BITS Pilani
Pilani Campus

QUERY PROCESSING

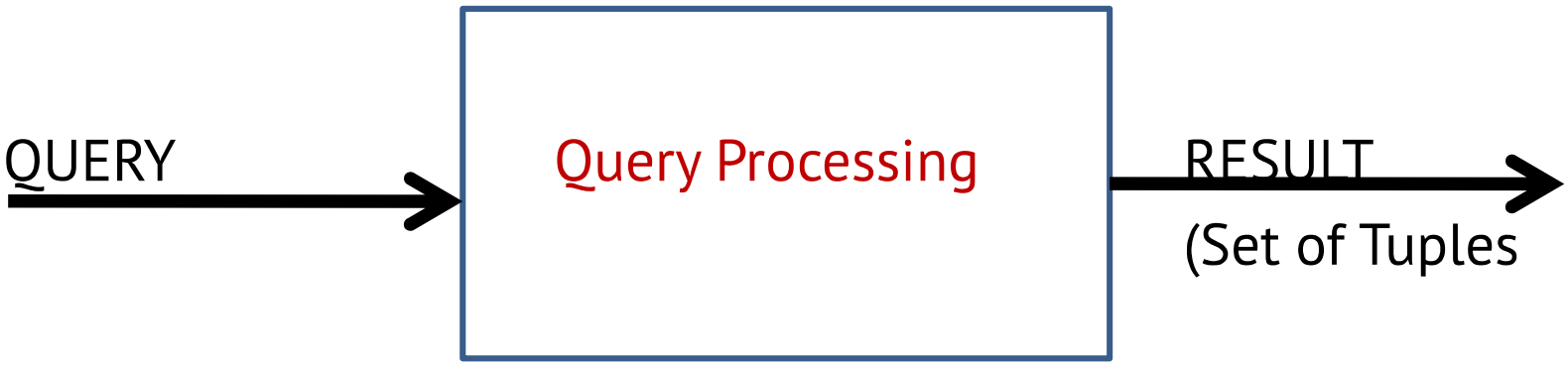
Learning Objectives

- What is meant by Query Processing?
- Different steps for executing a query
  - Parsing and Translation
  - Optimizing
  - Evaluating
- Step by step procedure for evaluating
  - Single relation
  - Multiple tables
- Measuring the cost

QUERY PROCESSING

Query processing is the list of activities that are performed to obtain the required tuples that satisfy the given query.

Query processing refers to the process to answer a **query** to a database or an information system, which usually involves interpreting the **query**, searching through the space storing data, and retrieving the results satisfying the **query**. Query processing is pictorially represented as follows:



QUERY PROCESSING

Example:

Assume that an Employee table has the following data

EMPLOYEE

Ename	Eno	Designation	Salary
Sachin	10001	Manager	98000
Sourav	10010	Asst. Manager	92000
Dhoni	10003	System Analyst	90000
Kohli	10012	HR Manager	85670
Pant	11111	Programmer	76700
Yuvaraj	12000	Programmer	72000

QUERY PROCESSING

When a query is written as

Select Ename, Eno, Salary

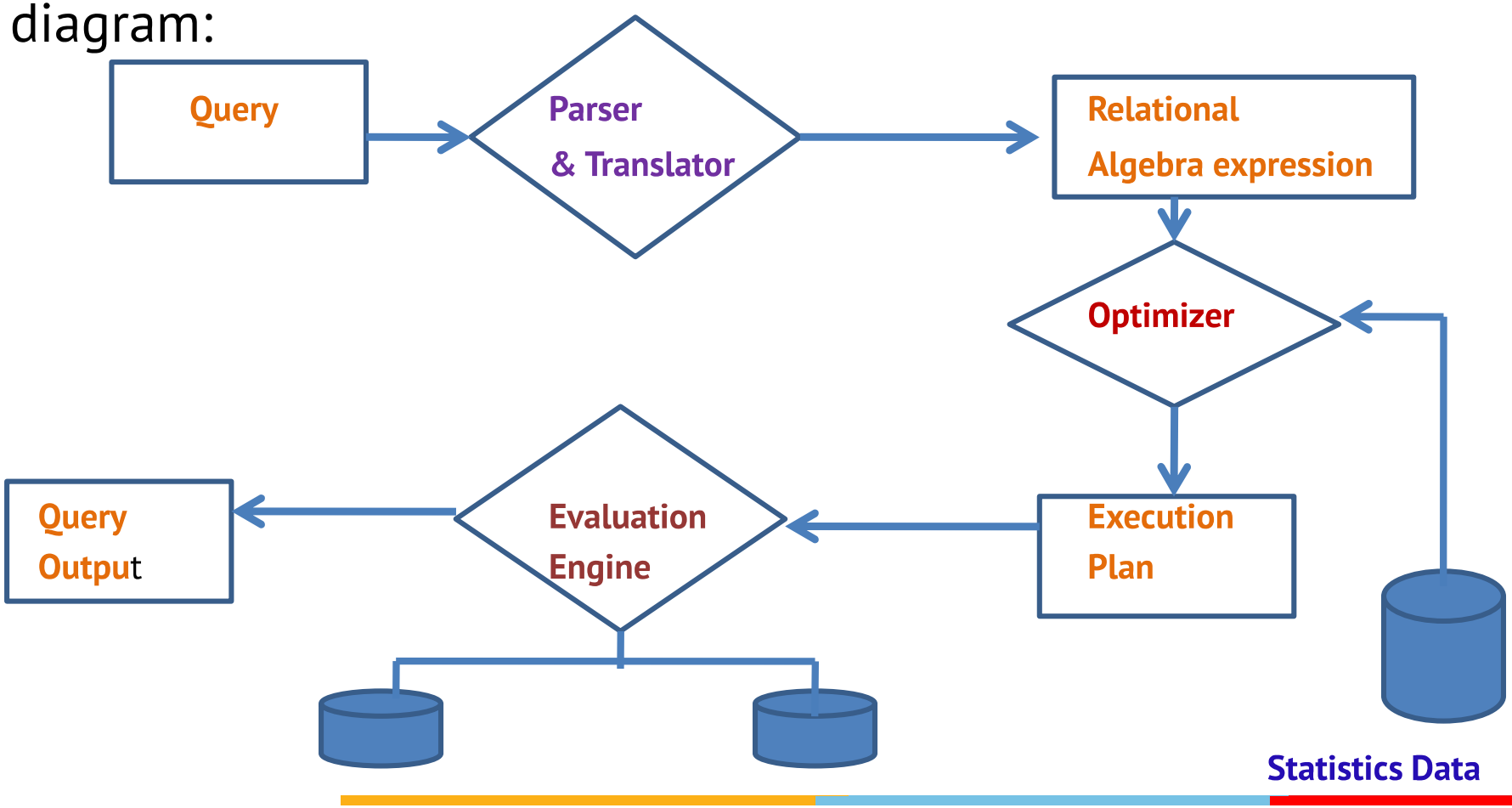
from EMPLOYEE where salary >= 90000;

It resulted in the following table

Ename	Eno	Designation	Salary
Sachin	10001	Manager	98000
Sourav	10010	Asst. Manager	92000
Dhoni	10003	System Analyst	90000

QUERY PROCESSING

A query is processed in various steps. The query which is executed in various steps are represented in the following diagram:



QUERY PROCESSING

PARSER AND TRANSLATOR

Parser does the following:

(i) Checks whether the SQL command (Query) is written in the correct syntax.

For example,

Assume that a query is entered as

```
select Ename,Eno,Salary from Employee having
                    salary > 90000;
```

The SQL query has a syntax error as it is invalid because of having option . So it displays the error message. The error message is generated by parser

QUERY PROCESSING

(ii) Checks whether elements (attributes and the table specified in the query) are available in the relational schema.

For Example,

```
select Empname,Eno,Salary from Employee where
        salary > 90000;
```

The attribute EmpName is not the element of relational schema.

So it displays the error message as

Such a Schema element does not exist

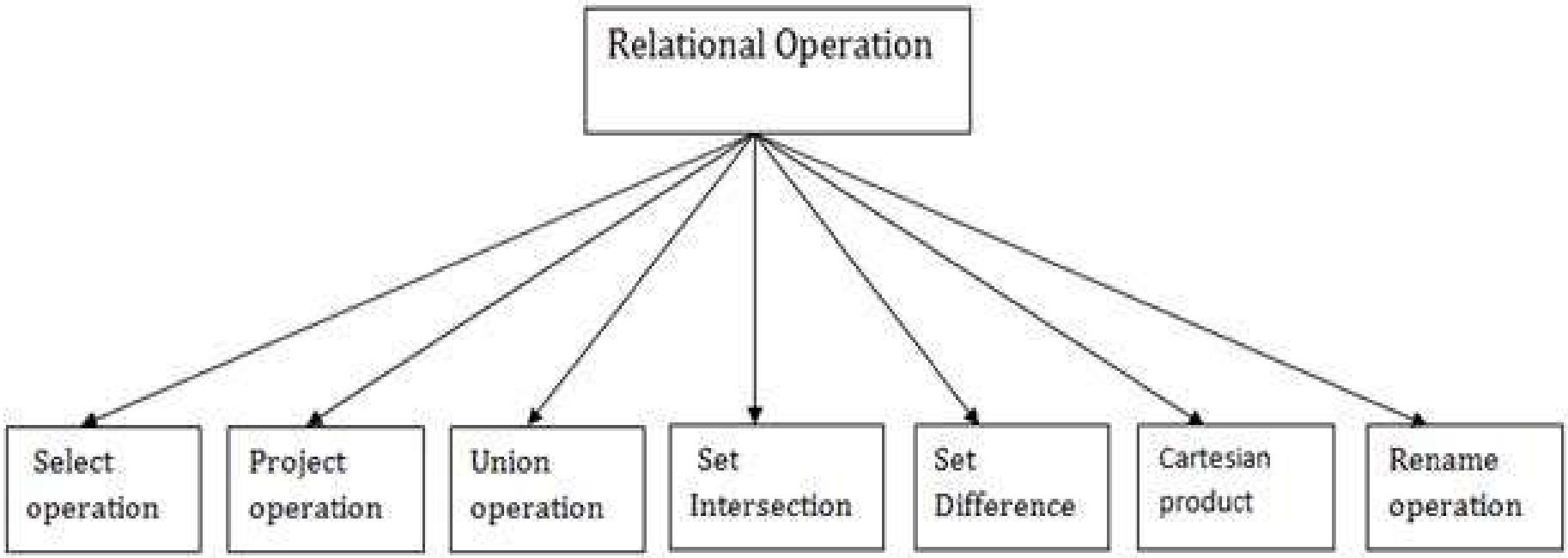
It also checks whether the table names specified in the query are available in the database

(iii) Converts the query into relational algebra expression.

QUERY PROCESSING

Revisiting reltional algebra expression

Relational algebra is a procedural query language. It gives a step by step process to obtain the result of the query. It uses operators to perform queries.



QUERY PROCESSING

1. Select Operation:

The select operation selects tuples that satisfy a given predicate. It is denoted by sigma (σ).

2. Project Operation:

This operation shows the list of those attributes that we wish to appear in the result. Rest of the attributes are eliminated from the table.It is denoted by ∏.

3. Union Operation:

Suppose there are two tuples R and S. The union operation contains all the tuples that are either in R or S or both in R & S. It eliminates the duplicate tuples. It is denoted by ∪.

QUERY PROCESSING

∏ CUSTOMER_NAME (BORROW) ∪ ∏ CUSTOMER_NAME (DEPOSITOR)

4. Set Intersection:

Suppose there are two tuples R and S. The set intersection operation contains all tuples that are in both R & S.

It is denoted by intersection ∩.

∏ CUSTOMER_NAME (BORROW) ∩ ∏ CUSTOMER_NAME (DEPOSITOR)

5. Set Difference:

Suppose there are two tuples R and S. The set intersection operation contains all tuples that are in R but not in S.

