Lecture 12

Database Management Systems

Winter 2004

CMPUT 391: XML and Querying XML

Dr. Osmar R. Zaïane

University of Alberta

Chapter 17 of Textbook

Overview

- Semi-Structured Data
- Introduction to XML
- Querying XML Documents

The Structure of Data

- In the real world data can be of any type and not necessarily following any organized format or sequence.
- Such data is said to be unstructured. Unstructured data is chaotic because it doesn’t follow any rule and is not predictable.
- Text data is usually unstructured. Many data on the Internet is unstructured (video streams, sound streams, images, etc).

Structured Data

- For applications manipulating data, the structure of data is very important to insure efficiency and effectiveness.
- The data is structured when:
  - Data is organized in semantic chunks (entities).
  - Similar entities are grouped together (relations or classes).
  - Entities in a same group have the same descriptions (attributes).
  - Entity descriptions for all entities in a group have the same defined format, a predefined length, are all present, and follow the same order (schema).
- This structure is sometimes too rigid for some applications.
- For many application, data is neither completely unstructured nor completely structured.

Semi-Structured Data

- Data is organized in semantic entities
- Similar entities are grouped together
- But
  - Entities in the same group may not have the same attributes
  - The presence of some attributes may not always be required
  - The size of same attributes of entities in a same group may not be the same
  - The type of the same attributes of entities in a same group may not be of the same type.

Semi-Structured Data (Cont.)

- To make it suitable for machine processing it should have these characteristics
  - Be ***object-like***
  - Be ***schemaless*** (doesn't guarantee to conform exactly to any schema, but different objects have some commonality among themselves)
  - Be ***self-describing*** (some schema-like information, like attribute names, is part of data itself)

Non-Self-Describing Data

Relational or Object-Oriented:

Data part:

```
(#123,  [“Students”,   {[“John”, 111111111, [123,”Main St”]],
                        [“Joe”, 222222222, [321, “Pine St”]] }
        ] )
```

Schema part:

```
PersonList[ ListName: String,
           Contents: [ Name: String,
                       Id: String,
                       Address: [Number: Integer,  Street: String]  ]
          ]
```

Self-Describing Data

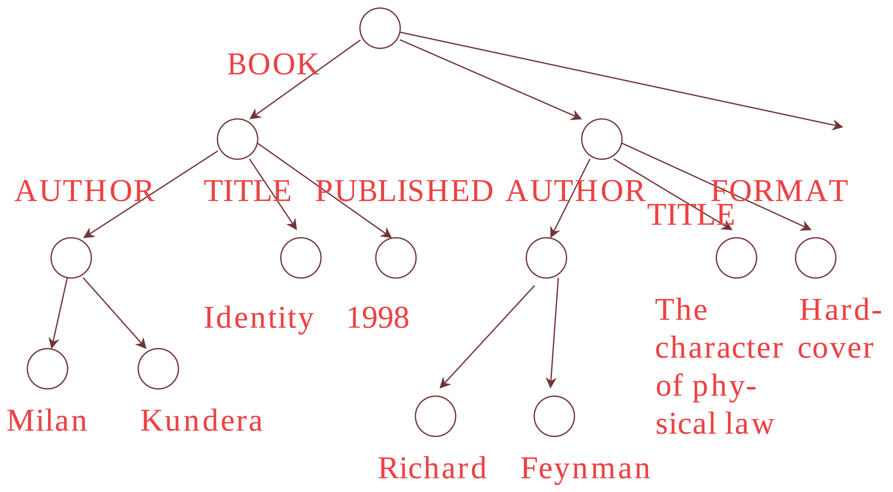
- Attribute names embedded in the data itself
- Doesn't need schema to figure out what is what
- (but schema might be useful nonetheless)

```
(#12345,
   [ListName: “Students”,
    Contents:  { [ Name: “John Doe”,
                   Id:  “111111111”,
                   Address: [Number: 123, Street: “Main St.”] ] ,
                 [Name: “Joe Public”,
                  Id:  “222222222”,
                  Address: [Number: 321, Street: “Pine St.”] ]  }
] )
```

