O-O: Day 2

Array and Multiset Types in SQL

- Example of array and multiset declaration:

```sql
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
  
  \texttt{array} ['Silberschatz', 'Korth', 'Sudarshan']

- Multisets
  
  \texttt{multiset} ['computer', 'database', 'SQL']

- To create a tuple of the type defined by the books relation:
  
  ('Compilers', \texttt{array} ['Smith', 'Jones'],
  \texttt{new Publisher} ('McGraw-Hill', 'New York'),
  \texttt{multiset} ['parsing', 'analysis'])

- To insert the preceding tuple into the relation books
  
  \texttt{insert into books values}
  
  ('Compilers', \texttt{array} ['Smith', 'Jones'],
  \texttt{new Publisher} ('McGraw-Hill', 'New York'),
  \texttt{multiset} ['parsing', 'analysis']);

Querying Collection-Valued Attributes

- To find all books that have the word “database” as a keyword,
  
  \texttt{select title}
  \texttt{from books}
  \texttt{where 'database' in (unnest(keyword-set))}

- We can access individual elements of an array by using indices
  
  E.g.: If we know that a particular book has three authors, we could write:
  
  \texttt{select author_array[1], author_array[2], author_array[3]}
  \texttt{from books}
  \texttt{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
  
  \texttt{select B.title, A.author}
  \texttt{from books as B, unnest (B.author_array) as A (author)}

- To retain ordering information we add a \texttt{with ordinality} clause
  
  \texttt{select B.title, A.author, A.position}
  \texttt{from books as B, unnest (B.author_array) with ordinality as A (author, position)
References, and 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.
- Also more efficient.

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
- **Good when you know exactly what you want,**
  - not a good idea in general
  - Similar argument as *network databases vs relational databases*

---

**Comparison of O-O and O-R 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

**ORMs! Peewee!**

- **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.
Topics

- Object Oriented, Object Relational
- Client-server, Parallel, Distributed Systems
- OLAP/Data Warehouses
- Information Retrieval
- Cloud Computing
  - Data centers, Map-reduce, NoSQL Systems

Client-Server Systems

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

- Parallel database systems consist of multiple processors and multiple disks connected by a fast interconnection network.
- A **coarse-grain parallel** machine consists of a small number of powerful processors
- A **massively parallel** or **fine grain parallel** machine utilizes thousands of smaller processors.
- Two main performance measures:
  - **throughput** --- the number of tasks that can be completed in a given time interval
  - **response time** --- the amount of time it takes to complete a single task from the time it is submitted
**Speed-Up and Scale-Up**

- **Speedup**: a fixed-sized problem executing on a small system is given to a system which is $N$-times larger.
  - Measured by:
    \[
    \text{speedup} = \frac{\text{small system elapsed time}}{\text{large system elapsed time}}
    \]
  - Speedup is **linear** if equation equals $N$.
- **Scaleup**: increase the size of both the problem and the system
  - $N$-times larger system used to perform $N$-times larger job
  - Measured by:
    \[
    \text{scaleup} = \frac{\text{small system small problem elapsed time}}{\text{big system big problem elapsed time}}
    \]
  - Scale up is **linear** if equation equals 1.

---

**Speedup**

![Graph showing linear and sublinear speedup](image-url)
Factors Limiting Speedup and Scaleup

Speedup and scaleup are often sublinear due to:

- **Startup costs:**
  - Cost of starting up multiple processes may dominate computation time, if the degree of parallelism is high.

- **Interference:**
  - Processes accessing shared resources (e.g., system bus, disks, or locks) compete with each other, thus spending time waiting on other processes, rather than performing useful work.

- **Skew:**
  - Increasing the degree of parallelism increases the variance in service times of executing tasks in parallel.
  - Overall execution time determined by *slowest* of parallelly executing tasks.
Parallel Databases

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

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

MapReduce Framework

- Provides a fairly restricted, but still powerful abstraction for programming

- Programmers write a pipeline of functions, called map or reduce
  - map programs
    - inputs: a list of “records” (record defined arbitrarily – could be images, genomes etc…)
    - output: for each record, produce a set of “(key, value)” pairs
  - reduce programs
    - input: a list of “(key, {values})” grouped together from the mapper
    - output: whatever

- Both can do arbitrary computations on the input data as long as the basic structure is followed
MapReduce Framework

Word Count Example

map(String key, String value):
   // key: document name
   // value: document contents
   for each word w in value:
      EmitIntermediate(w, "1");

reduce(String key, Iterator values):
   // key: a word
   // values: a list of counts
   int result = 0;
   for each v in values:
      result += ParseInt(v);
   Emit(AsString(result));
MapReduce Framework: Word Count

input files: a b a c d b
mappers: (a, 1) (b, 1) (a, 1) (c, 1) (d, 1) (b, 1)
intermediate files: (a, 1) (a, 1) (a, 1) (a, 1) (a, 1) ...
reducers: (a, 8) (c, 5)
output files: ...

More Efficient Word Count

input files: a b a c d b
mappers: (a, 2) (b, 2) (c, 1) (d, 1)
intermediate files: (a, 2) (a, 3) (c, 1) (c, 5)
reducers: (a, 8) (c, 5)
output files: (b, 6) (d, 2)

Called “mapper-side” combiner