The Advanced Database Management System

By

Bishnu Gautam

New Summit College

The Relational Model

The Relational Data Model has the relation at its heart but it is a whole series of rules governing:

- Keys
- Relationship
- Joins
- Functional dependencies
- Transitive depndencies
- Multi-valued dependencies
- Modification anomalies

Relation

The *Relation* is the basic element in a relational data model and its subject to these rules:

- Relation (file, table) is a two-dimensional table.
- Attribute (i.e. field or data item) is a **column** in the table.
- Each column in the table has a unique name within that table.
- Each column is homogeneous.
- Each column has a **domain**,
- A Tuple (i.e. record) is a row in the table.
- The order of the rows and columns is not important.
- Values of a row all relate to some thing or portion of a thing.
- Repeating groups are not allowed.
- Duplicate rows are not allowed
- Cells must be single-valued

Special Notation of Relation

A relation may be expressed using the notation R(<u>A</u>,B,C, ...) where:

- R = the name of the relation.
- (A,B,C, ...) = the attributes within the relation.
- <u>A</u> = the attribute(s) which form the primary key.

Keys

A **simple** key contains a single attribute.

A **composite key** is a key that contains more than one attribute.

A **candidate key** is an attribute (or set of attributes) that uniquely identifies a row. A candidate key must possess the following properties:

- Unique identification - For every row the value of the key must uniquely identify that row.
- Non redundancy - No attribute in the key can be discarded without destroying the property of unique identification.

A **primary key** is the candidate key which is selected as the principal unique identifier. Every relation must contain a primary key. The primary key is usually the key selected to identify a row when the database is physically implemented. For example, a part number is selected instead of a part description.

A **superkey** is any set of attributes that uniquely identifies a row. A superkey differs from a candidate key in that it does not require the non redundancy property.

Key….

A **foreign key** is an attribute (or set of attributes) that appears (usually) as a non key attribute in one relation and as a primary key attribute in another relation.

- A many-to-many relationship can only be implemented by introducing an intersection or link table which then becomes the child in two one-to-many relationships. The intersection table therefore has a foreign key for each of its parents, and its primary key is a composite of both foreign keys.
- A one-to-one relationship requires that the child table has no more than one occurrence for each parent, which can only be enforced by letting the foreign key also serve as the primary key

Keys…

A semantic or **natural** key is a key for which the possible values have an obvious meaning to the user or the data. For example a COUNTRY entity might contain the value 'USA' for the occurrence describing the United States of America.

A technical or **surrogate** or **artificial** key is a key for which the possible values have no obvious meaning to the user or the data. These are used instead of semantic keys for any of the following reasons:

- When the value in a semantic key is likely to be changed by the user, or can have duplicates. For example, on a PERSON table it is unwise to use PERSON_NAME as the key as it is possible to have more than one person with the same name, or the name may change such as through marriage.
- When none of the existing attributes can be used to guarantee uniqueness. In this case adding an attribute whose value is generated by the system, e.g from a sequence of numbers, is the only way to provide a unique value. Typical examples would be ORDER_ID and INVOICE_ID. The value '12345' has no meaning to the user as it conveys nothing about the entity to which it relates.

Keys…

A key functionally determines the other attributes in the row, thus it is always a determinant.

Note that the term 'key' in most DBMS engines is implemented as an index which does not allow duplicate entries.

The Relationship

One table (relation) may be linked with another in what is known as a **relationship**.

- A relationship is between two tables in what is known as a **one-to-many** or **parent-child** or **master-detail** relationship.
- It is possible for a record on the **parent** table to exist without corresponding records on the **child** table, but it should not be possible for an entry on the **child** table to exist without a corresponding entry on the **parent** table.
- A **child** record without a corresponding **parent** record is known as an **orphan**.
- It is possible for a table to be related to itself. For this to be possible it needs a **foreign key** which points back to the **primary key**.
- A **table** may be the subject of any number of relationships, and it may be the **parent** in some and the **child** in others.
- Some database engines allow a **parent** table to be linked via a **candidate key**, but if this were changed it could result in the link to the **child** table being broken.
- Some database engines allow relationships to be managed by rules known as **referential integrity** or **foreign key restraints**.

Relational Joins

- The join operator is used to combine data from two or more relations (tables) in order to satisfy a particular query.
- Two relations may be joined when they share at least one common attribute.
- The join is implemented by considering each row in an instance of each relation.
- A row in relation R1 is joined to a row in relation R2 when the value of the common attribute(s) is equal in the two relations.
- The join of two relations is often called a **binary join.**
- The join of two relations creates a new relation.
- The notation **R1 x R2** indicates the join of relations R1 and R2. For example, consider the following:

Eg.

Note that the instances of relation R1 and R2 contain the same data values for attribute B.

Data normalisation is concerned with decomposing a relation (e.g. R(A,B,C,D,E) into smaller relations (e.g. R1 and R2).

The data values for attribute B in this context will be identical in R1 and R2.

The instances of R1 and R2 are projections of the instances of R(A,B,C,D,E) onto the attributes (A,B,C) and (B,D,E) respectively.

A projection will not eliminate data values - duplicate rows are removed, but this will not remove a data value from any attribute

Relation R1

A	B	C
1	5	3
2	4	5
8	3	5
9	3	3
1	6	5
5	4	3
2	7	5

Relation R2

B	D	E
4	7	4
6	2	3
5	7	8
7	2	3
3	2	2

The join of relations R1 and R2 is possible because B is a common attribute.

The result of the join is:

Relation R1 x R2

A	B	C	D	E
1	5	3	7	8
2	4	5	7	4
8	3	5	2	2
9	3	3	2	2
1	6	5	2	3
5	4	3	7	4
2	7	5	2	3

Relation

- The row (2 4 5 7 4) was formed by joining the row (2 4 5) from relation R1 to the row (4 7 4) from relation R2.
- The two rows were joined since each contained the same value for the common attribute B.
- The row (2 4 5) was not joined to the row (6 2 3) since the values of the common attribute (4 and 6) are not the same.
- The relations joined in the preceding example shared exactly one common attribute.
- However, relations may share multiple common attributes. All of these common attributes must be used in creating a join.
- For example, the instances of relations R1 and R2 in the following example are joined using the common attributes B and C

Before join R1 and R2

After join R1 and R2

Relation R1

A	B	C
6	1	4
8	1	4
5	1	2
2	7	1

Relation R2

B	C	D
1	4	9
1	4	2
1	2	1
7	1	2
7	1	3

The row (6 1 4 9) was formed by joining the row (6 1 4) from relation R1 to the row (1 4 9) from relation R2.
The join was created since the common set of attributes (B and C) contained identical values (1 and 4)
The row (6 1 4) from R1 was not joined to the row (1 2 1) from R2 since the common attributes did not share identical values - (1 4) in R1 and (1 2) in R2.

Relation R1 x R2

A	B	C	D
6	1	4	9
6	1	4	2
8	1	4	9
8	1	4	2
5	1	2	1
2	7	1	2
2	7	1	3

The join operation provides a method for reconstructing a relation that was decomposed into two relations during the normalisation process.

The join of two rows, however, can create a new row that was not a member of the original relation.

Thus invalid information can be created during the join process.

Lossless Joins

- A set of relations satisfies the lossless join property if the instances can be joined without creating invalid data (i.e. new rows).
- A join that is not lossless will contain extra, invalid rows.
- Thus the term **gainless join** might be more appropriate.
- To give an example of incorrect information created by an invalid join let us take the following data structure:
- **R(student, course, instructor, hour, room, grade)**

Dependences

Assuming that only one section of a class is offered during a semester we can define the following functional dependencies:

- (HOUR, ROOM)   COURSE
- (COURSE, STUDENT)   GRADE
- (INSTRUCTOR, HOUR)   ROOM
- (COURSE)   INSTRUCTOR
- (HOUR, STUDENT)   ROOM

Dependencies and Relationship

STUDENT	COURSE	INSTRUCTOR	HOUR	ROOM	GRADE
Smith	Math 1	Jenkins	8:00	100	A
Jones	English	Goldman	8:00	200	B
Brown	English	Goldman	8:00	200	C
Green	Algebra	Jenkins	9:00	400	A

The following four relations, each in 4th normal form, can be generated from the given and implied dependencies:

R1(STUDENT, HOUR, COURSE)
R2(STUDENT, COURSE, GRADE)
R3(COURSE, INSTRUCTOR)
R4(INSTRUCTOR, HOUR, ROOM)

Note that the dependencies (HOUR, ROOM)   COURSE and (HOUR, STUDENT)   ROOM are not explicitly represented in the preceding decomposition.
The goal is to develop relations in 4th normal form that can be joined to answer any ad hoc inquiries correctly.
This goal can be achieved without representing every functional dependency as a relation.
Furthermore, several sets of relations may satisfy the goal.

Preceding set of relation

R1		
STUDENT	HOUR	COURSE
Smith	8:00	Math 1
Jones	8:00	English
Brown	8:00	English
Green	9:00	Algebra

R2		
STUDENT	COURSE	GRADE
Smith	Math 1	A
Jones	English	B
Brown	English	C
Green	Algebra	A

R3		
COURSE	INSTRUCTOR	
Math 1	Jenkins	
English	Goldman	
Algebra	Jenkins	

R4		
INSTRUCTOR	HOUR	ROOM
Jenkins	8:00	100
Goldman	8:00	200
Jenkins	9:00	400

Now suppose that a list of courses with their corresponding room numbers is required.

Relations R1 and R4 contain the necessary information and can be joined using the attribute HOUR.

The result of this join is:

R1 x R4

STUDENT	COURSE	INSTRUCTOR	HOUR	ROOM
Smith	Math 1	Jenkins	8:00	100
Smith	Math 1	Goldman	8:00	200
Jones	English	Jenkins	8:00	100
Jones	English	Goldman	8:00	200
Brown	English	Jenkins	8:00	100
Brown	English	Goldman	8:00	200
Green	Algebra	Jenkins	9:00	400

- This join creates the following invalid information (denoted by the coloured rows):
- Smith, Jones, and Brown take the same class at the same time from two different instructors in two different rooms.
- Jenkins (the Maths teacher) teaches English.
- Goldman (the English teacher) teaches Maths.
- Both instructors teach different courses at the same time.

- Another possibility for a join is R3 and R4 (joined on INSTRUCTOR).
- The result would be:

R3 x R4			
COURSE	INSTRUCTOR	HOUR	ROOM
Math 1	Jenkins	8:00	100
Math 1	Jenkins	9:00	400
English	Goldman	8:00	200
Algebra	Jenkins	8:00	100
Algebra	Jenkins	9:00	400

This join creates the following invalid information:

Jenkins teaches Math 1 and Algebra simultaneously at both 8:00 and 9:00.

A correct sequence is to join R1 and R3 (using COURSE) and then join the resulting relation with R4 (using both INSTRUCTOR and HOUR).

The result would be:

R1 x R3			
STUDENT	COURSE	INSTRUCTOR	HOUR
Smith	Math 1	Jenkins	8:00
Jones	English	Goldman	8:00
Brown	English	Goldman	8:00
Green	Algebra	Jenkins	9:00

(R1 x R3) x R4				
STUDENT	COURSE	INSTRUCTOR	HOUR	ROOM
Smith	Math 1	Jenkins	8:00	100
Jones	English	Goldman	8:00	200
Brown	English	Goldman	8:00	200
Green	Algebra	Jenkins	9:00	400

Extracting the COURSE and ROOM attributes (and eliminating the duplicate row produced for the English course) would yield the desired result:

COURSE	ROOM
Math 1	100
English	200
Algebra	400

The correct result is obtained since the sequence (R1 x r3) x R4 satisfies the lossless (gainless?) join property

- A relational database is in 4th normal form when the lossless join property can be used to answer unanticipated queries.
- However, the choice of joins must be evaluated carefully.
- Many different sequences of joins will recreate an instance of a relation.
- Some sequences are more desirable since they result in the creation of less invalid data during the join operation.
- Suppose that a relation is decomposed using functional dependencies and multi-valued dependencies.
- Then at least one sequence of joins on the resulting relations exists that recreates the original instance with no invalid data created during any of the join operations.

For example, suppose that a list of grades by room number is desired.

This question, which was probably not anticipated during database design, can be answered without creating invalid data by either of the following two join sequences:

R1 x R3	R1 x R3
(R1 x R3) x R2	(R1 x R3) x R4
((R1 x R3) x R2) x R4	((R1 x R3) x R4) x R2

The required information is contained with relations R2 and R4, but these relations cannot be joined directly.

In this case the solution requires joining all 4 relations.

Functional Dependency

- The database may require a 'lossless join' relation, which is constructed to assure that any ad hoc inquiry can be answered with relational operators.
- This relation may contain attributes that are not logically related to each other.
- This occurs because the relation must serve as a bridge between the other relations in the database.
- For example, the lossless join relation will contain all attributes that appear only on the left side of a functional dependency.
- Other attributes may also be required, however, in developing the lossless join relation.
- Consider relational schema R(A, B, C, D), A B and C D.
- Relations Rl($\underline{A}$, B) and R2($\underline{C}$, D) are in 4th normal form.
- A third relation R3($\underline{A}$, $\underline{C}$), however, is required to satisfy the lossless join property.
- This relation can be used to join attributes B and D.
- This is accomplished by joining relations R1 and R3 and then joining the result to relation R2.
- No invalid data is created during these joins. The relation R3($\underline{A}$, $\underline{C}$) is the lossless join relation for this database design.

Multi-valued Dependencies

- A relation is usually developed by combining attributes about a particular subject or entity.
- The lossless join relation, however, is developed to represent a relationship among various relations.
- The lossless join relation may be difficult to populate initially and difficult to maintain - a result of including attributes that are not logically associated with each other.
- The attributes within a lossless join relation often contain multi-valued dependencies.
- Consideration of 4th normal form is important in this situation.
- The lossless join relation can sometimes be decomposed into smaller relations by eliminating the multi-valued dependencies.
- These smaller relations are easier to populate and maintain.

Determinant and Dependent

- The expression X→Y means 'if I know the value of X, then I can obtain the value of Y' (in a table or somewhere).
- In the expression X→Y, X is the **determinant** and Y is the **dependent** attribute.
- The value X **determines** the value of Y.
- The value Y **depends on** the value of X.

Functional Dependencies (FD)

- An attribute is functionally dependent if its value is determined by another attribute *which is a key*.
- If we know the value of one (or several) data items, then we can find the value of another (or several).
- Functional dependencies are expressed as X→Y, where X is the determinant and Y is the functionally dependent attribute.
- If A→ (B,C) then A→B and A→C.
- If (A,B) →C, then it is not necessarily true that A→C and B→C.
- If A→B and B→A, then A and B are in a 1-1 relationship.
- If A→B then for A there can only ever be one value for B.

Transitive Dependencies (TD)

- An attribute is transitively dependent if its value is determined by another attribute *which is not a key*.
- If X→Y and X is not a key then this is a transitive dependency.
- A transitive dependency exists when A→B→C but NOT A→C.

