Data Base Management System

Unit -2

Relational Model

- Main idea:
  - Table: relation
  - Column header: attribute
  - Row: tuple
- Relational schema: **name(attributes)**
  - Example: employee(ssno,name,salary)
- Attributes:
  - Each attribute has a domain – domain constraint
  - Each attribute is atomic: we cannot refer to or directly see a subpart of the value.

Components of Relational Model

- There are three components:
  1. A set of domains and a set of relations
  2. Integrity rules
  3. Operations on relations

- Characteristics of Relations
  1. Ordering of Tuple in Relation
  2. Ordering of values within a Tuple
  3. Values in Tuples : Atomic and NULL values
  4. Interpretation of a Relation

Relation Example

Account

AccountId	CustomerId	Balance
150	20	11,000
160	23	2,300
180	23	32,000

Customer

Id	Name	Addr
20	Tom	Irvine
23	Jane	LA
32	Jack	Riverside

- **Database schema** consists of
  - a set of relation schema
    - Account(AccountId, CustomerId, Balance)
    - Customer(Id, Name, Addr)
  - a set of constraints over the relation schema
    - AccountId, CustomerId must be integers
    - Name and Addr must be string of characters
    - CustomerId in Account must be from Ids in Customer

NULL value

Customer(Id, Name, Addr)

Id	Name	Addr
20	Tom	Irvine
23	Jane	LA
32	Jack	**NULL**

- Attributes can take a special value: NULL
  - Either not known: we don't know Jack's address
  - or does not exist: savings account 1001 does not have "overdraft"
- This is the single-value constrain on Attr: at most one
  - Either one: a string
  - Or zero: NULL

Why Constraints?

- Make tasks of application programmers easier:
  - If DBMS guarantees account >=0, then debit application programmers do not worry about overdrawn accounts.
- Enable us to identify redundancy in schemas:
  - Help in database design
  - E.g., if we know course names are unique, then we may not need another "course id" attribute
- Help the DBMS in query processing.
  - They can help the query optimizer choose a good execution plan

Domain Constraints

- Every attribute has a type:
  - integer, float, date, boolean, string, etc.
- An attribute can have a domain. E.g.:
  - Id > 0
  - Salary > 0
  - age < 100
  - City in {Irvine, LA, Riverside}
- An insertion can violate the domain constraint.
  - DBMS checks if insertion violates domain constraint and reject the insertion.

Integer	String	String
Id	Name	City
20	Tom	Irvine
23	Jane	San Diego
-2	Jack	Riverside

violations

Key Constraints

- **Superkey**: a set of attributes such that if two tuples agree on these attributes, they must agree on all the attributes
  - All attributes always form a superkey.
- Example:
  - AccountID forms a superkey, I.e., if two records agree on this attribute, then they must agree on other attributes
  - Notice that the relational model allow duplicates
  - Any superset of {Account} is also a superkey
  - There can be multiple superkeys
- Log: assume LogID is a superkey

Log(LogId, AccountId, Xact#, Time, Amount) — Illegal

LogID	AccountID	Xact#	Time	Amount
1001	111	4	1/12/02	$100
1001	122	4	12/28/01	$20
1003	333	6	9/1/00	$60

Keys

- Key:
  - Minimal superkey (no proper subset is a superkey)
  - If more than one key: choose one as a **primary key**
- Example:
  - Key 1: LogID (primary key)
  - Key 2: AccountId, Xact#
  - Superkeys: all supersets of the keys

Log(LogId, AccountId, Xact#, Time, Ammount)

LogID	AccountID	Xact#	Time	Amount
1001	111	4	1/12/02	$100
1002	122	4	12/28/01	$20
1003	333	6	9/1/00	$60

OK

Integrity Rules

There are two Integrity Rules that every relation should follow :

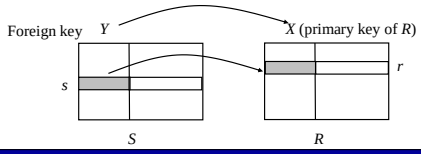
1. Entity Integrity (Rule 1)
2. Referential Integrity (Rule 2)

Entity Integrity states that –

If attribute A of a relation R is a prime attribute of R, then A can not accept null and duplicate values.

