Creating/Altering Tables

```
CREATE TABLE <TableName> (
        <columnName>        <columnType>, …,
        CONSTRAINT <constraintName> <constraintType> (<columnName(s)>), ….);
```

Column Types: Number(#), varchar2(#), date

Adding via Alter: ALTER TABLE <TableName> ADD <valid column or constraint>;

Constraint Examples

```
CONSTRAINT Table_PK PRIMARY KEY (keyColumn);
CONSTRAINT Table_FKCol_FK FOREIGN KEY (FKCol) REFERENCES OtherTable (key);
CONSTRAINT Table_UQ UNIQUE (uniqueColOne, uniqueColTwo);
CONSTRAINT Table_SetInVal CHECK (Set IN ('val1', 'val2'));
CONSTRAINT Table_ValNotNull CHECK (Val IS NOT NULL);
        Also: ALTER TABLE <TableName> MODIFY (<colName> NOT NULL);
CONSTRAINT Table_StringLikeVal CHECK (StringCol LIKE 'pattern');
CONSTRAINT Table_RangeVal CHECK (Range < upperLimit AND Range > lowerLimit);
```

Nulls

- 'Null' means UNKNOWN → causes undefined behavior in some operations
- Null groups into its own group
- Ignored in all aggregate functions except COUNT(*)
- NVL(exp1, exp2) = exp1 if exp1 is not null, else exp2

DML

```
INSERT INTO <TableName> [{<columName>}] VALUES ({<columnValues>});
DELETE FROM <TableName> WHERE <boolExp>;

UPDATE <TableName> SET <columnName> = <valueExpression> [WHERE <boolExp>];
```

Update on a condition:

```
        UPDATE Table SET col = 100 WHERE condition = 'yes';
```

Update from another table:

```
        UPDATE ATable A SET col = (
                SELECT B.value FROM BTABLE B WHERE A.id = B.id)
        WHERE id IN (SELECT B.id FROM BTABLE B);
```

```
SELECT [DISTINCT] _ FROM _ [WHERE _] [GROUP BY _ [HAVING _]] [ORDER BY _]
```

Execution Order: FROM/WHERE/GROUP BY/HAVING/ORDER BY/SELECT

Aggregation Operators: COUNT(*), COUNT(x), MIN(x), MAX(x), SUM(x), AVG(x)

Non-Aggregated columns in the SELECT clause must be in the GROUP BY clause

Renaming in the SELECT Clause: <colName or aggregation or expression> AS <newName>

Formatting Numbers: to_char(<number>, 990.99)

String concatenation: 'string1' || 'string2'

OPTIMIZATION OF RELATIONAL ALGEBRA

--> Do Selection first, then project, only join when you have to!

① JOIN v. CART. PRODUCT
— joins are better

$$\underline{\underline{R \bowtie_c S}} = \sigma_c(R \times S)$$

② PUSH SELECTION DOWN
⟹ DO SELECTION AS EARLY AS POSSIBLE
— it reduces the size of the data

$$\sigma_c(\pi_L(R)) \longleftrightarrow \underline{\underline{\pi_L(\sigma_c(R))}}$$

$$\sigma_{R.x=5}(R \bowtie_{R.a=S.b} S) \longleftrightarrow \underline{\underline{\sigma_{R.x=5}(R) \bowtie_{R.a=S.b} S}}$$

③ AVOID UNNESSARY JOINS

find customers having account balances below 100 and loans above 10000

$R1 \leftarrow \pi_{custName}(\text{Depositor} \bowtie \pi_{accountNum}(\sigma_{balance<100}(\text{Account})))$

$R2 \leftarrow \pi_{custName}(\text{Borrower} \bowtie \pi_{loanNum}(\sigma_{amount>10000}(\text{Loan})))$

$\text{Result} \leftarrow R1 \cap R2$

⟹ Much better than trying to join 4 tables!

