Miscellaneous Topics, and The Cloud

Rollups

```sql
SELECT 'ALL', 'ALL', 'ALL', SUM(Sales)
FROM Sales
WHERE Model = 'Chevy'
UNION
SELECT Model, 'ALL', 'ALL', SUM(Sales)
FROM Sales
WHERE Model = 'Chevy'
GROUP BY Model
UNION
SELECT Model, Year, 'ALL', SUM(Sales)
FROM Sales
WHERE Model = 'Chevy'
GROUP BY Model, Year
UNION
SELECT Model, Year, Color, SUM(Sales)
FROM Sales
WHERE Model = 'Chevy'
GROUP BY Model, Year, Color;
```

This is a simple 3-dimensional roll-up. Aggregating over N dimensions requires N such unions.
Crosstabs

Relationship between two or more variables:

<table>
<thead>
<tr>
<th>Age</th>
<th>Unlisted phone number</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>No</td>
</tr>
<tr>
<td>% within column n</td>
<td>24%</td>
</tr>
<tr>
<td>18-34</td>
<td>185</td>
</tr>
<tr>
<td>% within column n</td>
<td>20%</td>
</tr>
<tr>
<td>35-44</td>
<td>153</td>
</tr>
<tr>
<td>% within column n</td>
<td>17%</td>
</tr>
<tr>
<td>45-54</td>
<td>133</td>
</tr>
<tr>
<td>% within column n</td>
<td>17%</td>
</tr>
<tr>
<td>55-64</td>
<td>130</td>
</tr>
<tr>
<td>% within column n</td>
<td>23%</td>
</tr>
<tr>
<td>65+</td>
<td>178</td>
</tr>
<tr>
<td>% within column n</td>
<td>100%</td>
</tr>
<tr>
<td>NET</td>
<td>779</td>
</tr>
</tbody>
</table>

Problems with Groupbys

- roll-up can be asymmetric (e.g. not aggregate by year, or color)
- cross-tabs(spreadsheets)

![Table 5: Chevy Sales Cross Tab](image)

- even if SQL syntax can be devised, a 6D cross-tab requires 64 groupby queries to generate it and 64 scans and sorts of the data
- most of these are not relational expressions but are in many report writers
**Data Warehouses**

- A repository of integrated information for querying and analysis purposes
- A (usually) stand-alone system that integrates data from everywhere
  - Read-only, typically not kept up-to-date with the real data
  - Geared toward business analytics, data mining etc...
  - HUGE market today
- Heavily optimized
  - Specialized query processing and indexing techniques are used
  - High emphasis on pre-computed data structures like summary tables, data cubes
- Analysis cycle:
  - Extract data from databases with queries, visualize/analyze with desktop tools
  - E.g., Tableau

---

**Data Warehouses**

![Data Warehouse Diagram](image)
Data Warehouses

Query processing algorithms heavily optimized for these types of schemas

Many queries of the type:
- Selections on dimension tables (e.g., state = 'MD')
- Join fact table with dimension tables
- Aggregate on a "measure" attribute (e.g., Quantity, TotalPrice)

For example:
```
select c_city, o_year, SUM(quantity)
from Fact, Customer, Product
where p_category = 'Tablet';
```

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
Data Warehouses

- Data analytics a major industry right now, and likely to grow in near future
  - BIG Data !!
  - Extracting (actionable) knowledge from data really critical
    - Especially in real-time

- Some key technologies:
  - Parallelism – pretty much required
  - Column-oriented design
    - Lay out the data column-by-column, rather than row-by-row
  - Heavy pre-computation (like Cubes)
  - New types of indexes
    - Focusing on bitmap representations
  - Heavy compression
  - Map-reduce??

