Two Approaches to Data Sharing

1. Use a *centralized database* to store all of the data.
   - applications use SQL to access the data

2. Use separate *application databases*.
   - each database is only directly accessed by one application
   - exchange data between applications through *web services*
     - use an API based on HTTP
     - transfer the data in text form using XML or similar model
Recall: The Conventional Approach

- Use a DBMS that employs the relational model and SQL
- Typically follow a client-server model
  - the database server manages the data
  - applications act as clients
- Support *transactions* and the associated guarantees

The Rise of NoSQL

- The centralized approach to data sharing / app integration helped RDBMSs retain their dominance until the early 2000s.
- With the shift to application databases and web services, the importance of the relational model decreased.
  - SQL was no longer needed to access shared data
  - applications could change their own database schema as long as they continued to support the same API
- Thus, developers were now free to try alternative approaches.
The Rise of NoSQL (cont.)

- In addition, it became necessary for web-based applications to deal with massive amounts of:
  - data
  - traffic / queries
- Scalability is crucial.
  - load can increase rapidly and unpredictably
- Large servers are expensive and can only grow so large.
- Solution: use clusters of small commodity machines
  - use both fragmentation/sharding and replication
  - cheaper
  - greater overall reliability
  - can take advantage of cloud-based storage

The Rise of NoSQL (cont.)

- Problem: RDBMSs do not scale well to large clusters.

- Google and Amazon each developed their own alternative approaches to data management on clusters.
  - Google: BigTable
  - Amazon: DynamoDB

- The papers that Google and Amazon published about their efforts got others interested in developing similar DBMSs.
  ➔ noSQL
What Does NoSQL Mean?

• Not well defined.

• Typical characteristics of NoSQL DBMSs:
  • don't use SQL / the relational model
  • open-source
  • designed for use on clusters
    • support for sharding/fragmentation and replication
    • schema-less or flexible schema

• One good overview:
  Sadalage and Fowler, *NoSQL Distilled*
  (Addison-Wesley, 2013).

Flavors of NoSQL

• Various taxonomies have been proposed

• Three of the main classes of NoSQL databases are:
  • key-value stores
  • document databases
  • column-family (aka big-table) stores

• Some people also include graph databases.
  • very different than the others
  • example: they are *not* designed for clusters
Key-Value Stores

• We've already worked with one of these: Berkeley DB

• There are many others: Riak, Redis, MemcacheDB, Amazon's DynamoDB, Voldemort

• Simple data model: key/value pairs
  • the DBMS does not attempt to interpret the value

• Queries are limited to query by key.
  • get/put/update/delete a key/value pair
  • iterate over key/value pairs

Document Databases

• Examples include: MongoDB, CouchDB, Terrastore

• Also store key/value pairs

• Unlike key-value stores, the value is not opaque.
  • it is a document containing semistructured data
  • it can be examined and used by the DBMS

• Queries:
  • can be based on the key (as in key/value stores)
  • more often, are based on the contents of the document

• Here again, there is support for sharding and replication.
  • the sharding can be based on values within the document
Column-Family Databases

• Google's BigTable and systems based on it
  • HBase, Cassandra, Hbase, Amazon SimpleDB, etc.

• To understand the motivation behind their design, consider one type of problem BigTable was designed to solve:
  • You want to store info about web pages!
  • For each URL, you want to store:
    • its contents
    • its language
    • for each other page that links to it, the anchor text associated with the link (i.e., the text that you click on)

Storing Web-Page Data in a Traditional Table

<table>
<thead>
<tr>
<th>page URL</th>
<th>language</th>
<th>contents</th>
<th>anchor text from <a href="http://www.cnn.com">www.cnn.com</a></th>
<th>anchor from <a href="http://www.bu.edu">www.bu.edu</a></th>
<th>one col per page</th>
</tr>
</thead>
<tbody>
<tr>
<td><a href="http://www.cnn.com">www.cnn.com</a></td>
<td>English</td>
<td>&lt;html&gt;…</td>
<td></td>
<td></td>
<td>...</td>
</tr>
<tr>
<td><a href="http://www.bu.edu">www.bu.edu</a></td>
<td>English</td>
<td>&lt;html&gt;…</td>
<td></td>
<td></td>
<td>...</td>
</tr>
<tr>
<td><a href="http://www.nytimes.com">www.nytimes.com</a></td>
<td>English</td>
<td>&lt;html&gt;…</td>
<td>&quot;news story&quot;</td>
<td></td>
<td>...</td>
</tr>
<tr>
<td><a href="http://www.lemonde.fr">www.lemonde.fr</a></td>
<td>French</td>
<td>&lt;html&gt;…</td>
<td>&quot;French elections&quot;</td>
<td></td>
<td>...</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

