**ITM 422 Review**

**Week 1: basic concepts**

Row 🡺 tuple table 🡺 entity column 🡺 attribute

**Data Manipulation Language (DML):** select, insert, update, delete, merge, explain plan, lock table

(not modify data)

**Data Definition Language(DDL):** create, alter, drop, rename, , truncate, grant, revoke, audit, noautit, comment

**Transaction control**: commit, rollback, savepoint, set transaction

**Session control**: alter session, set role

A session is the point from which you are connected to the database until you disconnect.

**System control**: alter system

Each database needs two tablespaces: **system** and **sysaux**

**Schema**: the logical structure of the database

**Instance**: the actual content of the database at a particular point in time

Normalization: 1st, 2nd, 3nd

**Super key**: a set of one or more attributes that allow us to uniquely identify a tuple in a relation.

**Candidate key**: minimal super key

**Primary key**: one of the candidate key

**Foreign key**: a relation may include in its attributes the primary key of another relation

**Built-in data type**: char(<size>), varchar2(<size>), number(<p>,<s>), date, sysdate, timestamp[<precision>],

**TIMESTAMP** ′2008-03-24 03:25:34.123 PM′

**Like**: %🡺 matches any character and any number of characters.

\_🡺 matches any single character

**Week 2: DDL(create a table)**

**Basic types of objects**: tables, indexes, constraints, sequences, synonyms(alias).

**Index**: B-tree and bitmap indexes.

**Schema structures**: materialized view, dimension, cluster, database links, triggers, java objects(java classes), PL/SQL programs(procedures, functions, packages)

NCHAR[(<size>)] 2000bytes, NVARCHAR@(<size>) 4000bytes: Unicode

CLOB 4GB; LONG 2GB

BINARY\_FLOAT: 32\_bit floating-point number, 5 bytes

BINARY\_DOUBLE: 64\_bit floating-point number, 9 bytes

FLOAT [(precision)]

**Binary datatype**: RAW(<size>) 2000 bytes; LONG RAW 2GB; BLOB 4GB; BFILE 4GB.

**Tablespace types**: permanent, undo, temporary.

Create table:

**create table** *instructor* (  
 *ID* **char**(5),  
 *name* **varchar2**(20) **not null,***dept\_name* **varchar2**(20),  
 *salary* **number**(8,2))

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

* **NOT NULL**
* **UNIQUE**

CREATE TABLE employee (  
dept\_no VARCHAR2 (2),  
emp\_id NUMBER (4),  
name VARCHAR2 (20) NOT NULL,  
ssn VARCHAR2 (11),  
salary NUMBER (9,2) CHECK (salary > 0),  
CONSTRAINT pk\_employee primary key (dept\_no, emp\_id),  
CONSTRAINT uq\_ssn unique (ssn))

* **CHECK**
  + CREATE TABLE bonus (  
    emp\_id VARCHAR2 (40) NOT NULL,  
    salary NUMBER (9,2),  
    bonus NUMBER (9,2),  
    CONSTRAINT ck\_bonus check (bonus > 0));  
    ALTER TABLE bonus  
    ADD CONSTRAINT ck\_bonus2 **CHECK** (bonus < salary);
* **PRIMARY KEY** (*A*1, ..., *An* )
* **FOREIGN KEY** (*A*m, ..., *An* ) **references** *r*

create table orders (

order\_id number **primary key**,

order\_dt date,

cust\_id **references** customers )

**Create table as select (CTAS) : only the NOT NULL constraints associated with the columns are copied to the new table.**

* + **alter table *r* add *A D***
    - **ALTER TABLE STAFF   
       ADD INSURANCE\_PROVIDER *Varchar2*(35);**
    - **Add a default value to a column**

**ALTER TABLE STAFF  
 MODIFY INSURANCE\_PROVIDER Varchar2(35) DEFAULT 'ABC Ins';**

* + - **Add two columns to a table and remove a constraint**

**ALTER TABLE STAFF**

**ADD (STAFF\_ID INT, PENSION\_ID INT)**

**DROP CONSTRAINT cons\_SO;**

* **When adding a new column, you cannot specify the NOT NULL constraint if the table already has rows. To add a NOT NULL column, you need to follow three steps:**
  1. **Modify the table to add the column.**
  2. **Update the column with values for all the existing rows.**
  3. **Add a NOT NULL constraint.**

