# Midterm 2

Monday, April 3, 2017 2:58 AM

*7.1 Using High Level Conceptual Models for DB Design*

- The first step in making a DB is to do requirements collection and analysis, which is where DB designers learn what the data requirements are for a DB. These will include the objects needed to be represented, and the functional requirements of the app.
- Then, the conceptual design is made, which is usually an ER model that satisfies all end users with its simplicity.
- Then, we have to design the DB with DBMS, like SQLite or Postgres.
- Finally, we have to physically design the DB, by choosing which internal storage structures and hardware we'll use for the DB is.
- A **miniworld** is the part of a real world object that we are describing in the DB.

*7.3 Entity Types, Entity Sets, Attributes, and Keys*

- The basic ER object is the entity, which is a thing in the real world with an independent existence.
- Every entity has attributes, the properties that describe it.
- A composite attribute is like a tree, it can be broken into subattributes. An address can have a city, state, road, ZIP and other things. Attributes that cannot be broken down into subattributes are atomic/simple.
- Attributes can be multivalued, with something like a human having more than 1 college degree. These have double circles.
- A derived attribute is one that uses the values of other attributes to determine its own value. A stored attribute is one that exists on its own.
- A NULL value is an attribute with no data at all.
- An entity set is a collection of entities that have the same values. An entity type is the definer for an entity set. An entity set is a rectangular box in an ER diagram, and describes the schema for a table in a DB.
- An attribute is stored as a circle in an ER D.
- A key is an attribute that is unique for every entity in an entity set. Keys can combine, and sometimes we check for uniqueness using multiple key attributes. A key is underlined, and if you have a multikey, make it a composite attribute.
- A weak entity is one with no key.
- A value set is the range of data that may be assigned to an attribute.

*7.4 Relationship Types, Relationship Sets, Roles, and Structural Constraints*

- When an attribute refers to another entity, it should be changed into a relationship, represented with a diamond connected to both entities.
- The degree of a relationship is the amount of participating entities. For binary relationships, it's easy to think of them as attributes whose value set is the entity set for another entity.
- We can describe relationships using two tools, cardinality ratios, and participation. The CR describes the maximum number of relationship instances that an entity can participate in. A 1 on the side of an entity means that the entity on the other side can only be related to one entity on the side with the 1, or, the entity on the other side can only relate to a single entity on the number side. A N on the side of the entity means that the other side can relate to as many entities on the number side as it wants.
- The participation constraint specifies whether the existence of an entity depends on being related to another entity via a relationship. There are two, total and partial. Total participation means that every entity in an entity set must be in a relationship. Things that must be totally participating have a double line between entity and relationship. Partial participation means that only some of the entity has to be in the relationship, and other entities can have a null or no relationship. This is drawn as a single line. A good way to think about it is that in total participation, if the entity does not have at least one relationship, it cannot exist.

*7.5 Weak Entity Types*

- A weak entity type is one that does not have a key attribute. It must relate with total participation to a strong entity type with a key, called the owner over the identifying relationship.
- A partial key is the attribute that identifies weak entities that belong to the same owner entity.

*8,1 Subclasses, Superclasses, and Inheritance*

- The EER model has everything the ER model has, but also has the concepts of subclass and superclass, with specializations and generalizations. A subclass belongs to a superclass, like a secretary is an employee. A subclass has type inheritance, which means it gets the attributes belonging to it, as well as all the attributes of the superclass.

*8.2 Specialization and Generalization*

- Specialization is the process of defining a set of subclasses for an entity type. A subset symbol defines the relationship. The curved end points at the subclass, the open at the superclass. A circle with a letter in it designated the specialization. The attributes of a subclass are called local attributes.
- Generalization is creating a superclass based off of shared attributes of many more specific entities.

*8.3 Constraints and Characteristics of Specialization and Generalization Hierarchies*

- A predicate defined subclass is one that is classified as a subclass based on a value in an attribute of the superclass. This value is written on the line from the circle to the subclass, with the attribute that value belongs to on the line from the super to the circle.
- The circle can have an o or a d. The d means that a single entity can only be one subclass, its disjoint. An o means overlapping, an entity can be more than one subclass.
- The completeness constraint determines if every entity in the super has to be a sub. Two lines from super to circle if true, one line if not.

*8.4 Union Types Using Categories*

- A union type is a subclass defined by multiple superclasses. It is a combination of all the attributes in multiple superclasses. The U symbol in a circle represents a union. A total union has the category (subclass of a union) with two lines from circle to category, and means that it has all of the attributes of its supers, while a partial has one line and has a subset of the attributes of its supers.

*9.1 Relational Database Design Using ER to Relational Mapping*

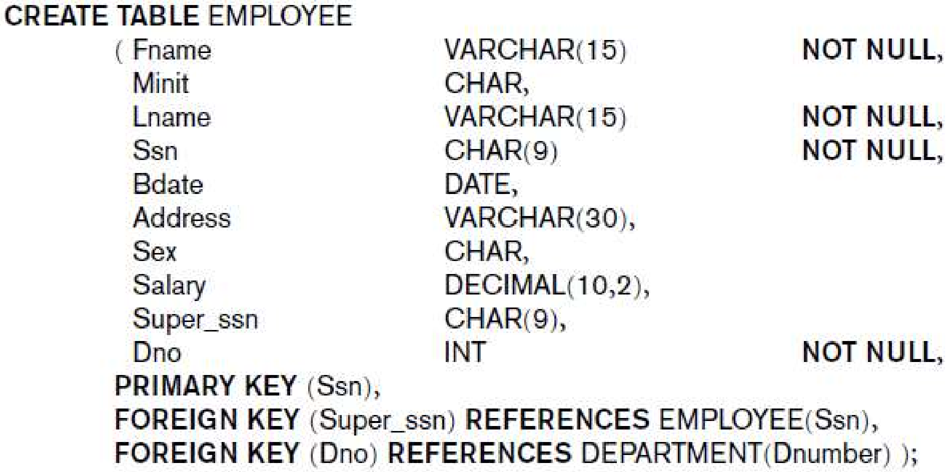
- This section describes how to create a database of relations based on your ER diagram.

1. For every strong entity, create a relation R that includes all of the simple attributes for that entity. Choose one of the key attributes as the primary key. If you choose a composite as your primary key, make the set of those attributes the primary key. We also remember in the relation that secondary keys exist as well.
2. For every weak entity, create a relation R and include all of the simple attributes of the weak entity as attributes of R. For R's foreign keys, make the primary key attributes of the relations that correspond to the strong owner entities the foreign keys. If a weak entity owns another weak entity, the owner weak entity should be mapped first. The primary key of the new relation is the set of foreign keys.
3. For a 1:1 Relationship, you can do one of three things:
   a. Foreign Key Approach: Choose one of the entities participating in the relationship, preferably the one with total participation if there is only one of those, and add the primary key of the other participating entity as the foreign key for the one with total participation, or just the initial entity you chose.
   b. Merge Relation Approach: Just make both of the entities one relation. You can do this when both participations are total.
   c. Create a relationship relation/lookup table with the tuples of the table having the primary keys of both entities.
4. For every 1:N relationship type, grab the entity type S that participates at the N side of the relationship, and give that relation a foreign key that is the primary key of the entity on the 1 side of the relation.
5. For every relationship type M:N, create a relation R with foreign keys that represent the primary keys of the two entities in the relationship. The combination of these foreign keys creates the primary key of R.
6. For every multivalued attribute, create a new relation, and assign the subattributes the attribute has in the ER diagram. Then, give it the foreign key that’s defined as the primary key of the entity it belongs to.
7. For a n-ary relationship, create a new relation with its foreign keys being the primary key of all participating entities.

**CHAPTER 4: BASIC SQL**

*4.1 SQL Data Definition and Data Types*

- A table in SQL is a relation in the relational model, a row is a tuple, and a column is an attribute.

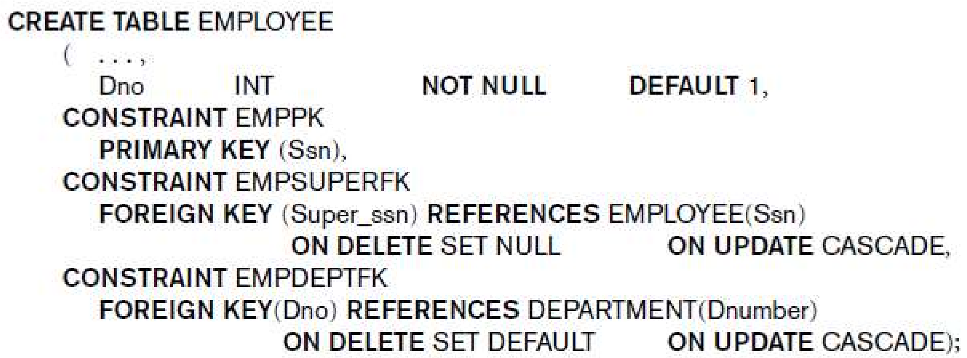
- An SQL schema is identified by a schema name, and includes an authorization ID to indicate the user or account who owns the schema, and descriptors for each element in the schema.
- Schema elements are tables, constraints, views, domains and other things that can be used to describe the schema.
- To create a schema, use the CREATE SCHEMA command. After "SCHEMA", you write the name in all uppercase letters, and you can use "AUTHORIZATION" to assign an auth ID.
- SQL statements end with semicolons.
- A catalog is a named collection of schemas in a SQL environment. An environment is an installation like SQLite or Postgres.
- A special schema exists in every catalog called the INFORMATION_SCHEMA, that describes every schema in the catalog and all the element descriptions in these schemas.
- CREATE TABLE SchemaName.TableName will create a new relation with the name TableName.
- When using the CREATE TABLE command, we have to specify the attributes and their initial constraints, like NOT NULL.
- The relations declared through this command are called **base tables**, meaning that the table and rows are created and stored as a file in a DBMS. **Virtual relations** are created by the CREATE VIEW command, and do not actually correspond to an actual physical file in all cases.
- When specifying attributes in CREATE TABLE, the order they are specified is considered their order in the table.
- 

```
CREATE TABLE EMPLOYEE
        ( Fname             VARCHAR(15)          NOT NULL,
          Minit             CHAR,
          Lname             VARCHAR(15)          NOT NULL,
          Ssn               CHAR(9)              NOT NULL,
          Bdate             DATE,
          Address           VARCHAR(30),
          Sex               CHAR,
          Salary            DECIMAL(10,2),
          Super_ssn         CHAR(9),
          Dno               INT                  NOT NULL,
        PRIMARY KEY (Ssn),
        FOREIGN KEY (Super_ssn) REFERENCES EMPLOYEE(Ssn),
        FOREIGN KEY (Dno) REFERENCES DEPARTMENT(Dnumber) );
```

- Above is an example of a create table statement. We usually leave foreign keys to be added once all the tables have been created, so we do not cause any errors. These are applied with ALTER TABLE.
- The attributes in SQL have basic data types, like INTEGER, INT, SMALLINT for integers, FLOAT, REAL, DOUBLE PRECISION for floats, DECIMAL,NUMERIC for formatted numbers. Character-string data types (Strings) can be fixed length, CHAR(size) CHARACTER(size) or varying length VARCHAR(maxsize). Strings are placed in single quotes. Fixed length strings are padded with white space to the right, but whitespace is ignored in comparison of fixed length strings. CLOB(file size) specifies long strings, with KB, MB, or GB as the size.
- Bit string data types are either fixed BIT(max num of bits) or varying BIT VARYING(max num of bits). We write bits like B'00001'. BLOB(file size) is a large binary value like an image.
- Boolean data types have TRUE or FALSE values.
- The DATE data type has ten positions, with components being YEAR, MONTH, DAY in the form YYYY-MM-DD. The TIME data type has 8 positions, with components HOUR, MINUTE, and SECOND in the form HH:MM:SS.
- A timestamp data type TIMESTAMP includes the DATE and TIME fields and a minimum of six positions for decimal fractions of seconds.
- We can create domains for attributes types by doing CREATE DOMAIN domainName AS base data value.

*4.2 Specifying Constraints in SQL*

- Since SQL allows NULL as an attribute value, a constraint like NOT NULL can be specified on an attribute as a constraint. This is implicitly done on the primary key of each relation.
- The DEFAULT constraint assigns a default value to a row attribute.
- The CHECK constrain will only allow values for an attribute that fulfill the statement within the CHECK. It can be used with CREATE DOMAIN to further restrict a custom data value.

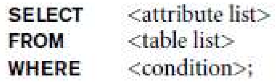


```
CREATE TABLE EMPLOYEE
    (   . . . ,
        Dno          INT              NOT NULL        DEFAULT 1,
    CONSTRAINT EMPPK
        PRIMARY KEY (Ssn),
    CONSTRAINT EMPSUPERFK
        FOREIGN KEY (Super_ssn) REFERENCES EMPLOYEE(Ssn)
                          ON DELETE SET NULL           ON UPDATE CASCADE,
    CONSTRAINT EMPDEPTFK
        FOREIGN KEY(Dno) REFERENCES DEPARTMENT(Dnumber)
                          ON DELETE SET DEFAULT        ON UPDATE CASCADE);
```

- The above image shows the DEFAULT constraint assigning every Dno to initially be 1.
- We can use constraints to specify primary keys, by using PRIMARY KEY where a constraint would be for an attribute. This is only possible for one primary key tables.
- The UNIQUE constrain specifies secondary/alternate keys, and ensures that only one row has that attribute value.
- When an attribute value is modified such that it creates an integrity violation, we block the update. This is done to make sure FOREIGN KEYS are not corrupted. We can attach a referential triggered action to a foreign key, so that if a foreign key is deleted, the attribute in the table will automatically get a new value. These can be specified with ON DELETE or ON UPDATE, and can take the actions SET NULL, SET DEFAULT, or CASCADE. CASCADE will take the new value of the foreign attribute and assign it to the foreign key.
- Constraints can get a constraint name, shown in the image above.

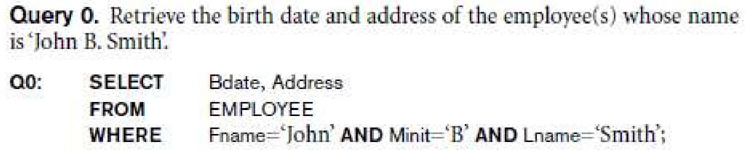
*4.3 Basic Retrieval Queries in SQL*

- Since a SQL table can have duplicate rows, we refer to tables as a multiset of tuples. Some SQL tables can be constrained to be sets, however due to a key constraint.
- A basic form of a SELECT statement is called a mapping or a select-from-where block.



```
SELECT    <attribute list>
FROM      <table list>
WHERE     <condition>;
```

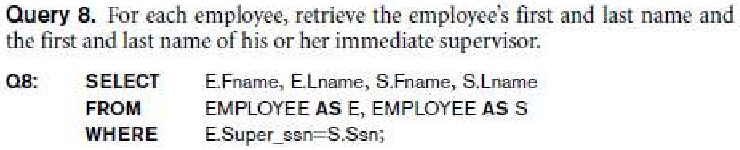
- Conditions can have =,<,>,<=,>=,>=,<>.
- An example SELECT:



**Query 0.** Retrieve the birth date and address of the employee(s) whose name is 'John B. Smith'.

```
Q0:    SELECT    Bdate, Address
       FROM      EMPLOYEE
       WHERE     Fname='John' AND Minit='B' AND Lname='Smith';
```

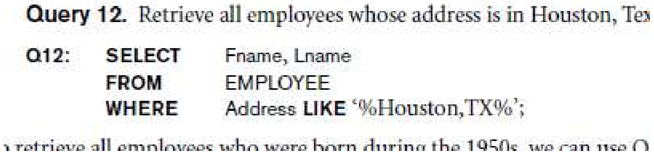
- A selection condition selects a particular tuple from a table, and a join condition combines two rows from two or more tables.
- If the same name is used for 2 attributes in different tables, we have to qualify the attribute name with the relation name, like TABLE.ATTRIBUTE.
- We can use the AS keyword in FROM clauses to **alias** tables to new names, and even assign the attribute names new names.



