UMD DATA605 - Big Data Systems

Issues with Relational DBs NoSQL Taxonomy (Apache) HBase

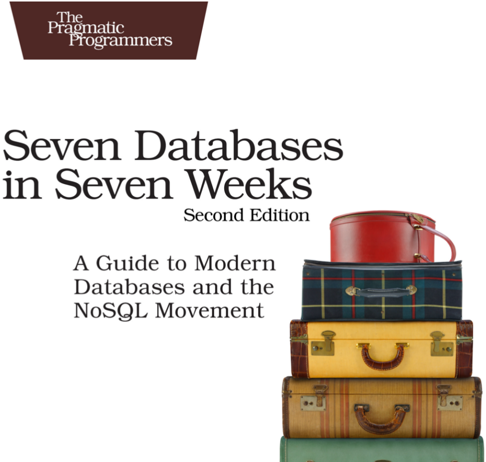
Instructor: Dr. GP Saggese - gsaggese@umd.edu**
TAs: Krishna Pratardan Taduri, kptaduri@umd.edu Prahar Kaushikbhai Modi, pmodi08@umd.edu
v1.1

Jupyter Tutorial

- Let's start with a tutorial of Jupyter notebooks
- Jupyter tutorial dir
- Readme
  - Explains how to run the tutorial
- Notebook to execute / study

Resources

- Concepts in the slides
- Tons of tutorials on line
- Silbershatz Chap 10.2
- Nice high-level view:
  - Seven Databases in Seven Weeks, 2e



From SQL to NoSQL

- **DBs are central tools to big data**
  - New applications, new constraints to data / storage
  - Around 2000s NoSQL "movement" started
    - Initially it meant "No SQL" -> "Not Only SQL"
- **DBs (e.g., SQL vs NoSQL) make different trade-offs**
  - Different worldviews
  - Schema vs schema-less
  - Rich vs fast ability of query
  - Strong consistency (ACID), weak, eventual consistency
  - APIs (SQL, JS, REST)
  - Horizontal vs vertical scaling, sharding, replication schemes
  - Indexing (for rapid lookup) vs no indexing
  - Tuned for reads or writes, how much control over tuning
- **The user base / applications have expanded**
  - IMO Postgres + Mongo cover 99% of use cases
  - Any data scientist / engineer needs to be familiar with both
  - "Which DB solves my problem best?"
- **Polyglot model**
  - Use more than one DB in each project
  - Relational DBs are not going to disappear any time soon

Issues with Relational DBs

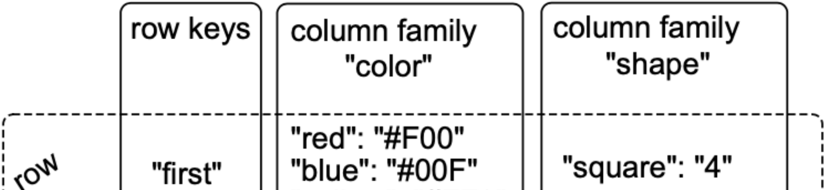
- **Relational DBs have drawbacks**
  - 1 Application-DB impedance mismatch
  - 2 Schema flexibility
  - 3 Consistency in distributed set-up
  - 4 Limited scalability
- In the next slides for each drawback we will discuss:
  - **What is the problem**
  - **Possible solutions**
    - Within relational SQL paradigm
    - With NoSQL approach

1 App-DB Impedance Mismatch: Problem

- **Mismatch between how data is represented in the code and in a relational DB**
  - Code thinks in terms of:
    - Data structures (e.g., lists, dictionaries, sets)
    - Objects
  - Relational DB thinks in terms of:
    - Tables (entities)
    - Rows (actual instances of entities)
    - Relationships between tables (relationships between entities)
- **Example of the app-DB mismatch**:
  - Application stores a simple Python map like: #### 1 App-DB Impedance Mismatch: Solutions
- **Ad-hoc mapping layer**
  - Translate objects and data structures into DB data model
    - E.g., you implement a layer that handles storing into the DB "Name to Tags" transparently
    - The code thinks in terms of a map, but there are 3 tables in the DB
  - Cons
    - You need to write / maintain code
- **Object-relational mapping (ORM)**
  - Pros
    - Convert automatically data between object code and relational DB
    - E.g., implement a Person object (e.g., name, phone number, addresses)

