BLG 317E
DATABASE SYSTEMS

WEEK 12

NoSQL (Part 2)

Main Categories of NoSQL Databases

- The followings are main categories.
  - **Key-value Pair Based**
  - **Document-oriented**
  - **Column-oriented**
  - **Graph-based**

- Most of the NoSQL databases are multi-model DBMSs.
  - They support more than one DBMS type.
  - Example: CosmosDB can be used to store data in Key-value Based, Column-oriented, Document-oriented, or Graph-based formats.

Column Family Stores

Column Family Store

- Example DBMSs: Google BigTable, Cassandra, HBase.
- Stores data in columns rather than rows.
- Each row has a unique key called Row Key.
- Each row can contain a different number of columns.
- Mostly used in customer relation management (CRM) systems, Library catalogs, etc.

Row Key1	Column Family1		
	Column1	Column2	Column3
	Value1	Value2	Value3
	Column Family2		
	Column4	Column5	
	Value4	Value5	

Column Family Store Example

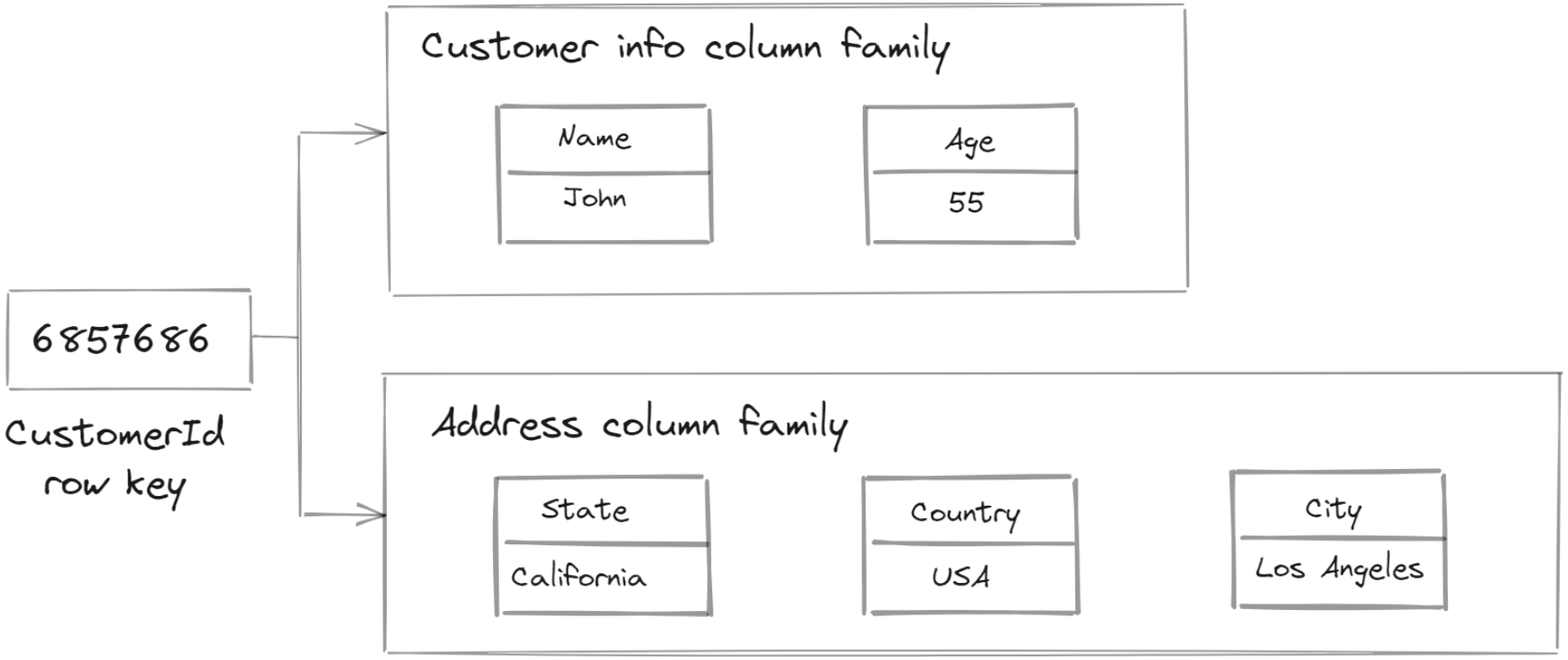
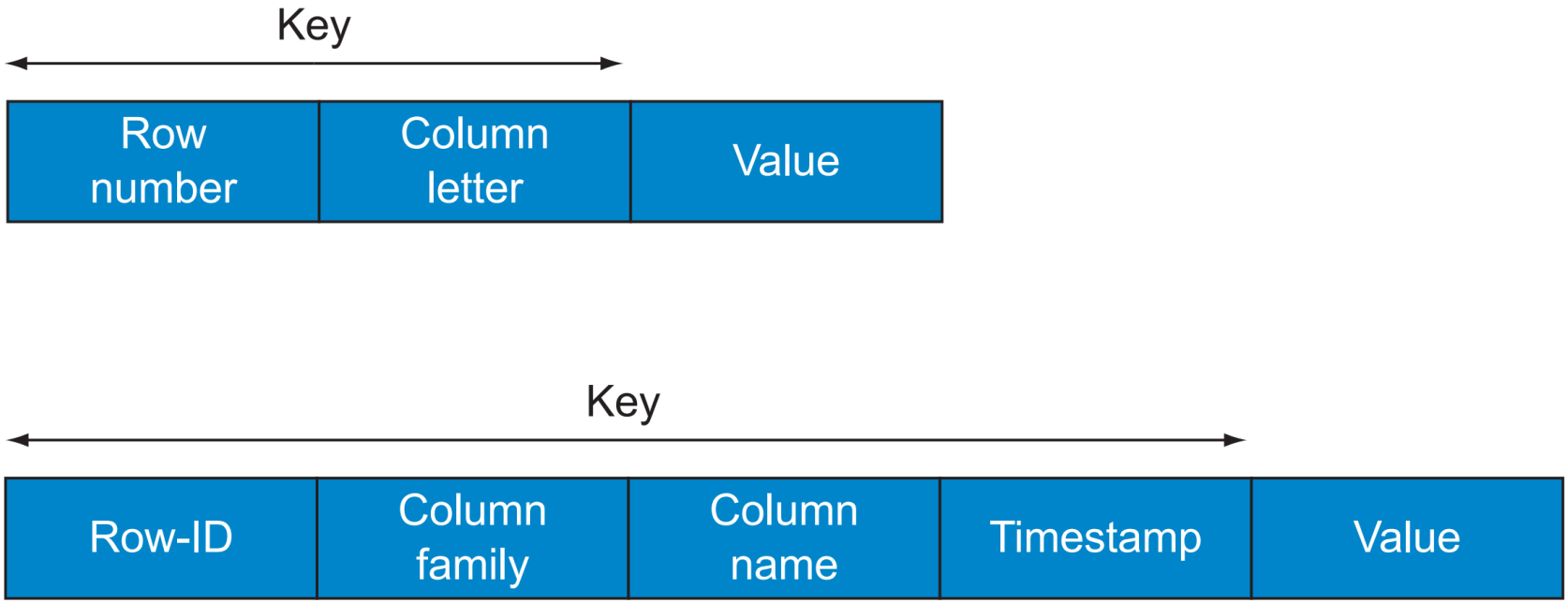


image credit: scaleyourapp.com

Column Family Store

- key is a (row, column) pair
- sparse matrix
- advanced keys: (row, column family, column, timestamp)
- column family: groups of columns
- timestamp: store multiple values over time

Column Family Store



Column Family Store Example

- user preferences
- privacy settings, contact information, notifications, . . .
- typically under 100 fields, 1 KB
- app Requirements:
  - only the associated user makes changes: no ACID requirements
  - mostly read
  - has to be fast and scalable

XML Databases

XML

- XML is not a language itself
- framework for defining languages
- XML-based languages: XHTML, DocBook, SVG, . . .
- XML processing languages:
- XPath, XQuery, XSL Transforms, . . .

