Database Development and Design (CPT201)

Lecture 8a: Introduction to XML

Dr. Wei Wang
Department of Computing

Learning Outcomes

- Structure of XML Data
- XML Document Schema
- Querying and Transformation
- Application Program Interfaces to XML
- Storage of XML Data
- XML Applications

Introduction

- XML: Extensible Markup Language
- Defined by the WWW Consortium (W3C)
- Derived from SGML (Standard Generalised Markup Language), but simpler to use than SGML
- Documents have tags giving extra information about sections of the document
  - E.g. <title> XML </title> <slide> Introduction …</slide>
- **Extensible**, unlike HTML
  - Users can add new tags, and *separately* specify how the tag should be handled for display

XML Introduction cont'd

- The ability to specify new tags, and to create nested tag structures make XML a great way to exchange **data**, not just documents.
- Tags make data (relatively) self-documenting

```
<university>
  <department>
    <dept_name> Comp. Sci. </dept_name>
    <building> Taylor </building>
    <budget> 100000 </budget>
  </department>
  <course>
    <course_id> CS-101 </course_id>
    <title> Intro. to Computer Science </title>
    <dept_name> Comp. Sci </dept_name>
    <credits> 4 </credits>
  </course>
</university>
```

XML: Motivation

- Data interchange is critical in a networked world
  - Examples:
    - Banking: funds transfer
    - Order processing (especially inter-company orders)
    - Scientific data
      - Chemistry: ChemML, …
      - Genetics: BSML (Bio-Sequence Markup Language), …
  - Paper flow of information between organisations is being replaced by electronic flow of information
- Each application area has its own set of standards for representing information
- XML has become the basis for all new generation data interchange formats

XML Motivation (Cont.)

- Earlier generation formats were based on plain text with line headers indicating the meaning of fields
  - Similar in concept to email headers
  - Does not allow for nested structures, no standard "type" language
  - Tied too closely to low level document structure (lines, spaces, etc)
- Each XML based standard defines what are valid elements, using
  - XML type specification languages to specify the syntax
    - DTD (Document Type Descriptors)
    - XML Schema
  - Plus textual descriptions of the semantics
- XML allows new tags to be defined as required
- A variety of tools is available for parsing, browsing and querying XML documents/data

Comparison with Relational Data

- Inefficient: tags, which represent schema information, are repeated
- Better than relational tuples as a data-exchange format
  - Self-documenting due to presence of tags
  - Non-rigid format: new tags can be added
  - Allows nested structures
  - Wide acceptance
    - not only in database systems, but also in browsers, tools, and applications

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>
```

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 documents, attributes are part of markup, while subelement contents are part of the basic document contents
  - In the context of data representation, the difference is unclear and may be confusing
    - Same information can be represented in both 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

Namespace

- XML data has to be exchanged between organisations
- Same tag name may have different meaning in different organisations, causing confusion on exchanged documents
- Specifying a unique string as an element name avoids confusion
- Better solution: use unique-name:element-name
- Avoid using long unique names all over document by using XML Namespaces

```
<university xmlns:yale="http://www.yale.edu">
    ...
    <yale:course>
        <yale:course_id> CS-101 </yale:course_id>
        <yale:title> Intro. to Computer Science</yale:title>
        <yale:dept_name> Comp. Sci. </yale:dept_name>
        <yale:credits> 4 </yale:credits>
    </yale:course>
    ...
</university>
```

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)
  - XML Schema

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, followed by
  - #PCDATA (parsed character data), i.e., character strings, or
  - 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)
  - 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 >

University DTD with Attributes

University DTD with ID, IDREF, and IDREFS attribute types.

```
<!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 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 Scheme is integrated with namespaces
- XML Schema is itself specified in XML syntax, unlike DTDs
  - More-standard representation, but verbose
  - XML Schema is significantly more complicated than DTDs.

```
<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>
<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>
```

Querying and Transforming XML Data

- Translation of information from XML to another (with different schemas)
- 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

XPath

- XPath is used to address (select) parts of documents using **path expressions**
- A path expression is a sequence of steps separated by "/"
  - Think of file names in a directory hierarchy
- Result of path expression: set of values that along with their containing elements/attributes match the specified path
- E.g., /university-3/instructor/name evaluated on the university-3 data we saw earlier returns (see the examples in textbook)

  <name>Srinivasan</name>
  <name>Brandt</name>
- E.g., /university-3/instructor/name/text( ) returns the same names, but without the enclosing tags

XSLT

- A **stylesheet** stores formatting options for a document, usually separately from document
  - e.g. an HTML style sheet may specify font colors and sizes for headings, etc.
- The **XML Stylesheet Language (XSL)** was originally designed for generating HTML from XML
- XSLT is a general-purpose transformation language
  - Can translate XML to XML, and XML to HTML
- XSLT transformations are expressed using rules called **templates**
  - Templates combine selection using XPath with construction of results

XQuery

- XQuery is a general purpose query language for XML data
- Standardised 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 FLWOR (i.e., **for** … **let** … **where** … **order by** …**return**…) syntax
  - **for** ⇔ SQL **from**
  - **let** allows temporary variables, and has no equivalent in SQL
  - **where** ⇔ SQL **where**
  - **order by** ⇔ SQL **order by**
  - **return** ⇔ SQL **select**

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

Storage of XML Data

- XML data can be stored in
  - Non-relational data stores
    - Flat files
      - Natural for storing XML
      - But has all problems, e.g., no concurrency, no recovery, …
    - XML database
      - Database built specifically for storing XML data, supporting DOM model and declarative querying
  - Relational databases
    - Data must be translated into relational form
    - Advantage: mature database systems
    - Disadvantages: overhead of translating data and queries

Storage of XML in Relational Databases

- Alternatives:
  - String Representation
  - Tree Representation
  - Map to relations

String Representation

- Store each top level element as a string field of a tuple in a relational database
  - Use a single relation to store all elements, or
  - Use a separate relation for each top-level element type
    - E.g. account, customer, depositor relations
      - Each with a string-valued attribute to store the element
- Indexing:
  - Store values of subelements/attributes to be indexed as extra fields of the relation, and build indices on these fields
    - E.g. customer_name or account_number
  - Some database systems support **function indices**, which use the result of a function as the key value.
    - The function should return the value of the required subelement/ attribute

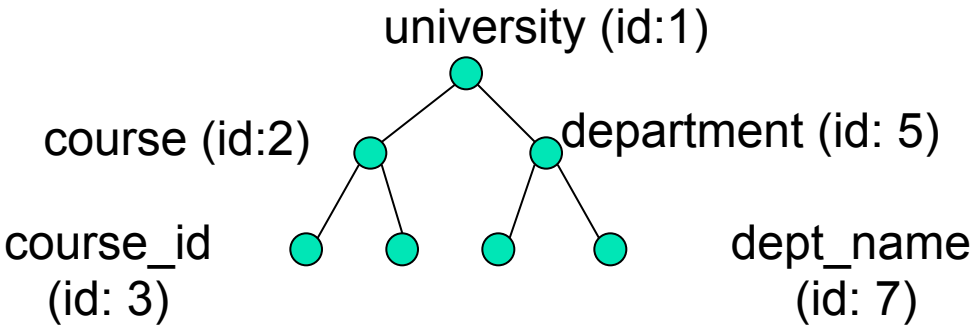
String Representation cont'd

- Benefits:
  - Can store any XML data even without DTD
  - Allows fast access to individual elements.
- Drawback:
  - Need to parse strings to access values inside the elements
  - Parsing is slow

Tree Representation

- **Tree representation:** model XML data as tree and store using relations

  nodes(id, parent_id, type, label, value)



- Each element/attribute is given a unique identifier
- Type indicates element/attribute
- Label specifies the tag name of the element/name of attribute
- Value is the text value of the element/attribute
- Can add an extra attribute *position* to record ordering of children

Tree Representation cont'd

- Benefit:
  - Can store any XML data, even without DTD
- Drawbacks:
  - Data is broken up into too many pieces, increasing space overheads
  - Even simple queries require a large number of joins, which can be slow

Mapping XML Data to Relations

- Relation created for each element type whose schema is known:
  - An id attribute to store a unique id for each element
  - A relation attribute corresponding to each element attribute
  - A parent_id attribute to keep track of parent element
    - As in the tree representation
    - Position information (i^{th} child) can be store too
- All subelements that occur only once can become relation attributes
  - For text-valued subelements, store the text as attribute value
  - For complex subelements, can store the id of the subelement
- Subelements that can occur multiple times represented in a separate table
  - Similar to handling of multivalued attributes when converting ER diagrams to tables

Application Program Interface

- There are two standard application programming interfaces to XML data:
  - **SAX** (Simple API for XML)
    - Based on parser model, user provides event handlers for parsing events
      - E.g. start of element, end of element
  - **DOM** (Document Object Model)
    - **XML** data is parsed into a tree representation
    - Variety of functions provided for traversing the DOM tree
    - E.g.: Java DOM API provides Node class with methods
      getParentNode( ), getFirstChild( ), getNextSibling( )
      getAttribute( ), getData( ) (for text node)
      getElementsByTagName( ), ...
    - Also provides functions for updating DOM tree

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, SensorML, etc
- 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
- More from *https://www.w3.org/2002/ws*

End of Lecture

- Summary
  - Structure of XML Data
  - XML Document Schema
  - Querying and Transformation
  - Application Program Interfaces to XML
  - Storage of XML Data
  - XML Applications
- Reading
  - Textbook 6th edition, chapter 23
  - Textbook 7th edition, shortened, refer to the uploaded PDF.

SQL Extensions – Example on SQL Output in XML

- **xmlelement** creates XML elements
- **xmlattributes** creates attributes

```
select xmlelement (name "course",
    xmlattributes (course_id as course_id, dept_name as dept_name),
    xmlelement (name "title", title),
    xmlelement (name "credits", credits))
from course
```

- **Xmlagg** creates a forest of XML elements

```
select xmlelement (name "department",
        dept_name,
        xmlagg (xmlforest(course_id)
                order by course_id))
from course
group by dept_name
```

Storing XML Data in Relational Systems

- Applying above ideas to department elements in university-1 schema, with nested course elements, we get
  department(*id*, *dept_name*, *building*, *budget*)
  course(*parent id*, *course_id*, *dept_name*, *title*, *credits*)
- **Publishing**: process of converting relational data to an XML format
- **Shredding**: process of converting an XML document into a set of tuples to be inserted into one or more relations
- XML-enabled database systems support automated publishing and shredding
- Many systems offer *native storage* of XML data using the xml data type. Special internal data structures and indices are used for efficiency

Xpath cont'd

- The initial "/" denotes root of the document (above the top-level tag)
- Path expressions are evaluated left to right
  - Each step operates on the set of instances produced by the previous step
- Selection predicates may follow any step in a path, in [ ]
  - E.g., /university-3/course[credits >= 4]
    - Returns elements with a value greater than 4
    - /university-3/course[credits] returns account elements containing a credits subelement
- Attributes are accessed using "@"
  - E.g., /university-3/course[credits >= 4]/@course_id
    - returns the course identifiers of courses with credits >= 4
  - IDREF attributes are not dereferenced automatically (more on this later)

Functions in XPath

- XPath provides several functions
  - The function count() at the end of a path counts the number of elements in the set generated by the path
    - E.g., /university-2/instructor[count(./teaches/course)> 2]
      - Returns instructors teaching more than 2 courses (on university-2 schema)
  - Also function for testing position (1, 2, ..) of node w.r.t. siblings
- Boolean connectives and and or and function not() can be used in predicates
- IDREFs can be referenced using function id()
  - id() can also be applied to sets of references such as IDREFS and even to strings containing multiple references separated by blanks
  - E.g., /university-3/course/id(@dept_name)
    - returns all department elements referred to from the dept_name attribute of course elements.

More XPath Features

- Operator "|" used to implement union
  - E.g., /university-3/course[@dept name="Comp. Sci"] | /university-3/course[@dept name="Biology"]
    - Gives union of Comp. Sci. and Biology courses
    - However, "|" cannot be nested inside other operators.
- "//" can be used to skip multiple levels of nodes
  - E.g., /university-3//name
    - finds any name element *anywhere* under the /university-3 element, regardless of the element in which it is contained.
- A step in the path can go to parents, siblings, ancestors and descendants of the nodes generated by the previous step, not just to the children
  - "//", described above, is a short from for specifying "all descendants"
  - ".." specifies the parent.
- doc(name) returns the root of a named document

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 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>
```

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>
```

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

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 XML Schema

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

Database Development and Design (CPT201)

Lecture 8b: Introduction to Semantic Web and RDF

Dr. Wei Wang

Department of Computing

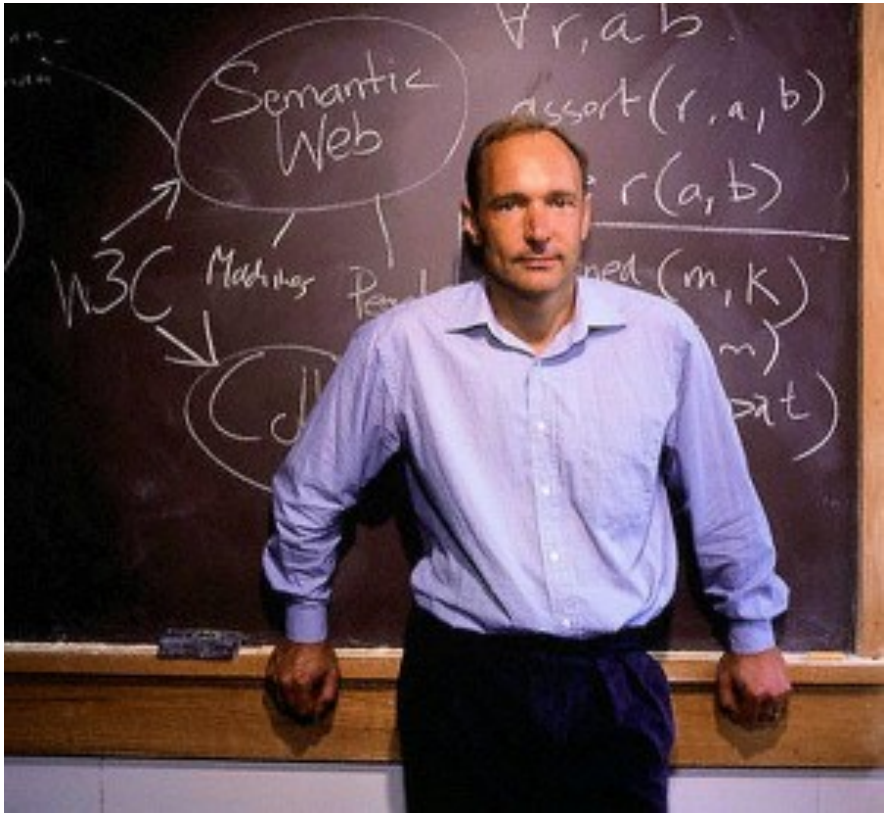
Learning Outcomes

- Introduction to the following
  - Semantic Web
  - Resource description framework
  - Ontology
  - Linked Open Data
  - Query the Web of Data

Semantic Web

"The Semantic Web" is an

- ... extension of the current web in which
- ... information is given well-defined meaning,
- ... better enabling computers and people to work in cooperation."



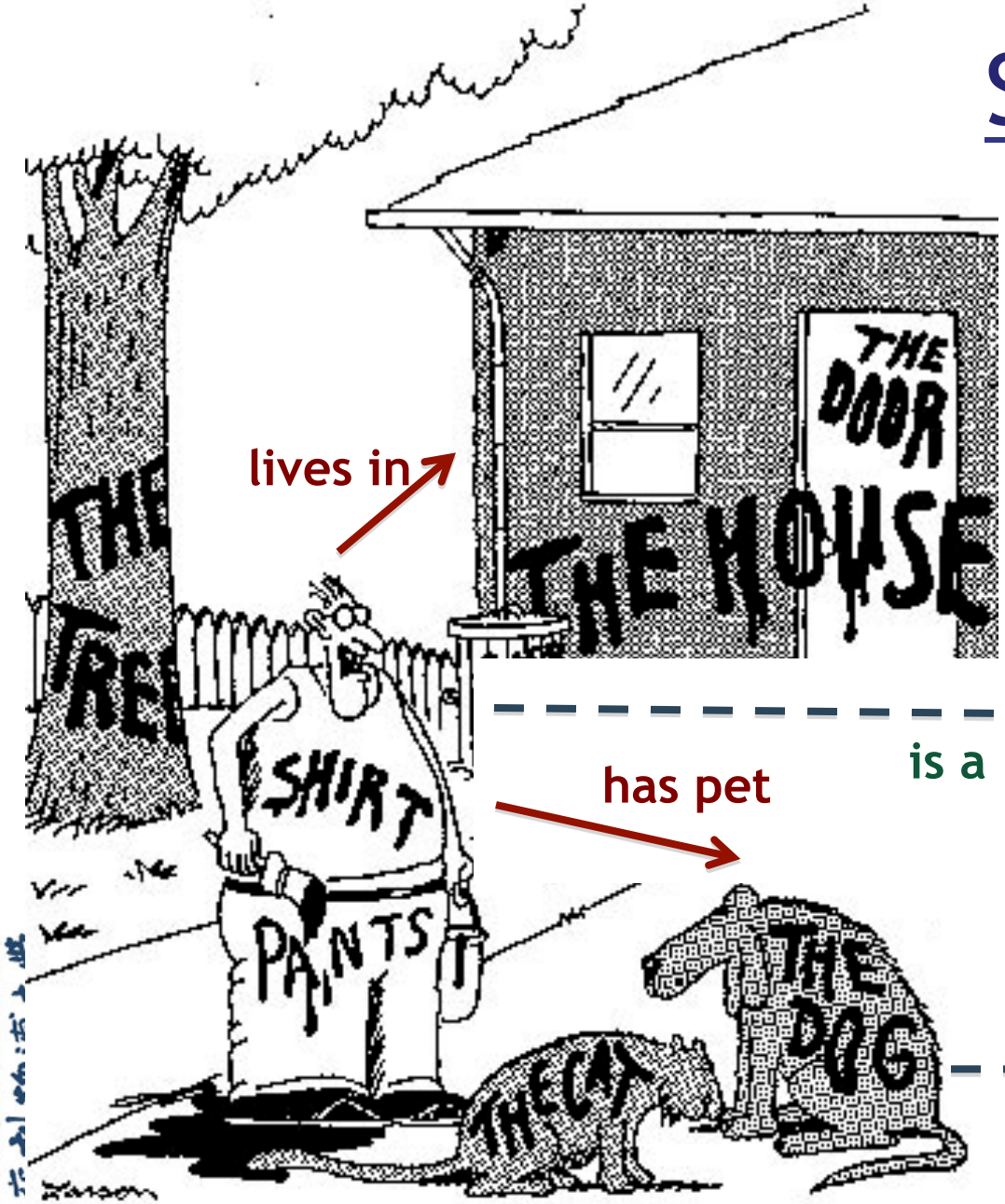
- The Semantic Web, Tim Berners-Lee, James Hendler and Ora Lassila, Scientific American, May 2001

Why Semantic Web?

- Tasks often require combining data across the Internet, e.g.:
  - Integrating data across the enterprise, e.g. hotel, transport, meeting, personal info come from different sites
  - Mining data from biochemical, genetic, pharmaceutical, patient databases
  - Cross-referencing disparate digital libraries
- Humans understand how to combine this information…
  - But not always easy (different vocabularies, languages, formats)
  - Machines aren't smart enough…

Semantic Web

(*according to Farside*)

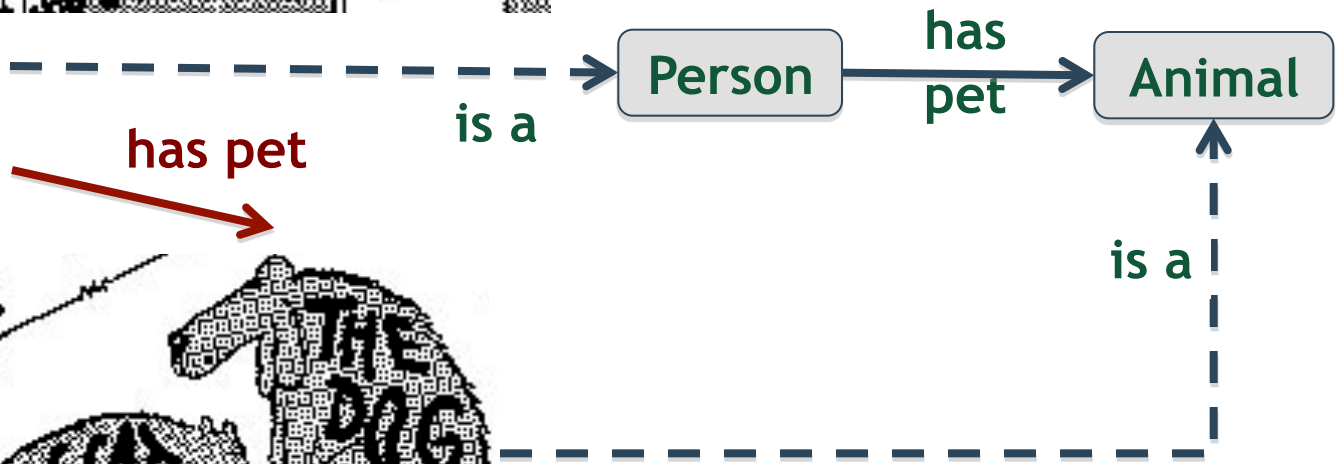


Concrete Facts

Resource Description Framework

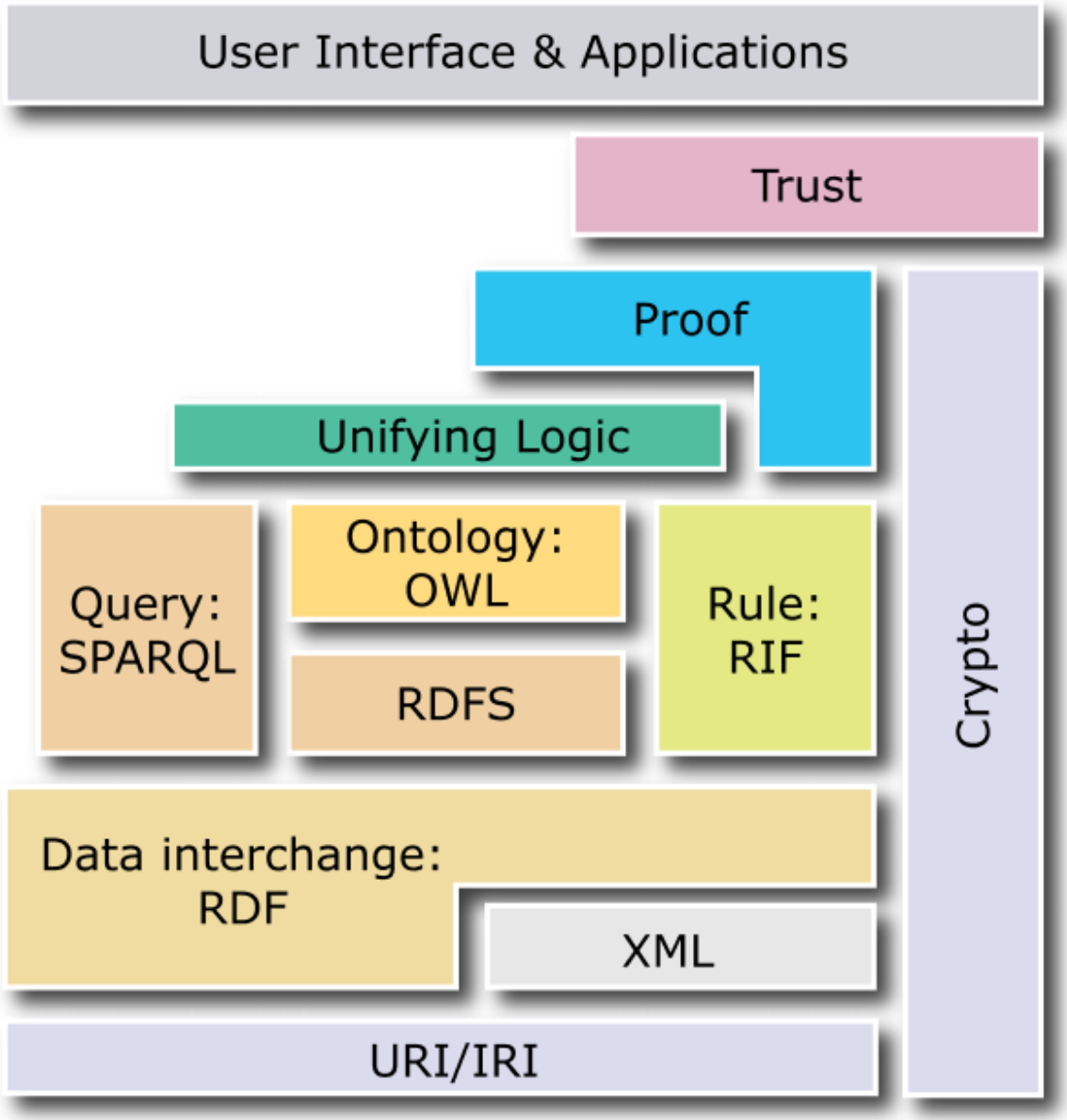
General Knowledge

Web Ontology Language

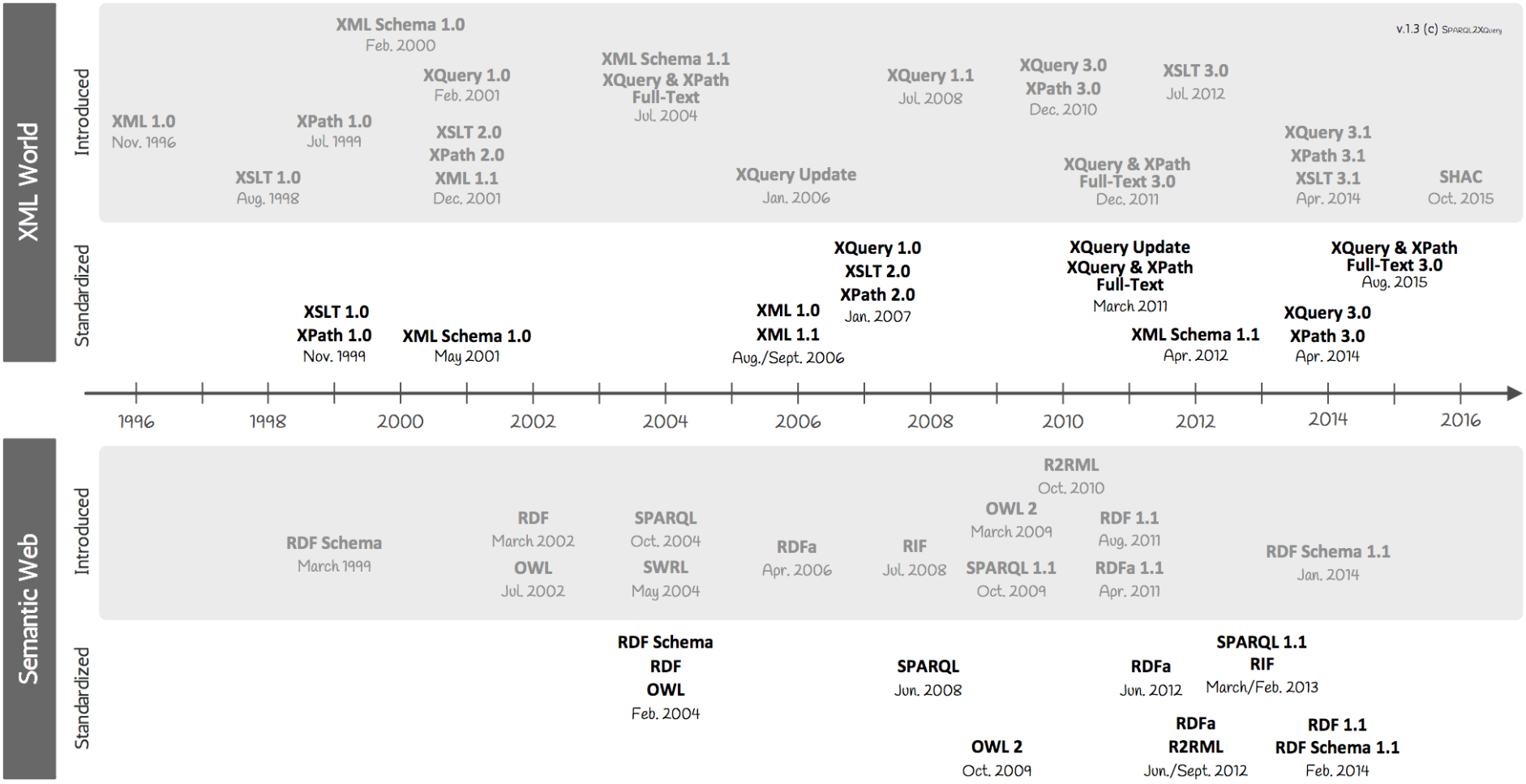


"Now! – *That* should clear up a few things around here!"