**Query 8.** For each employee, retrieve the employee's first and last name and the first and last name of his or her immediate supervisor.

```
Q8:    SELECT    E.Fname, E.Lname, S.Fname, S.Lname
       FROM      EMPLOYEE AS E, EMPLOYEE AS S
       WHERE     E.Super_ssn=S.Ssn;
```

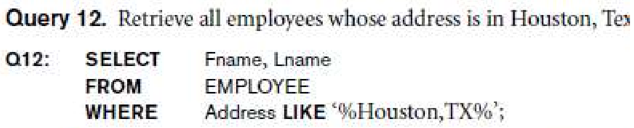
- E and S in this case can be thought of as two different copies of the table, and we join both.
- If we are missing the WHERE clause, we select all tuples from a database. If more than one table is specified, we retrieve the CROSS PRODUCT of the two tables, which is kinda useless.
- We can use SELECT * to retrieve all attributes from a table.
- We can use SELECT DISTINCT to specify we want a set of rows, not a multiset.
- The UNION, EXCEPT, and INTERSECT operations can be used between SELECTS to create a new result, the union, difference, and intersection of the tables as sets respectively. ALL can be added after these operations to not eliminate duplicates within the rows.
- In the WHERE clause, we can use LIKE to search for values on an attribute. An underscore is like a single character, and a % represents an arbitrary number of zero or more characters.



**Query 12.** Retrieve all employees whose address is in Houston, Tex

```
Q12:   SELECT    Fname, Lname
       FROM      EMPLOYEE
       WHERE     Address LIKE '%Houston,TX%';
```

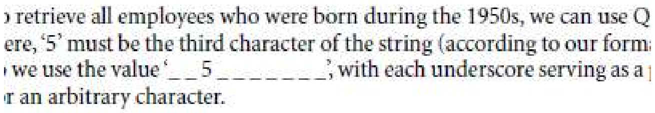
retrieve all employees who were born during the 1950s, we can use Q



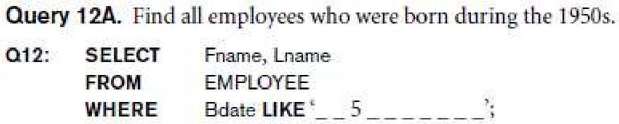
**Query 12.** Retrieve all employees whose address is in Houston, Tex

```
Q12:  SELECT  Fname, Lname
      FROM    EMPLOYEE
      WHERE   Address LIKE '%Houston,TX%';
```

-



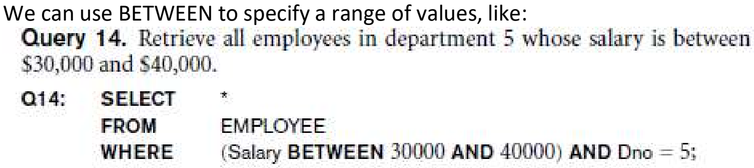
› retrieve all employees who were born during the 1950s, we can use Q
ere, '5' must be the third character of the string (according to our form
› we use the value '_ _ 5 _ _ _ _ _ _ _', with each underscore serving as a
r an arbitrary character.



**Query 12A.** Find all employees who were born during the 1950s.

```
Q12:  SELECT  Fname, Lname
      FROM    EMPLOYEE
      WHERE   Bdate LIKE '_ _ 5 _ _ _ _ _ _ _';
```

- We can use BETWEEN to specify a range of values, like:



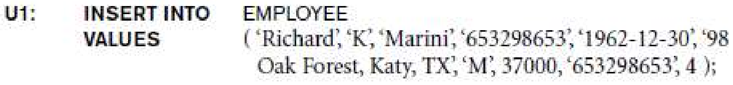
**Query 14.** Retrieve all employees in department 5 whose salary is between $30,000 and $40,000.

```
Q14:  SELECT  *
      FROM    EMPLOYEE
      WHERE   (Salary BETWEEN 30000 AND 40000) AND Dno = 5;
```

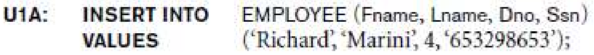
- We can use ORDER BY after WHERE to specify the order of the rows in the table returned by SELECT. DESC and ASC specify descending and ascending order.

*4.4 INSERT, DELETE, UPDATE Statements*

- INSERT can be used to add a single tuple to a relation. The values specified for the tuple should be in the same order as they appear in the table. We can also specify the attributes we want to insert into.

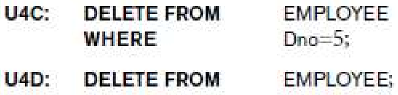


```
U1:   INSERT INTO  EMPLOYEE
      VALUES       ( 'Richard', 'K', 'Marini', '653298653', '1962-12-30', '98
                     Oak Forest, Katy, TX', 'M', 37000, '653298653', 4 );
```



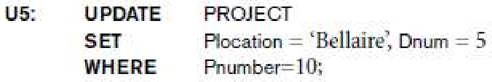
```
U1A:  INSERT INTO  EMPLOYEE (Fname, Lname, Dno, Ssn)
      VALUES       ('Richard', 'Marini', 4, '653298653');
```

- Attributes not listed are set to the value constrained by DEFAULT or NULL if there is no default.
- DELETE removes tuples from a relation.
- To delete a table from a schema, we use DROP TABLE tableName.



```
U4C:  DELETE FROM  EMPLOYEE
      WHERE        Dno=5;

U4D:  DELETE FROM  EMPLOYEE;
```

- If no WHERE is used, we delete all rows.
- The UPDATE command updates information within the table.



```
U5:   UPDATE  PROJECT
      SET     Plocation = 'Bellaire', Dnum = 5
      WHERE   Pnumber=10;
```

- The SET will be the place where the attributes are modified. WHERE selects the attributes to be updated.

**CHAPTER 5: ADVANCED SQL**

*5.1 More Complex SQL Retrieval Queries*

- NULL can have 3 different interpretations within SQL:
  - The value is unknown
  - The value is known, but we do not want the value to be known
  - The value belongs to an attribute not applicable to a specific row. Think derived attribute with all NULL attributes it is derived from.
- When we use NULL in a comparison, the return value will always be UNKNOWN:

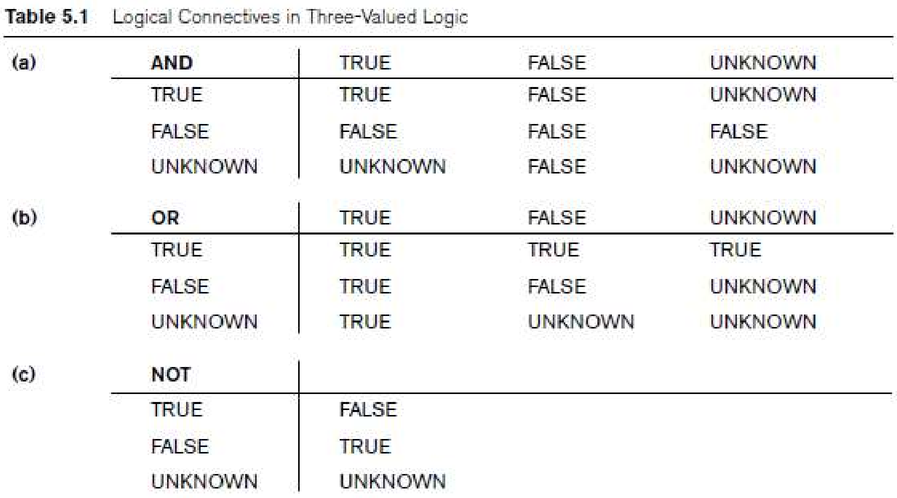
- 

**Table 5.1** Logical Connectives in Three-Valued Logic

(a)

| AND | TRUE | FALSE | UNKNOWN |
|---|---|---|---|
| TRUE | TRUE | FALSE | UNKNOWN |
| FALSE | FALSE | FALSE | FALSE |
| UNKNOWN | UNKNOWN | FALSE | UNKNOWN |

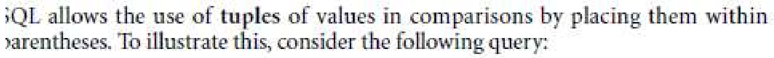
(b)

| OR | TRUE | FALSE | UNKNOWN |
|---|---|---|---|
| TRUE | TRUE | TRUE | TRUE |
| FALSE | TRUE | FALSE | UNKNOWN |
| UNKNOWN | TRUE | UNKNOWN | UNKNOWN |

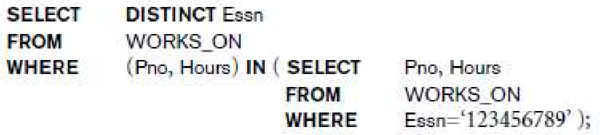
(c)

| NOT | |
|---|---|
| TRUE | FALSE |
| FALSE | TRUE |
| UNKNOWN | UNKNOWN |

- We typically do not select tuples/rows that evaluate to FALSE or UNKNOWN.
- To check if an attribute is NULL, use the keyword IS or IS NOT.
- We use nested queries when we have a query that requires existing values in the database be fetched and used in a comparison condition. We do this by using SELECT FROM blocks in the WHERE clause of an outer SELECT FROM block.
- Nested selects use the comparison operator IN to compare a value V with a set/multiset of values V, and returns true if v is one of the elements in V.
- We can use OR between nested SELECTS to get results from both fetches.

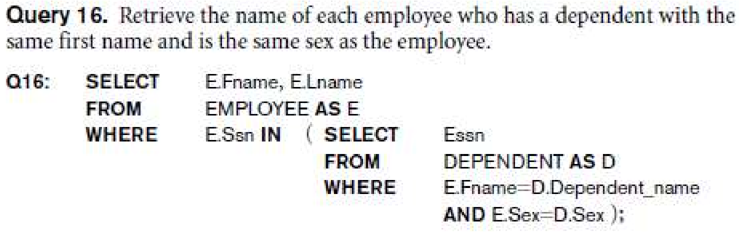


QL allows the use of tuples of values in comparisons by placing them within arentheses. To illustrate this, consider the following query:

- 

```
SELECT      DISTINCT Essn
FROM        WORKS_ON
WHERE       (Pno, Hours) IN ( SELECT     Pno, Hours
                              FROM       WORKS_ON
                              WHERE      Essn='123456789' );
```

- The =ANY operator in the WHERE clause will select all tuples that validate the operator placed before =ANY, so Salary > =ANY ( SELECT Salary FROM EMPLOYEE WHERE DNO = 5); would select all salaries greater than any salary from dno 5.
- The ALL operator will return the tuples that are compared to every tuple in the nested select, and return true for all of the comparisons against all the tuples.
- In nested queries, and all queries, it is usually better to create aliases for ALL TABLES REFERENCED IN a SQL QUERY. So, TABLE AS T in the FROM clause.
- Two queries are correlated if the inner query references an attribute of a table in the outer query, like so:

- 

**Query 16.** Retrieve the name of each employee who has a dependent with the same first name and is the same sex as the employee.

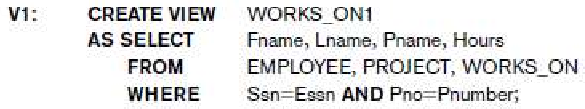
```
Q16:    SELECT      E.Fname, E.Lname
        FROM        EMPLOYEE AS E
        WHERE       E.Ssn IN  ( SELECT     Essn
                                FROM       DEPENDENT AS D
                                WHERE      E.Fname=D.Dependent_name
                                           AND E.Sex=D.Sex );
```

- The EXISTS function in SQL is used to see if a correlated nested query is empty or not. It will return TRUE if the nested query has one or more rows, and FALSE otherwise. NOT EXISTS does the opposite. It is good for checking for rows or attributes that associate with all the same value.
- The UNIQUE(Q) function returns TRUE if there are no duplicate rows in the result of query Q.
- You can use TABLENAME JOIN TABLENAME ON clause in a FROM statement. This will smash the two tables together into one large table, but only select the rows that fulfill the clause after ON. The default is an inner join, which does not include NULL attributes, but we can specify and OUTER JOIN.
- Aggregate functions in SQL are:
    - SUM, MAX, MIN, AVG, and COUNT, which return the sum of the parameter attribute, the maximum parameter, the minimum parameter, the average of every parameter attribute from every row, and the count of rows that have non NULL parameter attribute.
    - Placed within the SELECT statement, as they return a one row table.
- The GROUP BY clause will create a one row table, applying the aggregate functions on the rows

with identical attributes specified in the GROUP BY clause.

- We use the HAVING clause when working with aggregate functions to put operational constraints on aggregate functions, like COUNT(*) < 20.t

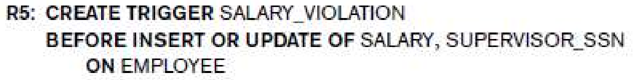
*Views*

- A view is a table defined by other tables, or a virtual table. An example of view syntax:
- 

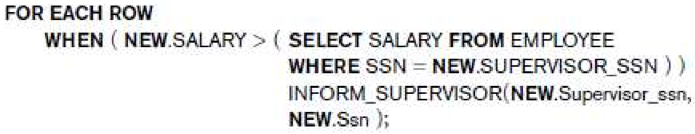
```
V1:    CREATE VIEW   WORKS_ON1
       AS SELECT     Fname, Lname, Pname, Hours
          FROM       EMPLOYEE, PROJECT, WORKS_ON
          WHERE      Ssn=Essn AND Pno=Pnumber;
```

- We use the ALTER command to change tables by adding or deleting table columns, and we use DROP to delete schemas, tables, views and constraints.

*Triggers*

- A SQL trigger is an action that takes place when a certain event occurs. These have 3 components:
  - The event, or the thing that sets off the trigger.
  - The condition, or the condition that causes the trigger to fire.
  - The action, or what we do if the condition is true?
- When the even happens, the system will check the condition of the event, and if it returns TRUE, will perform the action.
- The statement is made as CREATE TRIGGER. This example checks to see if an employee has a higher salary than their manager:
- 

```
R5: CREATE TRIGGER SALARY_VIOLATION
    BEFORE INSERT OR UPDATE OF SALARY, SUPERVISOR_SSN
        ON EMPLOYEE
```

- 

```
FOR EACH ROW
    WHEN ( NEW.SALARY > ( SELECT SALARY FROM EMPLOYEE
                          WHERE SSN = NEW.SUPERVISOR_SSN ) )
                          INFORM_SUPERVISOR(NEW.Supervisor_ssn,
                          NEW.Ssn );
```

**CHAPTER 6: Relational Algebra/Calculus**

- This chapter covers relational algebra and relational calculus.
- These concepts were developed before SQL was actually made, and are the formal languages for the relational model. The practical language is SQL, and it was based on these concepts.
- We have defined what a data model needs to be structurally sounds and correctly constrained, and these will describe how to interact with that data model.
- The basic set of concepts is the relational algebra. These perform basic retrieval requests. The results are new relations. A sequence of operations form a relational algebra expression.
- Relational calculus is a declarative language for specifying queries. There is no order of operations specifying HOW to get the data, it is just the information that the result should contain.
- Unary operations operate on single relations, binary operations operate on two tables. Set operations act on multiple tables.

