

Misc Topics

Amol Deshpande
CMSC424

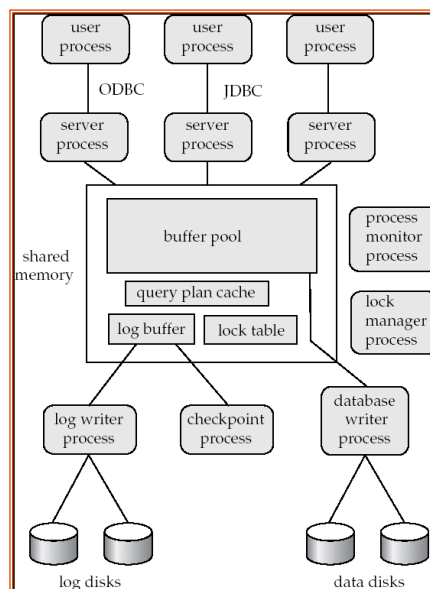
Topics

- Today
 - ★ Database system architectures
 - Client-server
 - ★ Parallel and Distributed Systems
 - ★ Object Oriented, Object Relational
 - ★ XML
 - ★ OLAP
 - ★ Data Warehouses
 - ★ Information Retrieval

Database System Architectures

- Centralized single-user
- *Client-Server Architectures*
 - ★ Connected over a network typically
 - ★ Back-end: manages the database
 - ★ Front-end(s): Forms, report-writers, *sqlplus*
 - ★ How they talk to each other ?
 - ODBC:
 - Interface standard for talking to the server in C
 - JDBC:
 - In Java
 - ★ Transaction servers vs. data servers

Database System Architectures

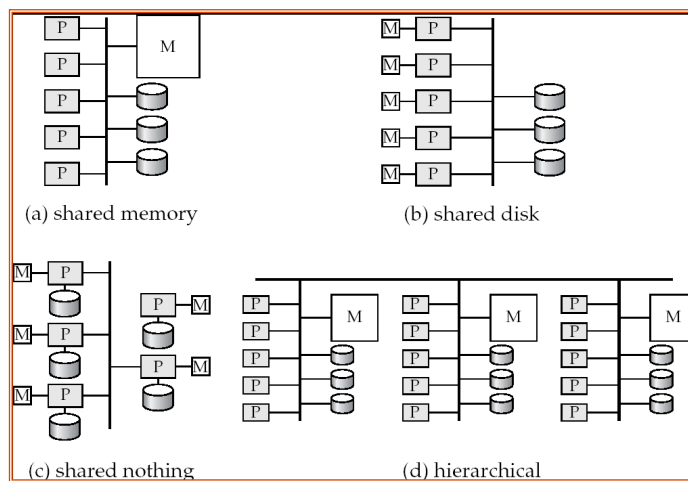


Parallel Databases

- Why ?
 - ★ More transactions per second, or less time per query
 - ★ Throughput vs. Response Time
 - ★ Speedup vs. Scaleup
- Database operations are *embarrassingly parallel*
 - ★ E.g. Consider a join between R and S on R.b = S.b
- But, perfect speedup doesn't happen
 - ★ Start-up costs
 - ★ Interference
 - ★ Skew

Parallel Databases

- Shared-nothing vs. shared-memory vs. shared-disk



Parallel Databases

	Shared Memory	Shared Disk	Shared Nothing
Communication between processors	Extremely fast	Disk interconnect is very fast	Over a LAN, so slowest
Scalability ?	Not beyond 32 or 64 or so (memory bus is the bottleneck)	Not very scalable (disk interconnect is the bottleneck)	Very very scalable
Notes	Cache-coherency an issue	Transactions complicated; natural fault-tolerance.	Distributed transactions are complicated (deadlock detection etc);
Main use	Low degrees of parallelism	Not used very often	Everywhere

Distributed Systems

- Over a wide area network
- Typically not done for *performance reasons*
 - ★ For that, use a parallel system
- Done because of necessity
 - ★ Imagine a large corporation with offices all over the world
 - ★ Also, for redundancy and for disaster recovery reasons
- Lot of headaches
 - ★ Especially if trying to execute transactions that involve data from multiple sites
 - Keeping the databases in sync
 - 2-phase commit for transactions uniformly hated
 - Autonomy issues
 - Even within an organization, people tend to be protective of their unit/department
 - Locks/Deadlock management
 - ★ Works better for query processing
 - Since we are only reading the data

Next...