Multi-Valued Dependencies (MVD)

- A table involves a multi-valued dependency if it may contain multiple values for an entity.
- A multi-valued dependency may arise as a result of enforcing 1st normal form.
- X →→Y, ie X multi-determines Y, when for each value of X we can have more than one value of Y.
- If A→→B and A→→C then we have a single attribute A which multi-determines two other independent attributes, B and C.
- If A→→(B,C) then we have an attribute A which multi-determines a set of associated attributes, B and C.

Join Dependencies (JD)

- If a table can be decomposed into three or more smaller tables, it must be capable of being joined again on common keys to form the original table

Modification Anomalies

A major objective of data normalisation is to avoid modification anomalies. These come in two flavours:

- An **insertion anomaly** is a failure to place information about a new database entry into all the places in the database where information about that new entry needs to be stored.
- In a properly normalized database, information about a new entry needs to be inserted into only one place in the database.
- In an inadequately normalized database, information about a new entry may need to be inserted into more than one place, and, human fallibility being what it is, some of the needed additional insertions may be missed.
- A **deletion anomaly** is a failure to remove information about an existing database entry when it is time to remove that entry.
- In a properly normalized database, information about an old, to-be-gotten-rid-of entry needs to be deleted from only one place in the database.

Anomalies…

- In an inadequately normalized database, information about that old entry may need to be deleted from more than one place, and, human fallibility being what it is, some of the needed additional deletions may be missed.
- An update of a database involves modifications that may be additions, deletions, or both.
- Thus **'update anomalies'** can be either of the kinds of anomalies discussed above.
- All three kinds of anomalies are highly undesirable, since their occurrence constitutes corruption of the database.
- Properly normalised databases are much less susceptible to corruption than are unnormalised databases.

Types of Relational Join

A JOIN is a method of creating a result set that combines rows from two or more tables (relations).

When comparing the contents of two tables the following conditions may occur:

- Every row in one relation has a match in the other relation.
- Relation R1 contains rows that have no match in relation R2.
- Relation R2 contains rows that have no match in relation R1.

INNER joins contain only matches. OUTER joins may contain mismatches as well.

Inner Join

This is sometimes known as a **simple** join. It returns all rows from both tables where there is a match. If there are rows in R1 which do not have matches in R2, those rows will **not** be listed.

There are two possible ways of specifying this type of join:

SELECT * FROM R1, R2 WHERE R1.r1_field = R2.r2_field;

SELECT * FROM R1 INNER JOIN R2 ON R1.field = R2.r2_field

If the fields to be matched have the same names in both tables then the **ON** condition, as in:

ON R1.fieldname = R2.fieldname

ON (R1.field1 = R2.field1 AND R1.field2 = R2.field2)

can be replaced by the shorter **USING** condition, as in:

- **USING fieldname**
- **USING (field1, field2)**

Natural Join

A natural join is based on all columns in the two tables that have the same name.

It is semantically equivalent to an INNER JOIN or a LEFT JOIN with a **USING** clause that names all columns that exist in both tables.

SELECT * FROM R1 NATURAL JOIN R2

The alternative is a **keyed** join which includes an **ON** or **USING** condition.

Left Outer Join

Returns all the rows from R1 even if there are no matches in R2.

If there are no matches in R2 then the R2 values will be shown as null.

SELECT * FROM R1 LEFT [OUTER] JOIN R2 ON R1.field = R2.field

Right Outer Join

Returns all the rows from R2 even if there are no matches in R1.

If there are no matches in R1 then the R1 values will be shown as null.

- **SELECT * FROM R1 RIGHT [OUTER] JOIN R2 ON R1.field = R2.field**

Full Outer Join

Returns all the rows from both tables even if there are no matches in one of the tables.

If there are no matches in one of the tables then its values will be shown as null.

SELECT * FROM R1 FULL [OUTER] JOIN R2 ON R1.field = R2.field

Self Join

This joins a table to itself.

This table appears twice in the FROM clause and is followed by table aliases that qualify column names in the join condition.

SELECT a.field1, b.field2 FROM R1 a, R1 b WHERE a.field = b.field

Cross Join

This type of join is rarely used as it does not have a join condition, so every row of R1 is joined to every row of R2.

For example, if both tables contain 100 rows the result will be 10,000 rows.

This is sometimes known as a **cartesian product** and can be specified in either one of the following ways:

- **SELECT * FROM R1 CROSS JOIN R2**
- **SELECT * FROM R1, R2**

Logical Design

- Constructing ERD

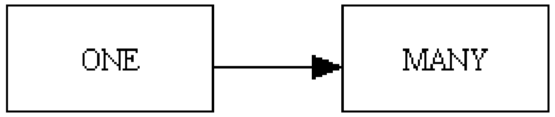
Entity Relationship Diagram (ERD)

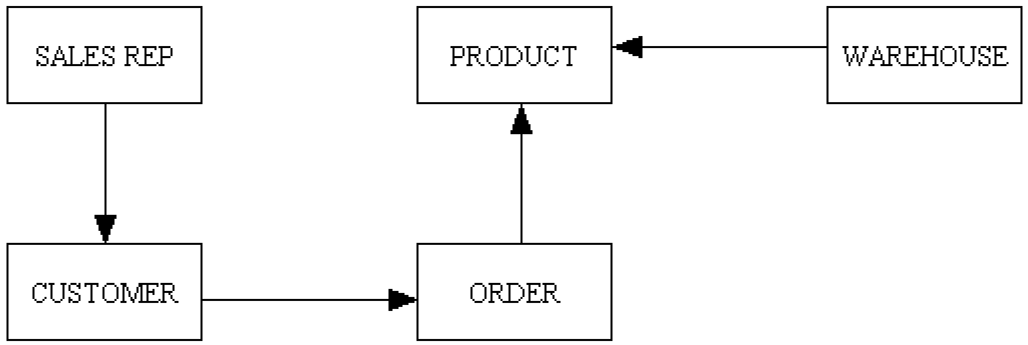
- An entity-relationship diagram (ERD) is a data modeling technique that creates a graphical representation of the entities, and the relationships between entities, within an information system.
- Any ER diagram has an equivalent relational table, and any relational table has an equivalent ER diagram.
- ER diagramming is an invaluable aid to engineers in the design, optimization, and debugging of database programs.
- The entity is a person, object, place or event for which data is collected.
- It is equivalent to a database table.
- An entity can be defined by means of its properties, called attributes.
- For example, the CUSTOMER entity may have attributes for such things as name, address and telephone number.

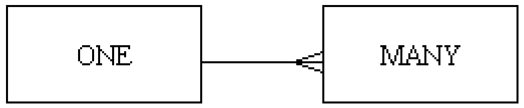
ERD…

- The relationship is the interaction between the entities. It can be described using a verb such as:
  - A customer *places* an order.
  - A sales rep *serves* a customer.
  - A order *contains* a product.
  - A warehouse *stores* a product.
- In an entity-relationship diagram entities are rendered as rectangles, and relationships are portrayed as lines connecting the rectangles.
- One way of indicating which is the 'one' or 'parent' and which is the 'many' or 'child' in the relationship is to use an arrowhead.

ERD Diagram Notation







ERD and Relationship

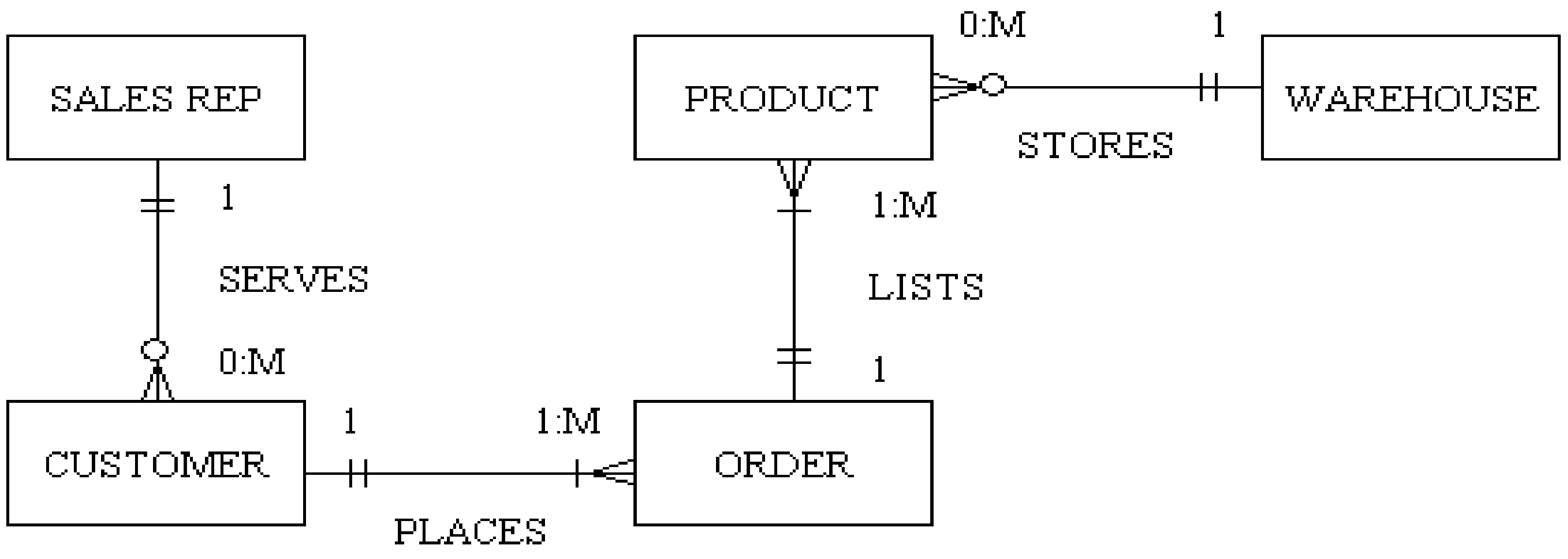
The relating line can be enhanced to indicate cardinality which defines the relationship between the entities in terms of numbers. An entity may be *optional* (zero or more) or it may be mandatory (one or more).

- A single bar indicates **one**.
- A double bar indicates **one and only one**.
- A circle indicates **zero**.
- A crowsfoot or arrowhead indicates **many**.

As well as using lines and circles the cardinality can be expressed using numbers, as in:

- One-to-One expressed as 1:1
- Zero-to-Many expressed as 0:M
- One-to-Many expressed as 1:M
- Many-to-Many expressed as N:M

Representation of ERD



In plain language the relationships can be expressed as follows:

- 1 instance of a SALES REP serves 1 to many CUSTOMERS
- 1 instance of a CUSTOMER places 1 to many ORDERS
- 1 instance of an ORDER lists 1 to many PRODUCTS
- 1 instance of a WAREHOUSE stores 0 to many PRODUCTS

ERD Rules

In order to determine if a particular design is correct simple test has been placed:

- Take the written rules and construct a diagram.
- Take the diagram and try to reconstruct the written rules.

- If the output from step (2) is not the same as the input to step (1) then something is wrong.
- If the model allows a situation to exist which is not allowed in the real world then this could lead to serious problems.
- The model must be an accurate representation of the real world in order to be effective.
- If any ambiguities are allowed to creep in they could have disastrous consequences.

ERD Rules…

Before construct the physical database there are several steps that must take place:

- Assign attributes (properties or values) to all the entities.
- After all, a table without any columns will be of little use to anyone.
- Refine the model using a process known as ‘normalization'.
- This ensures that each attribute is in the right place.
- During this process it may be necessary to create new tables and new relationships.

Physical Design

- General consideration

Database Names

- Database names should be short and meaningful, such as **products**, **purchasing** and **sales**.
  - Short, but not too short, as in **prod** or **purch**.
  - Meaningful but not verbose, as in 'the database used to store product details'.
- Do not waste time using a prefix such as **db** to identify database names.
- DBMS allows a mixture of upper and lowercase names, and it is case sensitive
- It is better to stick to a standard naming convention such as:
  - All uppercase.
  - All lowercase
  - Leading uppercase, remainder lowercase.
- Inconsistencies may lead to confusion, confusion may lead to mistakes, mistakes can lead to disasters.

Database Names...

- Database name contains more than one word, such as in **sales orders** and **purchase orders**, decide how to deal with it:
  - Separate the words with a single space, as in **sales orders**(not allowed)
  - Separate the words with an underscore, as in **sales_orders**
  - Separate the words with a hyphen, as in **sales-orders**.
  - Use camel caps, as in **SalesOrders**.
- Again, be consistent.
- Rather than putting all the tables into a single database it may be better to create separate databases for each logically related set of tables.
- This may help with security, archiving, replication, etc.

Table Names

- Table names should be short and meaningful, such as **part**, **customer** and **invoice**.
  - Short, but not too short.
  - Meaningful, but not verbose.
- Do not use a prefix or suffix such as **tbl** to identify table names.
- Table names should be in the singular (e.g. **customer** not **customers**).
- DBMS allows a mixture of upper and lowercase names, and it is case sensitive.
- It is better to stick to a standard naming convention such as:
  - All uppercase.
  - All lowercase.
  - Leading uppercase, remainder lowercase.
- Inconsistencies may lead to confusion, confusion may lead to mistakes, mistakes can lead to disasters.

Table Names..

- If a table name contains more than one word, such as in **sales order** and **purchase order**, decide how to deal with it:
  - Separate the words with a single space, as in **sales order**
  - Separate the words with an underscore, as in **sales_order**
  - Separate the words with a hyphen, as in **sales-order**
  - Use camel caps, as in **SalesOrder**.
- Again, be consistent.
- Be careful if the same table name is used in more than one database - it may lead to confusion.

Field Names

- Field names should be short and meaningful, such as **part_name** and **customer_name**.
  - Short, but not too short, such as in **ptnam**.
  - Meaningful, but not verbose, such as **the name of the part**.
- Do not use a prefix or suffix such as **col** or **fld** to identify column/field names.
- DBMS allows a mixture of upper and lowercase names, and it is case sensitive, it is better to stick to a standard naming convention such as:
  - All uppercase.
  - All lowercase.
  - Leading uppercase, remainder lowercase.
- Inconsistencies may lead to confusion, confusion may lead to mistakes, mistakes can lead to disasters.

Field Names..

- If a field name contains more than one word, such as in **part name** and **customer name**, decide how to deal with it:
  - Separate the words with a single space, as in **part name**.
  - Separate the words with an underscore, as in **part_name** .
  - Separate the words with a hyphen, as in **part-name**.
  - Use camel caps, as in **PartName**.
- Again, be consistent.
- Common words in field names may be abbreviated, but be consistent.
  - Do not allow a mixture of abbreviations, such as 'no', 'num' and 'nbr' for 'number'.
  - Publish a list of standard abbreviations and enforce it.
- Although field names must be unique within a table
- It is possible to use the same name on multiple tables even if they are unrelated, or they do not share the same set of possible values.
- It is recommended that this practice should be avoided, for reasons described in Field names should identify their content and The naming of Foreign Keys.