Data Model for Semi-Structured Data

- Semi-structured data doesn't have a schema.
- There are many data models to represent semi-structured data. Most of them use the notion of labeled graphs.
  - Nodes in the graph correspond to compound objects or atomic values.
  - Edges in the graph correspond to attributes
  - The graph is self describing (no need for a schema)
  - Object Exchange Model (OEM): each object is described by a triplet <label, type, value>
  - Complex objects are decomposed hierarchically into smaller objects

Example: Booklist Data in OEM



Overview

- Semi-Structured Data
- Introduction to XML
- Querying XML Documents

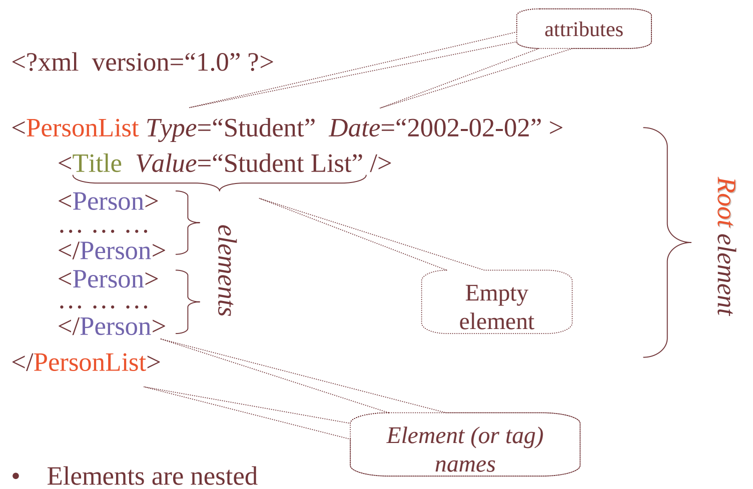
Introduction to XML

- XML: eXtensible Markup Language
- Suitable for semistructured data
  - Easy to describe object-like data
  - Selfdescribing
  - Doesn't require a schema (but can be provided optionally)
- Standard of the World-Wide Web Consortium for data exchange
- All major database products have been extended to store and construct XML documents

What is Special with XML

- It is a language to markup data
- There are no predefined tags like in HTML
- Extensible ➔ tags can be defined and extended based on applications and needs
  - Elements / Tags
  - Attributes
  - (*Eg*.: <BOOK page="453">…</BOOK>)

Example

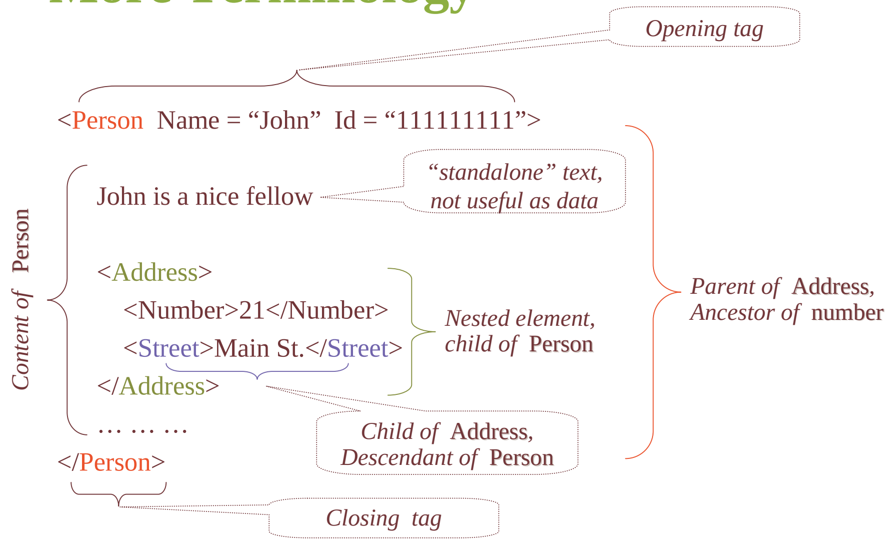


```
<?xml version="1.0" ?>

<PersonList Type="Student" Date="2002-02-02" >
    <Title Value="Student List" />
    <Person>
    … … …
    </Person>
    <Person>
    … … …
    </Person>
</PersonList>
```

- Elements are nested
- Root element contains all others

More Terminology



```
<Person Name = "John" Id = "111111111">
    John is a nice fellow

    <Address>
        <Number>21</Number>
        <Street>Main St.</Street>
    </Address>
    … … …
</Person>
```

