CS 91: Cloud Systems & Datacenter Networks

Data Retrieval
Tomorrow

- Faculty summer research

- SCI 128 @ 4:10 PM
Cloud Storage

• Most cloud systems store data for users
  – photos, business analytics, employee records, ...

• What good is storing data if we can’t find it?

• What good is finding it if it takes a long time?

• How well does the storage system scale?
Latency Matters

• “Latency is everywhere and it costs you sales”
  – Blog essay by Todd Hoff
  – Every 100 ms of latency costs Amazon 1% in sales.
  – An extra ½ second in search page generation time dropped Google traffic by 20%.

• Lots of great examples, explanations of latency sources, and what you might do
What kinds of data?

• So far: data “files”
  – NFS, Harp, GFS (mostly)

• Real data is more specific:
  – user profiles
  – pictures
  – videos
  – comments
  – customer orders
  – arbitrary “records” (data tuples, e.g. MR K/V pairs)
Structure?

• Structured data
  – Well-defined pieces of information always present
  – Typically fixed size (or known maximum size)
  – Example: a customer order (who, what, when, ...)

• Unstructured data
  – Basically just an opaque blob of bytes
  – Size often depends on the contents
  – Example: a video
How do we store them?

• File abstraction makes sense for some things:
  – pictures
  – videos
  – documents

• Maybe not for others:
  – comments
  – profiles
  – orders

Regardless, it often it doesn’t matter.

User tells the system what they want (via some identifier), system goes and gets it for them.
Finding Data: Databases

• Traditionally, RDBMS
  – Relational database management systems

• Great for structured data
  – e.g., every record has exactly the same fields

• “ACID” coined in this context
  – lots of work done to keep data consistent
  – implication: limits to parallelism
Relations & Structured Data

• Organized into tables with fixed columns

• “records” correspond to a row in the table

• Table defines a relationship between the columns in each row
## Example

**Table: Customers**

<table>
<thead>
<tr>
<th>Customer_ID</th>
<th>Name</th>
<th>Phone Number</th>
<th>Email address</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>0001</td>
<td>Alice</td>
<td>555-1234</td>
<td><a href="mailto:alice@a.net">alice@a.net</a></td>
<td></td>
</tr>
<tr>
<td>0002</td>
<td>Bob</td>
<td>555-5678</td>
<td><a href="mailto:bobby@b.com">bobby@b.com</a></td>
<td></td>
</tr>
<tr>
<td>0003</td>
<td>Charlie</td>
<td>555-9876</td>
<td><a href="mailto:c@charlie.org">c@charlie.org</a></td>
<td></td>
</tr>
<tr>
<td>0004</td>
<td>Bob</td>
<td>555-5432</td>
<td><a href="mailto:bob2@b.com">bob2@b.com</a></td>
<td></td>
</tr>
</tbody>
</table>

When you tell DB to create a table, you can specify invariants that must hold true (error otherwise).

For example:

- Customer ID and Email address must be unique
- Customer ID is the primary key
Queries

• Special language for finding / adding data:
  – SQL: Structured Query Language
  – Declarative rather than imperative
Example

Table: Customers

<table>
<thead>
<tr>
<th>Customer_ID</th>
<th>Name</th>
<th>Phone Number</th>
<th>Email address</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>0001</td>
<td>Alice</td>
<td>555-1234</td>
<td><a href="mailto:alice@a.net">alice@a.net</a></td>
<td></td>
</tr>
<tr>
<td>0002</td>
<td>Bob</td>
<td>555-5678</td>
<td><a href="mailto:bobby@b.com">bobby@b.com</a></td>
<td></td>
</tr>
<tr>
<td>0003</td>
<td>Charlie</td>
<td>555-9876</td>
<td><a href="mailto:c@charlie.org">c@charlie.org</a></td>
<td></td>
</tr>
<tr>
<td>0004</td>
<td>Bob</td>
<td>555-5432</td>
<td><a href="mailto:bob2@b.com">bob2@b.com</a></td>
<td></td>
</tr>
</tbody>
</table>

“Give me all the rows with customer name Bob”:

SELECT * FROM Customers WHERE Name = “Bob”

<table>
<thead>
<tr>
<th>Customer_ID</th>
<th>Name</th>
<th>Phone Number</th>
<th>Email address</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>0002</td>
<td>Bob</td>
<td>555-5678</td>
<td><a href="mailto:bobby@b.com">bobby@b.com</a></td>
<td></td>
</tr>
<tr>
<td>0004</td>
<td>Bob</td>
<td>555-5432</td>
<td><a href="mailto:bob2@b.com">bob2@b.com</a></td>
<td></td>
</tr>
</tbody>
</table>
“Join” Operation

• Tables can refer to one another, usually by key
  – Kind of like a pointer

• If a table stores a column whose value is a primary key in another, it’s a “foreign key”
**Example**

**Table: Customers**

<table>
<thead>
<tr>
<th>Customer_ID</th>
<th>Name</th>
<th>Phone Number</th>
<th>Email address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0001</td>
<td>Alice</td>
<td>555-1234</td>
<td><a href="mailto:alice@a.net">alice@a.net</a></td>
</tr>
<tr>
<td>0002</td>
<td>Bob</td>
<td>555-5678</td>
<td><a href="mailto:bobby@b.com">bobby@b.com</a></td>
</tr>
<tr>
<td>0003</td>
<td>Charlie</td>
<td>555-9876</td>
<td><a href="mailto:c@charlie.org">c@charlie.org</a></td>
</tr>
<tr>
<td>0004</td>
<td>Bob</td>
<td>555-5432</td>
<td><a href="mailto:bob2@b.com">bob2@b.com</a></td>
</tr>
</tbody>
</table>

...  