Semantic Web Stack



Evolution of Technologies

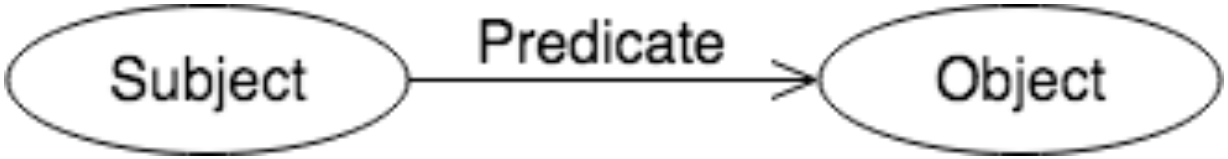


Semantic Web Today and En Route

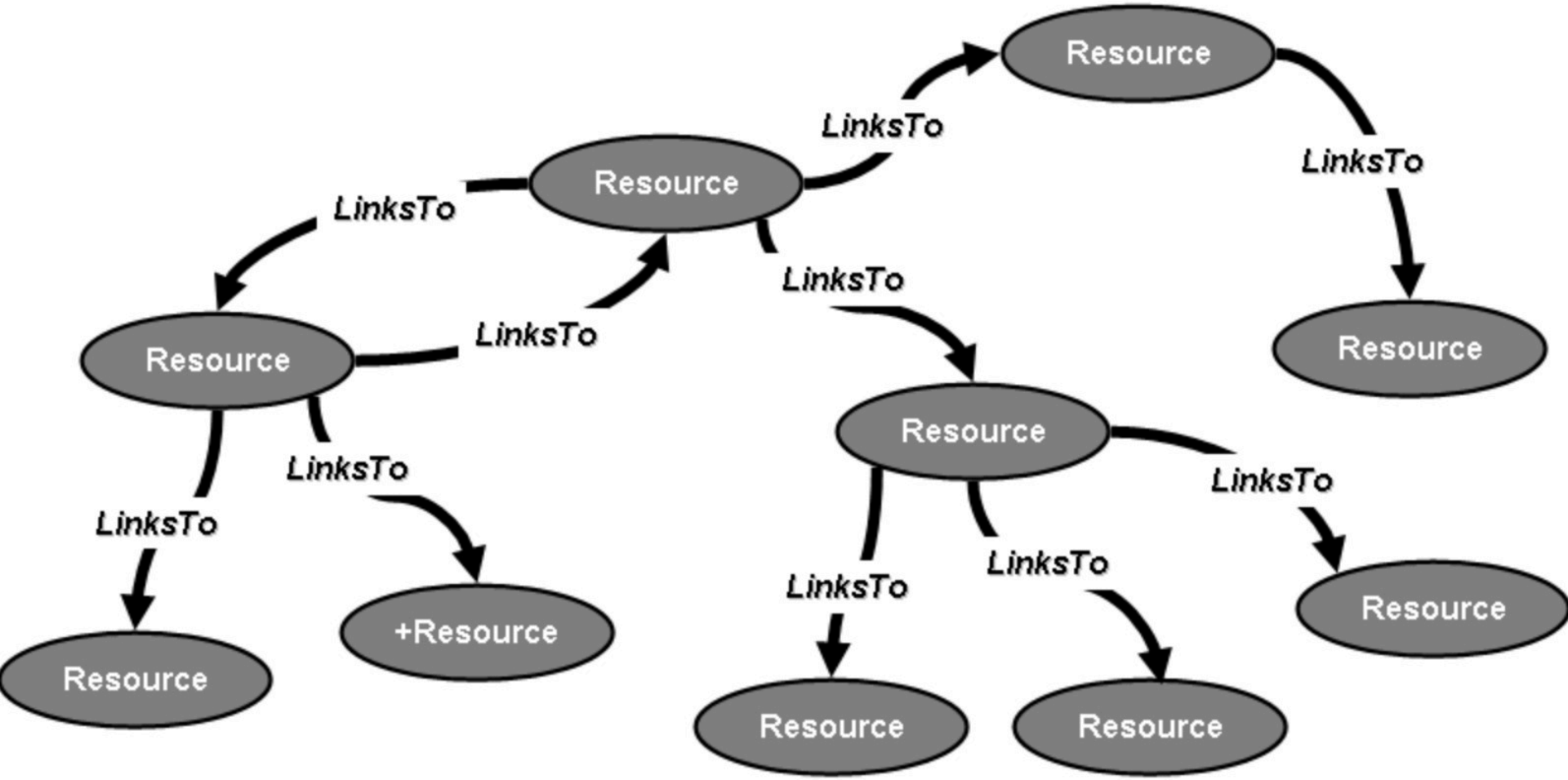
- Semantic Web helps build a technology stack to support a "Web of data".
- The ultimate goal of the Web of data is to enable computers to do more useful work and to develop systems that can support trusted interactions over the network.
- The term "Semantic Web" now refers to W3C's vision of the Web of linked data.
- Semantic Web technologies enable people to create data stores on the Web, build vocabularies, and write rules for handling data.

Resource Description Framework (RDF)

- RDF is a framework for representing information in the Web.
- Facilitate data merging even if the underlying schemas differ.
- The core structure of the abstract syntax is a set of triples, each consisting of a subject, a predicate and an object.
- A set of such triples is called an RDF graph. Each triple is represented as a node-arc-node link.

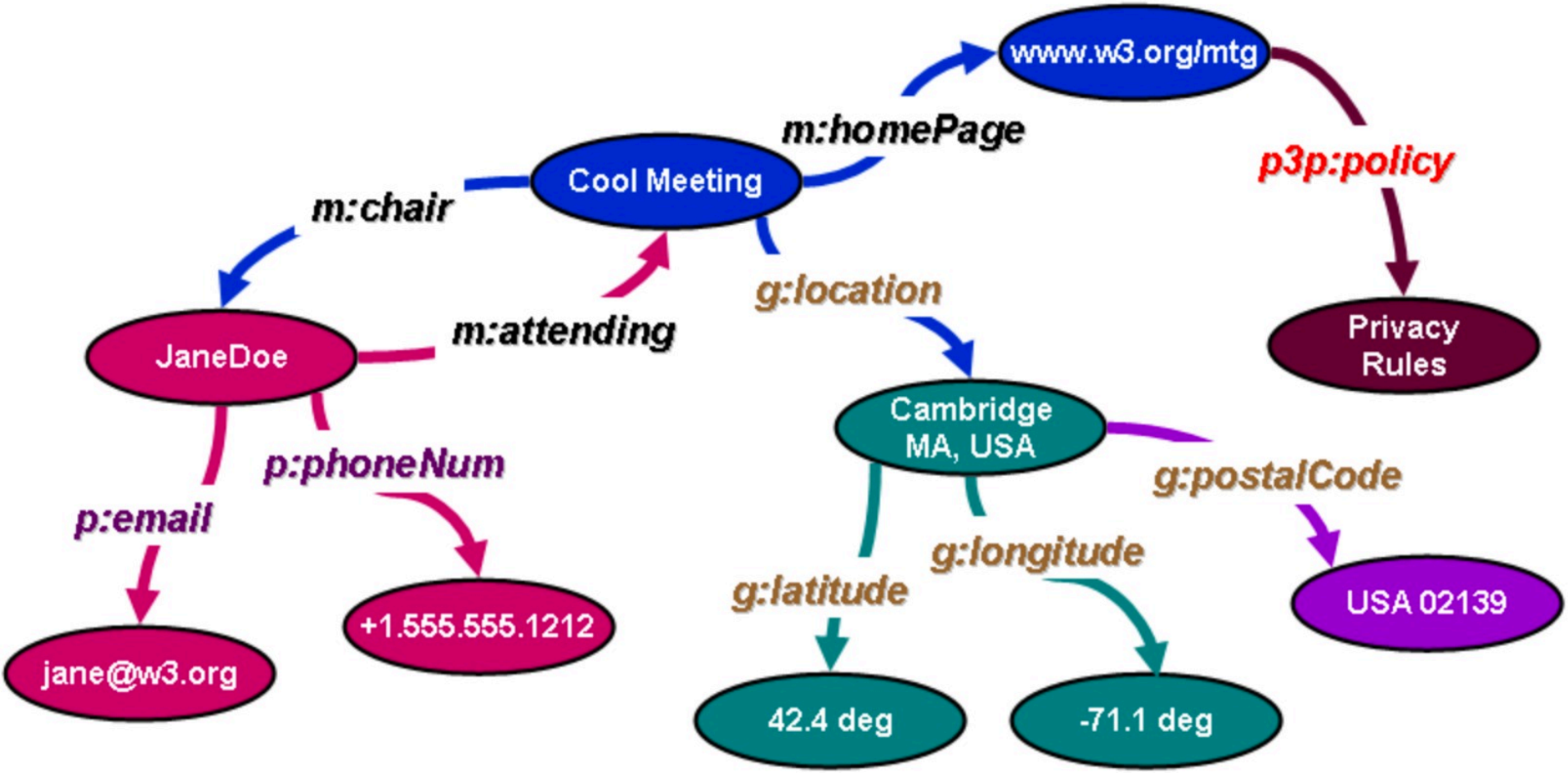


Most of the Current Web



https://www.w3.org/2007/Talks/0130-sb-W3CTechSemWeb/

With RDF Data



https://www.w3.org/2007/Talks/0130-sb-W3CTechSemWeb/

RDFS and OWL 2

- RDFS: Resource Description Framework Schema
  - Provides basic capabilities for describing RDF vocabularies
- OWL: Web Ontology Language
  - Built on top of RDFS and RDF
  - Provides additional capabilities in knowledge representation
- OWL is a Semantic Web language designed to represent rich and complex knowledge about things, groups of things, and relations between things.
  - a computational logic-based language
  - knowledge expressed in OWL can be exploited by computer programs, e.g., to verify the consistency of that knowledge or to make implicit knowledge explicit.
  - RDF and OWL documents, known as ontologies

Ontology

- "An ontology is an explicit specification of a conceptualisation."
- "While a conceptual schema defines relations on data, an ontology defines terms to represent knowledge."
  - Data: ground atomic facts
  - Knowledge: expressible in logical sentences with existentially and universally quantified variables.
- In the context of Semantic Web, ontology and knowledge base sometimes are used interchangeably (but with some subtle differences).

Gruber, T. R. (1993). A Translation Approach to Portable Ontology Specifications. Knowledge Acquisition

Linked Data

- huge amount of data on the Web available in a standard format
- reachable and manageable by Semantic Web tools
- relationships among data should be made available
- collection of interrelated datasets on the Web is also referred to as linked data.
- makes easy either conversion or on-the-fly access to existing databases (relational, XML, HTML, etc).
- setup query endpoints to access that data more conveniently
  - W3C provides a palette of technologies (RDF, GRDDL, POWDER, RDFa, the upcoming R2RML, RIF, SPARQL) to get access to the data.

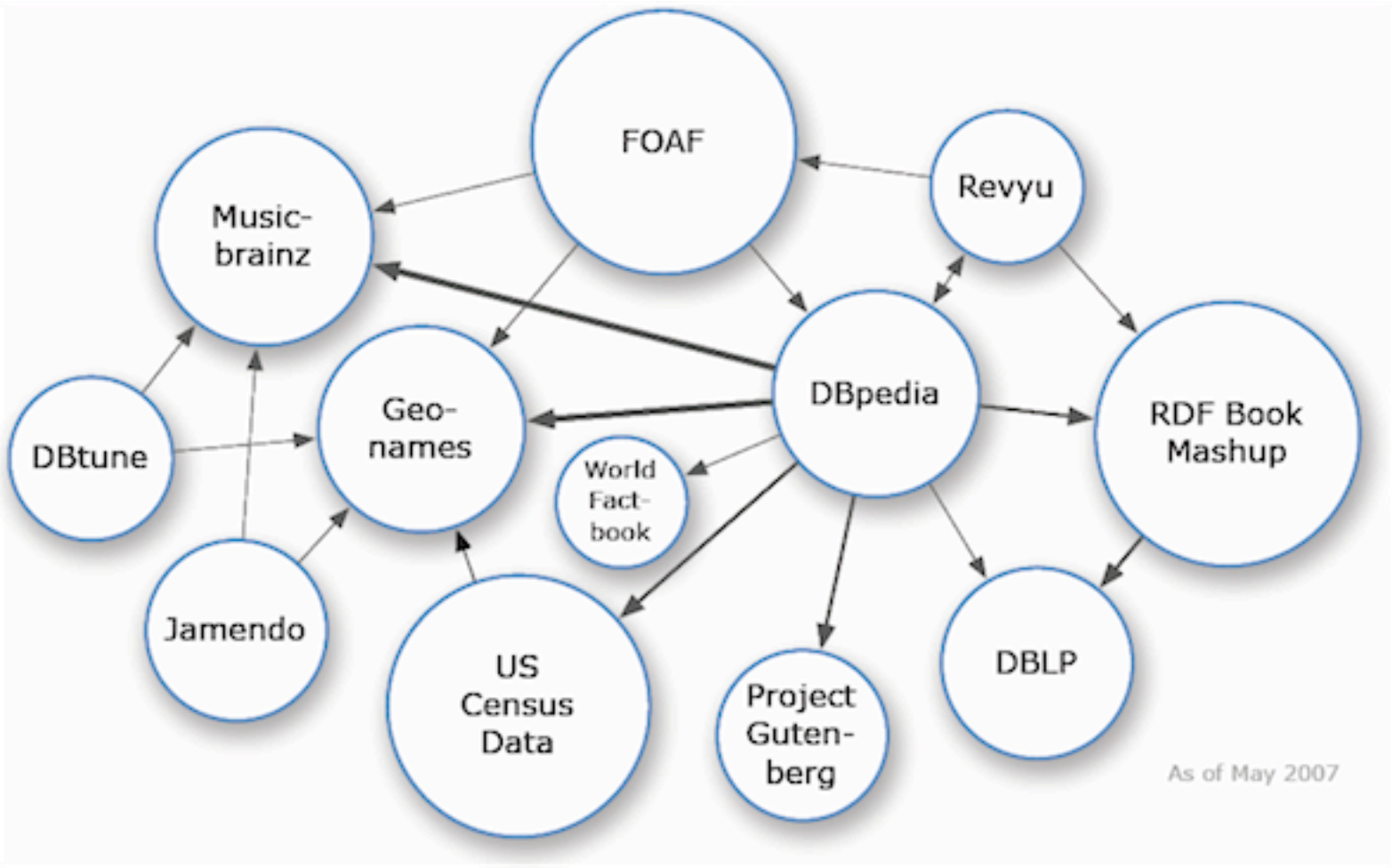
https://www.w3.org/standards/semanticweb/data

Linked Data - Connect Distributed Data across the Web

- Linked data design principles:
  - Use URIs as names for things
  - Use HTTP URIs so that people can look up those names.
  - When someone looks up a URI, provide useful information, using the standards (RDF*, SPARQL)
  - Include links to other URIs, so that they can discover more things.
- *Tutorials on publishing linked data available in the references at the end of lecture.*

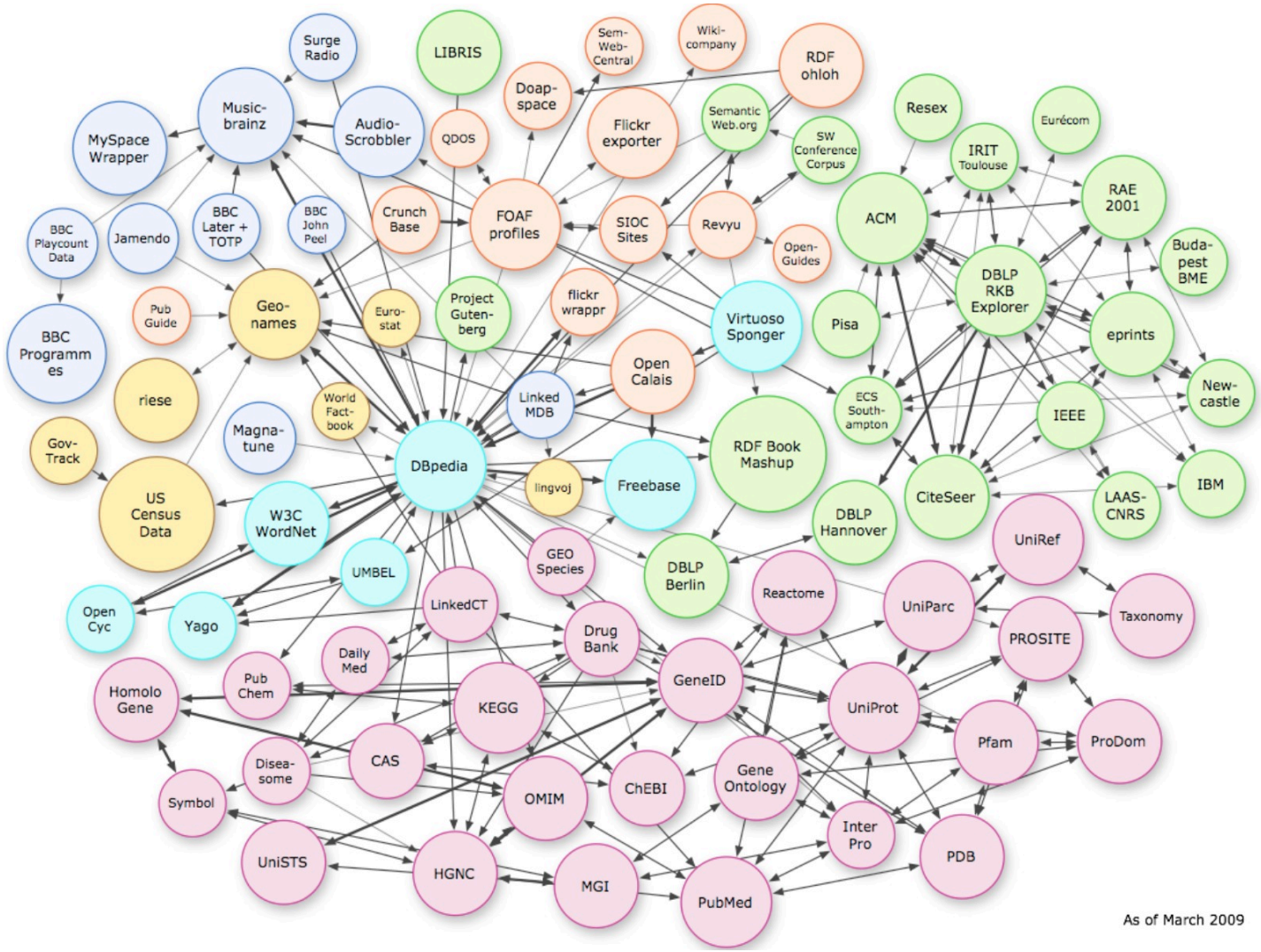
https://www.w3.org/DesignIssues/LinkedData.html

Linked Open Data by 2007



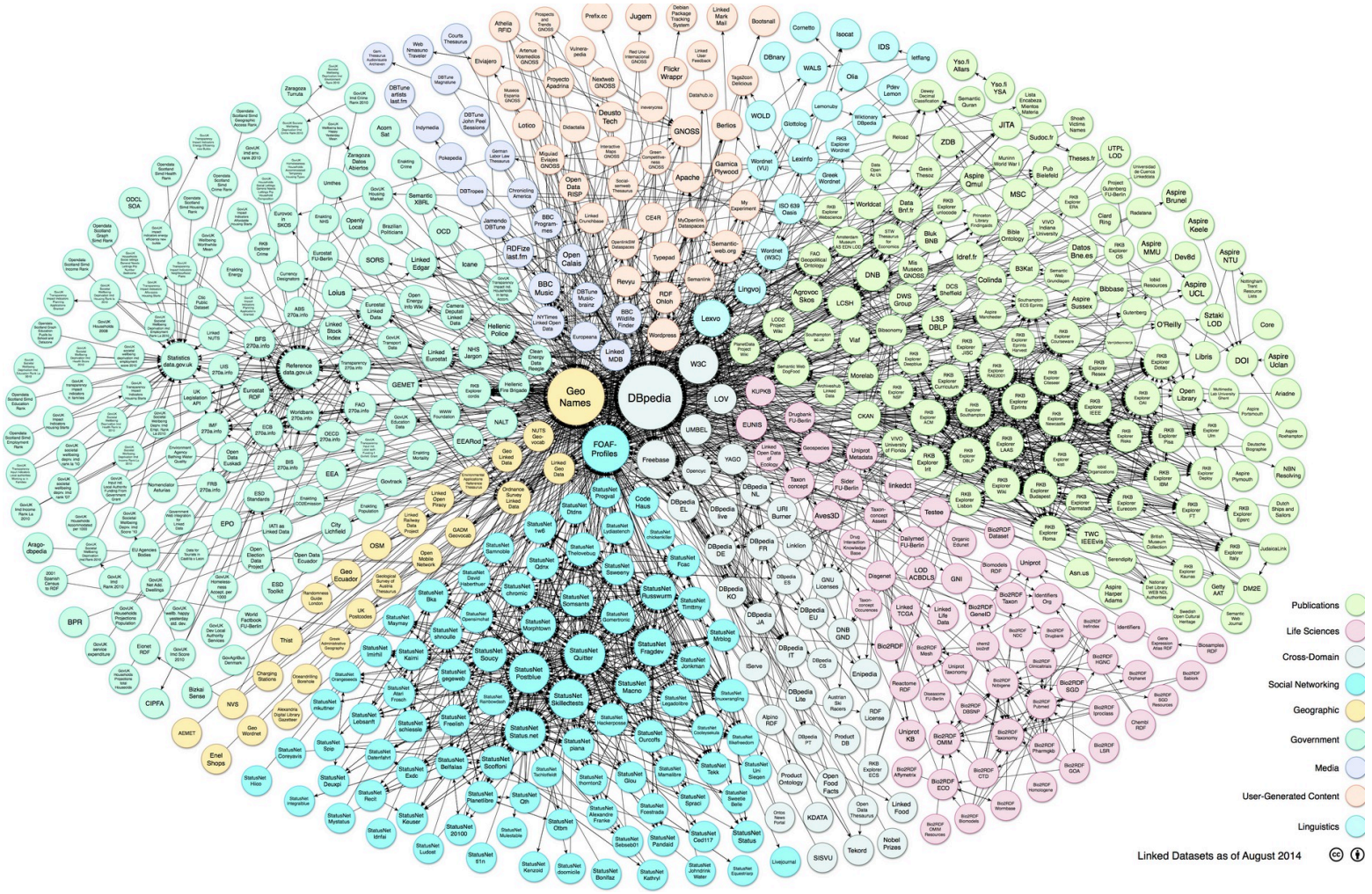
http://lod-cloud.net/

Linked Open Data by 2009



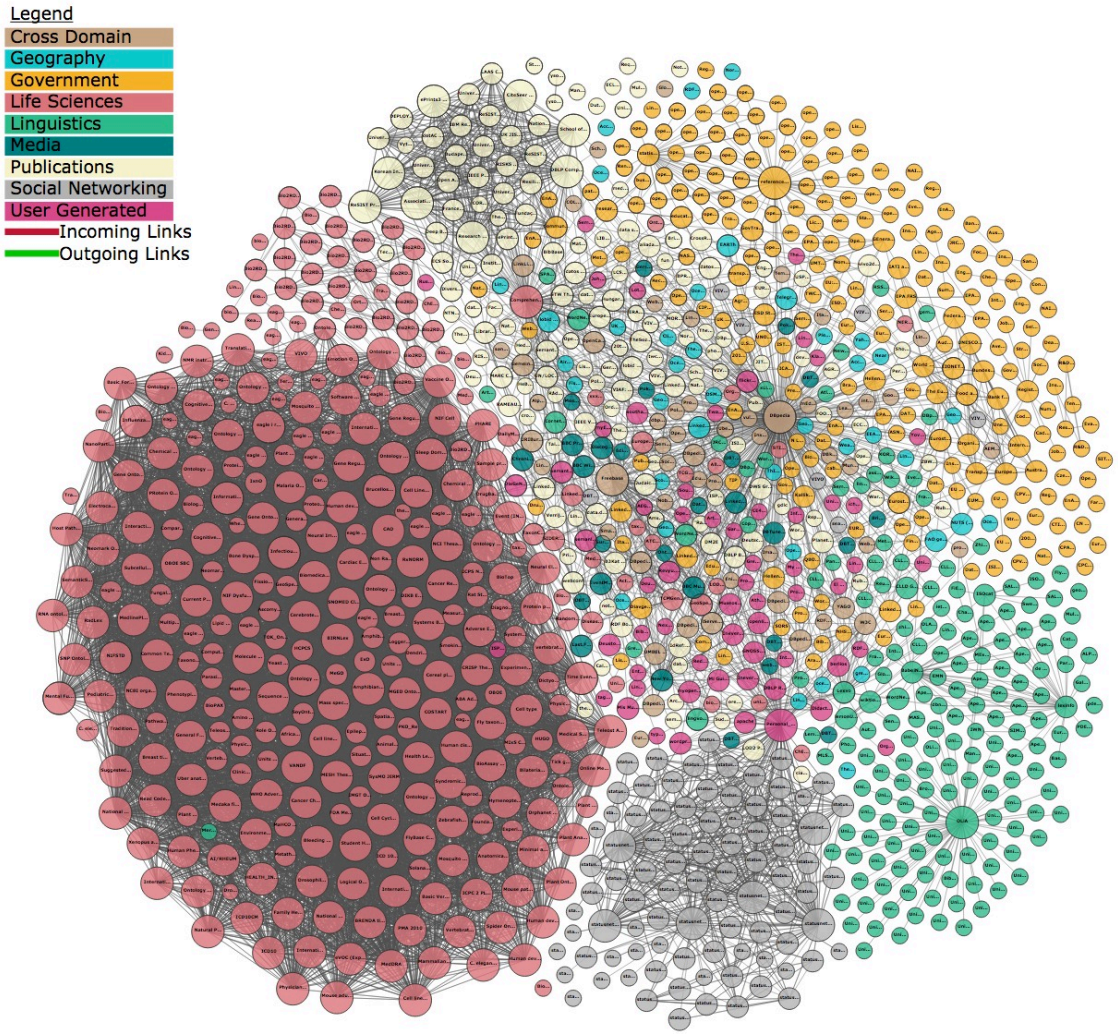
http://lod-cloud.net/

Linked Open Data by 2014



http://lod-cloud.net/

Linked Open Data by 2017



http://lod-cloud.net/

SPARQL: Query the Web of Data

- SPARQL can be used to express queries across diverse data sources
  - whether the data is stored natively as RDF or viewed as RDF via middleware.
- SPARQL contains capabilities for querying required and optional graph patterns
  - along with their conjunctions and disjunctions.
- SPARQL also supports extensible value testing and constraining queries by source RDF graph.
- The results of SPARQL queries can be results sets or RDF graphs.