Opening tag
"standalone" text, not useful as data
Content of Person
Nested element, child of Person
Parent of Address, Ancestor of number
Child of Address, Descendant of Person
Closing tag

Rules for Creating XML Documents

- **Rule 1**: All terminating tags shall be closed
  - Omitting a closing XML tag is an error.
    Example: **<FirstName>**Joerg**</FirstName>**
- **Rule 2**: All non-terminating tags shall be closed
  - Omitting a forward slash for non-terminating (empty) tags is an error.
    Example **<Available** answer="yes"**/>**
- **Rule 3**: XML shall be case sensitive
  - Using the wrong case is an error.
    Example: **<FirstName>**Osmar**</firstname>**
  - It is OK in HTML <H1>my header</h1>

Rules for Creating XML Documents

- **Rule 4**: An XML document shall have one root
  - Attempting to create more than one root element would generate a syntax error
- **Rule 5**: Terminating tags shall be properly nested
  - Closing a parent tag before closing a child's tag is an error. Example **<Author><name>**Osmar**</Author></name>**
  - It is OK in HTML <b><I>bold italic text</b></I>
- **Rule 6**: Attribute values shall be quoted
  - Omitting quotes, either single or double, around and XML attribute's value is an error. Example **<Product ID="123">**

What is needed?

- XML needs to be parsed to check whether the documents are well formed
- XML needs to be printed
- XML needs to be interpreted for information exchange or populating database
- XML needs to be queried efficiently

Parsers

Representations

XSL/XSLT

SOAP

Query Languages

XML security

Introduction to DTDs

- DTD stands for Document Type Definition
- A DTD is a set of rules that specify how to use an XML markup. It contains specifications for each element, the attributes of the elements, and the values the attributes can take.
- A DTD also specifies how elements are contained in each other
- A DTD ensures that XML documents created by different programs are consistent

Example1: Business Letter

```
<?xml version = "1.0"?>
<LETTER> <Urgency level="1"/>
 <contact type = "from">
              <name>John Doe</name>
              <address>123 Main St.</address>
             <city>Anytown</city>
             <province>Somewhere</province>
             <postalcode>A1B 2C3</postalcode>
</contact>
<contact type = "to">
             <name>Joe Schmoe</name>
             <address>123 Any Ave.</address>
              <city>Othertown</city>
             <province>Otherplace</province>
             <postalcode>Z9Y 8X7</postalcode>
</contact>
<paragraph>Dear Sir,</paragraph>
<paragraph>It is our privilege to inform you about our new  database managed with XML.
This new system will allow you to reduce the load of your inventory list server by  having the
client machine perform the work of sorting  and filtering the data.</paragraph>
<paragraph>Sincerely, Mr. Doe</paragraph>
</LETTER>
```

DTD Example for business letter

DTD Header

Unicode Transformation 8 bits

+ means one or more

Empty means no end tag

? means optional

CDATA means string #IMPLIED means that the attribute value can be unspecified.