Topics

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

Relevance Ranking Using Terms

- **TF-IDF** (Term frequency/Inverse Document frequency) ranking:
  - Let $n(d) =$ number of terms in the document $d$
  - $n(d, t) =$ number of occurrences of term $t$ in the document $d$.
  - Relevance of a document $d$ to a term $t$:
    \[ TF (d, t) = \log \left( 1 + \frac{n(d, t)}{n(d)} \right) \]
    - The log factor is to avoid excessive weight to frequent terms
  - Relevance of document to query $Q$:
    \[ r (d, Q) = \sum_{t \in Q} \frac{TF (d, t)}{n(t)} \]
PageRank  Google search back in the day...

- The probability that a random surfer (who follows links randomly) will end up at a particular page
  - **Intuitively:** Higher the probability, the more important the page
- Surfer model:
  - Choose a random page to visit with probability “alpha”
  - If the number of outgoing edges = n, then visit one of those pages with probability \((1 – \alpha)/n\)

Cloud Computing: Outline

- Technologies behind cloud computing
  - Data centers
  - Virtualization
  - Programming Framework: Map-reduce
  - Distributed Key-Value Stores
- Some observations about the marketplace
Key-Value Stores

- Some Interesting (somewhat old) numbers (http://highscalability.com)
  - Twitter: 177M tweets sent on 3/1/2011 (nothing special about the date), 572,000 accounts added on 3/12/2011
  - Dropbox: 1M files saved every 15 mins
  - Stackoverflow: 3M page views a day (Redis for caching)
  - Wordnik: 10 million API Requests a Day on MongoDB and Scala
  - Mollom: Killing Over 373 Million Spams at 100 Requests Per Second (Cassandra)
  - Facebook's New Real-time Messaging System: HBase to Store 135+ Billion Messages a Month
  - Reddit: 270 Million Page Views a Month in May 2010 (Memcache)

- How to support such scale?
  - Databases typically not fast enough
  - Facebook aims for 3-5ms response times

Issues

- Data Consistency, High Availability, and Low Latency hard to guarantee simultaneously
  - Impossible in some cases especially if networks can fail
  - CAP Theorem: Consistency, Availability, Partition tolerance: pick any 2

- Distributed transactions
  - If a transaction spans multiple machines, what to do?
  - Correct solution: Two-phase Commit
    - Two-round protocol
    - Latencies too high

- Dealing with replication
  - Replication of data is a must
  - How to keep them updated?
    - Eager vs lazy replication
    - Significant impact on consistency and availability

- Many systems in this space sacrifice consistency
Systems

• Numerous systems designed in last 10 years that look very similar
  • Differences often subtle, and if not hard to pin down, hard to understand
  • Often the differences are about the implementations

• Often called key-value stores
  • The main provided functionality is that of a hashtable

• Some earlier solutions
  • Still very popular
    • Memcached + MySQL + Sharding
      • Sharding == partitioning horizontally
        • Each partition has same schema and columns, different rows
      • Store data in MySQL -- use Memcached to cache the data
    • Memcached not really a database, just a cache
    • All kinds of consistency issues
    • But... very very fast

Systems

• Tokyo, Redis
  • Very efficient key value stores

• BigTable (Google), HBase (Apache open source), Cassandra (from Facebook, open sourced), Voldemort (originally LinkedIn)... 
  • At least in original iterations, focused on performance
  • Cassandra later developed more sophisticated \{em tunable\} consistency (maybe others too)

• PNUTS (Yahoo!)
  • Focus on geographical distribution
    • Easier to deal with some issues if we assume everything is a single data center
    • Support tunable consistency for reads: read-any, read-latest etc..
  • Form of master-slave replication
  • No real support for multi-record transactions
**Systems**

- **Megastore (Google)**
  - Built on top of BigTable -- powers Google App Engine
    - Full ACID using Paxos, replication, two-phase commit
  - Supports notion of “entity groups”
    - e.g., all emails of a user is a single entity group
  - Transactions that span a single entity group are generally fine
  - Transactions that span multiple entity groups would use two-phase commit -- not preferred

- **MongoDB**
  - Perhaps the poster child of key-value NoSQL stores
  - Very scalable
    - Document-oriented storage with JSON-style documents
    - JSON becoming more popular than XML as the interchange format
  - Very loose consistency guarantees