Primary Key

- It is recommended that the primary key of an entity should be constructed from the table name with a suffix of **_ID**.
- This makes it easy to identify the primary key in a long list of field names.
- Do not use prefix or suffix such as **pk** to identify primary key fields. This has absolutely no meaning to any database engine or any application.
- Avoid using generic names for all primary keys.
- It may seem a clever idea to use the name **ID** for every primary key field, but this causes problems:
  - It causes the same name to appear on multiple tables with totally different contexts.
  - The string ID='ABC123' is extremely vague as it gives no idea of the entity being referenced.
  - Is it an invoice id, customer id, or what?
- It also causes a problem with foreign keys.

Primary Key…

- There is no rule that says a primary key must consist of a single attribute - both simple and composite keys are allowed.
- Avoid the unnecessary use of technical keys.
- If a table already contains a satisfactory unique identifier, whether composite or simple, there is no need to create another one.
- The use of a technical key can be justified in certain circumstances, it takes intelligence to know when those circumstances are right.
- The indiscriminate use of technical keys shows a distinct lack of intelligence.

Foreign Keys

- It is recommended that where a foreign key is required that you use the same name as that of the associated primary key on the foreign table.
- It is a requirement of a relational join that two relations can only be joined when they share at least one common attribute, and this should be taken to mean the attribute name(s) as well as the value(s). Thus where the**customer** and **invoice** tables are joined in a parent-child relationship the following will result:
  - The primary key of **customer** will be **customer_id**.
  - The primary key of **invoice** will be **invoice_id**.
  - The foreign key which joins **invoice** to **customer** will be **customer_id**.
- For MySQL users this means that the shortened version of the join condition may be used:
  - Short: A LEFT JOIN B USING (a,b,c)
  - Long: A LEFT JOIN B ON (A.a=B.a AND A.b=B.b AND A.c=B.c)

Foreign Kesy…

- The only exception to this naming recommendation should be where a table contains more than one foreign key to the same parent table.
- Simply add a meaningful suffix to each name to identify the usage, such as:
  - To signify movement use **location_id_from** and **location_id_to**.
  - To signify positions in a hierarchy use **node_id_snr** and **node_id_jnr**.
  - To signify replacement use **part_id_old** and **part_id_new**.
- Prefer to use a suffix rather than a prefix as it makes the leading characters match (as in **PART_ID_old** and **PART_ID_new**) instead of having the trailing characters match (as in **old_PART_ID** and **new_PART_ID**)

Generating Unique ids

- Where a technical primary key is used a mechanism is required that will generate new and unique values.
- Such keys are usually numeric, so there are several methods available:
- Some database engines will maintain a set of sequence numbers for you which can be referenced using code such as :

```
SELECT <seq_name>.NEXTVAL FROM DUAL
```

- Using such a sequence is a two-step procedure:
  - Access the sequence to obtain a value.
  - Use the supplied value on an INSERT statement.

Generating Unique ids..

- It is sometimes possible to access the sequence directly from an INSERT statement, as in the following:

  INSERT INTO tablename (col1,col2,...) VALUES (tablename_seq.nextval,'value2',...)

- If the number just used needs to be retrieved so that it can be passed back to the application it can be done so with the following:

  SELECT <seq_name>.CURRVAL FROM DUAL

- But a disadvantage that found is that the DBMS has no knowledge of what primary key is linked to which sequence
- So it is possible to insert a record with a key not obtained from the sequence and thus cause the two to become unsynchronised.
- The next time the sequence is used it could therefore generate a value which already exists as a key and therefore cause an INSERT error.

Generating Unique ids…

- Some database engines will allow to specify a numeric field as 'auto-increment‘
- An INSERT they will automatically generate the next available number.
- This is better than the previous method because:
  - The sequence is tied directly to a particular database table and is not a separate object, thus it is impossible to become unsynchronised.
  - It is not necessary to access the sequence then use the returned value on an INSERT statement - just leave the field empty and the DBMS will fill in the value automatically.

Generating Unique ids..

- While the previous methods have their merits, they both have a common failing in that they are not-standard extensions to the SQL standard, therefore they are not available in all SQL-compliant database engines.
- This becomes an important factor if it is ever decided to switch to another database engine.
- A truly portable method which uses a standard technique and can therefore be used in any SQL-compliant database is to use an SQL statement similar to the following to obtain a unique key for a table:

  SELECT max(table_id) FROM <tablename> table_id = table_id+1

Generating Unique ids...

- Some people seem to think that this method is inefficient as it requires a full table search, but they are missing the fact that **table_id** is a primary key, therefore the values are held within an index.
- The **SELECT max(...)** statement will automatically be optimized to go straight to the last value in the index, therefore the result is obtained with almost no overhead.
- This would not be the case if I used **SELECT count(...)** as this would have to physically count the number of entries.
- Another reason for not using **SELECT count(...)** is that if records were to be deleted then record count would be out of step with the highest current value.

Generating Unique ids…

- The Radicore development framework has separate data access objects for each DBMS to which it can connect.
- This means that the different code for dealing with auto_increment keys can be contained within each object, so is totally transparent to the application.
- All that is necessary is that the key be identified as 'auto_increment' in the Data Dictionary and the database object will take care of all the necessary processing.

Object-Oriented Database Design using UML and ODMG

Object Oriented Databases (I) – Lecture 9

Advanced Databases

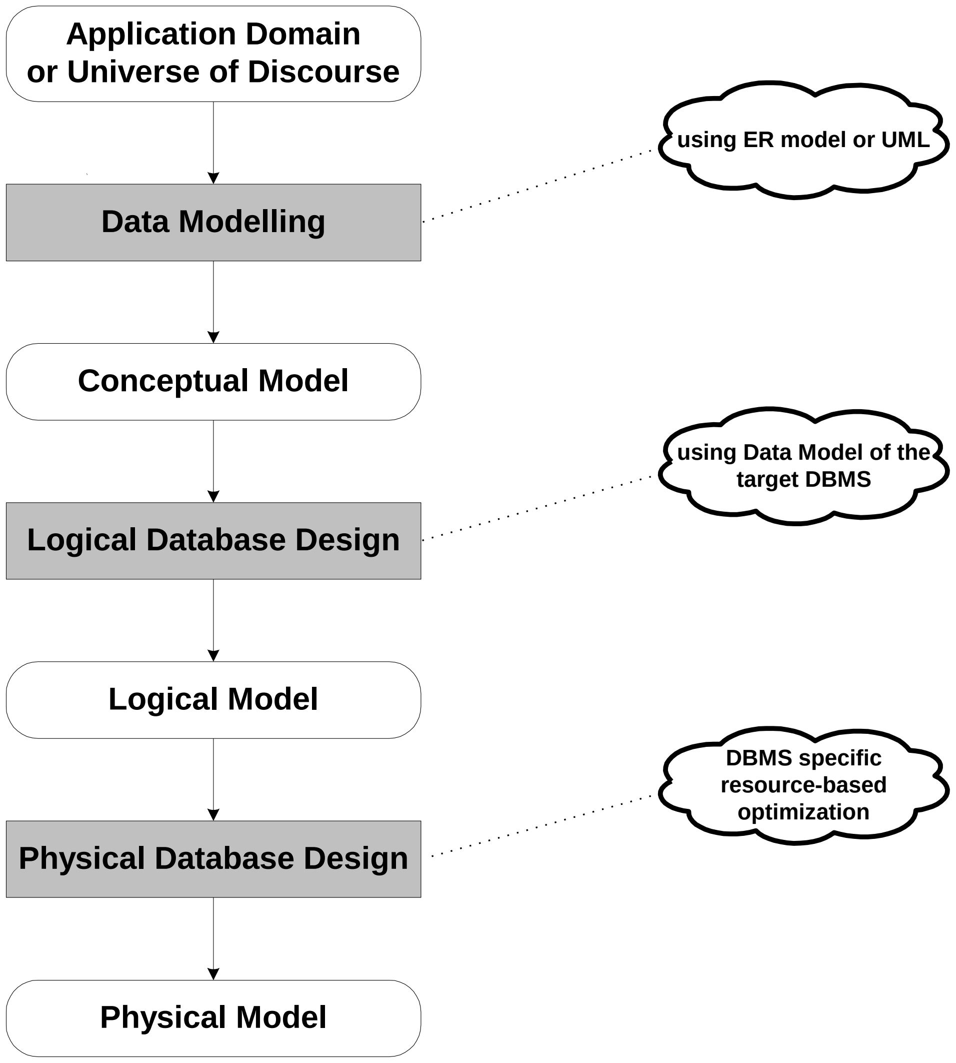
Lecture outline

- Database Design Process
- Object-Oriented concepts
  - Objects, Classes
  - Attributes, Operations, Associations
  - Encapsulation, Inheritance
- Object-Oriented Data Modelling
  - Identifying
    - Classes, attributes, and operations.
    - Associations among classes
  - Drawing class diagrams – conceptual model
- Introduction to ODMG 3.0 (the standard for Object-Oriented Databases)

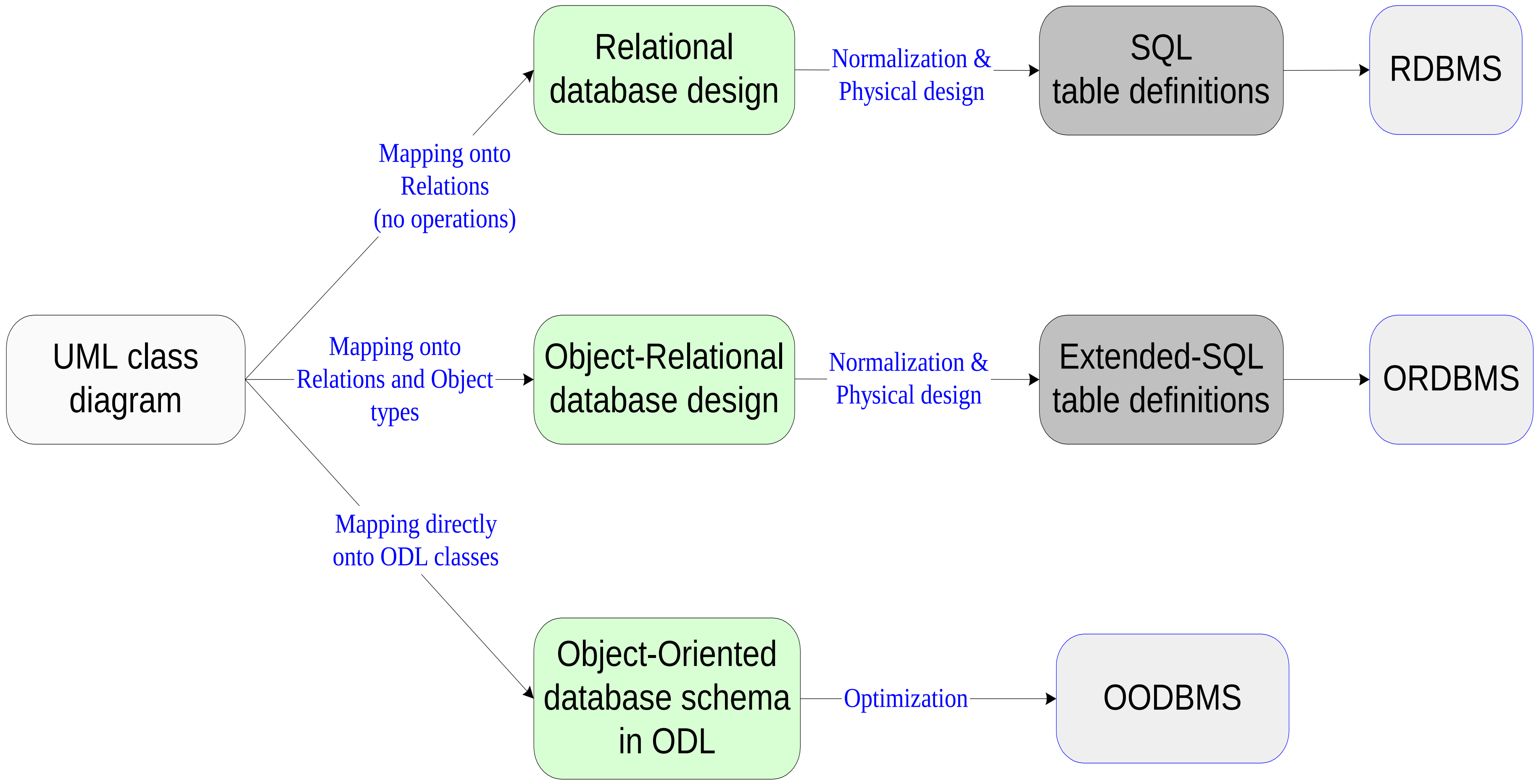
References

- ***Database Systems*** – 4th Edition (chapters 25 to 27) by Connolly & Begg, Addison Wesley, 2005
- ***Database Design for Smarties using UML for Data Modelling*** (chapter 13) by Robert J. Muller, Morgan Kaufmann Publishers, 1999 (specialist text in library)
- ***Fundamental of Database Systems*** – 5th Edition (chapters 20 & 21) by R. Elmasri and S. B. Navati, Addison Wesley, 2007
- ***Object Database Standard: ODMG 3.0*** by R.G.G. Cattell, Douglas K. Barry, Morgan Kaufmann Publishers, 2000 (reference in library)
- ***Object-Oriented Database Design clearly explained*** by Jan L. Harrington. Morgan Kaufmann Publishers, 2000 (reference in library)

Database Design Process



Logical/Physical database design


UML class diagram
Mapping onto Relations (no operations)
Mapping onto Relations and Object types
Mapping directly onto ODL classes
Relational database design
Object-Relational database design
Object-Oriented database schema in ODL
Normalization & Physical design
Normalization & Physical design
Optimization
SQL table definitions
Extended-SQL table definitions
RDBMS
ORDBMS
OODBMS