**Table: Orders**

<table>
<thead>
<tr>
<th>Order_ID</th>
<th>Date</th>
<th>Price</th>
<th>Customer_ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>0001</td>
<td>July 10</td>
<td>$8700</td>
<td>0003</td>
</tr>
<tr>
<td>0002</td>
<td>December 19</td>
<td>$3000</td>
<td>0001</td>
</tr>
</tbody>
</table>
**Example**

Table: Customers

<table>
<thead>
<tr>
<th>Customer_ID</th>
<th>Name</th>
<th>Phone Number</th>
<th>Email address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0001</td>
<td>Alice</td>
<td>555-1234</td>
<td><a href="mailto:alice@a.net">alice@a.net</a></td>
</tr>
<tr>
<td>0002</td>
<td>Bob</td>
<td>555-5678</td>
<td><a href="mailto:bobby@b.com">bobby@b.com</a></td>
</tr>
<tr>
<td>0003</td>
<td>Charlie</td>
<td>555-9876</td>
<td><a href="mailto:c@charlie.org">c@charlie.org</a></td>
</tr>
<tr>
<td>0004</td>
<td>Bob</td>
<td>555-5432</td>
<td><a href="mailto:bob2@b.com">bob2@b.com</a></td>
</tr>
</tbody>
</table>

...  

Foreign key “pointer” to Customers table.

Table: Orders

<table>
<thead>
<tr>
<th>Order_ID</th>
<th>Date</th>
<th>Price</th>
<th>Customer_ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>0001</td>
<td>July 10</td>
<td>$8700</td>
<td>0003</td>
</tr>
<tr>
<td>0002</td>
<td>December 19</td>
<td>$3000</td>
<td>0001</td>
</tr>
</tbody>
</table>
“Join” Operation

SELECT Customers.Name, Orders.Date, Orders.Price
FROM Customers, Orders
WHERE Customers.Customer_ID = Orders.Customer_ID

<table>
<thead>
<tr>
<th>Name</th>
<th>Date</th>
<th>Price</th>
</tr>
</thead>
<tbody>
<tr>
<td>Charlie</td>
<td>July 10</td>
<td>$8700</td>
</tr>
<tr>
<td>Bob</td>
<td>December 19</td>
<td>$3000</td>
</tr>
</tbody>
</table>
Performance

• Tons of work out there on making DBs faster:
  – Table design practices
  – Query Optimizers
  – Indexing

• Maintaining ACID semantics across data with strict relations is expensive.

• Distribution can help, but only scales so much.
  – Diminishing returns for adding nodes (overhead)
Finding Data: Alternatives?

• What about unstructured data?

• Suppose I told you I want you to lookup data for me based on an ID
  – e.g., YouTube ID: tDacjrSCEq4 -> shouting video

• If you were implementing a lab assignment, what would you use?
  Hash Table!
Distributed Hash Tables (DHT)

- Relatively recent invention (~2001)

- Very simple interface:
  - get(key)
  - put(key, value)

- Sometimes called “distributed key-value store” or “NoSQL store”
DHT Applicability

• Used *frequently* in industry:
  – Google (YouTube)
  – Reddit
  – Facebook
  – Twitter
  – Tumblr
  – Wikipedia

• (Also popular in peer-to-peer file sharing)
DHT Advantages

• Fully peer-to-peer, no central authority

• The nodes organize themselves

• Scales out very well, fast lookups

• Gracefully degrades performance during failures, data replicated for safety
DHT Disadvantages

• Self-organizing systems are more complex than centralized ones

• No relationship between values of different keys enforced by system

• Consistency often relaxed for performance
Reality: Data Retrieval Spectrum

• Many hybrid systems out there that occupy different points in design space (NoSQL)
  – Cassandra
  – MongoDB
  – Memcached
  – Dynamo
  – Project Voldemort
  – (Literally dozens of others. Trade-offs galore...)
Upcoming Papers

• Chord: A Scalable Peer-to-Peer Lookup Protocol for Internet Applications (2001)

• Dynamo: Amazon's Highly Available Key-value Store (2007)

• Scaling Memcache at Facebook (2013)

• Spanner: Google's Globally-Distributed Database (2012)
Monday: Special Guest

• Cathy Polinsky ‘99
  – VP of Engineering at SalesForce.com

• SalesForce.com:
  – flagship product: customer relationship management
  – Many other cloud-based services