Example 1: Colors and Shape

- Table with:
  - 2 column families
    - “color” and “shape”
  - 2 rows
    - “first” and “second”
- The row “first” has:
  - 3 columns in the column family “color”
    - “red”, “blue”, “yellow”
  - 1 column in the column family “shape”
    - shape = 4
- The row “second” has:
  - no columns in “color”
  - 2 columns in the column family “shape”
- Data is accessed using a row key and column (family:qualifier)

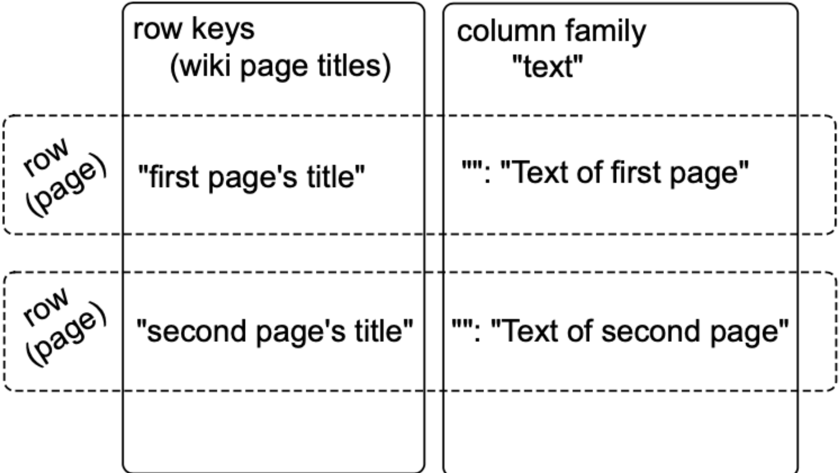


Why all this convoluted stuff?

- **A row in HBase is almost like a mini-database**
  - A cell has many different values associated with it
  - Data is stored in a sparse format
- **Rows in HBase are "deeper" than in relational DBs**
  - In relational DBs rows contain a lot of column values (fixed array with types)
  - In HBase rows contain something like a two-level nested dictionary and metadata (e.g., timestamp)
- **Applications**
  - Store versioned web-site data
  - Store a wiki

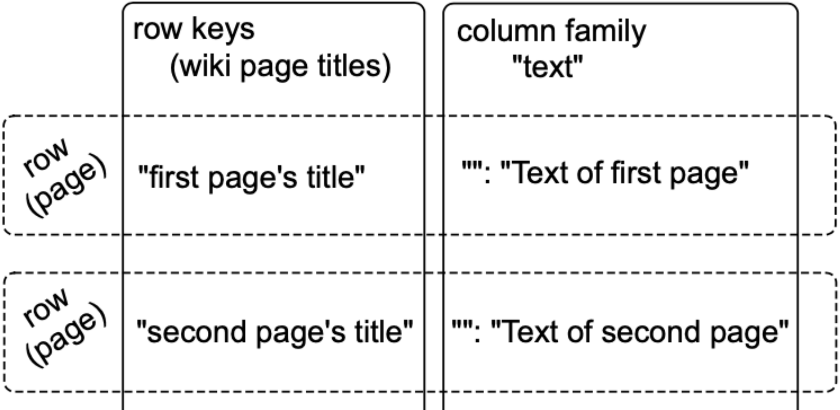
Example 2: Storing a Wiki

Wiki (e.g., Wikipedia) - Contains pages - Each page has a title, an article text varying over time **HBase data model** - Table name → `wikipedia` - Row → entire wiki page - Row keys → wiki identifier (e.g., title or URL) - Column family → `text` - Column → ” (empty) - Cell value → article text



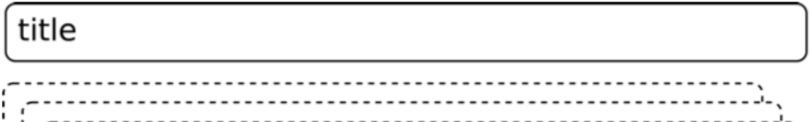
Example 2: Storing a Wiki

Add data - Columns don't need to be predefined when creating a table - The column is defined as `text` > put 'wikipedia', 'Home', 'text', 'Welcome!' **Query data** - Specify the table name, the row key, and optionally a list of columns > get 'wikipedia', 'Home', 'text' text: timestamp=1295774833226, value=Welcome! - HBase returns the timestamp (ms since the epoch 01-01-1970 UTC)