Object-oriented concepts

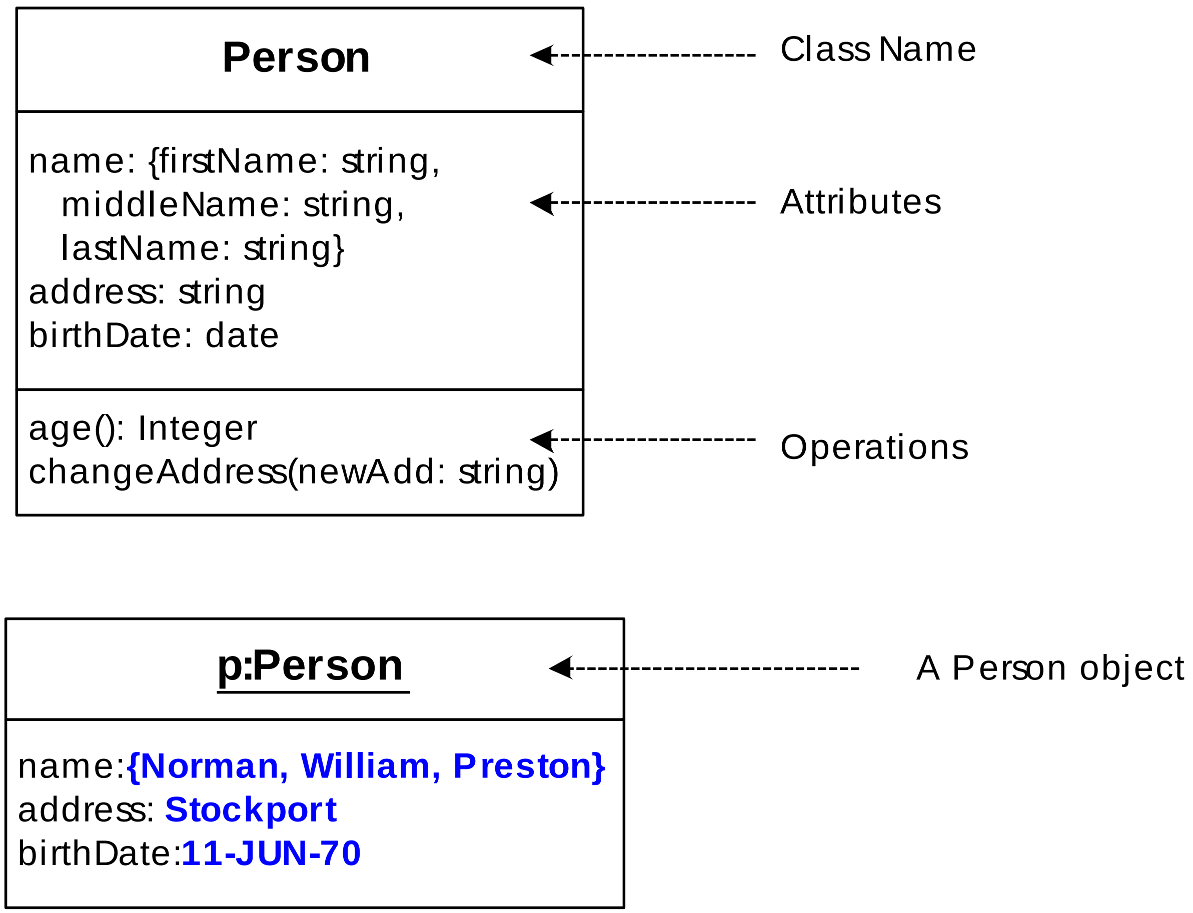
- Objects
  - Objects represent real world entities, concepts, and tangible as well as intangible things.
    - For example a person, a drama, a licence
  - Every object has a unique identifier (OID).
    - System generated
    - Never changes in the lifetime of the object
  - An object is made of two things:
    - **State:** attributes (name, address, birthDate of a person)
    - **Behaviour:** operations (age of a person is computed from birthDate and current date)
  - Objects are categorized by their type or class.
  - An object is an instance of a type or class.

Object-oriented concepts …

- Classification
  - Classification is the process of grouping together objects which have common features.
  - Programming languages have type systems and database systems have data models to classify object.
  - The name used for the classificatory group of values is usually called ***class***.
- Class
  - Provides a template for constructing objects.
  - Instances of a class have the same kind of data and identical behaviour.

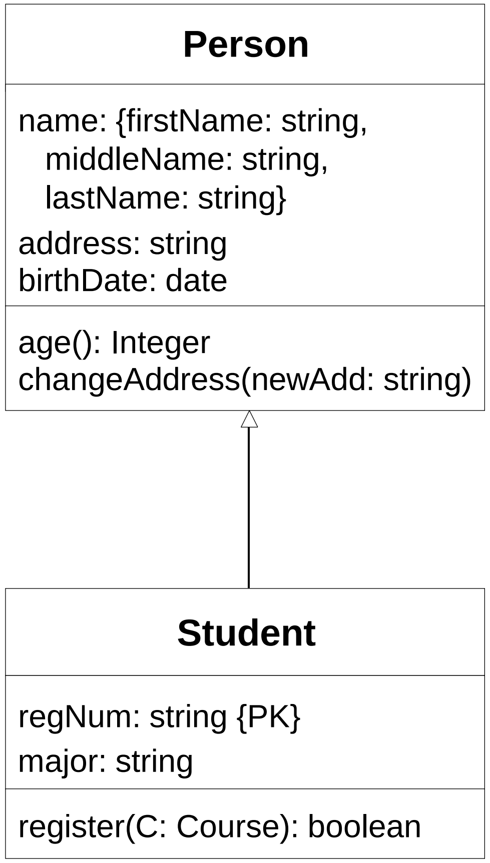
Object-oriented concepts ...

- An Example of a class in UML



Object-oriented concepts …

- Encapsulation
  - Merger of data structure and operations.
    - Objects are composed of attributes (values) and operations (behaviour).
- Inheritance
  - A class can be defined in terms of another one.
  - Person is super-class and Student is sub-class.
  - Student class inherits attributes and operations of Person.



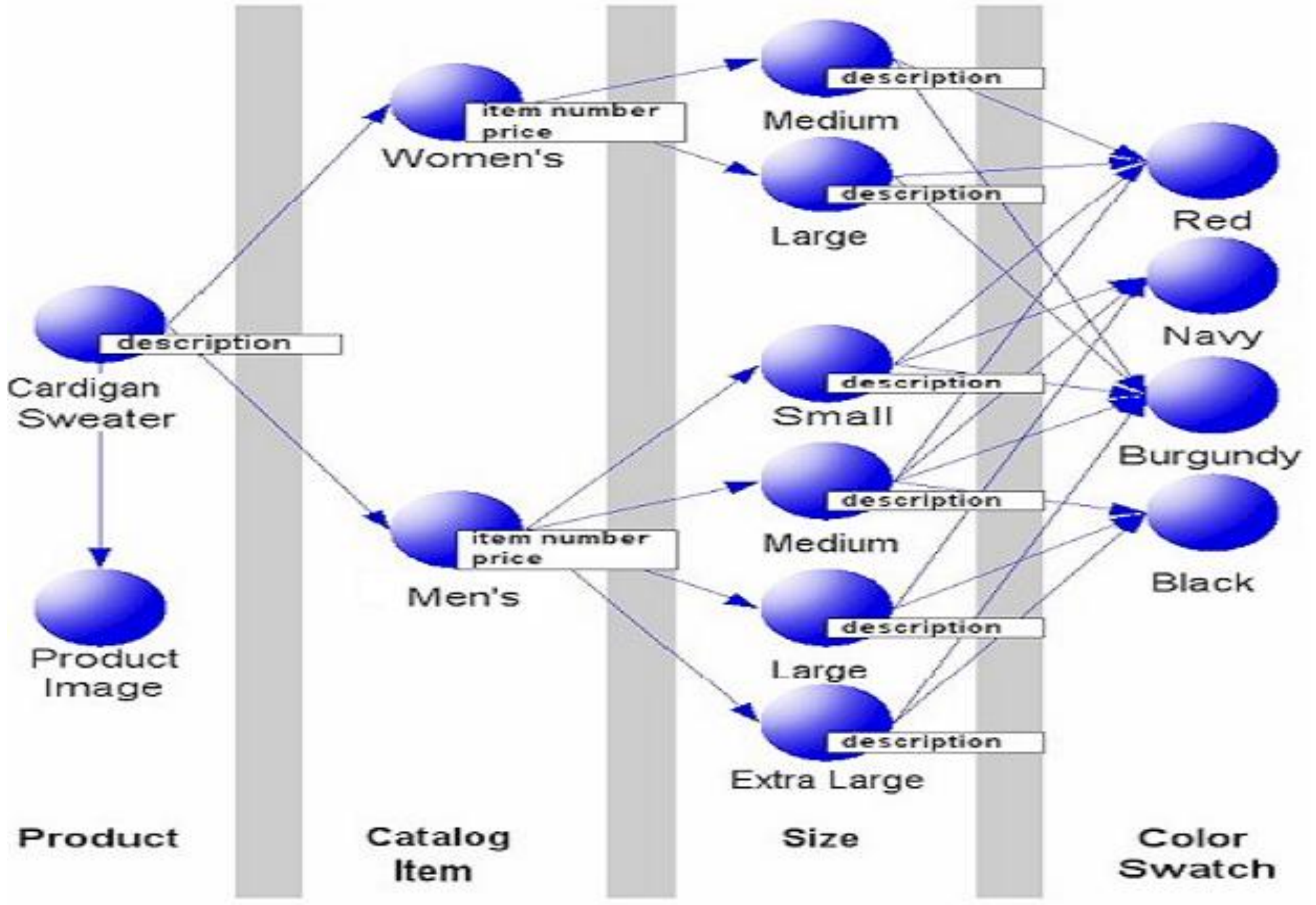
Object-oriented concepts …

- An *object system* or *object-based system* is one that supports the modeling of data as abstract entities, with object identity.
- An *object-oriented system* is an object system in which all data is created as instances of classes which take part in an inheritance hierarchy.
- An *object-oriented database management system* (ODBMS) is a DBMS with an object-oriented logical data model.
- An *object-oriented database* (ODB) is a database made up of objects and managed by an ODBMS.

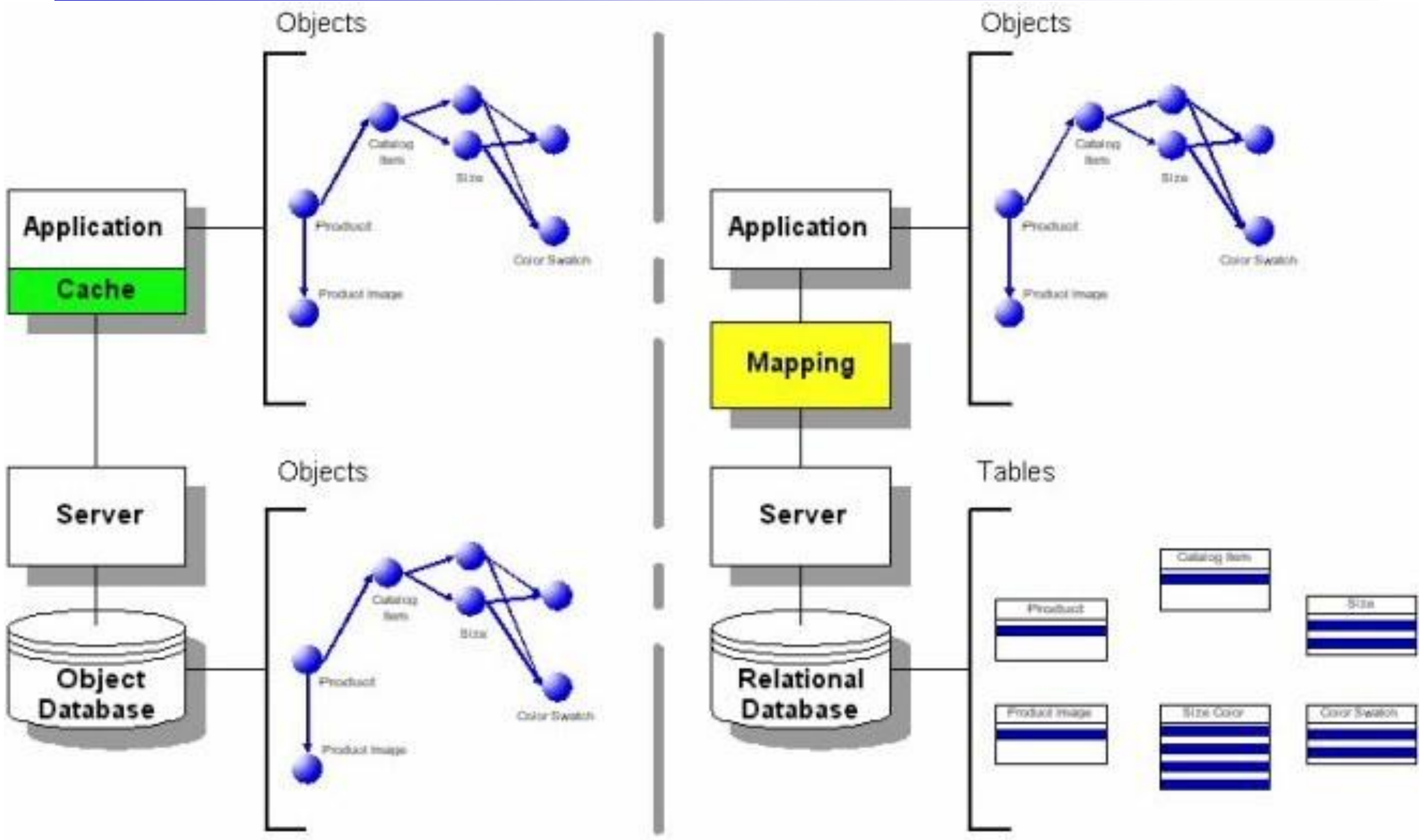
Why ODBs?

- ODBs are inevitable when:
  - Data is complex and variable in size
  - Complex structural and compositional relationships
  - Data is highly inter-related
  - Data is evolving rapidly over time
  - Richer data types
    - complex objects
    - inheritance
    - user extensibility
  - Behaviour with data
    - not just a *data* model but also
    - operations can be bundled together with data

Complex Data



ODBs are more Natural & Direct



Comparison

- RDBs vs. ORDBs
  - Very easy to compare because both are based on Relational Model.
  - An RDB does not support abstract data types (ADT), all attribute values must be atomic and relations must be in first normal form (flat relation).
  - An ORDB supports ADTs, attributes can be multi-valued, and does not require first normal form.
  - The underlying basic data structures of RDBs are much simpler but less versatile than ORDBs.
  - ORDBs support complex data whereas RDBs don’t.
  - ORDBs support wide range of applications.

Comparison – continued…

- RDBs vs. ODBs.
  - Not very easy to compare because of philosophical differences.
  - RDBs have only one construct i.e. Relation, whereas the type system of ODBs is much richer and complex.
  - RDBs require primary keys and foreign keys for implementing relationships, ODBs simply don't.
  - ODBs support complex data whereas RDBs don't.
  - ODBs support wide range of applications.
  - ODBs are much faster than RDBs but are less mature to handle large volumes of data.
  - There is more acceptance and domination of RDBs in the market than that for ODBs.

Comparison – continued…

- ODBs vs. ORDBs.
  - Both support ADTs, collections, OIDs, and inheritance, though philosophically quite different.
  - ORDBs extended RDBs whereas ODBs add persistence and database capabilities to OO languages.
  - Both support query languages for manipulating collections and nested and complex data.
  - SQL3 is inspired from OO concepts and is converging towards OQL (object query language).
  - ORDBs carries all the benefits of RDBs, whereas ODBs are less benefited from the technology of RDBs.
  - ODBs are seamlessly integrated with OOPLs with less mismatch in the type systems;
  - ORDBs (SQL3) have quite different constructs than those of OOPLs when used in embedded form.