EX: ALTER TABLE orders ADD updated\_by VARCHAR2 (30)  
DEFAULT 'JOHN' NOT NULL;

* **Renaming Columns** 
  + ALTER TABLE sample\_data  
    RENAME COLUMN data\_value to sample\_value;
* **Dropping Columns**
  + ALTER TABLE [<*schema*>.]<*table\_name*>  
    DROP {COLUMN <*column\_name*> | (<*column\_names*>)}  
    [CASCADE CONSTRAINTS]
* ALTER TABLE orders **SET UNUSED** COLUMN update\_dt;
  + **Will be dropped later**
* ALTER TABLE orders DROP UNUSED COLUMNS;
* 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*);

* Find the number of tuples in the *course* relation
  + **select count** (\*) **from** *course*;

**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

**Week 3: functions**

NULL function:

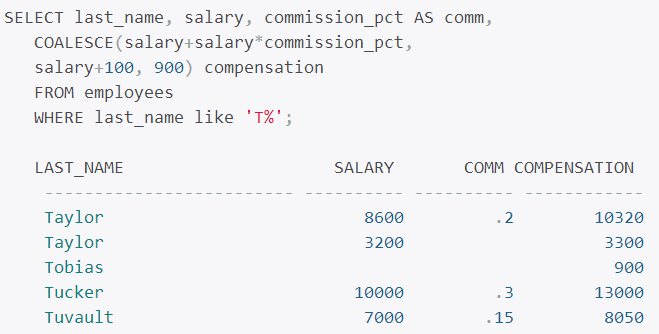
* NVL(x1, x2)

If x1 is null return x2 else return x1

* NVL2(x1, x2, x3)

If x1 is null return x3 else return x2.

* COALESCE(exp\_list)



Date:

* SQL> ALTER SESSION SET NLS\_DATE\_FORMAT='DD-Mon-YYYY HH24:MI:SS';  
    
  Session altered.  
    
  SQL> SELECT SYSDATE FROM dual;  
    
  SYSDATE  
  --------------------  
  31-Mar-2008 10:19:11
* **ALTER SESSION** command will set the *implicit conversion* mechanism to display date data in the format specified

**SYSTIMESTAMP: TIMESTAMP WITH TIME ZONE for the current database date and time (the time of the host server where the database resides).**

**Grouping function:**

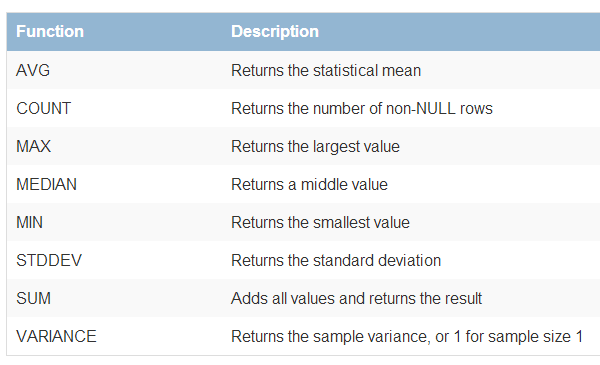
* **Group functions do not consider NULL values, except the COUNT(\*) and GROUPING functions.**
* **Use DISTINCT or ALL**

**Oracle defaults to all (counts non unique values)**

* **SELECT department\_id, COUNT(\*) "#Employees"  
  FROM employees  
  GROUP BY department\_id  
  ORDER BY 2 DESC, department\_id;**

**Group notes:**

* **If you have more than one column in the GROUP BY clause, Oracle creates groups within groups.**
* **The order of columns in the GROUP BY clause determines the grouping.**
* **Multiple columns in the GROUP BY clause are required when you have more than one non-aggregate column in the SELECT clause.**