XML Applications

- Storing and exchanging data with complex structures
  - E.g., Open Document Format (ODF) format standard for storing Open Office and Office Open XML (OOXML) format standard for storing Microsoft Office documents
  - Numerous other standards for a variety of applications
    - ChemML, MathML
- Standard for data exchange for Web services
  - remote method invocation over HTTP protocol
- Data mediation
  - Common data representation format to bridge different systems

Web Services

- The Simple Object Access Protocol (SOAP) standard:
  - Invocation of procedures across applications with distinct databases
  - XML used to represent procedure input and output
- A *Web service* is a site providing a collection of SOAP procedures
  - Described using the Web Services Description Language (WSDL)
  - Directories of Web services are described using the Universal Description, Discovery, and Integration (UDDI) standard

Structure of XML Data

- **Tag**: label for a section of data
- **Element**: section of data beginning with <*tagname*> and ending with matching </*tagname*>
- Elements must be properly nested
  - Proper nesting
    - <course> … <title> …. </title> </course>
  - Improper nesting
    - <course> … <title> …. </course> </title>
  - Formally: every start tag must have a unique matching end tag, that is in the context of the same parent element.
- Every document must have a single top-level element

Example of Nested Elements

```
<purchase_order>
   <identifier> P-101 </identifier>
   <purchaser>  … </purchaser>
   <itemlist>
       <item>
            <identifier> RS1 </identifier>
            <description> Atom powered rocket sled </description>
            <quantity> 2 </quantity>
            <price> 199.95 </price>
       </item>
       <item>
            <identifier> SG2 </identifier>
            <description> Superb glue </description>
            <quantity> 1 </quantity>
            <unit-of-measure> liter </unit-of-measure>
            <price> 29.95 </price>
       </item>
    </itemlist>
</purchase_order>
```

XML Example: Movies

```
<movies>
  <movie color="Color">
    <title>The Usual Suspects</title>
    <year>1995</year>
    ...
  </movie>
  <movie color="Color">
    <title>Being John Malkovich</title>
    <year>1999</year>
    ...
  </movie>
  ...
</movies>
```

Structure of XML Data (Cont.)

- Mixture of text with sub-elements is legal in XML.
  - Example:

```
<course>
    This course is being offered for the first time in 2009.
    <course id> BIO-399 </course id>
    <title> Computational Biology </title>
    <dept name> Biology </dept name>
    <credits> 3 </credits>
</course>
```

  - Useful for document markup, but discouraged for data representation

Attributes

- Elements can have **attributes**

```
<course course_id= “CS-101”>
    <title> Intro. to Computer Science</title>
    <dept name> Comp. Sci. </dept name>
    <credits> 4 </credits>
</course>
```

- Attributes are specified by *name=value* pairs inside the starting tag of an element
- An element may have several attributes, but each attribute name can only occur once

```
<course  course_id = “CS-101”  credits=“4”>
```

Attributes vs. Subelements

- Distinction between subelement and attribute
  - In the context of data representation, the difference is unclear and may be confusing
    - Same information can be represented in two ways
      - <course course_id= “CS-101”> … </course>
      - <course>
        <course_id>CS-101</course_id> …
        </course>
  - Suggestion: use attributes for identifiers of elements, and use subelements for contents

Well-Formed Documents

- well-formed: conforming to XML rules
- syntactically correct
- single root element
- proper nesting of elements: matched tags
- unique attributes within elements
- XML parsers convert well-formed XML documents into DOM objects (Document Object Model)

Well-Formed Documents

- well-formed: conforming to XML rules
- syntactically correct
- single root element
- proper nesting of elements: matched tags
- unique attributes within elements
- XML parsers convert well-formed XML documents into DOM objects (Document Object Model)

Valid Documents

- valid: conforming to domain rules
- semantically correct
- DTD, XML Schema
- validating XML parsers also check for validity

Valid Documents

- valid: conforming to domain rules
- semantically correct
- DTD, XML Schema
- validating XML parsers also check for validity

XML Document Schema

- Database schemas constrain what information can be stored, and the data types of stored values
- XML documents are not required to have an associated schema
- However, schemas are very important for XML data exchange
  - Otherwise, a site cannot automatically interpret data received from another site
- Two mechanisms for specifying XML schema
  - **Document Type Definition (DTD)**
    - Widely used
  - **XML Schema**
    - Newer, increasing use

Document Type Definition (DTD)