It is denoted by intersection minus (-). Notation: R - S

QUERY PROCESSING

Example: Using the above DEPOSITOR table and BORROW table

Input:

∏ CUSTOMER_NAME (BORROW) -
∏ CUSTOMER_NAME (DEPOSITOR)

6. Cartesian product

The Cartesian product is used to combine each row in one table with each row in the other table. It is also known as a cross product.

It is denoted by X.

EMPLOYEE X DEPARTMENT

QUERY PROCESSING

OPTIMIZER

It is a component that finds the best way / best plan to evaluate Relational algebra expression.

If we want to display names, salary of the employees in an employee table who get salary more than 90,000.

The SQL command for the above query is

```
Select Ename, Eno, Salary
        from    EMPLOYEE where salary > 90000;
```

The parser and translator convert the SQL command into the relational algebra expression as

$$\Pi_{\text{ename, salary}} \; \sigma_{\text{salary} > 90000} (\text{EMPLOYEE})$$

QUERY PROCESSING

This SQL command can be represented in the following relational algebra expression also

$$\sigma_{\text{salary} > 90000} \; \Pi_{\text{ename, salary}} \; (\text{EMPLOYEE})$$

Both the relational algebra expressions are valid expressions.

These expressions are called as equivalent relational algebra expressions.

The relational algebra expressions are called as the equivalent expressions if they produce the same output in the execution.

So, for any given SQL command there may be more than one relational algebra expressions..

- Though both the expressions produce the similar output , their execution may be different.

QUERY PROCESSING

- They have different CPU time, CPU cycles and disk access

Out of all relational algebra expressions, the expression with the least cost is to be selected. We know that cost depends upon the factors such as time, CPU utilization, etc.,

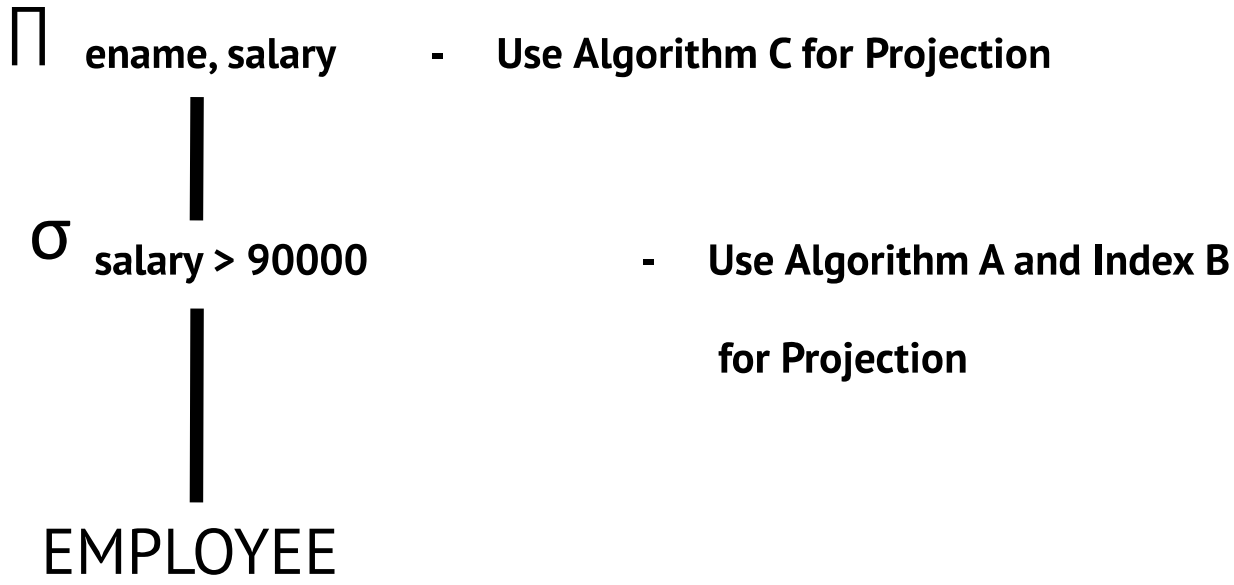
All the operations such as projection, selection are performed with various algorithms.

The optimizer creates the query evaluation plan that specifies which relational algebra expression is to be chosen and what algorithm is to be used to evaluate. Regarding the decision on cost of relational algebra expression, optimizer uses statistics about data that are available in the secondary storage device. Statistical data contains the information such as

QUERY PROCESSING

- Size of the relation
- No of blocks used
- No of records accessed

The query evaluation plan above SQL command has the following representation:



QUERY PROCESSING

- The above tree is called as operator tree
- It is also called as Query tree

The output of the optimizer is the query evaluation plan. The Query evaluation engine evaluates the above query evaluation plan and produce the result

QUERY EVALUATION

Query Execution Engine takes a query evaluation plan, executes the plan and returns the result (answer) in the form of the table to the query

The SQL command

Select balance from account where balance < 25000;

is translated into either of the following relational expressions

σ balance < 25000 (Π balance (account))

Π balance (σ balance < 25000 (account))

Though both the above expressions produce the same output but their execution will be different

The Query which takes less CPU time, CPU cycle or disk access will have less cost and will be executed

QUERY EVALUATION

- Different evaluation plans that are created in the query processing step for a given query can have different costs
- We can not expect users to write their queries in such a way that it suggests the most efficient evaluation plan
- Rather, it is the responsibility of the system to construct a query evaluation plan that minimizes the cost of query evaluation
- This is where query optimization comes into play

Query Processing with multiple Tables:

Assume that there is a query such as

- **Find names of all customers who have an account at any branch located in Chennai**

QUERY EVALUATION

Π Cname (σ branch_City =”Chennai” ((branch X (account X depositor))) - **Sql 1**

OR

Π Cname ((σ branch_City =”Chennai” (branch)) X (account X depositor)) - **Sql 2**

Assume that there are 100 tuples of branch, account and depositor tables.

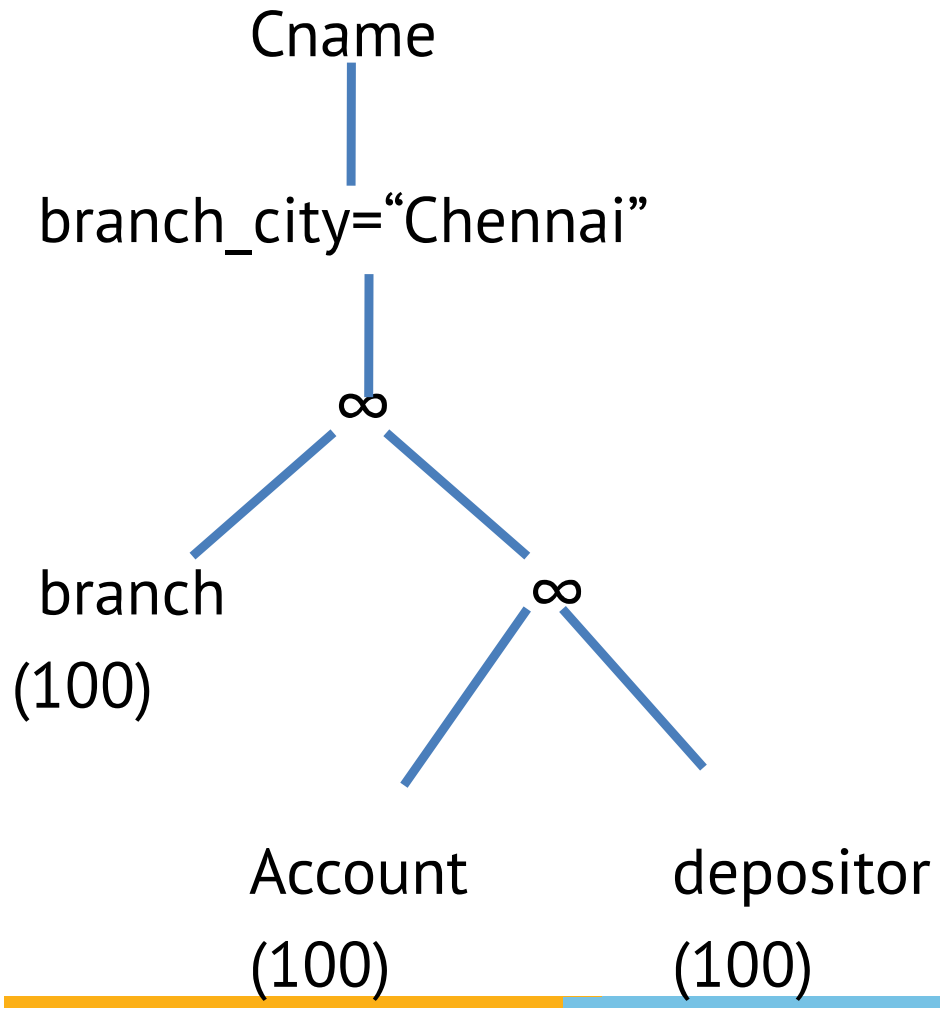
Then Sql 1 generates larger intermediate relation as 100 tuples of depositor are joined with 100 tuples of account and 100 tuples of branch

But for Sql 2, 100 tuples of account are joined with 100 tuples of depositor and then joined with 20 tuples of branch [we assume

QUERY EVALUATION

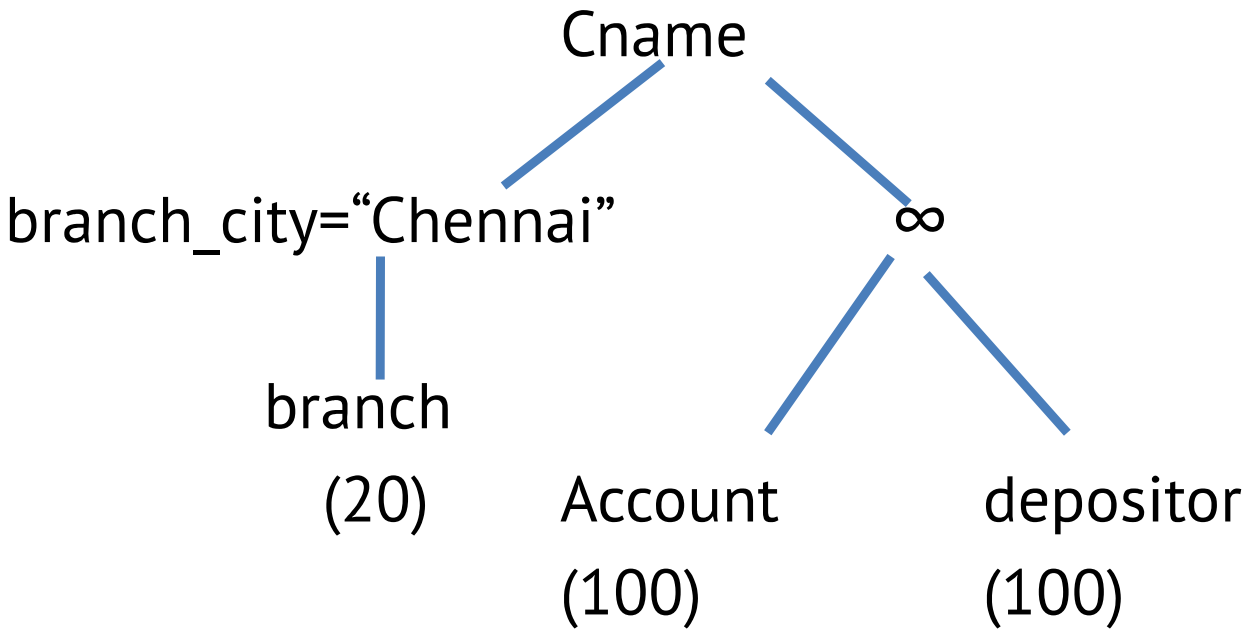
that there are 50 chennai data available in branch]