*6.1 Unary Operations: SELECT and PROJECT*

- Select is an operation that chooses a subset of the tuples from a relation that specifies a selection condition. That select condition is specified within the statement.
- Select is denoted by (little omega)<selection condition>(Relation). It will always select less than or equal tuples from the specified relation. The number of these is called the degree. Its commutative so you can select from a select relation.
- Project is an operation that chooses a subset of the columns/attributes from a relation.
- Denoted by (pi)<attribute name>(Relation)
- Project will eliminate duplicate tuples from the relation it returns and only return one.
- The RENAME option can rename a relation or its attributes.

*6.2 Relational Algebra Operations from Set Theory*

- The UNION operation lists all tuples from a left relation, a right relation, or both. It will only list them once.
- For two relations to be union compatible, they have to have the same number of columns and each corresponding pair of attributes has the same domain.
- INTERSECTION returns a relation with elements that are in both R and S relations.
- SET DIFFERENCE returns a relation with everything in relation R but not those that show up in both R and S and S alone. R – S.
- The CROSS PRODUCT/JOIN operation is denoted by X, and does not have to be union compatible like all other set operations above. This combines every tuple in one relation with every other

tuple in the other set. The tuples get each other's attributes, and these are set to NULL. You can perform this on any number of relations. In practice this means that each tuple will be repeated from the left with every tuple from the right, moving on when all left tuples are exhausted.

*6.3 Binary Relational Operations*

- The JOIN, denoted by |X| is used to combine related tuples from two relations into single longer tuples. For example, say we have a department with a list of every department, and those departments have a manager with an SSN listed. A join with the ssns of the manager with every ssn in the employee records would produce the names of the managers.
- This is pretty much the Cartesian product operation with a select after it. A JOIN will ONLY show the tuples that fulfill the requirements, like a select. It also eliminates all tuples in both relations who would just have null values for one side or the other.
- A JOIN with equality as the selector is called an EQUIJOIN.
- A NATURAL JOIN, denoted by *, gets rid of the duplicate attribute in an EQUIJOIN. Behaves just like EQUIJOIN otherwise.
- These above operations are called inner joins.
- An outer join can be done in three ways, and is defined as a JOIN, but it does not eliminate the tuples who have NULL values on the right or left. A LEFT OUTER JOIN keeps everything from the left, the RIGHT and FULL do what you think they'd do.

*6.6 The Tuple Relational Calculus*

- In both Tuple and domain relational calculus, we write on declarative expression to specify a retrieval request. There is no description of what order to retrieve the info, just what the info should look like after retrieval in a relation.
- The expressive powers of both language is identical, between algebra, and calculus with relations.
- Tuple relational calculus is based on specifying a number of tuple variables. These range over a particular relation. Written as {t | condition(t)}. For example, {t | EMPLOYEE(t) AND t.Salary>50000} specifies that we want t to be a tuple within the EMPLYEE table and it has an attribute with the value > 50000.
- To specify only certain attributes to be in a new relation, use {t.attribute1, t.attribute2 | conditions}
- A formula is the second part of the tuple set builder. It can have multiple predicate calculus atoms:
  - Of the form RELATION_NAME(t), which specifies which relation to look in.
  - Of the form ti.A op tj.B, where op is a comparison operator like = or > and ti is a tuple variable and A is an attriubte. You can also have tj.B just be a constant value like above.
- If an existential quantifier is in front of an atom, then the formula F must evaluate to true for at least one tuple in the atom. If a universal is in front, it has to evaluate to true for all of them.

6.17d
FLIGHTFAREJOIN <- FLIGHT |X| <Flight_number = FARE.FLight_number)
TOTAL FARE INFO <- SELECT SYMBOL <Flight_number = CO197> (FLIGHTFAREJOIN)
CO197FARES <- pi (Fare_code, Amount, Restriction)

- There are 2 levels that we can use to judge the goodness of relation schemas. A good **logical** level of a schema is allowing users to understand the meaning of the data in the relations clearly. A good **implementation** level is one that efficiently stores and updates the tuples in a base relation. A base relation is a table we make ourself, a logical relation can be a base relation and a virtual relation (view).f
- A **bottom down design methodology** considers the basic relationships between individual attributes as the starting point to construct relation schemas. This really isn't used that much today.
- A **top down design methodology** starts with a number of groupings of attributes into relations that exist together naturally, like an invoice, form or report. These relations are then analyzed individually and against each other, and we refactor them until the properties for normalization are met.

*15.1 Informal Design Guidelines for Relation Schemas*

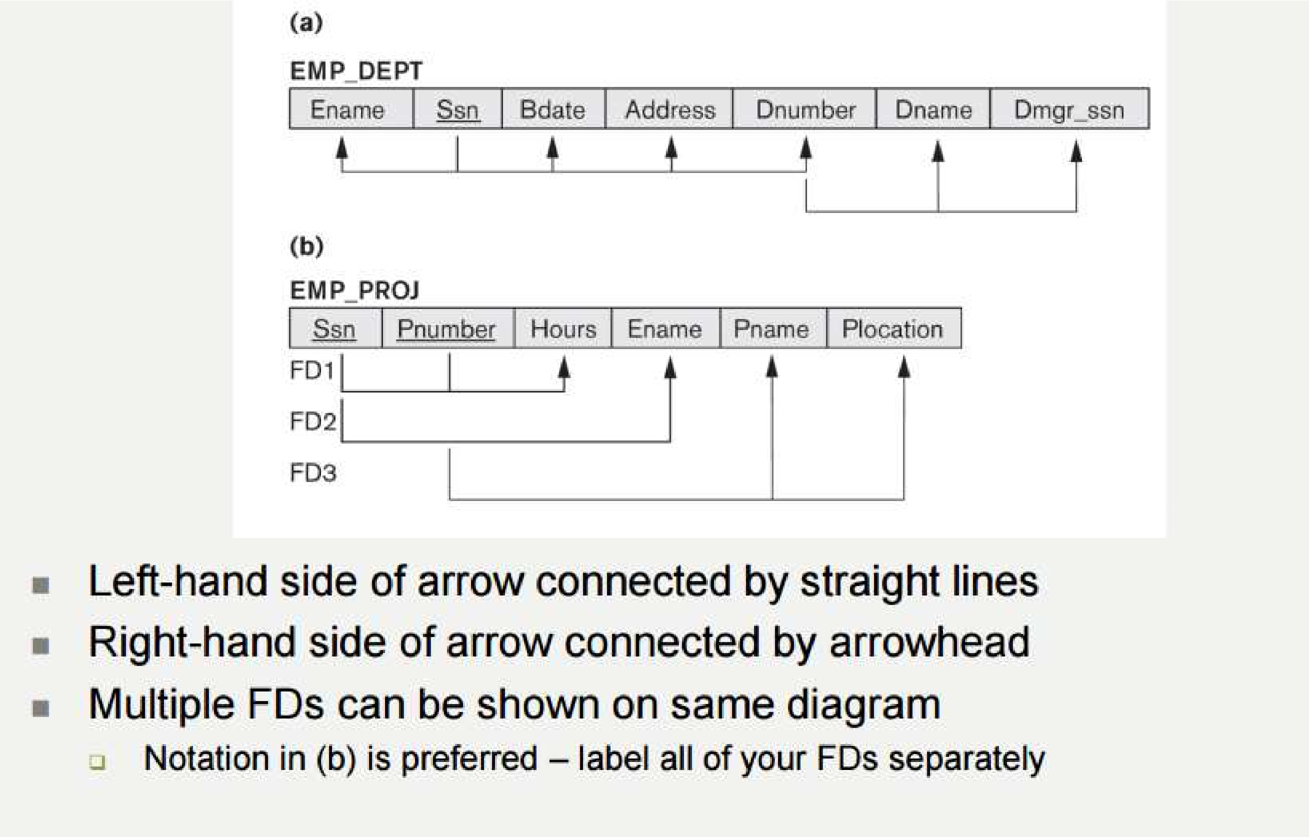
- There are four informal guidelines that may be used as measures to determine the quality of relational schema design:
  - The semantics of the attributes is clear in the schema
  - Reducing the redundant info in tuples
  - Reducing the NULL values in tuples
  - Disallowing the possibility of generating spurious tuples.
- The **semantics** of a relation refer to its meaning resulting from the interpretation of attribute values in a tuple. The easier to explain the semantics of the relation, the better the relation

schema design will be.

- This means our first guideline to creating a relation schema is to design it so that I
- t is easy to explain its meaning. This means:
  - Avoiding combining attributes from multiple entity types into a single relation.
  - Try to assign one relation to one entity or relationship type.
- One goal of schema design is to use the least storage space possible for the base relations that have to exist on hard disk.
- If we store natural joins of base relations, we can get problems called **update anomalies**. These are classified into insertion, deletion, and modification anomalies.
  - Insertion anomalies come in two types, with one being that you have to enter in the same exact info correctly for multiple relations if they are a natural join of two relations, while a well-designed relationship will only use one foreign key pointing at another table. The other problem is that if we need to insert only half of the data, we have to use a bunch of NULL values, and those can end up in primary key columns.
  - Deletion anomalies are involved in the last example. If a particular piece of info is stored with another set of info we need to delete, and if it is the last piece of info that exists, it is lost forever even if we do not want it to be.
  - If we change the value of one of the attributes that has a lot of duplicate attributes in other tuples, we have to update every tuple. Gross.
- The second guideline is to design the relation schemas such that these anomalies do not exist.
- Sometimes we have to violate these guidelines to make certain queries run better. These are handled with triggers and other methods.
- The third guideline is simple. Avoid placing attributes in a base relation whose values may frequently be NULL. If you do need them, make sure they do not apply to a majority of tuples in the relation.
- **Spurious tuples** are tuples that are created by two tables that have duplicate attributes. These tuples are incorrect and this means they report false data.
- Guideline 4 helps avoid this, with making sure relation schemas, when joined, do not have matching attributes except for the foreign keys and primary keys.

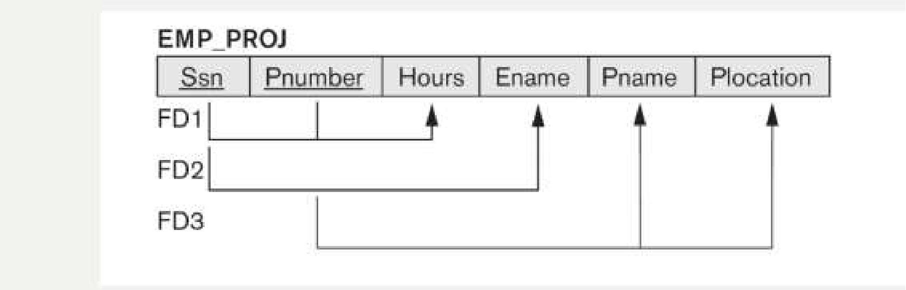
*15.2 Functional Dependencies*

- A **functional dependency** is a constraint between two sets of attributes from the database. It is denoted by X -> Y, and is formally defined as being between two sets of attributes X and Y that are subsets of R (A single relational table holding all attributes of all tables in the DB), and it specifies a constraint on the possible tuples that can form a relation state r of R. This constraint is that for any two tuples t1 and t2 in r, t1[X] = t2[X] <=> t1[Y] = t2[Y]. This means that the values of the Y component of a tuple in r depend on the values of the X component, and the values of the X component determine the values of the Y component. This is said by saying Y is **functionally dependent on** X. Abbreviated FD. The set of attributes X is the left side, the set of attributes Y is the right side.
- To break this down to English, iff whenever two tuples within r have the same X value, they have to have the same Y value.
- A **candidate key** is an attribute that cannot be repeated in r.
- The main use of functional dependencies is to describe further a relation schema R by specifying constraints on its attributes that must hold at all times.
- Real layman's terms would be, given a value of X, we know the value of Y.
- A functional dependency is a property of the relation schema R.
- One cannot state with certainty, until the meaning of the corresponding attributes is understood, which FD's hold/exist. We can state which do not exist though, if tuples exist that violate a functional dependency. Basically, FD's must be explicitly defined.

- 

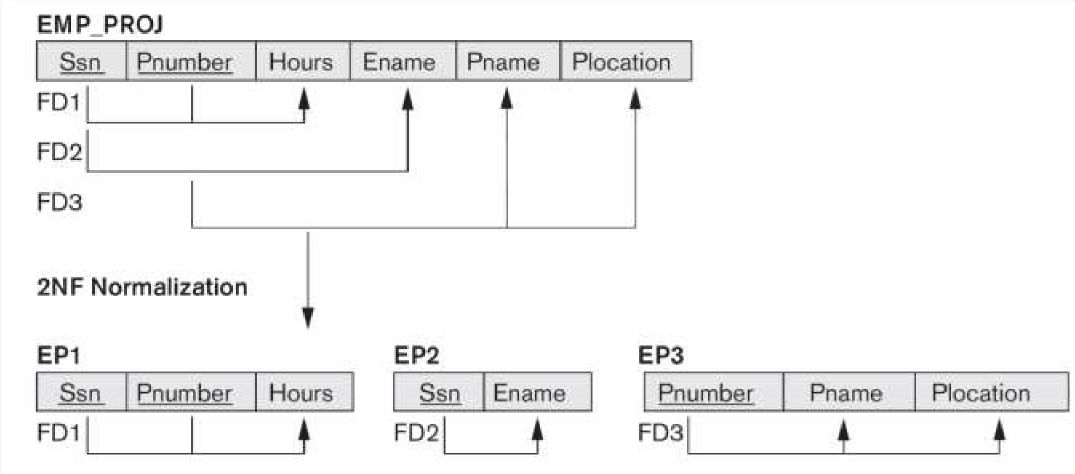
  - Left-hand side of arrow connected by straight lines
  - Right-hand side of arrow connected by arrowhead
  - Multiple FDs can be shown on same diagram
    - Notation in (b) is preferred – label all of your FDs separately

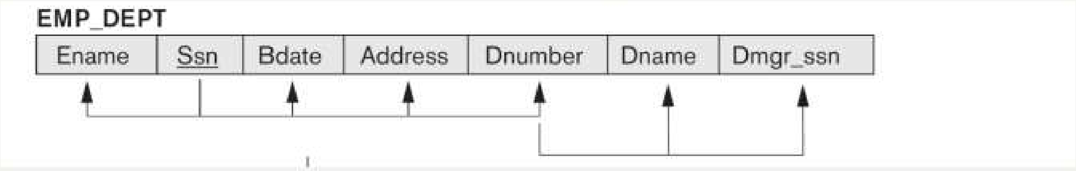
*15.3 Normal Forms Based on Primary Keys*

