Comparing SQL and NOSQL databases

<table>
<thead>
<tr>
<th></th>
<th>SQL</th>
<th>NOSQL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Types</td>
<td>One type (SQL database) with minor variations</td>
<td>Many different types including key-value stores, document databases, wide-column stores, and graph databases</td>
</tr>
<tr>
<td>Development History</td>
<td>Developed in 1970s to deal with first wave of data storage applications</td>
<td>Developed in 2000s to deal with scale, replication and unstructured data storage</td>
</tr>
<tr>
<td>Data Storage Model</td>
<td>Individual records (e.g., “employees”) are stored as rows in tables, with each column storing a specific piece of data about that record. Separate data types are stored in separate tables, and then joined together when more complex queries are executed.</td>
<td>Varies based on database type. Key-value stores function similarly to SQL databases, but have only two columns. Document databases do away with the table-and-row model altogether, e.g. nest values hierarchically.</td>
</tr>
</tbody>
</table>

Source: http://www.mongodb.com/learn/nosql
### Comparing SQL and NOSQL databases

<table>
<thead>
<tr>
<th>Schemas</th>
<th>Structure and data types are fixed in advance. To store information about a new data item, the entire database must be altered, during which time the database must be taken offline.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Typically dynamic. Records can add new information on the fly, and unlike SQL table rows, dissimilar data can be stored together as necessary.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Scaling</th>
<th>Vertically, meaning a single server must be made increasingly powerful in order to deal with increased demand. It is possible to spread SQL databases over many servers, but significant additional engineering is generally required.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Horizontally, meaning that to add capacity, a database administrator can simply add more commodity servers or cloud instances. The database automatically spreads data across servers as necessary</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Development Model</th>
<th>Mix of open-source (e.g., Postgres, MySQL) and closed source (e.g., Oracle Database)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Open Source</td>
</tr>
</tbody>
</table>

Source: [http://www.mongodb.com/learn/nosql](http://www.mongodb.com/learn/nosql)
HBase

- Column-Oriented data store, known as Hadoop Database
- Distributed - designed to serve large tables
  - Billions of rows and millions of columns
- Runs on a cluster of commodity hardware
  - Server hardware, not laptop/desktops
- Open-source, written in Java
- Type of “NoSQL” DB
  - Does not provide a SQL based access
  - Does not adhere to Relational Model for storage

Slide based on lecture http://www.coreservlets.com/hadoop-tutorial/

HBase

- Automatic fail-over
- Simple Java API
- Integration with Map/Reduce framework
- Based on google’s Bigtable
- Recommended Literature:
  http://labs.google.com/papers/bigtable.html

Slide based on lecture http://www.coreservlets.com/hadoop-tutorial/
HBase

- **Good for:**
  - Lots and lots of data
  - Large amount of clients/requests
  - Single random selects and range scans by key
  - Variable schema, e.g. rows may differ drastically

- **Bad for:**
  - Traditional RDBMs retrieval, e.g. transactional application
  - Relational analytics (e.g. ‘group by’, ‘join’, etc…)
  - Text-based search access


Hbase Data Model

- **Data is stored in Tables**
- **Tables contain rows**
  - Rows are referenced by a unique key
  - Key is an array of bytes - good news
  - Anything can be a key: string, long and your own serialized data structures
- **Rows made of columns which are grouped in column families**
- **Data is stored in cells**
  - Identified by row - column-family - column
  - Cell's content is also an array of bytes

Hbase Families

- Rows are grouped into families
  - Labeled as “family:column”, e.g. “user:first_name”
  - A way to organize your data
- Various features are applied to families
  - Compression
  - In-memory option
  - Stored together - in a file called HFile/StoreFile
- Family definitions are static
  - Created with table, should be rarely added and changed
  - Limited to small number of families
- Unlike columns that you can have millions of

HBase Families

- Family name must be composed of printable characters
  - Not bytes, unlike keys and values
- Think of family:column as a tag for a cell value and NOT as a spreadsheet
- Columns on the other hand are NOT static
  - Create new columns at run-time
  - Can scale to millions for a family

Slide based on lecture http://www.coreservlets.com/hadoop-tutorial/
Hbase Timestamps

- Cells' values are versioned
  - For each cell multiple versions are kept
  - 3 by default
  - Another dimension to identify your data
  - Either explicitly timestamped by region server or provided by the client
- Versions are stored in decreasing timestamp order
- Read the latest first - optimization to read the current value
- You can specify how many versions are kept