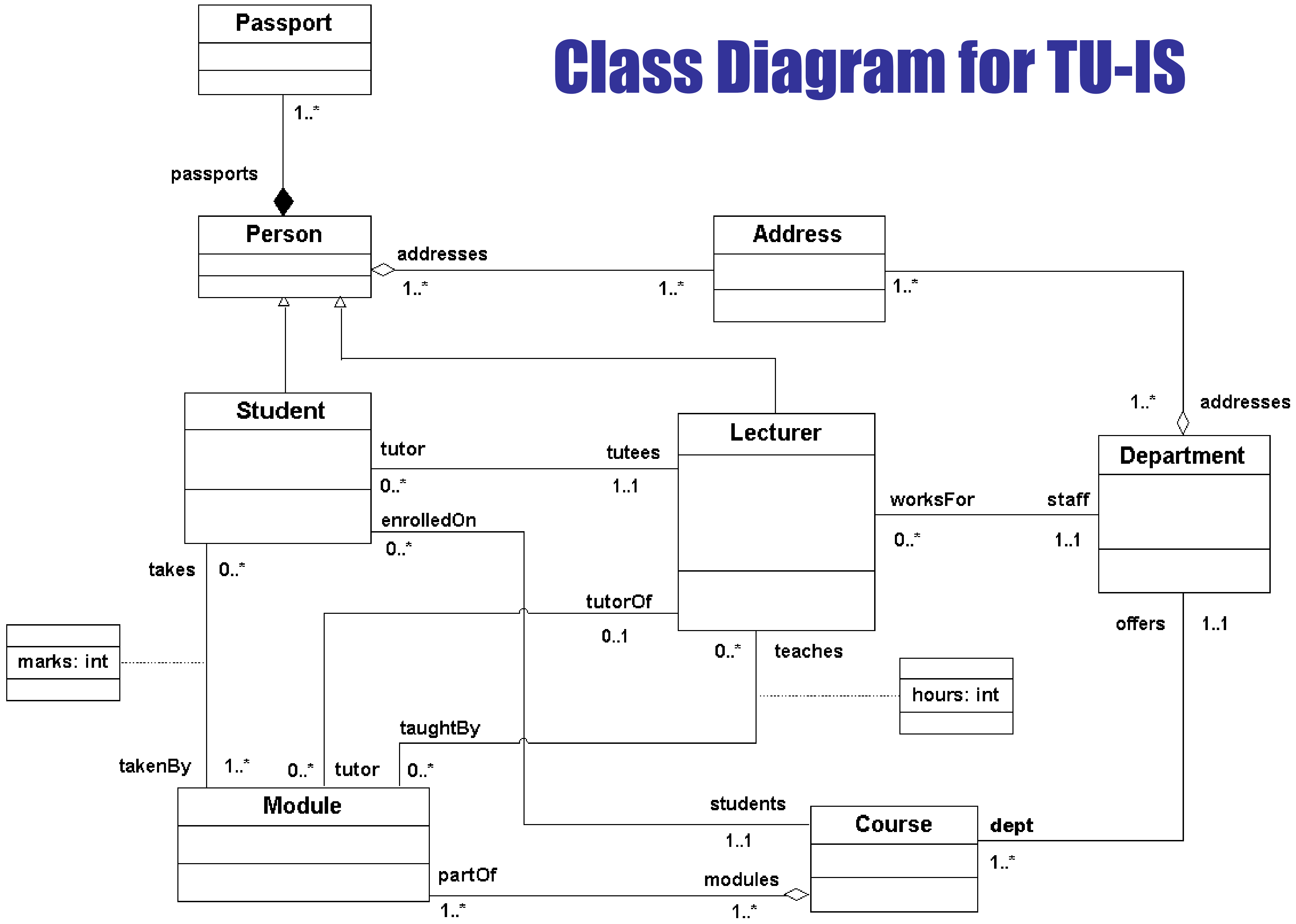
Object-Oriented Data Modelling

- Identification of objects in the system
  - Use UML analysis techniques e.g. use-cases, domain object models.
  - Potential sources are:
    - Things, People, Roles, Organizations, Concepts
    - Events, Processes, Places, Locations, etc
  - Devise an Object Model
- Refining the object model
  - Grouping objects in Classes
  - Identifying Attributes, Operations, Associations & Multiplicities
  - Drawing class diagrams (of **persistent** classes)
- Reconciling classes
  - Revisiting the classes for inheritance
  - Considering normalization of classes into simple classes
  - Producing a big picture: a class diagram (perhaps without showing attributes and operations).

OO Data Modelling: Example

- Universe of Discourse: **TU Information System (TU-IS)**
  - Tribhuwan University has several academic departments.
  - Each department provides one or more courses.
  - Each course is composed of several modules, where a module may be part of more than one course.
  - A student enrolls on a course and every year takes a specified number of modules. Note that several students are usually registered for a course.
  - Every student is assigned a tutor at the start of the course, who is a lecturer in the department providing the course.
  - A lecturer works for a department and usually teaches on several modules.
  - Each module has a module tutor who is a lecturer. A lecturer may be a tutor of several modules.

Class Diagram for TU-IS



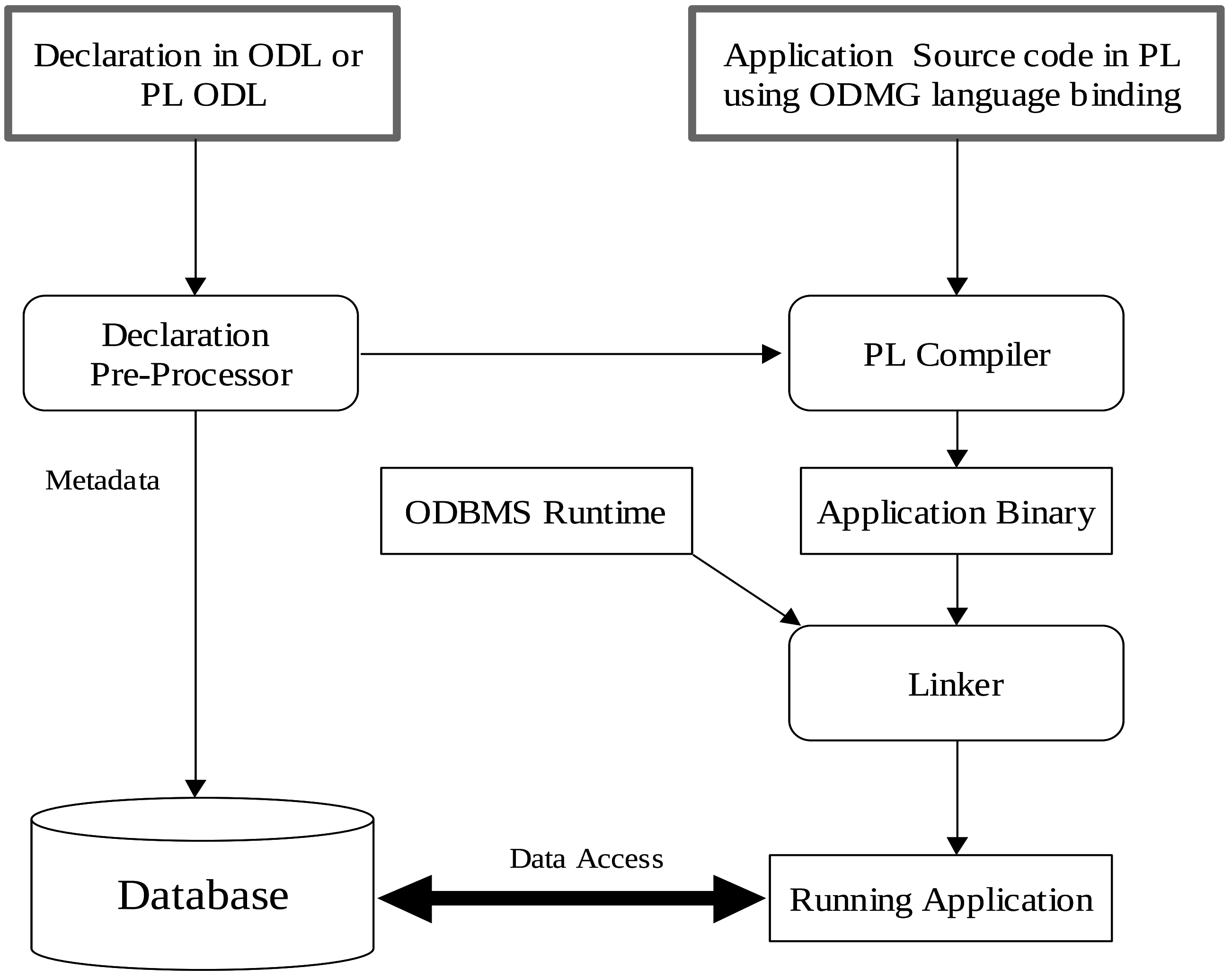
ODMG 3 Object Database Standard

- Object Database Management Group, formed 1991, intended to:
  - provide a standard where previously there was none
  - support portability between products
  - standardize model, querying and programming issues
- Enables both designs and implementations to be ported between compliant systems
- Currently on version 3.0
- Most vendor products are moving toward compliance; O2 is possibly the closest
- Vendors
  - Object Design,Objectivity, O2 Technology, POET, etc.
- URL: www.odmg.org
- We will be using lambda-DB, a freely available ODBMS.

ODMG Components

- An architecture for OODBMS.
- An object model.
  - that will act as the logical model for all OODBMS and provide a level of interoperability.
- A data definition language (ODL).
  - a concrete specification of the operations permitted over a schema defined in the data model.
- A query language (OQL).
  - for posing ad-hoc queries but not for data definition or data manipulation.
- Language bindings to existing OOPL (C++, Java, Smalltalk).
  - the object manipulation languages are the integration of a PL with the ODMG model, so the OOPL’s get persistence and the OODB gets a flexible and standard DB programming language.

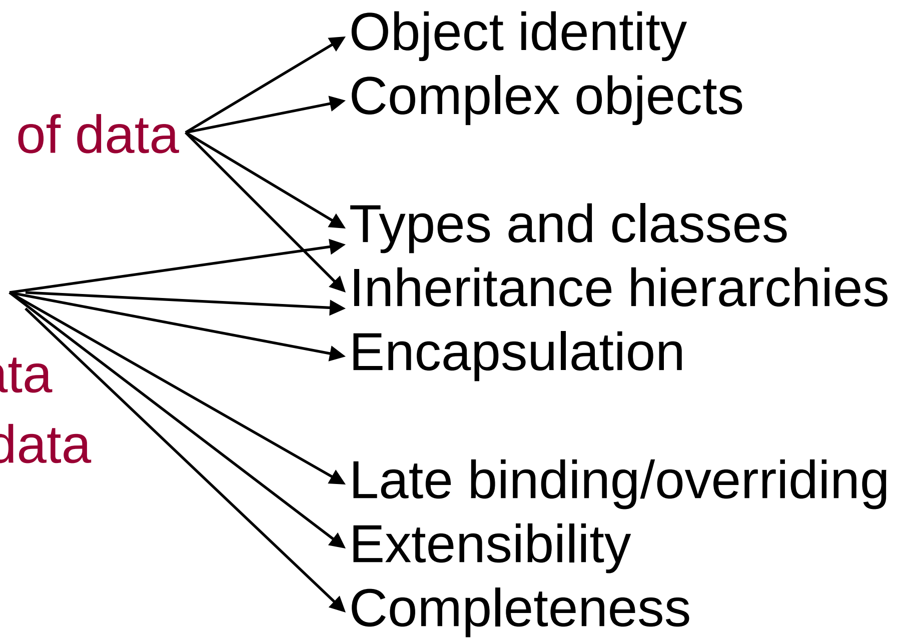
An Architecture for OODBMS


Declaration in ODL or PL ODL
Application Source code in PL using ODMG language binding
Declaration Pre-Processor
PL Compiler
Metadata
ODBMS Runtime
Application Binary
Linker
Database
Data Access
Running Application

Object Model

1. Data Model
   - state and structure of data

2. Behaviour model
   - dynamics of the data
   - operations on the data



3. Persistence model
   - the way the persistent and transient data is created & changes status

4. Naming model
   - naming and accessing objects

Object Definition Language (ODL)

- ODL is a specification language used to define the schema of an ODMG compliant database.
- ODL supports all semantic constructs of the ODMG object model.
- ODL is independent of any programming language, and hence provides means for the portability of database schema across complaint ODBMSs.
- The database schema may comprise of:
  - an ODL module (i.e. a higher level construct for grouping ODL specifications),
  - some generic object types using interface,
  - some concrete object types using class, and
  - some literal types using struct, etc.

Components of ODL (literal)

- **Literal Types**
  - Define values (not having OIDs)
  - Cannot stand alone i.e., must be embedded in objects
  - Can be simple, collection and structured
- **Simple**
  - `long`, `short`, `unsigned long`, `unsigned short`, `float`, `double`, `char`, `string`, `boolean`, `enum`
- **Collection**
  - `set`: unordered that do not allow duplicates,
  - `bag`: unordered that allow duplicates,
  - `list`: ordered that allow duplicates,
  - `array`: one-dimensional with variable length, and
  - `dictionary`: unordered sequence of key-and-value pairs without duplicate keys

Components of ODL (literal) ...

- Structured
  - `date`, `time`, `timestamp`, `interval`, and `struct`
- For example

```
struct Name {
  string firstName,
  string middleName,
  string lastName
};
```

Components of ODL (object)

- **Object Types**
  - `interface`: defines only the abstract behaviour of an object type.
    - Instances of an interface type cannot be created
    - For example

```
interface Object {
  ...
  boolean same_as(in Object other_object);
  Object copy();
  void delete();
};
```

  - `class`: defines both abstract state and behaviour of an object type
    - Instances of a class can be created
    - For example

```
class Person {
  ...
  attribute Name name;
  attribute date birthDate;
  unsigned short age();
};
```

Components of ODL (object) ...

- **State definition: Attributes**
  - An **`attribute`** is defined for each attribute in a UML class or an ER entity type.
  - An attribute belongs to a single class and is not a self-standing object.
  - The type of the values (domain) of an attribute is either object or literal (atomic, structured or collection).
  - For example:

    ```
    attribute set<string> qualifications;
    ```

    - Defines an attribute of `Lecturer` class called `qualifications` the value of which is of type **`set<string>`**.
  - Consider that `lec` represents a `Lecturer` object then

    ```
    lec.qualifications := set("BSc", "MSc", "PhD");
    ```

    - Will assign the set of strings as a value to the `qualifications` attribute of the lecturer object.

Components of ODL (object) ...

- **Behaviour definition: Operations**
  - Objects may have certain behaviour that is specified as a set of operations.
  - An object type includes an **operation signature** for each operation that specifies:
    - name of the operation,
    - names and types of each argument, and
    - the type of the returned value, if any.
  - For example:

    ```
    unsigned short age();
    ```

    - Defines the operation `age` without any arguments which return a value of type **`unsigned short`**.