SPARQL: Query the Web of Data cont'd

- SPARQL queries are based on (triple) patterns.
  - RDF can be seen as a set of relationships among resources (i.e., RDF triples);
  - SPARQL queries provide one or more patterns against such relationships.
  - These triple patterns are similar to RDF triples, except that one or more of the constituent resource references are variables.
- A SPARQL engine would return the resources for all triples that match these patterns.
  - A tutorial for querying semantic Web data: http://www.linkeddatatools.com/querying-semantic-data

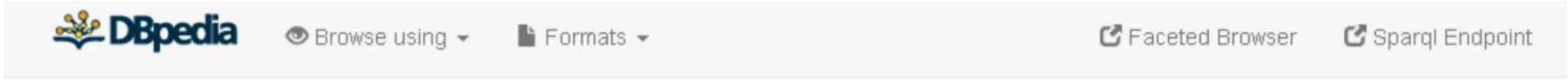
Using SPARQL to query the DBPedia 1

- DBPedia SPARQL endpoint:
  http://dbpedia.org/sparql
- Retrieve country instances:

```
SELECT ?country
   WHERE
      {?country rdf:type dbo:Country
      } LIMIT 100
```

Using SPARQL to query the DBPedia 1 - Results

Using SPARQL to query the DBPedia 1 – Results cont'd

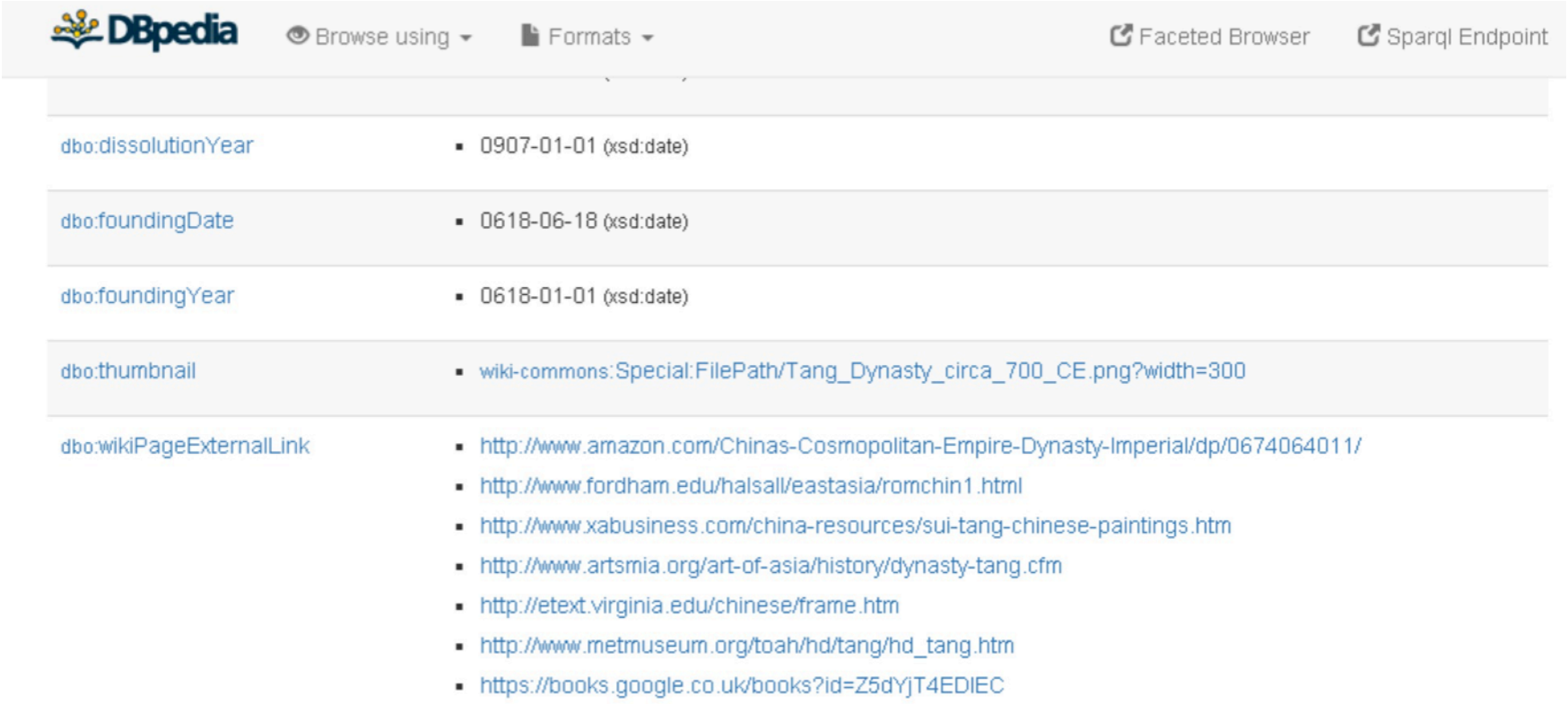


About: 唐朝

An Entity of Type : populated place, from Named Graph : http://dbpedia.org, within Data Space : dbpedia.org

"唐"重定向至此。關于唐和唐朝的其他意思，詳見唐 (消歧義)和唐朝 (消歧义)。 唐朝（618年-907年）共历289年，21位皇帝。由唐高祖李淵所建立，與隋朝合稱隋唐。唐室出身自關隴世族，先祖李虎在南北朝的西魏是八柱国之一，封为唐国公。其後代李淵爲隋朝晋阳（在今山西太原西南）留守，在隋末民變時出兵入關中以爭奪天下，於618年受隋恭帝杨侑禪位建國唐朝，在唐朝統一戰爭中統一了天下。唐朝定都長安（今陕西西安）。并设东都洛陽、北都晋阳等陪都。 唐朝的疆域廣大但時常變動，630年就超过隋朝极盛时的版图。唐朝也是自秦汉以来，第一个不使用前朝所筑长城及不筑长城的统一王朝。其鼎盛时爲7世纪，當時中亚的綠洲地帶受唐朝支配。其最大範圍南至罗伏州（今越南河静）、北括玄阙州（今俄罗斯安加拉河流域）、西及安息州（今乌兹别克斯坦布哈拉）、东临哥勿州（今吉林通化）的辽阔疆域，国土面积达1076万平方公里。中唐後漠北、西域的領地相繼失去，到晚唐時衰退到等同中國本土的大小，但仍然保有河套地區及河西走廊。唐代天宝十三年（754年）户口统计为五千二百八十八万四百八十八人，不过许多学者考虑到当时统计不严，存在大量没有计入统计的瞒报户

Using SPARQL to query the DBPedia 1 – Results cont'd



Using SPARQL to query the DBPedia 2

- Retrieve artist instances whose birth place is a country contains the string of "United Kingdom".

```
PREFIX dbprop: <http://dbpedia.org/property/>
PREFIX dbpedia-owl: <http://dbpedia.org/ontology/>

SELECT ?artist ?place
   WHERE
      {
         ?artist rdf:type dbpedia-owl:Artist.
         ?artist dbprop:birthPlace ?place.
         ?place rdf:type dbpedia-owl:Country.
         ?place rdfs:label ?label.
         FILTER regex(?label, "^United Kingdom").
      }
```

Using SPARQL to query the DBPedia 2 – results

artist	place
http://dbpedia.org/resource/Gary_Frank	http://dbpedia.org/resource/United_Kingdom
http://dbpedia.org/resource/The_Etherington_Brothers	http://dbpedia.org/resource/United_Kingdom
http://dbpedia.org/resource/Reg_Smythe	http://dbpedia.org/resource/United_Kingdom
http://dbpedia.org/resource/Ed_Furness	http://dbpedia.org/resource/United_Kingdom
http://dbpedia.org/resource/Rich_Johnston	http://dbpedia.org/resource/United_Kingdom
http://dbpedia.org/resource/Alan_Davis	http://dbpedia.org/resource/United_Kingdom
http://dbpedia.org/resource/Ho_Che_Anderson	http://dbpedia.org/resource/United_Kingdom
http://dbpedia.org/resource/Rhydian_Vaughan	http://dbpedia.org/resource/United_Kingdom
http://dbpedia.org/resource/Simon_Oliver	http://dbpedia.org/resource/United_Kingdom
http://dbpedia.org/resource/Mike_McMahon_(comics)	http://dbpedia.org/resource/United_Kingdom
http://dbpedia.org/resource/Nick_Landau	http://dbpedia.org/resource/United_Kingdom
http://dbpedia.org/resource/Steve_MacManus	http://dbpedia.org/resource/United_Kingdom
http://dbpedia.org/resource/Steve_Moore_(comics)	http://dbpedia.org/resource/United_Kingdom
http://dbpedia.org/resource/Alan_Martin_(writer)	http://dbpedia.org/resource/United_Kingdom
http://dbpedia.org/resource/Rufus_Dayglo	http://dbpedia.org/resource/United_Kingdom
http://dbpedia.org/resource/Jason_Chan_Chi-san	http://dbpedia.org/resource/United_Kingdom

Using SPARQL to query the DBPedia 2 – results cont'd

DBpedia Browse using Formats Faceted Browser

dbo:birthDate	1972-11-21 (xsd:date)
dbo:birthPlace	dbr:Gloucester
dbo:imdbId	1262679
dbo:nationality	dbr:British_citizenship
dbo:thumbnail	wiki-commons:Special:FilePath/Rich_Johnston,_2007.jpg?width=300
dbo:wikiPageExternalLink	http://www.dynamicforces.com/htmlfiles/tommy.html?showhistory=ok http://www.sequart.com/interviews/index.php?interview=740 http://www.comicbookresources.com/?page=column&id=11 http://www.richandmark.com http://www.2000adreview.co.uk/features/interviews/2006/johnston/rich-johnston.shtm http://www.bleedingcool.com

End of Lecture

- Summary
  - Semantic Web, ontologies
  - RDF, OWL
  - Linked Open Data
  - Query the Web of Data

- Reading
  - See references next slides

References

- Berners-Lee, T.; Hendler, J.; Lassila, O. (2001). "The Semantic Web". Scientific American. 2841 (5): 34.
- Nigel Shadbolt; Wendy Hall; Tim Berners-Lee (2006). "The Semantic Web Revisited" (PDF). IEEE Intelligent Systems. Retrieved April 13, 2007.
- Lee Feigenbaum (May 1, 2007). "The Semantic Web in Action". Scientific American. Retrieved February 24, 2010.
- RDF premier, https://www.w3.org/TR/rdf11-concepts/
- Gruber, T. R. (1993). A Translation Approach to Portable Ontology Specifications. Knowledge Acquisition, 5(2), 199-220.
- Linked data, https://www.w3.org/standards/semanticweb/data
- Semantic Web use cases and case studies, https://www.w3.org/2001/sw/sweo/public/UseCases/
- Linked data design issues, https://www.w3.org/DesignIssues/LinkedData.html
- Tom Heath and Christian Bizer, Linked Data: Evolving the Web into a Global Data Space, http://linkeddatabook.com/
- Chris Bizer, Richard Cyganiak, How to Publish Linked Data on the Web (Tutorial), http://wifo5-03.informatik.uni-mannheim.de/bizer/pub/LinkedDataTutorial/
- SPARQL, https://www.w3.org/2001/sw/wiki/SPARQL
- Querying Semantic Data, http://www.linkeddatatools.com/querying-semantic-data

Database Development and Design (CPT201)

Lecture 9: Database Connectivity

Dr. Wei Wang
Department of Computing

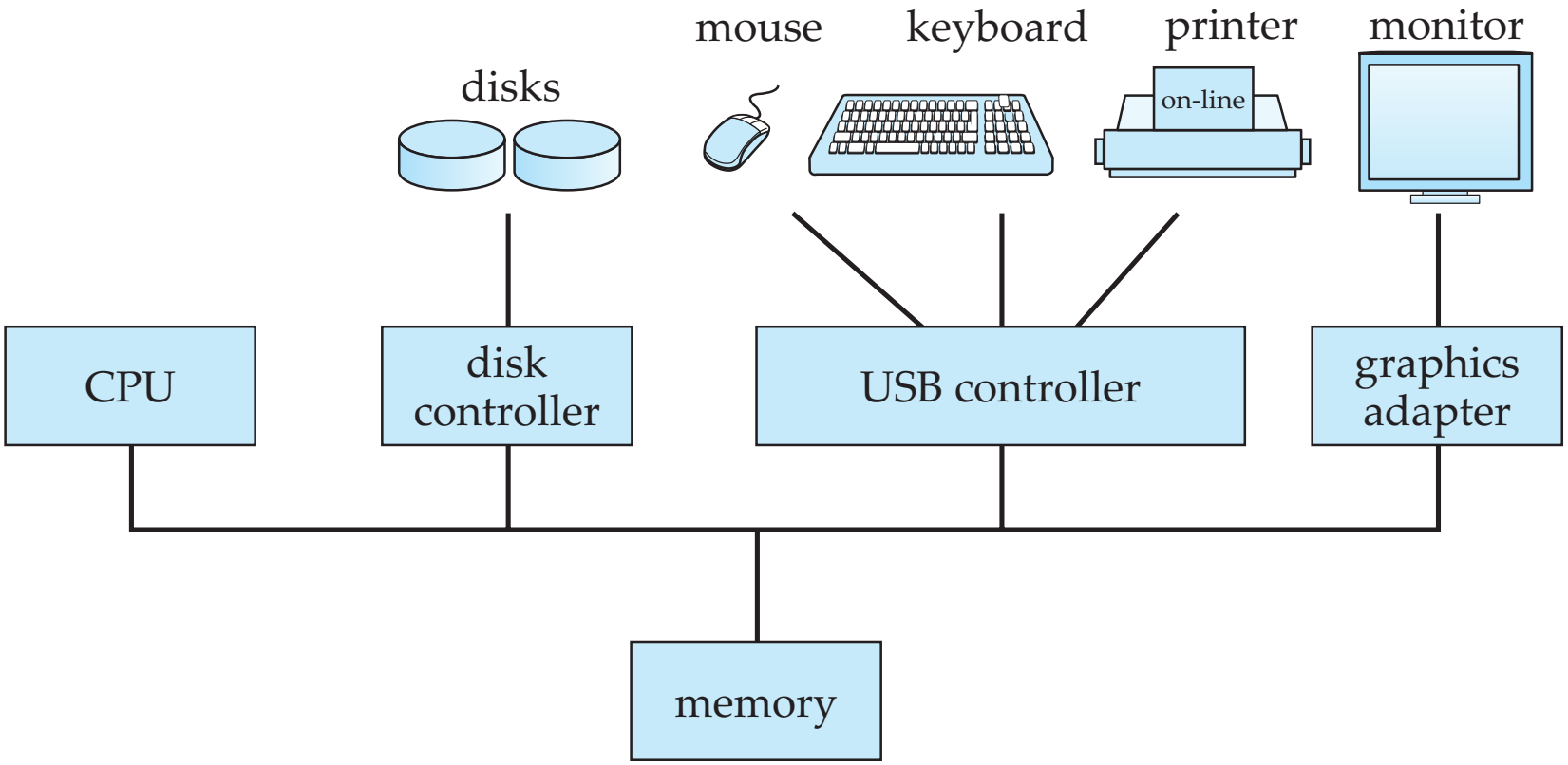
Learning Outcomes

- Centralised and Client-Server Systems
- Server System Architectures
- Database Connectivity
  - ODBC
  - JDBC
  - Java Programming wit JDBC

Centralised Systems

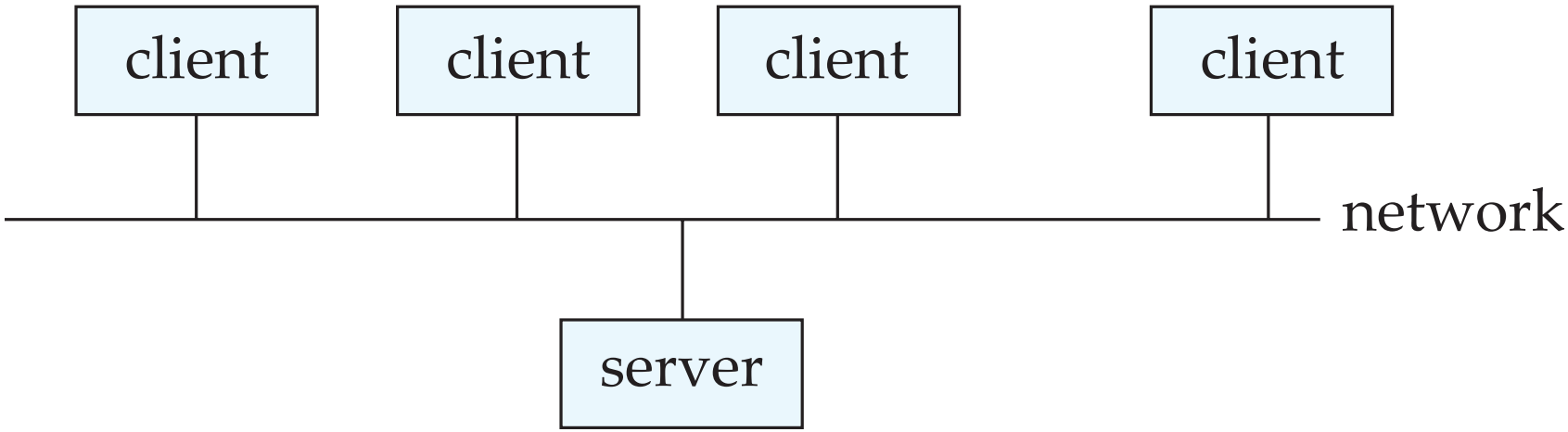
- Run on a single computer system and do not interact with other computer systems.
  - General-purpose computer system: one to a few CPUs and a number of device controllers that are connected through a common bus that provides access to shared memory.
  - Single-user system (e.g., personal computer or workstation): desk-top unit, single user, usually has only one CPU and one or two hard disks; the OS may support only one user.
  - Multi-user system: more disks, more memory, multiple CPUs, and a multi-user OS. Serve a large number of users who are connected to the system vie terminals. Often called *server* systems.

A Centralised Computer System



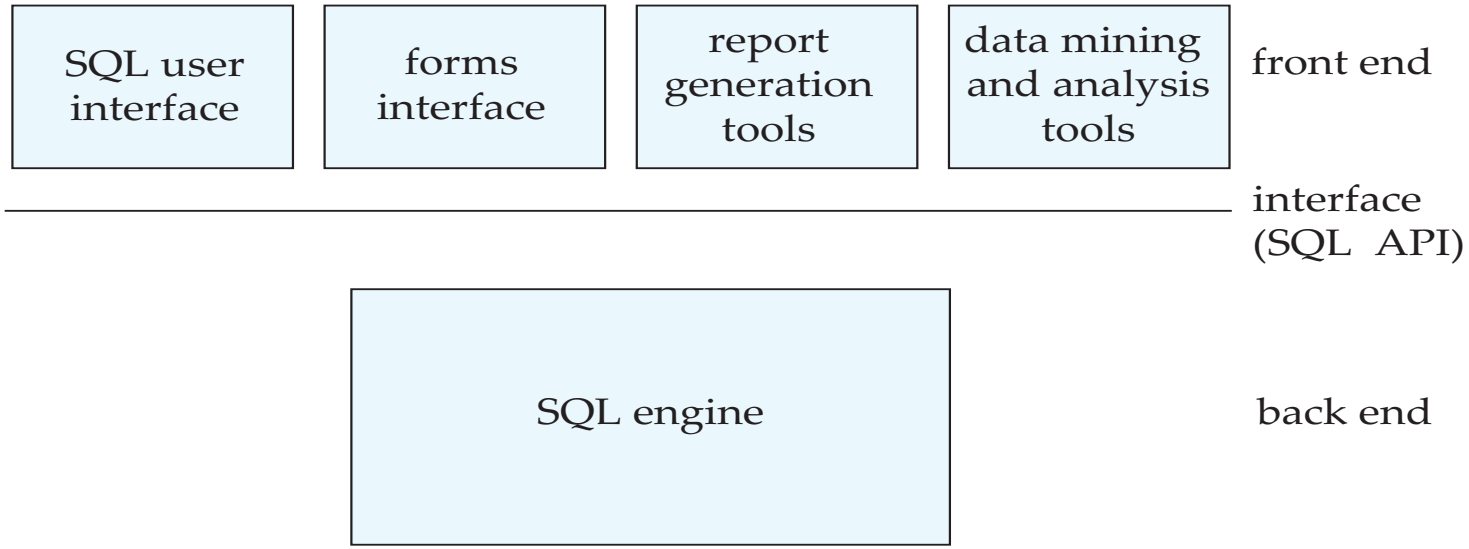
Client-Server Systems

- Server systems satisfy requests generated at client systems.



Client-Server Systems cont'd

- Database functionality can be divided into:
  - **Back-end**: manages access structures, query evaluation and optimisation, concurrency control and recovery, etc.
  - **Front-end**: consists of tools such as *forms*, *report-writers*, and graphical user interface facilities, etc.
- The interface between the front-end and the back-end is through SQL or through an application program interface.



Client-Server Systems cont'd

- Advantages of replacing mainframes with networks of workstations or personal computers connected to back-end server machines:
  - better functionality for the cost
  - flexibility in locating resources and expanding facilities
  - better user interfaces
  - easier maintenance

Server System Architecture

- Server systems can be broadly categorised into two kinds:
  - **transaction servers** widely used in relational database systems
  - **data servers** used in object-oriented database systems

Transaction Servers

- Also called **query server** systems or SQL *server* systems
  - Clients send requests to the server,
  - Transactions are executed at the server,
  - Results are shipped back to the client.
- Requests are specified in SQL, and communicated to the server through a *remote procedure call* (RPC) mechanism.
- Transactional RPC allows many RPC calls to form a transaction.
- *Open Database Connectivity* (ODBC) is a C language application program interface standard from Microsoft for connecting to a server, sending SQL requests, and receiving results.
- JDBC standard is similar to ODBC, but for Java.

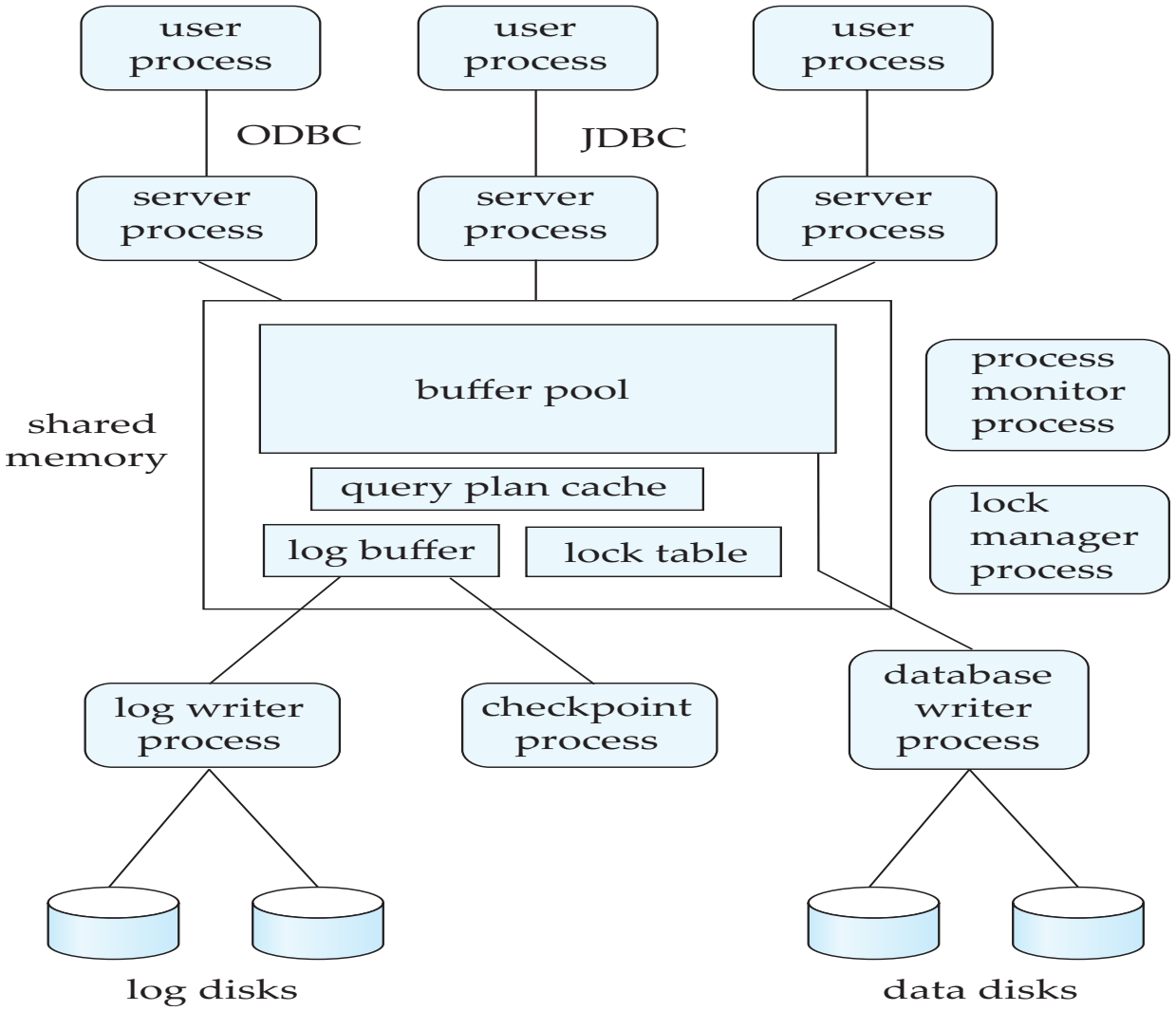
Transaction Server Process Structure

- A typical transaction server consists of multiple processes accessing data in shared memory.
- Server processes
  - receive user queries (transactions), execute them and send results back
  - may be **multithreaded** to support multiple user queries concurrently
  - may be multiple multithreaded server processes
- Lock manager process
- Database writer process
  - Output modified buffer blocks to disks continually

Transaction Server Processes cont'd

- Log writer process
  - Server processes simply add log records to log record buffer
  - Log writer process outputs log records to stable storage.
- Checkpoint process
  - Performs periodic checkpoints
- Process monitor process
  - Monitors other processes, and takes recovery actions if any of the other processes fail
    - e.g., aborting any transactions being executed by a server process and restarting it

Transaction System Processes (Cont.)