Slide based on lecture http://www.coreservlets.com/hadoop-tutorial/

Hbase Values and Row Keys

- Value = Table+RowKey+Family+Column+Timestamp
- Hbase Row Keys
  - Rows are sorted lexicographically by key
  - Compared on a binary level from left to right
  - For example keys 1,2,3,10,15 will get sorted as 1, 10, 15, 2, 3
- Somewhat similar to Relational DB primary index
  - Always unique

Slide based on lecture http://www.coreservlets.com/hadoop-tutorial/
Hbase Architecture

- Table is made of regions
- Region - a range of rows stored together
  - Used for scaling
  - Dynamically split as they become too big and merged if too small
- Region Server - serves one or more regions
  - A region is served by only 1 Region Server
- Master Server - daemon responsible for managing HBase cluster, aka Region Servers
- HBase stores its data into HDFS
  - relies on HDFS's high availability and fault-tolerance features
- HBase utilizes Zookeeper for distributed coordination

Hbase Components
Hbase Regions

- Region is a range of keys
  - start key → stop key (ex. k3cod → odiekd)
    - start key inclusive and stop key exclusive

- Addition of data
  - At first there is only 1 region
  - Addition of data will eventually exceed the configured maximum → the region is split

- Default is 256MB
  - The region is split into 2 regions at the middle key

- Regions per server depend on hardware specs, with today's hardware it's common to have:
  - 10 to 1000 regions per Region Server
  - Managing as much as 1GB to 2 GB per region

Slide based on lecture http://www.coreservlets.com/hadoop-tutorial/

- Splitting data into regions allows
  - Fast recovery when a region fails
  - Load balancing when a server is overloaded

- May be moved between servers
  - Splitting is fast

- Reads from an original file while asynchronous process performs a split
  - All of these happen automatically without user‘ involvement

Slide based on lecture http://www.coreservlets.com/hadoop-tutorial/
Data Storage

- Data is stored in files called HFiles/StoreFiles
  - Usually saved in HDFS
- HFile is basically a key-value map
  - Keys are sorted lexicographically
  - When data is added it’s written to a log called Write Ahead Log (WAL) and is also stored in memory (memstore)
- Flush: when in-memory data exceeds maximum value it is flushed to an Hfile
  - Data persisted to HFile can then be removed from WAL
  - Region Server continues serving read-writes during the flush operations, writing values to the WAL and memstore

Data Storage

- Recall that HDFS doesn’t support updates to an existing file therefore HFiles are immutable
  - Cannot remove key-values out of HFile(s)
  - Over time more and more HFiles are created
- Delete marker is saved to indicate that a record was removed
  - These markers are used to filter the data - to “hide” the deleted records
- At runtime, data is merged between the content of the HFile and WAL
Hbase Master

- Responsible for managing regions and their locations
  - Assigns regions to region servers
  - Re-balanced to accommodate workloads
  - Recovers if a region server becomes unavailable
  - Uses Zookeeper - distributed coordination service
  - Doesn't actually store or read data
  - Clients communicate directly with Region Servers
  - Usually lightly loaded
- Responsible for schema management and changes
  - Adding/Removing tables and column families

Zookeeper

- HBase uses Zookeeper extensively for region assignment
- Zookeeper is a centralized service for maintaining configuration information, naming, providing distributed synchronization, and providing group
- HBase can manage Zookeeper daemons for you or you can install/manage them separately
- Learn More at http://zookeeper.apache.org
Hbase Access

- HBase Shell
- Native Java API
  - Fastest and very capable options
- Avro Server
  - Apache Avro is also a cross-language schema compiler
  - Requires running Avro Server
- HBql
  - SQL like syntax for Hbase
  - [http://www.hbql.com](http://www.hbql.com)

---

Hbase shell

> hbase shell
HBase Shell; enter 'help<RETURN>' for list of supported commands.
Type "exit<RETURN>" to leave the HBase Shell
Version 0.90.4-cdh3u2, r, Thu Oct 13 20:32:26 PDT 2011
hbase(main):001:0> list
  TABLE
  0 row(s) in 0.4070 seconds
HBase shell

- Quote all names
  - Table and column name
  - Single quotes for text
    hbase> get 't1', 'myRowId'
- Double quotes for binary
- Use hexadecimal representation of that binary value
  hbase> get 't1', "key\x03\x3f\xcd"
- Uses ruby hashes to specify parameters
  {'key1' => 'value1', 'key2' => 'value2', ...}
- Example:
  hbase> get 'UserTable', 'userId1', {COLUMN => 'address: str'}