For query Sql 1, the expression can be drawn as



QUERY EVALUATION

For query Sql 2, the expression can be drawn in smaller intermediate relation as



QUERY EVALUATION

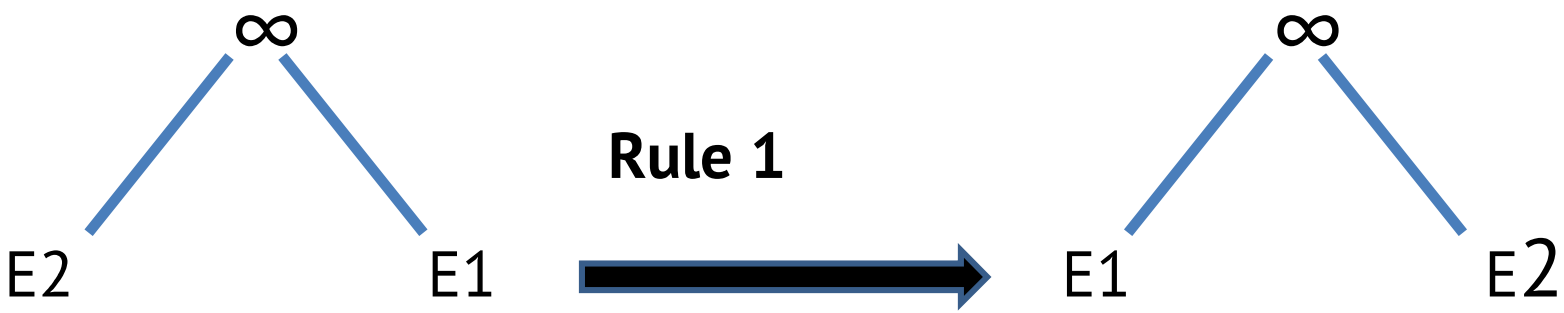
- Both the expressions are equivalence expressions. The expressions which produce the similar output are called as Equivalence Expressions]
- Generation of query – evaluation plan involves two steps: Generating expressions that are logically equivalent to the given expression
- Estimating the cost of each evaluation plan

To implement the first step, the query optimizer must generate expressions equivalent to a given expressions. It does so by means of equivalence rules that specify how to transform an expression into a logically equivalent one.

QUERY EVALUATION

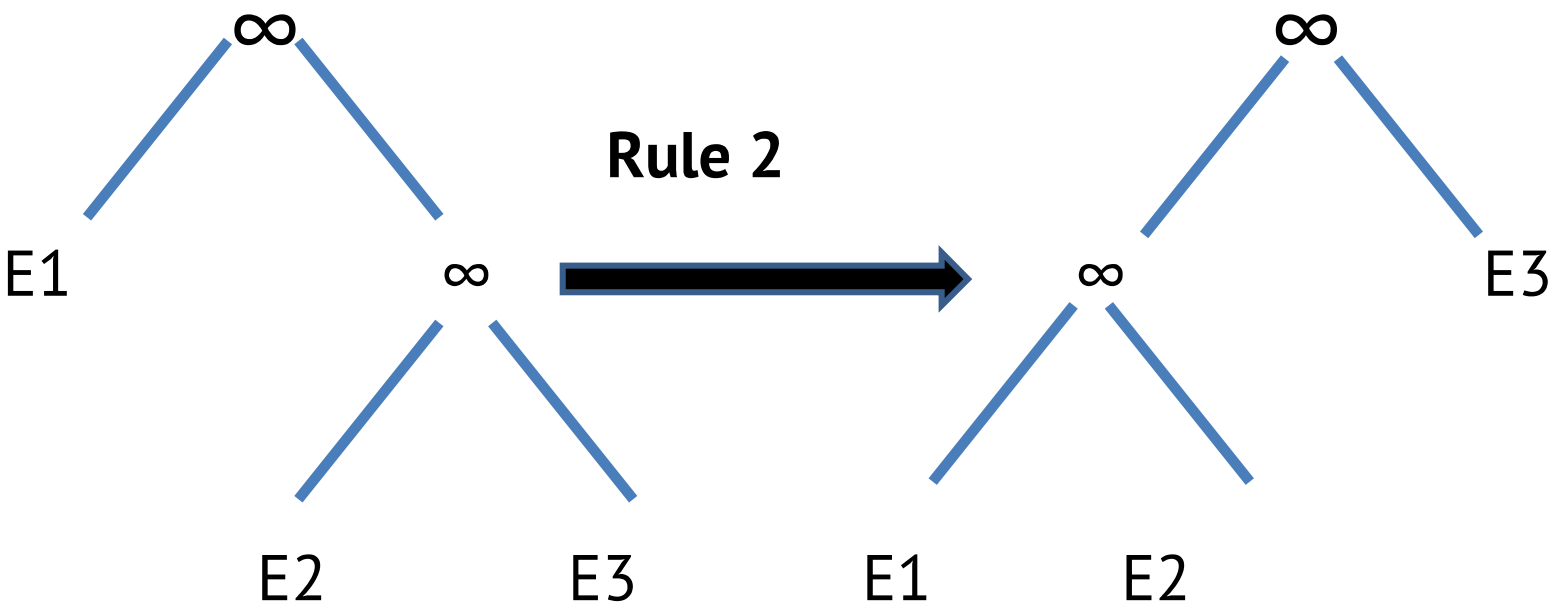
Equivalence Rules

An equivalence rule says that expressions of two forms are equivalent. We can replace an expression of the first form by an expression of the second form or vice versa.



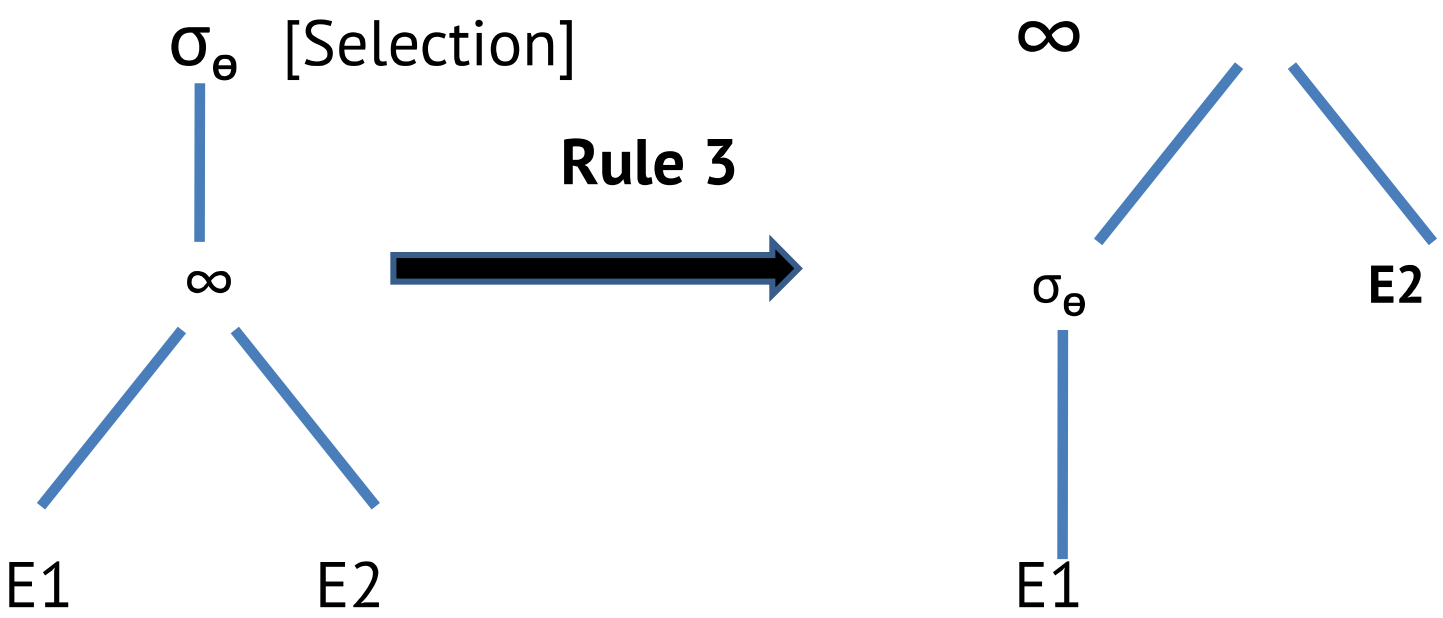
Equivalence Rule : 1

QUERY EVALUATION



Equivalence Rule : 2

QUERY EVALUATION



θ has the attributes only from E1

Equivalence Rule : 3

QUERY EVALUATION

Examples of transformation

We illustrate the use of equivalence rules , we use bank example with the following relation schemas:

Branch-Schema = (Branch_name,Branch_City,assets)

Account_Schema=(account_number,Branch_name,balance)

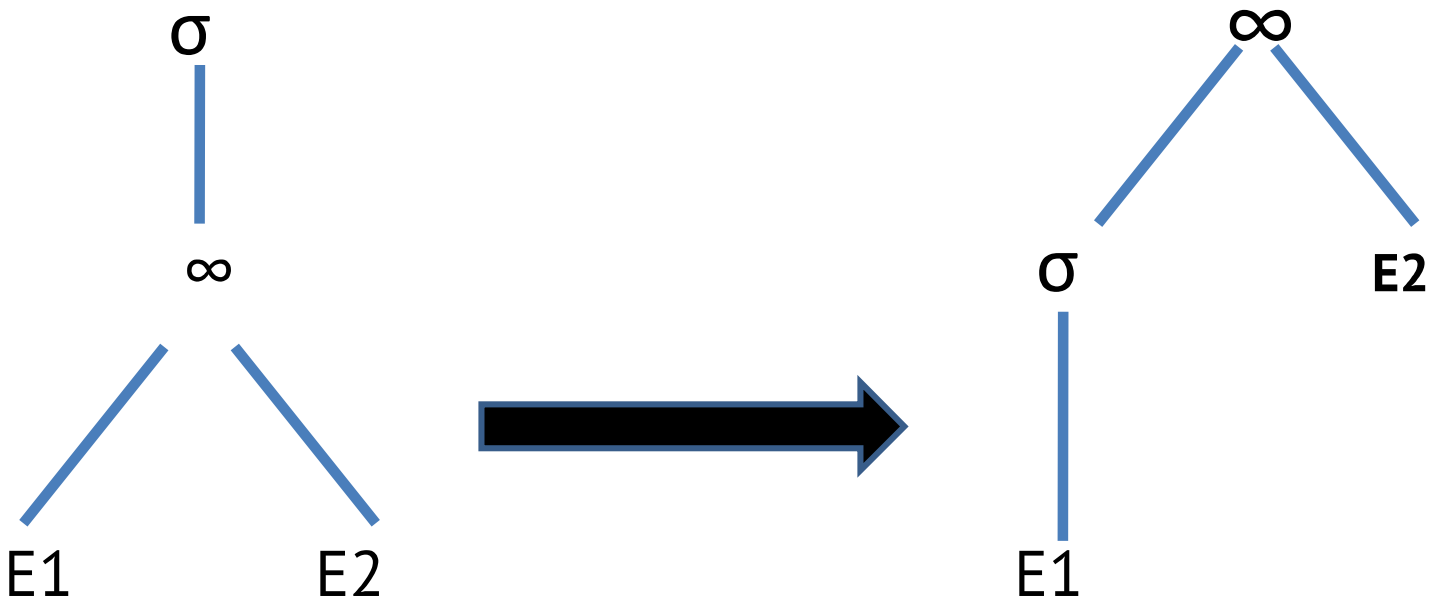
Depositor_Schema=(Customer_Name,account_number)

The relations branch, account and depositor are instances of these schemas. The expression

Π Cname (σ branch_City =”Chennai” ((branch X (account X depositor)))

QUERY EVALUATION

Applying Rule 3



Π Cname ((σ branch_City =”Chennai” (branch)) X (account X depositor))

(Selection / Projection is done first)