Components of ODL (object) …

- **Extent and Keys**
  - Besides, attributes and operations, a class definition may specify an extent and a unique key.
- **Extent**
  - Defines the set of all instances of a given class within an ODB.
  - Deleting an object removes the object from the extent of a corresponding class.
- **Key**
  - Uniquely identifies the instances of a class.
  - The key concept is similar to the concept of primary key in RDBs, however, keys are not must in ODBs and are not used to implement relationships (as in the case of RDBs).
  - A class must have an extent to have a key.
- **For example:**

```
class Student (extent Students key regNum) {...};
```

  - Defines `Students` to be the extent and `regNum` to be a unique key of the `Student` class.

Components of ODL (object) ...

- **Atomic object type**
  - Any user-defined object type e.g., `Person`
- **Collection object types**
  - **`Set`**: unordered that do not allow duplicates,
  - **`Bag`**: unordered that allow duplicates,
  - **`List`**: ordered that allow duplicates,
  - **`Array`**: one-dimensional with variable length, and
  - **`Dictionary`**: unordered sequence of key-and-value pairs without duplicate keys
- **Structured object types**
  - **`Date`**, **`Time`**, **`Timestamp`**, **`Interval`**
- **Watch out that ODL is case-sensitive e.g.,**
  - **`Set`** is a collection object type whereas **`set`** is a literal collection.
  - `Name` is a type name whereas `name` is an attribute in the `Person` class definition.

Mapping Class Diagrams into ODL

- At this stage, we are dealing with classes, attributes, and operations.
  - Different associations and inheritance will be covered next.
- Mapping (general case)
  - Each UML class becomes an ODL class.
  - Each attribute or method in a UML class becomes an attribute or operation of an ODL class with appropriate types.
  - Specify a suitable extent name unless the class diagram explicitly indicates otherwise.
  - Specify a unique key if one or more attributes of a UML class are shown in bold or tagged with {PK}.
  - For a composite attribute, specify a structure literal type.

Mapping TU-IS class diagram into ODL

```
module TU_IS1 {
  struct Name {
    string firstName;
    string middleName;
    string lastName; };
  class Person {
    attribute Name name;
    attribute date birthDate;
    attribute char gender;
    unsigned  short age();   };

  class Lecturer (extent  Lecturers key lecturerId) {
    attribute string       lecturerId;
    attribute unsigned     short room;
    attribute float        salary;
    attribute date         joinedOn;
    attribute set<string> qualifications;
    boolean   teachModule(in Module M); };
```

TU-IS schema in ODL …

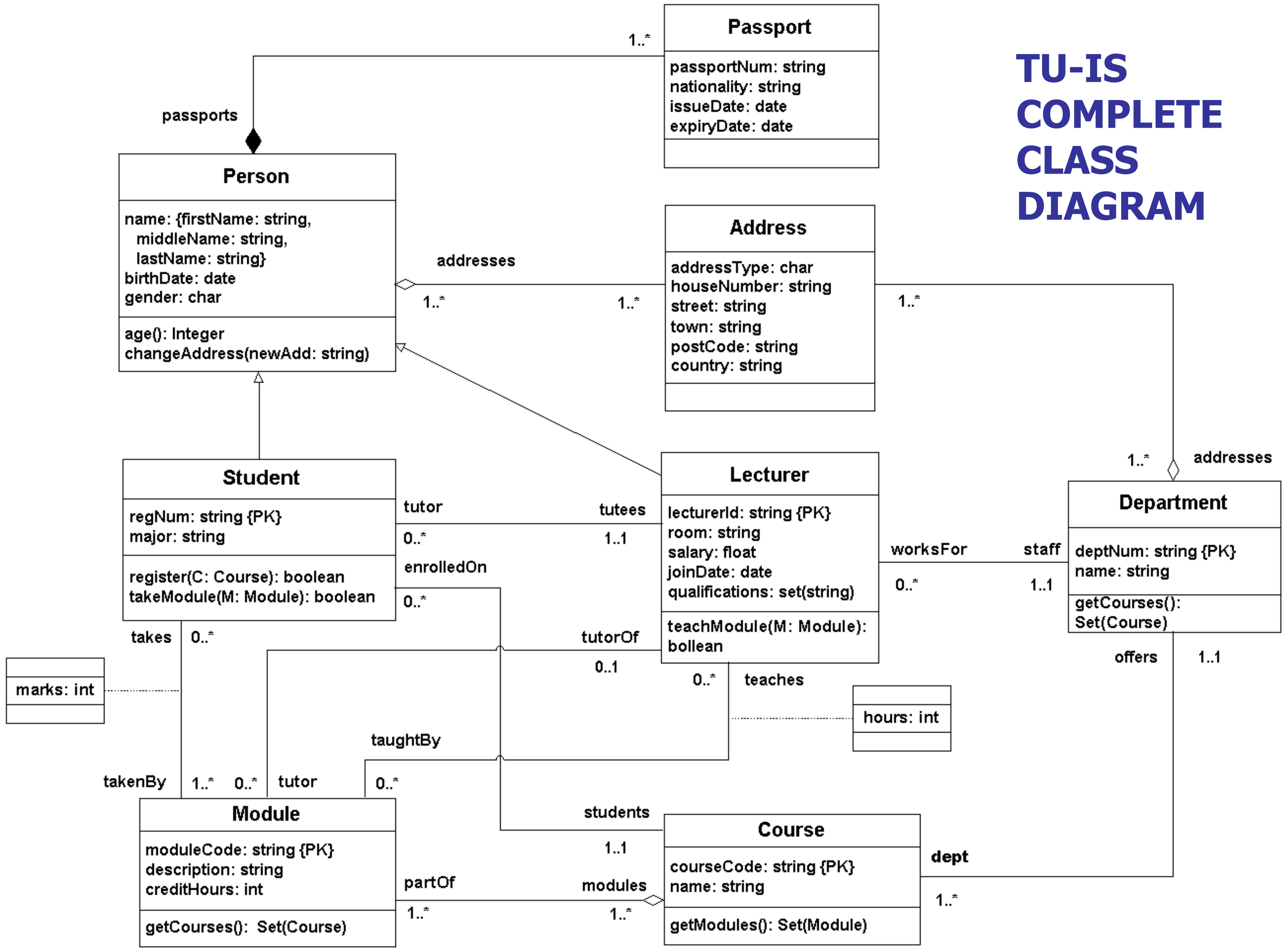
```
  class Department (extent Departments key deptNum) {
    attribute string deptNum;
    attribute string name; };

  class Course (extent Courses key courseCode) {
    attribute string courseCode;
    attribute string name; };

  class Module (extent Modules key moduleCode) {
    attribute string moduleCode;
    attribute string name;
    attribute unsigned short creditHours; };

  class Student (extent Students key regNum) {
    attribute string regNum;
    attribute string major;
    boolean   register(in Course C);
    boolean   takeModule(in Module M); };
};
```

TU-IS COMPLETE CLASS DIAGRAM



The Extended Entity Relationship Model and Object Model

By
Bishnu Gautam
New Summit College

ER Model (Revisited)

- Why ER model?
  - A very popular high-level conceptual data model
  - Facilitates database design by specifying schema that represent the overall logical structure of the DB
  - Entities and attributes: an attribute is a function which maps an entity set into a domain

    eg. Faculty (Name, Dept, SSN)

    domain for attribute Dept = {CS, EE, APMA, SYS}
  - A particular entity is described by a set of values:

    {(Name: John Doe), (Dept: CS), (SSN: 123-45-6789)}
  - Entity type plays a particular role in a relationship: usually implicit but must be specified if not distinct

    eg.Parents (Person, Person), War (Country, Country)

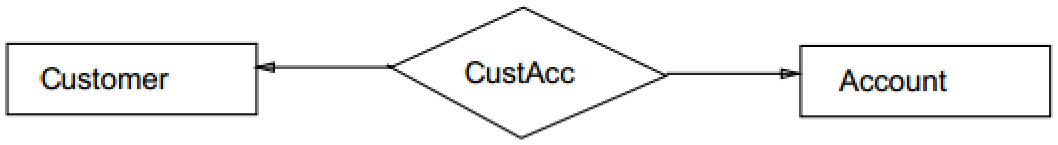
Mapping Cardinality

- Relationships
  - 1:1, 1:N, N:M are distinguished by a directed line
  - A directed line represents "at most one", not requiring there must be one corresponding entity for every entity
  - A description of all possible associations in the real-world that is being modeled
  - 1:1 relationship is rather rare in databases, while N:M relationships are quite common (hard to represent)
  - Naming relationships are sometimes tricky

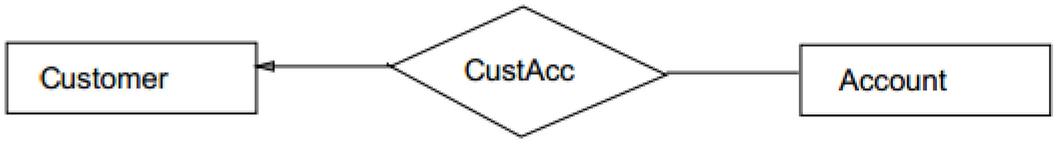
    eg. A relationship between Faculty and Students:

Should it be advisee or advisor?

Mapping Cardinality

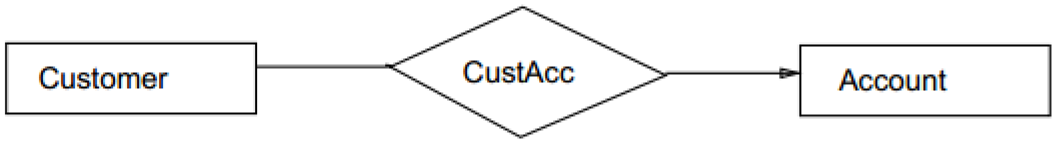


One-to-one relationship (one customer - one account)



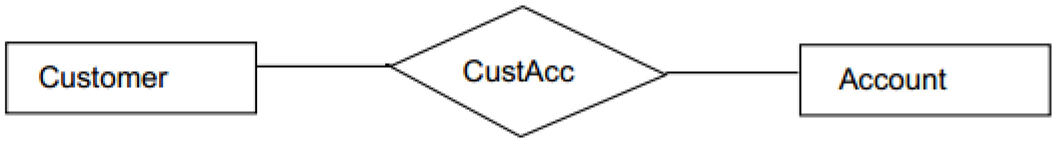
One-to-many from customer to account

A customer can have several accounts, but no account can be shared



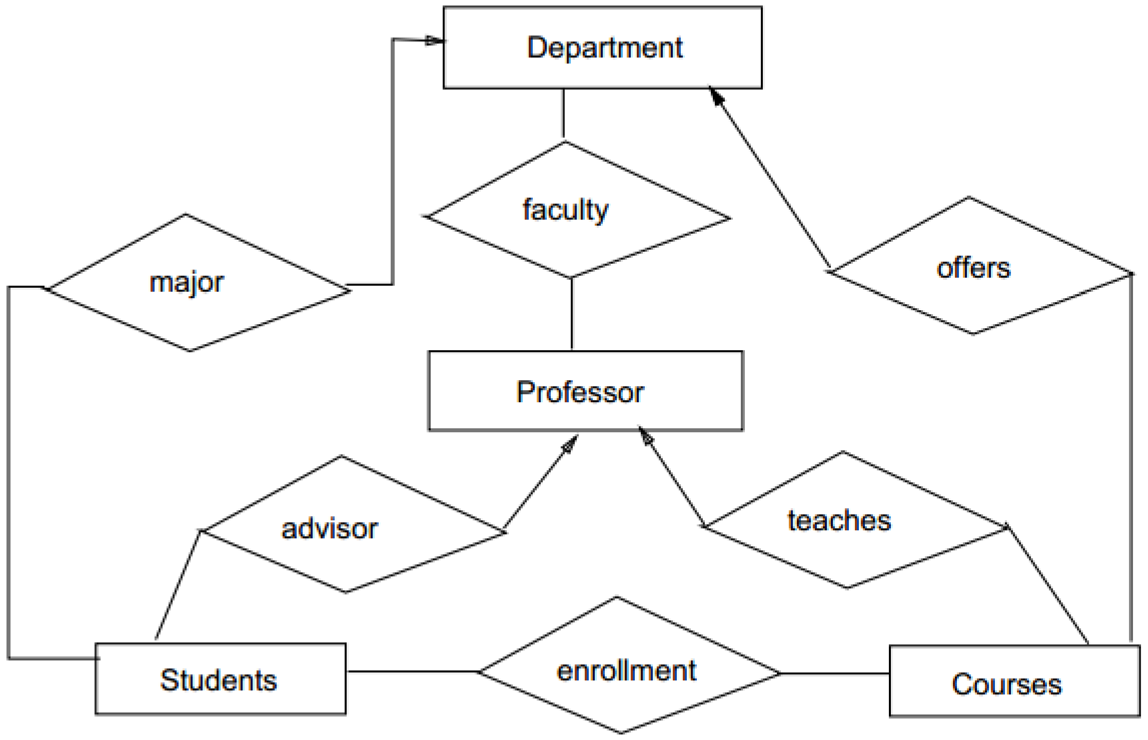
Many-to-one from customer to accountss

A customer can have only one account, but accounts can be shared



Many-to-many relationship

ER Diagram Design



An ER diagram represents several assertions about the real-world.
When attributes are added, more assertions are made.
How can we ensure that it is "faithful"?

- A database is judged correct if it captures ER diagram correctly.

- There is no way of verifying that ER diagram is logically correct.

Key Attributes

- Key and key attributes

  key: a unique value for an entity

  key attributes: a group of one or more attributes that uniquely identify an entity in the entity set
- Super key, candidate key, and primary key

  super key: a set of one or more attributes which allows to identify uniquely an entity in the entity set

  candidate key: minimal super key there can be many candidate keys

  eg. Employee (Name, Address, SSN, Salary, Project)

  (Name, Address) and (SSN) are candidate keys, but

  (Name, SSN) is not a candidate key

  primary key: a candidate key chosen by the DB designer denoted by underlining in ER diagram

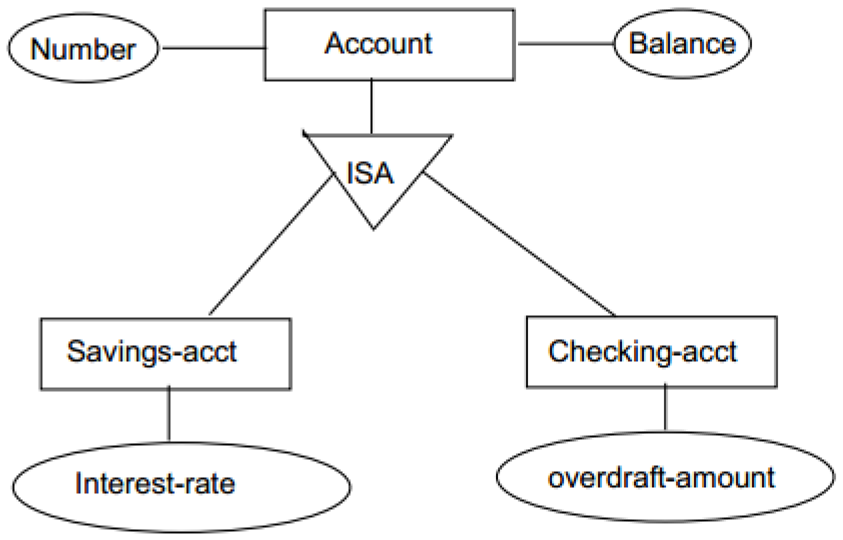
Weak Entity Types

- Weak entity type
  - Its existence depends on other entity (owner)
  - No key attributes of its own
  - Cannot be identified without an owner entity
  - Indicated in ER diagram by a double outlined boxes

    eg. Transaction (Txn#, Type, Date, Amount) is a weak entity with Account (Ac#, Balance) as its owner entity
- Partial key
  - A set of attributes that can uniquely identify weak entities related to the same owner entity eg. Txn# is a partial key in Transaction entity
- To use weak entity types or not?
  - Basically the designers choice.
  - Preferable if it has many attributes and participates in relationships besides its owner entity types.