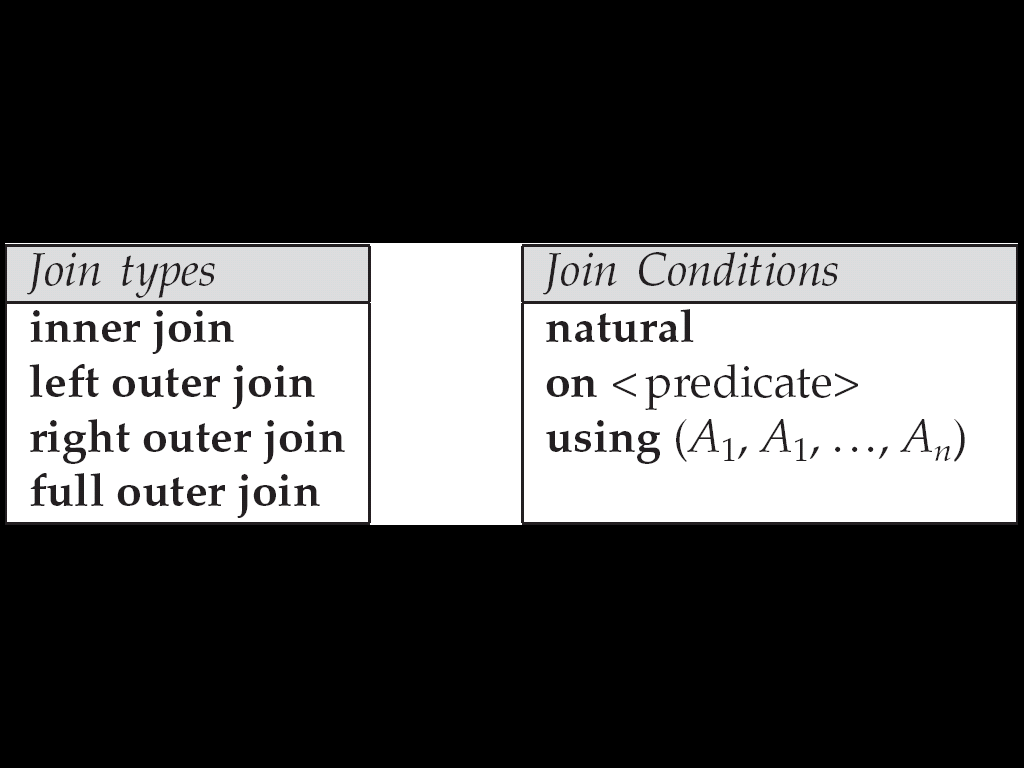
****

* **Group functions can be nested only one level. If you try to nest more than one level of nested group functions, you will encounter an error**

**Week 4: join**

**Oracle performs a join whenever multiple tables appear in the query's FROM clause.**

* **If multiple tables have the same column names, the duplicate column names should be qualified in the queries with their table name or table alias.**

****

**Join type:**

* **Cartesian Joins**

If you do not specify a join condition for the tables listed in the FROM clause, Oracle joins each row from the first table to every row in the second table

A join operation is a Cartesian product which requires that tuples in the two relations match (under some condition).

* + It also specifies the attributes that are present in the result of the join
* Oracle Syntax: SELECT region\_name, country\_name  
   FROM regions, countries  
   WHERE countries.country\_id LIKE ’J%’;
* ANSI Syntax: select --, --, -- from tbl\_1 ***cross join*** tbl\_2;
* **Inner Joins**

**Inner joins return only the rows that satisfy the join condition**

* + **Condition can use any of the comparison operations**
  + **When the “=“ sign is used the join called *equality join* or *equijoin***

SELECT locations.location\_id, city, department\_name  
FROM locations, departments  
WHERE locations.location\_id = departments.location\_id;

**ANSI Syntax: inner join**

<*table name*> NATURAL [INNER] JOIN <*table name*>  
  
<*table name*> [INNER] JOIN <*table name*> USING (<*columns*>)  
  
<*table name*> [INNER] JOIN <*table name*> ON <*condition*>

* **Outer Joins**

If tables A and B are outer-joined (FROM A, B) and you need all rows from B, the outer join operator is placed beside all columns of A.