#PCDATA is parsed character data, it means that the element contains text

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE LETTER [
<!ELEMENT LETTER (Urgency, contact+, paragraph+)>
<!ELEMENT Urgency (EMPTY)>
<!ATTLIST Urgency level CDATA #IMPLIED>
<!ELEMENT contact (name, address, city, province,  postalcode,
phone?, email?)>
<!ATTLIST contact type CDATA #REQUIRED>
<!ELEMENT name (#PCDATA)>
<!ELEMENT address (#PCDATA)>
…
]>
```

DTD Rules

<!ELEMENT elementName (components or content type)>

Examples:

<!ELEMENT name (#PCDATA)>
name is an element/tag for text data

<!ELEMENT Urgency (EMPTY)>
Urgency has no content

<!ELEMENT LETTER (Urgency, contact+, paragraph+)>
letter is an element that contains an Urgency element followed by one or more contact elements and one or more paragraph elements

Multiple Elements

<!ELEMENT LETTER (Urgency, contact+, paragraph+)>
<!ELEMENT contact (name, address, city, province, postalcode, phone?, email?)>

Are called multiple elements (lists of elements). They require the rule to specify their sequence and the number of times they can occur.

|	Any element may occur
,	Occur in specified sequence
?	Optional, may occur 0 or once
+	Occurs at least once (1 or many)
*	Occurs many times (0 or many)

Attributes in DTD

<!ATTLIST elementName attributeName Type Specification>

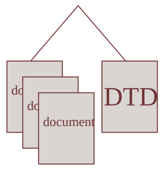
- elementName and attributeName associate the attribute with the element
- The Type specifies if the attribute is free text (CDATA) or a list of predefined values (value1 | value2 | value3)
- Example:

<!ATTLIST Urgency level CDATA #IMPLIED>
<!ATTLIST contact type CDATA #REQUIRED>
<!ATTLIST P align (center | right | left) #IMPLIED>

- Specification could be:

• #REQUIRED	attribute must be specified
• #IMPLIED	attributes can be unspecified
• #FIXED	attribute is preset to a specific value
• "defaultvalue"	default value for the attribute

Calling an External DTD



- A DTD can be referenced from XML documents
  - <!DOCTYPE LETTER SYSTEM "letter.dtd">
  - Any element, attribute not explicitly defined in the DTD generates an error in the XML document.
  - XML document conforming to a DTD is called valid and well-formed.
  - keyword SYSTEM/PUBLIC: intended for private/public use
- DTDs ensure consistency between XML documents
- Defining a DTD is equivalent to creating a customized markup language.
- There are many domain specific markup languages: MML (Mathematical Markup Language), CML (Chemical Markup Language),…many other XML-based languages

Beyond DTDs: XML Schema

- DTD are limited
  - very limited data types (just strings)
  - can’t express strong consistency constraints
  - can’t express unordered contents conveniently
  - all element names are global
    - can’t have one Name type for people and another for companies:
      - <!ELEMENT Name (Last, First)>
      - <!ELEMENT Name (#PCDATA)>
    - both can’t be in the same DTD
- XML Schema solves some of the problems with DTDs, but is much more complex than DTDs

Overview

- Semi-Structured Data
- Introduction to XML
- Querying XML Documents

XML Query Languages

- Problems
  - how will data be extracted from large documents?
  - how will XML data be exchanged, e.g., by shipping XML documents or by shipping queries?
  - how will XML data be exchanged between user communities using different DTDs?
  - how will XML data from multiple XML sources be integrated?
- Solution: An *XML Query Language* that allows to
  - extract pieces of data from XML documents
  - map XML data between DTDs (Schemas)
  - integrate XML data from different sources

XQuery

- W3C standard query language for XML
- SQL-like FLWR Expressions
  - FOR (LET)
  - WHERE
  - RETURN
- Integrates XPath for path expressions

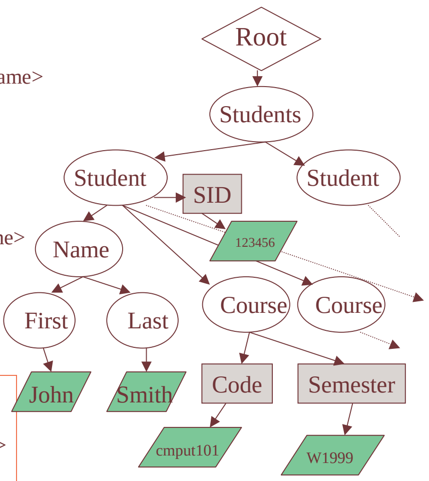
XPath

- Core query language
  - Simple selection operator for paths from a XML document-tree
  - Xpath expressions take a document tree and return a set of nodes in the tree
  - Used in XQuery and many other XML standards

Example: XPath & XML Document Tree

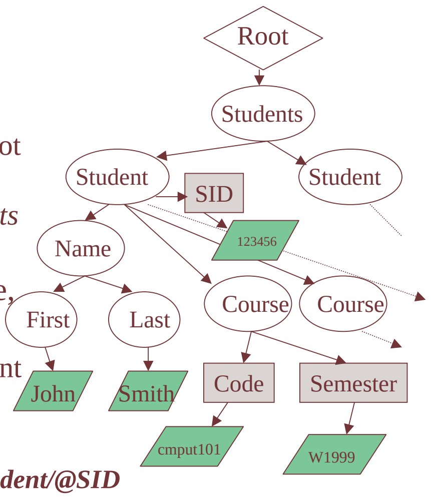
```
<? Xml version="1.0" ?>
<Students>
   <Student SID="123456">
      <Name><First>John</First><Last>Smith</Last></Name>
      <Status>Full-UndG</Status>
      <Course Code="cmput101"  Semester="W1999" />
      <Course Code="cmput291"  Semester="F1999" />
      <Course Code="cmput391"  Semester="F2000" />
   </Student>
 <Student SID="678123">
      <Name><First>Jane</First><Last>Doe</Last></Name>
      <Status>Full-UndG</Status>
      <Course Code="cmput114"  Semester="F1999" />
      <Course Code="cmput304"  Semester="F2000" />
</Student>
</Students>
```