Which is equivalent to our original algebraic expression but

QUERY EVALUATION

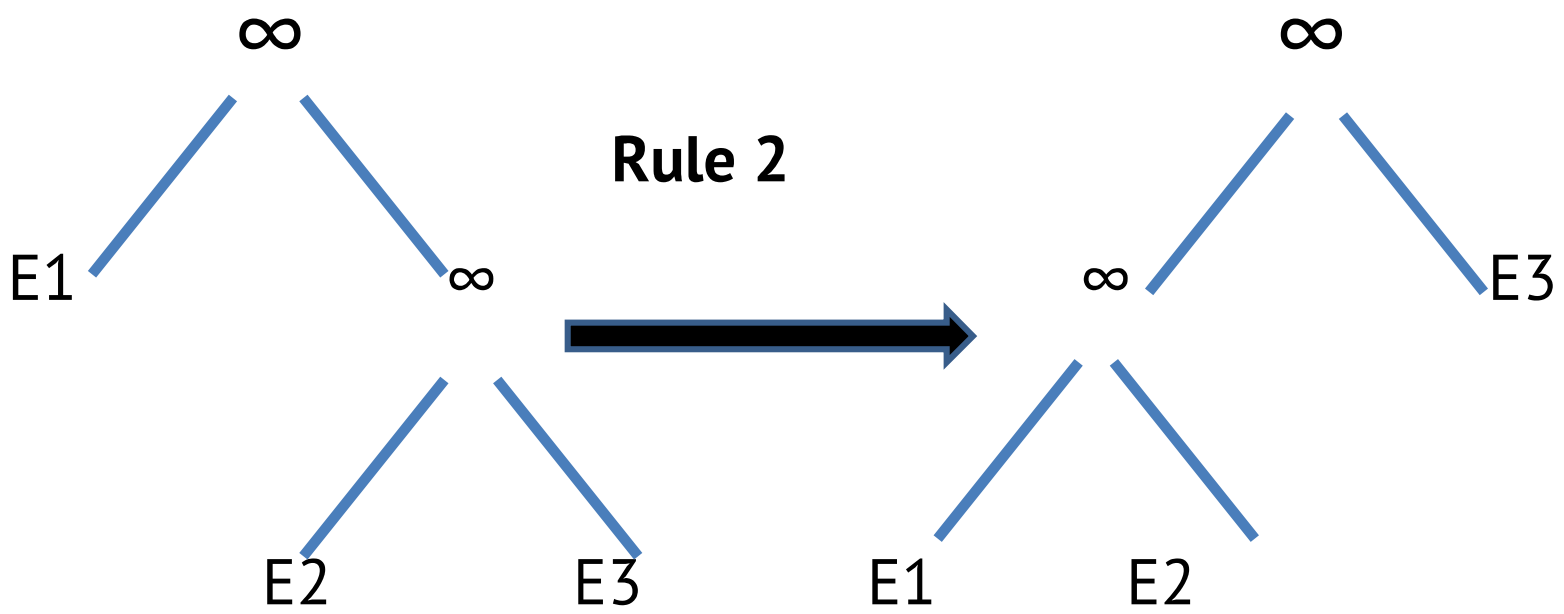
generates smaller intermediate relations

Suppose that we modify our original query to restrict attention to customers who have a balance over Rs10000

The new relational algebra query is

$$\Pi \text{ Cname } (\sigma \text{ branch\_City} = \text{"Chennai "} \wedge \text{ balance} > 10000 \text{ (branch } X \text{ (account } X \text{ depositor)))}$$

QUERY EVALUATION



Applying above rul2 (Rule 2)

branch X (account X depositor) -> (branch X account) X depositor

Π Cname (σ branch_City =”Chennai “^ balance > 10000)

((branch X account) X depositor))

100 100 100

QUERY EVALUATION

Using Rule No 3

$$\Pi_{Cname}\left(\left(\sigma_{branch\_City = \text{"Chennai"} \wedge balance > 10000}\left(branch \times account\right)\right) \times depositor\right)$$

(20 over the selection; 200 over (branch X account); 100 over depositor)

We can break the selection into two selections to get the following sub expression

$$\sigma_{branch\_City = \text{"Chennai"}}(branch) \times \sigma_{balance > 10000}(account)$$

(10 over each selection)

QUERY EVALUATION

Both the proceeding expressions, select tuples with branch_city="Chennai" and balance > 10000. However , the latter form of the expression provides a new opportunity to apply the "Perform Selections Early" rule,

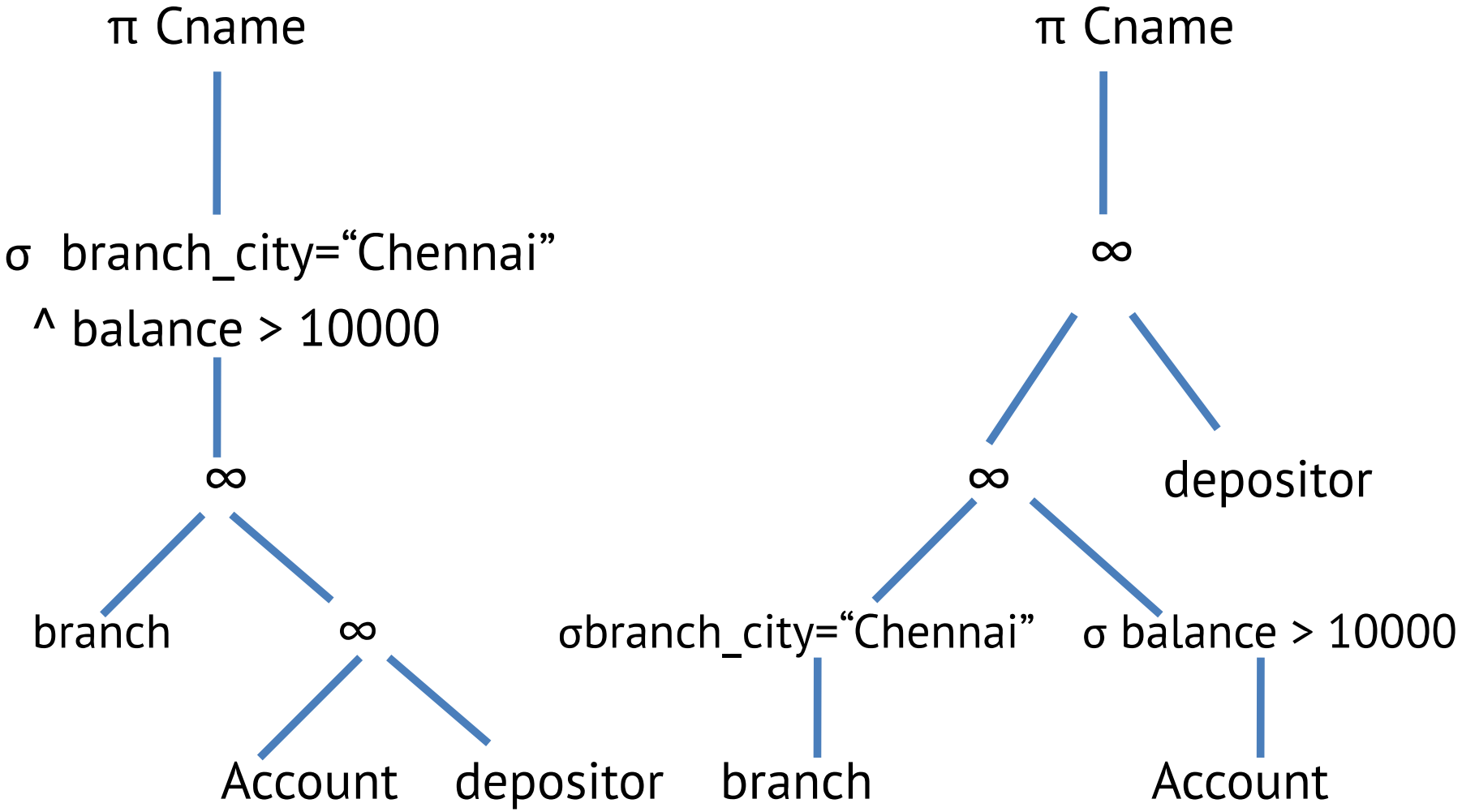
Smaller Intermediate results [20 rows]

10 ↓ 10 X 100

Π Cname ((σ branch_City ="Chennai " (branch)) X (σ balance > 10000 (account) X (depositor)

Figure depicts the initial expression and final expression after all these transformations.

QUERY EVALUATION



(a) Initial Expression Tree

(b) Expression tree after several transactions

QUERY EVALUATION

- Different evaluation plan for a given query can have different costs
- The query having least cost is selected by Optimizer
- Query evaluation plan is generated by Query evaluation plan module
- Once the query plan is chosen, the query is evaluated with that plan by query evaluation engine and the result of the query is the output

QUERY EVALUATION

Measuring the cost of SQL query

In DBMS, the cost involved in executing a query can be measured by considering the number of different resources that are listed below;

- The number of disk accesses / the number of disk block transfers / the size of the table
- Time taken by CPU for executing the query
- The time taken by CPU is negligible in most systems when compared with the number of disk accesses.

If we consider the number of block transfers as the main component in calculating the cost of a query, it would include more sub-components. Those are;

.

QUERY EVALUATION

- Rotational latency – time taken to bring and spin the required data under the read-write head of the disk.
- Seek time – time taken to position the read-write head over the required track or cylinder.
- Sequential I/O – reading data that are stored in contiguous blocks of the disk
- Random I/O – reading data that are stored in different blocks that are not contiguous. That is, the blocks might be stored in different tracks, or different cylinders, etc.
- Whether read/write? – read takes less time, write takes more.

From these sub-components, we would list the components of a more accurate measure as follows;

QUERY EVALUATION

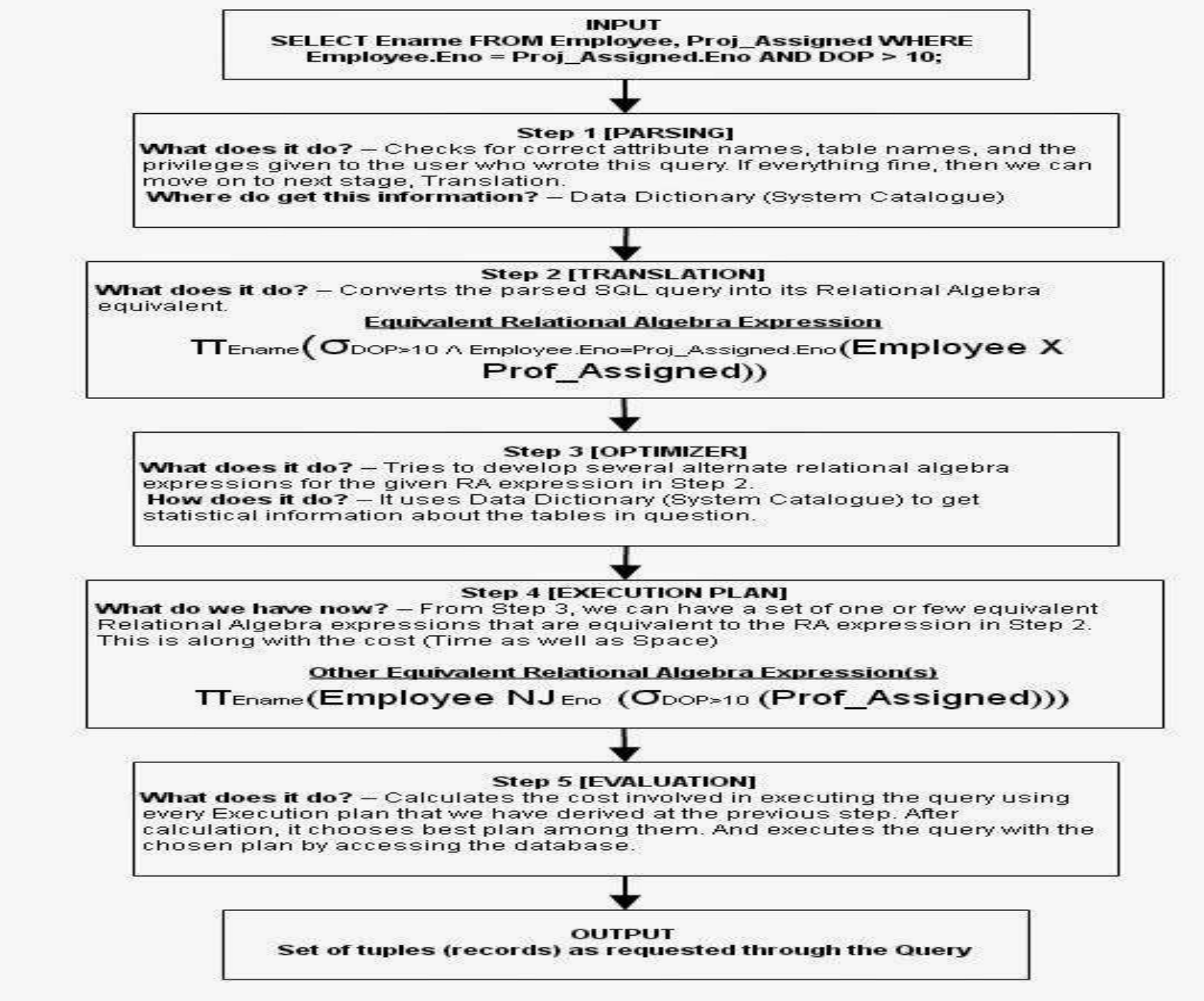
- The number of seek operations performed
- The number of block read
- The number of blocks written