* + Right Outer Join

***Course(main body)* left outer join *prereq***

***course* right outer join *prereq(main body)***

***course(main body)* natural full outer join *prereq(main body)***

* **Natural Join**

**Natural join matches tuples with the same values for all common attributes, and retains only one copy of each common column**

* + **select *name*, *course\_id*from *instructor, teaches*where *instructor.ID* = *teaches.ID*; 等于**
  + **select *name*, *course\_id*from *instructor* natural join *teaches*;**

**To prevent to join wrong attributes, you should use inner join or give some conditions**

* + Correct version
    - select *name*, *title*from *instructor* natural join *teaches*, *course*where *teaches*.*course\_id* = *course*.*course\_id*;
  + Another correct version
    - select *name*, *title*from (*instructor* natural join *teaches*)  
       join *course* using (*course\_id*);
* **Join…Using: If there are many columns that have the same names in the tables you are joining and they do not have the same data type, or you want to specify the columns that should be considered for an equijoin**

SELECT region\_name, country\_name, city  
FROM regions  
JOIN countries USING (region\_id)  
JOIN locations USING (country\_id)  
WHERE country\_id = 'US'  
ORDER BY 1;

* **Join…On: When you do not have common column names between tables to make a join or if you want to specify arbitrary join conditions, you can use the JOIN…ON syntax.**

SELECT region\_name, country\_name, city  
FROM regions r  
JOIN countries c ON r.region\_id = c.region\_id  
JOIN locations l ON c.country\_id = l.country\_id  
WHERE c.country\_id = 'US';

* **Self-joins**
  + **When performing self-joins in the ANSI syntax, you must always use the JOIN…ON syntax. You cannot use NATURAL JOIN and JOIN…USING**

SELECT e.last\_name Employee, m.last\_name Manager  
FROM employees e, employees m  
WHERE m.employee\_id = e.manager\_id;

* **Nonequality Joins**

SELECT last\_name, salary, grade  
FROM employees, grades  
WHERE last\_name LIKE 'R%'  
AND salary >= low\_salary  
AND salary <= NVL(high\_salary, salary);

ANSI Syntax

SELECT last\_name, salary, grade  
FROM employees JOIN grades  
ON salary >= low\_salary  
AND salary <= NVL(high\_salary, salary)  
WHERE last\_name LIKE 'R%';

**Week 4 sub-query**

**Sub query’s rule:**

* If columns in a sub query have the same name as columns in the containing statement, then you must **prefix** any reference to the column of the table from the containing statement with the table name or alias
  + To make your statements easier to read, always qualify the columns in a sub query with the name or alias of the table, view, or materialized view.
* The parent statement can be a **SELECT**, **UPDATE**, or **DELETE** statement in which the sub query is nested

**Use subqueries for the following purposes:**

* + To define the set of rows to be **inserted** into the **target table** of an INSERT or CREATE TABLE statement
  + To define the set of rows to be included in a **view** or materialized view in a CREATE VIEW or CREATE MATERIALIZED VIEW statement
  + To define one or more values to be assigned to existing rows in an **UPDATE** statement
  + To provide values for conditions in a WHERE clause, HAVING clause of SELECT, UPDATE, and DELETE statements
  + To define a table to be operated on by a containing query
  + You do this by placing the subquery in the **FROM** clause of the containing query as you would a table name.
* You may use subqueries in place of tables in this way as well in INSERT, UPDATE, and DELETE statements.

**Scalar sub query: aggregate function**

* + - **SELECT last\_name, department\_id, (SELECT MAX(salary)**

**FROM employees sq**

**WHERE sq.department\_id = e.department\_id)**

**FROM EMPLOYEE**

**WHERE first\_name = Tom;**

**Another name: inline view (from clause)**

* + you can nest any number of sub queries in an inline view (no limit)
  + SELECT \*

FROM **(**SELECT salary, department\_id

FROM employees

WHERE salary BETWEEN 1000 and 2000**)**;

**Another name: nested sub query (where clause)**

* + **255** levels of subqueries
  + **Correlated subquery**: When a column from the parent query is used in the subquery
* **regular** subquery,
  + the outer query depends on values provided by the inner query,
* **correlated** subquery
  + the inner query depends on values provided by the outer query.