- Object oriented, Object relational, XML

Motivation

- Relational model:
 - ★ Clean and simple
 - ★ Great for much enterprise data
 - ★ But lot of applications where not *sufficiently rich*
 - Multimedia, CAD, for storing *set data* etc
- Object-oriented models in programming languages
 - ★ Complicated, but very useful
 - Smalltalk, C++, now Java
 - ★ Allow
 - Complex data types
 - Inheritance
 - Encapsulation
- People wanted to manage objects in databases.

History

- In the 1980's and 90's, DB researchers recognized benefits of objects.
- Two research thrusts:
 - ★ OODBMS: extend C++ with transactionally persistent objects
 - Niche Market
 - CAD etc
 - ★ ORDBMS: extend Relational DBs with object features
 - Much more common
 - Efficiency + Extensibility
 - SQL:99 support
- Postgres – First ORDBMS
 - ★ Berkeley research project
 - ★ Became Illustra, became Informix, bought by IBM

Object-Relational: Example

- Create User Defined Types (UDT)

```
CREATE TYPE BarType AS (  
  name CHAR(20),  
  addr CHAR(20)  
);  
CREATE TYPE BeerType AS (  
  name CHAR(20),  
  manf CHAR(20)  
);  
CREATE TYPE MenuType AS (  
  bar REF BarType,  
  beer REF BeerType,  
  price FLOAT  
);
```
- Create Tables of UDTs
 - ★ CREATE TABLE Bars OF BarType;
 - ★ CREATE TABLE Beers OF BeerType;
 - ★ CREATE TABLE Sells OF MenuType;

Example

- Querying:
 - ★ SELECT * FROM Bars;
 - ★ Produces “tuples” such as:
 - BarType('Joe's Bar', 'Maple St.')
- Another query:
 - ★ SELECT bb.name(), bb.addr()
 - ★ FROM Bars bb;
- Inserting tuples:
 - ★ SET newBar = BarType();
 - ★ newBar.name('Joe's Bar');
 - ★ newBar.addr('Maple St.');
 - ★ INSERT INTO Bars VALUES(newBar);

Example

- UDT's can be used as types of attributes in a table

```
CREATE TYPE AddrType AS (  
    street CHAR(30),  
    city CHAR(20),  
    zip INT  
);  
CREATE TABLE Drinkers (  
    name CHAR(30),  
    addr AddrType,  
    favBeer BeerType  
);
```
- Find the beers served by Joe:

```
SELECT ss.beer()->name  
FROM Sells ss  
WHERE ss.bar()->name = 'Joe's Bar';
```

An Alternative: OODBMS

- Persistent OO programming
 - ★ Imagine declaring a Java object to be “persistent”
 - ★ Everything reachable from that object will also be persistent
 - ★ You then write plain old Java code, and all changes to the persistent objects are stored in a database
 - ★ When you run the program again, those persistent objects have the same values they used to have!
- Solves the “impedance mismatch” between programming languages and query languages
 - ★ E.g. converting between Java and SQL types, handling rowsets, etc.
 - ★ But this programming style doesn’t support declarative queries
 - For this reason (??), OODBMSs haven’t proven popular
- OQL: A declarative language for OODBMSs
 - ★ Was only implemented by one vendor in France (Altair)

OODBMS

- Currently a Niche Market
 - ★ Engineering, spatial databases, physics etc...
- Main issues:
 - ★ Navigational access
 - Programs specify go to this object, follow this pointer
 - ★ Not declarative
- Though advantageous when you know exactly what you want, not a good idea in general
 - ★ Kinda similar argument as *network databases vs relational databases*

Summary, cont.