---

**In Summary…**

- **Three key pieces of cloud computing**
  - Data centers
    - Increasingly growing in numbers
    - Many challenges in building them, maintaining them etc..
  - Virtualization
  - Programming frameworks
    - Simplest (to explain): just use the virtual machines directly
      - But much harder to manage
    - Using Hadoop or HBase (as appropriate) simplifies the programming quite a bit
      - But Hadoop is open source, and managing hadoop installations not much easier
  - Still many technical challenges to be solved
Introduction

- Core operation in peer-to-peer systems is to efficiently locate the node that stores a particular data item.
- Chord is a scalable distributed protocol for lookup in a dynamic peer-to-peer system with frequent node arrivals and departures.
- Only one operation: given a key, it maps the key onto a node.
- Simplicity, provable correctness, and provable performance.
The lookup problem

Centralized lookup (Napster)

- Simple, but $O(N)$ states and single point of failure
Flooded queries (Gnutella)

- Robust, but worst case $O(N)$ messages per lookup

Routed queries (Freenet, Chord, etc.)
Related Work

- Freenet (Clarke, Sandberg, Wiley, Hong)
- CAN (Ratnasamy, Francis, Handley, Karp, Shenker)
- Pastry (Rowstron, Druschel)
- Tapestry (Zhao, Kubiatowicz, Joseph)
- ... ...
- Chord emphasizes simplicity

Design Objectives

- Load Balance:
  - Distributed hash function spreads keys evenly over the nodes (Consistent hashing).
- Decentralization:
  - Fully distributed (Robustness).
- Scalability:
  - Lookup grows as a log of number of nodes.
- Availability:
  - Automatically adjusts internal tables to reflect changes.
- Flexible Naming:
  - No constraints on key structure.
Applications

- Example applications:
  - Cooperative Mirroring
  - Time-shared storage
  - Distributed indexes
  - Large-Scale combinatorial search

Routing challenges

- Define a useful key nearness metric.
- Keep the hop count small.
- Keep the routing tables small.
- Stay robust despite rapid changes.
Chord properties

- **Efficient:**
  - $O(\log(N))$ messages per lookup.
- **Scalable:**
  - $O(\log(N))$ states per node.
- **Robust:**
  - survives massive failures, join or leave. $O(\log^2(N))$ messages.
- An Nth node joins (or leaves),
  - only $O(1/N)$ keys are moved to a different location.
- Proofs are in paper / tech report.
  - (Assuming no malicious participants)

Chord overview

- Provides peer-to-peer hash lookup:
  - Lookup(key) → IP address.
  - Chord does not store the data.
- How does Chord route lookups?
- How does Chord maintain routing tables?
- How does Chord cope with changes in membership?
Chord IDs

- m-bit identifier space for both keys and nodes.
  - Key identifier = SHA-1(key).
  - Node identifier = SHA-1(IP address).

- Both are uniformly distributed.

- How to map key IDs to node IDs?

Consistent hashing [Karger 97]

- A key is stored at its successor: node with next higher ID
Basic lookup

“Where is key 80?”

“Finger table” allows \( \log(N) \)-time lookups

Every node knows \( m \) other nodes in the ring
Finger $i$ points to successor of $n+2^i-1$

Each node knows more about portion of circle close to it

Lookups take $O(\log(N))$ hops
Reconciliation Algorithm

- Leaves, joins mess up structure
  - runs periodically
  - aggregates multiple changes
  - redistributes ownerships of keys
  - fixes finger tables
  - fixes successor pointers
  - fixes replication

*Doesn’t always work under heavy churn*

Chord Summary

- Chord provides peer-to-peer hash lookup:
  - Efficient: $O(\log(n))$ messages per lookup.
  - Robust as nodes fail and join.
  - Good primitive for peer-to-peer systems.

- http://www.pdos.lcs.mit.edu/chord