**SELECT** department\_id, last\_name, salary

**FROM** employees **x**

**WHERE** salary > **(SELECT** AVG(salary)

**FROM** employees

**WHERE** **x**.department\_id = department\_id)

**ORDER BY** department\_id;

**Multiple Column Sub Query:**

SELECT first\_name, job\_id, salary

FROM emp\_history

WHERE **(salary, department\_id)** **in**

**(**SELECT **salary, department\_id**

FROM employees

WHERE salary BETWEEN 1000 and 2000 AND department\_id BETWEEN 10 and 20**)**

ORDER BY first\_name;

***Single-row sub queries:***

* SELECT first\_name, salary, department\_id

FROM employees

WHERE salary = **(**SELECT MIN (salary)

FROM employees**)**;

* SELECT last\_name, first\_name, salary  
  FROM employees  
  WHERE department\_id = **(**SELECT department\_id   
  FROM departments  
  WHERE department\_name = 'Accounting'**)**;
* SELECT department\_id, MIN (salary)

FROM employees

GROUP BY department\_id

HAVING MIN (salary) < **(SELECT AVG (salary)**

**FROM employees)**

**Multiple-row subqueries**: most commonly in WHERE and HAVING clauses.

* SELECT first\_name, department\_id

FROM employees

WHERE department\_id IN (SELECT department\_id

FROM departments

WHERE LOCATION\_ID = 1700);

**Week 5: relation algebra**

**mathematical Query Languages:**

* ***Relational Algebra*: More operational, very useful for representing execution plans.**

***Basic operations:***

* + ***Selection ( ) Selects a subset of rows from relation.***

*unary operation written as*

* + - *{\sigma\_{a \theta b}( R )*
    - *is a binary operation in the set*
    - *and are attributes*
    - *Relation r*
    - *σA=B ^ D > 5 (r)*

*Notation: σ p(r)*

*p is called the* ***selection predicate***

*Defined as:  
 σ****p(r) = {t | t*** *∈* ***r and p(t)}***

*Where p is a formula in propositional calculus consisting of* ***terms*** *connected by : ∧ (****and****), ∨ (****or****), ¬ (****not****)  
Each* ***term*** *is one of:*

*<attribute> op <attribute> or <constant>*

*where op is one of: =, ≠, >, ≥. <. ≤*

*Example of selection: σ dept\_name=“Physics”(instructor)*

* + ***Projection ( ) Deletes unwanted columns from relation.***

*a****projection****is a unary operation*

*Notation:*

*where A1, A2 are attribute names and r is a relation name.*

*The result is defined as the relation of k columns obtained by erasing the columns that are not listed*

*Duplicate rows removed from result, since relations are sets*

*Example: To eliminate the dept\_name attribute of instructor  
 ∏ID, name, salary (instructor)*

* + ***Rename operation***

*A****rename****is a unary operation written as   where:*

*the result is identical to R except that the b attribute in all tuples is renamed to an a attribute*

*\****rho****\_{a/b}(R)*

*Allows us to name, and therefore to refer to, the results of relational-algebra expressions.*

*Allows us to refer to a relation by more than one name.*

*Formally the semantics of the rename operator is defined as follows:*

*\rho_{a/b}(R) = \{ \ t[a/b] : t \in R \ \}*

***Rho*** *could also be used to* ***rename a relation*** *or the results of a relational expression which is a relation:*

*ρ x (E)*

*returns the expression E under the name X*

*If a relational-algebra expression E has* ***arity*** *n, then*

*returns the result of expression E under the name X, and with the*

*attributes renamed to A1 , A2 , …., An .*

* + ***Cross-product ( ) Allows us to combine two relations.***

*Notation r x s*

*Defined as:*

***r x s = {t q | t ∈ r and q ∈ s}***

*Assume that attributes of r(R) and s(S) are disjoint.*

*That is, R ∩ S = ∅*

*No common attributes between R and S*

*If attributes of r(R) and s(S) are not disjoint,* ***then renaming must be used.***