Example 2: Improved Wiki

- **Improved wiki using versioning**
- A page
  - Is uniquely identified by its title
  - Can have multiple revisions
- A revision
  - Is made by an author
  - Contains optionally a commit comment
  - Is identified by its timestamp
  - Contains text
- **HBase data model**
- Add a family column “revision” with multiple columns
  - E.g., author, comment, . . .
- Timestamp is automatic and binds article text and metadata
- The title is not part of the revision
  - It’s fixed and identified uniquely the page (like a primary key)
  - If you want to change the title you need to re-write all the row



Data in Tabular Form

	Name	Home	Office			
Key	First	Last	Phone	Email	Phone	Email
101	Florian	Krepsbach	555-1212	florian@wobegon.org	666-1212	fk@phc.com
102	Marilyn	Tollerud	555-1213		666-1213	
103	Pastor	Inqvist			555-1214	inqvist@wel.or

- Fundamental operations
  - CREATE table, families
  - PUT table, rowid, family:column, value
  - PUT table, rowid, whole-row
  - GET table, rowid
  - SCAN table *WITH filters*
  - DROP table

Data in Tabular Form

	Name	Home	Office	Social				
Key	First		Last	Phone	Email	Phone	Email	FacebookID
101	Florian	Garfield	Krepsbach	555-1212	florian@wobegon.org	666-1212	fk@phc.com	
102	Marilyn		Tollerud	555-1213		666-1213		
103	Pastor		Inqvist			555-1214	inqvist@wel.org	

New columns can be added at runtime

Column families cannot be added at runtime

```
Table People(Name, Home, Office)
{
    101: {
        Timestamp: T403;
        Name: {First="Florian", Middle="Garfield", Last="Krepsbach
        Home: {Phone="555-1212", Email="florian@wobegon.org"},
        Office: {Phone="666-1212", Email="fk@phc.com"},
```

Nested Data Representation

```
**GET People:101**
    {
        Timestamp: T403;
        Name: {First="Florian", Last="Krepsbach"},
        Home: {Phone="555-1212", Email="florian@wobegon.org"},
        Office: {Phone="666-1212", Email="fk@phc.com"}
    }
**GET People:101:Name**
    {First="Florian", Last="Krepsbach"}
**GET People:101:Name:First**
    "Florian"
```

	Name	Home	Office			
Key	First	Last	Phone	Email	Phone	Email
101	Florian	Krepsbach	555-1212	florian@wobegon.org	666-1212	fk@phc.com
102	Marilyn	Tollerud	555-1213		666-1213	
103	Pastor	Inqvist			555-1214	inqvist@wel.or

Column Family vs Column

- **Adding a column**
  - Is cheap
  - Can be done at run-time
- **Adding a column family**
  - Can't be done at run-time
  - Need a copy operation of the table (expensive)
  - This tells you something about how the data is stored
    - Easy to add is a map
    - Hard to add is some sort of static array
    - E.g., MongoDB document vs Relational DB column
- **Why differentiating column families vs columns?**
  - Why not storing all the row data in a single column family?
  - Each column family can be configured independently, e.g.,
    - Compression
    - Performance tuning
    - Stored together in files
  - Everything is designed to accommodate a special kind of data
    - E.g., timestamped web data for search engine

Consistency Model

- **Atomicity**
  - Entire rows are updated atomically or not at all
  - Independently of how many columns are affected
- **Consistency**
  - A GET is guaranteed to return a complete row that existed at some point in the table's history
    - Weak / eventual consistency
    - Check the timestamp to be sure!
  - A SCAN
    - Must include all data written prior to the scan
    - May include updates since it started
- **Isolation**
  - Concurrent vs sequential semantics
  - Not guaranteed outside a single row
  - The atom of information is a row
- **Durability**
  - All successful writes have been made durable on disk