RELATIONSHIPS AMONG OPERATORS

① JOIN ⟷ CARTESIAN PRODUCT + SELECT

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

② SELECTION is Commutative

$$\sigma_{c2}(\sigma_{c1}(R)) = \sigma_{c1}(\sigma_{c2}(R)) = \sigma_{c1 \text{ AND } c2}(R)$$

③ ORDER BETWEEN SELECTION & PROJECTION

$$\sigma_c(\pi_L(R)) \longrightarrow \pi_L(\sigma_c(R))$$

$\pi_L(\sigma_c(R)) \longrightarrow \sigma_c(\pi_L(R))$ ★ ONLY if L has all cols needed for C

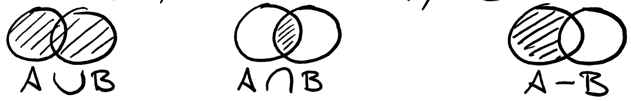
④ JOIN IS COMMUTATIVE

$$R \bowtie_c S = S \bowtie_c R$$

⑤ ORDER BETWEEN SELECTION & JOIN

$$\sigma_{c1}(R \bowtie_{c2} S) \longrightarrow (\sigma_{c1}(R) \bowtie_{c2} S)$$

① Set operations: UNION, INTERSECTION, DIFFERENCE



② PROJECTION $\pi_L(R)$: SELECT clause — chooses some columns to use
L is a list consisting of col name, col renames (eg A as B), or expressions (eg A+B as Z)

③ SELECTION $\sigma_C(R)$: WHERE clause — choose some tuples to use
C is a conditional to apply to each tuple in R

③ COMBINING TABLES: FROM clause - choose which tables to use & how to join

① CROSS-PRODUCT	$A \times B$:	pairs each $a \in A$ with each $b \in B$ ⟹ makes huge tables
② NATURAL JOIN	$A \bowtie B$:	pairs each matching attribute in matching colums
③ THETA JOIN	$A \bowtie_c B$:	pairs each $a \in A$ with each $b \in B$ if $C(a,b)$ holds true
④ OUTER JOIN	$A \overset{\circ}{\bowtie}_c B$:	pairs according to theta join, then pass the dangling tuples with ⊥

A:

X	Y
0	1
1	0

B:

X	C
1	0
1	1

$A \times B$:

AX	Y	BX	C
0	1	1	0
1	0	1	0
0	1	1	1
1	0	1	1

$A \bowtie B$:

X	Y	C
1	0	0
1	0	1

$A \bowtie_{Y=C} B$

AX	Y	BX	C
0	1	1	1

$A \overset{\circ}{\bowtie} B$:

X	Y	C
1	0	0
1	0	1
0	1	⊥

④ RENAMING: $\rho_{S(A_1, A_2, \ldots, A_n)}(R)$: AS operator

⑤ DUPLICATE ELIMINATION: $\delta(R)$: DISTINCT operator — returns R with one copy of each tuple in R
⟹ Turns a bag into a set

⑥ SORTING: $\tau_L(R)$: ORDER BY CLAUSE
— L is a list of fields to sort by, where ties are broken by fields later in the list

⑦ AGGREGATION & GROUPING: $\gamma_L(R)$: GROUP BY clause
— L is a list of GROUPING ATTRIBUTES (attributes ∈ R) and AGGREGATED ATTRIBUTES (operator applied to cols of R)

$\gamma_{activity,\ SUM(hours) \rightarrow timespent}(\text{HoursLog})$

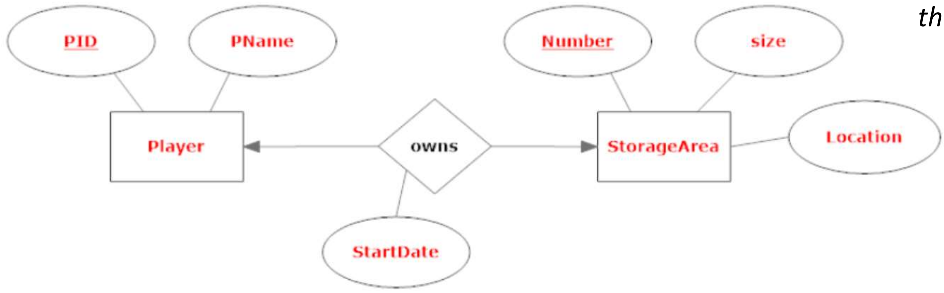
— executing $\gamma_L(R)$: ① Partition R into groups, where each group has tuples with a distinct assignment of the grouping attributes
→ if no grouping Attrs specified, R is one group