*R3 := R1 Χ R2*

*Pair each tuple t1 of R1 with each tuple t2 of R2.*

*Concatenation t1t2 is a tuple of R3.*

*Schema of R3 is the attributes of R1 and then R2, in order.*

*But beware attribute A of the same name in R1 and R2: use R1.A and R2.A.*

* + ***Set-difference ( ) Tuples in reln. 1, but not in reln. 2.***

*Notation r – s*

*Defined as:*

***r – s = {t | t ∈ r and t ∉ s}***

*Set differences must be taken between* ***compatible*** *relations.*

*r and s must have the same arity (union-compatible )*

*attribute domains of r and s must be compatible*

***Example****: to find all courses taught in the Fall 2009 semester, but not in the Spring 2010 semester  
 ∏course\_id (σ semester=“Fall” Λ year=2009 (section))* ***−***  *∏course\_id (σ semester=“Spring” Λ year=2010 (section))*

* + ***Union ( ) Tuples in reln. 1 and in reln. 2.***

*Notation: r ∪ s*

*Defined as:*

***r ∪ s = {t | t ∈ r or t ∈ s}***

*For r ∪ s to be valid.*

*1. r, s must have the same* ***arity*** *(same number of attributes)*

*union-compatible*

*2. The attribute domains must be* ***compatible*** *(example: 2nd column   
 of r deals with the same type of values as does the 2nd   
 column of s)*

***Example****: to find all courses ids taught in the Fall 2009 semester, or in the Spring 2010 semester, or in both  
 ∏course\_id (σ semester=“Fall” Λ year=2009 (section)) ∪   
 ∏course\_id (σ semester=“Spring” Λ year=2010 (section))*

*Union, intersection, and difference: the schemas of the two operands must be the same, so use that schema for the result.*

*For set union, set intersect and set difference, the two relations involved must be* ***union-compatible***

*that is, the two relations must have the same set of attributes.*

***Additional operations:***

* + ***Intersection, join, division, renaming: Not essential, but (very!) useful.***

*The operators take one or two relations as inputs and produce a new relation as a result.*

*For set union and set difference, the two relations involved must be* ***union-compatible****—that is, the two relations must have the same set of attributes.*

We define additional operations that do not add any power to the relational algebra, but that simplify common queries.

* **Set intersection**

Notation: *r* ∩ *s*

Defined as:

*r* ∩ *s* = { *t* | *t* ∈ *r* **and** *t* ∈ *s* }

Assume:

* + *r*, *s* have the *same arity*
  + attributes of *r* and *s* are compatible
  + Note: *r* ∩ *s* = *r* – (*r* – *s*)
* **Natural join**

Notation: r s

Let *r* and *s* be relations on schemas *R* and *S* respectively.   
Then, r s is a relation on schema *R* ∪ *S* obtained as follows:

Consider each pair of tuples *tr* from *r* and *ts* from *s*.

If *tr* and *ts* have the same value on each of the attributes in *R* ∩ *S*, add a tuple *t* to the result, where

*t* has the same value as *tr* on *r*

*t* has the same value as *ts* on *s*

Example:

*R* = (*A, B, C, D*)

*S* = (*E, B, D*)

Result schema = (*A, B, C, D, E*)

*r* *s* is defined as:  
 ∏*r.A, r.B, r.C, r.D, s.E* (σ*r.B = s.B* ∧ *r.D = s.D* (*r*  x *s*))

Natural join is associative

(*instructor teaches*) *course* is equivalent to  
*instructor* (*teaches course*)

Natural join is commutative

*instruct teaches* is equivalent to  
*teaches instructor*

* **Theta-Join**

R3 := R1 ⋈*C* R2

Take the product R1 Χ R2.

Then apply σ*C* to the result.

As for σ, *C* can be any boolean-valued condition.

The theta join operation *r θ s* is defined as

*r* θ*s*  = σθ (*r* x  *s)*

*Where* θ (theta) is one of the comparison operators:

{=, <, ≤, >, ≥, ≠}

* **Assignment**
* **Outer join**

An extension of the join operation that avoids loss of information.

Computes the join and then adds tuples form one relation that does not match tuples in the other relation to the result of the join.

Uses *null* values:

*null* signifies that the value is unknown or does not exist

All comparisons involving *null* are (roughly speaking) false by definition.

We shall study precise meaning of comparisons with nulls later

* ***Condition* Joins**



*Result schema* same as that of cross-product.

Fewer tuples than cross-product, might be able to compute more efficiently

Sometimes called a *theta-join*.