- The type of an XML document can be specified using a DTD
- DTD constraints structure of XML data
  - What elements can occur
  - What attributes can/must an element have
  - What subelements can/must occur inside each element, and how many times.
- DTD does not constrain data types
  - All values represented as strings in XML
- DTD syntax
  - <!ELEMENT element (subelements-specification) >
  - <!ATTLIST element (attributes) >

Element Specification in DTD

- Subelements can be specified as
  - names of elements, or
  - #PCDATA (parsed character data), i.e., character strings
  - EMPTY (no subelements) or ANY (anything can be a subelement)
- Example

  <! ELEMENT department (dept_name building, budget)>
  <! ELEMENT dept_name (#PCDATA)>
  <! ELEMENT budget (#PCDATA)>
- Subelement specification may have regular expressions

  <!ELEMENT university ((department | course | instructor | teaches)+)>
  - Notation:
    - "|" - alternatives
    - "+" - 1 or more occurrences
    - "*" - 0 or more occurrences

University DTD

```
<!DOCTYPE  university [
   <!ELEMENT university (
   (department|course|instructor|teaches)+)>
   <!ELEMENT department ( dept name, building, budget)>
   <!ELEMENT course ( course id, title, dept name, credits)>
   <!ELEMENT instructor (IID, name, dept name, salary)>
   <!ELEMENT teaches (IID, course id)>
   <!ELEMENT dept name( #PCDATA )>
   <!ELEMENT building( #PCDATA )>
   <!ELEMENT budget( #PCDATA )>
   <!ELEMENT course id ( #PCDATA )>
   <!ELEMENT title ( #PCDATA )>
   <!ELEMENT credits( #PCDATA )>
   <!ELEMENT IID( #PCDATA )>
   <!ELEMENT name( #PCDATA )>
   <!ELEMENT salary( #PCDATA )>
]>
```

Attribute Specification in DTD

- Attribute specification : for each attribute
  - Name
  - Type of attribute
    - CDATA
    - ID (identifier) or IDREF (ID reference) or IDREFS (multiple IDREFs)
      - more on this later
  - Whether
    - mandatory (#REQUIRED)
    - has a default value (value),
    - or neither (#IMPLIED)
- Examples
  - <!ATTLIST course course_id CDATA #REQUIRED> or
  - <!ATTLIST course
    course_id ID #REQUIRED
    dept_name IDREF #REQUIRED
    instructors IDREFS #IMPLIED >

IDs and IDREFs

- An element can have at most one attribute of type ID
- The ID attribute value of each element in an XML document must be distinct
  - Thus the ID attribute value is an object identifier
- An attribute of type IDREF must contain the ID value of an element in the same document
- An attribute of type IDREFS contains a set of (0 or more) ID values. Each ID value must contain the ID value of an element in the same document

University DTD with Attributes

```
<!DOCTYPE university-3 [
      <!ELEMENT university ( (department|course|instructor)+)>
      <!ELEMENT department ( building, budget )>
      <!ATTLIST department
            dept_name ID #REQUIRED >
      <!ELEMENT course (title, credits )>
      <!ATTLIST course
            course_id ID #REQUIRED
            dept_name IDREF #REQUIRED
            instructors IDREFS #IMPLIED >
      <!ELEMENT instructor ( name, salary )>
      <!ATTLIST instructor
            IID ID #REQUIRED
            dept_name IDREF #REQUIRED >
      · · · declarations for title, credits, building,
            budget, name and salary · · ·
]>
```

XML data with ID and IDREF attributes

```
<university-3>
        <department dept name="Comp. Sci.">
                <building> Taylor </building>
                <budget> 100000 </budget>
        </department>
        <department dept name="Biology">
                <building> Watson </building>
                <budget> 90000 </budget>
        </department>
        <course course id="CS-101" dept name="Comp. Sci"
                        instructors="10101 83821">
                 <title> Intro. to Computer Science </title>
                 <credits> 4 </credits>
        </course>
        …
        <instructor IID="10101" dept name="Comp. Sci.">
                 <name> Srinivasan </name>
                 <salary> 65000 </salary>
        </instructor>
        …
</university-3>
```