Transaction System Processes cont'd

- Shared memory contains shared data
  - Buffer pool
  - Lock table
  - Log buffer
  - Cached query plans (reused if same query submitted again)
- All database processes can access shared memory
- To ensure that no two processes are accessing the same data structure at the same time, databases systems implement **mutual exclusion** using either
  - Operating system semaphores
  - Atomic instructions such as test-and-set
- To avoid overhead of inter-process communication for lock request/grant, each database process operates directly on the lock table
  - instead of sending requests to lock manager process
- Lock manager process still used for deadlock detection

Data Servers

- Used in high-speed LANs, in cases where
  - The clients are comparable in processing power to the server
  - The tasks to be executed are computationally intensive.
- Data is shipped to clients where processing is performed, and then shipped results back to the server.
- This architecture requires full back-end functionality at the clients.
- Used in many object-oriented database systems
- Issues:
  - Page-Shipping versus Item-Shipping
  - Locking
  - Data Caching
  - Lock Caching

Data Servers cont'd

- **Page-shipping** versus **item-shipping**
  - Smaller unit of shipping VS more messages
  - Worth **prefetching** related items along with requested item
  - Page shipping can be thought of as a form of prefetching
- Locking
  - Overhead of requesting and getting locks from server is high due to message delays.
  - Can grant locks on requested and prefetched items; with page shipping, transaction is granted lock on whole page.
  - Locks on a prefetched item can be called back by the server, and returned by client transaction if the prefetched item has not been used.
  - Locks on the page can be **deescalated** to locks on items in the page when there are lock conflicts. Locks on unused items can then be returned to server.

Data Servers cont'd

- **Data Caching**
  - Data can be cached at client even in between transactions
  - But check that data is up-to-date before it is used (**cache coherency**)
  - Check can be done when requesting lock on data item
- **Lock Caching**
  - Locks can be retained by client system even in between transactions
  - Transactions can acquire cached locks locally, without contacting server
  - Server **calls back** locks from clients when it receives conflicting lock request.
  - Client returns lock once no local transaction is using it.

Accessing Database from Applications

- SQL commands can be called from within a host language (e.g., C++ or Java) program.
  - SQL statements can refer to host variables (including special variables used to return status).
  - Must include statement to *connect* to right database.
- Two main integration approaches:
  - Embed SQL in the host language (e.g., Pro*C, Embedded SQL, SQLJ)
  - Create special API (Call Level Interface) to call SQL commands (eg: JDBC, ODBC, PHP)

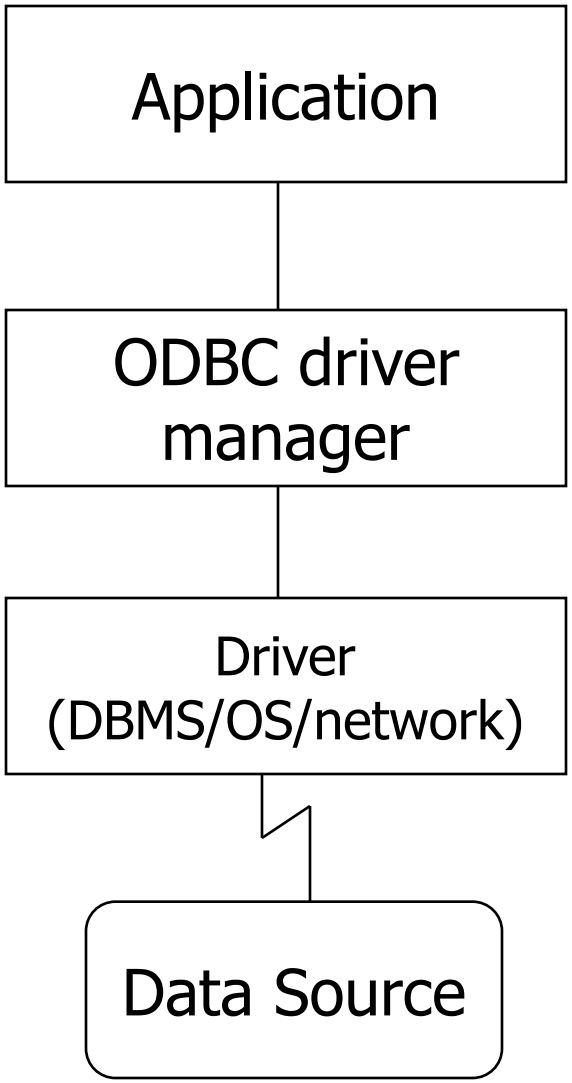
What is ODBC?

- ODBC is short for Open Database Connectivity
  - A standard open application programming interface (API) for accessing a database.
  - SQL Access Group, chiefly Microsoft, in 1992
  - By using ODBC statements in a program, you can access files in a number of different databases, including Access, dBase, DB2, Excel, and Text.
  - It allows programs to use SQL requests that will access databases without having to know the proprietary interfaces to the databases.
  - ODBC handles the SQL request and converts it into a request the individual database system understands.

More on ODBC

- You need:
  - the ODBC software, and
  - a separate module or *driver* for each database to be accessed (library that is dynamically connected to the application).
- Driver masks the heterogeneity of DBMS operating system and network protocol.
  - e.g. (Sybase, Windows/NT, Novell driver)

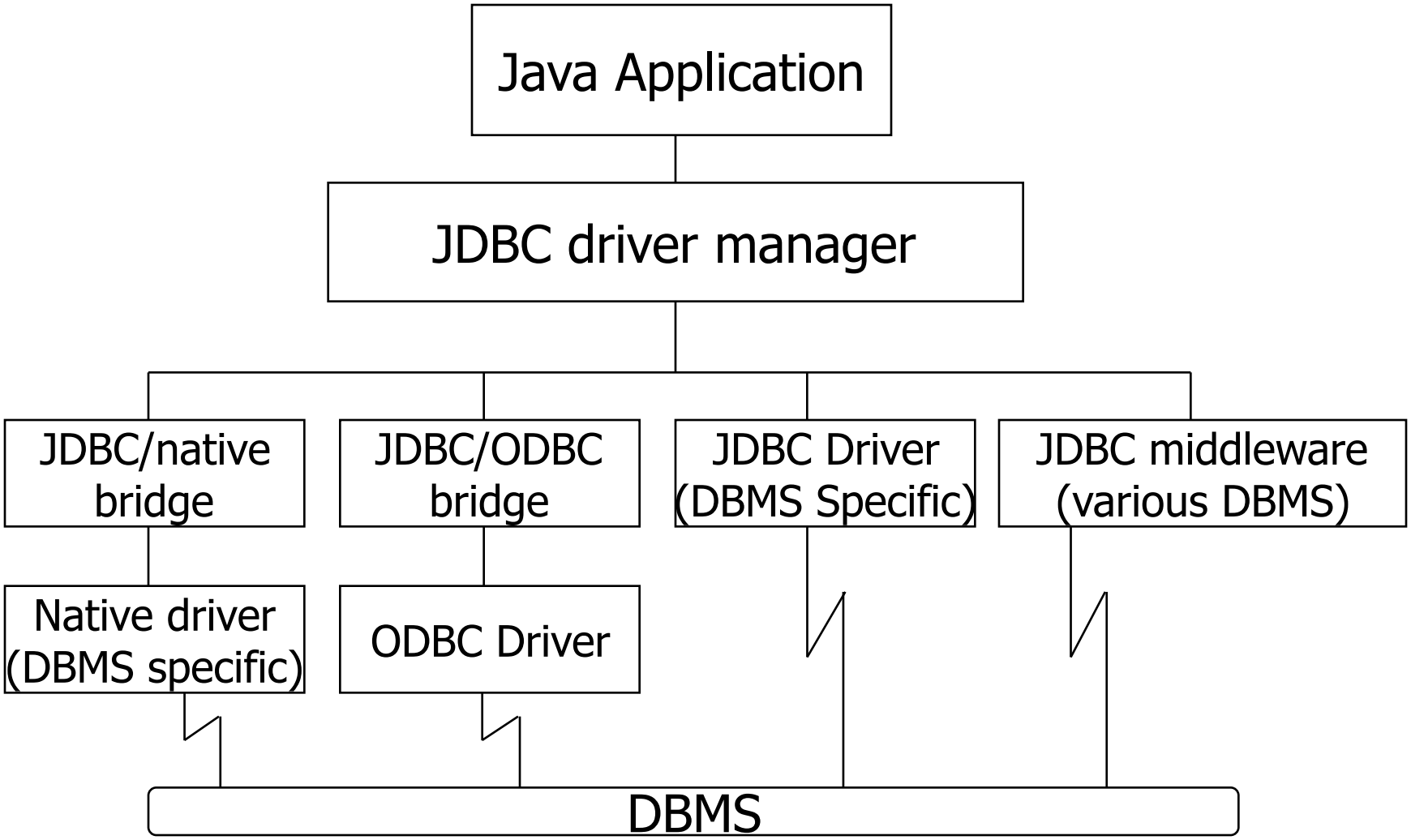
ODBC Architecture



What is JDBC?

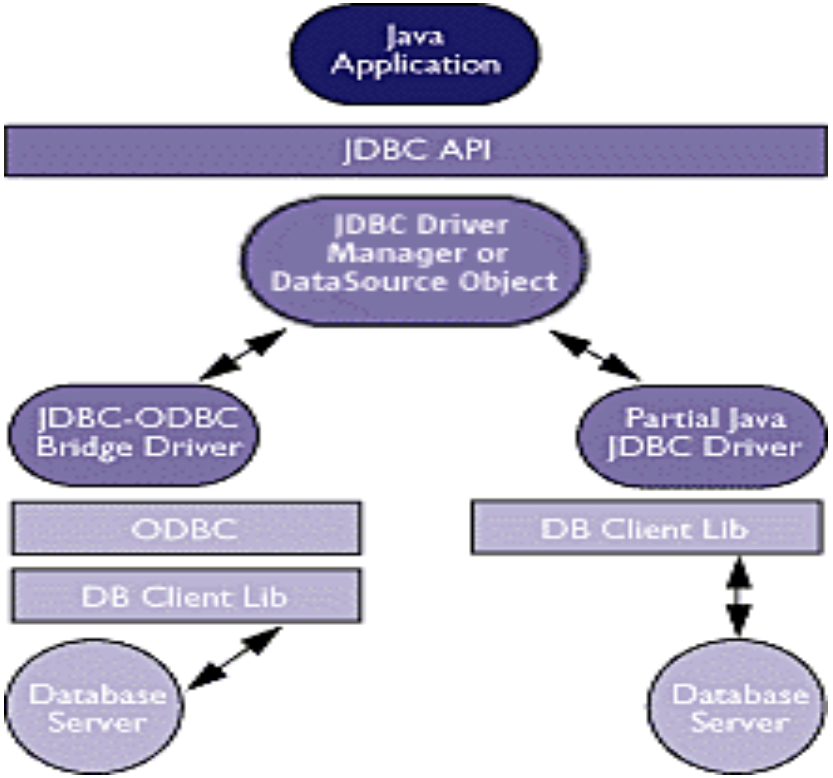
- JDBC is short for Java Database Connectivity
  - is a Java API for connecting programs written in Java to the data in relational databases.
  - consists of a set of classes and interfaces written in the Java programming language.
  - provides a standard API for tool/database developers and makes it possible to write database applications using a pure Java API.
  - The standard defined by Sun Microsystems, allowing individual providers to implement and extend the standard with their own JDBC drivers.
- JDBC:
  - establishes a connection with a database
  - sends SQL statements
  - processes the results

JDBC Architectures



JDBC connectivity using ODBC drivers

- With a small "bridge" program, you can use the JDBC interface to access ODBC-accessible databases.



JDBC vs ODBC

- ODBC is used between applications
- JDBC is used by Java programmers to connect to databases
- With a small "bridge" program, you can use the JDBC interface to access ODBC-accessible databases.
- JDBC allows SQL-based database access for EJB (Enterprise JavaBeans) persistence and for direct manipulation from CORBA and other server objects.

JDBC API

- The JDBC API supports both two-tier and three-tier models for database access.
- Two-tier model - a Java applet or application interacts directly with the database.
- Three-tier model - introduces a middle-level server for execution of business logic:
  - the middle tier to maintain control over data access.
  - user can employ an easy-to-use higher-level API which is translated by the middle tier into the appropriate low-level calls.

The JDBC Steps

- Importing Packages
- Registering the JDBC Drivers
- Opening a Connection to a Database
- Creating a Statement Object
- Executing a Query and Returning a Result Set Object
- Processing the Result Set
- Closing the Result Set, Statement and Connection Objects

1: Importing Packages

```
// Program name: JDBCExample.java
// Purpose: Basic selection using prepared statement
//

//Import packages

    import java.sql.*;  //JDBC packages
    import java.math.*;
    import java.io.*;
    import oracle.jdbc.driver.*;
```

2: Registering JDBC Drivers

```
class JDBCExample {

public static void main (String args []) throws SQLException
    {

//Load Oracle driver
DriverManager.registerDriver (new
    oracle.jdbc.driver.OracleDriver());
```

3: Opening connection to a Database

```
//Prompt user for username and password

  String user;
  String password;

  user = readEntry("username: ");
  password = readEntry("password: ");

//Connect to the local database
  Connection conn = DriverManager.getConnection
   ("jdbc:oracle:thin:@aardvark:1526:teach",
        user,
        password);
```

4. Creating a Statement Object

```
// Query the hotels table for resort = 'palma nova'

PreparedStatement pstmt = conn.prepareStatement
    ("SELECT hotelname, rating FROM hotels WHERE
    trim(resort) = ?");
pstmt.setString(1, "palma nova");
```

5. Executing a Query, Returning a Result Set Object & 6. Processing the Result Set

```
ResultSet rset = pstmt.executeQuery();

//Print query results
while (rset.next ())
    System.out.println (rset.getString(1)+" "+
                        rset.getString(2));
```

7. Closing the Result Set and Statement Objects & 8. Closing the Connection

```
// close the result set, statement, and the connection
        rset.close();
        pstmt.close();
        conn.close();
    }
```

Mapping Data Types

- There are data types specified to SQL that need to be mapped to Java data types if the user expects Java to be able to handle them.
- Conversion falls into three categories:
  - SQL type to Java direct equivalents
    - SQL INTEGER direct equivalent of Java int data type
  - SQL type can be converted to a Java equivalent.
    - SQL CHAR, VARCHAR, and LONGVARCHAR can all be converted to the Java String data type
  - SQL data type is unique and requires a special Java data class object to be created specifically for their SQL equivalent
    - SQL DATE converted to the Java Date object that is defined in java.Date especially for this purpose

End of Lecture

- Summary
  - Centralised and Client-Server Systems
  - Server System Architectures
  - Database Connectivity: ODBC, JDBC, Java Programming wit JDBC

- Reading
  - Textbook 6th edition, chapters 5.1, 9.1, 9.2, 17.1, and 17.2
  - Textbook 7th edition, chapters 5.1, 9.1, 9.2, 20.2, and 20.3

Database Development and Design (CPT201)

Lecture 10: Introduction to Object-Oriented Databases

Dr. Wei Wang

Department of Computing

Learning Outcomes

- Complex Data Types and Object Orientation
- Structured Data Types and Inheritance in SQL
- Table Inheritance
- Array and Multiset Types in SQL
- Object Identity and Reference Types in SQL
- Implementing O-R Features
- Persistent Programming Languages
- Comparison of Object-Oriented and Object-Relational Databases

Object-Relational Data Models

- Extend the relational data model by including object orientation and constructs to deal with added data types.
- Allow attributes of tuples to have complex types, including non-atomic values such as nested relations.
- Preserve relational foundations, in particular the declarative access to data, while extending modeling power.
- Upward compatibility with existing relational languages.

Complex Data Types

- Motivation:
  - Permit non-atomic domains (atomic ≡ indivisible)
  - Example of non-atomic domain: set of integers, or set of tuples
  - Allows more intuitive modeling for applications with complex data
- Intuitive definition:
  - allow relations whenever we allow atomic (scalar) values - relations within relations
  - Retains mathematical foundation of relational model
  - Violates first normal form (1NF)

Example of a Nested Relation

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

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

4NF Decomposition of Nested Relation

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

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

authors

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

keywords

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

books4

Complex Types and SQL

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

Structured Types and Inheritance in SQL

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

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

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

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

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

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

Structured Types

- **User-defined row types**

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

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

```
create table person of PersonType
```

- Alternative using **unnamed row types**

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

Methods

- Can add a method declaration with a structured type.

  method *ageOnDate* (*onDate* **date**)

  returns interval year

- Method body is given separately.

  create instance method *ageOnDate* (*onDate* **date**)

  returns interval year

  for *CustomerType*

  begin

  return *onDate* - **self**.*dateOfBirth*;

  end

- We can now find the age of each customer:

  select *name.lastname, ageOnDate* (**current_date**)

  from *customer*

Constructor Functions

- **Constructor functions** are used to create values of structured types

```
create function Name(firstname varchar(20), lastname varchar(20))
returns Name
begin
    set self.firstname = firstname;
    set self.lastname = lastname;
end
```

- To create a value of type *Name*, we use
  new *Name*('John', 'Smith')
- Normally used in insert statements

```
insert into Person values
    (new Name('John', 'Smith),
    new Address('20 Main St', 'New York', '11001'),
    date '1960-8-22');
```

Type Inheritance

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

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

- Using inheritance to define the student and teacher types

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

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

Multiple Type Inheritance

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

  create type *Teaching_Assistant*
  under *Student, Teacher*

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

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

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

Table Inheritance

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

Consistency Requirements for Subtables

- Consistency requirements on subtables and supertables.
  - Each tuple of the supertable (e.g. *people)* can correspond to at most one tuple in each of the subtables (e.g. *students* and *teachers)*
  - Additional constraint in SQL:1999: All tuples corresponding to each other (that is, with the same values for inherited attributes) must be derived from one tuple (inserted into one table).
    - Each entity must have a most specific type
    - We cannot have a tuple in *people* corresponding to a tuple each in *students* and *teachers*

Array and Multiset Types in SQL

- Example of array and multiset declaration:

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

Creation of Collection Values

- Array construction

  array ['Silberschatz',` Korth',` Sudarshan']

- Multisets

  multiset ['computer', 'database', 'SQL']

- To create a tuple of the type defined by the books relation:

  ('Compilers', **array**[` Smith',` Jones'],
  new *Publisher* (` McGraw-Hill',` New York'),
  multiset [` parsing',` analysis'])

- To insert the preceding tuple into the relation books

```
insert into books
values
    ('Compilers', array[` Smith',` Jones'],
            new Publisher (` McGraw-Hill',` New York'),
            multiset [` parsing',` analysis' ]);
```

Querying Collection-Valued Attributes

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

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

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

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

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

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

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

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

Unnesting

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

```
select B.title, A.author, B.publisher.name as pub_name,
       B.publisher.branch as pub_branch, K.keyword
from books as B, unnest(B.author_array ) as A (author ),
     unnest (B.keyword_set ) as K (keyword )
```

- Result relation: *flat_books*

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

Nesting

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

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

- To nest on both authors and keywords:

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

Nesting cont'd

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

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

Object-Identity and Reference Types

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

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

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

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

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

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

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

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

Initialising Reference-Typed Values

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

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

update departments
     set head = (select p.person_id
                        from people as p
                  where name = `John')
          where name = `CS'
```

User Generated Identifiers

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

```
create type Person
(name varchar(20)
 address varchar(20))
ref using varchar(20)

