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
Recall: Limitations of the Conventional Approach

- Can be overkill for applications that don't need all the features
- Can be hard / expensive to setup / maintain / tune
- May not provide the necessary functionality
- Footprint may be too large
  - example: can't put a conventional RDBMS on a small embedded system
- May be unnecessarily slow for some tasks
  - overhead of IPC, query processing, etc.
- May not scale well to large clusters

Recall: What Other Options Are There?

- View a DBMS as being composed of two layers.
- At the bottom is the storage layer or storage engine.
  - stores and manages the data
- Above that is the logical layer.
  - provides an abstract representation of the data
  - based on some data model
  - includes some query language, tool, or API for accessing and modifying the data
- To get other approaches, choose different options for the layers.
Recall: Options for the Logical Layer (partial list)

- relational model + SQL
- object-oriented model + associated query language
- XML + XPath or XQuery
- JSON + associated API
- key-value pairs + associated API
- graph-based model + associated API/query language
- comma-delimited or tab-delimited text + tool for text search

Recall: Options for the Storage Layer (partial list)

- transactional storage engine
  - supports transactions, recovery, etc.
- a non-transactional engine that stores data on disk
- an engine that stores data in memory
- a column store that stores columns separately from each other
  - vs. a traditional row-oriented approach
  - beneficial for things like analytical-processing workloads
Relational vs. Object-Oriented

- As we've seen, OO databases provided the first serious threat to the conventional approach.

- They attempted to address an *impedance mismatch* between:
  - the way that applications represent data in memory
  - the way that data can be stored using the relational model

- What are some of the ways that there is a mismatch?

---

Relational vs. Object-Oriented (cont.)

- Relational DBMSs survived the OO "threat".
  - incorporated some key OO features
    - object-relational model
  - companies were reluctant to change legacy systems

- RDMSs also benefited from the need for applications to share data.
  - example: the various "Link" applications at BU
Two Approaches to Data Sharing

1. Use a *centralized database* to store all of the data.
   • applications use SQL to access the data
   • advantages of this approach?
   • disadvantages?

Two Approaches to Data Sharing (cont.)

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
   • advantages?
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
  - other features of conventional DBMSs became less important
    - example: security features
    - can be handled by the application, rather than the DBMS
  - applications could change their own database schema as long as they continued to support the same API
    - the benefits of encapsulation!

- Thus, developers were now free to try alternative approaches.

The Rise of NoSQL (cont.)

- Since the 2000s, web-based applications have needed 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 (aka sharding) and replication
  - cheaper
  - greater overall reliability
  - can take advantage of cloud-based storage
The Rise of NoSQL (cont.)

- Problem: RDBMs 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:
Flavors of NoSQL

- Various taxonomies have been proposed
- Sadalage and Fowler group them into four main types:
  - key-value stores
  - document databases
  - column-family (aka big-table) stores
  - graph databases
- Graph databases are very different from the other three, so we will consider them separately.

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
Is Berkeley DB a NoSQL DBMS?

• It depends on what you mean by noSQL!
• It does provide support for replication.
• It does not provide support for automated sharding/fragmentation.
• Many other NoSQL systems have been built on top of it.
  • Amazon's DynamoDB (BDB is one of several options)
  • MemcacheDB
  • Voldemort
  • Oracle NoSQL
• These systems:
  • take care of the sharding themselves
  • use BDB as the local storage engine at each node.

Sharding in a Key-Value Store

• Typical approach:
  • hash the key ➔ the shard/partition for that key/value pair
    • this is an example of what type of fragmentation?
      horizontal (like distributing the rows of a table)
  • each shard is replicated
Document Databases

- Examples include: MongoDB, CouchDB, Terrastore
- Also store key/value pairs
- However, the value is a *document*.
  - expressed using some sort of semistructured data model
    - XML
    - more often: JSON or BSON (JSON's binary counterpart)
  - the value *can* be examined and used by the DBMS
    (unlike in key/data stores)
- Queries can be based on the key (as in key/value stores),
  but more often they 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

---

JSON

- JSON is an alternative data model for semistructured data.
  - JavaScript Object Notation
- Built on two key structures:
  - an *object*, which is a collection of name/value pairs
    
```json
    {  "id"  :  "1000",
    "name"  :  "Sanders Theatre",
    "capacity"  :  1000  }
```
  - an *array* of values
    
```json
    [  "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"
        }
    ]
}
```

What features of semistructured data does JSON exhibit?

Column-Family Databases

- Google’s BigTable and systems based on it.
  - HBase, Cassandra, Hypertable, 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>
</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 Big Table

- Rather than defining all possible columns, define a set of *column families* that each row should have.
  - example: create a *column family* called anchor
  - can also have ones that are more like typical columns
- In a given row, only store columns with an actual value, representing them as (column key, value) pairs
  - ex: ("anchor:www.bu.edu", "news story")
  - column key = column family:qualifier
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 from previous slide:
    - ("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.

Row Keys and Locality

- We referred on the last slide to a multi-dimensional map:
  - (row key, column key, timestamp) → value

- You can also think of it as a multi-stage map:
  - row key → row
  - once you're at the row, column key → column value

- Rows are sorted by their row key.
  - allows for locality: related rows are stored together
  - example: if using URLs as the key, reverse them to keep related pages together:
    - edu.bu.cs
    - edu.bu.smg
    - edu.bu.www
  - what type of index structure would allow for this sorting?
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?

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 (plus column-family sub-aggregates)</td>
</tr>
</tbody>
</table>

- Relational databases can't fully support aggregation.
- Focus on normalization and 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 need to refer to the address
  - Aggregate-oriented system: store the full customer address in several places:
    - Customer aggregates
    - Order aggregates
    - Etc.

- Why does an aggregate-based approach make sense in the context of NoSQL databases?
Choosing Your Aggregate Boundaries

- There are often multiple ways to group your data into aggregates.

- Example:
  - option 1: group info. about all of a customer's orders with the customer's personal info in a single aggregate
  - option 2: have a separate order aggregate for each order, plus a separate customer aggregate

- Consequences of the aggregate boundaries:
  - impacts efficiency
  - in some systems, atomicity is only guaranteed for changes within a single aggregate.

Issues Related to Aggregation

- A given application may have different access patterns.
  - some accesses may work well within the aggregate boundaries
  - others may not – may need to span aggregates

- One simple way to capture relationships between aggregates:
  - embed the ID of one aggregate in another
  - what is this like?

- In addition:
  - some systems allow you to make these relationships explicit
    - so that the DBMS can form appropriate indices
  - some systems allow you to precompute and cache the results of queries
    - useful for queries than span aggregates
**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:
  - specify the column families in a given table
  - no restrictions on the columns within a given column family
  - different rows can have different columns

**Schemalessness (cont.)**

- Advantages?

- Despite the fact that a schema is not required, programs that use the data need at least an *implicit* schema.
- Disadvantages of having only 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
Schemalessness (cont.)

- How to prevent conflicts over the schema?
  - make the database an *application database*
    - managed by a single application
    - give access to other applications through web services
    - see the earlier notes
  - divide up the aggregates between applications
    - different apps access different column families, or different sections of a document

- Note: schema flexibility only applies within a given type of aggregate.

- Changing aggregate boundaries is as complex as changing the schema of a relational database.