Limitations of DTDs

- No typing of text elements and attributes
  - All values are strings, no integers, reals, etc.
- Difficult to specify unordered sets of subelements
  - Order is usually irrelevant in databases (unlike in the document-layout environment from which XML evolved)
  - (A | B)* allows specification of an unordered set, but
    - Cannot ensure that each of A and B occurs only once
- IDs and IDREFs are untyped
  - The *instructors* attribute of an course may contain a reference to another course, which is meaningless
    - *instructors* attribute should ideally be constrained to refer to instructor elements

XML Schema

- XML Schema is a more sophisticated schema language which addresses the drawbacks of DTDs. Supports
  - Typing of values
    - E.g., integer, string, etc.
    - Also, constraints on min/max values
  - User-defined, complex types
  - Many more features, including
    - uniqueness and foreign key constraints, inheritance
- XML Schema is itself specified in XML syntax, unlike DTDs
  - More-standard representation, but verbose
- XML Schema is integrated with namespaces
- BUT: XML Schema is significantly more complicated than DTDs.

XML Schema Version of Univ. DTD

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
<xs:element name="university" type="universityType" />
<xs:element name="department">
    <xs:complexType>
        <xs:sequence>
            <xs:element name="dept name" type="xs:string"/>
            <xs:element name="building" type="xs:string"/>
            <xs:element name="budget" type="xs:decimal"/>
        </xs:sequence>
    </xs:complexType>
</xs:element>
....
<xs:element name="instructor">
    <xs:complexType>
        <xs:sequence>
            <xs:element name="IID" type="xs:string"/>
            <xs:element name="name" type="xs:string"/>
            <xs:element name="dept name" type="xs:string"/>
            <xs:element name="salary" type="xs:decimal"/>
        </xs:sequence>
    </xs:complexType>
</xs:element>
```

… Contd.

XML Schema Version of Univ. DTD (Cont.)

```
….
<xs:complexType name="UniversityType">
   <xs:sequence>
      <xs:element ref="department" minOccurs="0" maxOccurs="unbounded"/>
      <xs:element ref="course" minOccurs="0" maxOccurs="unbounded"/>
      <xs:element ref="instructor" minOccurs="0" maxOccurs="unbounded"/>
      <xs:element ref="teaches" minOccurs="0" maxOccurs="unbounded"/>
   </xs:sequence>
</xs:complexType>
</xs:schema>
```

- Choice of "xs:" was ours -- any other namespace prefix could be chosen
- Element "university" has type "universityType", which is defined separately
  - xs:complexType is used later to create the named complex type "UniversityType"

More features of XML Schema

- Attributes specified by xs:attribute tag:
  - <xs:attribute name = “dept_name”/>
  - adding the attribute use = “required” means value must be specified
- Key constraint: “department names form a key for department elements under the root university element:

```
<xs:key name = “deptKey”>
        <xs:selector xpath = “/university/department”/>
        <xs:field xpath = “dept_name”/>
<\xs:key>
```

- Foreign key constraint from course to department:

```
<xs:keyref name = “courseDeptFKey” refer=“deptKey”>
        <xs:selector xpath = “/university/course”/>
        <xs:field xpath = “dept_name”/>
<\xs:keyref>
```

XML Databases

- variant of document stores
- document: XML formatted data

- products: Oracle Berkeley DBXML, BaseX, eXist

Querying and Transforming XML Data

- Translation of information from one XML schema to another
- Querying on XML data
- Above two are closely related, and handled by the same tools
- Standard XML querying/translation languages
  - XPath
    - Simple language consisting of path expressions
  - XSLT
    - Simple language designed for translation from XML to XML and XML to HTML
  - XQuery
    - An XML query language with a rich set of features

Tree Model of XML Data

- Query and transformation languages are based on a **tree model** of XML data
- An XML document is modeled as a tree, with **nodes** corresponding to elements and attributes
  - Element nodes have child nodes, which can be attributes or subelements
  - Text in an element is modeled as a text node child of the element
  - Children of a node are ordered according to their order in the XML document
  - Element and attribute nodes (except for the root node) have a single parent, which is an element node
  - The root node has a single child, which is the root element of the document

XPath

- XPath: selecting nodes and data from XML documents
- path of nodes to find: chain of location steps
  - starting from the root (absolute)
  - starting from the current node (relative)

XPath Examples

- all movies:
  `/movies/movie`
- actors of current movie:
  `./cast/actor`
- `../../year`