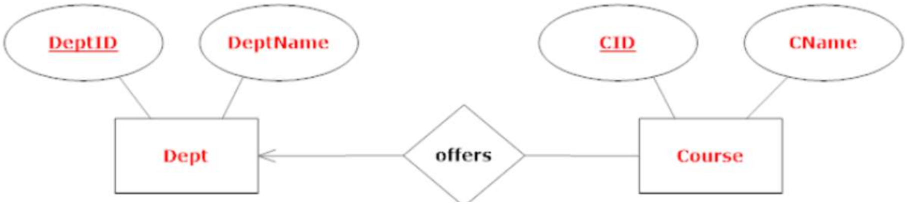
② for each group, produce one tuple consisting of:
— the grouping attributes for that group
— the aggregations over the tuples in that group

ERDs: Basic Rules

1:1 or 1:M (with no relationship keys) relationships become *part of one of the entity tables*

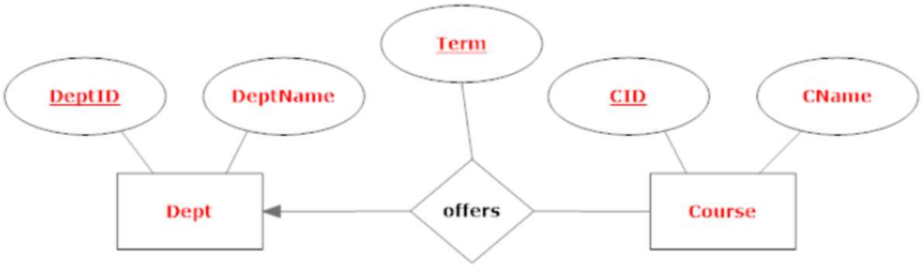
a. If 1:1, PK of on side is copied to the other as a FK
b. If 1:M, PK of the "one" side is copied to the "many" side as a FK
c. Any relationship attributes go on the side with the FK

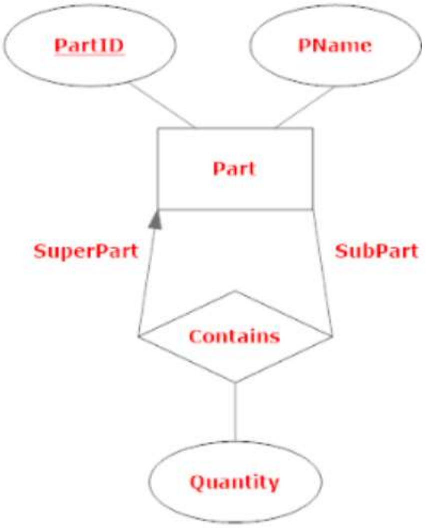




1:1 or 1:M (with relationship keys) relationships map to a *separate table*

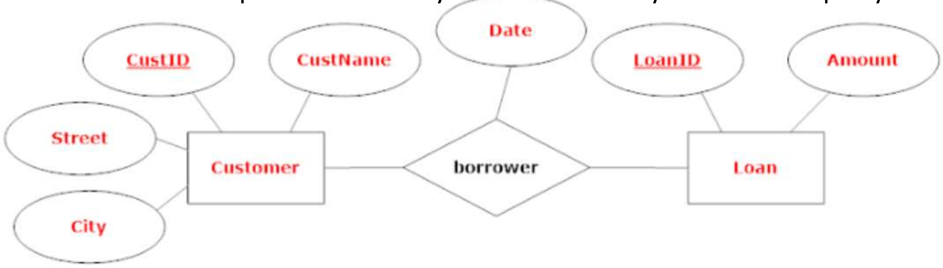
a. Relationship maps to a table with it's PK and Attributes, plus the PK from the "many" side
   i. Does NOT get the PK from the "one" side
b. "many" side has a FK that references the "one" side's PK
c. In a recursive relationship, the entity contains a FK to itself





