NoSQL

Columnar Databases

Problem Set #5 is due tonight
Problem Set #6 will be online tonight, or you all get 100.
Agenda

❖ Data Model
  ▪ Column families, super columns, two points of view

❖ Column-family Stores
  ▪ Google BigTable, Cassandra, HBase

❖ Cassandra as an Example
  ▪ Cassandra data model 1.0 vs. 2.0
  ▪ Cassandra Query Language (CQL)
  ▪ Data partitioning, replication
  ▪ Local Data Persistence
  ▪ Query processing, Indexes, Lightweight Transactions
Column-family Stores: Basics

❖ AKA: wide-column, columnar
  ▪ not to confuse with column-oriented RDBMS

❖ Data model: rows that have many columns associated with a row key

❖ Column families are groups of related data (columns) that are often accessed together
  ▪ e.g., for a customer we typically access all profile information at the same time, but not customer’s orders
Data Model

❖ Columns within Rows = the basic data item
  ▪ a 3-tuple consisting of
    • column name
    • value
    • timestamp

  ▪ Can be modeled as follows

    { name: "firstName",
      value: "Martin",
      timestamp: 12345667890 }

❖ In the following, we will ignore the timestamp
Data Model

❖ **Row**: a collection of columns with a common row key
  - Columns can be added to any row at any time
    - without having to add it to other rows

```javascript
// row
"martin-fowler" : { ← Row key
  firstName: "Martin",
  lastName: "Fowler",
  location: "Boston"
}
```

| Row key1  | Column Key1 | Column Key2 | Column Key3 |...
|-----------|-------------|-------------|-------------|
| Column Value1 | Column Value2 | Column Value3 |...

...
Data Model: Column Family

- **CF** = *Set* of columns containing “related” data

<table>
<thead>
<tr>
<th>user_id (row key)</th>
<th>column key</th>
<th>column key</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><em>column value</em></td>
<td><em>column value</em></td>
<td>...</td>
</tr>
<tr>
<td>1</td>
<td>login</td>
<td>first_name</td>
<td>...</td>
</tr>
<tr>
<td></td>
<td>gonzo</td>
<td>Sam</td>
<td>...</td>
</tr>
<tr>
<td>4</td>
<td>login</td>
<td>age</td>
<td>...</td>
</tr>
<tr>
<td></td>
<td>david</td>
<td>35</td>
<td>...</td>
</tr>
<tr>
<td>5</td>
<td>first_name</td>
<td>last_name</td>
<td>...</td>
</tr>
<tr>
<td></td>
<td>Kathy</td>
<td>Wright</td>
<td>...</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Data Model: Column Family (2)

❖ Column family - example as JSON

```
{ // row (columns from a CF)  // row (cols from the same CF)
  "pramod-sadalage" : {  "martin-fowler" : {
    firstName: "Pramod",  firstName: "Martin",
    lastName: "Sadalage",  lastName: "Fowler",
    lastVisit: "2012/12/12"  location: "Boston",
  }  activ: "true" }  }
```

source: Sadalage & Fowler: NoSQL Distilled, 2012
Data Model: Super Column Family

❖ Super column
  ▪ A column whose value is composed of a map of columns
  ▪ Used in some column-family stores (Cassandra 1.0)

❖ Super column family
  ▪ A column family consisting of super columns
### Super Column Family: Example