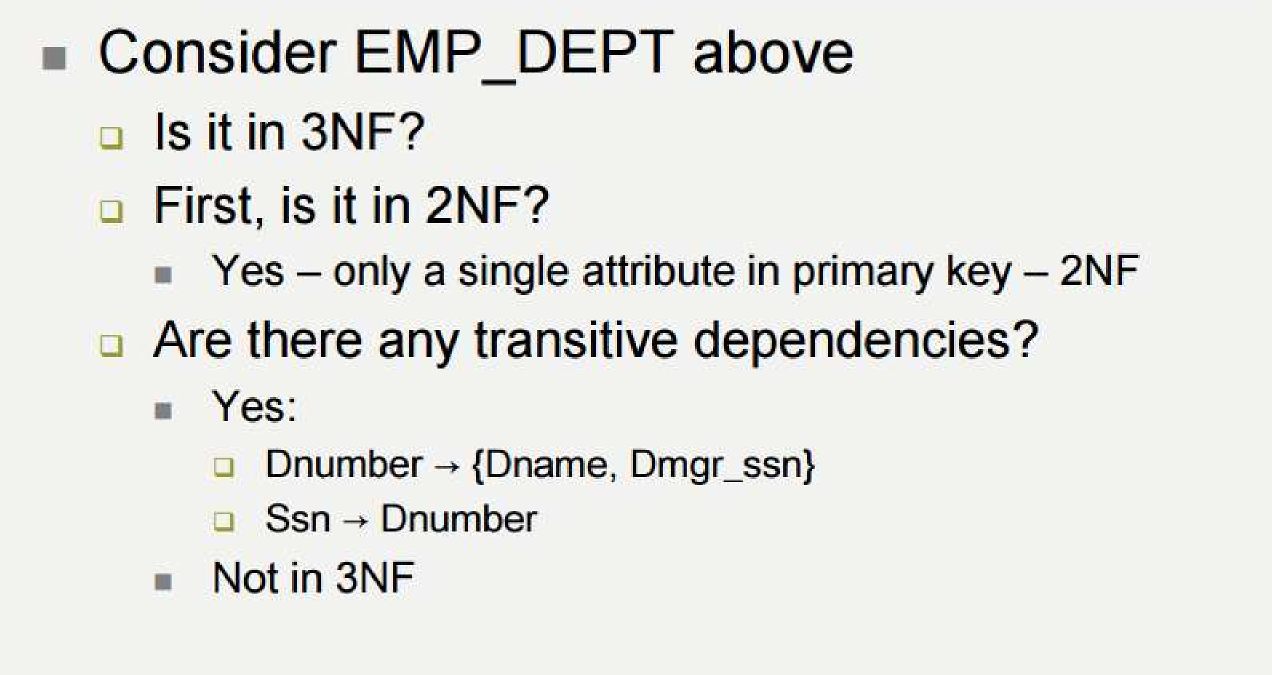
- Normalization is a process where we examine and revise our relation schemas based on their FD's and primary keys, and attempts to minimize redundancy in stored data, and minimize the number of update anomalies. It allows DB designers a formal framework for analyzing relation schemas, and a series of tests that allow us to have different degrees of normalization in our data.
- A **normal form** is a criteria for determine how vulnerable a relation is to logical inconsistencies (breaking the guidelines). It has different levels.
- The **1st normal form** states that the domain of an attribute must contain only atomic (simple, indivisible) values and the value of any attribute in a tuple must be a single value from the attribute domain. This means no lists or anything as a value.
- To change a shitty database into 1NF, we can change a value with a list to multiple tuples with the same info, each with a different item from the list in the tuple (bad, adds redundancy), or we can create a new relation with the primary key of the tuple and a list item in separate tuples corresponding to that primary key. We would make the primary key in the original table a foreign key as well.
- The **2nd normal form** is stricter. A relation schema is in 2NF is every nonprime attribute A in R is fully functionally dependent on the primary key of R. A **prime attribute** is any attribute that is part of some candidate key of R (any unique key). **Fully FD** means that if you remove an attribute in the X in X->Y, then the FD is destroyed.
- If you have only a single candidate key, this is easy. If you have a multiattribute candidate key, it gets harder. For example:
- 

- Hours is dependent on both SSN and Pnumber, but Ename is only dependent on SSN, and Pname and Plocation are only dependent on Pnumber. Thus, none of the last 3 nonprime attributes are FFD on the primary key, and the table is not 2NF.
- To normalize into 2NF, we have to break relations down into a set of smaller relations. One

relation will be made for each existing FFD in the original relation. We also keep the relation with the full primary key.

- 

- **3rd Normal Form** is even more strict. In this, a relation schema R is in 3NF if it is in 2NF and if no nonprime attribute of R is transitively dependent on the primary key. **Transitive Dependency** is if when the FD X->Y holds for a relation and there exist a set of attributes Z in R that is not a candidate key of R such that X->Z, and Z->Y, the TD is the FD X->Y.
- 



- Consider EMP_DEPT above
    - Is it in 3NF?
    - First, is it in 2NF?
        - Yes – only a single attribute in primary key – 2NF
    - Are there any transitive dependencies?
        - Yes:
            - Dnumber → {Dname, Dmgr_ssn}
            - Ssn → Dnumber
        - Not in 3NF

- To fix this, we create new relations based off of the old one, with the attribute that can define other attributes as a primary key. EX, Dnumber would be a primary key now. Before we do this we also turn it into 2NF. We would also keep the new primary key in the old relation, and make it a foreign key.

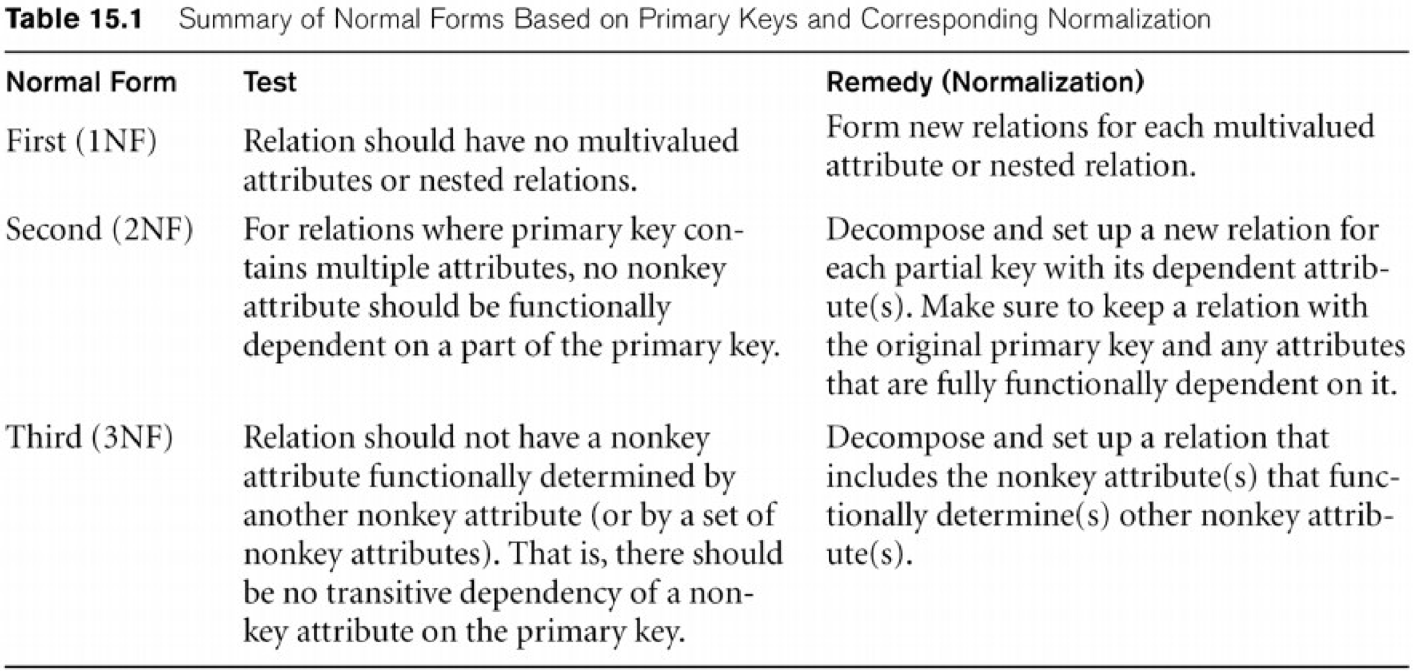
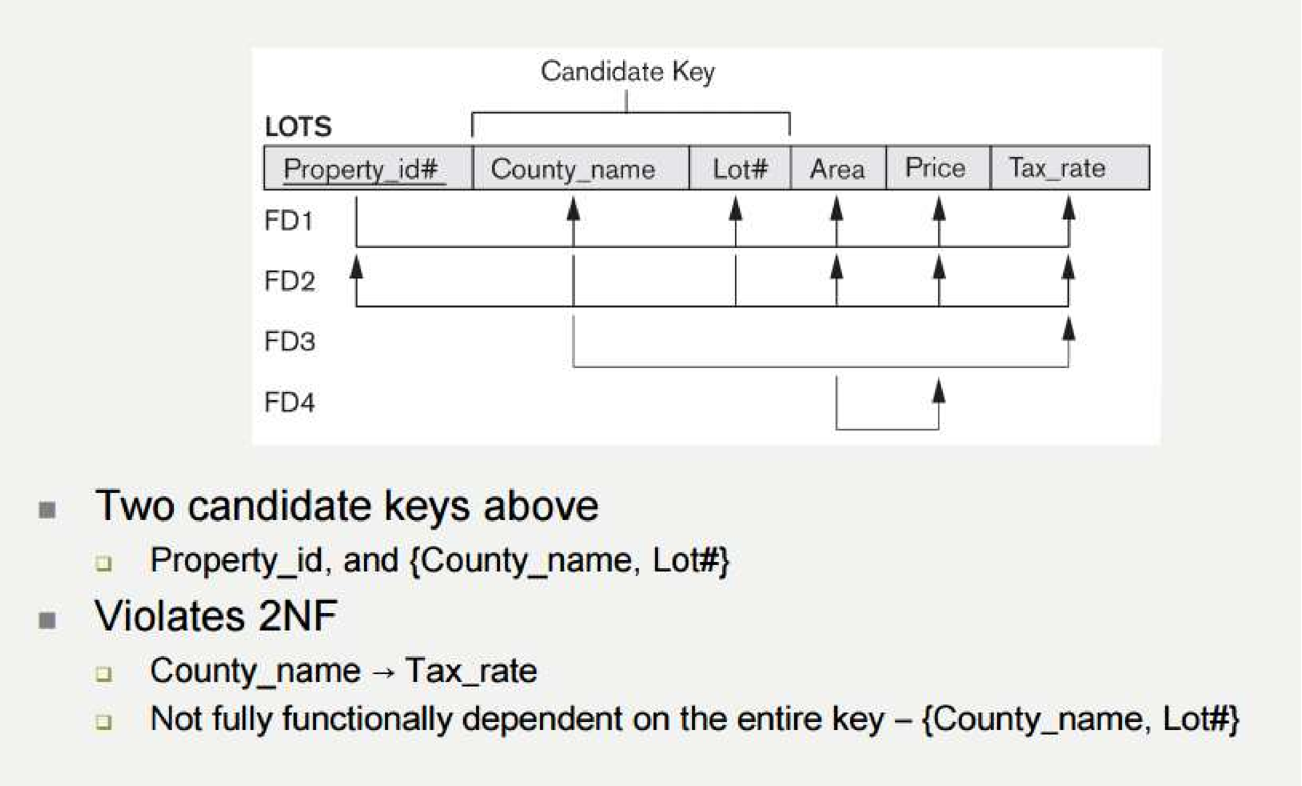


**Table 15.1** Summary of Normal Forms Based on Primary Keys and Corresponding Normalization

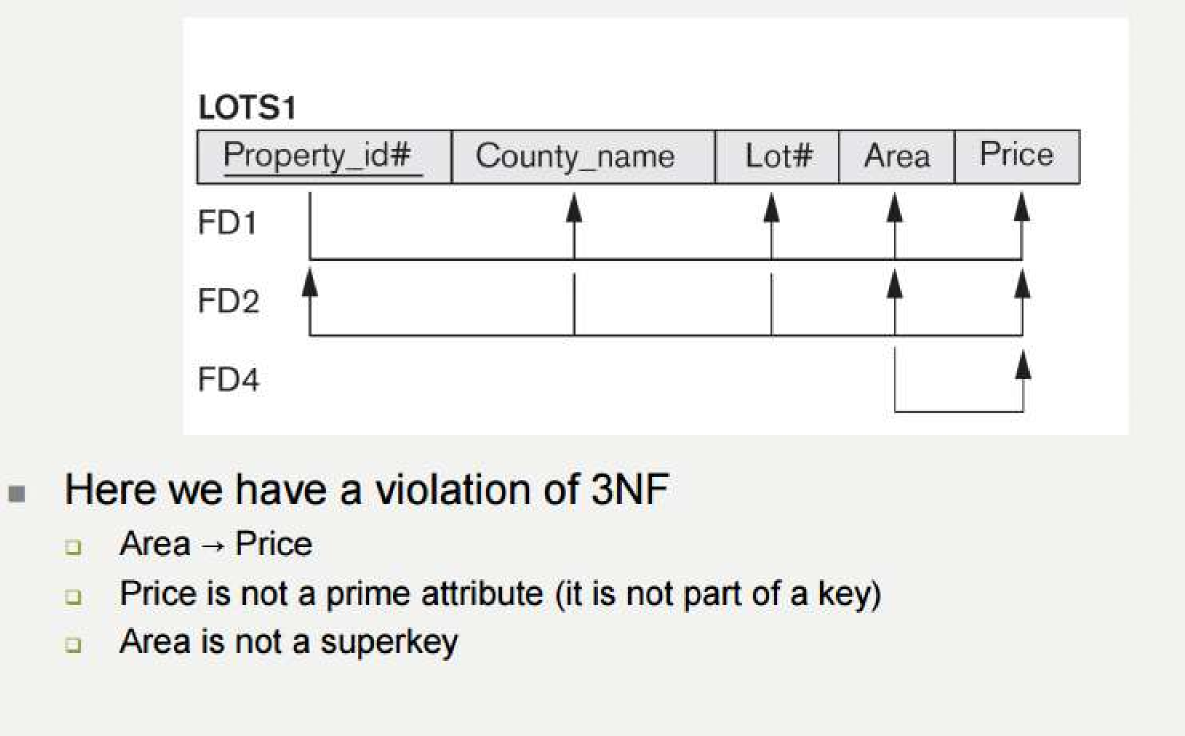
| Normal Form | Test | Remedy (Normalization) |
|---|---|---|
| First (1NF) | Relation should have no multivalued attributes or nested relations. | Form new relations for each multivalued attribute or nested relation. |
| Second (2NF) | For relations where primary key contains multiple attributes, no nonkey attribute should be functionally dependent on a part of the primary key. | Decompose and set up a new relation for each partial key with its dependent attribute(s). Make sure to keep a relation with the original primary key and any attributes that are fully functionally dependent on it. |
| Third (3NF) | Relation should not have a nonkey attribute functionally determined by another nonkey attribute (or by a set of nonkey attributes). That is, there should be no transitive dependency of a nonkey attribute on the primary key. | Decompose and set up a relation that includes the nonkey attribute(s) that functionally determine(s) other nonkey attribute(s). |