create table people of Person
ref is person_id user generated
```

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

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

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

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

User Generated Identifiers

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

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

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

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

Path Expressions

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

  select *head->name, head->address*
  from *departments*

- An expression such as "head->name" is called a **path expression**
- Path expressions help avoid explicit joins
  - If department head were not a reference, a join of *departments* with *people* would be required to get at the address
  - Makes expressing the query much easier for the user

Implementing O-R Features

- Similar to how E-R features are mapped onto relation schemas
  - Object-relational database systems are basically extensions of existing relational database systems.
  - To minimise changes to the storage-system code (relation storage, indices, etc.), the complex data types can be translated to the simpler type system of relational databases.
- Subtable implementation
  - Each table stores primary key and those attributes defined in that table, or
  - Each table stores both locally defined and inherited attributes

Why Persistent Programming Language?

- Persistent programming language is a programming language extended with constructs to handle persistent data.
- Why need persistent programming language ?
  - Access to a database is only one component of any real-world application.
  - Data-manipulation language like SQL is effective for accessing data.
  - But programming language is required for implementing other components such as user interfaces or communication with other computers.
- The traditional way of interfacing database languages to programming languages is by embedding SQL within the programming language.

Difference between two Languages

- Persistent programming languages can be distinguished from languages with embedded SQL in at least two ways:
  - With an embedded language, the type system of the host language usually differs from the type system of the data-manipulation language. The programmer is responsible for any type conversions between the host language and SQL. The programmer using an embedded query language is responsible for writing explicit code to fetch/store data from/to databases.
  - In contrast, in a persistent programming language, the programmer can manipulate persistent data without writing code explicitly.

Persistent Programming Languages

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

Object Identity and Pointers

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

Object-Relational Mapping

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

Comparison of Databases

- **Relational systems**
  - simple data types, powerful query languages, high protection.
- **Persistent-programming-language-based OODBs**
  - complex data types, integration with programming language, high performance.
- **Object-relational systems**
  - complex data types, powerful query languages, high protection.
- **Object-relational mapping systems**
  - complex data types integrated with programming language, but built as a layer on top of a relational database system

- Note: Many real systems blur these boundaries, e.g.
  - persistent programming language built as a wrapper on a relational database offers first two benefits, but may have poor performance.

End of Lecture

- Summary
  - Complex Data Types and Object Orientation
  - Structured Data Types and Inheritance in SQL, Table Inheritance
  - Array and Multiset Types in SQL
  - Object Identity and Reference Types in SQL
  - Implementing O-R Features
  - Persistent Programming Languages
  - Comparison of Object-Oriented and Object-Relational Databases
- Reading
  - Textbook 6th edition, chapter 22
  - Textbook 7th edition, shortened. PDF will be provided.

Database Development and Design (CPT201)

Lecture 11: Introduction to Distributed Databases

Dr. Wei Wang

Department of Computing

Learning Outcomes

- Distributed System Concepts
- Distributed Data Storage
- Distributed Transactions
- Distributed Query Processing
- Concurrency Control in Distributed Databases
- Failure Recovery in Distributed Databases

Distributed Database System

- A distributed database system consists of loosely coupled sites that share no physical component
- Database systems that run on each site are independent of each other
- Transactions may access data at one or more sites

Introduction

- Data is stored across several sites, each managed by a DBMS that can run independently.
- The location of data on each individual sites impacts query optimisation, concurrency control and recovery.
- Distributed data is governed by factors such as local ownership, increased availability, and performance issues.

Introduction cont'd

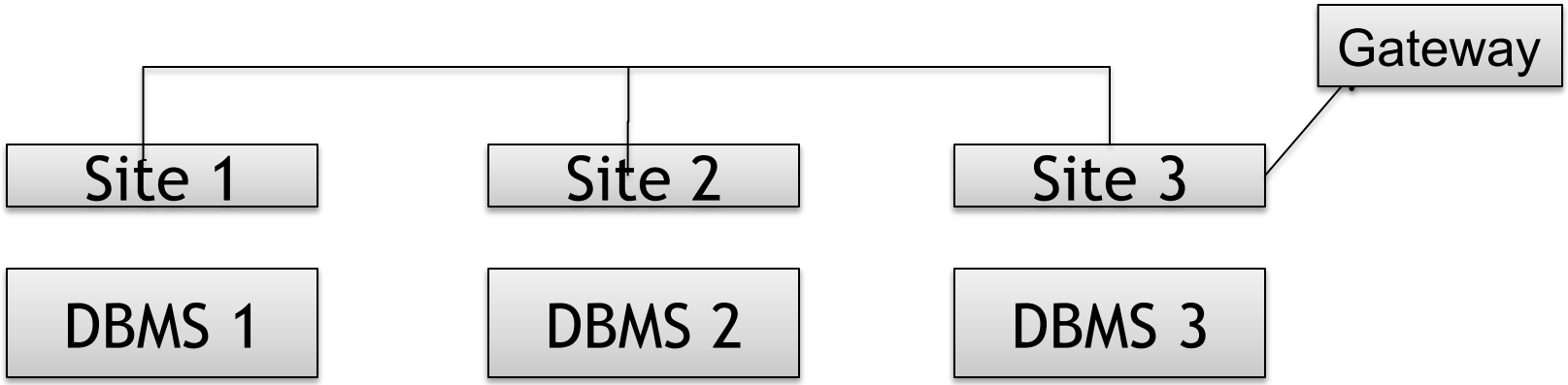
- Distributed Data Independence: Users should not have to know where data is located.
  - reference relations, copies or fragments of the relations.
  - extends Physical and Logical Data Independence principles
- Distributed Transaction Atomicity: Users should be able to write transactions that access and update data at several sites.
  - Transactions are atomic, all changes persist if the transaction commits, or rollback if transaction aborts.

Introduction cont'd

- If sites are connected by slow networks, these properties are hard to support efficiently.
- Users have to be aware of where data is located, i.e. Distributed Data Independence and Distributed Transaction Atomicity are not supported.
- For globally distributed sites, these properties may not even be desirable due to administrative overheads of making locations of data transparent.

Types of Distributed Databases

- **Homogeneous** – data is distributed but all servers run the same DBMS software.
- **Heterogeneous** – different sites run different DBMSs separately and are connected to enable access to data from multiple sites.
  - Gateway protocols - API that exposes DBMS functionality to external applications.
  - Examples: ODBC and JDBC

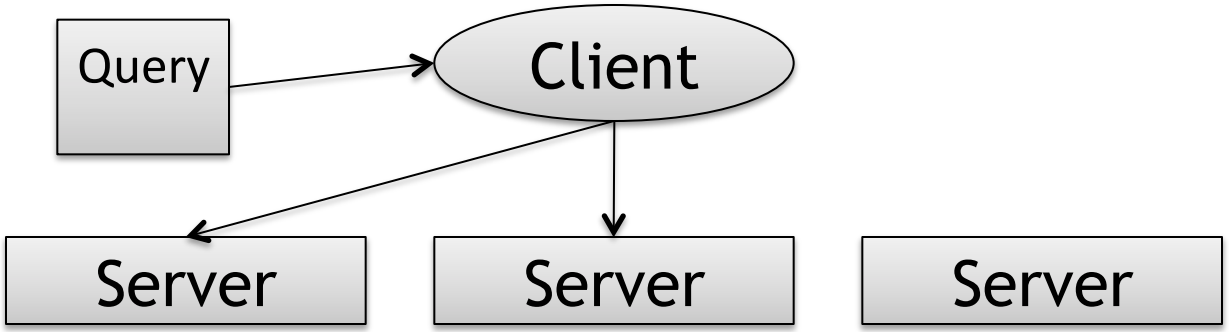


Architectures

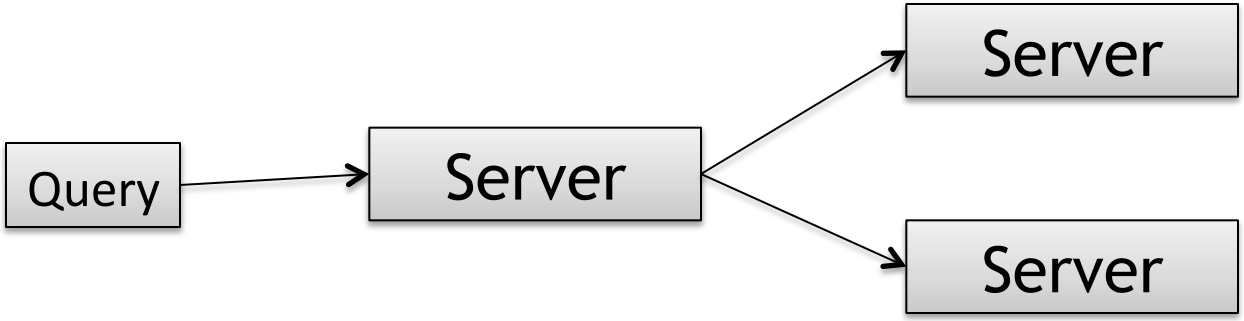
- **Client Server** – a system that has one or more client processes and one or more server processes. Client sends a query to a server, and the server processes the query returning the result to the client.
- **Collaborating Server** - capable of running queries against local data and executes transactions across multiple servers.
- **Middleware** – One database server can manage queries and transactions spanning across multiple servers. A layer that executes relational operations on data from other servers but does not maintain any data.

Architectures cont'd

Client-Server Architecture



Collaborated Server Architecture



Storing data

- Relations are stored across several sites. To reduce message-passing costs a relation maybe fragmented across sites.
- Fragmentation: breaks a relation to smaller relations and stores the fragments at different sites.
  - Horizontal fragments (HF) - rows of the original data.
    - Selection queries, fragments by city
    - **Disjoint union** of the HF must be equal to the original relation.
  - Vertical fragments (VF) - columns of the original data.
    - Projection queries, e.g., fragments of the first two columns
    - Collection of VF must be a **loss-less join decomposition**.

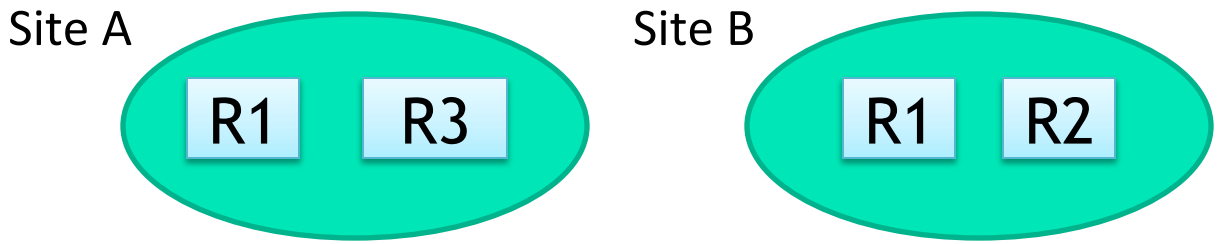
T1	Eid	Name	City
T2	123	Smith	Chicago
T3	124	Smith	Chicago
T4	125	Jones	Madras

HF

VF

Storing data cont'd

- Replication – storing several copies of a relation or fragment. Entire relation can be stored at one or more sites.
  - Increased Availability – If a site contains replicated data goes down, then we can use another site.
  - Faster Query Evaluation – Queries are executed faster by using local copy of a relation instead of going to a remote site.
  - Two kinds of replication are Synchronous and Asynchronous replication.



Distributed Catalog Management

- Must keep track of how data is distributed across sites.
- Must be able to give a unique identifier to each replica of each fragment/relation.
  - Global relation name- <local-name>, <birth-site>
  - Global replica name – *replica id* plus global relation name
- Site catalog - Describes all objects (fragments, replicas) at a site and keeps track of replicas of relations created at this site.
  - To find a relation look up its birth-site catalog
  - Birth-site never changes even if the relation is moved

Updating Distributed Data

- Users should be able to update data without worrying where relations are stored.
- **Synchronous replication** – all copies of a modified relation are updated before the modifying transaction commits.
- **Asynchronous replication** – copies of modified relation are updated over a period of time, and a transaction that reads different copies of the same relation may see different values.
  - Widely used in commercial distributed DBMSs
  - Users must be aware of distributed databases

Synchronous Replication

- **Voting technique** – a transaction must write a majority of copies to modify an object; read at least enough copies to make sure one of the copies is current.
  - For example, 10 copies, 7 are updatable, 4 are read
  - Each copy has a version number, the highest is the most current.
  - Not attractive and efficient, because reading an object requires reading several copies. Objects are read more than updated.
- **Read-any-write-all technique** – a transaction can read only one copy, but must write to all copies.
  - Reads are faster than writes especially if it's a local copy
  - Attractive when reads occur more than writes
  - Most common technique

Cost of Synchronous Replication

- **Read-any-write-all** cost - Before an update transaction can commit, it must lock all copies
  - Transaction sends lock requests to remote sites and waits for the locks to be granted, during a long period, it continues to hold all locks.
  - If there is a site or communication failure then transaction cannot commit until all sites are recovered
  - Committing creates several additional messages to be sent as part of a commit protocol
- Since synchronous replication is expensive, Asynchronous replication is gaining popularity even though different copies can have different values.

Asynchronous Replication

- Allows modifying transactions to commit before all copies have been changed.
  - Users must be aware of which copy they are reading, and that copy may be out-of-sync for short period of time.
- Two approaches: Primary Site and Peer-to-Peer replication.
  - Difference lies in how many copies are "updatable" or "master copies".

Peer to Peer Asynchronous Replication

- More than one copy can be designated as updateable (i.e. master copy).
- Changes to a master copy must be propagated to other copies somehow.
- Conflict resolution is used to deal with changes at different sites.
  - Each master is allowed to update only one fragment of the relation, and any two fragments updatable by different masters are disjoint.
  - Updating rights are held by one master at a time.

Primary Site Replication

- **Primary site** – one copy of a relation is the master copy
- **Secondary site**- replicas of the entire relation are created at other sites. They cannot be updated.
- Users register/publish a relation at the primary site and subscribe to a fragment of the relation at the secondary site.
- Changes to the primary copy transmitted to the secondary copies are done in two steps.
  - First capture changes made by committed transactions, then apply these changes.

Primary Site Asynchronous Replication - Capture

- **Log Based Capture** - the log maintained for recovery is used to generate a Change Data Table (CDT)
- **Procedural Capture** – A procedure that is invoked by the DBMS which takes a snapshot of the primary copy
- Log based capture is generally better because it deals with changes to the data and not the entire database. However it relies on log details which may be system specific.

Primary Site Asynchronous Replication - Apply

- The *Apply* process at the secondary site periodically obtains a snapshot of the primary copy or changes to the CDT table from the primary site, and updates the copy.
  - Period can be timer or user's application program based.
- Replica can be a view over the modified relation.
- Log-Based Capture plus continuous Apply minimises delay in propagating changes.
- Procedural Capture plus application-driven Apply is the most flexible way to process updates.
  - Used in data warehousing applications

Data Warehousing

- Creating giant warehouses of data from many sites
  - Create a copy of all the data at some locations
  - Use the copy rather than going to individual sources.
  - Enable complex decision support queries
- Seen as an instance of asynchronous replication; copies are updated infrequently.
- Source data controlled by different DBMSs
- Need to clean data and remove mismatches while creating replicas.

Distributed Queries

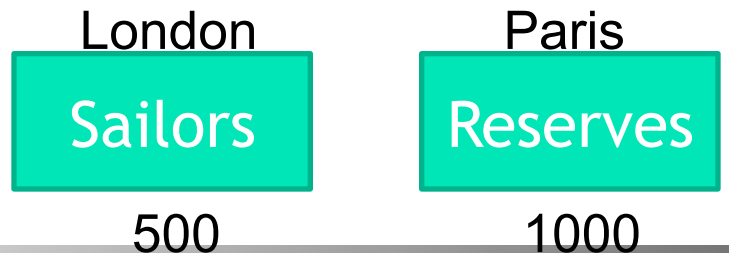
Example query with a relation S (fragmented at Shanghai and Tokyo sites):

```
SELECT AVG(S.age)
FROM Sailors S
WHERE S.rating > 3 AND S.rating < 7
```

- Horizontally Fragmented
  - Tuples with rating < 5 at Shanghai, >= 5 at Tokyo. When calculating average, must compute sum and count at both sites
  - If WHERE contained just S.rating > 6, just one site.
- Vertically Fragmented
  - *sid* and *rating* at Shanghai, *sname* and *age* at Tokyo, *tid* at both.
  - Joining two fragments by a common *tid* and execute the query over this reconstructed relation
- Replicated
  - Since relation is copied to more than one site, choose a site based on local cost.

Distributed Joins

- Joins of relations across different sites can be very expensive.
- Fetch as needed
  - For example: Sailors relation is stored at London and Reserves relation is stored in Paris. There are 500 blocks of Sailors and 1,000 blocks of Reserves
  - Use the block nested loops join in London with Sailors as the outer join, and for each Sailors block, fetch all Reserves blocks from Paris.
  - The cost is: 500 D + 500 * 1000 (D + S)
    - 500D = the time to scan Sailors
    - 500*1000 (D+S) = for each Sailor's block the cost of scanning and shipping all of Reserves
      - D is the cost of read/write blocks
      - S is the cost to ship a block
  - If query was not submitted at London, must add cost of shipping result to query site.
  - Can also do index nested loops join in London, fetching matching Reserves tuples for each Sailors tuple as needed.



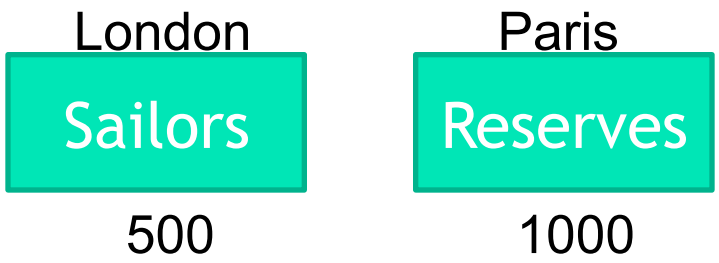
Distributed Joins cont'd

```
SELECT *
FROM Sailors S, Reserves R
WHERE S.sid = Reserves.sid
```

- Fetch as Needed, block nested loop join, Sailors as outer:
  - **Cost:** 500 D + 500 * 1000 (D+S)
  - **D** is cost to read/write page; **S** is cost to ship page.
  - S is very large, so cost is essentially 500,000S
- Ship to One Site – merge join, Ship Reserves to London.
  - Assume only one relation can be read in memory (sorting can be done in memory), cost = 3*(500+1000)
  - Cost: 1000S + 4500D, essentially 1000S as D<<S.

Semi Join & Bloom Join

- Assume shipping Reserves to London and computing the join at London, and some of those tuples in Reserves do not join with any tuples in Sailors.
  - Need to identify Reserve tuples that guarantee not to join with any Sailors tuples.
  - Two techniques: Semi Joins and Bloom Joins



Semi Join

- Three steps in reducing the number of Reserves tuples to be shipped.
  - At London, compute projection of Sailors onto the join attribute and ship this projection to Paris. (sids)
  - At Paris, join Sailors projection with Reserves. Ship the join result to London. This result is called 'reduction of Reserves with respect to Sailors'.
  - At London, compute the join of the reduction of Reserves with sailors.
- Idea: tradeoff the cost of computing and shipping projection for cost of shipping full Reserves relation.
  - Especially useful if there is a selection on Sailors, and answer desired at London.

Bloom Join

- Bloom Join is similar to Semi Join but there is a bit-vector shipped in the first step instead of a projection.
- At London, compute a bit-vector of some size k:
  - Hash each tuple of Sailors (using sid) into range 0 to k-1.
  - If some tuple hashes to i, set bit i to 1 ($0 \leq i \leq k-1$).
  - Ship bit-vector to Paris.
- At Paris, hash each tuple of Reserves (using sid) similarly, and discard tuples that are hashed to 0 in Sailors bit-vector (no Sailors tuples hash to the *i*th partition).
  - Result is called 'reduction of Reserves with respect to Sailors'.
- Ship bit-vector reduced Reserves to London.
- At London, join Sailors with reduced Reserves.
- Bit-vector cheaper to ship, almost always efficient.

Distributed Query Optimisation

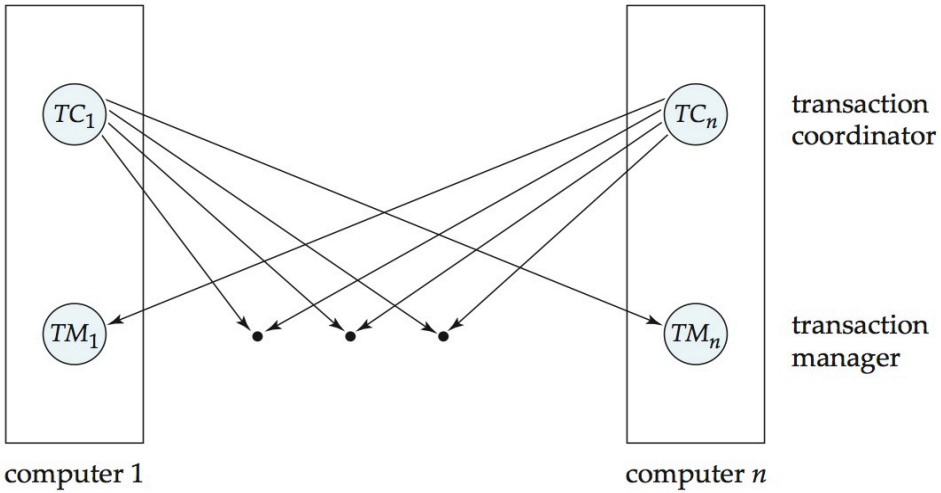
- Consider all plans, pick cheapest; similar to centralised optimisation.
  - Communication costs, if there are several copies of a relation, need to decide which to use.
  - If individual sites are running a different DBMS, autonomy of each local site must be preserved while doing global query planning.
  - Use new distributed join methods.
- Query site constructs a global plan, with suggested local plans describing processing at each site. If a site can improve suggested local plan, free to do so.

Distributed Transactions

- Transaction is submitted at one site but can access data at other sites.
- Each site has its own local transaction manager, whose function is to ensure the ACID properties of those transactions that execute at that site, i.e.,
  - Maintaining a log for recovery purposes.
  - Participating in an appropriate concurrency-control scheme to coordinate the concurrent execution of the transactions executing at that site.
- Concurrency control and recovery
  - Assume Strict two-Phase Locking with deadlock detection is used
    - If a transaction wants to read an object it first requests a shared lock on the object.
    - All exclusive locks held by a transaction are released when the transaction is completed.

Transaction Coordinator

- Transaction coordinator is responsible for:
  - Starting the execution of the transaction.
  - Breaking the transaction into a number of sub-transactions and distributing these sub-transactions to the appropriate sites for execution.
  - Coordinating the termination of the transaction, which may result in the transaction being committed at all sites or aborted at all sites.

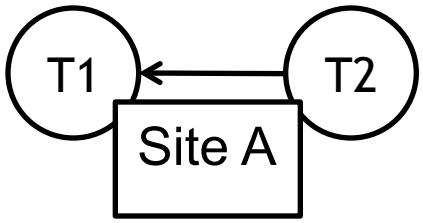


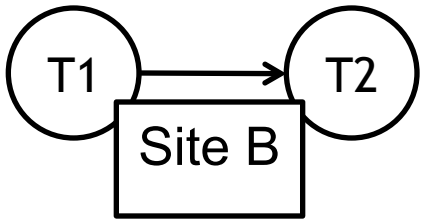
Distributed Locking Protocols

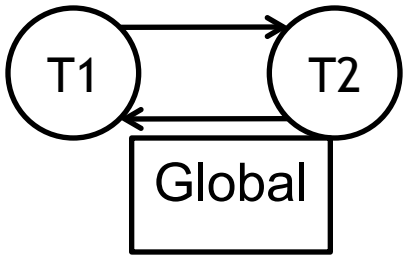
- When locks are obtained and released is determined by the concurrency control protocol.
- Lock management can be distributed across many sites:
  - Single-lock manager (Centralised) – One site does all the locking; vulnerable if one site goes down.
  - Primary Copy – Only one copy of each object is designated a primary copy, requests to lock/unlock are handled by lock manager at the primary site regardless where the copy is stored.
  - Distributed lock manager – Requests to lock/unlock a copy of an object stored at a site are handled by the lock manager at the site where the copy is stored.

Distributed Deadlock

- Each site maintains local waits-for graph, and a cycle in a local graph indicates a **deadlock**.
- A global deadlock might exist even if the local graphs contain no cycles







- Three algorithms of deadlock detection
  - Centralised – send all local graphs to one site that is responsible for deadlock detection.
  - Hierarchal – organise sites into a hierarchy and send local graphs to parent in the hierarchy.
  - Timeout – abort transaction if it waits too long

Distributed Recovery and Concurrency Control

- Recovery in distributed DBMSs is more complicated than in centralised DBMSs
  - e.g., failure of communication links; failure of a remote site at which a sub transaction is executing.
  - All of the sub transactions must commit or not commit at all. This must be guaranteed despite link failures.
  - Need a commit protocol – the most common one is Two-Phase Commit.
- A log is maintained at each site, as in a centralised DBMS, and commit protocol actions are additionally logged.

Commit Protocols: Two Phase Commit (2PC)

- The site at which the transaction originated is the coordinator. Other sites at which sub transactions are executed are subordinates.
- When a user decides to commit a transaction, the commit command is sent to the coordinator for the transaction. This initiates the **2PC**:
  - Phase 1: Coordinator sends a prepare message to each subordinate; When subordinate receives a prepare message, it decides to abort or commit its sub transaction; Subordinate forces writes an abort or ready log record and sends a no or yes message to coordinator accordingly.
  - Phase 2: If coordinator receives a yes message from all subordinates, force-writes a commit log record and sends commit message to all subordinates. Else force-writes a an abort log record and sends an abort message; Subordinates force-write abort or commit log record based on the message they receive.
  - In some implementations, after the two phases, subordinates send acknowledgement message to coordinator; After coordinator receives ack messages from all subordinates it writes an end log for the transaction.

READ TEXTBOOK

Recover After Failure

- When a site comes back from a crash there is a recovery process that reads the log and processes all transactions which executed the commit protocol at the time of the crash.
- Failure of a participating site
  - Examine its own logs, if there is commit or abort log record for transaction T, then redo/undo T respectively.
  - The log contains a <ready T> record, repeatedly contact the coordinator or other active sites to find the status of T, then performs redo/undo accordingly and write commit/ abort log records depending on coordinator's response;
  - The log contains no control records (abort, commit, ready) concerning T, undo.
- Failure of the coordinator
  - participating sites must decide the fate of T.
  - If an active site contains a <commit T>/<abort T> record in its log, then T must be committed/aborted.
  - If some active site does not contain a <ready T> , preferable to abort T.
  - If active sites have a <ready T> record in their logs, but no additional control records (such as <abort T> or <commit T>), it is impossible to determine if a decision has been made, and what that decision is, until the coordinator recovers. (***in-doubt transaction***)

Blocking – Coordinator Failure

- If locking is used, an in-doubt transaction T may hold locks on data at active sites. Such a situation is undesirable, because it may be hours or days before coordinator is again active. This situation is called the blocking problem, because T is blocked pending the recovery of coordinator site.
- Solution:
  - Use **<ready T, L>** log record, where **L** is a list of all write locks held by the transaction T when the log record is written. At recovery time, after performing local recovery actions, for every in-doubt transaction T, all the write locks noted in the <ready T, L> log record (read from the log) are reacquired.
  - Can also be solved using the 3-phase commit protocol (under certain situations)

Recovery from Network Partitions

- The coordinator and all its participants remain in one partition. In this case, the failure has no effect on the commit protocol.
- The coordinator and its participants belong to several partitions. From the viewpoint of the sites in one of the partitions, it appears that the sites in other partitions have failed.
  - Sites that are not in the partition containing the coordinator simply execute the protocol to deal with failure of the coordinator.
  - The coordinator and the sites that are in the same partition: the coordinator follow the usual commit protocol, assuming that the sites in the other partitions have failed.

End of Lecture

- Summary
  - Distributed System Concepts
  - Distributed Data Storage
  - Distributed Transactions
  - Distributed Query Processing
  - Concurrency Control in Distributed Databases
  - Failure Recovery in Distributed Databases
- Reading
  - Textbook 6th edition, chapters 17.4, 17.5, 19.1, 19.2, 19.3, 19.4, 19.5
  - Textbook 7th edition, chapters 20.5, 21.2, 21.4, 22.9, 23.1, 23.2, 23.3

Database Development and Design (CPT201)

Lecture 12: NoSQL Database and Big Data Storage

Dr. Wei Wang

Department of Computing

Acknowledgement

- Thanks to Prof. Tok Wang LING, National University of Singapore. Some of the slides are based on his talk, "Conceptual Modeling Views of Relational Databases vs Big Data" at DSIT 2019.

Learning Outcomes

- Introduction to Big Data
- Issues and performance problems in RDBMS
- NoSQL and categories
- SQL vs NoSQL
- Big data storage
- MapReduce
- MapReduce vs database

Big Data

- Big data is a broad term for data sets and it can be described by the following 3Vs characteristics:
  - Volume (huge large amount of data: terabytes, petabytes, exabytes)
  - Velocity (speed of data in and out: real-time, streaming)
  - Variety (range of data types and sources, non-relational data such as nested relation, documents, XML data, web data, graph, multimedia, flexible schema or no schema)

Big Data cont'd

- Two more Vs
  - Veracity (correctness and accuracy of information: data quality and reliability)
  - Value (use machine learning, data mining, statistics, visualisation, decision analysis techniques to extract/mine/derive previously unknown insights from data and become actionable knowledge, business value)
- Traditional Relational database management systems (RDBMSs) using SQL are inadequate to handle big data efficiently.

Database Models

- File system
- Hierarchical Model (IMS)
- Network Model (IDMS)
- Relational Model
- Nested Relational Model
- Entity-Relationship Approach
- Object-Oriented(OO) Data Model
- Deductive and Object-Oriented (DOOD)
- Object Relational Data Model
- Semi-structured Data Model (XML)
- RDF and Linked Data
- …

Issues and performance problems in RDBMS

- Normal forms in relational models are to remove redundancies and to reduce updating anomalies. Does data redundancy definitely incur updating anomalies?
  - Example 1: supply (S#, Sname, P#, Pname, price). This supply relation is not even in 2NF. It has redundant information on Sname and Pname, but it does not suffer from updating anomalies as we don't change Sname and Pname of suppliers and parts, resp.
  - Example 2: For Sales transactions, we can add and store the item name, price of item, amount for each item ordered (use the quantity ordered), and total amount of each sales transaction. They are all needed to print the receipts. These are redundant, but they don't incur updating anomalies as we don't change the transactions once they are done.
  - This gives better performance for data analytics! No need to join relations again.

Issues and performance problems in RDBMS cont'd

- Adding redundancy in physical database design may not incur updating anomalies and instead may improve performance significantly, avoid joins.
  - RDBMS cannot handle multi-valued attributes and composite attributes efficiently.
  - Example: nested relation: employee (e#, name, sex, dob, hobby*, qualification(degree, university, year)*)
  - To store employee information, we need 3 normal form relations:
    - employee (e#, name, sex, dob)
    - employee_hobby (e#, hobby)
    - employee_qual (e#, degree, university, year)
  - To get information of an employee, we need to join the 3 relations, very inefficient and very slow and has a lot of redundant information as the joined relation is not in 4NF.

Issues and performance problems in RDBMS cont'd

- RDBMS join operation is very expensive. So it may not be suitable for some applications.
- ACID (Atomicity, Consistency, Isolation, Durability) is to ensure the consistency of data.
  - Two-Phase locking is required. But overhead for enforcing ACID (using locks) is extremely high for managing the locks.
  - Example: large data volume applications which don't modify existing data or at most only add new data, don't require ACID.

NoSQL (not-only SQL)

- Flexible schema or no schema; avoidance of unneeded complexity, e.g. expensive object-relational mapping.
  - NoSQL databases are designed to store data structures that are either simple or more similar to the ones of object-oriented programming languages compared to relational data structures.
- Massive scalability/high throughput
- Relaxed consistency for higher performance and availability
- Quicker/cheaper to set up
- etc

Eventual consistency

- NoSQL softens the ACID properties in relational databases to allow horizontal scalability.
- Desirable properties of horizontally distributed systems:
  - Consistency in a distributed system requires a total order on all operations throughout all nodes of the system
  - A system satisfies availability, if all operations, executed on a node of the system, terminate in a response. Partition cannot be guaranteed in large distributed systems.

Eventual consistency cont'd

- BASE (proposed for Scalable systems) stands for basically available, soft state, eventually consistent.
  - focuses mainly on availability of a system, at the cost of loosening the consistency.
- The eventual consistency property of a BASE system accepts periods where clients might read inconsistent (i.e. out-dated) data.
  - Though, it guarantees that those periods will eventually end.

Eventual consistency cont'd

- Strong consistency: after the update completes, any subsequent access will return the updated value.
- Weak consistency: a number of conditions need to be met before the updated value will be returned.
- Eventual consistency: a consistency model used in distributed computing to achieve high availability that informally guarantees that, if no new updates are made to a given data item, eventually all accesses to that item will return the last updated value. (*Vogels, W. (2009. "Eventually consistent". Communications of the ACM. 52: 40.*)

NoSQL categories

- NoSQL databases are categorised according to the way they store the data.
- Four major categories for NoSQL databases:
  - Key-value Stores
  - Wide-Column Stores
  - Document Stores
  - Graph Database

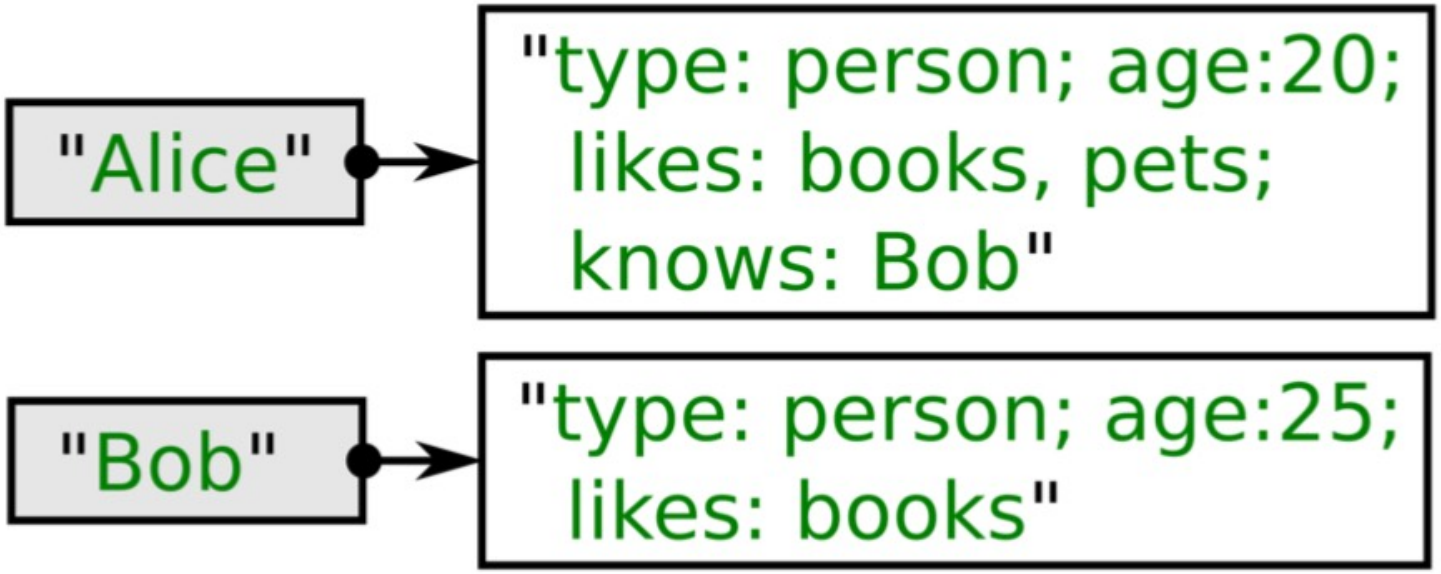
Key-value store

- Key-value storage systems store large numbers (billions or even more) of small (KB-MB) sized records
- Records are partitioned across multiple machines
- Queries are routed by the system to appropriate machines
- Records are also replicated across multiple machines, to ensure availability even if a machine fails
  - Key-value stores ensure that updates are applied to all replicas, to ensure that their values are consistent

Key-value store cont'd

- A key-value store is like associate array
  - data is represented in the form of array["key"] = value or hash table in main memory.
- Each data/object is stored, indexed, and accessed using a key value to access the hash table or array.
- Value is a single opaque collection of objects or data items
  - can be structured, semi-structured, or unstructured. It is just an un-interpreted string of bytes of arbitrary length.
- The meaning of the value in a key-value pair has to be interpreted by programmers.
- No concept of "foreign key", no join
  - data can be horizontally partitioned and distributed

Key-value store cont'd



- Stored value typically can not be interpreted by the storage system.

Key-value store cont'd

- Key-value stores may store
  - Un-interpreted bytes, with an associated key
    - E.g., Amazon S3, Amazon Dynamo
  - Wide-table (can have arbitrarily many attribute names) with associated key
    - Google BigTable, Apache Cassandra, Apache Hbase, Amazon DynamoDB
    - Allows some operations (e.g., filtering) to execute on storage node
  - JSON
    - MongoDB, CouchDB (document model)
- Some key-value stores support multiple versions of data, with timestamps/version numbers

Data representation

- An example of a JSON object is:

```
{
    "ID": "22222",
    "name": {
        "firstname: "Albert",
        "lastname: "Einstein"
        },
    "deptname": "Physics",
    "children": [
        { "firstname": "Hans", "lastname":
    "Einstein" },
        { "firstname": "Eduard", "lastname":
    "Einstein" }
    ]
}
```

Key-value store cont'd

- Typical operations include (but no modification):
  - INSERT new Key-Value pairs (or put)
  - LOOKUP value for a specified key (or get)
  - DELETE key and the value associated with it
- Some systems also support ***range queries*** on key values
- Key value stores are not full database systems
  - Have no/limited support for transactional updates
  - Applications must manage query processing on their own
- Not supporting above features makes it easier to build scalable data storage systems,
  - Also called ***NoSQL*** systems (this is from the textbook, a bit controversial)

Wide-Column Stores

- Data is stored as tables. A table has a row-key and a pre-defined set of column-family columns.
- Each row in the table is uniquely identified by a row-key value.
- Each column family has a large and flexible number of columns (i.e. the No of columns may change from row to row)
  - Each column has a name together with one or more values.
- A column-oriented DBMS stores data tables as column families of data rather than as rows of data.
  - Better for data compression.

Wide-Column Stores cont'd

- A keyspace in a wide-column store contains all the column families (like tables in the relational model), which contain rows, which contain columns.

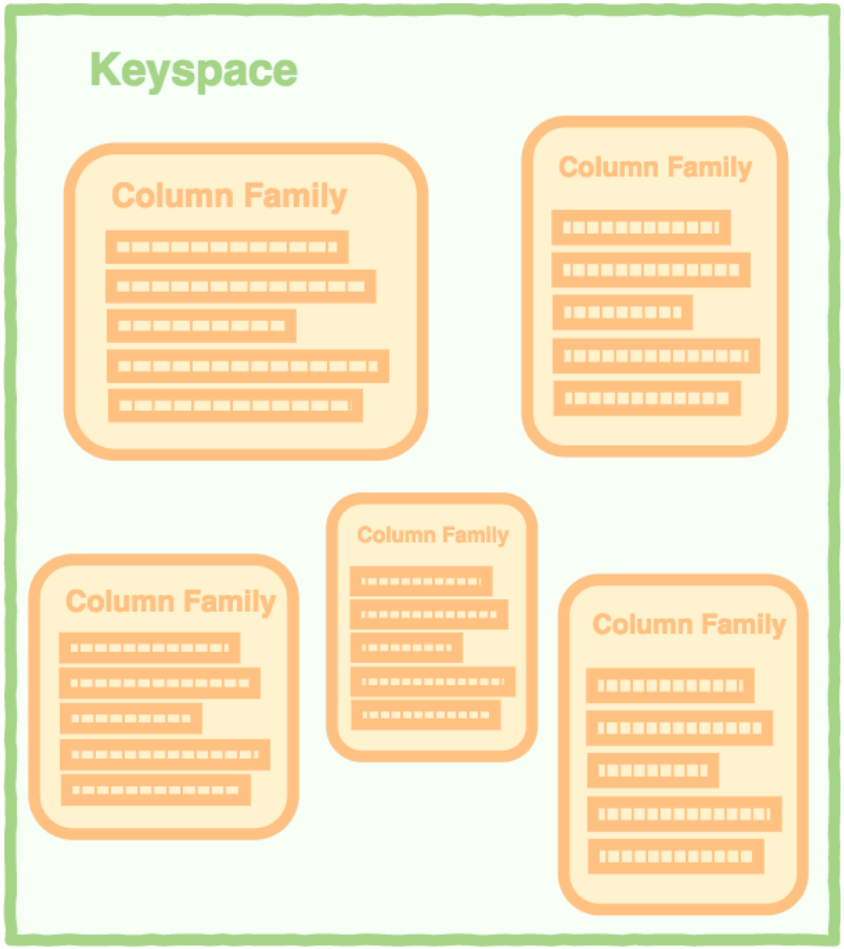


Image source: https://database.guide/what-is-a-column-store-database/

Wide-Column Stores cont'd

- A column family containing 3 rows. Each row contains its own set of columns.

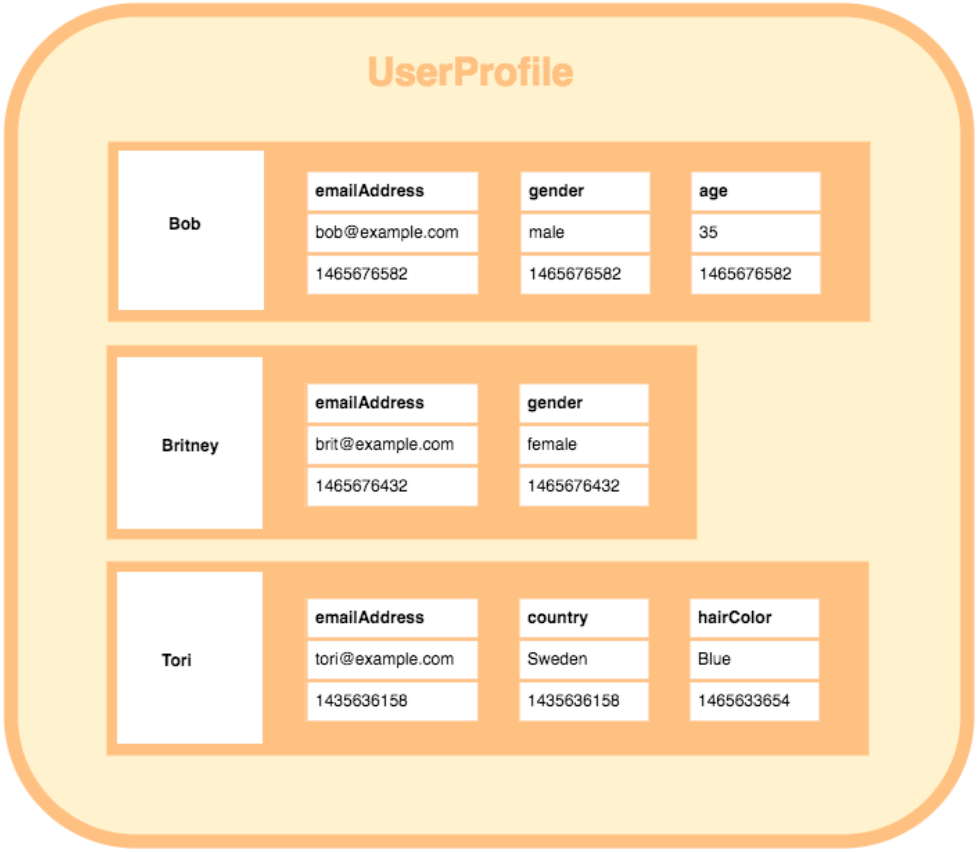


Image source: https://database.guide/what-is-a-column-store-database/

Wide-Column Stores cont'd

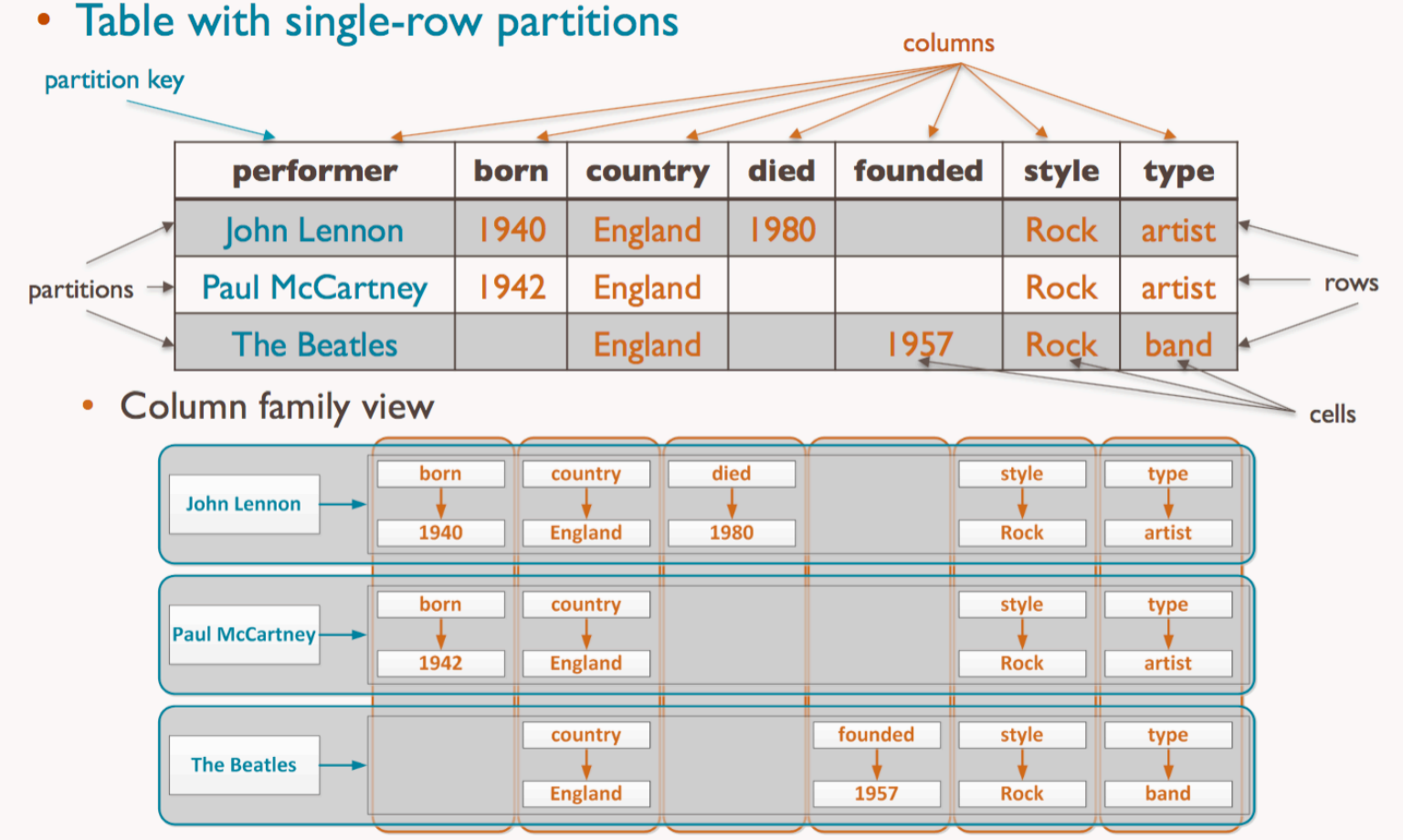


Image source: https://studio3t.com/knowledge-base/articles/nosql-database-types/#wide-column-store

Google's BigTable

- A sparse, distributed, persistent multi-dimensional sorted map.
- Used by several Google applications such as web indexing, MapReduce, Google Maps, Google Earth, YouTube, Gmail, etc.
- The map is indexed by a row key, column key, and a timestamp; each value in the map is an un-interpreted array of bytes.
  - For webpage, the row key value is a reversed url.
  - BigTable maintains data in lexicographic order by row key.
  - So webpages in the same domain are grouped together into contiguous rows.

BigTable: Storing Web Pages

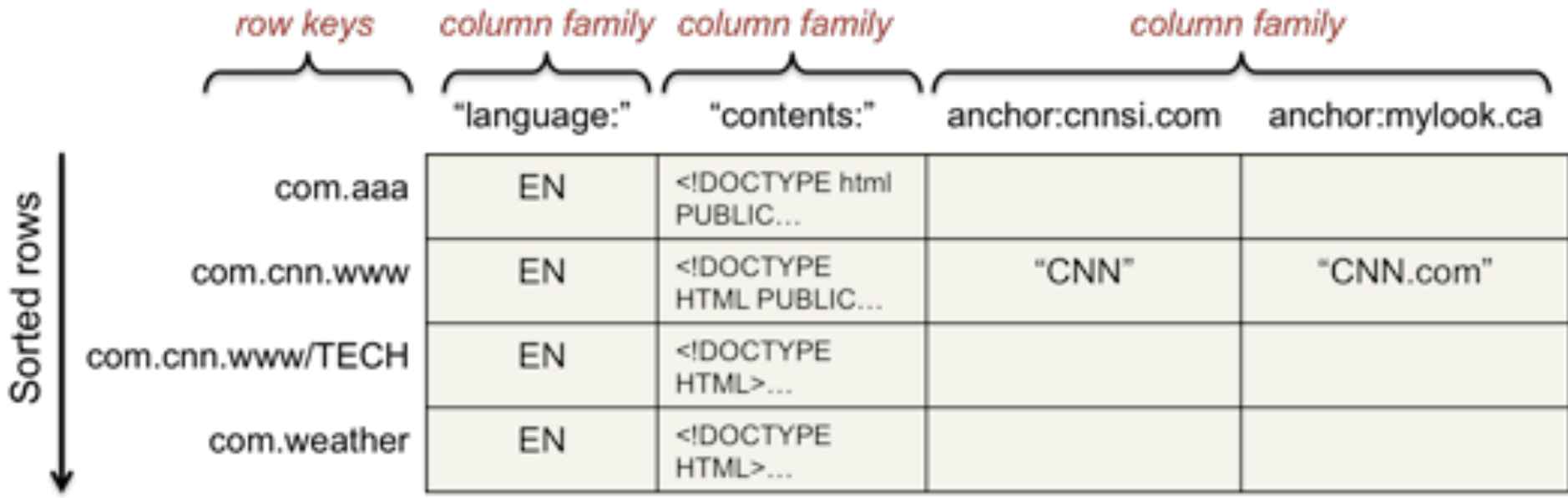


Image source: https://www.cs.rutgers.edu/~pxk/417/notes/content/bigtable.html

Graph Database

- Best suited for representing data with a large number of interconnections
  - especially when information about those interconnections is at least as important as the represented data
  - for example, social relations or geographic data.
- Graph databases allow for queries on the graph structure, e.g., relations between nodes or shortest paths.
- Examples:
  - RDF graph and linked data
  - Google knowledge graph

Document Stores

- Schema languages are not powerful to express Object- Relationship-Attribute semantics in ER model.
- Data is stored in so-called documents, i.e. arbitrary data in some (semi-)structured format.
  - JSON
  - BSON
  - XML
- Data format is typically fixed, but the structure is flexible.
  - in a JSON-based document store, documents with completely different sets of attributes can be stored together.

SQL VS NoSQL

Characteristics	SQL	NoSQL
Schema	Yes. Schema must be predefined and fixed. Schema evolution is difficult.	Schema is optional, may not be predefined. Schema can be semi-structured or un-structured.
Data type	Flat relations. Fixed length field/record for each relation defined.	Tree/graph structured data. Variable length, multi-valued attribute (repeating, nested tree), can add new tag names any time.

SQL VS NoSQL cont'd

Characteristics	SQL	NoSQL
Data persistence	Databases are stored on disk drive, data persistence. Very slow to access as compared to in-memory stores.	In-memory, use pointer and hashing. Very fast to access. Need to convert data in memory from/to disk drive to archive data persistence.
OLTP or OLAP	OLTP, mission critical online transaction applications. Only keep current database state. If historical data is required then need to use temporal database with time period to store.	OLAP for data warehouse and data analytics. Keep the historical data, time/date dimension is a must for meaningful data analysis. (Seldom mention time attribute in key-value store and data graph.)

SQL VS NoSQL cont'd

Characteristics	SQL	NoSQL
Language to access the data	Standard DBMS declarative query language SQL. Operate on a set of tuples at a time basis.	Different imperative programming languages. Write programs (e.g. MapReduce, JSON programs with API's).
Update to data	Frequent update to database (transaction)	mainly have new data, no or seldom updates (deletion and addition)
Query optimisation	Query optimiser of RDBMS	Optimisation done by programmers for each of their programs.
DBMS	RDBMS	Not really, just as data stores

SQL VS NoSQL cont'd

Characteristics	SQL	NoSQL
Answers for queries	Return accurate/precise query answers	Return "best guess" or "an opinion" answers – similar to data mining and IR answers
ACID	Yes, consistency is the most important issue for OLTP applications	Emphasis on speed performance, use eventual consistent. If no updates, then ACID is not required.
Join operation	Yes, queries may involve many joins, can be very slow.	Avoid or no join. Use redundant data to speed up processing
Ad hoc user queries	Write SQL programs or RDB keyword query search	Need programmers to write programs.
Distributed & parallel processing	Limited	Yes. Data can be partitioned horizontally and/or vertically and distributed to nodes, and process the partitioned data in parallel. Efficient for such applications.

Big Data Storage Systems

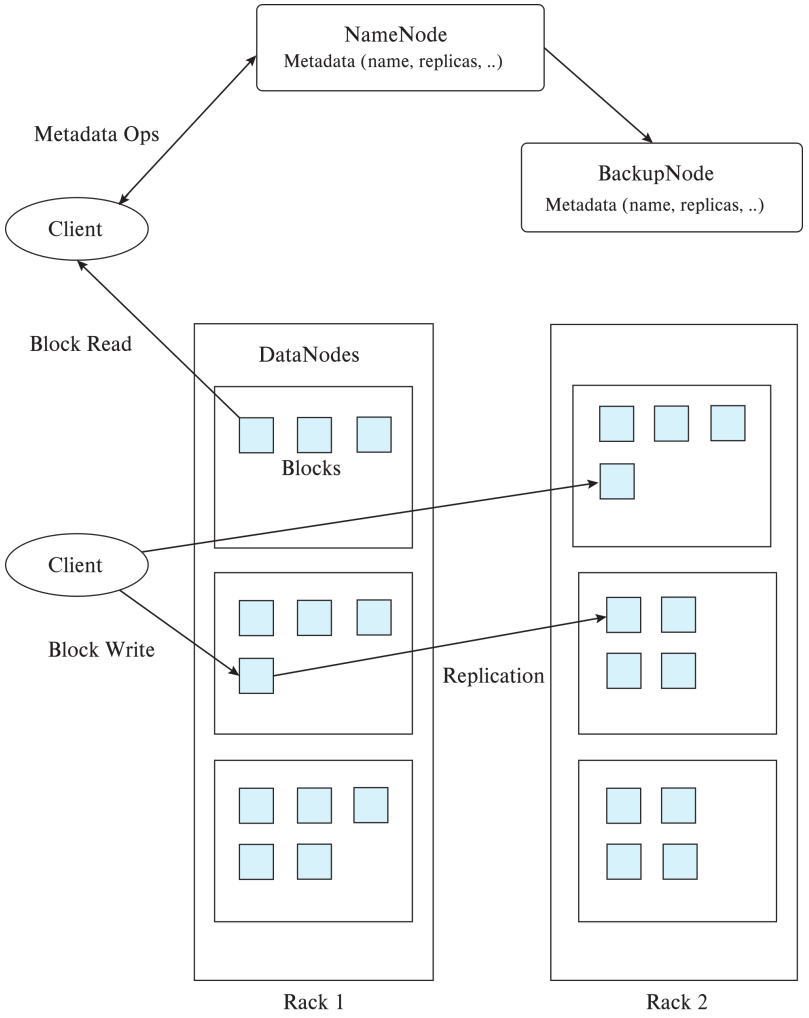
- Distributed file systems
- Sharding across multiple databases
- Key-value storage systems
  - The textbook categorises all the NoSQL storage systems as key-value stores
- Parallel and distributed databases

Distributed File Systems

- A distributed file system stores data across a large collection of machines, but provides single file-system view
- Highly scalable distributed file system for large data-intensive applications.
  - E.g., 10K nodes, 100 million files, 10 PB
- Provides redundant storage of massive amount of data on cheap and unreliable computers
  - Files are replicated to handle hardware failure
  - Detect failures and recovers from them
- Examples:
  - Google File System (GFS)
  - Hadoop File System (HDFS)

Hadoop File System Architecture

- Single Namespace for entire cluster
- Files are broken up into blocks
  - Typically 64 MB block size
  - Each block replicated on multiple DataNodes
- Client
  - Finds location of blocks from NameNode
  - Accesses data directly from DataNode



Hadoop Distributed File System

- **NameNode**
  - Maps a filename to list of Block IDs
  - Maps each Block ID to DataNodes containing a replica of the block
- **DataNode**: Maps a Block ID to a physical location on disk
- Data Coherency
  - Write-once-read-many access model
  - Client can only append to existing files
- Distributed file systems are good for millions of large files
  - But have very high overheads and poor performance with billions of smaller tuples

Sharding

- **Sharding**: partition data across multiple databases
- Partitioning usually done on some ***partitioning attributes*** (also known as ***partitioning keys*** or ***shard keys*** e.g. user ID
  - records with key values from 1 to 100,000 on database 1, records with key values from 100,001 to 200,000 on database 2, etc.
- Application must track which records are on which database and send queries/updates to that database
- Positives: scales well, easy to implement
- Drawbacks:
  - Not transparent: application has to deal with routing of queries, queries that span multiple databases
  - If a database is overloaded, moving part of its load out is not easy
  - Chance of failure more with more databases
    - need to keep replicas to ensure availability, which is more work for application

Parallel Databases and Data Stores

- Supporting scalable data access
  - Approach 1: memcache or other caching mechanisms at application servers, to reduce database access
    - Limited in scalability
  - Approach 2: Partition ("shard") data across multiple separate database servers
  - Approach 3: Use existing parallel databases
    - Historically: parallel databases that can scale to large number of machines were designed for decision support not OLTP
  - Approach 4: Massively Parallel Key-Value Data Store
    - Partitioning, high availability etc. completely transparent to application
- Sharding systems and key-value stores don't support many relational features, such as joins, integrity constraints, etc., across partitions.

Parallel and Distributed Databases

- Parallel databases run multiple machines (cluser)
  - Developed in 1980s, well before Big Data
- Parallel databases were designed for smaller scale (10s to 100s of machines)
  - Did not provide easy scalability
- Replication used to ensure data availability despite machine failure
  - But typically restart query in event of failure
    - Restarts may be frequent at very large scale
    - Map-reduce systems (coming up next) can continue query execution, working around failures

The MapReduce Paradigm

- Platform for reliable, scalable parallel computing
- Abstracts issues of distributed and parallel environment from programmer
  - Programmer provides core logic (via map() and reduce() functions)
  - System takes care of parallelisation of computation, coordination, etc.
- Paradigm dates back many decades
  - But very large scale implementations running on clusters with 10^3 to 10^4 machines are more recent
  - Google Map Reduce, Hadoop, ..
- Data storage/access typically done using distributed file systems or key-value stores

MapReduce Example: Word Count

- Consider the problem of counting the number of occurrences of each word in a large collection of documents
- Solution:
  - Divide documents among workers
  - Each worker parses document to find all words, map function outputs (word, count) pairs
  - Partition (word, count) pairs across workers based on word
  - For each word at a worker, reduce function locally add up counts
- Input: "One a penny, two a penny, hot cross buns."
  - Records output by the map() function would be
    - ("One", 1), ("a", 1), ("penny", 1),("two", 1), ("a", 1), ("penny", 1), ("hot", 1), ("cross", 1), ("buns", 1).
  - Records output by reduce function would be
    - ("One", 1), ("a", 2), ("penny", 2), ("two", 1), ("hot", 1), ("cross", 1), ("buns", 1)

Pseudo-code of Word Count

```
map(String record):
    for each word in record
      emit(word, 1);

