation r1
        for each tuple t2 in relation r2
                ...
                for each tuple tm in relation rm
                        Concatenate t1, t2, ..., tm into a single tuple t
                        Add t into the result relation
```

In general, the meaning of an SQL query can be understood as follows:

1. Generate a Cartesian product of the relations listed in the from clause.
2. Apply the predicates specified in the where clause on the result of Step 1.
3. For each tuple in the result of Step 2, output the attributes (or results of expressions)specified in the select clause.

Additional Basic Operations

The additional basic operations that are supported in SQL are:

- The Rename Operation
- String Operations
- Attribute Specification in the Select Clause
- Ordering the Display of Tuples
- Where-Clause Predicates

The Rename Operation:

- The '***as'*** clause can appear in both the select and from clauses.

```
select name as instructor_name, course_id
from instructor, teaches
where instructor.ID= teaches.ID;
```

```
select T.name, S.course_id
from instructor as T, teaches as S
where T.ID= S.ID;
```

- Here ‘T’ and ‘S’ are declared as aliases

String Operations

- SQL specifies strings by enclosing them in single quotes.

  ‘*Scalable Databases*’

- A single quote character that is part of a string can be specified by using two single quote characters.

'It''s right'

- Functions on character strings.

Functions on character strings

- Concatenating (“||”)
- extracting substrings using ‘substring’ (string, start position, number of characters)
- length of strings (len())
- converting strings to uppercase (upper(s)) or lower case (lower(s))
- Removing spaces at the end of the string (using trim(s))
- Pattern matching ‘like’ and Pattern mismatching ‘not like’. We describe patterns by using two special characters:
- Percent (%): The % character matches any substring. • Underscore ( ): The character matches any character.
- 'Intro%' matches any string beginning with “Intro”.
- '%Comp%' matches any string containing “Comp” as a substring, for example,

- 'Intro. to Computer Science', and 'Computational Biology'.
- '___' matches any string of exactly three characters.
- '___ %' matches any string of at least three characters.

Ordering the Display of Tuples

- SQL offers the user some control over the order in which tuples in a relation are displayed.
- The ***order by*** clause causes the tuples in the result of a query to appear in sorted order.

```
select name
from instructor
where dept_name = 'Physics'
order by name;
```

```
select *
from instructor
order by salary desc, name asc;
```

Where-Clause Predicates

- SQL includes a ***between*** comparison operator to simplify where clauses that specify that a value be less than or equal to some value and greater than or equal to some other value.

```
select name
from instructor
where salary between 90000 and 100000;
```

- ***not between*** comparison operator.
- Tuple comparison **select** *name*, *course_id* **from** *instructor*, *teaches*
  where (*instructor.ID*, *dept_name*) = (*teaches.ID*, 'Biology');

Set Operations

- The SQL which correspond to the mathematical set operations ∪, ∩, and − are:
- The Union Operation

- The Intersect Operation
- The Except Operation
- Each of the above operations automatically eliminates duplicates.
- To retain all duplicates use the
- union all,
- intersect all
- except all.

Set Operations

- 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)

- Find courses that ran in Fall 2017 but not in Spring 2018

(**select** *course_id* **from** *section* **where** *sem* = 'Fall' **and** *year* = 2017) **except**
(**select** *course_id* **from** *section* **where** *sem* = 'Spring' **and** *year* = 2018)

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**

- The predicate **is not null** succeeds if the value on which it is applied is not null.

Null Values

- SQL treats as **unknown** the result of any comparison involving a null value (other than predicates **is null** and **is not null**).
- Example*: 5* < **null** or **null** <> **null** or **null = null**
- The predicate in a **where** clause can involve Boolean operations (**and**, **or**, **not**); thus the definitions of the Boolean operations need to be extended to deal with the value **unknown**.
- **and** : *(true* **and** *unknown)* = *unknown,* *(false* **and** *unknown)* = *false,*
  (unknown **and** *unknown)* = *unknown*
- **or:** (*unknown* **or** *true*) = *true*,
  (*unknown* **or** *false*) = *unknown*
  (*unknown* **or** *unknown)* = *unknown*
- Result of **where** clause predicate is treated as *false* if it evaluates to *unknown*

Aggregate Functions

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

- Average: avg
- Minimum: min
- Maximum: max
- Total: sum
- Count: count

- **Basic Aggregation**
- **Aggregation with Grouping**
- **The Having Clause**
- **Aggregation with Null and Boolean Values**

Basic Aggregation

- 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 in the *course* relation
- **select count** (*) **from** *course*;

Aggregation with Grouping

- Find the average salary of instructors in each department

- **select** *dept_name*, **avg** (*salary*) **as** *avg_salary* **from** *instructor* **group by** *dept_name*;

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

Aggregation with Grouping

- Attributes in **select** clause outside of aggregate functions must appear in **group by** list
- /* erroneous query */ **select** *dept_name*, *ID*, **avg** (*salary*) **from** *instructor* **group by** *dept_name*;

The Having Clause

- Find the names and average salaries of all departments whose average salary is greater than 42000

select *dept_name*, **avg** (*salary*) **as** *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

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.
- The nesting can be done in the following SQL query

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

as follows:

- **From clause:** r_i can be replaced by any valid subquery
- **Where clause:** P can be replaced with an expression of the form: B <operation> (subquery)
  B is an attribute and <operation> to be defined later.
- **Select clause:**
  A_i can be replaced be a subquery that generates a single value.

Set Membership

- 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);
```