Generalization

- Relationships among entity types
  - To emphasize the similarities among lower-level entity types and to hide their differences
  - Attributes of higher-level entity sets are inherited by lower-level entity sets



Transforming ISA into Relations

- Create a relation for the higher-level entity set, and for each lower-level entity set, create a relation with the primary key of the higher-level entity set

  Account (Number, Balance)

  Savings-acct (Number, Interest-rate)

  Checking-acct (Number, Overdraft-amount)

- Do not create for higher-level entity set. For each lower-level entity set, create a relation with all the attributes of the higher-level entity set

  Savings-acct (Number, Balance, Interest-rate)

  Checking-acct (Number, Balance, Overdraft-amount)

The second method is possible only when the generalization is

Disjoint: no entity belongs to more than 2 subclass

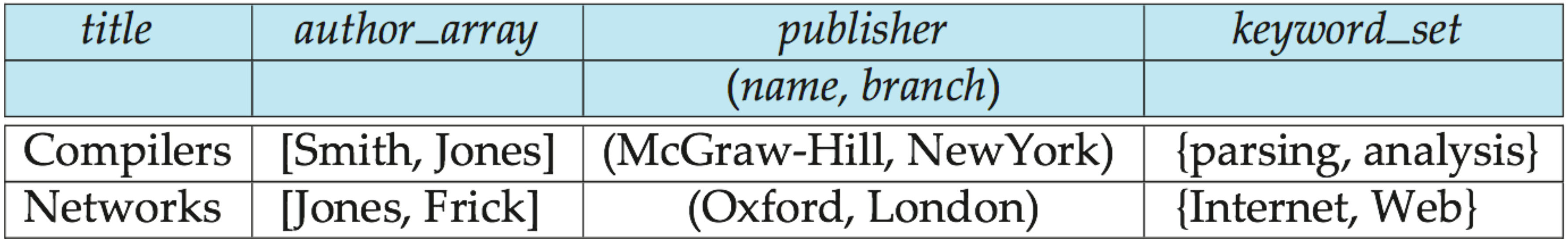
Complete: every member of superclass is a member of subclass

Complex Data Types

- Motivation:
  - Permit non-atomic domains (atomic $\equiv$ indivisible)
  - Example of non-atomic domain: set of integers or set of tuples
  - Allows more intuitive modeling for applications with complex data
- Intuitive definition:
  - allow relations whenever we allow atomic (scalar) values
    - relations within relations
  - Retains mathematical foundation of relational model
  - Violates first normal form

Example of a Nested Relation

- Example: library information system
- Each book has
  - title,
  - a list (array) of authors,
  - Publisher, with subfields *name* and *branch*, and
  - a set of keywords
- Non-1NF relation *books*



title	*author_array*	*publisher*	*keyword_set*
		(*name*, *branch*)	
Compilers	[Smith, Jones]	(McGraw-Hill, NewYork)	{parsing, analysis}
Networks	[Jones, Frick]	(Oxford, London)	{Internet, Web}

4NF Decomposition of Nested Relation

- Suppose for simplicity that title uniquely identifies a book
  - In real world ISBN is a unique identifier
- Decompose *books* into 4NF using the schemas:
  - (*title, author, position*)
  - (*title, keyword*)
  - (*title, pub-name, pub-branch*)
- 4NF design requires users to include joins in their queries.

title	*author*	*position*
Compilers	Smith	1
Compilers	Jones	2
Networks	Jones	1
Networks	Frick	2

authors

title	*keyword*
Compilers	parsing
Compilers	analysis
Networks	Internet
Networks	Web

keywords

title	*pub_name*	*pub_branch*
Compilers	McGraw-Hill	New York
Networks	Oxford	London

books4

Complex Types and SQL

- Extensions introduced in SQL:1999 to support complex types:
  - Collection and large object types
    - Nested relations are an example of collection types
  - Structured types
    - Nested record structures like composite attributes
  - Inheritance
  - Other object orientation features
    - Including object identifiers and references
- Not fully implemented in all the database system
  - But some features are present in each of the major commercial database systems
    - Read the manual of your database system to see what it supports

Structured Types and Inheritance in SQL

- **Structured types** (**user-defined types**) can be declared and used in SQL

```
create type Name as
    (firstname      varchar(20),
     lastname       varchar(20))
     final

create type Address as
    (street      varchar(20),
     city           varchar(20),
     zipcode      varchar(20))
     not final
```

  - Note: **final** and **not final** indicate whether subtypes can be created
- Structured types can be used to create tables with composite attributes

```
create table person (
    name        Name,
    address     Address,
    dateOfBirth date)
```

- Dot notation used to reference components: ***name.firstname***

Structured Types (cont.)

- **User-defined row types**

```
create type PersonType as (
    name Name,
    address Address,
    dateOfBirth date)
    not final
```

- Can then create a table whose rows are a user-defined type

```
create table customer of PersonType
```

- Alternative using **unnamed row types**.

```
create table person_r(
        name       row(firstname  varchar(20),
                       lastname   varchar(20)),
        address    row(street     varchar(20),
                       city       varchar(20),
                       zipcode    varchar(20)),
        dateOfBirth date)
```

Methods

- Can add a method declaration with a structured type.

  method *ageOnDate* (*onDate* **date**)
          returns interval year

- Method body is given separately.

```
create instance method ageOnDate (onDate date)
        returns interval year
        for CustomerType
begin
        return onDate - self.dateOfBirth;
end
```

- We can now find the age of each customer:

```
select name.lastname, ageOnDate (current_date)
from customer
```

Constructor Functions

- **Constructor functions** are used to create values of structured types
- E.g.
  create function *Name*(*firstname* **varchar**(20), *lastname* **varchar**(20))
  returns *Name*
  begin
      set self.*firstname = firstname;*
      set self.*lastname = lastname;*
  end
- To create a value of type *Name,* we use
    new *Name*('John', 'Smith')
- Normally used in insert statements
  insert into *Person* **values**
      (**new** *Name*('John', 'Smith),
       new *Address*('20 Main St', 'New York', '11001'),
       date '1960-8-22');

Type Inheritance

- Suppose that we have the following type definition for people:

```
create type Person
    (name varchar(20),
     address varchar(20))
```

- Using inheritance to define the student and teacher types

```
create type Student     under Person
  (degree        varchar(20),
   department  varchar(20))
create type Teacher     under Person
  (salary          integer,
   department  varchar(20))
```

- Subtypes can redefine methods by using **overriding method** in place of **method** in the method declaration

Multiple Type Inheritance

- SQL:1999 and SQL:2003 do not support multiple inheritance
- If our type system supports multiple inheritance, we can define a type for teaching assistant as follows:

  create type *Teaching Assistant*
  under *Student, Teacher*

- To avoid a conflict between the two occurrences of *department* we can rename them

  create type *Teaching Assistant*
  under
  Student **with** (*department* **as** *student_dept*),
  Teacher **with** (*department* **as** *teacher_dept*)

- Each value must have a **most-specific type**

Table Inheritance

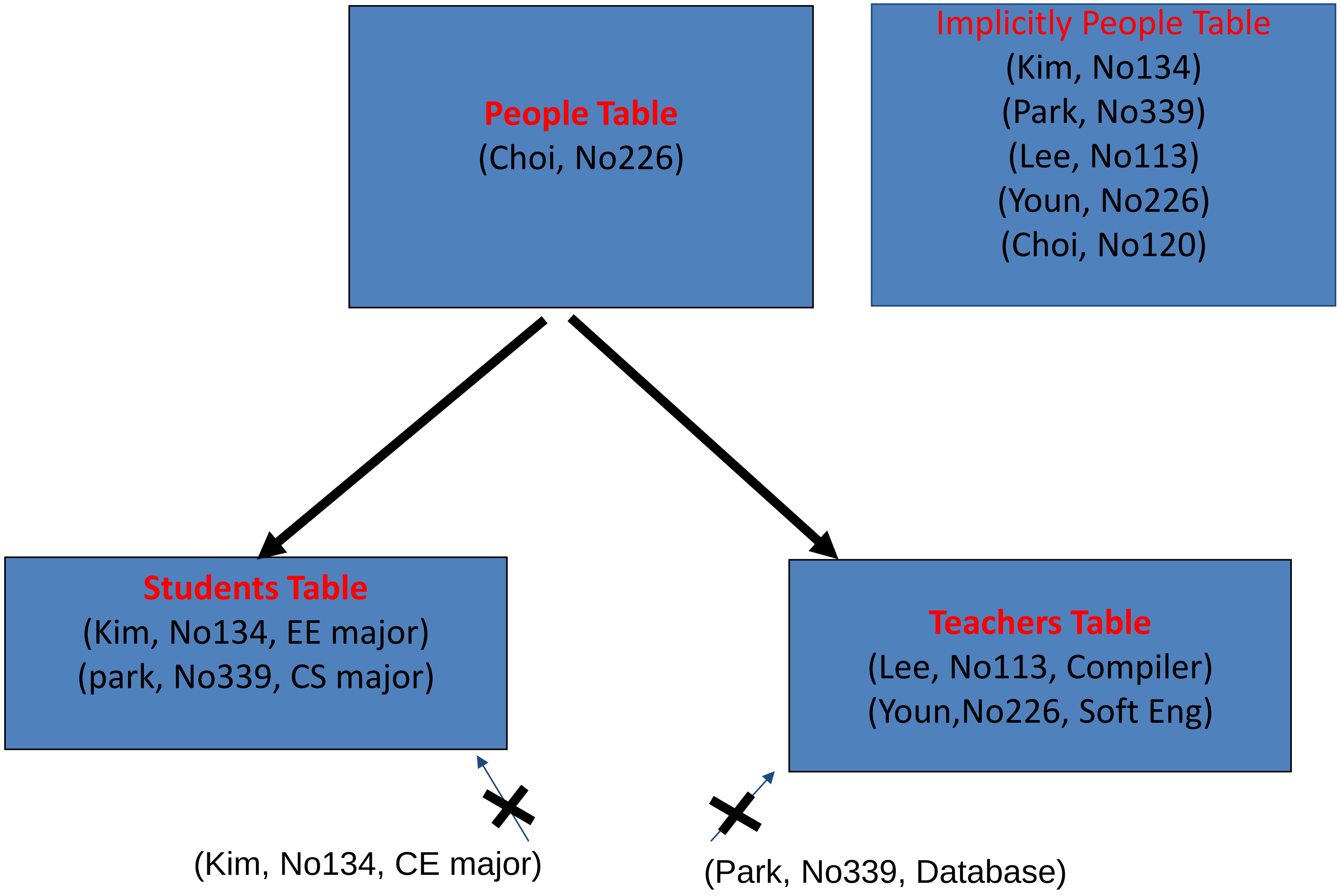
- Tables created from subtypes can further be specified as **subtables**
- E.g. **create table** *people* **of** *Person;*
  create table *students* **of** *Student* **under** *people;*
  create table *teachers* **of** *Teacher* **under** *people;*
- Tuples added to a subtable are automatically visible to queries on the supertable
  - E.g. query on *people* also sees *students* and *teacher*s.
  - Similarly updates/deletes on *people* also result in updates/deletes on subtables
  - To override this behaviour, use "**only** *people"* in query
- Conceptually, multiple inheritance is possible with tables
  - e.g. *teaching_assistants* under *students* and *teachers*
  - *But is not supported in SQL currently*
    - So we cannot create a person (tuple in *people*) who is both a student and a teacher

Consistency Requirements for Subtables

- Consistency requirements on subtables and supertables.
  - Each tuple of the supertable (e.g. *people)* can correspond to at most one tuple in each of the subtables (e.g. *students* and *teachers)*
  - Additional constraint in SQL:1999:

    All tuples corresponding to each other (that is, with the same values for inherited attributes) must be derived from one tuple (inserted into one table).
    - That is, each entity must have a most specific type
    - We cannot have a tuple in *people* corresponding to a tuple each in *students* and *teachers*

Subtable Consistency



Array and Multiset Types in SQL

- Example of array and multiset declaration:

```
create type Publisher as
  (name          varchar(20),
   branch        varchar(20));
create type Book as
  (title         varchar(20),
   author_array  varchar(20) array [10],
   pub_date      date,
   publisher     Publisher,
   keyword-set   varchar(20) multiset);
 create table books of Book;
```

Creation of Collection Values

- Array construction
  array [‘Silberschatz’,`Korth’,`Sudarshan’]

- Multisets
  multiset [‘computer’, ‘database’, ‘SQL’]

- To create a tuple of the type defined by the books relation:
  (‘Compilers’, **array**[`Smith’,`Jones’],
  new *Publisher* (`McGraw-Hill’,`New York’),
  multiset [`parsing’,`analysis’])

- To insert the preceding tuple into the relation books
  insert into *books*
  values
  (‘Compilers’, **array**[`Smith’,`Jones’],
  new *Publisher* (`McGraw-Hill’,`New York’),
  multiset [`parsing’,`analysis’]);

Querying Collection-Valued Attributes

- To find all books that have the word "database" as a keyword,

  select *title*
  from *books*
  where **'**database' **in** (**unnest**(*keyword-set*))

- We can access individual elements of an array by using indices
  - E.g.: If we know that a particular book has three authors, we could write:

  select *author_array*[1], *author_array*[2], *author_array*[3]
  from *books*
  where *title* = `Database System Concepts'

- To get a relation containing pairs of the form "title, author_name" for each book and each author of the book

  select *B.title, A.author*
  from *books* **as** *B*, **unnest** (*B.author_array*) **as** *A* (*author*)

- To retain ordering information we add a **with ordinality** clause

  select *B.title, A.author, A.position*
  from *books* **as** *B*, **unnest** (*B.author_array*) **with ordinality as**
  A (*author, position*)

Unnesting

- The transformation of a nested relation into a form with fewer (or no) relation-valued attributes is called **unnesting**.
- E.g.