// First attribute of emit above is called reduce key
// In effect, group by is performed on reduce key to create a
// list of values (all 1's in above code).  This requires shuffle
  step across machines.
// The reduce function is called on list of values in each group

reduce(String key, List value_list):
    String word = key
    int count = 0;
    for each value in value_list:
      count = count + value
    Output(word, count);
```

MapReduce Programming Model

- Inspired from map and reduce operations commonly used in functional programming languages like Lisp.
- Input: a set of key/value pairs
- User supplies two functions:
  - **map**(k,v) → list(k1,v1)
  - **reduce**(k1, list(v1)) → v2
- (k1,v1) is an intermediate key/value pair
- Output is the set of (k1,v2) pairs
- For our example, assume that system
  - Breaks up files into lines, and
  - Calls map function with value of each line
    - Key is the line number

MapReduce Example: Log Processing

- Given log file in following format:

2013/02/21 10:31:22.00EST /slide-dir/11.ppt
2013/02/21 10:43:12.00EST /slide-dir/12.ppt
2013/02/22 18:26:45.00EST /slide-dir/13.ppt
2013/02/22 20:53:29.00EST /slide-dir/12.ppt
...

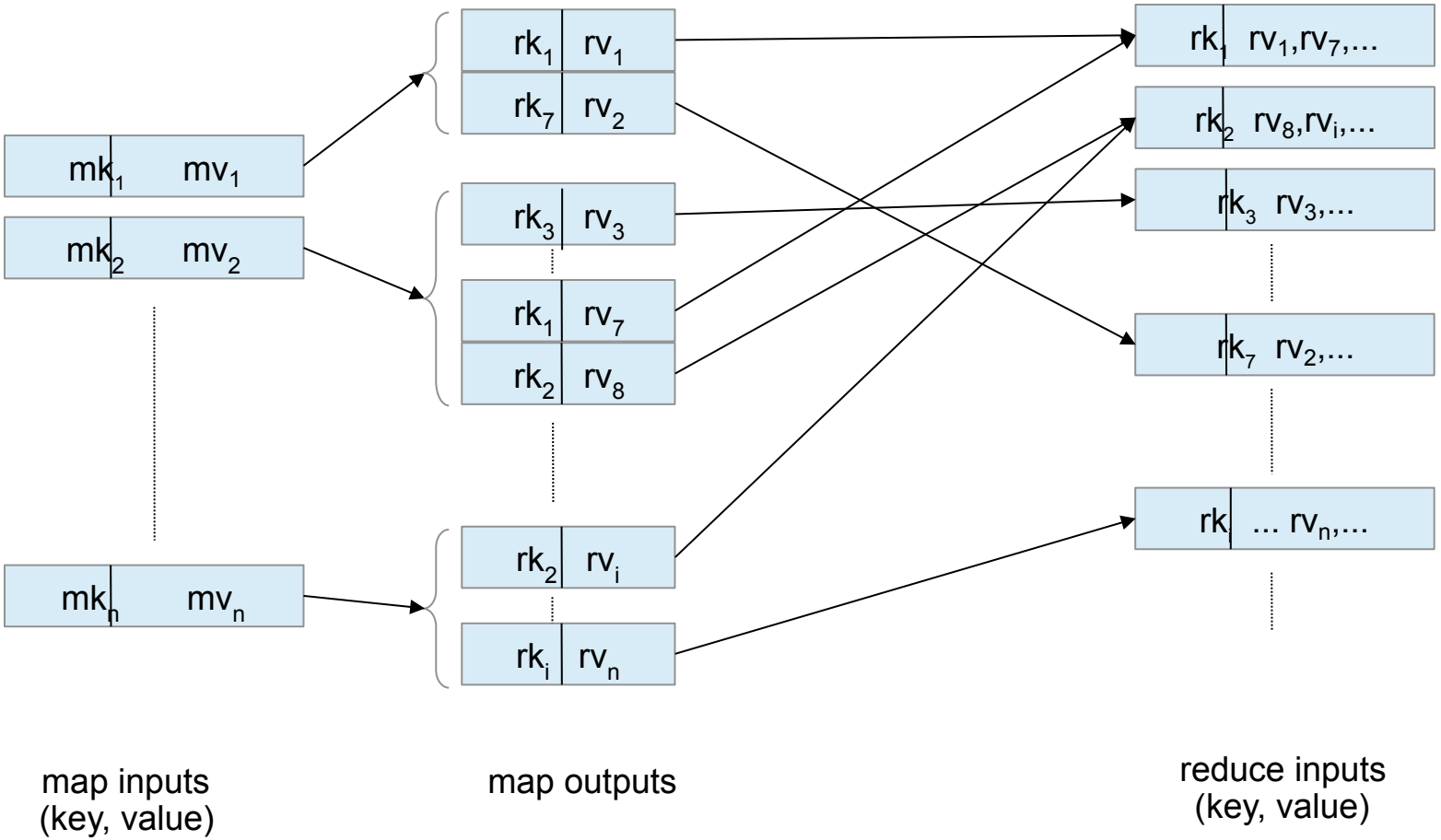
- Goal: find how many times each of the files in the *slide-dir* directory was accessed between 2013/01/01 and 2013/01/31.
- Options:
  - Sequential program too slow on massive datasets
  - Load into database expensive, direct operation on log files cheaper
  - Building parallel program is possible, but very laborious
  - Map-reduce paradigm

MapReduce Example: Log Processing cont'd

```
map(String key, String record) {
    String attribute[3];
    //break up record into tokens (based on space character), and store
     the tokens in array attributes
    String date = attribute[0];
    String time = attribute[1];
    String filename = attribute[2];
    if (date between 2013/01/01 and 2013/01/31
            and filename starts with "/slide-dir/")
        emit(filename, 1).
}

reduce(String key, List recordlist) {
    String filename = key;
    int count = 0;
    For each record in recordlist
       count = count + 1
    output(filename, count)
}
```

Flow of keys and values in a map-reduce task



Hadoop MapReduce

- Google pioneered map-reduce implementations that could run on thousands of machines (nodes), and transparently handle failures of machines
- Hadoop is a widely used open source implementation of Map Reduce written in Java
  - Map and reduce functions can be written in several different languages.
- Input and output to map reduce systems such as Hadoop must be done in parallel
  - Google used GFS distributed file system
  - Hadoop uses Hadoop Distributed File System (HDFS)
  - Input files can be in several formats
    - Text/CSV
    - compressed representation such as Avro, ORC and Parquet
  - Hadoop also supports key-value stores such as Hbase, Cassandra, MongoDB, etc.

Types in Hadoop

- Generic Mapper and Reducer interfaces both take four type arguments, that specify the types of the
  - input key, input value, output key and output value
- Map class in next slide implements the Mapper interface
  - Map input key is of type LongWritable, i.e. a long integer
  - Map input value which is (all or part of) a document, is of type Text.
  - Map output key is of type Text, since the key is a word,
  - Map output value is of type IntWritable, which is an integer value.

Hadoop Code in Java: Map Function

```
public static class Map extends Mapper<LongWritable, Text, Text,
    IntWritable>
{
   private final static IntWritable one = new IntWritable(1);
   private Text word = new Text();
   public void map(LongWritable key, Text value, Context context)
          throws IOException, InterruptedException
   {
       String line = value.toString();
       StringTokenizer tokenizer = new StringTokenizer(line);
       while (tokenizer.hasMoreTokens()) {
             word.set(tokenizer.nextToken());
             context.write(word, one);
       }
   }
}
```

Hadoop Code in Java: Reduce Function

```
public static class Reduce extends Reducer<Text, IntWritable,
    Text, IntWritable> {
   public void reduce(Text key, Iterable<IntWritable> values,
         Context context)  throws IOException,
    InterruptedException
   {
       int sum = 0;
       for (IntWritable val : values) {
            sum += val.get();
       }
       context.write(key, new IntWritable(sum));
   }
}
```

Hadoop Code in Java: Overall Program