To get the final result, these numbers to be multiplied by the average time required to complete the task. Hence, it can be written as follows;

Query cost = (number of seek operations X average seek time) + (number of blocks read X average transfer time for reading a block) + (number of blocks written X average transfer time for writing a block)

Note: here, CPU cost and few other costs like cost of writing the final result are omitted

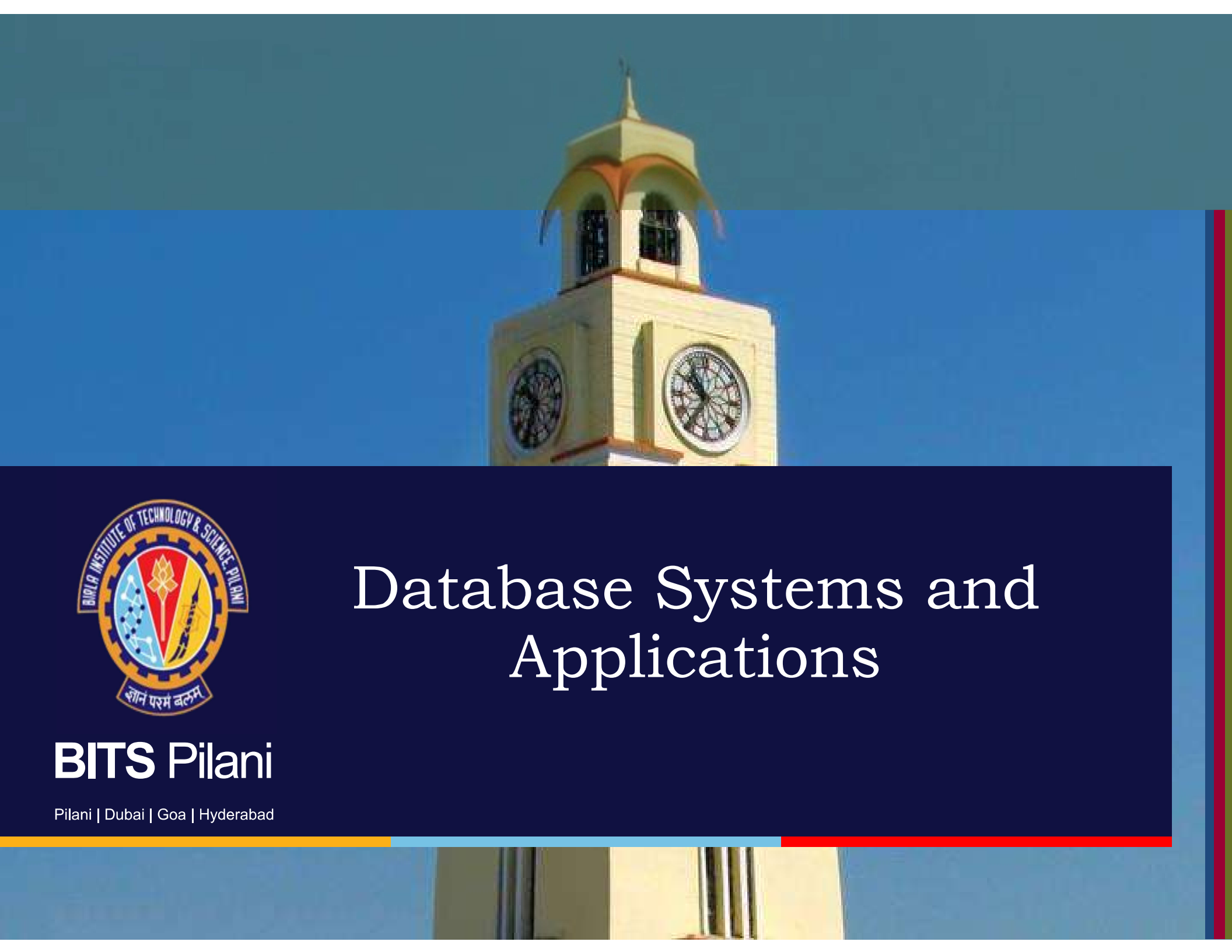
QUERY PROCESSING



Thanks

need of query processing | DBMS - YouTube

query cost measures | DBMS - YouTube



Database Systems and Applications

BITS Pilani

Pilani | Dubai | Goa | Hyderabad

BITS Pilani

Pilani | Dubai | Goa | Hyderabad

Contact Session 16: XML data model

Chapter Outline

Introduction

Structured, Semi structured, and Unstructured Data.

XML Hierarchical (Tree) Data Model.

XML Documents, DTD, and XML Schema.

XML Documents and Databases.

XML Querying.

- XPath
- XQuery

SQL Rules

SQL follows the following rules:

- Structure query language is not case sensitive. Generally, keywords of SQL are written in uppercase.
- Statements of SQL are dependent on text lines. We can use a single SQL statement on one or multiple text line.
- Using the SQL statements, you can perform most of the actions in a database.
- SQL depends on tuple relational calculus and relational algebra.

Introduction

- Although **HTML** is widely used for formatting and structuring *Web documents*, it is not suitable for specifying *structured data* that is extracted from databases.

- A new language—namely **XML** (eXtended Markup Language) has emerged as the standard for structuring and exchanging data over the Web.
  - XML can be used to provide more information about the structure and meaning of the data in the Web pages rather than just specifying how the Web pages are formatted for display on the screen.

- The formatting aspects are specified separately—for example, by using a formatting language such as **XSL** (eXtended Stylesheet Language).

Structured, Semi Structured and Unstructured Data

Three characterizations:

- **Structured** Data
- **Semi-Structured** Data
- **Unstructured** Data

1. Structured Data:

- Information stored in databases is known as **structured** data because it is represented in a strict format.
- The DBMS then checks to ensure that all data follows the structures and constraints specified in the schema.

Structured, Semi Structured and Unstructured Data (contd.)

2. **Semi-Structured** Data:
   - In some applications, data is collected in an ad-hoc manner before it is known how it will be stored and managed.
   - This data may have a certain structure, but not all the information collected will have identical structure. This type of data is known as **semi-structured** data.
   - In semi-structured data, the schema information is mixed in with the data values, since each data object can have different attributes that are not known in advance. Hence, this type of data is sometimes referred to as self-describing data.

Structured, Semi Structured and Unstructured Data (contd.)

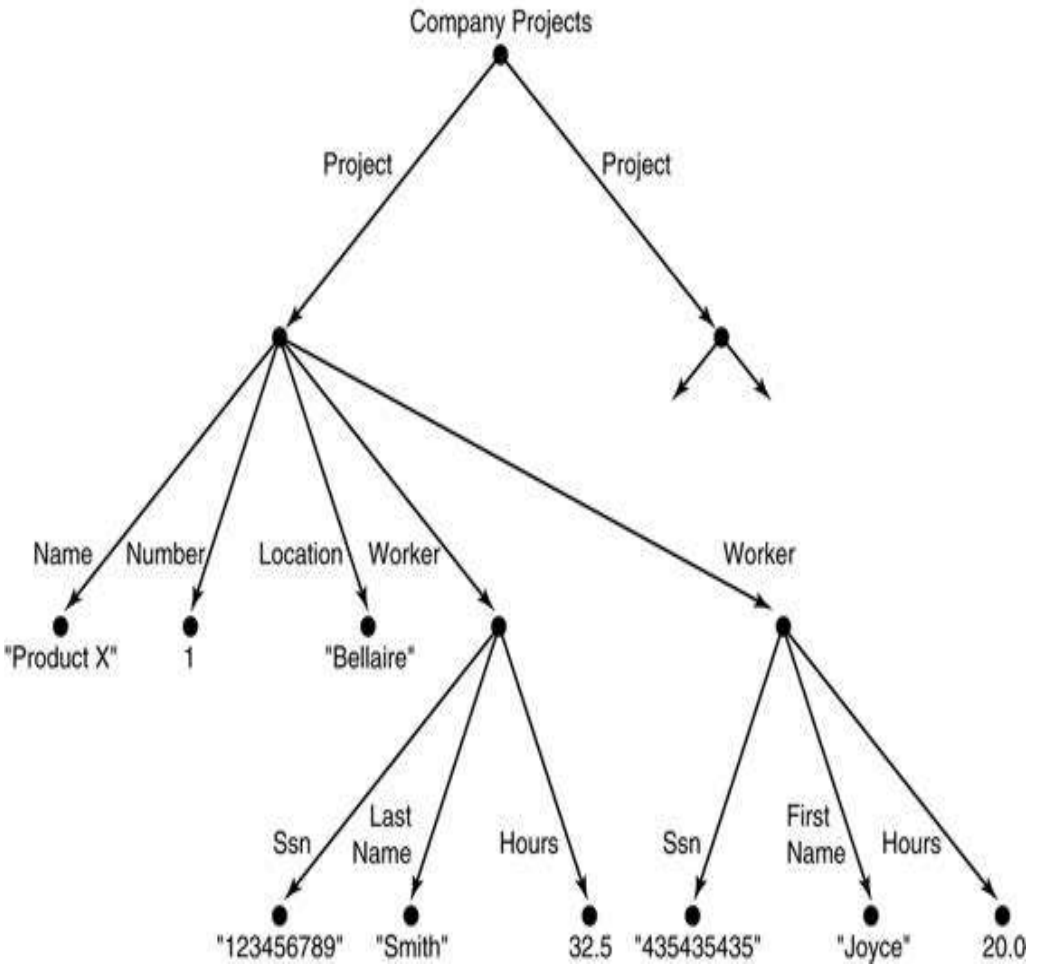
3. Unstructured Data:

- A third category is known as **unstructured** data, because there is very limited indication of the type of data.
- A typical example would be a text document that contains information embedded within it. Web pages in HTML that contain some data are considered as unstructured data.