Referential Integrity Constraints

- Given two relations R and S, R has a primary key *X* (a set of attributes)
- A set of attributes Y is a *foreign key* of S if:
  - Attributes in Y have same domains as attributes X
  - For every tuple s in S, there exists a tuple r in R: s[Y] = r[X].
- A referential integrity constraint from attributes Y of S to R means that Y is a foreign key that refers to the primary key of R.
- The foreign key must be either equal to the primary key or be entirely null.



Examples of Referential Integrity

Account

AccountId	CustomerId	Balance
150	20	11,000
160	23	2,300
180	23	32,000

Customer

Id	Name	Addr
20	Tom	Irvine
23	Jane	LA
32	Jack	Riverside

Account.customerId to Customer.Id

Student

Id	Name	Dept
1111	Mike	ICS
2222	Harry	CE
3333	Ford	ICS

Dept

Name	chair
ICS	Tom
CE	Jane
MATH	Jack

Student.dept to Dept.name: every value of Student.dept must also be a value of Dept.name.

Relational Algebra

Relational Algebra is :

1. The formal description of how a relational database operates
2. An interface to the data stored in the database itself.
3. The mathematics which underpin SQL operations

The DBMS must take whatever SQL statements the user types in and translate them into relational algebra operations before applying them to the database.

Operators - Retrieval

There are two groups of operations:

1. Mathematical set theory based operations: UNION, INTERSECTION, DIFFERENCE, and CARTESIAN PRODUCT.
2. Special database oriented operations: SELECT , PROJECT and JOIN.

Symbolic Notation

- SELECT σ (sigma)
- PROJECT π (pi)
- PRODUCT × (times)
- JOIN ⋈ (bow-tie)
- UNION ∪ (cup)
- INTERSECTION ∩ (cap)
- DIFFERENCE - (minus)
- RENAME ρ (rho)

SET Operations - requirements

For set operations to function correctly the relations R and S must be union compatible. Two relations are union compatible if

- They have the same number of attributes
- The domain of each attribute in column order is the same in both R and S.

Set Operations - semantics

Consider two relations R and S.

- **UNION of R and S**
  the union of two relations is a relation that includes all the tuples that are either in R or in S or in both R and S. Duplicate tuples are eliminated.
- **INTERSECTION of R and S**
  the intersection of R and S is a relation that includes all tuples that are both in R and S.
- **DIFFERENCE of R and S**
  the difference of R and S is the relation that contains all the tuples that are in R but that are not in S.

Union ∪, Intersection ∩, Difference -

Set operators. Relations must have the same schema.

R(name, dept)

Name	Dept
Jack	Physics
Tom	ICS

S(name, dept)

Name	Dept
Jack	Physics
Mary	Math

R∪S

Name	Dept
Jack	Physics
Tom	ICS
Mary	Math

R ∩ S

Name	Dept
Jack	Physics

R-S

Name	Dept
Tom	ICS

Relational SELECT

SELECT is used to obtain a subset of the tuples of a relation that satisfy a *select condition*.

For example, find all employees born after 1st Jan 1950:

SELECT $_{dob > '01/JAN/1950'}$ (employee)

or

$\sigma_{dob > '01/JAN/1950'}$ (employee)

Conditions can be combined together using ^ (AND) and v (OR). For example, all employees in department 1 called `Smith':

$\sigma_{depno = 1 \ \wedge \ surname = `Smith`}$ (employee)

Selection σ

$\sigma_c(R)$: return tuples in R that satisfy condition *C*.

Emp (name, dept, salary)

Name	Dept	Salary
Jane	ICS	30K
Jack	Physics	30K
Tom	ICS	75K
Joe	Math	40K
Jack	Math	50K

$\sigma_{salary>35K}(Emp)$

Name	Dept	Salary
Tom	ICS	75K
Joe	Math	40K
Jack	Math	50K

$\sigma_{dept=ics \text{ and } salary<40K}(Emp)$

Name	Dept	Salary
Jane	ICS	30K

Relational PROJECT

The PROJECT operation is used to select a **subset of the attributes** of a relation by specifying the names of the required attributes.

For example, to get a list of all employees with their salary

PROJECT $_{ename, salary}$ (employee)

OR

$\pi_{ename, salary}$(employee)

Projection Π

$\Pi_{A1,\ldots,Ak}(R)$: pick columns of attributes A1,…,Ak of R.