- Find courses offered in Fall 2017 but not in Spring 2018

```
select distinct course_id
from section
where semester = 'Fall' and year= 2017 and
        course_id  not in (select course_id
                           from section
                           where semester = 'Spring' and year= 2018);
```

Set Membership

- Find names of instructors with salary greater than that of some (at least one) instructor in the Biology department.

```
select distinct T.name from
instructor as T, instructor as S
where T.salary > S.salary and S.dept name = 'Biology';
```

- Same query using > **some** clause

```
select name
from instructor
where salary > some (select salary
                     from instructor
                     where dept name = 'Biology');
```

Set Comparison –

"some" Clause

- Find names of instructors with salary greater than that of some (at least one) instructor in the Biology department.

```
select distinct T.name from
instructor as T, instructor as S
where T.salary > S.salary and S.dept name = 'Biology';
```

- Same query using > **some** clause

Set Comparison –

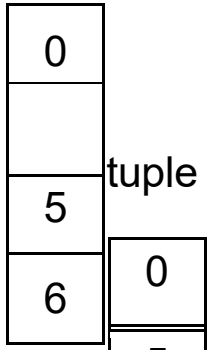
```
select name
from instructor
where salary > some (select salary from instructor
                     where dept name =
                     'Biology');
```

Set Comparison –

"some" Clause

(5 < **some**) = true

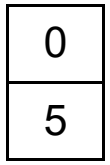
- F <comp> **some** *r* □ □ *t* □ *r* such (read: 5 < some tuple in the relation) that (F <comp> *t*)
  Where <comp> can be: □□ □□ □□ =□
  □



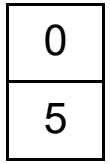
(5 < **some**) = false

Set Comparison –

(5 = **some**) = true



(5 □ **some**) = true (since 0 □ 5)



(= **some**) □ **in**

However, (□ **some**) □ **not in**

“all” Clause

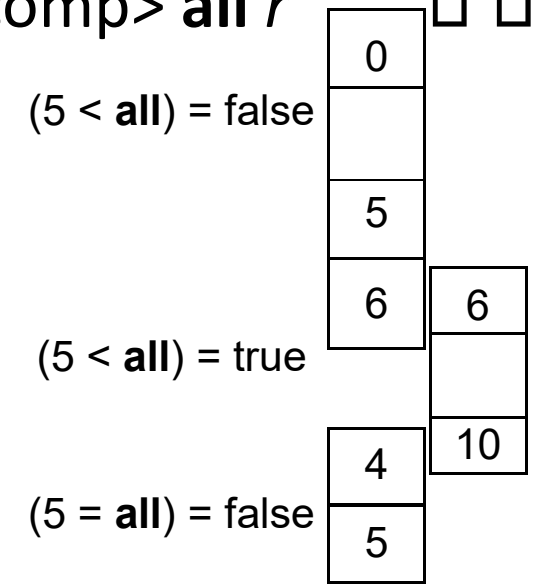
Find the names of all instructors whose salary is greater than the salary of all instructors in the Biology department.