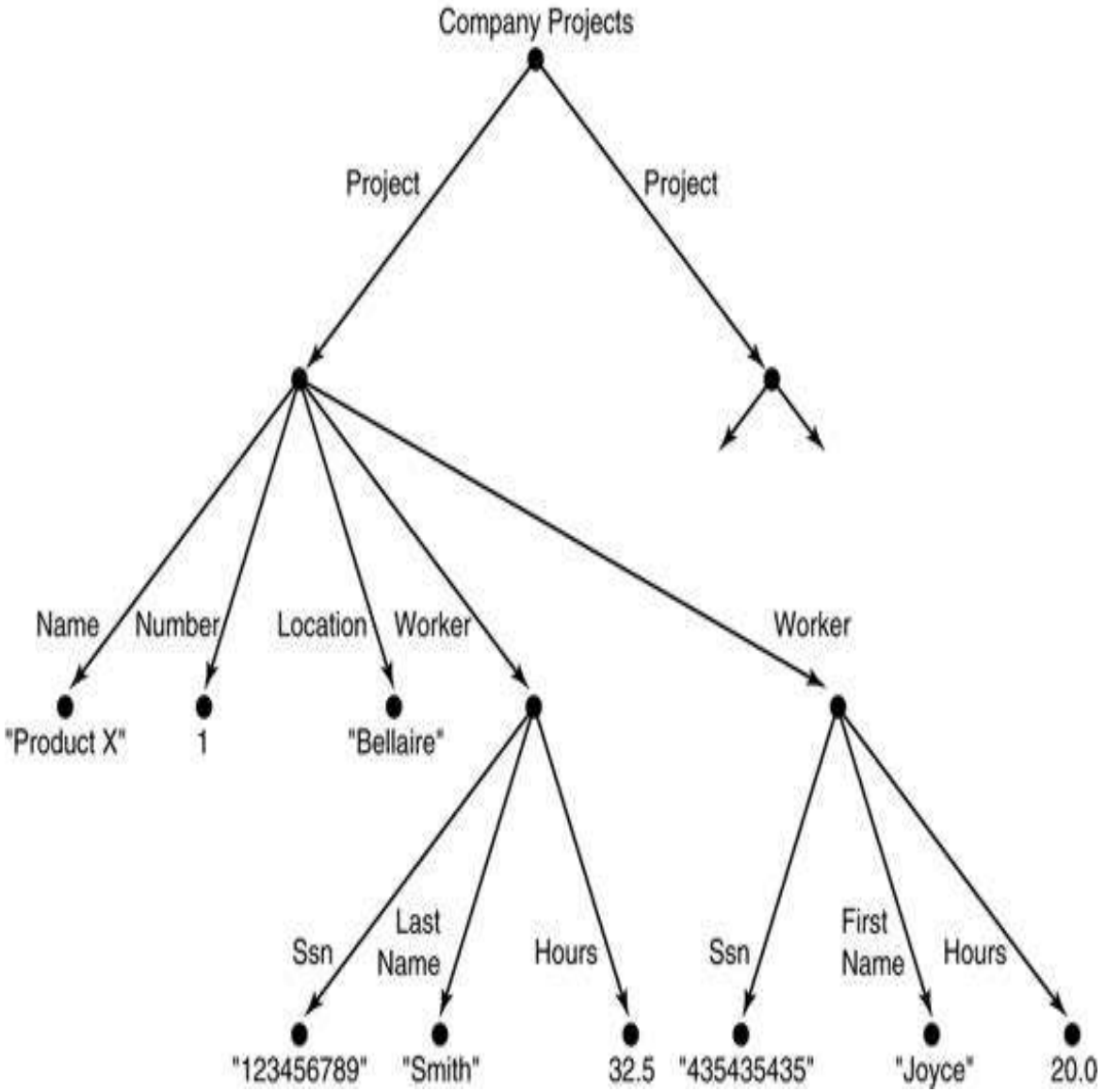
Structured, Semi Structured and Unstructured Data (contd.)

Semi-structured data may be displayed as a directed graph...

- The labels or tags on the directed edges represent the schema names—the names of attributes, object types (or entity types or classes), and relationships.
- The internal nodes represent individual objects or composite attributes.
- The leaf nodes represent actual data values of simple (atomic) attributes.



Representing semistructured data as a graph.




```
<?xml version="1.0" standalone="yes"?>
<projects>

 <project>
  <Name>ProductX</Name>
  <Number>1</Number>
  <Location>Bellaire</Location>
  <DeptNo>5</DeptNo>
  <Worker>
   <SSN>123456789</SSN>
   <LastName>Smith</LastName>
   <hours>32.5</hours>
  </Worker>
  <Worker>
   <SSN>453453453</SSN>
   <FirstName>Joyce</FirstName>
   <hours>20.0</hours>
  </Worker>
 </project>
 </project>
  <Name>ProductY</Name>
  <Number>2</Number>
  <Location>Sugarland</Location>
  <DeptNo >5</DeptNo >
  <Worker>
   <SSN>123456789</SSN>
   <hours>7.5</hours>
  </Worker>
  <Worker>
   <SSN>453453453</SSN>
   <hours>20.0</hours>
  </Worker>
  <Worker>
   <SSN>333445555</SSN>
   <hours>10.0</hours>
  </Worker>
 </project>

...

</projects>
```

XML Hierarchical (Tree) Data Model

XML Hierarchical (Tree) Data Model (contd.)

The basic object is XML in the **XML document**.

There are two main structuring concepts that are used to construct an XML document:

- **Elements**
- **Attributes**

Attributes in XML provide additional information that describe elements.

As in HTML, elements are identified in a document by their **start tag** and **end tag**.

The tag names are enclosed between angled brackets <...>, and end tags are further identified by a backslash </...>.

Complex elements are constructed from other elements hierarchically, whereas **simple** elements contain data values.

It is straightforward to see the correspondence between the XML textual representation and the tree structure.

In the tree representation, internal nodes represent complex elements, whereas leaf nodes represent simple elements.

That is why the XML model is called a **tree** model or a **hierarchical** model.

XML Hierarchical (Tree) Data Model (contd.)

It is possible to characterize three main types of XML documents:

1. Data-centric XML documents
   These documents have many small data items that follow a specific structure, and hence may be extracted from a structured database. They are formatted as XML documents in order to exchange them or display them over the Web.
2. Document-centric XML documents:
   These are documents with large amounts of text, such as news articles or books. There is little or no structured data elements in these documents.
3. Hybrid XML documents:
   These documents may have parts that contains structured data and other parts that are predominantly textual or unstructured.

Two types of XML

Well-Formed XML

Valid XML

Well-Formed XML

It must start with an XML declaration to indicate the version of XML being used—as well as any other relevant attributes.

It must follow the syntactic guidelines of the tree model.

This means that there should be a **single root element**, and every element must include a matching pair of start tag and end tag within the start and end tags of the **parent element**.

A well-formed XML document is **syntactically correct**

This allows it to be processed by generic processors that traverse the document and create an internal tree representation.

DOM (Document Object Model) - Allows programs to manipulate the resulting tree representation corresponding to a well-formed XML document. The whole document must be parsed beforehand when using dom.

SAX - Allows processing of XML documents on the fly by notifying the processing program whenever a start or end tag is encountered.

Valid XML

A stronger criterion is for an XML document to be **valid**.

In this case, the document must be well-formed, and in addition the element names used in the start and end tag pairs must follow the structure specified in a separate XML **DTD (Document Type Definition)** file or XML schema file.

```
<!DOCTYPE projects [
  <!ELEMENT projects (project+)>
  <!ELEMENT project (Name, Number, Location, DeptNo?, Workers)>
  <!ELEMENT Name (#PCDATA)>
  <!ELEMENT Number (#PCDATA)>
  <!ELEMENT Location (#PCDATA)>
  <!ELEMENT DeptNo (#PCDATA)>
  <!ELEMENT Workers (Worker*)>
  <!ELEMENT Worker (SSN, LastName?, FirstName?, hours)>
  <!ELEMENT SSN (#PCDATA)>
  <!ELEMENT LastName (#PCDATA)>
  <!ELEMENT FirstName (#PCDATA)>
  <!ELEMENT hours (#PCDATA)>
] >
```

XML Documents, DTD, and XML Schema (contd.)

XML DTD Notation

- A * following the element name means that the element can be repeated zero or more times in the document. This can be called an optional multivalued (repeating) element.
- A + following the element name means that the element can be repeated one or more times in the document. This can be called a required multivalued (repeating) element.
- A ? following the element name means that the element can be repeated zero or one times. This can be called an optional single-valued (non-repeating) element.
- An element appearing without any of the preceding three symbols must appear exactly once in the document. This can be called an required single-valued (non-repeating) element.

XML Documents, DTD, and XML Schema (contd.)

XML DTD Notation (contd.)

- The type of the element is specified via parentheses following the element.
  - If the parentheses include names of other elements, these would be the children of the element in the tree structure.
  - If the parentheses include the keyword #PCDATA or one of the other data types available in XML DTD, the element is a leaf node. **PCDATA** stands for **parsed character data**, which is roughly similar to a string data type.
- Parentheses can be nested when specifying elements.
- A bar symbol (e1 | e2) specifies that either e1 or e2 can appear in the document.

XML Documents, DTD, and XML Schema (contd.)

Limitations of XML DTD

- First, the data types in DTD are not very general.
- Second, DTD has its own special syntax and so it requires specialized processors.
  - It would be advantageous to specify XML schema documents using the syntax rules of XML itself so that the same processors for XML documents can process XML schema descriptions.
- Third, all DTD elements are always forced to follow the specified ordering of the document so unordered elements are not permitted.

XML Documents, DTD, and XML Schema (contd.)

An XML schema file called company

```
<?xml version="1.0" encoding="UTF-8" ?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:annotation>
      <xsd:documentation xml:lang="en">Company Schema (Element Approach) -
        Prepared by Babak Hojabri</xsd:documentation>
  </xsd:annotation>
<xsd:element name="company">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="department" type="Department" minOccurs="0"
            maxOccurs="unbounded" />
        <xsd:element name="employee" type="Employee" minOccurs="0"
            maxOccurs="unbounded">
          <xsd:unique name="dependentNameUnique">
              <xsd:selector xpath="employeeDependent" />
              <xsd:field xpath="dependentName" />
            </xsd:unique>
        </xsd:element>
        <xsd:element name="project" type="Project" minOccurs="0"
            maxOccurs="unbounded" />
      </xsd:sequence>
    </xsd:complexType>
```

XML Documents, DTD, and XML Schema (contd.)

An XML schema file called company (contd.)

```
<xsd:unique name="departmentNameUnique">
      <xsd:selector xpath="department" />
      <xsd:field xpath="departmentName" />
</xsd:unique>
<xsd:unique name="projectNameUnique">
      <xsd:selector xpath="project" />
      <xsd:field xpath="projectName" />
</xsd:unique>
<xsd:key name="projectNumberKey">
      <xsd:selector xpath="project" />
      <xsd:field xpath="projectNumber" />
</xsd:key>
<xsd:key name="departmentNumberKey">
      <xsd:selector xpath="department" />
      <xsd:field xpath="departmentNumber" />
</xsd:key>
<xsd:key name="employeeSSNKey">
      <xsd:selector xpath="employee" />
      <xsd:field xpath="employeeSSN" />
</xsd:key>
<xsd:keyref name="departmentManagerSSNKeyRef" refer="employeeSSNKey">
      <xsd:selector xpath="department" />
      <xsd:field xpath="departmentManagerSSN" />
</xsd:keyref>
<xsd:keyref name="employeeDepartmentNumberKeyRef"
      refer="departmentNumberKey">
      <xsd:selector xpath="employee" />
      <xsd:field xpath="employeeDepartmentNumber" />
</xsd:keyref>
<xsd:keyref name="employeeSupervisorSSNKeyRef" refer="employeeSSNKey">
      <xsd:selector xpath="employee" />
      <xsd:field xpath="employeeSupervisorSSN" />
</xsd:keyref>
<xsd:keyref name="projectDepartmentNumberKeyRef"
      refer="departmentNumberKey">
      <xsd:selector xpath="project" />
      <xsd:field xpath="projectDepartmentNumber" />
</xsd:keyref>
<xsd:keyref name="projectWorkerSSNKeyRef" refer="employeeSSNKey">
      <xsd:selector xpath="project/projectWorker" />
      <xsd:field xpath="SSN" />
</xsd:keyref>
<xsd:keyref name="employeeWorksOnProjectNumberKeyRef"
      refer="projectNumberKey">
      <xsd:selector xpath="employee/employeeWorksOn" />
      <xsd:field xpath="projectNumber" />
</xsd:keyref>
</xsd:element>
```

XML Documents, DTD, and XML Schema (contd.)