*15.4 General Definitions of Second and Third Normal Forms*

- The previous definitions have been based off of primary keys, not candidate keys. We have to update our process.
- For 2NF, we have to make sure that every nonprime attribute A in R is not partially dependent on ANY key of R. This means if we have a multivalued candidate key, if a nonprime is dependent on only one part of it, it is not 2NF.
- 

  - Two candidate keys above
    - Property_id, and {County_name, Lot#}
  - Violates 2NF
    - County_name → Tax_rate
    - Not fully functionally dependent on the entire key – {County_name, Lot#}
- The multiple arrows are confusing, but follow this guide:
  - Each FD is labelled. The blank vertical line corresponds to the left side of the FD, the arrowhead on the same level refers to the right side. So FD1 has X as Property ID -> everything, FD2 has County Name and Lot Pointing to everything, FD 3 has County name pointing to Tax Rate, and FD 4 has area point at price.
- To fix this, we break it down into smaller relations, with any functional dependencies that do not include the entire candidate key being put in their own relation. The example above would make County Name a foreign key, and create a new table with county name as the key and tax rate as a

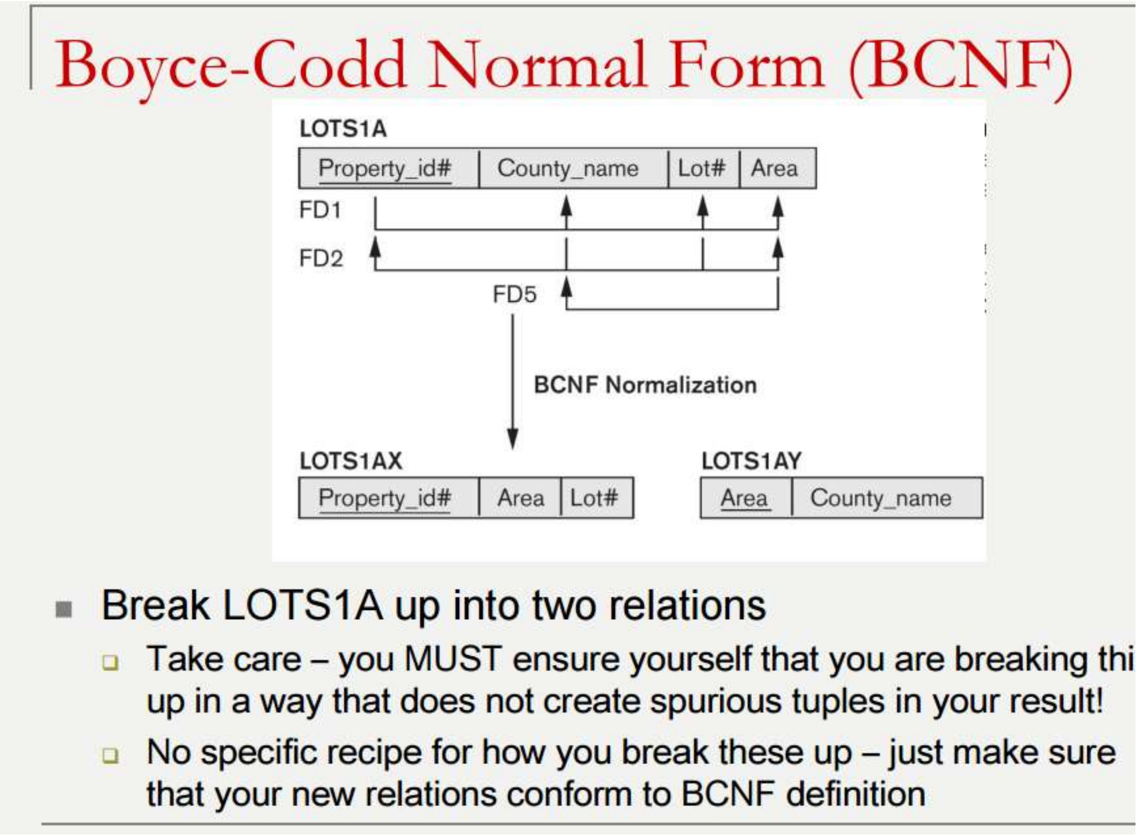
nonprime. We also keep both keys that have fully FD's attributes in a relation with all their fully FD's attributes.

- The general form of 3NF is that a relation schema R is in 3NF if whenever a nontrivial FD X-A holds in R, either X is a superkey of R or A is a prime attribute of R. This means as long as a nonprime FD's a primary or candidate key, it is in 3NF.
- 

  - Here we have a violation of 3NF
    - □ Area → Price
    - □ Price is not a prime attribute (it is not part of a key)
    - □ Area is not a superkey
- To normalize something to 3NF, we take the nonsuperkey attributes that functionally determine other nonsuperkey attributes as a primary key. This is all done after making it 2NF.

*15.5 Boyce Codd Normal Form*

- BCNF is the same thing as 3NF, but X can only be a superkey. Thus BCNF is in 3NF.
- A superkey is either a primary or candidate key.
- To get a relation into BCNF, we get it into 3NF, and decompose into relations where the FD described by the nonkey attribute becomes a FD on a key attribute:

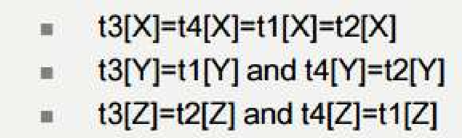
-


## Boyce-Codd Normal Form (BCNF)

- Break LOTS1A up into two relations
  - Take care – you MUST ensure yourself that you are breaking thi up in a way that does not create spurious tuples in your result!
  - No specific recipe for how you break these up – just make sure that your new relations conform to BCNF definition

- As a best practice, database designers should work to get 3NF/BCNF for every relation schema in a DB.

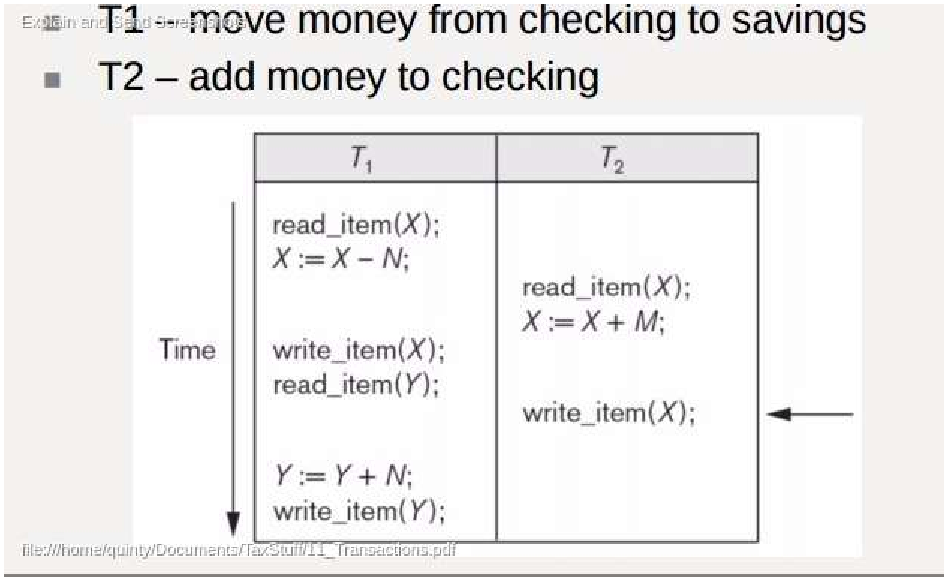
*15.6 Multivalued Dependency and Fourth Normal Form*

- A multivalued dependency X--Y specifies the following constraint on any relation state r: If t1, t2 exist in r such that t1[x] = t2[x], then t3 and t4 should exist in r such that:
-


  - t3[X]=t4[X]=t1[X]=t2[X]
  - t3[Y]=t1[Y] and t4[Y]=t2[Y]
  - t3[Z]=t2[Z] and t4[Z]=t1[Z]
- Z are the attributes in R that are not in X or Y.
- 4NF form is achieved by having every nontrivial MVD X--Y having X as a superkey of R, the relation. To fix this, put every decomposing each nontrivial MVD into new relations.

**Transactions**

- A transaction is a collection of operations on a database.
- Concurrency control in a database ensures that processes running simultaneously on a DB behave in an appropriate way. The result of running two processes in an interleaved fashion is always predictable. This means only allowing one set of transactions to run at once.
- An issue that exists with concurrency control are lost updates, which occurs when two processes try to update the same item. This can result in incorrect data to be written to a DB.

- 

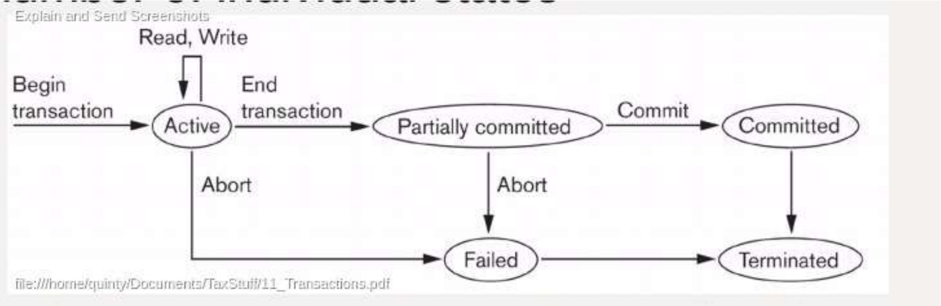
Explain and Send Screenshots

- T1 – move money from checking to savings
- T2 – add money to checking

| $T_1$ | $T_2$ |
|---|---|
| read_item($X$);<br>$X := X - N$; | |
| | read_item($X$);<br>$X := X + M$; |
| write_item($X$);<br>read_item($Y$); | |
| | write_item($X$); |
| $Y := Y + N$;<br>write_item($Y$); | |

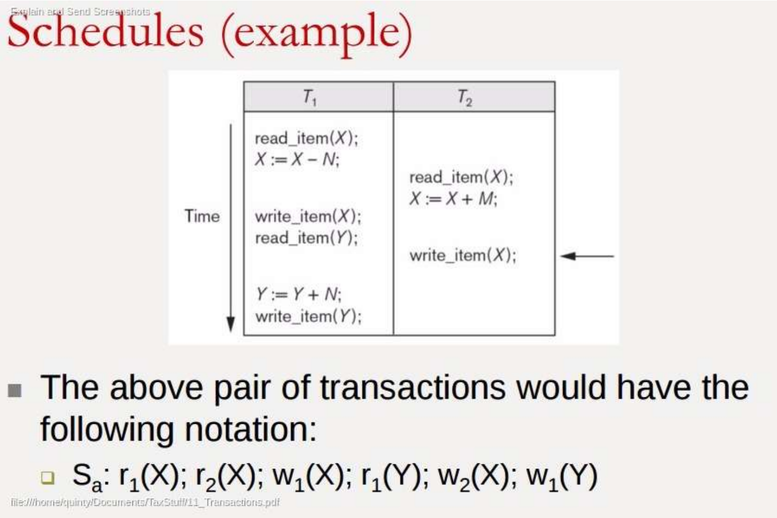
file:///home/quinty/Documents/TaxStuff/11_Transactions.pdf

- By definition, a transaction is a logical unit of database processing. K calls it a bundle of operations grouped together.
- To handle concurrency, transactions must follow ACID, and must be
  - Atomic, meaning they need to be entirely performed or not done at all, no half steps.
  - Consistent, meaning that transactions have to maintain a consistent state from the start to the end of the process. Consistency here means the process succeeds.
  - Isolated, meaning transactions should always function as if they were executing in isolation from other transactions.
  - Durable, meaning that database changes made by a completed transaction must persist, even if the database fails.

- As a transaction is processed, it moves through a number of states:
- 

file:///home/quinty/Documents/TaxStuff/11_Transactions.pdf

- Active transactions can have any number of I/O operations, and when I/O fails, the transaction aborts and no changes are made.
- We need to be able to restore the DB to the last transaction commit point after a failure. A DBMS keeps a system log of transactions in a sequential, append only file kept on the disk which, upon DB failure, will play back transactions to the database until the commit point before failure.
- Total ordering in transaction order means that every transaction has a set time to run when compared to others, partial ordering mean there may be ambiguities as to when to run one transaction vs the other,
- Each schedule is noted by Sa, Sb, and each transaction is noted by T1, and T2. r1 and r2 and w1 and w2 denote read and write operations on a data item in the DB.
-

- 

## Schedules (example)

|  | $T_1$ | $T_2$ |
|---|---|---|
| Time | read_item($X$);<br>$X := X - N$; |  |
|  |  | read_item($X$);<br>$X := X + M$; |
|  | write_item($X$);<br>read_item($Y$); |  |
|  |  | write_item($X$); |
|  | $Y := Y + N$;<br>write_item($Y$); |  |

- The above pair of transactions would have the following notation:
  - $S_a$: $r_1(X)$; $r_2(X)$; $w_1(X)$; $r_1(Y)$; $w_2(X)$; $w_1(Y)$

file:///home/quinty/Documents/TaxStuff/11_Transactions.pdf

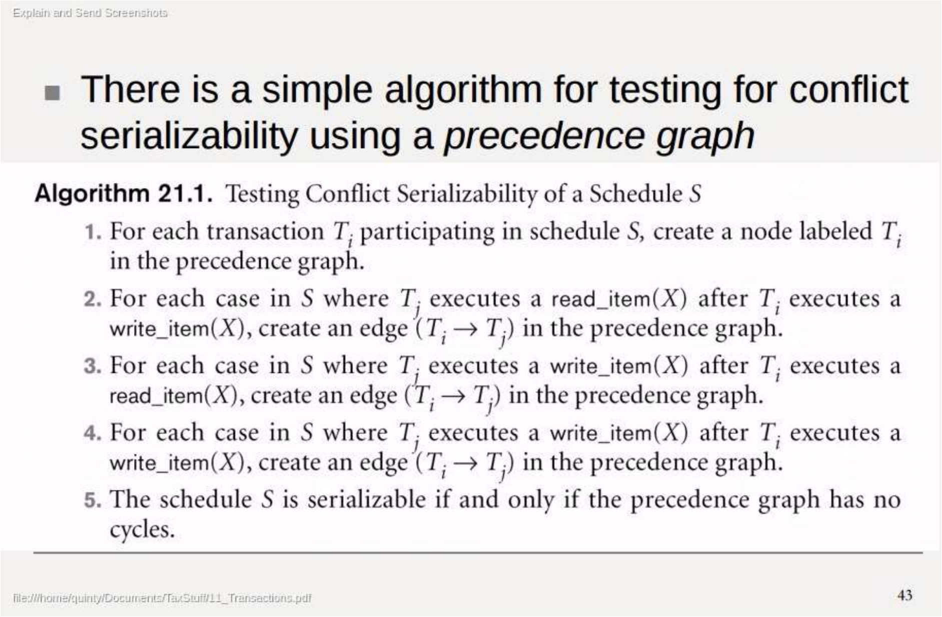
- The read/write subnumerals and transaction subnumerals match up in the notation.
- To detect a conflict in the operations of a transaction, there are a couple of conditions:
  - Two operations belong to different transactions
  - The operate on the same DB item
  - At least one of the two operations is a write operation.
  - As an example, if two operations change the final result of the schedule by changing their order, they conflict.

- A schedule is *complete* if:
  - The operations in the schedule are exactly those of the transactions that make up the schedule.
  - For any 2 of the operations, their order in the schedule is the same as their order in the transaction
  - For any pair of conflicting transactions, they cannot occur concurrently.