```
<movies>
   <movie color="Color">
      <title>The Usual
      Suspects</title>
      <year>1995</year>
      ...
   </movie>
   <movie color="Color">
      <title>Being John
      Malkovich</title>
      <year>1999</year>
      ...
   </movie>
   ...
</movies>
```

Location Steps

- location step structure:

  `axis::node_selector[predicate]`

- axis: where to search
- selector: what to search
- predicate: under which conditions

Axes

- `child`: all children, one level (default axis)
- `descendant`: all children, recursively (shorthand: `//`)
- `parent`: parent node, one level
- `ancestor`: parent nodes, up to document element
- `attribute`: attributes (shorthand: `@`)
- `following-sibling`: siblings that come later
- `preceding-sibling`: siblings that come earlier
- ...

Node Selectors

- node tag
- node attribute
- node text: `text()`
- all children: `*`

XPath Examples

- names of all directors:

```
/movies/movie/director/text()
//director/text()
```

- all actors in this movie:

```
./cast/actor
.//actor
```

- colors of all movies:

```
//movie/@color
```

- scores of movies after this one:

```
./following-sibling::movie/s
```

```
<movies>
  <movie color="Color">
    <title>The Usual
    Suspects</title>
    <year>1995</year>
    ...
  </movie>
  <movie color="Color">
    <title>Being John
    Malkovich</title>
    <year>1999</year>
    ...
  </movie>
  ...
</movies>
```

XPath Examples

- names of all directors:

```
/movies/movie/director/text()
//director/text()
```

- all actors in this movie:

```
./cast/actor
.//actor
```

- colors of all movies:

```
//movie/@color
```

- scores of movies after this one:

```
./following-sibling::movie/s
```

```
<movies>
   <movie color="Color">
      <title>The Usual
      Suspects</title>
      <year>1995</year>
      ...
   </movie>
   <movie color="Color">
      <title>Being John
      Malkovich</title>
      <year>1999</year>
      ...
   </movie>
   ...
</movies>
```

XPath Examples

- names of all directors:

```
/movies/movie/director/text()
//director/text()
```

- all actors in this movie:

```
./cast/actor
.//actor
```

- colors of all movies:

```
//movie/@color
```

- scores of movies after this one:

```
./following-sibling::movie/s
```

```
<movies>
  <movie color="Color">
    <title>The Usual
    Suspects</title>
    <year>1995</year>
    ...
  </movie>
  <movie color="Color">
    <title>Being John
    Malkovich</title>
    <year>1999</year>
    ...
  </movie>
  ...
</movies>
```

XPath Examples

- names of all directors:

```
/movies/movie/director/text(
//director/text()
```

- all actors in this movie:

```
./cast/actor
.//actor
```

- colors of all movies:

```
//movie/@color
```

```
<movies>
  <movie color="Color">
    <title>The Usual
    Suspects</title>
    <year>1995</year>
    ...
  </movie>
  <movie color="Color">
    <title>Being John
    Malkovich</title>
    <year>1999</year>
    ...
  </movie>
  ...
</movies>
```

- scores of movies after this one:

```
./following-sibling::movie/score
```

XPath Predicates

- testing node position: `[position]`
- testing existence of a child: `[child_tag]`
- testing value of a child: `[child_tag="value"]`
- testing existence of an attribute: `[@attribute]`
- testing value of an attribute: `[@attribute="value"]`

XPath Predicates

- testing node position: `[position]`
- testing existence of a child: `[child_tag]`
- testing value of a child: `[child_tag="value"]`
- testing existence of an attribute: `[@attribute]`
- testing value of an attribute: `[@attribute="value"]`

XPath Predicates

- testing node position: `[position]`
- testing existence of a child: `[child_tag]`
- testing value of a child: `[child_tag="value"]`
- testing existence of an attribute: `[@attribute]`
- testing value of an attribute: `[@attribute="value"]`

XPath Examples

- title of the first movie:
  `/movies/movie[1]/title`
- all movies in the year 1997:
  `movie[year="1997"]`
- black-and-white movies:
  `movie[@color="BW"]`