Emp (name, dept, salary)

Name	Dept	Salary
Jane	ICS	30K
Jack	Physics	30K
Tom	ICS	75K
Joe	Math	40K
Jack	Math	50K

$\Pi_{name,dept}$ (Emp)

Name	Dept
Jane	ICS
Jack	Physics
Tom	ICS
Joe	Math
Jack	Math

Π_{name} (Emp)

Name
Jane
Jack
Tom
Joe

Duplicates ("Jack") eliminated.

CARTESIAN PRODUCT

The Cartesian Product is also an operator which works on two sets. It is sometimes called the CROSS PRODUCT or CROSS JOIN.

It combines the tuples of one relation with all the tuples of the other relation.

Cartesian Product: ×

R × S: pair each tuple r in R with each tuple s in S.

Emp (name, dept)

Name	Dept
Jack	Physics
Tom	ICS

Contact(name, addr)

Name	Addr
Jack	Irvine
Tom	LA
Mary	Riverside

Emp × Contact

E.name	Dept	C.Name	Addr
Jack	Physics	Jack	Irvine
Jack	Physics	Tom	LA
Jack	Physics	Mary	Riverside
Tom	ICS	Jack	Irvine
Tom	ICS	Tom	LA
Tom	ICS	Mary	Riverside

JOIN Example

- JOIN is used to combine related tuples from two relations R and S.
- In its simplest form the JOIN operator is just the cross product of the two relations and is represented as $(R \bowtie S)$.
- JOIN allows you to evaluate a join condition between the attributes of the relations on which the join is undertaken.

The notation used is

$$R \bowtie_{\text{Join Condition}} S$$

Join

$$R \bowtie_C S = \sigma_c(R \times S)$$

- Join condition C is of the form:

  <cond_1> AND <cond_2> AND … AND <cond_k>

  Each cond_i is of the form A *op* B, where:
  - A is an attribute of R, B is an attribute of S
  - *op* is a comparison operator: =, <, >, ≥, ≤, or ≠.
- Different types:
  - Theta-join
  - Equi-join
  - Natural join

Theta-Join

$$R \bowtie_{R.A>S.C} S$$

R(A,B)

A	B
3	4
5	7

S(C,D)

C	D
2	7
6	8

R × S

R.A	R.B	S.C	S.D
3	4	2	7
3	4	6	8
5	7	2	7
5	7	6	8

Result

R.A	R.B	S.C	S.D
3	4	2	7
5	7	2	7

Theta-Join

$R \bowtie_{R.A>S.C,\ R.B \neq S.D} S$

R(A,B)

A	B
3	4
5	7

S(C,D)

C	D
2	7
6	8

R × S

R.A	R.B	S.C	S.D
3	4	2	7
3	4	6	8
5	7	2	7
5	7	6	8

Result

R.A	R.B	S.C	S.D
3	4	2	7

Equi-Join

- Special kind of theta-join: C only uses the equality operator.

R(A,B)

A	B
3	4
5	7

S(C,D)

C	D
2	7
6	8

$R \bowtie_{R.B=S.D} S$

R × S

R.A	R.B	S.C	S.D
3	4	2	7
3	4	6	8
5	7	2	7
5	7	6	8

Result

R.A	R.B	S.C	S.D
5	7	2	7

Natural-Join ⋈

- Relations R and S. Let *L* be the union of their attributes.
- Let A1,…,Ak be their common attributes.

$$R \bowtie S = \Pi_L (R \bowtie_{R.A1=S.A1,\ldots,R.Ak=S.Ak} S)$$

Natural-Join ⋈

Emp (name, dept)

Name	Dept
Jack	Physics
Tom	ICS

Contact(name, addr)

Name	Addr
Jack	Irvine
Tom	LA
Mary	Riverside

Emp ⋈ Contact: all employee names, depts, and addresses.

Emp × Contact

Emp.name	Emp.Dept	Contact.name	Contact.addr
Jack	Physics	Jack	Irvine
Jack	Physics	Tom	LA
Jack	Physics	Mary	Riverside
Tom	ICS	Jack	Irvine
Tom	ICS	Tom	LA
Tom	ICS	Mary	Riverside

Result

Name	Dept	Addr
Jack	Physics	Irvine
Tom	ICS	LA

Outer Joins

- Motivation: "join" can lose information
- E.g.: natural join of R and S loses info about Mary, since the tuple does not join with other tuples.
  - Called "dangling tuples".