  select *title*, *A* **as** *author*, *publisher.name* **as** *pub_name*,
  publisher.branch **as** *pub_branch*, *K.keyword*
  from *books* **as** *B*, **unnest**(*B.author_array*) **as** *A* (*author*),
  unnest (*B.keyword_set*) **as** *K* (*keyword*)
- Result relation *flat_books*

title	*author*	*pub_name*	*pub_branch*	*keyword*
Compilers	Smith	McGraw-Hill	New York	parsing
Compilers	Jones	McGraw-Hill	New York	parsing
Compilers	Smith	McGraw-Hill	New York	analysis
Compilers	Jones	McGraw-Hill	New York	analysis
Networks	Jones	Oxford	London	Internet
Networks	Frick	Oxford	London	Internet
Networks	Jones	Oxford	London	Web
Networks	Frick	Oxford	London	Web

Querying Collection-Valued Attributes

- To find all books that have the word “database” as a keyword,

```
select title
        from books
        where ‘database’ in (unnest(keyword-set ))
```

- We can access individual elements of an array by using indices
  - E.g.: If we know that a particular book has three authors, we could write:

select *author-array*[1], *author-array*[2], *author-array*[3]
  from *books*
  where *title* = `Database System Concepts’

- To get a relation containing pairs of the form “title, author-name” for each book and each author of the book

select *B.title, A.author*

from *books* **as** *B*, **unnest** (*B.author-array*) **as** *A* (*author*)

- To retain ordering information we add a **with ordinality** clause **select** *B.title, A.author, A.position*

from *books* **as** *B*, **unnest** (*B.author-array*) **with ordinality as** *A* (*author, position*)

Nesting

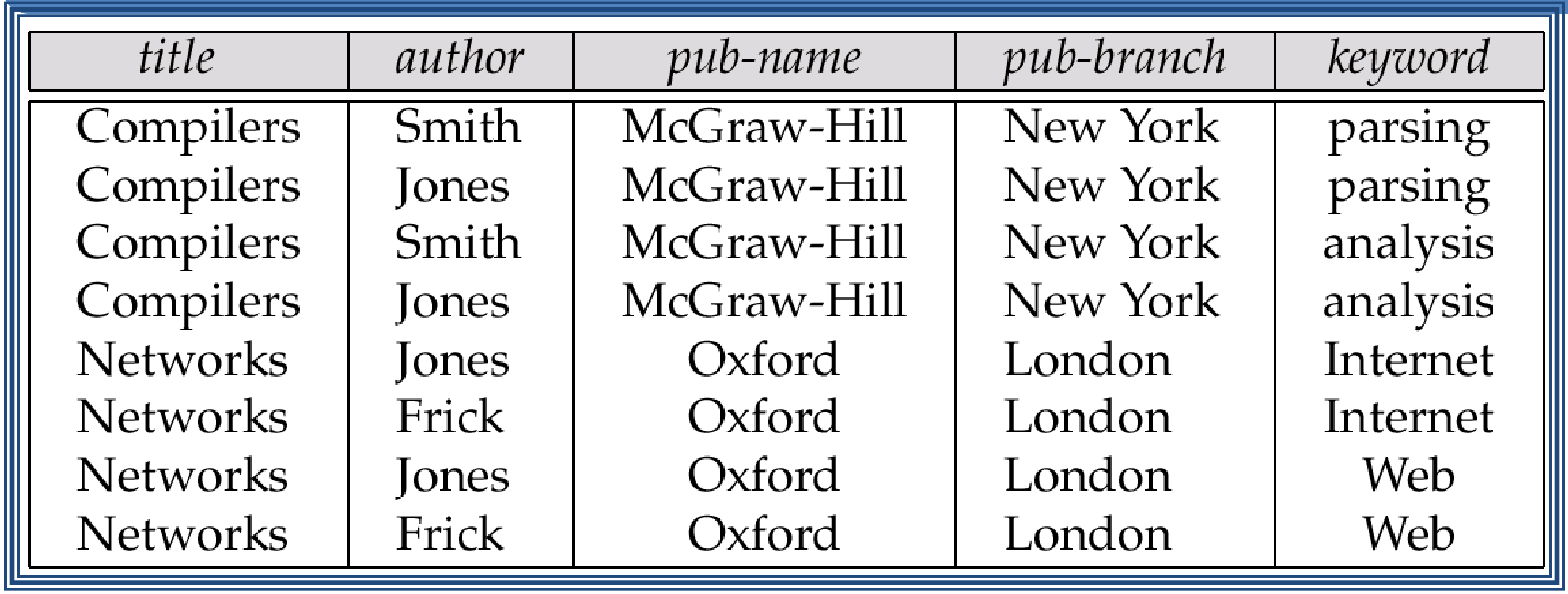
- **Nesting** is the opposite of unnesting, creating a collection-valued attribute
- Nesting can be done in a manner similar to aggregation, but using the function **colect**() in place of an aggregation operation, to create a multiset
- To nest the *flat_books* relation on the attribute *keyword*:

```
select title, author, Publisher (pub_name, pub_branch ) as publisher,
        collect (keyword)  as keyword_set
from flat_books
groupby title, author, publisher
```

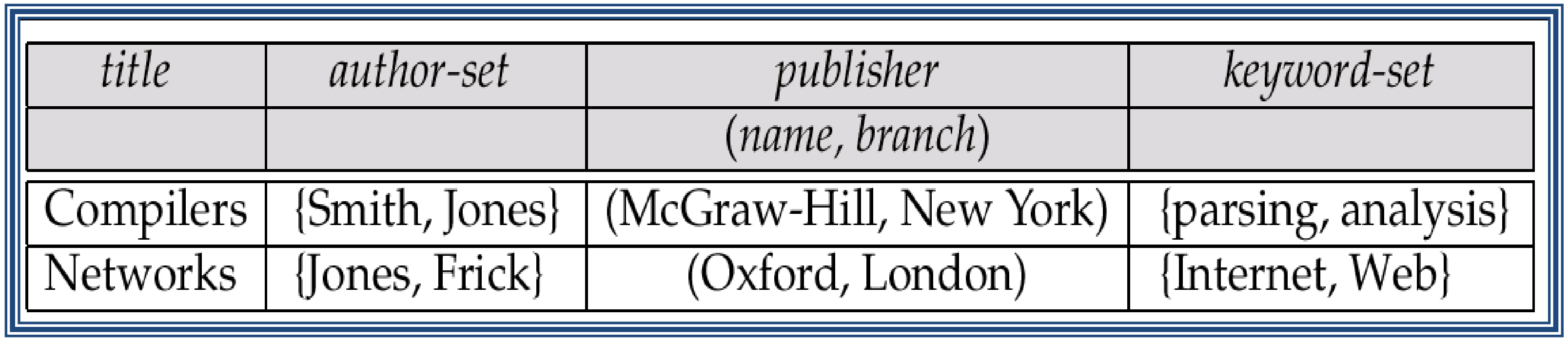
- To nest on both authors and keywords:

```
select title, collect (author ) as author_set,
        Publisher (pub_name, pub_branch) as publisher,
                collect  (keyword ) as keyword_set
from   flat_books
group by title, publisher
```

Nesting(Collect)



title	*author*	*pub-name*	*pub-branch*	*keyword*
Compilers	Smith	McGraw-Hill	New York	parsing
Compilers	Jones	McGraw-Hill	New York	parsing
Compilers	Smith	McGraw-Hill	New York	analysis
Compilers	Jones	McGraw-Hill	New York	analysis
Networks	Jones	Oxford	London	Internet
Networks	Frick	Oxford	London	Internet
Networks	Jones	Oxford	London	Web
Networks	Frick	Oxford	London	Web



title	*author-set*	*publisher*	*keyword-set*
		(name, branch)	
Compilers	{Smith, Jones}	(McGraw-Hill, New York)	{parsing, analysis}
Networks	{Jones, Frick}	(Oxford, London)	{Internet, Web}

** note: group by title, publisher

Nesting (Cont.)

- Another approach to creating nested relations is to use subqueries in the **select** clause, starting from the 4NF relation *books4*

```
select title,
        array (select author
                  from authors as A
                  where A.title = B.title
                                      order by A.position) as
author_array,
        Publisher (pub-name, pub-branch) as publisher,
        multiset (select keyword
                      from keywords as K
                      where K.title = B.title) as keyword_set
from books4 as B
```

Object-Identity and Reference Types

- Define a type *Department* with a field *name* and a field *head* which is a reference to the type *Person,* with table *people* as scope:

  create type *Department* (
                name **varchar** (20),
                head **ref** (*Person*) **scope** *people*)

- We can then create a table *departments* as follows

      create table *departments* **of** *Department*

- We can omit the declaration **scope** people from the type declaration and instead make an addition to the **create table** statement:

        create table *departments* **of** *Department*
            (*head* **with options scope** *people*)

- Referenced table must have an attribute that stores the identifier, called the **self-referential attribute**

        create table *people* **of** *Person*
        ref is *person_id* **system generated;**

Initializing Reference-Typed Values

- To create a tuple with a reference value, we can first create the tuple with a null reference and then set the reference separately:

  insert into *departments*
          values (`CS’, null)
  update *departments*
        set *head* = (**select** *p.person_id*
                          from *people* **as** *p*
                        where *name* = `John’)
        where *name* = `CS’

User Generated Identifiers

- The type of the object-identifier must be specified as part of the type definition of the referenced table, and
- The table definition must specify that the reference is user generated

```
create type Person
    (name varchar(20)
     address varchar(20))
   ref using varchar(20)
create table people of Person
  ref is person_id user generated
```

- When creating a tuple, we must provide a unique value for the identifier:

```
insert into people (person_id, name, address ) values
    ('01284567', 'John', `23 Coyote Run')
```

- We can then use the identifier value when inserting a tuple into *departments*
  - Avoids need for a separate query to retrieve the identifier:

```
insert into departments
values(`CS', `02184567')
```

User Generated Identifiers (Cont.)

- Can use an existing primary key value as the identifier:

  create type *Person*
  　　(*name* **varchar** (20) **primary key**,
  　　 address **varchar**(20))
  　ref from (*name*)
  create table *people* **of** *Person*
  　ref is *person_id* **derived**

- When inserting a tuple for *departments*, we can then use

  insert into *departments*
  　values(`CS’,`John’)

Path Expressions

```
create type Department (
                  name  varchar (20),
                  head   ref (Person) scope people)
create type Person
         ( name          varchar(20)
           address       varchar(20) )
           ref   using   varchar(20)
```

•Find the names and addresses of the heads of all departments:

```
           select head →name, head → address
             from departments
```

•An expression such as "head → name" is called a **path expression**

•Path expressions help avoid explicit joins

- If department head were not a reference, a join of *departments* with *people* would be required to get at the address
- Makes expressing the query much easier for the user

Implementing O-R Features in RDB

- If we want to keep existing RDBMS and utilize O-R advantages
  - Structured Type, Array, Multiset, Nested relations, Inheritance, Subtable
- Convert tables with O-R tables into Relational Tables
  - Similar to how E-R features are mapped onto relation schemas
  - Multivalued attribute vs Multi-Set valued attribute
  - Composite attribute vs Structured Type
  - ISA vs Table Inheritance
- Subtable implementation
  - Each table stores primary key and those attributes locally defined in that table or,
  - Each table stores both locally defined and inherited attributes

Persistent OO Programming Languages

- Languages extended with constructs to handle persistent data
- Programmer can manipulate persistent data directly
  - no need to fetch it into memory and store it back to disk (unlike embedded SQL)
- Supporting Persistent Objects inside Programming Language!
- Persistent objects:
  - **Persistence by class** - explicit declaration of persistence
  - **Persistence by creation** - special syntax to create persistent objects
  - **Persistence by marking** - make objects persistent after creation
  - **Persistence by reachability** - object is persistent if it is declared explicitly to be so or is reachable from a persistent object

Concerns in Persistent PL

- Object Identifiers
  - We need stronger version of in-memory pointers in Persistent PL
  - Degrees of permanence of object identity
    - Intraprocedure: only during execution of a single procedure
    - Intraprogram: only during execution of a single program or query
    - Interprogram: across program executions, but not if data-storage format on disk changes
    - Persistent: interprogram, plus persistent across data reorganizations
- How to represent class and its instances
- How to support Query
- How to support Transaction

Object Identity and Pointers

- Degrees of permanence of object identity
  - **Intraprocedure**: only during execution of a single procedure
  - **Intraprogram**: only during execution of a single program or query
  - **Interprogram**: across program executions, but not if data-storage format on disk changes
  - **Persistent**: interprogram, plus persistent across data reorganizations
- Persistent versions of C++ and Java have been implemented
  - C++
    - ODMG C++
    - ObjectStore
  - Java
    - Java Database Objects (JDO)

Persistent C++ Systems

- Extensions of C++ language to support persistent storage of objects
- Several proposals, ODMG standard proposed, but not much action of late
  - **persistent pointers**: e.g. d_Ref<T>
  - **creation of persistent objects**: e.g. new (db) T()
  - **Class extents**: access to all persistent objects of a particular class
  - **Relationships:** Represented by pointers stored in related objects
    - Issue: consistency of pointers
    - Solution: extension to type system to automatically maintain back-references
  - **Iterator interface**
  - **Transactions**
  - **Updates:** mark_modified() function to tell system that a persistent object that was fetched into memory has been updated
  - **Query language**

Persistent Java Systems

- Standard for adding persistence to Java : **Java Database Objects (JDO)**
  - Persistence by reachability
  - Byte code enhancement
    - Classes separately declared as persistent
    - Byte code modifier program modifies class byte code to support persistence
      - E.g. Fetch object on demand
      - Mark modified objects to be written back to database
  - Database mapping
    - Allows objects to be stored in a relational database
  - Class extents
  - Single reference type
    - no difference between in-memory pointer and persistent pointer
    - Implementation technique based on **hollow objects** (a.k.a. **pointer swizzling**)

Object-Relational Mapping

- **Object-Relational Mapping (ORM)** systems built on top of traditional relational databases
- Implementor provides a mapping from objects to relations
  - Objects are purely transient, no permanent object identity
- Objects can be retried from database
  - System uses mapping to fetch relevant data from relations and construct objects
  - Updated objects are stored back in database by generating corresponding update/insert/delete statements
- The **Hibernate** ORM system is widely used
  - described in Section 9.4.2
  - Provides API to start/end transactions, fetch objects, etc
  - Provides query language operating direcly on object model
    - queries translated to SQL
- Limitations: overheads, especially for bulk updates

Comparison of O-O and O-R Databases

- **Relational systems**
  - simple data types, powerful query languages, high protection.
- **Persistent-programming-language-based OODBs**
  - complex data types, integration with programming language, high performance.
- **Object-relational systems**
  - complex data types, powerful query languages, high protection.
- **Object-relational mapping systems**
  - complex data types integrated with programming language, but built as a layer on top of a relational database system
- Note: Many real systems blur these boundaries
  - E.g. persistent programming language built as a wrapper on a relational database offers first two benefits, but may have poor performance.