- ORDBMS offers many new features
 - ★ but not clear how to use them!
 - ★ schema design techniques not well understood
 - No good logical design theory for non-1st-normal-form!
 - ★ query processing techniques still in research phase
 - a moving target for OR DBA's!

- OODBMS
 - ★ Has its advantages
 - ★ Niche market
 - ★ Lot of similarities to XML as well...

XML

- Extensible Markup Language
- Derived from SGML (Standard Generalized Markup Language)
 - ★ Similar to HTML, but HTML is not *extensible*
 - Extensible == can add new tags etc
- Emerging as the *wire format (data interchange format)*

XML

```
<bank-1>
  <customer>
    <customer-name> Hayes </customer-name>
    <customer-street> Main </customer-street>
    <customer-city> Harrison </customer-city>
    <account>
      <account-number> A-102 </account-number>
      <branch-name> Perryridge </branch-name>
      <balance> 400 </balance>
    </account>
    <account>
      ...
    </account>
  </customer>
  .
  .
</bank-1>
```

Attributes

- Elements can have **attributes**

```
<account acct-type = "checking" >
  <account-number> A-102 </account-number>
  <branch-name> Perryridge </branch-name>
  <balance> 400 </balance>
</account>
```
- 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
 - `<account acct-type = "checking" monthly-fee="5">`

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 two ways
 - `<account account-number = "A-101"> </account>`
 - `<account>`
`<account-number>A-101</account-number> ...`
`</account>`
 - ★ Suggestion: use attributes for identifiers of elements, and use subelements for contents

Namespaces

- XML data has to be exchanged between organizations
- Same tag name may have different meaning in different organizations, 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

```
<bank xmlns:FB='http://www.FirstBank.com'>
...
  <FB:branch>
    <FB:branchname>Downtown</FB:branchname>
    <FB:branchcity> Brooklyn </FB:branchcity>
  </FB:branch>
...
</bank>
```

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) >

Bank DTD

```
<!DOCTYPE bank [  
  <!ELEMENT bank ( ( account | customer | depositor)+)>  
  <!ELEMENT account (account-number branch-name balance)>  
  <! ELEMENT customer(customer-name customer-street  
                        customer-city)>  
  <! ELEMENT depositor (customer-name account-number)>  
  <! ELEMENT account-number (#PCDATA)>  
  <! ELEMENT branch-name (#PCDATA)>  
  <! ELEMENT balance(#PCDATA)>  
  <! ELEMENT customer-name(#PCDATA)>  
  <! ELEMENT customer-street(#PCDATA)>  
  <! ELEMENT customer-city(#PCDATA)>  
>
```

IDs and IDREFs

- An element can have at most one attribute of type ID
- The ID attribute value of each element in an XML document must be distinct
 - ★ Thus the ID attribute value is an object identifier
- An attribute of type IDREF must contain the ID value of an element in the same document

Bank DTD with Attributes

- Bank DTD with ID and IDREF attribute types.

```
<!DOCTYPE bank-2[
  <!ELEMENT account (branch, balance)>
  <!ATTLIST account
    account-number ID      # REQUIRED
    owners             IDREFS # REQUIRED>
  <!ELEMENT customer(customer-name, customer-street,
    custome-city)>
  <!ATTLIST customer
    customer-id  ID      # REQUIRED
    accounts    IDREFS # REQUIRED>
  ... declarations for branch, balance, customer-name,
    customer-street and customer-city
]>
```