R

Name	Dept
Jack	Physics
Tom	ICS

S

Name	Addr
Jack	Irvine
Mike	LA
Mary	Riverside

- Outer join: natural join, but use NULL values to fill in dangling tuples.
- Three types: "left", "right", or "full"

Left Outer Join

R

Name	Dept
Jack	Physics
Tom	ICS

Name	Addr
Jack	Irvine
Mike	LA
Mary	Riverside

S

Left outer join

R ⟕ S

Name	Dept	Addr
Jack	Physics	Irvine
Tom	ICS	NULL

Pad null value for left dangling tuples.

Right Outer Join

R

Name	Dept
Jack	Physics
Tom	ICS

Name	Addr
Jack	Irvine
Mike	LA
Mary	Riverside

S

Right outer join

R ⟖ S

Name	Dept	Addr
Jack	Physics	Irvine
Mike	NULL	LA
Mary	NULL	Riverside

Pad null value for right dangling tuples.

Full Outer Join

R

Name	Dept
Jack	Physics
Tom	ICS

Name	Addr
Jack	Irvine
Mike	LA
Mary	Riverside

S

Full outer join

R ⟗ S

Name	Dept	Addr
Jack	Physics	Irvine
Tom	ICS	NULL
Mike	NULL	LA
Mary	NULL	Riverside

Pad null values for both left and right dangling tuples.

OUTER JOIN Example 1

R

ColA	ColB
A	1
B	2
D	3
F	4
E	5

R LEFT OUTER JOIN R.ColA = S.SColA S

A	1	A	1
D	3	D	3
E	5	E	4
B	2	-	-
F	4	-	-

S

SColA	SColB
A	1
C	2
D	3
E	4

R RIGHT OUTER JOIN R.ColA = S.SColA S

A	1	A	1
D	3	D	3
E	5	E	4
-	-	C	2

OUTER JOIN Example 2

R

ColA	ColB
A	1
B	2
D	3
F	4
E	5

S

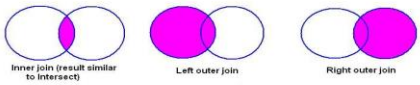
SColA	SColB
A	1
C	2
D	3
E	4

R FULL OUTER JOIN S R.ColA = S.SColA

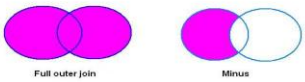
A	1	A	1
D	3	D	3
E	5	E	4
B	2	-	-
F	4	-	-
-	-	C	2

Joins Revised

JOINS AND SET OPERATIONS IN RELATIONAL DATABASES



Joins may be represented as Venn diagrams, as shown above along with other common set operations:



Result of applying these joins in a query:
INNER JOIN: Select only those rows that have values in common in the columns specified in the ON clause.
LEFT, RIGHT, or FULL OUTER JOIN: Select all rows from the table on the left (or right, or both) regardless of whether the other table has values in common and (usually) enter NULL where data is missing.

Combining Different Operations

- Construct general expressions using basic operations.
- Schema of each operation:
  - $\cup$, $\cap$, -: same as the schema of the two relations
  - Selection σ: same as the relation's schema
  - Projection Π: attributes in the projection
  - Cartesian product $\times$: attributes in two relations, use prefix to avoid confusion
  - Theta Join $\bowtie_C$: same as $\times$
  - Natural Join $\bowtie$: union of relations' attributes, merge common attributes
  - Renaming: new renamed attributes

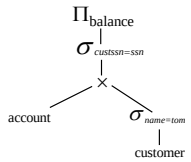
Example 1

customer(ssn, name, city)
account(custssn, balance)

"List account balances of Tom."

$$\Pi_{balance}(\sigma_{custssn=ssn}(account \times(\sigma_{name=tom} customer)))$$

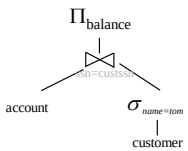
Tree representation



Example 1(cont)

customer(ssn, name, city)
account(custssn, balance)

"List account balances of Tom."



Division Operator

Not supported as a primitive operator, but useful for expressing queries like:

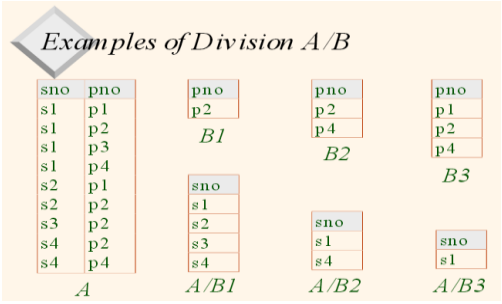
"Find sailors who have reserved all boats."