Set Comparison –

```
select        name
from instructor
where salary > all (select salary from
                        instructor
                        where dept name = 'Biology');
```

Definition of “all” Clause

- F <comp> **all** *r* □ □ *t* □ *r* (F <comp> *t)*



(5 □ **all**) = true (since 5 □ 4 and

4
6

5 □ 6)

(□ **all**) □ **not in**

However, (= **all**) □ **in**

Test for Empty Relations

- The **exists** construct returns the value **true** if the argument subquery is nonempty.
- **exists** *r* □ *r* □ ∅
- **not exists** *r* □ *r* = ∅

Use of “exists” Clause

- Yet another way of specifying the query “Find all courses taught in both the Fall 2017 semester and in the Spring 2018 semester”

```
select      course_id
from section as S
where semester = 'Fall' and year = 2017 and
        exists  (select * from section as T
                 where  semester  =  'Spring'  and  year=  2018
                        and S.course_id = T.course_id);
```

- **Correlation name** – variable S in the outer query
- **Correlated subquery** – the inner query

Use of “not exists” Clause

- Find all students who have taken all courses offered in the Biology department.

```
select distinct S.ID, S.name
from student as S
where not exists ( (select course_id
                    from course where
                    dept_name = 'Biology')
                   except
                    (select T.course_id
                     from takes as T
                     where S.ID = T.ID));
```

- First nested query lists all courses offered in Biology
- Second nested query lists all courses a particular student took

- Note that X – Y = Ø □ X □ Y
- Note: Cannot write this query using = all and its variants

Test for Absence of Duplicate Tuples

- The **unique** construct tests whether a subquery has any duplicate tuples in its result.
- The **unique** construct evaluates to “true” if a given subquery contains no duplicates .
  - Find all courses that were offered at most once in 2017

```
select T.course_id
from course as T
where unique ( select R.course_id
               from section as R
               where  T.course_id=  R.course_id
                      and R.year = 2017);
```

Subqueries in the Form Clause

- SQL allows a subquery expression to be used in the **from** clause
- Find the average instructors' salaries of those departments where the average salary is greater than $42,000."

```
select dept_name, avg_salary
        from ( select dept_name, avg (salary) as avg_salary
                from instructor group by dept_name)
        where avg_salary > 42000;
```

- Note that we do not need to use the **having** clause
- Another way to write above query

```
 select dept_name, avg_salary
from ( select dept_name, avg (salary)
        from instructor group by
        dept_name)
         as dept_avg (dept_name, avg_salary)
```

where *avg_salary* > 42000;

With Clause

- The **with** clause provides a way of defining a temporary relation whose definition is available only to the query in which the **with** clause occurs.
- Find all departments with the maximum budget

```
with max_budget (value) as
        (select        max(budget)
         from department)
select department.name from
department, max_budget
where department.budget = max_budget.value;
```

Complex Queries using With Clause

- Find all departments where the total salary is greater than the average of the total salary at all departments

```
with dept _total (dept_name, value) as
        (select dept_name, sum(salary)
         from   instructor   group   by
         dept_name),
dept_total_avg(value) as
       (select   avg(value)
       from dept_total)
select       dept_name       from
dept_total, dept_total_avg
```

where *dept_total.value* > *dept_total_avg.value*;

Scalar Subquery

- Scalar subquery is one which is used where a single value is expected
- List all departments along with the number of instructors in each department **select** *dept_name*,

```
        ( select count(*) from
            instructor
            where department.dept_name = instructor.dept_name)
          as num_instructors
    from department;
```

- Runtime error if subquery returns more than one result tuple

Modification of the Database

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

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');

Deletion

- 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 used in SQL:
  1. First, compute **avg** (salary) and find all tuples to delete
  2. Next, delete all tuples found above (without recomputing **avg** or retesting the tuples)

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*);

Insertion

- 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.

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

- The **select from where** statement is evaluated fully before any of its results are inserted into the relation.

  Otherwise queries like **insert into** *table*1

  select * **from** *table*1 would cause problem

Updates

- Give a 5% salary raise to all instructors **update** *instructor*
  set *salary* = *salary* * 1.05
- 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 **update** *instructor*

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

Updates

- Increase salaries of instructors whose salary is over $100,000 by 3%, and all others by a 5% • 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)

Case Statement for Conditional Updates

- Same query as before but with case statement **update**

instructor

set *salary* = **case when** *salary* <= 100000 **then** *salary* * 1.05 **else** *salary* * 1.03 **end**

Updates with Scalar Subqueries

- Recompute and update tot_creds value for all students **update** *student S*

set *tot_cred* = (**select sum**(*credits*) **from** *takes, course* **where** *takes.course_id* = *course.course_id* **and**

*S.ID= takes.ID.***and**
takes.grade <> 'F' **and** *takes.grade* **is not null**);

- Sets *tot_creds* to null for students who have not taken any course
- Instead of **sum**(*credits*), use:

case when sum(*credits*) **is not null then**
sum(*credits*) **else** 0 **end**

END OF SESSION