XML data with ID and IDREF attributes

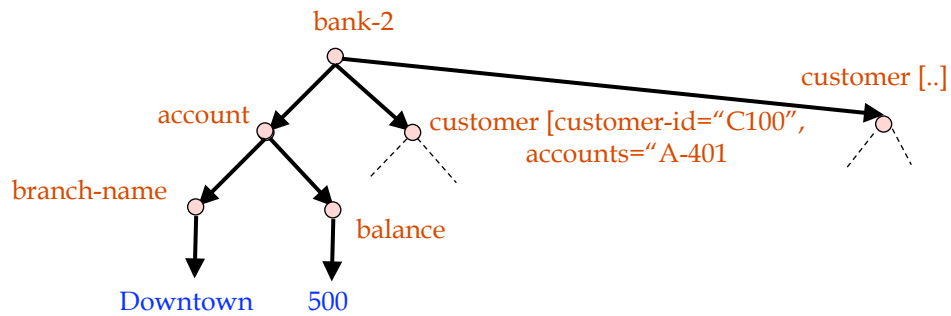
```
<bank-2>
  <account account-number="A-401" owners="C100 C102">
    <branch-name> Downtown </branch-name>
    <balance> 500 </balance>
  </account>
  <customer customer-id="C100" accounts="A-401">
    <customer-name>Joe </customer-name>
    <customer-street> Monroe </customer-street>
    <customer-city> Madison</customer-city>
  </customer>
  <customer customer-id="C102" accounts="A-401 A-402">
    <customer-name> Mary </customer-name>
    <customer-street> Erin </customer-street>
    <customer-city> Newark </customer-city>
  </customer>
</bank-2>
```

Querying and Transforming XML Data

- Standard XML querying/translation languages
 - ★ XPath
 - Simple language consisting of path expressions
 - Forms a basic component of the next two
 - ★ XSLT
 - Simple language designed for translation from XML to XML and XML to HTML
 - ★ XQuery
 - An XML query language with a rich set of features

Tree Model of XML Data

- Query and transformation languages are based on a tree model of XML data



XPath

- `/bank-2/customer/customer-name`
 - `<customer-name>Joe</customer-name>`
 - `<customer-name>Mary</customer-name>`
- `/bank-2/customer/customer-name/text()`
 - Joe
 - Mary
- `/bank-2/account[balance > 400]`
 - ★ returns account elements with a balance value greater than 400
- `/bank-2/account[balance > 400]/@account-number`
 - ★ returns the account numbers of those accounts with balance > 400

Functions in XPath

- `/bank-2/account[customer/count() > 2]`
 - ★ Returns accounts with > 2 customers
- Boolean connectives `and` and `or` and function `not()` can be used in predicates
- IDREFs can be referenced using function `id()`
 - ★ E.g. `/bank-2/account/id(@owner)`
 - returns all customers referred to from the owners attribute of account elements.

More XPath Features

- `//` can be used to skip multiple levels of nodes
 - ★ E.g. `/bank-2//customer-name`
 - finds any `customer-name` element *anywhere* under the `/bank-2` element, regardless of the element in which it is contained.
- Wild-cards
 - ★ `/bank-2/*/customer-name`
 - ★ Match any *element* name

XSLT

- A stylesheet stores formatting options for a document, usually separately from document
 - ★ E.g. 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

XSLT Templates

- Example of XSLT template with **match** and **select** part

```
<xsl:template match="/bank-2/customer">
  <xsl:value-of select="customer-name"/>
</xsl:template>
<xsl:template match="*" />
```
- The **match** attribute of **xsl:template** specifies a pattern in XPath
- Elements in the XML document matching the pattern are processed by the actions within the **xsl:template** element
 - ★ **xsl:value-of** selects (outputs) specified values (here, **customer-name**)
- For elements that do not match any template
 - ★ Attributes and text contents are output as is
 - ★ Templates are recursively applied on subelements
- The **<xsl:template match="*" />** template matches all elements that do not match any other template
 - ★ Used to ensure that their contents do not get output.