Let A have 2 fields, x and y; B have only field y:

A/B contains all x tuples (sailors) such that for every y tuple (boat) in B, there is an xy tuple in A Or : If the set of y values (boats) associated with an x value (sailor) in A contains all y values in B, the x value is in A/B.

- In general, x and y can be any lists of fields; y is the list of fields in B , and xy is the list of fields of A

Division Example



Example 1

STUDENT(Sno, Name, Major, Ddate)
COURSE(Cno, cname, Dept)
ENROLL(Sno, Cno, quarter, BISBN)
TEXT(BISBN, title, Publisher, Author)

1. List the names of courses taken by at least one student with quarter ='w99'
2. List any department that has all its books published by 'BPB'

Example 1 cont…

SQL Solution:

```
SELECT  dept FROM Course
WHERE cno IN (
(SELECT E.cno FROM Enroll E, Text T
WHERE E.BISBN = T.BISBN
AND T.Publisher = 'BPB')
MINUS
(SELECT E.cno FROM Enroll E, Text T
WHERE E.BISBN = T.BISBN
AND T.Publisher <>'BPB')
)
```

Example 2

EMPLOYEE(eno, name, age, dno, salary)

WORK_ON(pno, eno)

PROJECT(pno, pname, location)

1. Display the names of projects at “Delhi”
2. Find the project name of employee whose salary is greater than 10000.
3. Retrieve the name and eno of employee working on pno =100.

Example 3

SUPPLIER(sid, sname, saddr)

PARTS(pid, pname, color)

CATALOG(sid, pid, cost)

1. Find the name of all suppliers who supply yellow parts.
2. Find the name of suppliers who supply both blue and black parts.
3. Find supplier ids who supply all parts.
4. Find supplier ids who do not supply red parts.

Example 3 cont..

SQL Solution (2):

```
SELECT sname FROM Supplier S, Catalog C
WHERE  S. sid = C.sid  AND  C.sid =
(SELECT C.sid FROM Catalog C, Part P WHERE
   C.pid=P.pid and P.color= ‘Blue’)
INTERSECT
SELECT sname FROM Supplier S, Catalog C
WHERE  S. sid = C.sid  AND  C.sid =
(SELECT C.sid FROM Catalog C, Part P WHERE
   C.pid=P.pid and P.color= ‘Black’)
```

Example 3 cont..

SQL Solution (3):

```
SELECT sid FROM Catalog GROUP BY sid
HAVING count(*) = (SELECT count(pid) FROM Parts)
```

SQL Solution (4):

```
SELECT sid FROM Catalog C, Parts P
WHERE C.pid = P.id
AND P.color<>'Red'
```


Comparing RA and SQL

Relational algebra:

- is closed (the result of every expression is a relation)
- has a rigorous foundation
- has simple semantics
- is used for reasoning, query optimisation, etc.

SQL:

- is a superset of relational algebra
- has convenient formatting features, etc.
- provides aggregate functions
- has complicated semantics
- is an end-user language.

Introduction to SQL

Outline

- 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)
- ANSI and ISO standard SQL:
  - 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.
  - Not all examples here may work on your particular system.

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 each relation.
  - The physical storage structure of each relation on disk.

Domain Types in SQL

Data Type Syntax	Oracle 9i	Oracle 10g	Oracle 11g	Explanation
char(size)	Maximum size of 2000 bytes.	Maximum size of 2000 bytes.	Maximum size of 2000 bytes.	Where ***size*** is the number of characters to store. Fixed-length strings. Space padded.
varchar2(size)	Maximum size of 4000 bytes. Maximum size of 32KB in PLSQL.	Maximum size of 4000 bytes. Maximum size of 32KB in PLSQL.	Maximum size of 4000 bytes. Maximum size of 32KB in PLSQL.	Where ***size*** is the number of characters to store. Variable-length string.
long	Maximum size of 2GB.	Maximum size of 2GB.	Maximum size of 2GB.	Variable-length strings. (backward compatible)
raw	Maximum size of 2000 bytes.	Maximum size of 2000 bytes.	Maximum size of 2000 bytes.	Variable-length binary strings
long raw	Maximum size of 2GB.	Maximum size of 2GB.	Maximum size of 2GB.	Variable-length binary strings. (backward compatible)
number(p,s)	Precision can range from 1 to 38. Scale can range from -84 to 127.	Precision can range from 1 to 38. Scale can range from -84 to 127.	Precision can range from 1 to 38. Scale can range from -84 to 127.	Where ***p*** is the precision and ***s*** is the scale.
date	A date between Jan 1, 4712 BC and Dec 31, 9999 AD.	A date between Jan 1, 4712 BC and Dec 31, 9999 AD.	A date between Jan 1, 4712 BC and Dec 31, 9999 AD.	