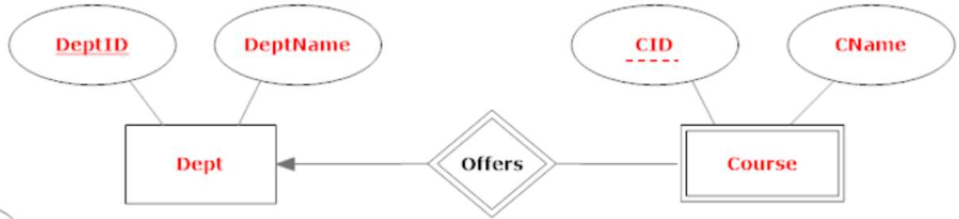
M:M or Multi-way relationships map to a *separate table*

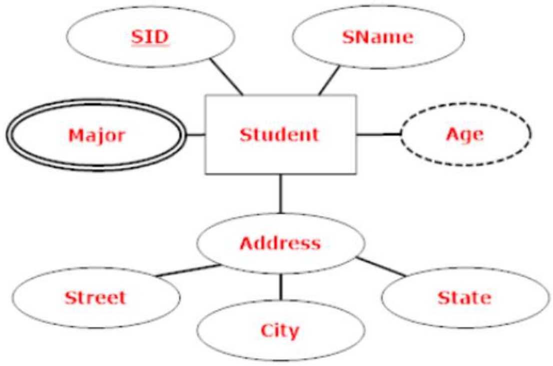
a. Relationship table's PK = Keys from each entity + Relationship keys



Weak Entity Sets become a *seperate table*

a. Contains all it's attributes and FKs to the PKs of identifying entity sets
b. PK = ID'ing Keys from the identifing Entity sets + Discriminator
c. The total, 1:M supporting relationship is not mapped





Composite Attributes only *use the 2nd level attributes* as columns
Derrived Attributes map *as-is* (and enforce via triggers)
Multi-Valued Attributes become a *separate table*
PK = Attribute itself + PK of main entity set

ERDs: IsA

A: Relation for each Entity Set

- Redundancy is inherint to the design (a student is in both the Students and People tables)
- Need multiple tables to complete a record for a student or employee

A1: Partial/Overlapping

Person (SSN, Name, DOB)

Employee (*SSN*, Department, Salary)

FK Employee(SSN) Ref Person(SSN)

Student (*SSN*, GPA, StartDate)

FK Students(SSN) Ref Person(SSN)

A2: Disjoint/Total

Person (SSN, Role, Name, DOB)

Unique (SSN, Role)

Role in {'Student', 'Employee'}

Employee (*SSN*, *Role*, Deparment, Salary)

Employee.Role = 'Employee'

Student (*SSN*, *Role*, GPA, StartDate

Role = 'Student'

B: One Big Table

- Full of nulls
- Less joins

B1: Partial/Overlapping

Person (SSN, Name, DOB, Department, Salary, GPA, StartDate)

B2: Disjoint/Total

- Need triggers to enforce attribute values based on role

Person (SSN, Name, DOB, Role, Department, Salary, GPA, StartDate)

Unique (SSN, Role)

Role in {'Student', 'Employee'}

C: Relations only for Specialization

- Cannot be used for Partial, but good for total
- If overlapping, need to update both tables on updates
- If disjoint, need triggers to ensure not in both tables
- Best for total disjoint

Employee (SSN, Name, DOB, Department, Salary)