An XML schema file called company (contd.)

```
<xsd:complexType name="Department">
   <xsd:sequence>
      <xsd:element name="departmentName" type="xsd:string" />
      <xsd:element name="departmentNumber" type="xsd:string" />
      <xsd:element name="departmentManagerSSN" type="xsd:string" />
      <xsd:element name="departmentManagerStartDate" type="xsd:date" />
      <xsd:element name="departmentLocation" type="xsd:string"
         minOccurs="0" maxOccurs="unbounded" />
   </xsd:sequence>
</xsd:complexType>
<xsd:complexType name="Employee">
   <xsd:sequence>
      <xsd:element name="employeeName" type="Name" />
      <xsd:element name="employeeSSN" type="xsd:string" />
      <xsd:element name="employeeSex" type="xsd:string" />
      <xsd:element name="employeeSalary" type="xsd:unsignedInt" />
      <xsd:element name="employeeBirthDate" type="xsd:date" />
      <xsd:element name="employeeDepartmentNumber" type="xsd:string" />
      <xsd:element name="employeeSupervisorSSN" type="xsd:string" />
      <xsd:element name="employeeAddress" type="Address" />
      <xsd:element name="employeeWorksOn" type="WorksOn" minOccurs="1"
         maxOccurs="unbounded" />
      <xsd:element name="employeeDependent" type="Dependent" minOccurs="0"
         maxOccurs="unbounded" />
   </xsd:sequence>
</xsd:complexType>
<xsd:complexType name="Project">
   <xsd:sequence>
      <xsd:element name="projectName" type="xsd:string" />
      <xsd:element name="projectNumber" type="xsd:string" />
      <xsd:element name="projectLocation" type="xsd:string" />
      <xsd:element name="projectDepartmentNumber" type="xsd:string" />
      <xsd:element name="projectWorker" type="Worker" minOccurs="1"
         maxOccurs="unbounded" />
   </xsd:sequence>
</xsd:complexType>
<xsd:complexType name="Dependent">
   <xsd:sequence>
      <xsd:element name="dependentName" type="xsd:string" />
      <xsd:element name="dependentSex" type="xsd:string" />
      <xsd:element name="dependentBirthDate" type="xsd:date" />
      <xsd:element name="dependentRelationship" type="xsd:string" />
   </xsd:sequence>
</xsd:complexType>
<xsd:complexType name="Address">
   <xsd:sequence>
      <xsd:element name="number" type="xsd:string" />
      <xsd:element name="street" type="xsd:string" />
      <xsd:element name="city" type="xsd:string" />
      <xsd:element name="state" type="xsd:string" />
   </xsd:sequence>
```

XML Documents, DTD, and XML Schema (contd.)

An XML schema file called company (contd.)

```
    </xsd:complexType>
    <xsd:complexType name="Name">
      <xsd:sequence>
        <xsd:element name="firstName" type="xsd:string" />
        <xsd:element name="middleName" type="xsd:string" />
        <xsd:element name="lastName" type="xsd:string" />
      </xsd:sequence>
    </xsd:complexType>
    <xsd:complexType name="Worker">
      <xsd:sequence>
        <xsd:element name="SSN" type="xsd:string" />
        <xsd:element name="hours" type="xsd:float" />
      </xsd:sequence>
    </xsd:complexType>
    <xsd:complexType name="WorksOn">
      <xsd:sequence>
        <xsd:element name="projectNumber" type="xsd:string" />
        <xsd:element name="hours" type="xsd:float" />
      </xsd:sequence>
    </xsd:complexType>
</xsd:schema>
```

XML Documents, DTD, and XML Schema (contd.)

XML Schema

- **Schema Descriptions and XML Namespaces**
  - It is necessary to identify the specific set of XML schema language elements (tags) by a file stored at a Web site location.
    - The second line in our example specifies the file used in this example, which is: "http://www.w3.org/2001/XMLSchema".
  - Each such definition is called an **XML namespace**.
  - The file name is assigned to the variable xsd using the attribute xmlns (XML namespace), and this variable is used as a prefix to all XML schema tags.
- **Annotations, documentation, and language used:**
  - The xsd:annotation and xsd:documentation are used for providing comments and other descriptions in the XML document.
  - The attribute XML:lang of the xsd:documentation element specifies the language being used. E.g., “en”

XML Documents, DTD, and XML Schema (contd.)

XML Schema (contd.)

- **Elements and types:**
  - We specify the **root** element of our XML schema. In XML schema, the name attribute of the xsd:element tag specifies the element name, which is called company for the root element in our example.
  - The structure of the company root element is a xsd:complexType.
- **First-level elements in the company database:**
  - These elements are named *employee*, *department*, and *project*, and each is specified in an xsd:element tag. If a tag has only attributes and no further sub-elements or data within it, it can be ended with the back slash symbol (/>) and termed **Empty Element**.

XML Documents, DTD, and XML Schema (contd.)

XML Schema (contd.)

- **Specifying element type and minimum and maximum occurrences**:
  - If we specify a type attribute in an xsd:element, this means that the structure of the element will be described separately, typically using the xsd:complexType element. The minOccurs and maxOccurs tags are used for specifying lower and upper bounds on the number of occurrences of an element. The default is exactly one occurrence.
- **Specifying Keys:**
  - For specifying **primary keys**, the tag xsd:key is used.
  - For specifying **foreign keys**, the tag xsd:keyref is used.
    - When specifying a foreign key, the attribute refer of the xsd:keyref tag specifies the referenced primary key whereas the tags xsd:selector and xsd:field specify the referencing element type and foreign key.

XML Documents, DTD, and XML Schema (contd.)

XML Schema (contd.)

- **Specifying the structures of complex elements via complex types:**
  - Complex elements in our example are *Department*, *Employee*, *Project*, and *Dependent*, which use the tag **xsd:complexType**. We specify each of these as a sequence of subelements corresponding to the database attributes of each entity type by using the xsd:sequence and xsd:element tags of XML schema. Each element is given a **name** and **type** via the attributes name and type of xsd:element.
  - We can also specify minOccurs and maxOccurs attributes if we need to change the default of exactly one occurrence. For (optional) database attributes where null is allowed, we need to specify minOccurs = 0, whereas for multivalued database attributes we need to specify maxOccurs = "unbounded" on the corresponding element.

XML Documents, DTD, and XML Schema (contd.)

XML Schema (contd.)

- **Composite (compound) attributes:**
  - Composite attributes from ER Schema are also specified as complex types in the XML schema, as illustrated by the Address, Name, Worker, and WorksOn complex types. These could have been directly embedded within their parent elements.

XML Documents and Databases.

Approaches to Storing XML Documents

- Using a DBMS to store the documents as text:
  - We can use a relational or object DBMS to store whole XML documents as text fields within the DBMS records or objects. This approach can be used if the DBMS has a special module for document processing, and would work for storing schemaless and document-centric XML documents.
- Using a DBMS to store the document contents as data elements:
  - This approach would work for storing a collection of documents that follow a specific XML DTD or XML schema. Since all the documents have the same structure, we can design a relational (or object) database to store the leaf-level data elements within the XML documents.

XML Documents and Databases.

Approaches to Storing XML Documents (contd.)

- Designing a specialized system for storing native XML data:
  - A new type of database system based on the hierarchical (tree) model would be designed and implemented. The system would include specialized indexing and querying techniques, and would work for all types of XML documents.
- Creating or publishing customized XML documents from pre-existing relational databases:
  - Because there are enormous amounts of data already stored in relational databases, parts of these data may need to be formatted as documents for exchanging or displaying over the Web.

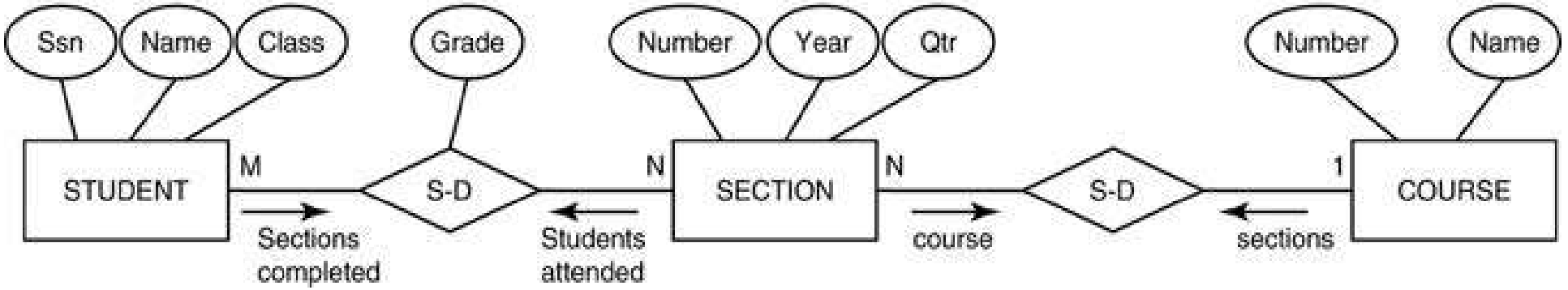
XML Documents, DTD, and XML Schema (contd.)

Extracting XML Documents from Relational Databases.

- Suppose that an application needs to extract XML documents for student, course, and grade information from the university database.
- The data needed for these documents is contained in the database attributes of the entity types ***course***, ***section***, and ***student*** as shown below (part of the main ER), and the relationships s-s and c-s between them.

Subset of the UNIVERSITY database schema

Subset of the UNIVERSITY database schema needed for XML document extraction.



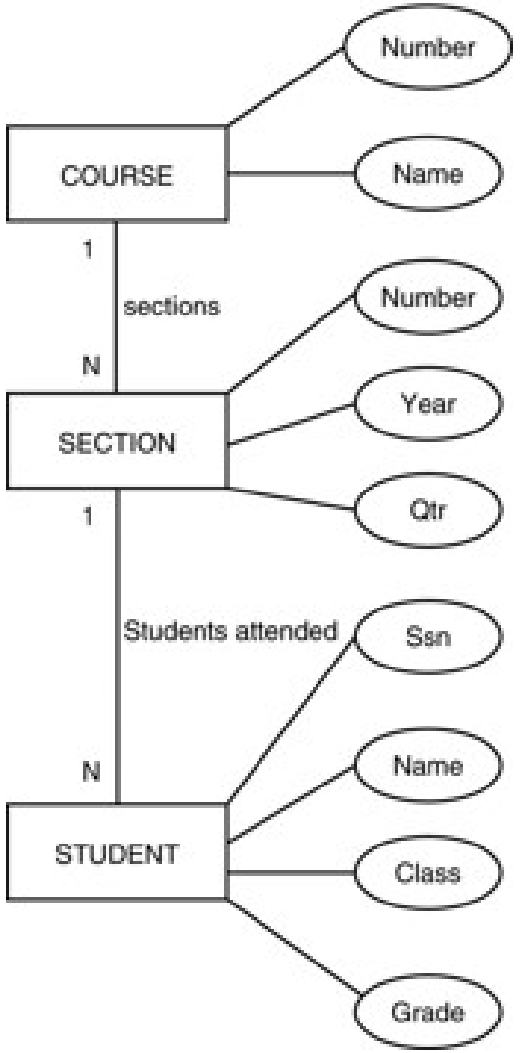
Extracting XML Documents from Relational Databases

One of the possible hierarchies that can be extracted from the database subset could choose COURSE as the root.