• One row per web page
• Single columns for its language and contents
• One column for the anchor text from each possible page, since in theory any page could link to any other page!
• Leads to a huge sparse table – most cells are empty/unused.
Storing Web-Page Data in BigTable

- Rather than defining all possible columns, define a set of column families that each row should have.
  - example: a column family called anchor that replaces all of the separate anchor columns on the last slide
  - can also have column families that are like typical columns

- In a given row, only store columns with an actual value, representing them as (column key, value) pairs
  - column key = column family: qualifier
  - ex: ("anchor:www.bu.edu", "news story")

What is the key-value pair for the highlighted entry?

<table>
<thead>
<tr>
<th>page URL</th>
<th>language</th>
<th>contents</th>
<th>anchor text from <a href="http://www.cnn.com">www.cnn.com</a></th>
<th>anchor from <a href="http://www.bu.edu">www.bu.edu</a></th>
<th>one col per page</th>
</tr>
</thead>
<tbody>
<tr>
<td><a href="http://www.cnn.com">www.cnn.com</a></td>
<td>English</td>
<td>&lt;html&gt;...</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><a href="http://www.bu.edu">www.bu.edu</a></td>
<td>English</td>
<td>&lt;html&gt;...</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><a href="http://www.nytimes.com">www.nytimes.com</a></td>
<td>English</td>
<td>&lt;html&gt;...</td>
<td>&quot;news story&quot;</td>
<td></td>
<td></td>
</tr>
<tr>
<td><a href="http://www.lemonde.fr">www.lemonde.fr</a></td>
<td>French</td>
<td>&lt;html&gt;...</td>
<td>&quot;French elections&quot;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

B. (anchor:www.cnn.com, "French elections")
C. (anchor:www.lemonde.fr, "French elections")
What is the key-value pair for the highlighted entry?

<table>
<thead>
<tr>
<th>page URL</th>
<th>language</th>
<th>contents</th>
<th>anchor text from <a href="http://www.cnn.com">www.cnn.com</a></th>
<th>anchor from <a href="http://www.bu.edu">www.bu.edu</a></th>
<th>one col per page</th>
</tr>
</thead>
<tbody>
<tr>
<td><a href="http://www.bu.edu">www.bu.edu</a></td>
<td>English</td>
<td>&lt;html&gt;…</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><a href="http://www.nytimes.com">www.nytimes.com</a></td>
<td>English</td>
<td>&lt;html&gt;…</td>
<td>&quot;news story&quot;</td>
<td></td>
<td></td>
</tr>
<tr>
<td><a href="http://www.lemonde.fr">www.lemonde.fr</a></td>
<td>French</td>
<td>&lt;html&gt;…</td>
<td>&quot;French elections&quot;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

B. (anchor:www.cnn.com, "French elections")
C. (anchor:www.lemonde.fr, "French elections")

---

Data Model for Column-Family Databases

- In addition to column keys:
  - row keys are used to index the rows
  - can also associate a timestamp with a given column value

- You thus have a multi-dimensional map:
  - (row key, column key, timestamp) → value
  - example:
    - ("www.nytimes.com", "anchor:www.bu.edu", t1) → "news story"

- Different rows can have different schema.
  - i.e., different sets of column keys
  - (column key, value) pairs can be added or removed from a given row over time

- The set of column families in a given table rarely change.
Advantages of Column Families

- Gives an additional unit of data, beyond just a single row.
- Column families are used for access controls.
  - can restrict an application to only certain column families
- Column families can be divided up into *locality groups* that are stored together.
  - based on which column families are typically accessed together
  - advantage? enables more efficient reads

Picturing a Row In a Column-Family Database

(source: Sadalage and Fowler)
Aggregate Orientation

- Key-value, document, and column-family stores all lend themselves to an *aggregate-oriented* approach.
- Group together data that "belongs" together
  - i.e., that will tend to be accessed together