```
public class WordCount {
    public static void main(String[] args) throws Exception {
        Configuration conf = new Configuration();
        Job job = new Job(conf, "wordcount");
        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(IntWritable.class);
        job.setMapperClass(Map.class);
        job.setReducerClass(Reduce.class);
        job.setInputFormatClass(TextInputFormat.class);
        job.setOutputFormatClass(TextOutputFormat.class);
        FileInputFormat.addInputPath(job, new Path(args[0]));
        FileOutputFormat.setOutputPath(job, new Path(args[1]));
        job.waitForCompletion(true);
    }
}
```

Map Reduce vs. Databases

- Map Reduce is widely used for parallel processing
  - Google, Yahoo, and 100's of other companies
  - Example uses: compute PageRank, build keyword indices, do data analysis of web click logs, ….
  - Allows procedural code in map and reduce functions
  - Allows data of any type
- Many real-world uses of MapReduce cannot be expressed in SQL
- But many computations are much easier to express in SQL
  - Map Reduce is cumbersome for writing simple queries
- Relational operations (select, project, join, aggregation, etc.) can be expressed using Map Reduce
- SQL queries can be translated into Map Reduce infrastructure for execution
  - Apache Hive SQL, Apache Pig Latin, Microsoft SCOPE

End of Lecture

- Summary
  - Introduction to Big Data
  - Issues and performance problems in RDBMS
  - NoSQL and categories
  - SQL vs NoSQL
  - Big data storage
  - MapReduce
  - MapReduce vs database
- Reading
  - Textbook 7th edition, chapters 10.1, 10.2, 10.3
  - Zollmann, Johannes. "Nosql databases." Retrieved from Software Engineering Research Group: http://www.webcitation.org/6hA9zoqRd, (2012).
  - Strauch, Christof, Ultra-Large Scale Sites, and Walter Kriha. "NoSQL databases." Lecture Notes, Stuttgart Media University 20 (2011).
  - BigTable, https://www.cs.rutgers.edu/~pxk/417/notes/content/bigtable.html

Database Development and Design (CPT201)

Lecture 13a: Data Mining 1 – Classification, OLAP and Decision Tree

Dr. Wei Wang

Department of Computing

Learning Outcome

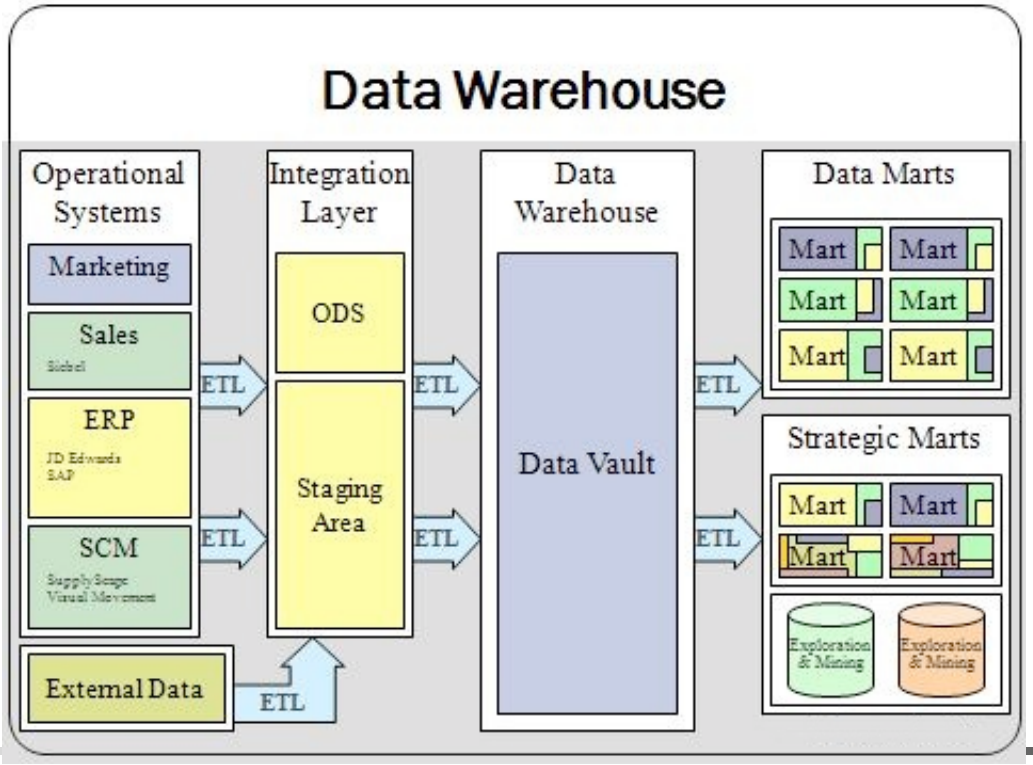
- Data warehouse
- OLAP: Online Analytical Processing
- Classification and Decision Trees

Introduction

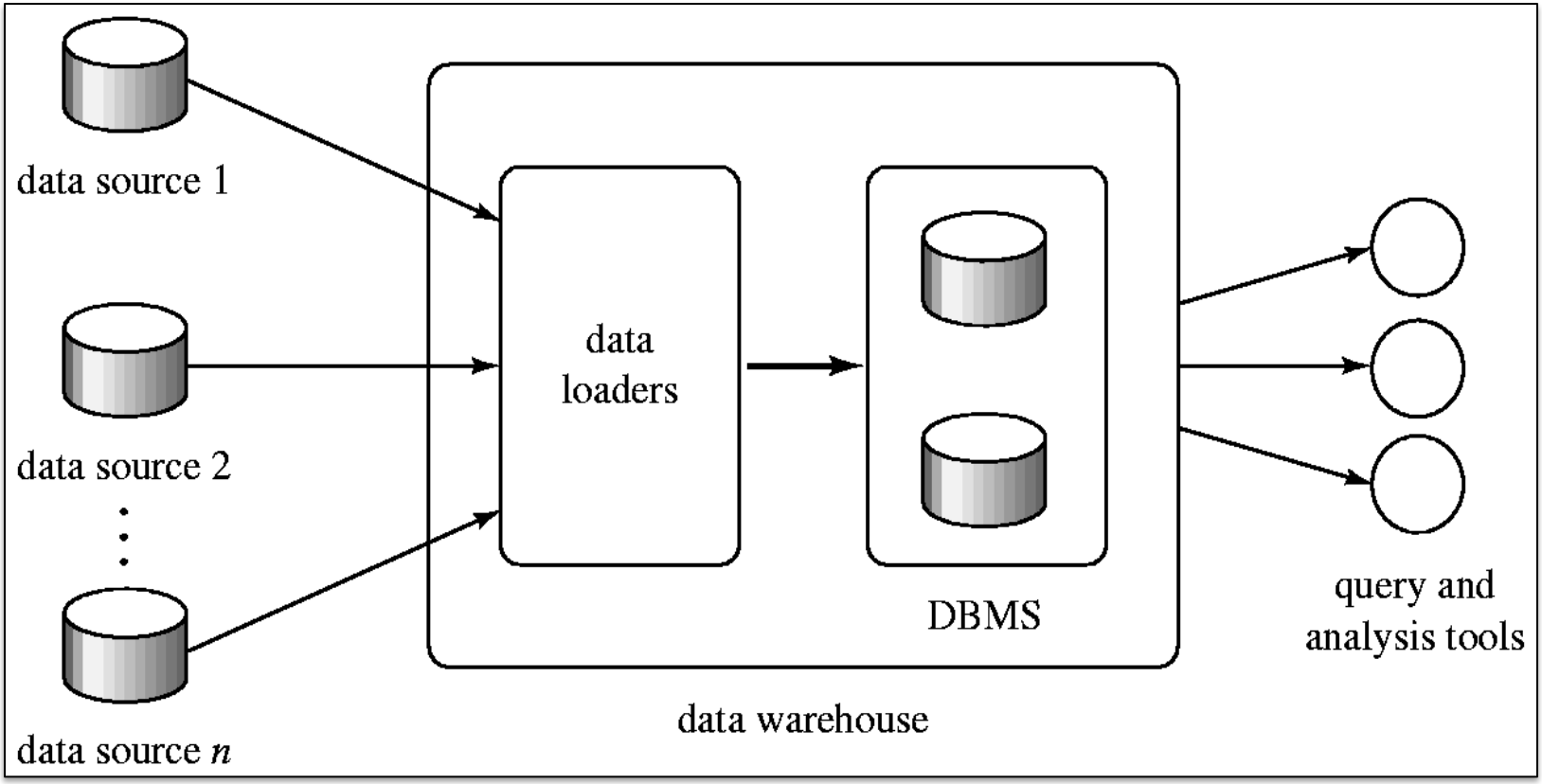
- Increasingly, organisations are analysing current and historical data to identify useful patterns and support business strategies.
- Emphasis is on complex, interactive, exploratory analysis of very large datasets created by integrating data from across all parts of an enterprise.
- Three main trends:
  - Data Warehousing: Consolidate data from many sources in one large repository, e.g., loading, periodic synchronisation of replicas, semantic integration.
  - OLAP, e.g., complex SQL queries and views, queries based on spreadsheet-style operations and multidimensional" view of data, interactive and "online" queries.
  - Data Mining: exploratory search for interesting trends and anomalies.

Data Warehousing

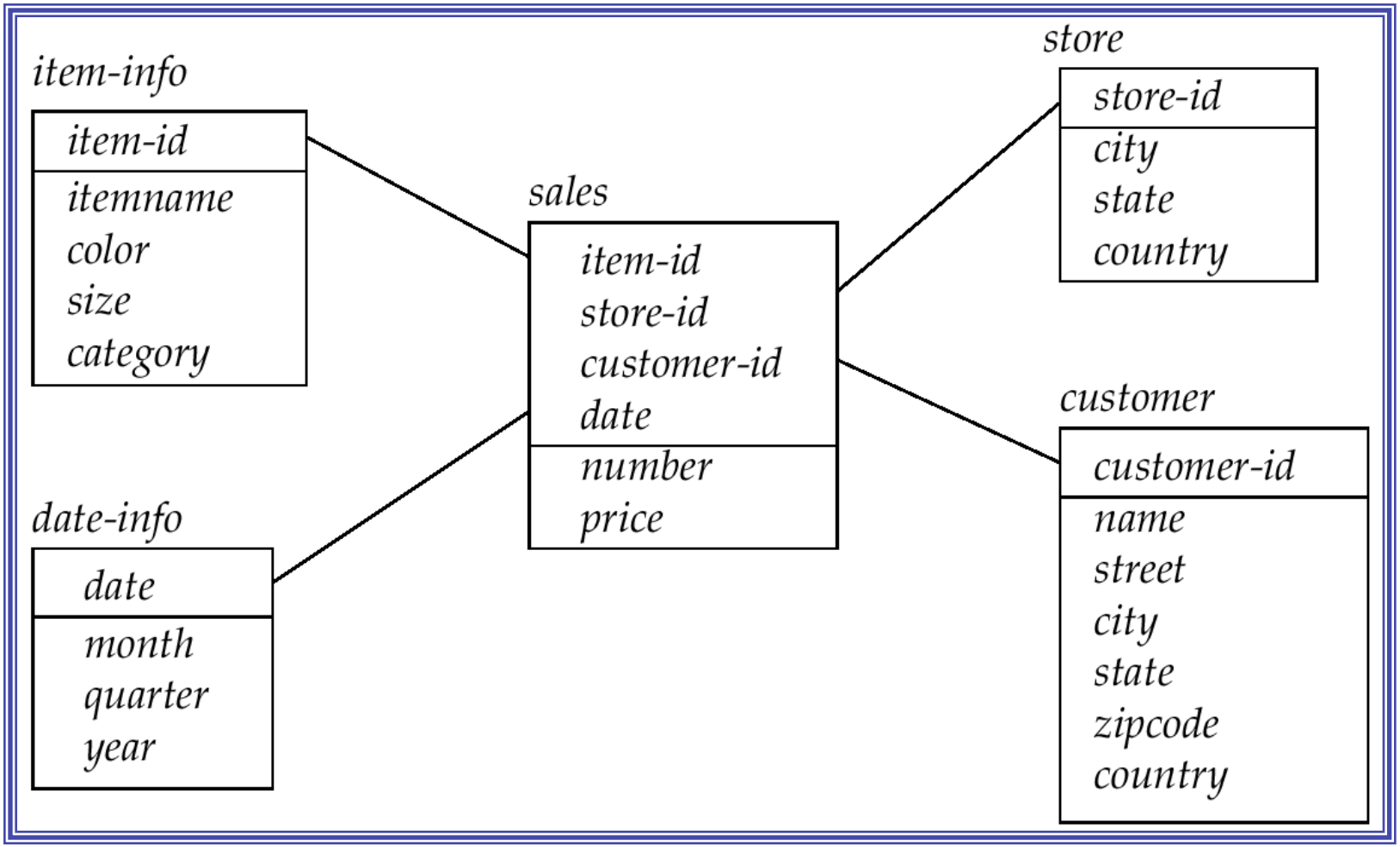
- Integrated data spanning long time periods, often augmented with summary information (ETL: Extract-Transform-Load).
- Several gigabytes to terabytes common.
- Interactive response times expected for complex queries.
- Ad-hoc updates uncommon.



Data Warehouse Architecture



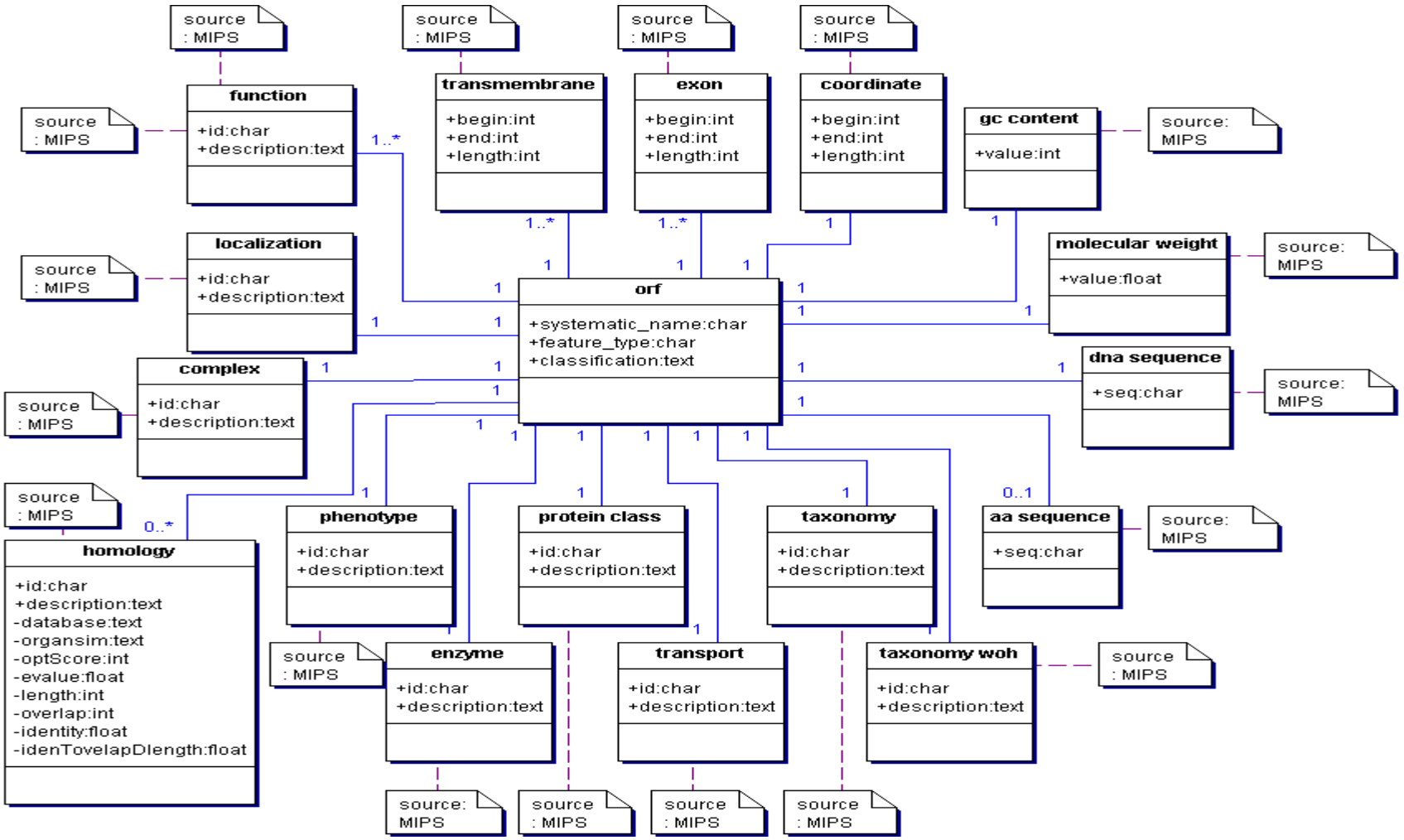
Data Warehouse Schema Example



BLID (Bio-Logical Intelligent Database)

- BLID (Bio-Logical Intelligent Database) is a bioinformatics system designed to help biologists extract new knowledge from raw genome data by providing high-level facilities for both data browsing and analysis.
- Providing intelligent queries and improving scalability.
  - The Yeast genome is the most well-characterised eukaryotic genome and one of the simplest in terms of identifying open reading frames (ORFs).
  - Over 6,000 ORFs, yet about 40% ORFs have no known function.
  - Existing Yeast Databases (KEGG, MIPS, SGD etc.) only support data queries, not knowledge queries.
  - Most existing KDD methods working with dataset, poor scalability.

Example: BLID Relational schema



Warehousing Issues

- Heterogeneous Sources: may access data from a variety of sources and repositories.
- Semantic Integration: when getting data from multiple sources, must eliminate mismatches, e.g., different currencies, schemas.
- Load, Refresh, Purge: must load data, periodically refresh it, and purge too-old data.
- Metadata Management: must keep track of source, loading time, and other information for all data in the warehouse.

Data Analysis and OLAP

- **Online Analytical Processing (OLAP)**
  - Interactive analysis of data, allowing data to be summarised and viewed in different ways in an online fashion (with negligible delay)
- Data that can be modeled as dimension attributes and measure attributes are called **multidimensional data**.
  - As a example, given *sales* relation on clothes shop

    sales(item-name, color, size, number)
  - **Measure attributes**
    - measure some value
    - can be aggregated upon
    - e.g., the attribute *number* of the *sales* relation
  - **Dimension attributes**
    - define the dimensions on which measure attributes (or aggregates thereof) are viewed
    - e.g., the attributes *item_name, color,* and *size* of the *sales* relation

Cross Tabulation of ***sales*** by ***item-name*** and ***color***

size: **all**

item-name \ *color*	dark	pastel	white	Total
skirt	8	35	10	53
dress	20	10	5	35
shirt	14	7	28	49
pant	20	2	5	27
Total	62	54	48	164

- The table above is an example of a cross-tabulation (cross-tab), also referred to as a pivot-table.
  - Values for one of the dimension attributes form the row headers
  - Values for another dimension attribute form the column headers
  - Other dimension attributes are listed on top
  - Values in individual cells are (aggregates of) the values of the dimension attributes that specify the cell.

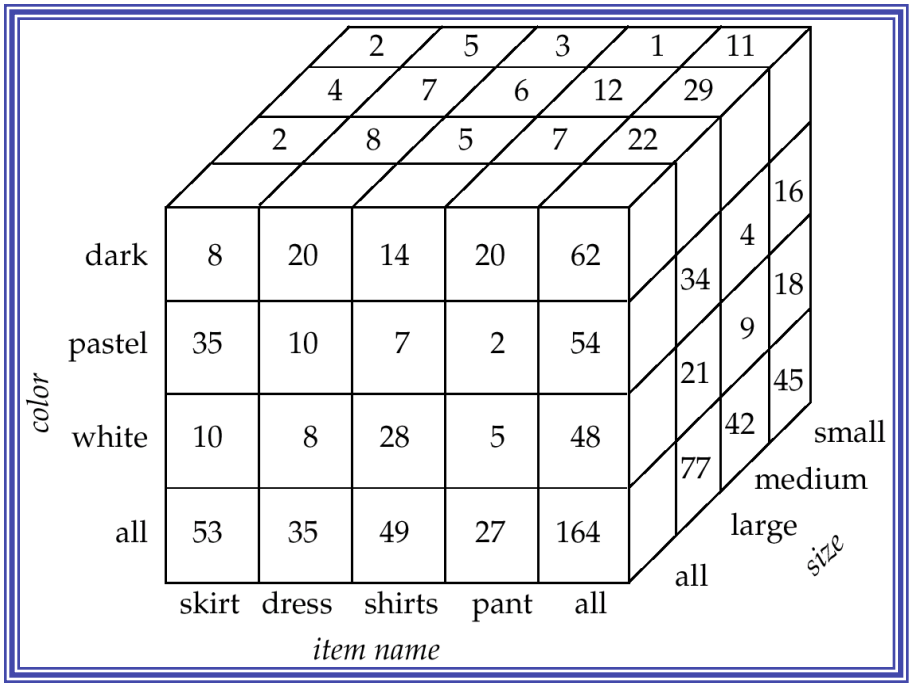
Relational Representation of Cross-tabs

- Cross-tabs can be represented as relations
- We use the value **all** to represent aggregates
  - The SQL:1999 standard actually uses null values in place of **all** despite confusion with regular null values

item-name	*color*	*number*
skirt	dark	8
skirt	pastel	35
skirt	white	10
skirt	**all**	53
dress	dark	20
dress	pastel	10
dress	white	5
dress	**all**	35
shirt	dark	14
shirt	pastel	7
shirt	white	28
shirt	**all**	49
pant	dark	20
pant	pastel	2
pant	white	5
pant	**all**	27
all	dark	62
all	pastel	54
all	white	48
all	**all**	164

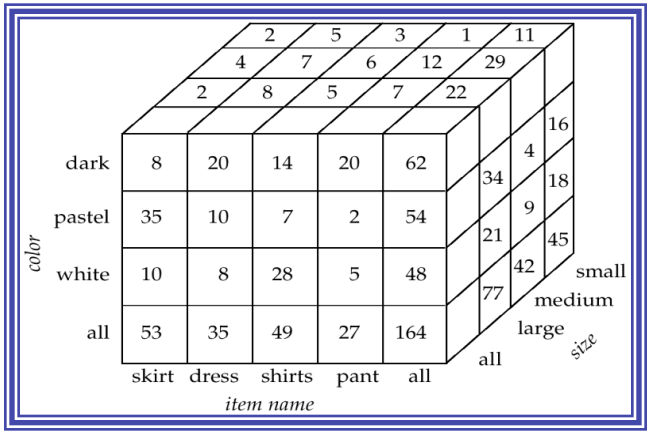
Data Cube

- A **data cube** is a multidimensional generalisation of a cross-tab
- Can have *n* dimensions; we show 3 below
- Cross-tabs can be used as views on a data cube



OLAP Queries

- **Pivoting**: changing the dimensions used in a cross-tab.
- **Slicing**: creating a cross-tab for fixed values only.
  - Sometimes called **dicing**, particularly when values for multiple dimensions are fixed.
- **Rollup**: moving from finer-granularity data to a coarser granularity.
  - the following is the rolling up on the attribute *size*



Rolling up on size

size: **all**

color

item-name	dark	pastel	white	Total
skirt	8	35	10	53
dress	20	10	5	35
shirt	14	7	28	49
pant	20	2	5	27
Total	62	54	48	164

- **Drill down**: The opposite operation - that of moving from coarser-granularity data to finer-granularity data

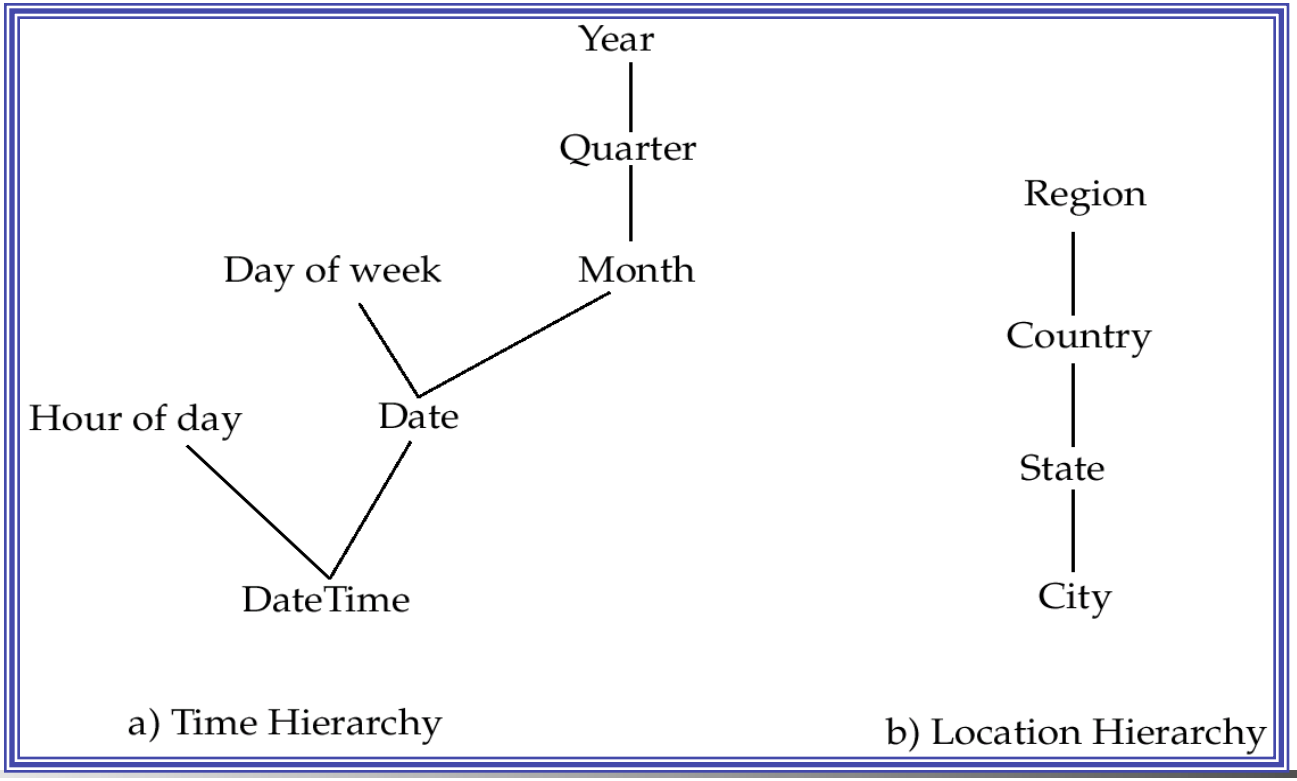
Slicing (Dicing) Query

- The general form of "slicing or dicing" query looks like

SELECT grouping attributes and aggregations
FROM fact table joined with zero or more dimension tables
WHERE certain attributes are constant
GROUP BY grouping attributes

Hierarchies on Dimensions

- **Hierarchy** on dimension attributes: lets dimensions to be viewed at different levels of detail
- E.g. the dimension DateTime can be used to aggregate by hour of day, date, day of week, month, quarter or year



Cross Tabulation With Hierarchy

- Cross-tabs can be easily extended to deal with hierarchies
- Can drill down or roll up on a hierarchy
- Size: ALL

category	*item-name*					
		dark	pastel	white	total	
womenswear	skirt	8	35	10	53	
	dress	20	10	5	35	
	subtotal	28	45	15		88
menswear	pants	14	7	28	49	
	shirt	20	2	5	27	
	subtotal	34	9	33		76
total		62	54	48		164

Data Mining

- Data mining is the exploration and analysis of large amount of data in order to discover valid, novel, potentially useful, and ultimately understandable patterns in data.
  - Valid - The patterns hold in general.
  - Novel - We did not know the pattern beforehand.
  - Useful - We can devise actions from the patterns.
  - Understandable - We can interpret and comprehend the patterns.

What is a Data Mining Model?

- A data mining model is a description of a certain aspect of a dataset. It produces output values for an assigned set of inputs.
- Well studied examples:
  - Clustering
  - Regression
  - Classification
  - Frequent itemsets and association rules
  - Etc.

Classification

- Goal: learn a function that assigns a record to one of several pre-defined classes.
- Requirements on the model:
  - High accuracy
  - Understandable and interpretable by humans
  - Fast construction for very large training databases

Definitions

- Random variables $X_1, \ldots, X_k$ (*predictor variables*) and Y (*dependent variable*)
- X_i has domain $dom(X_i)$, Y has domain $dom(Y)$
- P is a probability distribution on $dom(X_1) \times \ldots \times dom(X_k) \times dom(Y)$; training database D is a random sample from P
- *A predictor* d is a function $d: dom(X_1) \ldots dom(X_k) \rightarrow dom(Y)$

Types of Variables

- *Numerical*: Domain is ordered and can be represented on the real line (e.g., age, income)
- *Nominal* or *categorical*: Domain is a finite set without any natural ordering (e.g., occupation, marital status, race)
- *Ordinal*: Domain is ordered, but absolute differences between values is unknown (e.g., preference scale, severity of an injury)

Classification Problem

- If Y is categorical, the problem is a *classification problem*, and we use C instead of Y. |dom(C)| = J, the number of classes.
- C is the *class label*, d is called a *classifier.*
- Let r be a record randomly drawn from P. Define the *misclassification rate* of d:

$$RT(d,P) = Prop(d(r.X_1, \ldots, r.X_k) != r.C)$$

- Problem definition: Given dataset D that is a random sample from probability distribution P, find classifier d such that RT(d,P) is minimised.

Regression Problem

- If Y is numerical, the problem is a *regression problem.*
- Y is called the dependent variable, d is called a *regression function.*
- Let r be a record randomly drawn from P. Define **mean squared error** rate of d:

$$RT(d,P) = E(r.Y - d(r.X_1, \ldots, r.X_k))^2$$

- Problem definition: Given dataset D that is a random sample from probability distribution P, find regression function d such that RT(d,P) is minimised.

Classification Example

- Training database:
  - Two predictor attributes: *Age* and *Car*-type (*Sport*, *Minivan* and *Truck*). Age is numerical, Car-type is categorical.
  - Class label indicates whether the person will buy the product
  - Dependent attribute is *categorical*

Age	Car	Class
20	M	Yes
30	M	Yes
25	T	No
30	S	Yes
40	S	Yes
20	T	No
30	M	Yes
25	M	Yes
40	M	Yes
20	S	No

Regression Example

- Example training database
  - Two predictor attributes: *Age* and *Car type* (**S**port, **M**inivan and **T**ruck)
  - Spent indicates how much the person spent for online shopping per month
  - Dependent attribute is *numerical*

Age	Car	Spent
20	M	$200
30	M	$150
25	T	$300
30	S	$220
40	S	$400
20	T	$80
30	M	$100
25	M	$125
40	M	$500
20	S	$420

Approaches for Classification

- Decision trees are one approach for classification.
- Other approaches include:
  - Linear Discriminant Analysis
  - Naïve Bayes
  - *k*-nearest neighbor
  - Logistic regression
  - Neural networks
  - Support Vector Machines
  - Adaboosting
  - Random forests
  - etc

Decision Trees

- A *decision tree* T encodes d (a classifier or regression function) in form of a tree
- A node t in T without children is called a *leaf node*. Otherwise t is called an *internal node* (except the root)
- Classification problem: Each leaf node is labeled with one class label c in dom(C)
- Each internal node has an associated splitting predicate. Common ones are binary predicates. Example predicate:
  - Age <= 20

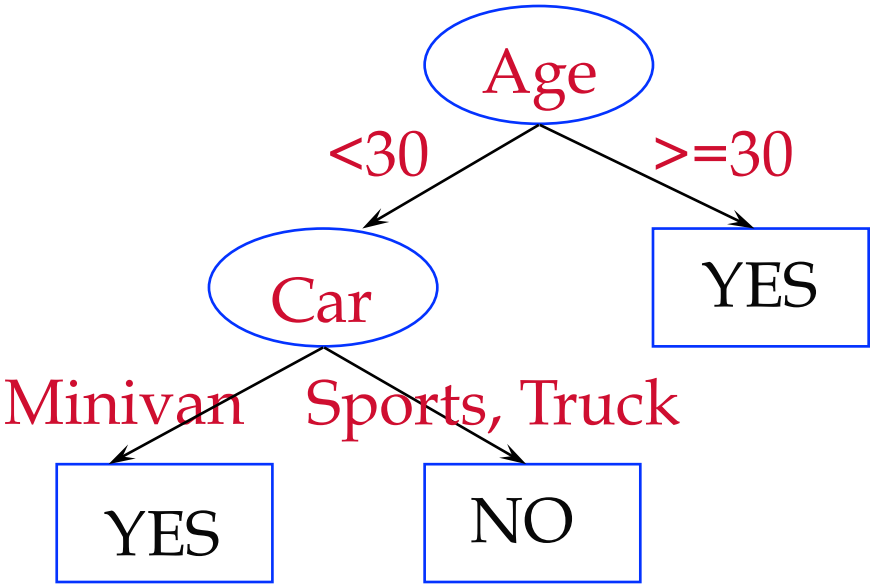
Example

Encoded classifier as rules:

If (age<30 and
carType=Minivan)
Then YES

If (age <30 and
(carType=Sports or
carType=Truck))
Then NO

If (age >= 30)
Then YES



Age	Car	Class
20	M	Yes
30	M	Yes
25	T	No
30	S	Yes
40	S	Yes
20	T	No
30	M	Yes
25	M	Yes
40	M	Yes
20	S	No

Construction of Decision Trees

- **Training set**: data samples in which the classification is already known.
- **Greedy top down** generation of decision trees.
  - Each internal node of the tree partitions the data into groups based on a **partitioning attribute**, and a **partitioning condition** for the node
  - **Leaf** node:
    - all (or most) of the items at the node belong to the same class, or
    - all attributes have been considered, and no further partitioning is possible.

Best Splits

- Pick best attributes and conditions to partition
- The purity of a set S of training instances can be measured quantitatively in several ways.
  - Notation: number of classes = k, number of instances = $|S|$, fraction of instances in class i: p_i (= (# of instances of class i in S)/$|S|$
- The purity(S) is defined as
  - Gini measure

$$\text{Gini}(S) = 1 - \sum_{i=1}^{k} p_i^2$$

  - or Entropy

$$\text{Entropy}(S) = -\sum_{i=1}^{k} p_i \log_2 p_i$$

  - When all instances are in a single class, both have values 0
  - Both reaches their maximum (for Gini measure is $1 - 1/k$) if each class has the same number of instances.

Best Splits cont'd

- When a set S is split into multiple sets S_i, i=1, 2, …, r, we can measure the purity of the resultant set of sets as (also known as average entropy or information):

$$\text{purity}(S_1, S_2, \ldots\ldots, S_r) = \sum_{i=1}^{r} \frac{|S_i|}{|S|} \text{purity}(S_i)$$

- The information gain due to particular split of S into S_i, i = 1, 2, …., r

$$\text{Information-gain}(S, \{S_1, S_2, \ldots., S_r) = \text{purity}(S) - \text{purity}(S_1, S_2, \ldots S_r)$$

Best Splits cont'd

- Measure of "cost" of a split (also known as intrinsic information):

$$\text{Information-content}(S, \{S_1, S_2, \ldots\ldots, S_r\})) = -\sum_{i=1}^{r} \frac{|S_i|}{|S|} \log_2 \frac{|S_i|}{|S|}$$

- **Information-gain ratio** $= \dfrac{\text{Information-gain}(S, \{S_1, S_2, \ldots\ldots, S_r\})}{\text{Information-content}(S, \{S_1, S_2, \ldots\ldots, S_r\})}$

- The best split is the one that gives the maximum information gain ratio.

Finding Best Splits

- Categorical attributes (with no meaningful order):
  - Multi-way split, one child for each value
  - Binary split: try all possible breakup of values into two sets, and pick the best
- Continuous-valued attributes (can be sorted in a meaningful order)
  - Binary split:
    - Sort values, try each as a split point, e.g. if values are 1, 10, 15, 25, split at ≤1, ≤ 10, ≤ 15
    - Pick the value that gives best split
  - Multi-way split:
    - A series of binary splits on the same attribute has roughly equivalent effect

Decision-Tree Construction Algorithm

Procedure *GrowTree* (*S: Data set*, δ_p: *Real*, δ_s: *Nat*)
  Partition (*S*, δ_p , δ_s);

Procedure Partition (*S*)
  if (*purity* (*S*) < δ_p or |*S*| < δ_s) **then**
    return;
  for each attribute *A*
    evaluate splits on attribute *A*;
  Use best split found (across all attributes) to partition
    S into S_1, S_2, …., S_r,
  for *i* = 1, 2, ….., *r*
    Partition (S_i , δ_p , δ_s);

End of Lecture

- Summary
  - Data warehouse
  - OLAP: Online Analytical Processing
  - Introduction to Data mining: Classification and Decision Trees

- Reading
  - Textbook 6th edition, chapter 20
  - Textbook 7th edition, chapter 11
  - Tutorial on decision tree slides 1-28

Database Development and Design (CPT201)

Lecture 13b: Data Mining 2 – Clustering and Market Basket Analysis

Dr. Wei Wang

Department of Computing

Learning Outcome

- Intro to Clustering
- Intro to Market Basket Analysis

Problem

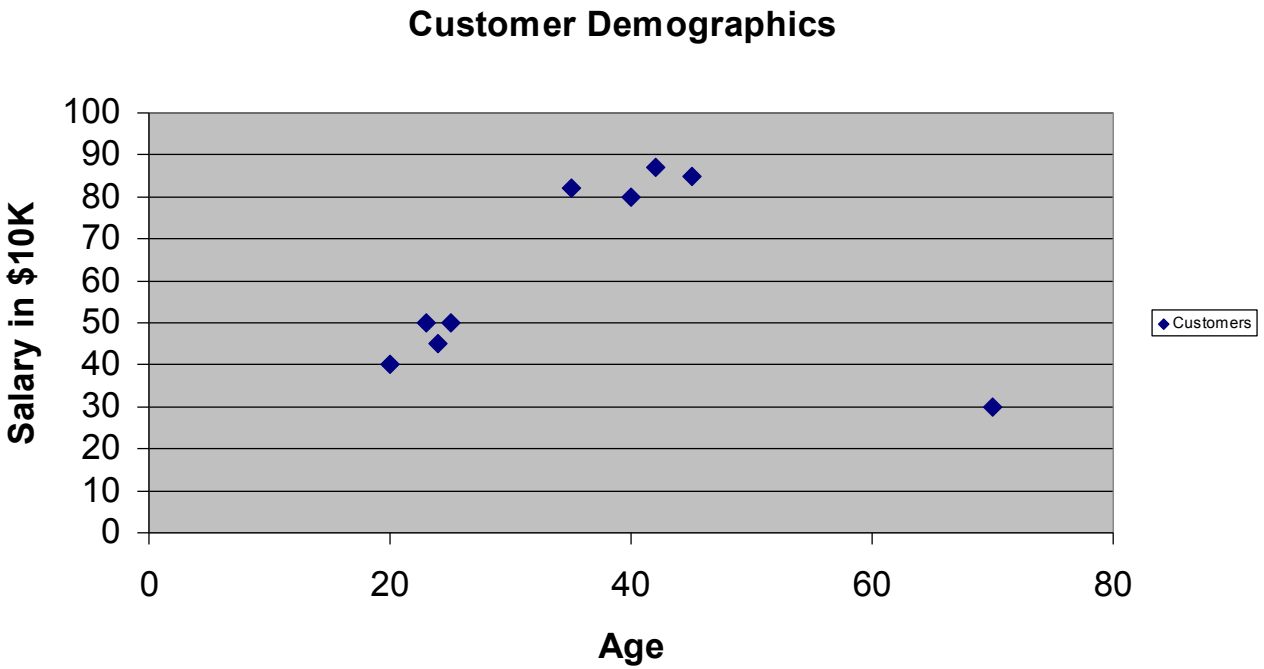
- Given data points in a multidimensional space, group them into a number of clusters, using some measure of "nearness", e.g.,
  - cluster documents by topic
  - cluster users by similar interests

Clustering

- Output: *k* groups of records called clusters, such that the records within a group are more similar than records in other groups
  - Representative points for each cluster
  - Labeling of each record with each cluster number
- This is unsupervised learning: no record labels are given to learn from
- Usage:
  - Exploratory data mining
  - Preprocessing step (e.g. outlier detection)

An Example of Clustering

- Example input database: two numerical variables
- How many groups are here?



Age	Salary
20	40
25	50
24	45
23	50
40	80
45	85
42	87
35	82
70	30

Similarity

- Need to define "similarity" between records
- Use the "right" similarity (distance) function
  - Scale or normalise all attributes. Example: seconds, hours, days
  - Assign different weights to reflect importance of the attribute
  - Choose appropriate measure

Properties of Distances: Metric Spaces

- A metric space is a set S with a global distance function d. For every two points x, y in S, the distance d(x,y) is a nonnegative real number.
- A metric space must also satisfy
  - d(x,y) = 0 iff x = y
  - d(x,y) = d(y,x) (symmetry)
  - d(x,y) + d(y,z) >= d(x,z) (triangle inequality)

Minkowski Distance (L_p Norm)

- There exist a lot of definitions of distance. Minkowski Distance is one of them.
- Consider two records $x=(x_1,\ldots,x_d)$, $y=(y_1,\ldots,y_d)$, Minkowski Distance is defined by:

$$d(x,y)=\sqrt[p]{|x_1-y_1|^p+|x_2-y_2|^p+\cdots+|x_d-y_d|^p}$$

- Special cases:
  - p=1: Manhattan distance

$$d(x,y) = |x_1-y_1|+|x_1-y_2|+\ldots+|x_d-y_d|$$

  - p=2: Euclidean distance

$$\sqrt[2]{|x_1-y_1|^2+|x_2-y_2|^2+\cdots+|x_d-y_d|^2}$$

K-means Clustering Algorithm

- Choose *k* initial means
- Assign each point to the cluster with the closest mean
- Compute new mean for each cluster
- Iterate until the *k* means stabilise

Example

We want to cluster the data into three groups.

- Randomly assign three initial means (centroid)
  - μ_1=(20, 40), μ_2=(40, 80), μ_3=(70, 30)
- Each mean represents a cluster. Assign each sample into a cluster based on its distance to the mean.
  - C_1={(20, 40), (25, 50), (24, 45), (23, 50)}
  - C_2={(40, 80), (45, 85), (42, 87), (35, 82)}
  - C_3={(70, 30)}
- Compute and update new mean for each cluster:
  - μ_1=(23, 46.25), μ_2=(40.5, 83.5), μ_3=(70, 30)
- Repeat previous two steps until changes are less a pre-defined threshold.

Age	Salary
20	40
25	50
24	45
23	50
40	80
45	85
42	87
35	82
70	30

Market Basket Analysis

- Consider a shopping cart filled with a number of items.
- Market basket analysis tries to answer questions similar to the following:
  - Who makes purchases?
  - What do customers buy?
  - What they buy together?

Market Basket Analysis cont'd

- Given:
  - A database of customer transactions
  - Each transaction consists of a set of items
  - TID: transaction ID
  - CID: customer ID
- Goal:
  - Extract rules

TID	CID	Date	Item	Qty
111	201	5/1/99	Pen	2
111	201	5/1/99	Ink	1
111	201	5/1/99	Milk	3
111	201	5/1/99	Juice	6
112	105	6/3/99	Pen	1
112	105	6/3/99	Ink	1
112	105	6/3/99	Milk	1
113	106	6/5/99	Pen	1
113	106	6/5/99	Milk	1
114	201	7/1/99	Pen	2
114	201	7/1/99	Ink	2
114	201	7/1/99	Juice	4

Frequent Itemsets

- Itemset is a set of items
- The support of an itemset is the fraction of transactions in the database that contain all items in the itemset.
- Given a minimum support *minsup*, Frequent itemsets with respect to the minimum support are the itemsets whose support is higher than *minsup*.
- The "A Priori Property": Every subset of a frequent itemset is also a frequent itemset.

Example

Given a minsup = 0.6, compute all frequent itemsets.

{pen}, {ink}, {milk}, {pen, ink}, {pen, milk},

Market Basket Analysis

- Co-occurrences
  - 80% of all customers purchase items X, Y and Z together.
- Association rules
  - 60% of all customers who purchase X and Y also buy Z.
- Sequential patterns
  - 60% of customers who first buy X also purchase Y within three weeks.

Confidence and Support of Rules

- We prune the set of all possible association rules using two measures:
- Support of a rule:
  - X => Y has support s if P(X, Y) = s
  - (#of transactions contain both X and Y/total of transactions)
- Confidence of a rule:
  - X => Y has confidence c if P(Y|X) = c
  - (#of transactions contain X and Y/# of transactions contain X)
- We can also define Support of a co-occurrence XY:
  - XY has support s if P(X, Y) = s
  - (#of transactions contain X and Y/total of transactions)
  - Same as X => Y

Examples and Questions

- Treat each transaction as a market basket
  - Example rule: {Pen} => {Milk}
    Support = 75%
    Confidence = 75%
  - Another example: {Ink} => {Pen}
    Support = 75%
    Confidence = 100%
- Treat each customer as a market basket
  - Example rule: {Pen} => {Milk}
    Support = 100%
    Confidence = 100%
  - Another example: {Ink} => {Pen}
    Support = 66.67%
    Confidence = 100%

TID	CID	Date	Item	Qty
111	201	5/1/99	Pen	2
111	201	5/1/99	Ink	1
111	201	5/1/99	Milk	3
111	201	5/1/99	Juice	6
112	105	6/3/99	Pen	1
112	105	6/3/99	Ink	1
112	105	6/3/99	Milk	1
113	106	6/5/99	Pen	1
113	106	6/5/99	Milk	1
114	201	7/1/99	Pen	2
114	201	7/1/99	Ink	2
114	201	7/1/99	Juice	4

Other Examples

- Can you find all itemsets with support >= 75%?
- Can you find all association rules with confidence >= 50%?

TID	CID	Date	Item	Qty
111	201	5/1/99	Pen	2
111	201	5/1/99	Ink	1
111	201	5/1/99	Milk	3
111	201	5/1/99	Juice	6
112	105	6/3/99	Pen	1
112	105	6/3/99	Ink	1
112	105	6/3/99	Milk	1
113	106	6/5/99	Pen	1
113	106	6/5/99	Milk	1
114	201	7/1/99	Pen	2
114	201	7/1/99	Ink	2
114	201	7/1/99	Juice	4

A priori algorithm

- General idea: only sets with single items are considered in the first pass. In the second pass, sets with two items are considered, and so on…
- At the end of a pass, all sets with sufficient support are output as large itemsets.
  - Sets found to have too little support at the end of the pass are eliminated.
  - Once a set is eliminated, none of its supersets needs to be considered.
- At the end of some pass *i*, we would find that no set of size *i* has sufficient support, so we do not need to consider any set of size *i+1*.
- Computation then terminates.

Extensions

- Imposing constraints
  - Only find rules involving the dairy department
  - Only find rules involving expensive products
  - Only find rules with "whiskey" on the right hand side
  - Only find rules with "milk" on the left hand side
  - Hierarchies on the items
  - Calendars (every Sunday, every 1st of the month)

Market Basket Analysis: Applications

- Direct marketing
- Fraud detection for medical insurance
- Floor/shelf planning
- Web site layout
- Cross-selling
- etc…

End of Lecture

- Summary
  - Intro to Clustering
  - Intro to Market Basket Analysis

- Reading
  - Textbook 6th edition, chapter 20
  - Textbook 7th edition, chapter 11