<table>
<thead>
<tr>
<th>user_id (row key)</th>
<th>super column key</th>
<th>super column key</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>subcolumn key</td>
<td>subcolumn key</td>
<td>...</td>
</tr>
<tr>
<td></td>
<td>subcolumn key</td>
<td>subcolumn key</td>
<td>...</td>
</tr>
<tr>
<td></td>
<td>subcolumn value</td>
<td>subcolumn value</td>
<td>...</td>
</tr>
<tr>
<td></td>
<td>subcolumn value</td>
<td>subcolumn value</td>
<td>...</td>
</tr>
<tr>
<td>1</td>
<td>home_address</td>
<td>work_address</td>
<td></td>
</tr>
<tr>
<td></td>
<td>city</td>
<td>street</td>
<td>...</td>
</tr>
<tr>
<td></td>
<td>Raleigh</td>
<td>Hillsborough St</td>
<td>...</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Chapel Hill</td>
<td>Raleigh St</td>
</tr>
<tr>
<td>4</td>
<td>home_address</td>
<td>temporary_address</td>
<td></td>
</tr>
<tr>
<td></td>
<td>city</td>
<td>street</td>
<td>...</td>
</tr>
<tr>
<td></td>
<td>Durham</td>
<td>Chapel Hill St</td>
<td>...</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Chapel Hill</td>
<td>Raleigh St</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Super Column Family in JSON

```json
{
    "Cathy": {
        "username": {
            "firstname": "Cathy",
            "lastname": "Qi"
        },
        "address": {
            "city": "New York",
            "zip": "10001"
        }
    },
    "Terry": {
        "username": {
            "firstname": "Terry",
            "lastname": "Martin"
        },
        "account": {
            "bank": "Citi",
            "account": 12346789
        },
        "background": {
            "birthdate": "1990-03-04"
        }
    }
}
```
Column Family Stores: Features

- **Data model**: Column families
- **System architecture**
  - Data partitioning
- **Local persistence**
  - Update log, memory, disk...
- **Data replication**
  - Balancing of the data
- **Query processing**
  - Query language
- **Indexes**
Representatives
BigTable

- **Google’s paper:**

- **Proprietary, not distributed outside Google**
  - used in Google Cloud Platform

- **Data model:** column families as defined above
  - “A table in Bigtable is a sparse, distributed, persistent multidimensional sorted map.”

\[
(row: \text{string}, \ column: \text{string}, \ time: \text{int64}) \rightarrow \text{string}
\]
“BigTable = sparse, distributed, persistent, multi-dimensional sorted map indexed by \((row\_key, column\_key, timestamp)\)”
BigTable: Architecture
Cassandra

- Developed at Facebook
  - now, Apache Software License 2.0
- Initial release: 2008
- Written in: Java
- OS: cross-platform
- Operations:
  - CQL (Cassandra Query Language)
  - MapReduce support (can cooperate with Hadoop)
Cassandra: Data Model

- **Column families, super column families**
  - Can define metadata about columns
  - Now denoted as: Thrift API

- **Static** – similar to a relational database table
  - **Rows** have the **same** limited set of **columns**
  - However, rows are **not required** to have all columns

- **Dynamic** – takes advantage of Cassandra's ability to use **arbitrary new column names**

http://cassandra.apache.org/
## Cassandra: Data Model

### Row Key Columns

<table>
<thead>
<tr>
<th>Row Key</th>
<th>First</th>
<th>Last</th>
<th>College</th>
<th>DOB</th>
<th>Height</th>
<th>Weight</th>
<th>2016</th>
</tr>
</thead>
<tbody>
<tr>
<td>24911</td>
<td>Alex</td>
<td>Smith</td>
<td>Utah</td>
<td>1984-05-07</td>
<td>6-4</td>
<td>212</td>
<td>chiefs</td>
</tr>
<tr>
<td>27073</td>
<td>Blaine</td>
<td>Gabbert</td>
<td>Missouri</td>
<td>1989-10-15</td>
<td>high school</td>
<td>Parkway West</td>
<td></td>
</tr>
<tr>
<td>27154</td>
<td>Colin</td>
<td>Kaepernick</td>
<td>Nevada</td>
<td>6-4</td>
<td>225</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Key Spaces

- Databases are called "KEYSPACES" in Cassandra
- They are created as follows:

```
CREATE KEYSPACE db
    WITH replication = {
        'class': 'SimpleStrategy',
        'replication_factor': '2'
    }
    AND durable_writes = 'true';

DESCRIBE KEYSPACES;
```
KEYSPACES can have TABLES defined as follows

```sql
CREATE TABLE Player ( pid int PRIMARY KEY,
                      first TEXT,
                      last TEXT,
                      college TEXT);

DESCRIBE TABLES;
```
CREATE TABLE users (  
    user_id int PRIMARY KEY,  
    login text,  
    name text,  
    email text );

INSERT INTO users (user_id, login, name)  
VALUES (3, 'cathyqi', 'Cathy Qi');