```
<movies>
   <movie color="Color">
      <title>The Usual
      Suspects</title>
      <year>1995</year>
      ...
   </movie>
   <movie color="Color">
      <title>Being John
      Malkovich</title>
      <year>1999</year>
      ...
   </movie>
   ...
</movies>
```

XPath Examples

- title of the first movie:
  `/movies/movie[1]/title`
- all movies in the year 1997:
  `movie[year="1997"]`
- black-and-white movies:
  `movie[@color="BW"]`

```
<movies>
   <movie color="Color">
      <title>The Usual
      Suspects</title>
      <year>1995</year>
      ...
   </movie>
   <movie color="Color">
      <title>Being John
      Malkovich</title>
      <year>1999</year>
      ...
   </movie>
   ...
</movies>
```

XPath Examples

- title of the first movie:
  `/movies/movie[1]/title`
- all movies in the year 1997:
  `movie[year="1997"]`
- black-and-white movies:
  `movie[@color="BW"]`

```
<movies>
   <movie color="Color">
      <title>The Usual
      Suspects</title>
      <year>1995</year>
      ...
   </movie>
   <movie color="Color">
      <title>Being John
      Malkovich</title>
      <year>1999</year>
      ...
   </movie>
   ...
</movies>
```

Xpath Examples

- /university-3/instructor/name evaluated on the university-3 data we saw earlier returns

  <name>Srinivasan</name>
  <name>Brandt</name>

- /university-3/instructor/name/text( )

  returns the same names, but without the enclosing tags

  Srinivasan
  Brandt

XPath Examples (Cont.)

/university-3/course[credits >= 4]

- returns course elements with a credit value greater than 4

/university-3/course[credits]

- returns course elements containing a credits subelement

/university-3/course[credits >= 4]/@course_id

- returns the course identifiers of courses with credits >= 4

XQuery

- XQuery is a general purpose query language for XML data
- Currently being standardized by the World Wide Web Consortium (W3C)
  - The textbook description is based on a January 2005 draft of the standard. The final version may differ, but major features likely to stay unchanged.
- XQuery is derived from the Quilt query language, which itself borrows from SQL, XQL and XML-QL
- XQuery uses a
  for … let … where … order by …result …
  syntax
  for ⇔ SQL **from**
  where ⇔ SQL **where**
  order by ⇔ SQL **order by**
  return ⇔ SQL **select**

  let allows temporary variables, and has no equivalent in SQL

FLWOR Syntax in XQuery

- For clause uses XPath expressions, and variable in for clause ranges over values in the set returned by XPath
- Simple FLWOR expression in XQuery
  - find all courses with credits > 3, with each result enclosed in an <course_id> .. </course_id> tag

```
for  $x in /university-3/course
let   $courseId := $x/@course_id
where $x/credits > 3
return <course_id> { $courseId } </course id>
```

  - Items in the **return** clause are XML text unless enclosed in {}, in which case they are evaluated
- Let clause is not really needed in this query, and selection can be done In XPath. Query can be written as:

```
for $x in /university-3/course[credits > 3]
return <course_id> { $x/@course_id } </course_id>
```

- Alternative notation for constructing elements:

```
return element course_id { element  $x/@course_id }
```

Joins

- Joins are specified in a manner very similar to SQL

```
for $c in /university/course,
    $i in /university/instructor,
    $t in /university/teaches
where $c/course_id= $t/course id and $t/IID = $i/IID
return <course_instructor> { $c $i } </course_instructor>
```

- The same query can be expressed with the selections specified as XPath selections:

```
for $c in /university/course,
    $i in /university/instructor,
    $t in /university/teaches[ $c/course_id= $t/course_id
                               and $t/IID = $i/IID]
return <course_instructor> { $c $i } </course_instructor>
```

Graph-based NoSQL

Graph-based NoSQL

- Example Products: Neo4j, OrientDB, Microsoft Cosmos DB, ArangoDB, Amazon Neptune.
- Stores entities as well the relations amongst those entities.
- The entity is stored as a node with the relationship as edges.
- An edge gives a relationship between nodes.
- Every node and edge has a unique identifier.
- Traversals instead of joins
- Mostly used for social networks, logistics, spatial data.

Graph Databases

- better suited for tasks like: shortest path, friends of friends,
- neighboring nodes with specific properties
- difficult to scale out

- declarative query languages: Cypher, Gremlin

Cypher

- locate the initial nodes
- select and traverse relationships
- change and/or return values

Cypher: Nodes

- `(name)`
- `(name:Type)`
- `(name:Type {attributes})`