Student (SSN, Name, DOB, GPA, StartDate

D: Relation for Every Combination

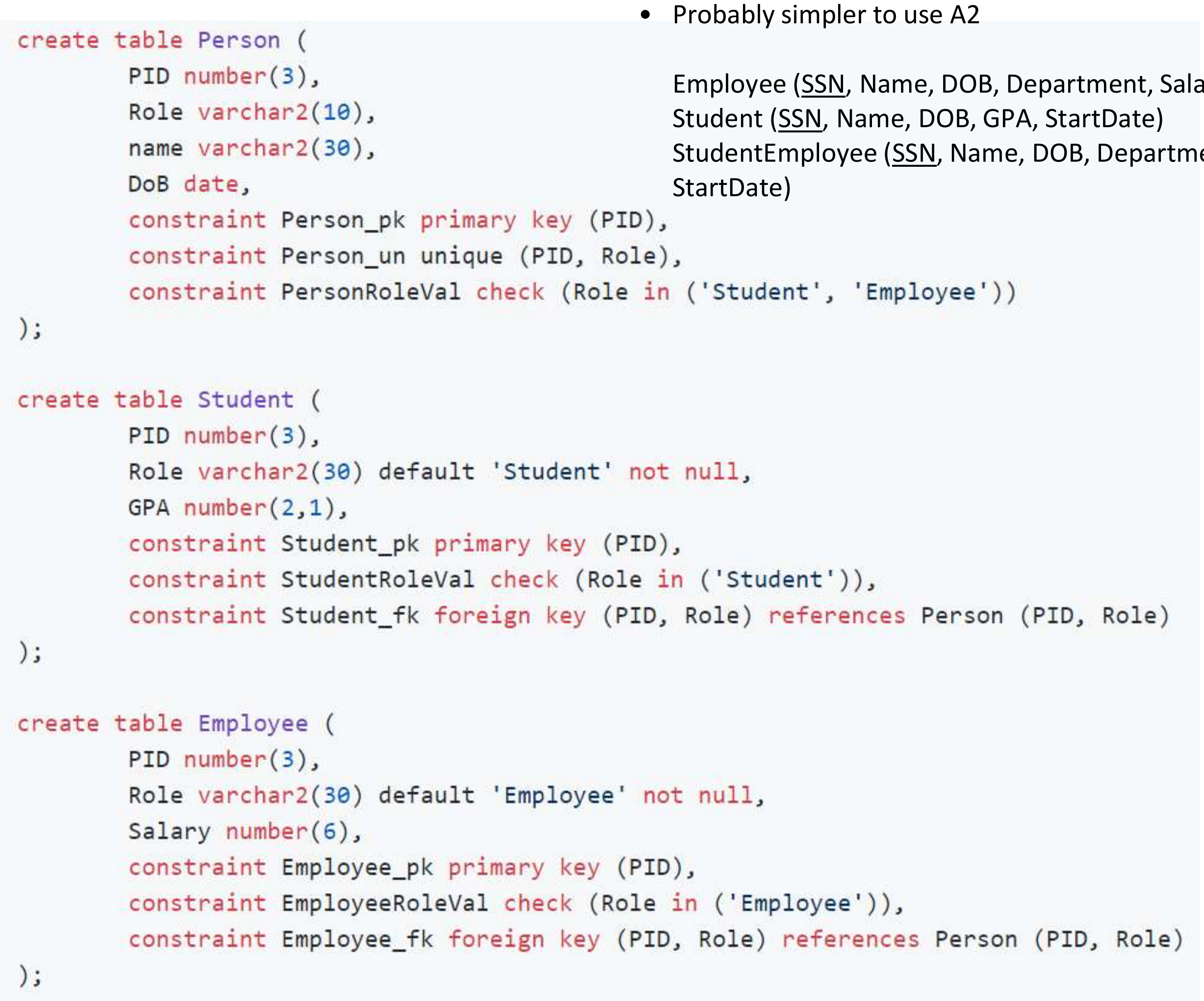
- Could get out of hand with lots of overlapping specializations
- Needs triggers to ensure a record is only in one table
- Good for overlapping relationships
- If Partial, need an additional table:

  Person (SSN, Name, DOB)
- Probably simpler to use A2

Employee (SSN, Name, DOB, Department, Salary)

Student (SSN, Name, DOB, GPA, StartDate)

StudentEmployee (SSN, Name, DOB, Department, Salary, GPA, StartDate)



```
create table Person (
        PID number(3),
        Role varchar2(10),
        name varchar2(30),
        DoB date,
        constraint Person_pk primary key (PID),
        constraint Person_un unique (PID, Role),
        constraint PersonRoleVal check (Role in ('Student', 'Employee'))
);

create table Student (
        PID number(3),
        Role varchar2(30) default 'Student' not null,
        GPA number(2,1),
        constraint Student_pk primary key (PID),
        constraint StudentRoleVal check (Role in ('Student')),
        constraint Student_fk foreign key (PID, Role) references Person (PID, Role)
);

create table Employee (
        PID number(3),
        Role varchar2(30) default 'Employee' not null,
        Salary number(6),
        constraint Employee_pk primary key (PID),
        constraint EmployeeRoleVal check (Role in ('Employee')),
        constraint Employee_fk foreign key (PID, Role) references Person (PID, Role)
);
```