SELECT * FROM users;  
user_id | email | login | name  
---------|-------|-------|-------  
3 | null | cathyqi | Cathy Qi
An alternative to tables are COLUMN FAMILIES

Requires a Name and Comparators

- A **key** must be specified
- Data **types** for columns can be specified
- Options can be specified

```
CREATE COLUMNFAMILY Fish (key blob PRIMARY KEY);
CREATE COLUMNFAMILY FastFoodPlaces (name text PRIMARY KEY)
    WITH comparator=timestamp AND default_validation=int;
CREATE COLUMNFAMILY MonkeyTypes (
    key uuid PRIMARY KEY,
    species text,
    alias text,
    population varint
) WITH comment='Important biological records'
    AND read_repair_chance = 1.0;
```
 Comparator = data type for a column name
 Validator = data type of a column value
 - or content of a row key

 Data types do not need to be defined
 - Default: BytesType, i.e. arbitrary hexadecimal bytes

 Basic operations: GET, SET, DEL
create column family users
    with key_validation_class = Int32Type
    and comparator = UTF8Type
    and default_validation_class = UTF8Type;

// set column values in row with key 7
set users[7]['login'] = utf8('cathyqi');
set users[7]['name'] = utf8('Cathy Qi');
set users[7]['email'] = utf8('qi@best.com');

set users[13]['login'] = utf8('fantom');
set users[13]['name'] = utf8('Un Known');
get users[7]["login"];
=> (name=login, value=cathyqi, timestamp=1429268223462000)

get users[13];
=> (name=login, value=fantom, timestamp=1429268224554000)
=> (name=name, value=Un Known, timestamp=1429268224555000)

list users;
RowKey: 7
=> (name=email, value=qi@best.com, timestamp=1429268223471000)
=> (name=login, value=cathyqi, timestamp=1429268223462000)
=> (name=name, value=Cathy Qi, timestamp=1429268223471000)
-------------------
RowKey: 13
=> (name=login, value=fantom, timestamp=1429268224554000)
=> (name=name, value=Un Known, timestamp=1429268225231000)
Cassandra: Sparse Tables

- **CQL:** Cassandra Query Language
  - SQL-like commands
    - CREATE, ALTER, UPDATE, DROP, DELETE, TRUNCATE, INSERT, …
  - Simpler than SQL

- Since CQL 3 (Cassandra 1.2)
  - Column -> cell
  - Column family -> table

- **Dynamic columns** (wide rows) still supported
  - CQL supports everything that was possible before
  - “Old” approach (Thrift API) can be used as well
Values can use “collection” types:
- set – unordered unique values
- list – ordered list of elements
- map – name + value pairs
  - a way to realize super-columns

Realization of the original idea of free columns
- Internally, all values in collections as individual columns
- Cassandra can well handle “unlimited” number of columns
CREATE TABLE users (  
  login text PRIMARY KEY,  
  name text,  
  emails set<text>, // column of type “set”  
  profile map<text, text> // column of type “map”  
)

INSERT INTO users (login, name, emails, profile)  
VALUES ( 'honza', 'Jan Novák', { 'honza@novak.cz' },  
  { 'colorschema': 'green', 'design': 'simple' }  
);

UPDATE users  
SET emails = emails + { 'jn@firma.cz' }  
WHERE login = 'honza';
Dynamic Columns: Another Way

- **Compound primary key**

```sql
CREATE TABLE mytable (  
  row_id int, column_name text, column_value text,  
  PRIMARY KEY (row_id, column_name)  
);

INSERT INTO mytable (row_id, column_name, column_value)  
VALUES (3, 'login', 'honza');

INSERT INTO mytable (row_id, column_name, column_value)  
VALUES (3, 'name', 'Jan Novák');

INSERT INTO mytable (row_id, column_name, column_value)  
VALUES (3, 'email', 'honza@novak.cz');
```
Data Sharding in Columnar Systems