```
<!DOCTYPE students [
   <!ELEMENT Students (Student*)>
   <!ELEMENT Student (Name, Status, Course*)>
   <!ELEMENT Name (First, Last)>
   <!ELEMENT First (#PCDATA)>
…]>
```



XPath Expressions

- Absolute path expressions:
  /Students/Student/Name
  refers to the composite *Name*
  //Name
  refers to *Name* descendent of the root
  /Students//First
  refers to descendent *First* of *Students*
- Relative path expressions:
  if current node corresponds to Name,
  ./First is first name of current
  ../Course a course of the current student
  ../..//First is the first name of siblings,
  (// denotes arbitrary sub-path)
- @ is used for attributes: ***/Students/Student/@SID***
- ***/Students/Student[1]/Course[3]*** for specific nodes



XPath with Selection Conditions

- Select all student who took a course in F2000

//Student[Course/@Semester="F2000"]

- Select undergrad students

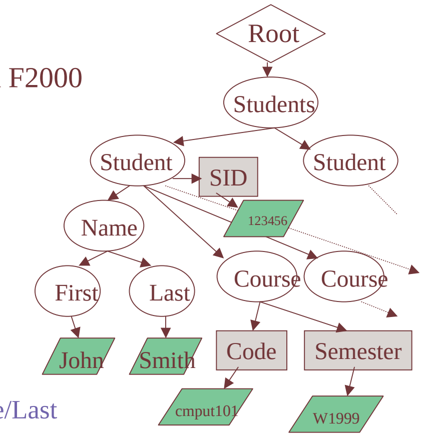
//Student[Status="UndG"]

- Select last names of students who took cmput391

//Student[Course/@Code="cmput391"]/Name/Last

- Select all students who took cmput391 in F2001

//Student[Course/@Code="cmput391" AND   Course/@Semester="F2001"]



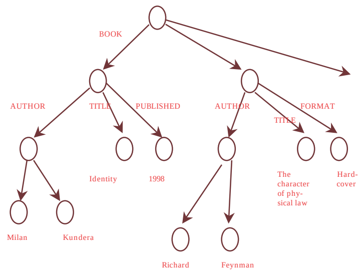
Example

Consider a set of XML documents defined by the following DTD

```
<!DOCTYPE BOOKLIST> [
<!ELEMENT BOOLIST (BOOK)*>
   <!ELEMENT BOOK ( AUTHOR+, TITLE, PUBLISHED?)>
     <!ELEMENT AUTHOR ( FIRSTNAME,LASTNAME)>
               <!ELEMENT FIRSTNAME(#PCDATA)>
               <!ELEMENT LASTNAME(#PCDATA)>
     <!ELEMENT TITLE (#PCDATA)>
     <!ELEMENT PUBLISHED (#PCDATA) >
   <!ATTLIST BOOK GENRE(Science|Fiction) #REQURIED>
   <!ATTLIST BOOK FORMAT (Paperback|Hardcover) “Paperback”>
]>
```

Example (Cont.)