Views

CREATE [OR REPLACE] VIEW <vName> AS <Query>;

Triggers

Exec Order:

1. BEFORE (Statement-Level);
2. ∀ affected records: (a) BEFORE (Row-Level); (b) Event (Row; (c) AFTER (Row);
3. AFTER (Statement);

Variables

- Declaration like normal but with a semicolon: <varName> <varType>;
- Can also declare Cursors: CURSOR <cName> IS <query>;
    - Parameterized Cursor: CURSOR <cName> (<param> <type>, …) IS…;
- Set variables from a table by SELECT <column[s]> INTO <varName[s]>...;
    - System Variables: SELECT sysdate INTO temp FROM Dual;

Code Body

- If statements are explicit: IF (<condition>) THEN <code> END IF;
- Looping through a cursor: FOR row IN cName LOOP <code> END LOOP;
- In Row-Level, may get :new and :old variables (depending on triggering operation)
- Output to console: dbms_output.put_line('message');
- Raise Error: RAISE_APPLICATION_ERROR(-20001, 'errMessage');

```
CREATE [OR REPLACE] TRIGGER <TriggerName>
[BEFORE | AFTER] [INSERT | UPDATE [OF <columnName>]] ON <TableName>
[FOR EACH ROW] -- "FOR EACH STATEMENT" implicit if omitted
[DECLARE
        <Decleration>; ...]
BEGIN
        <PL/SQL Code>
END; /
```

Procedures & Functions

- Procedures can't output except via output parameters
- Invoke procedures: EXEC <pName>(<params>); (May need to SET serveroutput on;)
- Invoke Functions anywhere with <fName>(<params>), including in WHERE clause
- Access variables declared with the name of procedure or function: <name>.<varName>

```
CREATE [OR REPLACE] [PROCEDURE | FUNCTION] <name>
        [(<paramName> [IN | OUT] <paramType>, …)]
        [RETURN <returnType] -- only if this is a function
        IS
        [<varDeclaration>;...]
BEGIN
        <PL/SQL Code>
END <pName>; /
```

```
import java.sql.*;
public class OTest {
    // These are for the Database you are connecting to
    private static final String USERID = "USERID"; // also set PASSWORD
    private static final String DB_SERVER = "jdbc:oracle:thin:@oracle.wpi.edu:1521:orcl"
public static void main(String[] args) {
    try { Class.forName("oracle.jdbc.driver.OracleDriver");
    } catch (ClassNotFoundException e) { // Driver not installed…
         e.printStackTrace(); return; }
    Connection conn = null;
    try { conn = DriverManager.getConnection(DB_SERVER, USERID, PASSWORD);
    } catch (SQLException e) { // Connection Failed…
        return; }
    try {
        Statement stmt = conn.createStatement(); // Basic way (How we did in class)
        String str = "SELECT * FROM TableName";
        ResultSet rset = stmt.executeQuery(str);
        // Process the results
        int custID = 0; String custName = ""; String city = ""; int age = 0;
        while (rset.next()) { // For each row that was returned…
            custID = rset.getInt("id"); // also getString, getDate, …
        }
        rset.close(); stmt.close(); // Close unneeded resources in this order

        Scanner reader = new Scanner(System.in); // Get User Input
        System.out.println("Enter parameter: ");
        String parameter = reader.nextLine();
        reader.close();
        // Using Prepared Statements (More secure)
        String selectTemplate = "SELECT colName FROM tName WHERE col = ?";
        PreparedStatement pstmt = conn.prepareStatement(selectTemplate);
        pstmt.setString(1, parameter); // Also setInt, setDate, ….
        ResultSet rset = pstmt.executeQuery(); // process rset, then close resources

        int numRowsAffected = stmt.executeUpdate(); // Inserting needs different exec

        conn.close(); // Close the connection
     } catch (SQLException e) { // Something was wrong with the SQL
         return;
     }
}
}
```