- A good way to compare schedules is to characterize their ease of recovery, tracking how easy it is to recover from a failed transaction or a system failure.
- To make a schedule recoverable, we should do a couple of things:
  - Once a transaction T is committed, it should never be necessary to rollback T.
  - A1 means transaction 1 has aborted/failed in a schedule.
  - If a r occurs in a transaction after another transaction (T1) writes to it, and T1 fails, it is nonrecoverable. This is because the new data for the transaction is gone forever.

- A cascading rollback is one that is done on a transaction after another transaction in the schedule is aborted and rolled back. We want to avoid these, as they are time consuming. To have a cascadeless rollback, every transaction in the schedule should read only items that were written by transactions that have already been committed.
- A strict schedule means that a transaction cannot read OR write a data item until the last transaction that read/wrote to it has been committed or aborted.
- Serial schedules are those that have transactions that do not interleave, with no concurrency. Each transaction is independent.
- A concurrent schedule that is equivalent to a serial schedule is a *serializable* schedule.
- To test for serializability, there are a couple of tests:
  - Result equivalence is the simplest way of testing for equivalence. Two schedules are result equivalent if they produce the same final state of the database. This isn't a good approach, due to accidental commits.
  - A better test is conflict equivalence. This is when two schedules have the same operations in each schedule AND the conflicting operations occur in the same order in both schedules.

-

- 

  Explain and Send Screenshots

  - There is a simple algorithm for testing for conflict serializability using a *precedence graph*

  **Algorithm 21.1.** Testing Conflict Serializability of a Schedule $S$

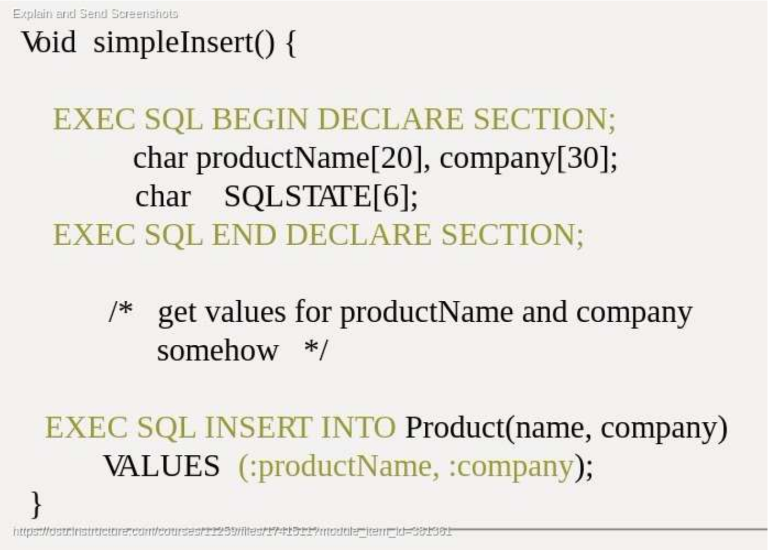
  1. For each transaction $T_i$ participating in schedule $S$, create a node labeled $T_i$ in the precedence graph.
  2. For each case in $S$ where $T_j$ executes a read_item($X$) after $T_i$ executes a write_item($X$), create an edge ($T_i \rightarrow T_j$) in the precedence graph.
  3. For each case in $S$ where $T_j$ executes a write_item($X$) after $T_i$ executes a read_item($X$), create an edge ($T_i \rightarrow T_j$) in the precedence graph.
  4. For each case in $S$ where $T_j$ executes a write_item($X$) after $T_i$ executes a write_item($X$), create an edge ($T_i \rightarrow T_j$) in the precedence graph.
  5. The schedule $S$ is serializable if and only if the precedence graph has no cycles.

  file:///home/quinty/Documents/TaxStuff/11_Transactions.pdf 43

- The idea of serializability is used for concurrency control, and that is why we try to implement it into our DBMS.

**Embedded SQL**

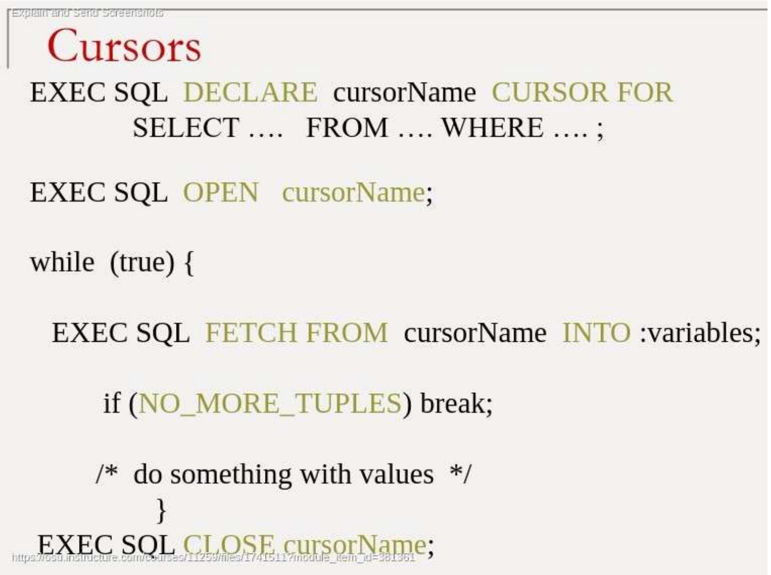
- Most database access is not via raw SQL, but by database applications. These are dedicated front end apps developed for other purposes. Examples are Java, Rails and other languages.
- There are a couple of approaches to app programming with databases:
  - We can use a library to read/write to the database. This can be an API or a built in library.
  - We can use SQL directly in our app code. We might need to process this before we submit it to the DB to make sure it is correct/not malicious.
  - We can design a new language to query the DB. Why.
- **Impedance mismatch** is when the database model and the programming language model are different. A database will return much different data than a programming language is used to processing, and we will have to adapt. We do this by binding database data types to language data types through libraries.
- When we get a returned query from a database, it can have multiple rows. So our approach to this is to bind the returned relation from the query to a multivalued data structure in the language. We also need a way to iterate over this multivalued data structure in the language.
- A **cursor** is used to iterate over these results.
- The client server model is typical between applications and DB's. The client will talk to a variety of databases, by connecting to the database, submitting a query, and closing the connection after done.
- An example of an API that connects to a database is the JDBC. This is the **Java Database Connectivity library**. It interacts with a relational DBMS server. It takes a driver unique to each type of database, like Oracle DB2 or Postgres. Something called a *driver* is used to act as an interface between Java and the DB, and must be loaded so that JDBC library functions operate on the correct DB parts.
- The JDBC usage goes as follows:
  - We import the library.
  - We load the driver for our DBMS.
  - We create a connection object to connect to the DB we want.
  - We create a Statement object that is created by the Connection object, and is used to execute queries and updates on the DB. This object uses strings containing SQL code.
  - We get the returned relation in a ResultSet object, and iterate over the ResultSet using a cursor and operating on the individual tuples in the ResultSet using get methods.
  - We close the Statement and Connection to prevent memory leaks.
- Embedded SQL allows for SQL queries to be embedded right into the code, but is popular at the application level like in Excel. Not in the code itself.
- To write some embedded SQL, we put EXEC SQL right before write the SQL code we want to run on the DB. We can also define shared variables that pass values to and from the DB. When we use shared vars in a statement, we put colons before them.
-

- 

Explain and Send Screenshots

```
Void  simpleInsert() {

   EXEC SQL BEGIN DECLARE SECTION;
          char productName[20], company[30];
          char    SQLSTATE[6];
   EXEC SQL END DECLARE SECTION;

        /*   get values for productName and company
             somehow   */

  EXEC SQL INSERT INTO Product(name, company)
       VALUES  (:productName, :company);
 }
```

https://osu.instructure.com/courses/11259/files/1741511?module_item_id=381361

- Cursors can be used in Embedded SQL as well. They can modify relations, read them and other things. We can use the ORDER BY keyword in the query in the embedded SQL to change the order the cursor gets tuples. These cursors can be protected against new changes to the relation it looks at, and can go forwards and backwards in the relation.
- 

Explain and Send Screenshots

Cursors

```
EXEC SQL  DECLARE  cursorName  CURSOR FOR
          SELECT ….   FROM …. WHERE …. ;

EXEC SQL  OPEN   cursorName;

while  (true) {

   EXEC SQL  FETCH FROM  cursorName  INTO :variables;

        if (NO_MORE_TUPLES) break;

       /*  do something with values  */
            }
 EXEC SQL CLOSE cursorName;
```

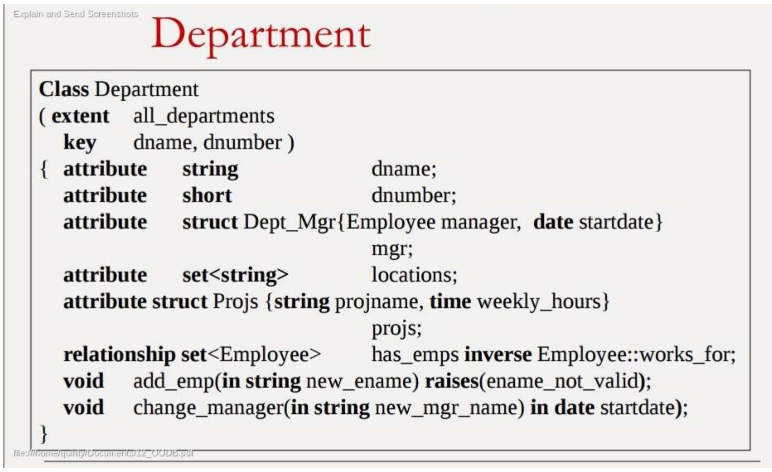
https://osu.instructure.com/courses/11259/files/1741511?module_item_id=381361

- Stored procedures are modules that are stored on the database persistently. This means they are queries or code that are run on the DB a lot.
- These are useful as they keep results consistent across apps, and also reduce network traffic between the apps and the DB.
- Procedures are created using CREATE PROCEDURE, and these are then defined using the language used by the RDBMS software. Different vendors have different languages.
- To use a stored procedure, we use a CALL statement, which takes in a procedure name then a list of arguments.
- Stored functions are the same kind of thing as stored procedures, but return values instead of executing statements and returning nothing.

*Objects and Object Relational Data*

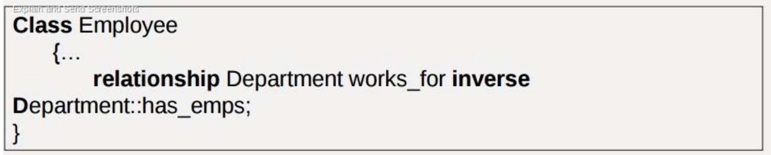
- OOL supports a grouping operation called group by similar to SQL. It allows for the ordering of the selection results. The group by command is followed by the group defining attribute, then a colon, then the sorting attribute.
- To map EER to ODL, we create an ODL class for each EER entity type or subclass. Multivalued attributes are declared by sets, bags or lists constructors. Composite attributes are mapped into tuple constructors.
- We add relationship properties or reference attributes for each binary relationship into the ODL classes being in the relationship.
- We add appropriate operations for each class, as well as constructors and destructors.
- We specify inheritance relationships using the extends clause.

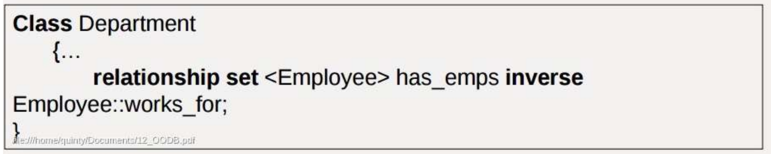
- We map weak entity types in the same way as regular entities, but if they have no relationships they just become multivalued attributes of the owner entity type.
- We map union types to ODL, and n-ary relationships into separate classes.

- When apps interface to an OODB, the app sees the data stored almost as if they were objects. It makes a strong connection between app and db, with natural data structures.
- However, these DB can perform poorly, and optimizing queries for OODMS is really hard. They can't scale well either.
- Objects and literals are the basic building blocks of **object model**, analogous to the relational model. Objects have an object ID and a state, and their state can change. Literals have a value/state, no ID and cannot change as they are constant.
- Going further, an **object** has the OID, an optional name, and a structure that describes how it is constructed. It can be atomic, or a collection with a type t, key k, and v value. It can be a Set, Bag, List, or Array. Objects are specified by an interface.
- We can use a primary key attribute in an object which makes for easier interpretation. The OID is not visible to the user. It cannot be reused, and is commonly derived from a hash involving longs.
- **Literals** are described by a state or value, have no OID, and can have a complex structure. Atomic literals can be simple, like the int value 102, structured, like the Data(19,May,2001), or Collections, like a List.
- Atomic Objects can be defined by users! These are specified using the keyword class in ODL. Properties of these are attributes and relationships (kind of like columns and foreign keys), and operations are the behaviors they have.
- Objects can have a type structure that contains all of the needed info that describes the object or literal. Info about a complex object in a relational DB can be scattered over many relations. A complex type is made in an ODB using a nesting of type constructors, the atom, struct/tuple, and collection.
- Atomic objects support all basic atomic values like ints, strings and the like that are directly supported by the DB system.
- A tuple object is usually an object made up of multiple attributes defined in any way. Think of it like a collection of objects. Every complex object in a tuple has the value of that object's OID.
- A collection object is just that collection object containing any type of object.
-
- 

  Department

  ```
  Class Department
  ( extent    all_departments
     key      dname, dnumber )
  {  attribute     string                 dname;
     attribute     short                  dnumber;
     attribute     struct Dept_Mgr{Employee manager,  date startdate}
                                          mgr;
     attribute     set<string>            locations;
     attribute struct Projs {string projname, time weekly_hours}
                                          projs;
     relationship set<Employee>           has_emps inverse Employee::works_for;
     void     add_emp(in string new_ename) raises(ename_not_valid);
     void     change_manager(in string new_mgr_name) in date startdate);
  }
  ```

- Above is an example of an object definition. We can see the atom types dname, and dnumber. Dept_Mgr is a tuple, with manager and startdate, and locations is a collection attribute.
- 

  ```
  Class Employee
       {...
            relationship Department works_for inverse
  Department::has_emps;
  }
  ```
- 

  ```
  Class Department
       {...
            relationship set <Employee> has_emps inverse
  Employee::works_for;
  }
  ```

- The above picture focuses on the relationship definitions in an object. Usually, only binary relationships are explicitly represented. Inverse references are always given, to ensure the referential integrity of the relationship. In the above example, if the Employee e has their works for value set to Department d, then Department D has to have Employee e in their set of

Employees. Simple.

- If a relationship is only one way, then we set that as an attribute with the type being the object it refers to.
- You can have interfaces and subclasses in OODBs. To specify an interface, write class Triangle: Shape {} to set Shape as the interface for Triangle. To specify a superclass, write class Poodle extends Dog.
- The Object Query Language has similar syntax to SQL, and has additional features. To write a select from where, we first have to define an extent name in a Class so the OQL can have an entry part. So to get the names of all department names in the Engi college, just write SELECT d.name FROM d in all_departments WHERE d.college = 'Engi'. In the FROM clause, we can also use all_departments d or all_departments as d.
- The query results can be of any type defined in the ODMG model. It will be a collection of those types of objects. To select just a single element from those objects, use the element operator.
- Basically, to find information in the OOQL, you do

  ```
  SELECT variableName.stringofrelationshipstodesiredattribute.attributeyouwant
  FROM variableName in entryPointIntoStringOfRelationships
  WHERE whatever
  ```

- To create a view, use the define keyword which specifies an ID for a named query. For example, define has_minor(name) as *query* will save a view that takes in a name argument.
- You can use ordered by to order by the specified attribute.