<table>
<thead>
<tr>
<th>type of database</th>
<th>unit of aggregation</th>
</tr>
</thead>
<tbody>
<tr>
<td>key-value store</td>
<td>the value part of the key/value pair</td>
</tr>
<tr>
<td>document database</td>
<td>a document</td>
</tr>
<tr>
<td>column-family store</td>
<td>a row</td>
</tr>
<tr>
<td></td>
<td>(plus column-family sub-aggregates)</td>
</tr>
</tbody>
</table>

- Relational databases can't fully support aggregation.
  - No multi-valued attributes; focus on avoiding duplicated data
  - Give each type of entity its own table, rather than grouping together entities/attributes that are accessed together

Aggregate Orientation (cont.)

- Example: data about customers
  - RDBMS: store a customer's address in only one table
    - Use foreign keys in other tables that refer to the address
  - Aggregate-oriented system: store the full customer address in several places:
    - Customer aggregates
    - Order aggregates
    - Etc.

- Benefits of an aggregate-based approach in a NoSQL store:
  - Provides a unit for sharding across the cluster
  - Allows us to get related data without needing to access many different nodes
Schemalessness

- NoSQL systems are completely or mostly schemaless.
- Key-value stores: put whatever you like in the value
- Document databases: no restrictions on the schema used by the semistructured data inside each document.
  - although some do allow a schema, as with XML
- Column-family databases:
  - we do specify the column families in a given table
  - but no restrictions on the columns in a given column family and different rows can have different columns

Schemalessness (cont.)

- Advantages:
  - allows the types of data that are stored to evolve over time
  - makes it easier to handle nonuniform data
    - e.g., sparse tables
- Despite the fact that a schema is not required, programs that use the data need at least an *implicit* schema.
- Disadvantages of an implicit schema:
  - the DBMS can't enforce it
  - the DBMS can't use it to try to make accesses more efficient
  - different programs that access the same database can have conflicting notions of the schema
Example Document Database: MongoDB

- Mongo (from humongous)

- Key features include:
  - replication for high availability
  - auto-sharding for scalability
  - documents are expressed using JSON/BSON
  - queries can be based on the contents of the documents

- Related documents are grouped together into collections.
  - what does this remind you of?
    - Berkeley DB XML

JSON

- JSON is an alternative data model for semistructured data.
  - JavaScript Object Notation

- Built on two key structures:
  - an object, which is a sequence of fields (name:value pairs)
    
    ```javascript
    { id: "1000",
      name: "Sanders Theatre",
      capacity: 1000 }
    ```
  
  - an array of values
    
    ```javascript
    [ "123-456-7890", "222-222-2222", "333-333-3333" ]
    ```

- A value can be:
  - an atomic value: string, number, true, false, null
  - an object
  - an array
Example: JSON Object for a Person

```json
{   firstName: "John",
lastName: "Smith",
age: 25,
address: {
    streetAddress: "21 2nd Street",
city: "New York",
state: "NY",
postalCode: "10021"
},
phoneNumbers: [
    {   type: "home",
number: "212-555-1234"
},
    {   type: "mobile",
number: "646-555-4567"
}
]
}
```

BSON

- MongoDB actually uses BSON.
  - a binary representation of JSON
  - BSON = marshalled JSON!

- BSON includes some additional types that are not part of JSON.
  - in particular, a type called ObjectID for unique id values.

- Each MongoDB document is a BSON object.
The `_id` Field

- Every MongoDB document must have an `_id` field.
  - its value must be unique within the collection
  - acts as the primary key of the collection
  - it is the key in the key/value pair

- If you create a document without an `_id` field:
  - MongoDB adds the field for you
  - assigns it a unique BSON ObjectID

MongoDB Terminology

<table>
<thead>
<tr>
<th>relational term</th>
<th>MongoDB equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>database</td>
<td>database</td>
</tr>
<tr>
<td>table</td>
<td>collection</td>
</tr>
<tr>
<td>row</td>
<td>document</td>
</tr>
<tr>
<td>attributes</td>
<td>fields (name:value pairs)</td>
</tr>
<tr>
<td>primary key</td>
<td>the <code>_id</code> field, which is the key associated with the document</td>
</tr>
</tbody>
</table>

- Documents in a given collection typically have a similar purpose.
- However, no schema is enforced.
  - different documents in the same collection can have different fields