```
<?XML VERSION=“1.0” ENCODING=“UTF-8” STANDALONE=“YES”>
<BOOKLIST>
    <BOOK GENRE=“Science” FORMAT= “Hardcover”>
        <AUTHOR> <FIRSTNAME> Richard</FIRSTNAME>
                 <LASTNAME>Feynman</LASTNAME>
        </AUTHOR>
        <TITLE> The Character of Physical Law</TITLE>
        <PUBLISHED> 1980</PUBLISHE>
    </BOOK>
    <BOOK GENRE=“Fiction”>
        <AUTHOR> <FIRSTNAME>R.K.</FIRSTNAME>
                 <LASTNAME>Narayan</LASTNAME>
        </AUTHOR>
        <TITLE> Waiting for the Mahatma</TITLE>
        <PUBLISHED> 1981</PUBLISHE>
    </BOOK>
    <BOOK GENRE=“Fiction”>
        <AUTHOR> <FIRSTNAME>R.K.</FIRSTNAME>
                 <LASTNAME>Narayan</LASTNAME>
        </AUTHOR>
        <TITLE> The English Teacher</TITLE>
        <PUBLISHED> 1980</PUBLISHE>
    </BOOK>
</BOOKLIST>
```



XQuery Basics

- General structure:
  - [FOR | LET] variable declarations
  - WHERE   condition
  - RETURN   document
  - variable declaration:
    "$" VarName "in" Expression
- Variable binding
  - FOR binds a variable to each element satisfying the expression
  - LET binds a variable to the whole collection of elements that satisfy the expression

Example – FOR clause

- Assume the previous document is stored at www.outbookstore.com/books.xml
- QUERY: Find the last names of all authors

```
FOR
  $l IN doc(www.ourbookstore.com/books.cml)//AUTHOR/LASTNAME
RETURN
 <RESULT> $l </RESULT>
```

ANSWER

```
  <RESULT><LASTNAME> Feynman</LASTNAME></RESULT>
  <RESULT><LASTNAME> Narayan</LASTNAME></RESULT>
```

Example – LET clause

```
LET
  $l IN doc(www.ourbookstore.com/books.cml)//AUTHOR/LASTNAME
RETURN
 <RESULT> $l </RESULT>
```

ANSWER

```
 <RESULT>
   <LASTNAME> Feynman</LASTNAME>
   <LASTNAME> Narayan</LASTNAME>
 </RESULT>
```

Example: WHERE clause

```
FOR
  $1 IN doc(www.ourbookstore.com/books.cml)/BOOKLIST/BOOK
WHERE $1/PUBLISHED = "1980"
RETURN
   <RESULT> $1/AUTHOUR/FIRSTNAME, $1/AUTHOR/LASTNAME
   </RESULT>
```

ANSWER

```
<RESULT>
  <FIRSTNAME> Richard <.FIRSTNAME>
  <LASTNAME>Feynman</LASTNAME>
</RESULT>
<RESULT>
  <FIRSTNAME> R.K.<.FIRSTNAME>
  <LASTNAME>Narayan</LASTNAME>
</RESULT>
```

Example: Nested Queries & Grouping

```
FOR $l IN DISTINCT
    doc(www.ourbookstore.com/books.cml)/BOOKLIST/BOOK/PUBLISHED
RETURN
<RESULT>
  $1,
  FOR $a IN DISTINCT doc(www.ourbookstore.com/books.cml)
            /BOOKLIST/BOOK[PUBLISHED=$1]/AUTHOUR/LASTNAME
  RETURN $a
</RESULT>
```

ANSWER

```
<RESULT>
  <PUBLISHED> 1980 </PUBLISHED>
  <LASTTNAME> Feynman<LASTNAME>
  <LASTNAME>Narayan</LASTNAME>
</RESULT>
<RESULT>
  <PUBLISHED> 1981<PUBLISHED>
  <LASTNAME>Narayan</LASTNAME>
</RESULT>
```

Example: Join & Aggregation

```
FOR $a IN DISTINCT
    doc(www.ourbookstore.com/books.cml)/BOOKLIST/BOOK/AUTHOR
LET $t IN
    doc(www.ourbookstore.com/books.cml)//BOOK/[AUTHOR=$a]/TITLE
RETURN
<RESULT>
        $a/LASTNAME, <TotalBooks> count(distinct($t)) </TotalBooks>
</RESULT>
SORT BY (LASTNAME descending)
```

ANSWER (e.g.)

```
<RESULT>
  <LASTTNAME>Narayan<LASTNAME>
  <TotalBooks> 5 </TotalBooks>
</RESULT>
<RESULT>
  <LASTNAME>Feynman</LASTNAME>
  <TotalBooks> 2 </TotalBooks>
</RESULT>
```

How to store and retrieve XML Data?

- Storing XML data in the file system
- Storing XML in BLOB/CLOB
- Native XML databases
- XML enabled databases

Open Research Questions

- Query Optimization
- Indexing XML Data
  - –Value Index (e.g. B$^+$-tree)
  - –Structure Index (Path indexing)