```
(matrix)
(matrix:Movie)
(matrix:Movie {title: "The Matrix"})
(matrix:Movie {title: "The Matrix", released: 1997})
```

Cypher: Relationships

- undirected: `--`
- directed: `--> <--`
- with details: `-[]-`

```
-[role]->
-[role:ACTED_IN]->
-[role:ACTED_IN {roles: ["Neo"]}]->
```

Cypher: Patterns

- combine nodes and relationships
- give names to patterns

```
(keanu:Person   {name:  "Keanu Reeves"} )
-[role:ACTED_IN {roles: ["Neo"] } ]->
(matrix:Movie   {title: "The Matrix"} )

acted_in = (:Person)-[:ACTED_IN]->(:Movie)
```

Cypher: Creating Data

```
CREATE (:Movie {title: "The Matrix", released: 1997})

CREATE (p:Person {name: "Keanu Reeves", born: 1964})
RETURN p

CREATE (a:Person {name: "Tom Hanks", born:1956 })
  -[r:ACTED_IN {roles: ["Forrest"]}]->
  (m:Movie {title: "Forrest Gump", released: 1994})
CREATE (d:Person {name: "Robert Zemeckis", born: 1951})
  -[:DIRECTED]-> (m)
RETURN a, d, r, m
```

Cypher: Matching Patterns

```
MATCH (m:Movie)
RETURN m

MATCH (p:Person {name:"Keanu Reeves"})
RETURN p

MATCH (p:Person {name:"Tom Hanks"})
  -[r:ACTED_IN]-> (m:Movie)
RETURN m.title, r.roles
```

References

Supplementary Reading

- Making Sense of NoSQL, by Dan McCreary and Ann Kelly, Manning Publications
- The Neo4J Manual: Tutorials
  http://neo4j.com/docs/stable/tutorials.html

Extra Slides on XQuery

Nested Queries

- The following query converts data from the flat structure for university information into the nested structure used in university-1

```
<university-1>
{    for $d in /university/department
     return <department>
                { $d/* }
                { for $c in /university/course[dept name = $d/dept name]
                  return $c }
            </department>
}
{    for $i in /university/instructor
     return  <instructor>
                { $i/* }
                { for $c in /university/teaches[IID = $i/IID]
                  return $c/course id }
             </instructor>
}
</university-1>
```

- $c/* denotes all the children of the node to which $c is bound, without the enclosing top-level tag

Grouping and Aggregation

- Nested queries are used for grouping

```
for $d in /university/department
return
    <department-total-salary>
        <dept_name> { $d/dept name } </dept_name>
         <total_salary> { fn:sum(
             for $i in /university/instructor[dept_name = $d/dept_name]
             return $i/salary
           ) }
         </total_salary>
     </department-total-salary>
```

Sorting in XQuery

- The **order by** clause can be used at the end of any expression. E.g., to return instructors sorted by name

```
 for $i in /university/instructor
order by $i/name
return <instructor> { $i/* } </instructor>
```

- Use **order by** $i/name **descending** to sort in descending order
- Can sort at multiple levels of nesting (sort departments by dept_name, and by courses sorted to course_id within each department)

```
<university-1> {
  for $d in /university/department
  order by $d/dept name
  return
      <department>
         { $d/* }
         { for $c in /university/course[dept name = $d/dept name]
           order by $c/course id
           return <course> { $c/* } </course> }
      </department>
} </university-1>
```

Functions and Other XQuery Features

- User defined functions with the type system of XMLSchema

```
declare function local:dept_courses($iid as xs:string)
        as element(course)*
{
    for $i in /university/instructor[IID = $iid],
        $c in /university/courses[dept_name = $i/dept name]
    return $c
}
```

- Types are optional for function parameters and return values
- The * (as in decimal*) indicates a sequence of values of that type
- Universal and existential quantification in where clause predicates
  - **some** $e **in** *path* **satisfies** *P*
  - **every** $e **in** *path* **satisfies** *P*
  - Add **and fn:exists($e)** to prevent empty $e from satisfying **every** clause
- XQuery also supports If-then-else clauses