*Mapping ER to ODL*

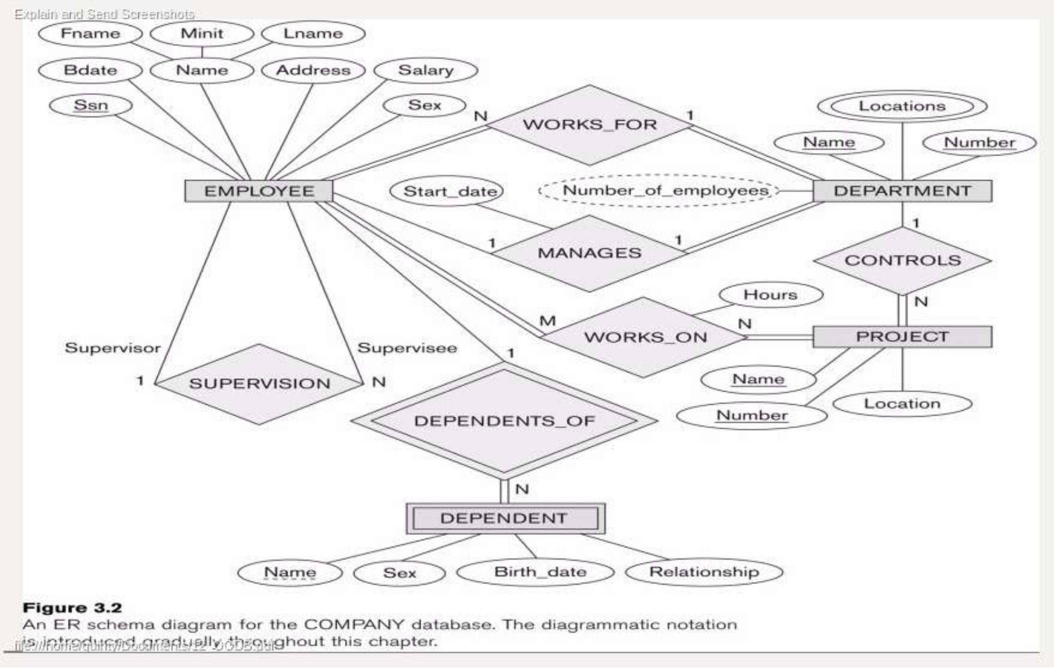
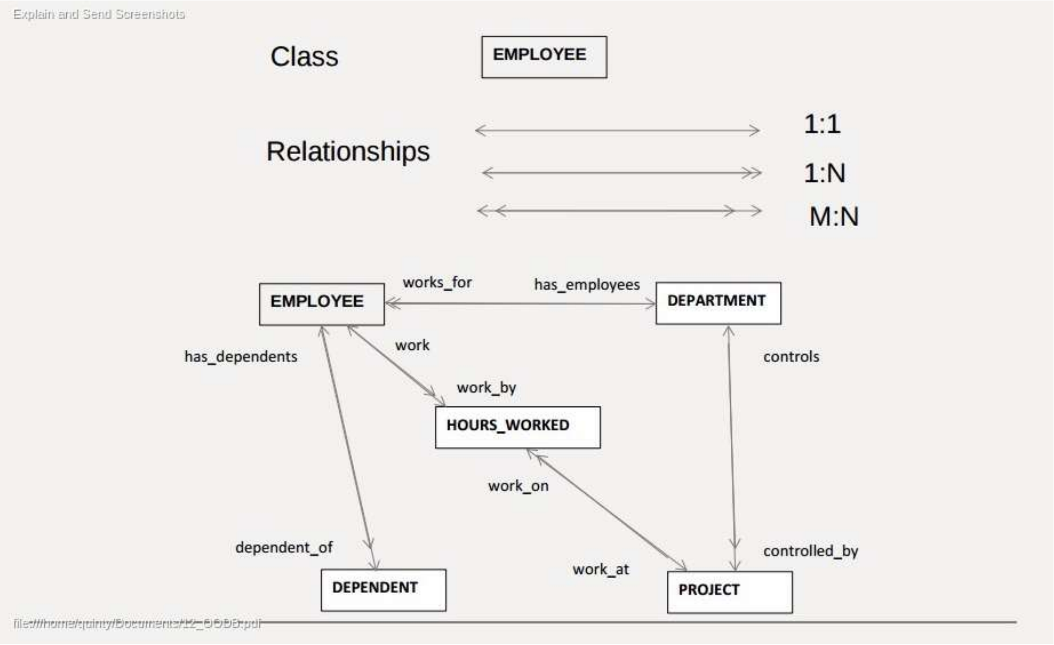
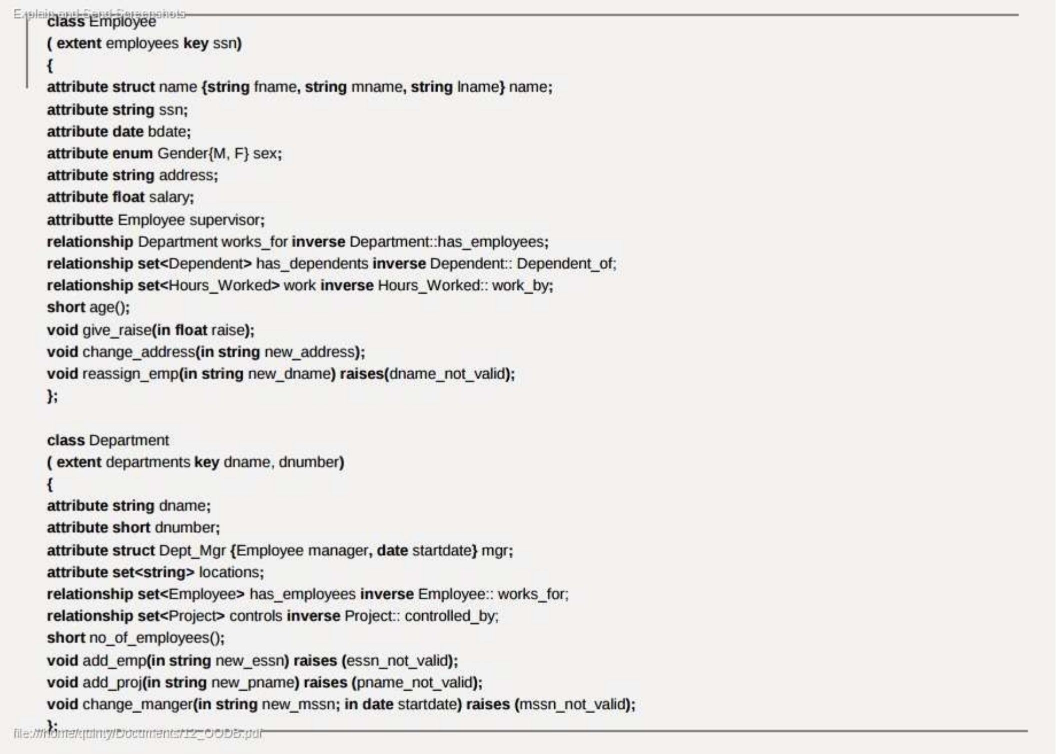
- First, we create an ODL class for each ER entity type or subclass. Multivalued attributes are declared by collections, and composite attributes are mapped into tuple constructors. Regular attributes become atomic object attributes.
- Next, we add relationship properties or reference attributes for each binary relationship into the ODL classes in that relationship. If it is 1:1 or N:1, the relationship is atomic type, it is 1:N or M:N, it is set typed object.
- We add appropriate operations/methods for each class, including constructors and destructors. These aren't in the original diagram, make some shit up.
- Next, if an EER diagram, every subclass will extend its superclass, and must copy the superclasses attributes and methods. Follow 1-3 to fill out the rest of the subclass.
- Weak entities are mapped like regular entities, but if they do not participate in any relationships can become tuples of the owner entity type.
- N-Ary relationships with degree greater than 2 have each relationship mapped into a separate class with appropriate references to each participating class.
-
- 

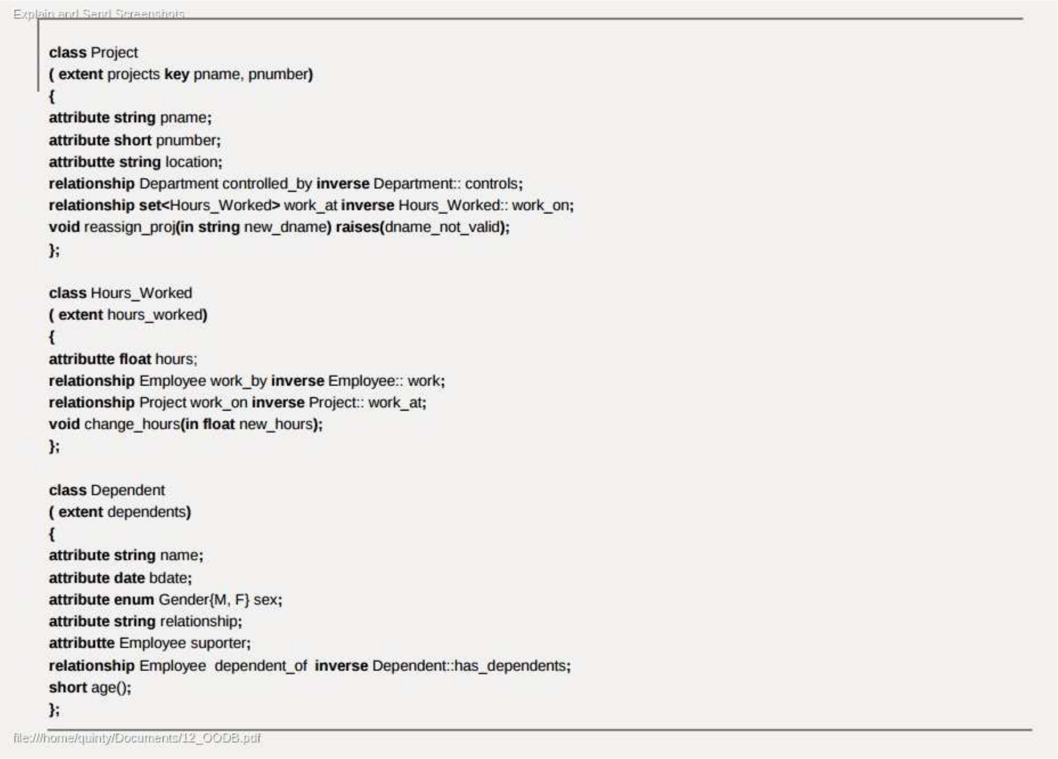
  **Figure 3.2**
  An ER schema diagram for the COMPANY database. The diagrammatic notation is introduced gradually throughout this chapter.
-

- 

- 
- 

```
class Employee
( extent employees key ssn)
{
attribute struct name {string fname, string mname, string lname} name;
attribute string ssn;
attribute date bdate;
attribute enum Gender{M, F} sex;
attribute string address;
attribute float salary;
attributte Employee supervisor;
relationship Department works_for inverse Department::has_employees;
relationship set<Dependent> has_dependents inverse Dependent:: Dependent_of;
relationship set<Hours_Worked> work inverse Hours_Worked:: work_by;
short age();
void give_raise(in float raise);
void change_address(in string new_address);
void reassign_emp(in string new_dname) raises(dname_not_valid);
};

class Department
( extent departments key dname, dnumber)
{
attribute string dname;
attribute short dnumber;
attribute struct Dept_Mgr {Employee manager, date startdate} mgr;
attribute set<string> locations;
relationship set<Employee> has_employees inverse Employee:: works_for;
relationship set<Project> controls inverse Project:: controlled_by;
short no_of_employees();
void add_emp(in string new_essn) raises (essn_not_valid);
void add_proj(in string new_pname) raises (pname_not_valid);
void change_manger(in string new_mssn; in date startdate) raises (mssn_not_valid);
};
```

Explain and Send Screenshots
file:///home/quinty/Documents/12_OODB.pdf
-

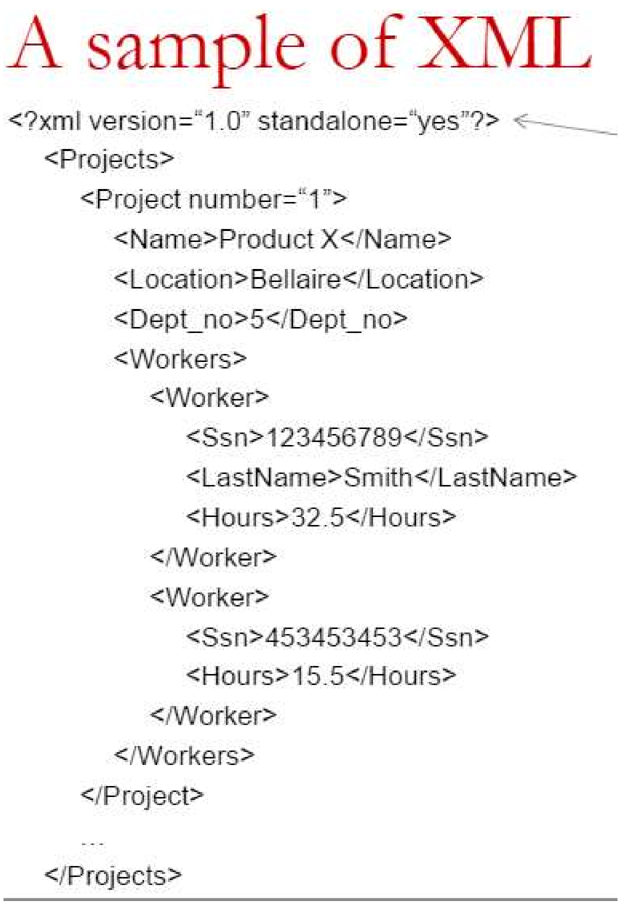
- 

Explain and Send Screenshots

```
class Project
( extent projects key pname, pnumber)
{
attribute string pname;
attribute short pnumber;
attributte string location;
relationship Department controlled_by inverse Department:: controls;
relationship set<Hours_Worked> work_at inverse Hours_Worked:: work_on;
void reassign_proj(in string new_dname) raises(dname_not_valid);
};

class Hours_Worked
( extent hours_worked)
{
attributte float hours;
relationship Employee work_by inverse Employee:: work;
relationship Project work_on inverse Project:: work_at;
void change_hours(in float new_hours);
};

class Dependent
( extent dependents)
{
attribute string name;
attribute date bdate;
attribute enum Gender{M, F} sex;
attribute string relationship;
attributte Employee suporter;
relationship Employee  dependent_of  inverse Dependent::has_dependents;
short age();
};
```

file:///home/quinty/Documents/12_OODB.pdf

- XML is the eXtensible Markup Language, which is used to annotate a document for a purpose. It uses tags that are distinguished from the content of the document to provide that annotation.
- It provides a language for structured data and semi structured data. **Structured Data** is data that is represented in a strict format, like in a database. **Semi structured data** is data that has some structure, but it's more loosely defined. This means that there's no predefined schema, and identical type entities can have different amounts of attributes.
- Unstructured data is data with no external structure at all. XML provides a framework to define. An XML document is a collection of related data items. These items are made with tags known as elements, which provide structure. A simple element has data values, a complex element has other elements as well as data values, and elements can also have attributes.