* **Other Joins**

***Equi-Join*:** A special case of condition join where the condition *c* contains only ***equalities*.**



*Result schema* similar to cross-product, but only one copy of fields for which equality is specified.

***Natural Join***: Equijoin on *all* common fields.

* **Extended Relational-Algebra-Operations**
* Generalized Projection

Extends the **projection** operation by allowing arithmetic functions to be used in the projection list.



*E* is any relational-algebra expression

Each of *F*1, *F*2, …, *Fn* are are arithmetic expressions involving constants and attributes in the schema of *E*.

Given relation *instructor(ID, name, dept\_name,* salary) where salary is annual salary, get the same information but with monthly salary

∏*ID, name, dept\_name,* ***salary/12*** *(instructor)*

* Aggregate Functions

**Aggregation function** takes a collection of values and returns a single value as a result.

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

**Aggregate operation** in relational algebra



*E* is any relational-algebra expression

* + *G1*, *G2* …, *Gn* is a list of attributes on which to group (can be empty)
  + Each *Fi* is an aggregate function
  + Each *Ai* is an attribute name

Note: Some books/articles use γ instead of (Calligraphic G)

* *Relational Calculus*: Lets users describe what they want, rather than how to compute it. (Non-operational, *declarative*.)
* A query is applied to *relation instances*, and the result of a query is also a relation instance.

**Schemas for Results**

* Union, intersection, and difference:
  + the schemas of the two operands must be the same, so use that schema for the result.

Selection:

* + schema of the result is the same as the schema of the operand.

Projection:

* + list of attributes tells us the schema.

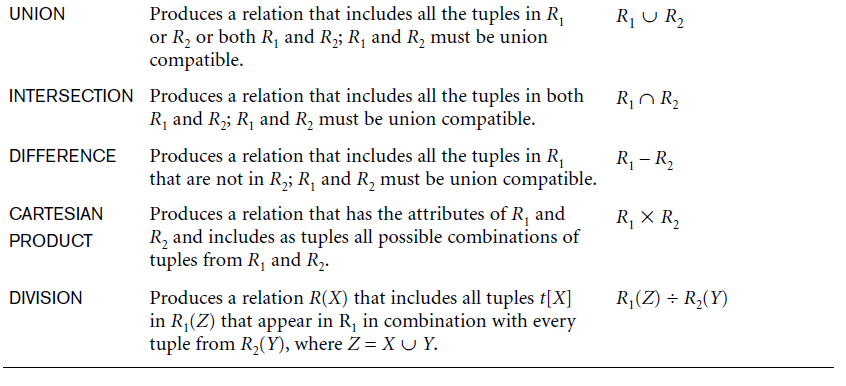
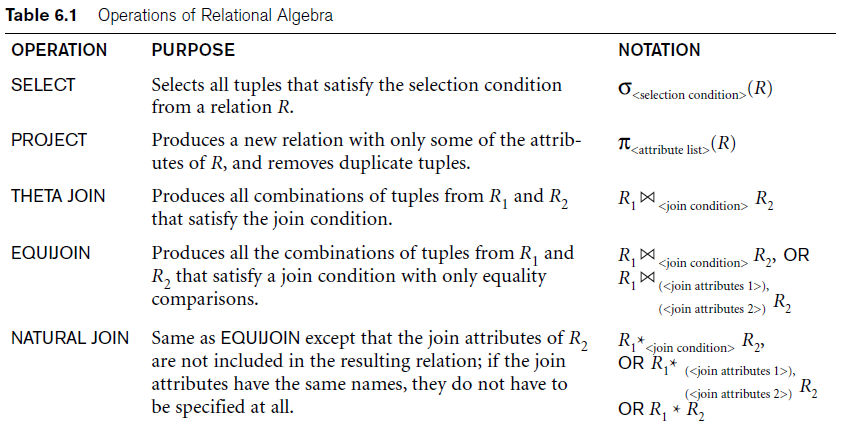
Product:

* + schema is the attributes of both relations.
    - Use R.*A*, etc., to distinguish two attributes named *A*.

Theta-join: same as product.

Natural join: union of the attributes of the two relations.

Renaming: the operator tells the schema.



.