<table>
<thead>
<tr>
<th>System</th>
<th>Terminology</th>
</tr>
</thead>
<tbody>
<tr>
<td>BigTable</td>
<td>tablets</td>
</tr>
<tr>
<td>HBase</td>
<td>regions</td>
</tr>
<tr>
<td>Cassandra</td>
<td>partitions</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>user_id (row key)</th>
<th>login</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>cathyqi</td>
<td>Cathy Qi</td>
</tr>
<tr>
<td>4</td>
<td>davidm</td>
<td>David Man</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1000</td>
<td>iamboo</td>
<td>Nota James</td>
</tr>
<tr>
<td>1001</td>
<td>violet</td>
<td>Ziwei Chen</td>
</tr>
<tr>
<td>1003</td>
<td>ernie</td>
<td>Bernard ...</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2000</td>
<td>peach</td>
<td>Joseph Ash</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Data Sharding in Cassandra

- Entries in each table are **split by partition key**
  - Which is a selected **column** (or a set of columns)
  - Specifically, the **first column** (or columns) from the **primary** key is the **partition key** of the table

```sql
CREATE TABLE tab ( a int, b text, c text, d text,
    PRIMARY KEY ( a, b, c)
);

CREATE TABLE tab ( a int, b text, c text, d text,
    PRIMARY KEY ( (a, b), c)
);
```
All entries with the same partition key
  ▪ Will be stored on the same physical node
  ▪ => efficient processing of queries on one partition key

```
CREATE TABLE mytable (
    row_id int, column_name text, column_value text,
    PRIMARY KEY (row_id, column_name)
);
```

The rest of the columns in the primary key
Are so called clustering columns
  ▪ Rows are locally sorted by values in the clustering columns
    • the order for physical storing rows
Data Replication

❖ Cassandra adopts peer-to-peer replication
  ▪ The same principles like in key-value stores & document DB
  ▪ Read/Write quora to balance between availability and consistency guarantees

❖ Google BigTable
  ▪ Physical data distribution & replication is done by the underlying distributed file system
  ▪ GFS
Cassandra Query Language (CQL)

- The syntax of CQL is similar to SQL
  - But search just in one table (no joins)

```
SELECT <selectExpr>
FROM [<keyspace>.]<table>
WHERE <clause>
ORDER BY <clustering_colname> [DESC]
LIMIT m;
```

```
SELECT column_name, column_value
FROM mytable
WHERE row_id=3
ORDER BY column_value;
```
The search condition can be:

- on columns in the partition key
  - And only using operators == and IN

... WHERE row_id IN (3, 4, 5)
  - Therefore, the query hits only one or several physical nodes (not all)

- on columns from the clustering key
  - Especially, if there is also condition on the partitioning key

... WHERE row_id=3 AND column_name='login'
  - If it is not, the system must filter all entries

SELECT * FROM mytable
WHERE column_name IN ('login', 'name') ALLOW FILTERING;
Other columns can be queried
- If there is an index built on the column

Indexes can be built also on collection columns (set, list, map)
- And then queried by CONTAINS like this

```sql
SELECT login FROM users
WHERE emails CONTAINS 'jn@firma.cz';
SELECT * FROM users
WHERE profile CONTAINS KEY 'colorschema';
```
Secondary indexes on any column
  ▪ B⁺-Tree indexes
  ▪ User-defined implementation of indexes

CREATE INDEX ON users (emails);
Transactions

- Cassandra 2.x supports “lightweight transactions”
  - compare and set operations
  - using Paxos consensus protocol
    - nodes agree on proposed data additions/modifications
    - faster than Two-phase commit protocol (P2C)

```sql
INSERT INTO users (login, name, emails)
VALUES ('cathyqi', 'Cathy Qi', { 'qi@best.com' })
IF NOT EXISTS;
```

```sql
UPDATE mytable SET column_value = 'qi@best.org'
WHERE row_id = 3 AND column_name = 'email'
IF column_value = 'qi@best.com';
```
Summary

❖ Column-family stores
  ▪ are worth only for **large data** and large query **throughput**
  ▪ two ways to see the **data model**:
    • large sparse **tables** or multidimensional (nested) **maps**
  ▪ data distribution is via row key
    • analogue of **document ID** or **key** in **document** or **key-value** stores
  ▪ efficient disk + memory local data storage

❖ Cassandra
  ▪ **CQL**: structured after **SQL**, easy transition from RDBMS