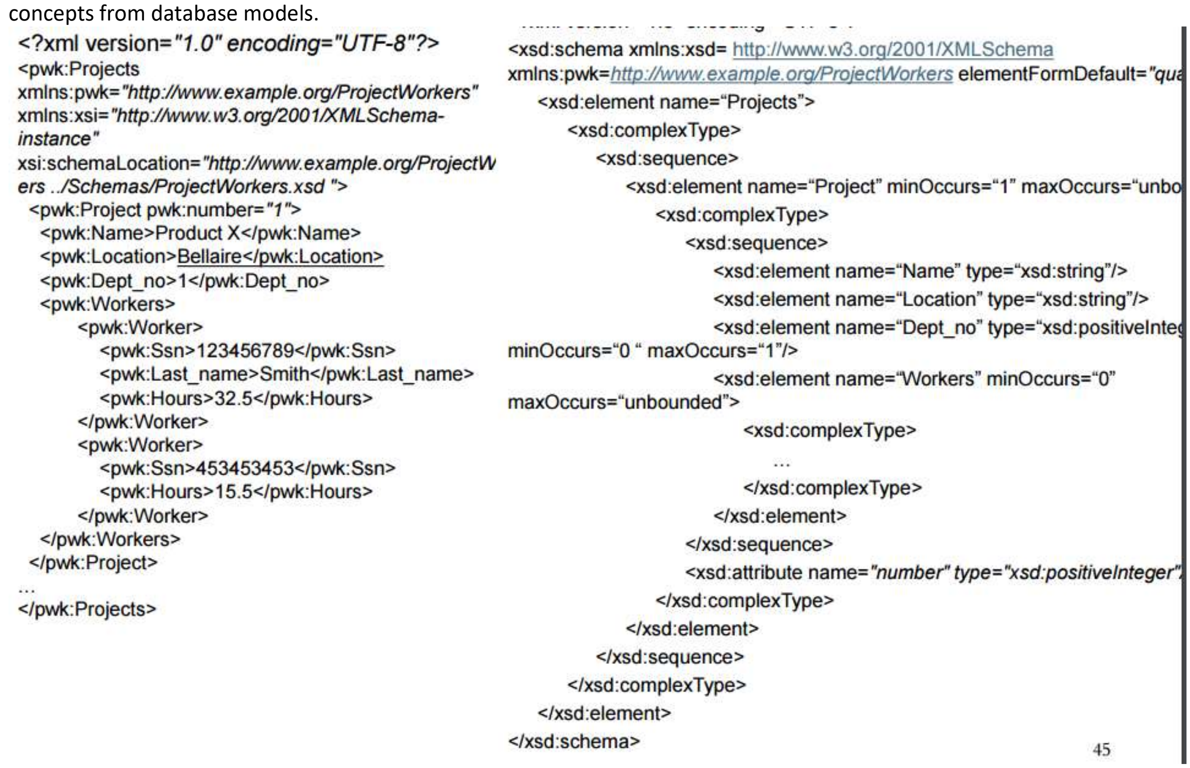


# A sample of XML

```
<?xml version="1.0" standalone="yes"?>
  <Projects>
    <Project number="1">
      <Name>Product X</Name>
      <Location>Bellaire</Location>
      <Dept_no>5</Dept_no>
      <Workers>
        <Worker>
          <Ssn>123456789</Ssn>
          <LastName>Smith</LastName>
          <Hours>32.5</Hours>
        </Worker>
        <Worker>
          <Ssn>453453453</Ssn>
          <Hours>15.5</Hours>
        </Worker>
      </Workers>
    </Project>
    ...
  </Projects>
```

- XML uses a **hierarchical model**, also known as a tree model. Documents can be represented as trees. Each simple element contains one data value, and is a leaf of the tree. The complex elements can contain multiple child elements, which are internal nodes in the tree. Each complex element can belong to one complex parent element, the parent node of the tree. The root contains everything else.
- Best practice is to use attributes for info that describes/modifies the element. You use an element to hold the actual data values.
- There are three types of XML documents:
  - Data Centric XML, which have many small items and is highly structured. This kind of XML is used for data exchange purposes, and is also used to create web pages dynamically from databases. It follows a schema document that determines their structure.
    - A schema document define names for expected XML tags/elements, define which elements are required and which are optional, and define which elements can be contained within other elements.
    - These docs are used to communicate semantic info about an XML document.
  - Document centric XML, which has large amounts of text, a few structural elements, and doesn't need a pre-defined schema to work.
  - Hybrid XML has some parts that are highly structured, some parts unstructured, and may/may not have a predefined schema. Those without schemas as well as all XML docs without them are called **schemaless**. This property is determined by the attribute 'standalone=yes' in the XML declaration in the top line.
- A **well-formed XML doc** has the XML declaration line (<?xml version="1.0" standalone="yes"?>), and forms a tree with one root element and every child element having start and end tags within a parent element. Well-formed XML can be processed by generic processors, like DOM and SAX (Simple API for XML).
- **Valid** XML is well formed, and follows a particular schema defined by a DTD document, or an XML schema document.
- DTD was the original method of specifying a schema definition. Each possible element is defined, with the kids it has to have, which it can have, which attributes it needs and can have, and the values of the leaf elements it can have if they exist.
- Data types in DTD are not general, meaning that child nodes all have to hold PCDATA values, which are strings. DTD also has its own syntax, meaning we have to write parsers aside from XML parsers to read it. Another bad thing is that all DTD elements must follow the ordering laid out in the DTD, meaning unordered elements are not allowed.
- XML schema documents provide a way to specify an XML schema using XML as a language, overcoming the limitations of DTD. These are much more complicated than DTDs, and have a tree model and

concepts from database models.

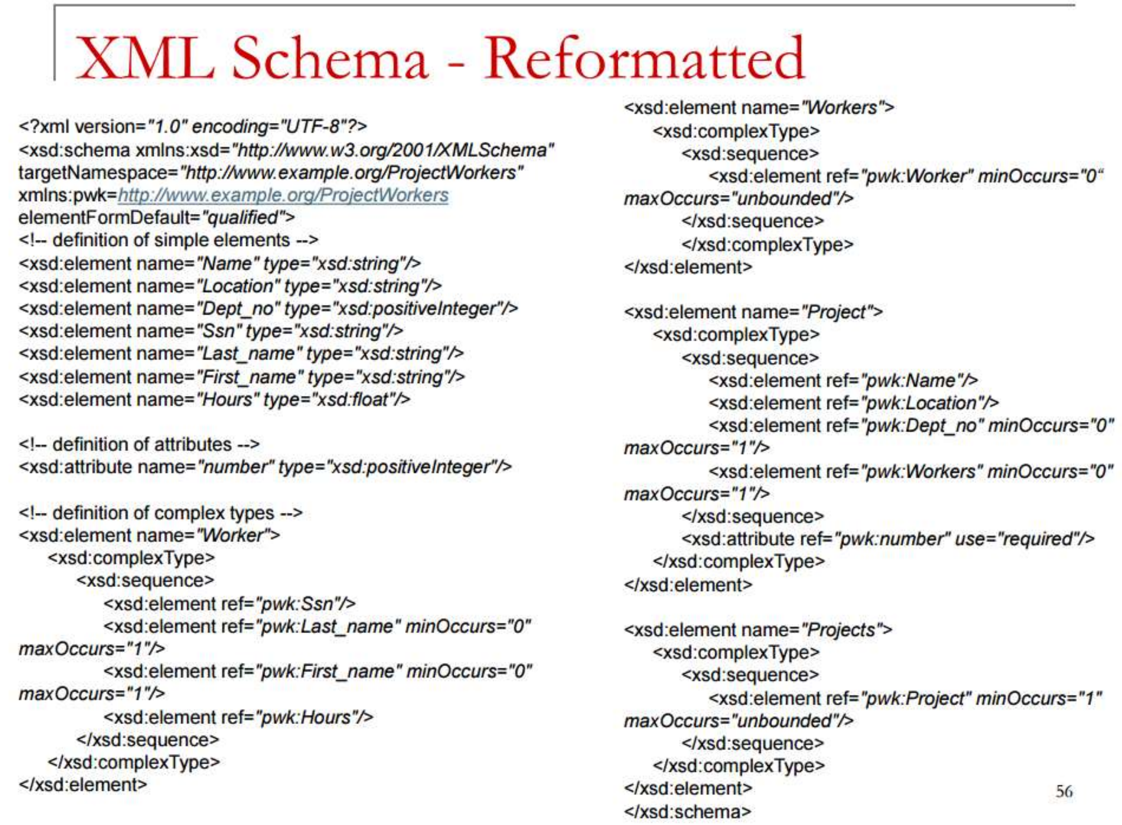
- 

```
<?xml version="1.0" encoding="UTF-8"?>
<pwk:Projects
xmlns:pwk="http://www.example.org/ProjectWorkers"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance"
xsi:schemaLocation="http://www.example.org/ProjectW
ers ../Schemas/ProjectWorkers.xsd ">
  <pwk:Project pwk:number="1">
    <pwk:Name>Product X</pwk:Name>
    <pwk:Location>Bellaire</pwk:Location>
    <pwk:Dept_no>1</pwk:Dept_no>
    <pwk:Workers>
        <pwk:Worker>
          <pwk:Ssn>123456789</pwk:Ssn>
          <pwk:Last_name>Smith</pwk:Last_name>
          <pwk:Hours>32.5</pwk:Hours>
        </pwk:Worker>
        <pwk:Worker>
          <pwk:Ssn>453453453</pwk:Ssn>
          <pwk:Hours>15.5</pwk:Hours>
        </pwk:Worker>
    </pwk:Workers>
  </pwk:Project>
...
</pwk:Projects>
```

```
<xsd:schema xmlns:xsd= http://www.w3.org/2001/XMLSchema
xmlns:pwk=http://www.example.org/ProjectWorkers elementFormDefault="qua
    <xsd:element name="Projects">
        <xsd:complexType>
            <xsd:sequence>
                <xsd:element name="Project" minOccurs="1" maxOccurs="unbo
                    <xsd:complexType>
                        <xsd:sequence>
                            <xsd:element name="Name" type="xsd:string"/>
                            <xsd:element name="Location" type="xsd:string"/>
                            <xsd:element name="Dept_no" type="xsd:positiveInte
minOccurs="0 " maxOccurs="1"/>
                            <xsd:element name="Workers" minOccurs="0"
maxOccurs="unbounded">
                                <xsd:complexType>
                                    ...
                                </xsd:complexType>
                            </xsd:element>
                        </xsd:sequence>
                        <xsd:attribute name="number" type="xsd:positiveInteger"
                    </xsd:complexType>
                </xsd:element>
            </xsd:sequence>
        </xsd:complexType>
    </xsd:element>
</xsd:schema>
```

45

- The first line in the schema defines both the schema element, and the namespace we will be defining.
- You define an element and its type name using the xsd:element tag.
- If you have a complex element, you use the xsd:complexType tag to show this after an element.
- You can define a sequence of child elements for the parent element using xsd:sequence.
- You can define a minimum and max amount of occurrences of a type of element when it is first defined.
- To define the attributes on an element, you would have an xsd:attribute tag right before it's closing tag for the complex type tag that falls beneath its element tag.
- Simple elements are defined with just a name and a type in a single element tag.

- 

## XML Schema - Reformatted

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
targetNamespace="http://www.example.org/ProjectWorkers"
xmlns:pwk=http://www.example.org/ProjectWorkers
elementFormDefault="qualified">
<!-- definition of simple elements -->
<xsd:element name="Name" type="xsd:string"/>
<xsd:element name="Location" type="xsd:string"/>
<xsd:element name="Dept_no" type="xsd:positiveInteger"/>
<xsd:element name="Ssn" type="xsd:string"/>
<xsd:element name="Last_name" type="xsd:string"/>
<xsd:element name="First_name" type="xsd:string"/>
<xsd:element name="Hours" type="xsd:float"/>

<!-- definition of attributes -->
<xsd:attribute name="number" type="xsd:positiveInteger"/>

<!-- definition of complex types -->
<xsd:element name="Worker">
    <xsd:complexType>
        <xsd:sequence>
            <xsd:element ref="pwk:Ssn"/>
            <xsd:element ref="pwk:Last_name" minOccurs="0"
maxOccurs="1"/>
            <xsd:element ref="pwk:First_name" minOccurs="0"
maxOccurs="1"/>
            <xsd:element ref="pwk:Hours"/>
        </xsd:sequence>
    </xsd:complexType>
</xsd:element>
```

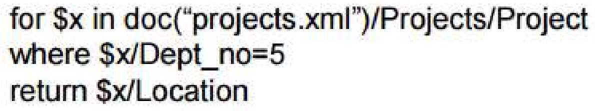
```
<xsd:element name="Workers">
    <xsd:complexType>
        <xsd:sequence>
            <xsd:element ref="pwk:Worker" minOccurs="0"
maxOccurs="unbounded"/>
        </xsd:sequence>
        </xsd:complexType>
</xsd:element>

<xsd:element name="Project">
    <xsd:complexType>
        <xsd:sequence>
            <xsd:element ref="pwk:Name"/>
            <xsd:element ref="pwk:Location"/>
            <xsd:element ref="pwk:Dept_no" minOccurs="0"
maxOccurs="1"/>
            <xsd:element ref="pwk:Workers" minOccurs="0"
maxOccurs="1"/>
        </xsd:sequence>
        <xsd:attribute ref="pwk:number" use="required"/>
    </xsd:complexType>
</xsd:element>

<xsd:element name="Projects">
    <xsd:complexType>
        <xsd:sequence>
            <xsd:element ref="pwk:Project" minOccurs="1"
maxOccurs="unbounded"/>
        </xsd:sequence>
    </xsd:complexType>
</xsd:element>
</xsd:schema>
```

56

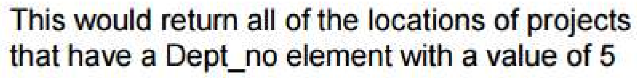
- There are 7 different query languages proposed for XML, but the three major ones are XPath, XQuery, and XSLT.
- Xpath is a simple query language used to select parts of an XML doc. These queries are written as paths

through an XML document. Nodes are separated by /, and the result returned is whatever is at the end of the Xpath expression.

- Something like /Projects/Project[1] would select all the children of the first element with the name project under the projects root node. You can also specify things like /Projects/Project/Workers/Worker[Hours>20.0] to select the children of the workers who have a value more than 20 in their Hours element.
- Selecting based on an attribute is done like: /Projects/Project/[@number=1]/Workers, which will grab all the Workers elements from the project with attribute number 1. Without an equals sign, Xpath will grab all the number attributes from all the project elements.
- You can use the OR bar | to select multiple elements at once.
- Xquery is an extension of Xpath, uses the same model as Xpath, but provides a language for more complex queries. It uses FLWOR expressions, meaning For, Let, Where, Orderby, Return.



```
for $x in doc("projects.xml")/Projects/Project
where $x/Dept_no=5
return $x/Location
```

- 

  This would return all of the locations of projects that have a Dept_no element with a value of 5