Creating XML Output

- Any text or tag in the XSL stylesheet that is not in the xsl namespace is output as is
- E.g. to wrap results in new XML elements.

```
<xsl:template match="/bank-2/customer">
  <customer>
    <xsl:value-of select="customer-name"/>
  </customer>
</xsl:template>
<xsl:template match="*" />
```

- ★ Example output:

```
<customer> Joe </customer>
<customer> Mary </customer>
```

XQuery

- XQuery is a general purpose query language for XML data
- Currently being standardized by the World Wide Web Consortium (W3C)
 - ★ The textbook description is based on a March 2001 draft of the standard. The final version may differ, but major features likely to stay unchanged.
- Alpha version of XQuery engine available free from Microsoft
- XQuery is derived from the Quilt query language, which itself borrows from SQL, XQL and XML-QL
- XQuery uses a

for ... let ... where .. result ...

syntax

for ⇔ SQL from

where ⇔ SQL where

result ⇔ SQL select

let allows temporary variables, and has no equivalent in SQL

FLWR Syntax in XQuery

- For clause uses XPath expressions, and variable in for clause ranges over values in the set returned by XPath
- Simple FLWR expression in XQuery
 - ★ find all accounts with balance > 400, with each result enclosed in an <account-number> .. </account-number> tag

```
for $x in /bank-2/account
let $acctno := $x/@account-number
where $x/balance > 400
return <account-number> $acctno </account-number>
```
- Let clause not really needed in this query, and selection can be done in XPath. Query can be written as:

```
for $x in /bank-2/account[balance>400]
return <account-number> $x/@account-number
</account-number>
```

Joins

- Joins are specified in a manner very similar to SQL

```
for $a in /bank/account,
   $c in /bank/customer,
   $d in /bank/depositor
where $a/account-number = $d/account-number
and $c/customer-name = $d/customer-name
return <cust-acct> $c $a </cust-acct>
```
- The same query can be expressed with the selections specified as XPath selections:

```
for $a in /bank/account
   $c in /bank/customer
   $d in /bank/depositor[
       account-number = $a/account-number and
       customer-name = $c/customer-name]
return <cust-acct> $c $a</cust-acct>
```

XML: Summary

- Becoming the standard for data exchange
- Many details still need to be worked out !!
- Active area of research...
 - ★ Especially optimization/implmentation

Topics

- OLAP
- Data Warehouses
- Information Retrieval

OLAP

- On-line Analytical Processing
- Why ?
 - ★ Exploratory analysis
 - Interactive
 - Different queries than typical SPJ SQL queries
 - ★ Data CUBE
 - A summary structure used for this purpose
 - E.g. *give me total sales by zipcode; now show me total sales by customer employment category*
 - Much much faster than using SQL queries against the raw data
 - The tables are *huge*
- Applications:
 - ★ Sales reporting, Marketing, Forecasting etc etc

Data Warehouses

- A repository of integrated information for querying and analysis purposes
- Tend to be very very large
 - ★ Typically not kept up-to-date with the *real data*
- Specialized query processing and indexing techniques are used
- Very widely used

Data Mining

- Searching for patterns in data
 - ★ Typically done in data warehouses
- Association Rules:
 - ★ When a customer buys X, she also typically buys Y
 - ★ Use ?
 - Move X and Y together in supermarkets
 - ★ A customer buys a lot of shirts
 - Send him a catalogue of shirts
 - ★ Patterns are not always obvious
 - Classic example: It was observed that men tend to buy *beer* and *diapers* together (may be an urban legend)
- Other types of mining
 - ★ Classification
 - ★ Decision Trees

Information Retrieval

- Relational DB == Structured data
- Information Retrieval == Unstructured data
- Evolved independently of each other
 - ★ Still very little interaction between the two
- Goal: Searching within documents
 - ★ Queries are different; typically a list of words, not SQL
- E.g. Web searching
 - ★ If you just look for documents containing the words, millions of them
 - Mostly useless
- Ranking:
 - ★ This is the key in IR
 - ★ Many different ways to do it
 - E.g. something that takes into account *term frequencies*
 - ★ Pagerank (from Google) seems to work best for Web.