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

- Example:

  create table *instructor* (
  ID **char**(5),
  name **varchar**(20)**,**
  dept_name **varchar**(20),
  salary **numeric**(8,2))

Integrity Constraints in Create Table

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

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);*

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

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

Updates to tables

- **Insert**
  - **insert into** *instructor* **values** ('10211', 'Smith', 'Biology', 66000);
- **Delete**
  - Remove all tuples from the *student* relation
    - **delete from** *student*
- **Drop Table**
  - **drop table** *r*
- **Alter**
  - **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 table** *r* **drop** *A*
    - where *A* is the name of an attribute of relation *r*
    - Dropping of attributes not supported by many databases.

Basic Query Structure

- 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 lists 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*
- NOTE: SQL names are case insensitive (i.e., you may use upper- or lower-case letters.)
  - E.g., *Name* ≡ *NAME* ≡ *name*
  - Some people use upper case wherever we use bold font.

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**.**
- Find the department names of all instructors, and remove duplicates
  select distinct *dept_name*
  from *instructor*
- The keyword **all** specifies that duplicates should not be removed.

  select all *dept_name*
  from *instructor*

The select Clause (Cont.)

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

  select *
  from *instructor*
- An attribute can be a literal with no **from** clause

  select '437'
  - Results is a table with one column and a single row with value "437"
  - Can give the column a name using:

    select '437' **as** *FOO*
- An attribute can be a literal with **from** clause

  select 'A'
  from *instructor*
  - Result is a table with one column and *N* rows (number of tuples in the *instructors* table), each row with value "A"

The select Clause (Cont.)

- The **select** clause can contain arithmetic expressions involving the operation, +, –, *, and /, and operating on constants or attributes of tuples.
  - 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.
  - Can rename "*salary/12"* using the **as** clause:

    select *ID, name, salary/12* **as** *monthly_salary*

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

  select *name*
  from *instructor*
  where *dept_name* = 'Comp. Sci.'
- Comparison results can be combined using the logical connectives **and, or,** and **not**
  - To find all instructors in Comp. Sci. dept with salary > 80000

    select *name*
    from *instructor*
    where *dept_name* = 'Comp. Sci.' **and** *salary* > 80000
- Comparisons can be applied to results of arithmetic expressions.

The from Clause

- 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.
  - For common attributes (e.g., *ID*), the attributes in the resulting table are renamed using the relation name (e.g., *instructor.ID*)
- Cartesian product not very useful directly, but useful combined with where-clause condition (selection operation in relational algebra).

Cartesian Product

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

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
...	...	...	...	...	...	...	...	...
...	...	...	...	...	...	...	...	...

Examples

- Find the names of all instructors who have taught some course and the course_id
  - **select** *name, course_id*
    from *instructor , teaches*
    where *instructor.ID = teaches.ID*
- Find the names of all instructors in the Art department who have taught some course and the course_id
  - **select** *name, course_id*
    from *instructor , teaches*
    where *instructor.ID = teaches.ID* ***and*** *instructor. dept_name* = 'Art'

The Rename Operation

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

  old-name **as** *new-name*

- Find the names of all instructors who have a higher salary than some instructor in 'Comp. Sci'.
  - **select distinct** *T.name*
    from *instructor* **as** *T, instructor* **as** *S*
    where *T.salary > S.salary* **and** *S.dept_name = 'Comp. Sci.'*

- Keyword **as** is optional and may be omitted

  instructor **as** *T* ≡ *instructor T*

Self Join Example

- Relation *emp-super*

person	*supervisor*
Bob	Alice
Mary	Susan
Alice	David
David	Mary

- Find the supervisor of "Bob"
- Find the supervisor of the supervisor of "Bob"
- Find ALL the supervisors (direct and indirect) of "Bob

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%'
- Match the string "100%"

  like '100 \%' **escape** '\'

  in that above we use backslash (\) as the escape character.