Slide based on lecture http://www.coreservlets.com/hadoop-tutorial/

HBase shell commands

- General
  - status, version
- Data Definition Language (DDL)
  - alter, create, describe, disable, drop, enable, exists, list, ...
- Data Manipulation Language (DML)
  - count, delete, deleteall, get, incr, put, scan, truncate, ...
- Cluster administration
  - balancer, close_region, compact, flush, major_compact, move,
    split, unassign, zk_dump, add_peer, disable_peer,
    enable_peer, ...
- Learn more about each command
  hbase> help "<command>"

Slide based on lecture http://www.coreservlets.com/hadoop-tutorial/
- Create Table with two families (info, content)
  
  hbase> create 'Blog', {NAME=>'info'}, {NAME=>'content'}
  
  0 row(s) in 1.3580 seconds

- Populate table with data records
  
  Format: put 'table', 'row_id', 'family:column', 'value'

- Example
  
  hbase> put 'Blog', 'Matt-001', 'info:title', 'Elephant'
  hbase> put 'Blog', 'Matt-001', 'info:author', 'Matt'

Access data - count

- Count: display the total number of records
  - get: retrieve a single row
  - scan: retrieve a range of rows
  
  Format: count 'table_name'

- Will scan the entire table!
  
  hbase> count 'Blog', {INTERVAL=>2}
  
  Current count: 2, row: John-005
  Current count: 4, row: Matt-002
  5 row(s) in 0.0220 seconds
Access data - get

- Select single row with 'get' command
  Format: get 'table', 'row_id'
  - Returns an entire row
  - Requires table name and row id
  - Optional: timestamp or time-range, and versions

- Select specific columns
  hbase> get 't1', 'r1', {COLUMN => 'c1'}
  hbase> get 't1', 'r1', {COLUMN => ['c1', 'c2', 'c3']}

- Select specific timestamp or time-range
  hbase> get 't1', 'r1', {TIMERANGE => [ts1, ts2]}
  hbase> get 't1', 'r1', {COLUMN => 'c1', TIMESTAMP => ts1}

- Select more than one version
  hbase> get 't1', 'r1', {VERSIONS => 4}

Access data - Scan

- Scan entire table or a portion of it
- Load entire row or explicitly retrieve column families, columns or specific cells
- To scan an entire table
  Format: scan 'table_name'

- Limit the number of results
  Format: scan 'table_name', {LIMIT=>1}

- Scan a range
  Format: scan 'table_name', {STARTROW=>'startRow', STOPROW=>'stopRow'}
  - Start row is inclusive, stop row is exclusive
  - Can provide just start row or just stop row
Edit data

- Put command inserts a new value if row id doesn't exist
- Put updates the value if the row does exist
- But does it really update?
  - Inserts a new version for the cell
  - Only the latest version is selected by default
  - N versions are kept per cell configured per family at creation
  - 3 versions are kept by default

Format: create 'table', {NAME => 'family', VERSIONS=>7}

hbase> put 'Blog', 'Michelle-004', 'info:date', '1990.07.06'
0 row(s) in 0.0520 seconds
hbase> put 'Blog', 'Michelle-004', 'info:date', '1990.07.07'
0 row(s) in 0.0080 seconds
hbase> put 'Blog', 'Michelle-004', 'info:date', '1990.07.08'
0 row(s) in 0.0060 seconds
hbase> get 'Blog', 'Michelle-004', {COLUMN=>'info:date', VERSIONS=>3}
COLUMNS CELL
info:date timestamp=1326071670471, value=1990.07.08
info:date timestamp=1326071670442, value=1990.07.07
info:date timestamp=1326071670382, value=1990.07.06
3 row(s) in 0.0170 seconds
- Delete cell by providing table, row id and column coordinates
  - delete 'table', 'rowId', 'column'
  - Deletes all the versions of that cell
- Optionally add timestamp to only delete versions before the provided timestamp
  Format: delete 'table', 'rowId', 'column', timestamp

- Must disable before dropping
  - puts the table “offline” so schema based operations can be performed
  hbase> disable 'table_name'
  hbase> drop 'table_name'
- For a large table it may take a long time....
hbase> list
TABLE
Blog
1 row(s) in 0.0120 seconds
hbase> disable 'Blog'
0 row(s) in 2.0510 seconds
hbase> drop 'Blog'
0 row(s) in 0.0940 seconds
hbase> list
TABLE
0 row(s) in 0.0200 seconds