Hierarchical (tree) view with COURSE as the root

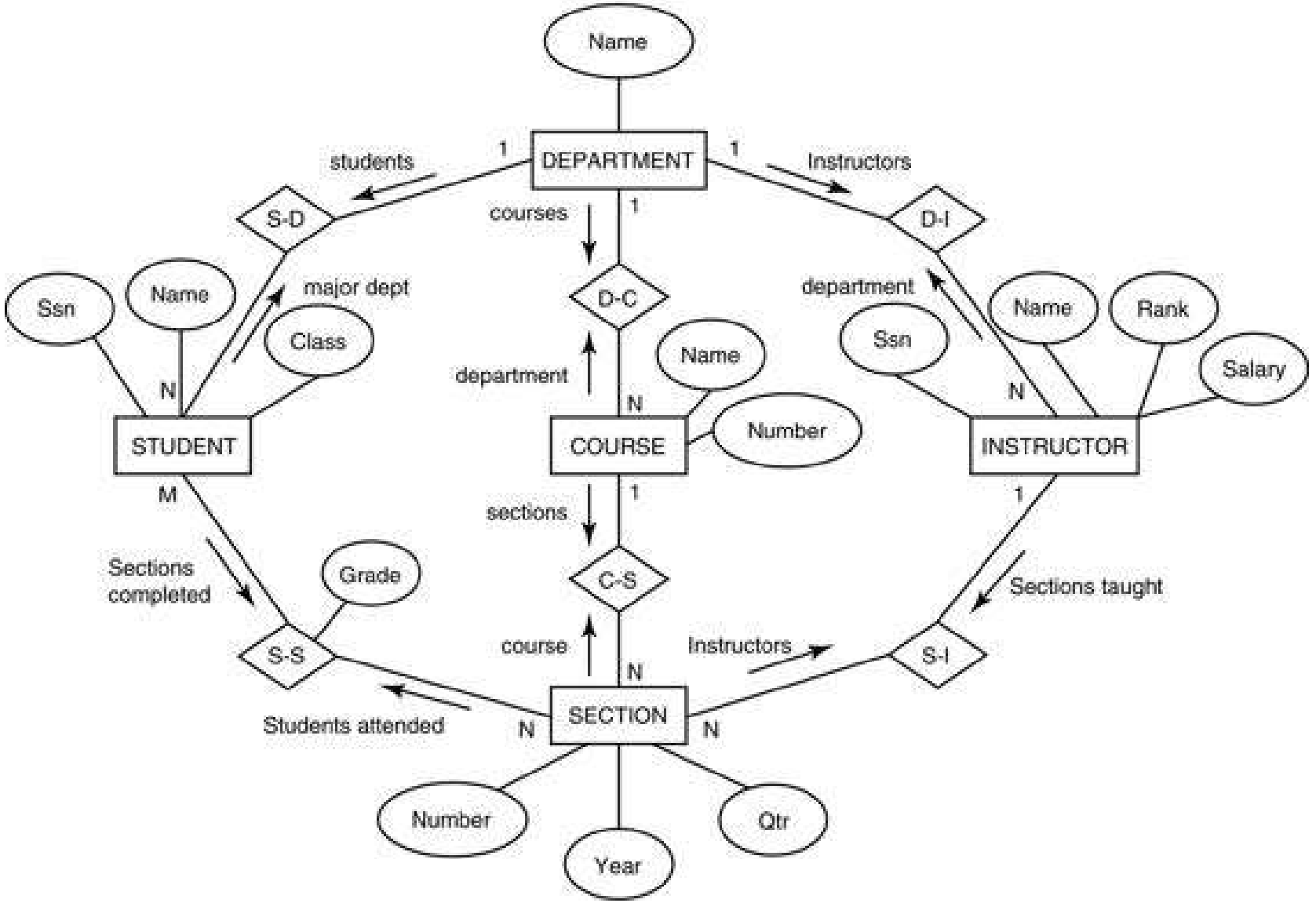
XML schema document with COURSE as the root

```
<xsd:element name="root">
<xsd:sequence>
<xsd:element name="course" minOccurs="0" maxOccurs="unbounded">
  <xsd:sequence>
    <xsd:element name="cname" type="xsd:string" />
    <xsd:element name="cnumber" type="xsd:unsignedInt" />
    <xsd:element name="section" minOccurs="0" maxOccurs="unbounded">
      <xsd:sequence>
        <xsd:element name="secnumber" type="xsd:unsignedInt" />
        <xsd:element name="year" type="xsd:string" />
        <xsd:element name="quarter" type="xsd:string" />
        <xsd:element name="student" minOccurs="0" maxOccurs="unbounded">
          <xsd:sequence>
            <xsd:element name="ssn" type="xsd:string" />
            <xsd:element name="sname" type="xsd:string" />
            <xsd:element name="class" type="xsd:string" />
            <xsd:element name="grade" type="xsd:string" />
          </xsd:sequence>
        </xsd:element>
      </xsd:sequence>
    </xsd:element>
  </xsd:sequence>
</xsd:element>
</xsd:sequence>
</xsd:element>
```



ee) view with COURSE as the root.

An ER schema diagram for a simplified UNIVERSITY database.



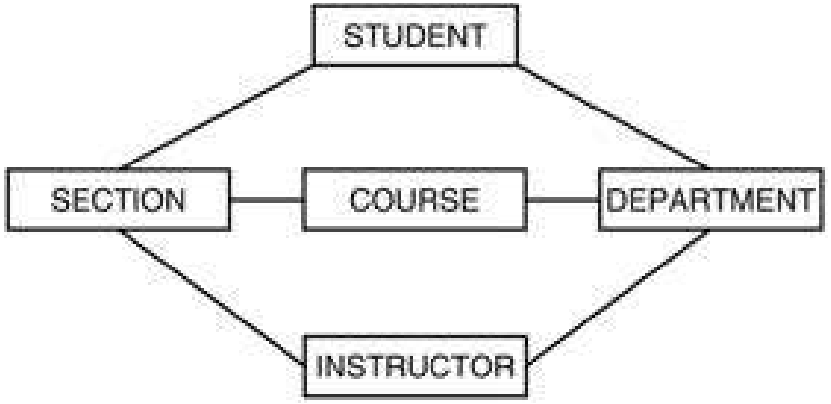
XML Documents, DTD, and XML Schema (contd.)

Breaking Cycles To convert Graphs into Trees

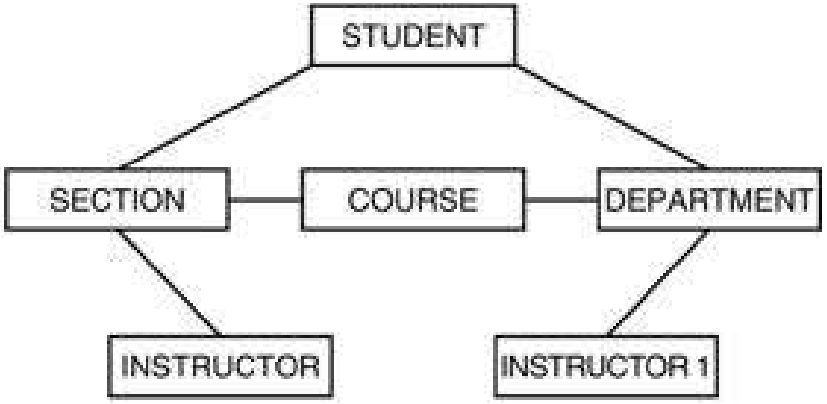
- It is possible to have a more complex subset with one or more cycles, indicating multiple relationships among the entities.
- Suppose that we need the information in all the entity types and relationships in figure below for a particular XML document, with student as the root element.

– One way to break the cycles is to replicate the entity types involved in cycles.
  - First, we replicate INSTRUCTOR as shown in part (2) of Figure, calling the replica to the right INSTRUCTOR1. The INSTRUCTOR replica on the left represents the relationship between instructors and the sections they teach, whereas the INSTRUCTOR1 replica on the right represents the relationship between instructors and the department each works in.
  - We still have the cycle involving COURSE, so we can replicate COURSE in a similar manner, leading to the hierarchy shown in part (3) . The COURSE1 replica to the left represents the relationship between courses and their sections, whereas the COURSE replica to the right represents the relationship between courses and the department that offers each course.

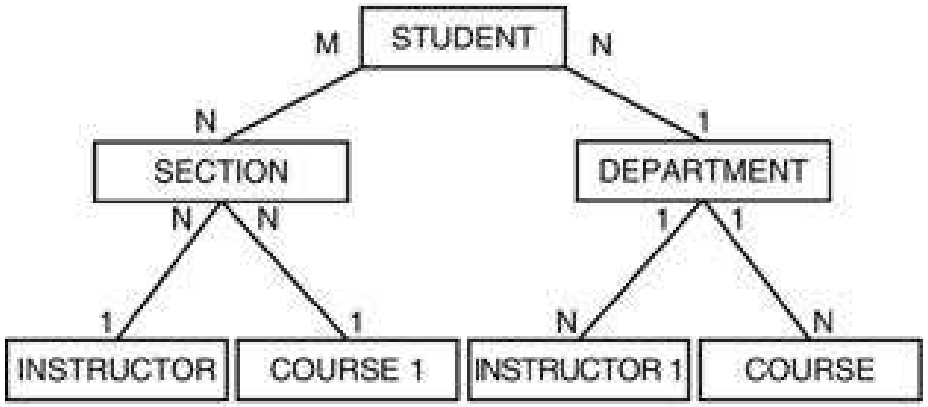
Converting a graph with cycles into a hierarchical (tree) structure



(1)



(2)



(3)

XML Querying

XPath

- An XPath expression returns a collection of element nodes that satisfy certain patterns specified in the expression.
- The names in the XPath expression are node names in the XML document tree that are either tag (element) names or attribute names, possibly with additional **qualifier conditions** to further restrict the nodes that satisfy the pattern.

There are two main separators when specifying a path:
single slash (/) and double slash (//)
A single slash before a tag specifies that the tag must appear as a direct child of the previous (parent) tag, whereas a double slash specifies that the tag can appear as a descendant of the previous tag at any level.

It is customary to include the file name in any XPath query allowing us to specify any local file name or path name that specifies the path.
doc(www.company.com/info.XML)/company => COMPANY XML doc

XML Querying

1. Returns the COMPANY root node and all its descendant nodes, which means that it returns the whole XML document.
2. Returns all department nodes (elements) and their descendant subtrees.
3. Returns all employeeName nodes that are direct children of an employee node, such that the employee node has another child element employeeSalary whose value is greater than 70000.
4. This returns the same result as the previous one except that we specified the full path name in this example.

```
1.  /company
2.  /company/department
3.  //employee [employeeSalary gt 70000]/employeeName
4.  /company/employee [employeeSalary gt 70000]/employeeName
5.  /company/project/projectWorker [hours ge 20.0]
```

Some examples of XPath expressions on XML documents that follow the XML schema file COMPANY

XML Querying

XQuery

- XQuery uses XPath expressions, but has additional constructs.
- XQuery permits the specification of more general queries on one or more XML documents.
- The typical form of a query in XQuery is known as a **FLWR** expression, which stands for the four main clauses of XQuery and has the following form:
  - **FOR** <variable bindings to individual nodes (elements)>
  - **LET** <variable bindings to collections of nodes (elements)>
  - **WHERE** <qualifier conditions>
  - **RETURN** <query result specification>

XML Querying

1. This query retrieves the first and last names of employees who earn more than 70000. The variable $x is bound to each employeeName element that is a child of an employee element, but only for employee elements that satisfy the qualifier that their employeeSalary is greater that 70000.
2. This is an alternative way of retrieving the same elements retrieved by the first query.
3. This query illustrates how a join operation can be performed by having more than one variable. Here, the $x variable is bound to each projectWorker element that is a child of project number 5, whereas the $y variable is bound to each employee element. The join condition matches SSN values in order to retrieve the employee names.

```
1. FOR $x IN
       doc(www.company.com/info.xml)
       //employee [employeeSalary gt 70000]/employeeName
       RETURN <res> $x/firstName, $x/lastName </res>
2. FOR $x IN
       doc(www.company.com/info.xml)/company/employee
       WHERE $x/employeeSalary gt 70000
       RETURN <res> $x/employeeName/firstName,
              $x/employeeName/lastName </res>
3. FOR $x IN
       doc(www.company.com/info.xml)/company
              /project[projectNumber = 5]/projectWorker,
          $y IN
       doc(www.company.com/info.xml)/company/employee
       WHERE $x/hours gt 20.0 AND $y.ssn = $x.ssn
       RETURN <res> $y/employeeName/firstName,
              $y/employeeName/lastName, $x/hours </res>
```