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.
  - '_ _ _ %' matches any string of at least three characters.
- SQL supports a variety of string operations such as
  - concatenation (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
```

- We may 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, ≥ $90,000 and ≤ $100,000)
  - **select** *name*
    from *instructor*
    where *salary* **between** 90000 **and** 100000
- Tuple comparison
  - **select** *name*, *course_id*
    from *instructor*, *teaches*
    where (*instructor.ID*, *dept_name*) = (*teaches.ID*, 'Biology');

Set Operations

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

  (**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 (Cont.)

- Find the salaries of all instructors that are less than the largest salary.
  - **select distinct** *T.salary*
    from *instructor* **as** *T, instructor* **as** *S*
    where *T.salary* < *S.salary*
- Find all the salaries of all instructors
  - **select distinct** *salary*
    from *instructor*
- Find the largest salary of all instructors.
  - (**select** "second query")
    except
    (**select** "first query")

Set Operations (Cont.)

- 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.**
- Suppose a tuple occurs *m* times in *r* and *n* times in *s,* then, it occurs:
  - *m + n* times in *r* **union all** *s*
  - min(*m,n)* times in *r* **intersect all** *s*
  - max(0, *m – n)* times in *r* **except all** *s*

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*;

Aggregate Functions – Group By

- 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 (Cont.)

- 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*;

Aggregate Functions – Having Clause

- 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

- Total all salaries

  select sum (*salary*)
  from *instructor*

  - Above statement ignores null amounts
  - Result is *null* if there is no non-null amount
- All aggregate operations except **count(*)** ignore tuples with null values on the aggregated attributes
- What if collection has only null values?
  - count returns 0
  - all other aggregates return null

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:
  - A_i can be replaced be a subquery that generates a single value.
  - r_i can be replaced by any valid subquery
  - P can be replaced with an expression of the form:

    B <operation> (subquery)

    Where B is an attribute and <operation> to be defined later.

Subqueries in the Where Clause

Subqueries in the Where Clause

- A common use of subqueries is to perform tests:
  - For set membership
  - For set comparisons
  - For set cardinality.

Set Membership

- Find courses offered in Fall 2009 and in Spring 2010

```
select distinct course_id
from section
where semester = 'Fall' and year= 2009 and
      course_id in (select course_id
                    from section
                    where semester = 'Spring' and year= 2010);
```

- Find courses offered in Fall 2009 but not in Spring 2010

```
select distinct course_id
from section
where semester = 'Fall' and year= 2009 and
      course_id  not in (select course_id
                         from section
                         where semester = 'Spring' and year= 2010);
```


Set Membership (Cont.)

- Find the total number of (distinct) students who have taken course sections taught by the instructor with *ID* 10101

```
select count (distinct ID)
from takes
where (course_id, sec_id, semester, year) in
                    (select course_id, sec_id, semester, year
                     from teaches
                     where teaches.ID= 10101);
```

- Note: Above query can be written in a much simpler manner. The formulation above is simply to illustrate SQL features.

Set Comparison – "some"

- 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 – "all" Clause

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

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


Test for Empty Relations

- The **exists** construct returns the value **true** if the argument subquery is nonempty.
- **exists** $r \Leftrightarrow r \neq \emptyset$
- **not exists** $r \Leftrightarrow r = \emptyset$

Use of "exists" Clause

- Yet another way of specifying the query "Find all courses taught in both the Fall 2009 semester and in the Spring 2010 semester"

```
select course_id
from section as S
where semester = 'Fall' and year = 2009 and
        exists (select *
                from section as T
                where semester = 'Spring' and year= 2010
                      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 $X - Y = \emptyset \Leftrightarrow X \subseteq Y$
- *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 2009

```
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 = 2009);
```

Subqueries in the From Clause

Subqueries in the From 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;
```


Subqueries in the Select Clause

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 (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 deposit, 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 (Cont.)

- Add all instructors to the *student* relation with tot_creds set to 0

  insert into *student*
  select *ID, name, dept_name, 0*
  from *instructor*

- 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

- 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
```


THANK YOU