Checking for Row or Column Existence

- HBase supports Bloom filters to check whether a row or column exists
  - It's like a cache for key in keys, instead of keys[key]
  - E.g., instead of querying one can keep track of what's present
- **Hashset complexity**
  - Space needed to store data is unbounded
  - No false positives
  - O(1) in average / amortized (because of reallocations, re-balancing)
- **Bloom filter implementation**
  - Bloom filter is like a probabilistic hash set
  - Array of bits initially all equal to 0
  - When a new blob of data is presented, turning the blob into a hash, and then use hash to set some bits to 1
  - To test if we have seen a blob, compute the hash, check the bits
    - If all bits are 0s, then for sure we didn't see it
    - If all bits are 1s, then it's likely but not sure you have seen that blob (false positive)
- **Bloom filter complexity**
  - Use a constant amount of space
  - Has false positives (no false negatives)
  - O(1)

SCIENCE ACADEMY

Write-Ahead Log (WAL)

- Write-Ahead Log is a general technique used by DBs
  - Provide atomicity and durability
  - Protect against node failures
  - Equivalent to journaling in file system
- HBase and Postgres uses WAL
- **WAL mechanics**
- For performance reasons, the updated state of tables are:
  - Not written to disk immediately
  - Buffered in memory
  - Written to disk as checkpoints periodically
- **Problem**
  - If the server crashes during this limbo period, the state is lost
- **Solution**
  - Use append-only disk-resident data structure
  - Log of operations performed since last table checkpoint are appended to the WAL (it's like storing deltas)
  - When tables are stored to disk, the WAL is cleared
  - If the server crashes during the limbo period, use WAL to recover the state that was not written yet
- When running a big import job, disable the WAL to improve performance
  - Trade off disaster recovery protection for speed

Storing variable-length data in DBs

ID FirstName LastName Phone 101 Florian Krepsbach **555-3434** 102 Marilyn Tollerud 555-1213 103 Pastor Ingvist 555-1214

{ 101: { Timestamp: T403; Name: {First="Florian", Middle="Garfield", Last="Krepsbach"}, Home: {Phone="555-1212", Email="florian@wobegon.org"}, Office: {Phone="666-1212", Email="fk@phc.com"} }, ... }

SQL Table People(ID: Integer, FirstName: CHAR[20], LastName: CHAR[20], Phone: CHAR[8]) UPDATE People SET Phone="555-3434" WHERE ID=403;

HBase Table People(ID, Name, Home, Office) PUT People, 403, Home:Phone, 555-3434

- Each row is exactly $4 + 20 + 20 + 8 = 52$ bytes long
- To move to the next row: fseek(file,+52)
- To get to Row 401 fseek(file, 401*52);
- Overwrite the data in place

Need to use pointers

SCIENCE ACADEMY

HBase Implementation

- **How to store the web on disk?**
- **HBase is backed by HDFS**
  - Store each table (e.g., Wikipedia) in one file
  - "One file" means one gigantic file stored in HDFS
    - HDFS splits/replicate file into blocks on different servers
- Here is the idea in several steps:
  - **Idea 1: Put an entire table in one file**
    - Need to overwrite the file every time there is a change in any cell
    - Too slow
  - `text Idea 2: One file + WAL`
    - Better, but doesn't scale to large data
  - `text Idea 3: One file per column family + WAL`
    - Getting better!
  - `text Idea 4: Partition table into regions by key`
    - Region = a chunk of rows [a, b)
    - Regions never overlap

Idea 1: Put the Table in a Single File

- How do we do the following operations?
  - CREATE, DELETE (easy / fast)
  - SCAN (easy / fast)
  - GET, PUT (difficult / slow)

Table People(Name, Home, Office) { 101: { Timestamp: T403; Name: {First="Florian", Middle="Garfield", Last="Krepsbach"}, Home: {Phone="555-1212", Email="florian@wobegon.org"}, Office: {Phone="666-1212", Email="fk@phc.com"} }, 102: { Timestamp: T593; Name: {First="Marilyn", Last="Tollerud"}, Home: {Phone="555-1213"}, Office: {Phone="666-1213"} }, . . . }

File "People"

UNIVERSITY OF MARYLAND 18 56 SCIENCE ACADEMY

Idea 2: One file + WAL

Table People(Name, Home, Office)

PUT 101:Office:Phone = “555-3434” PUT 102:Home:Email = mt@yahoo.com

. . . .

WAL for Table People

- Changes are applied only to the log file
- The resulting record is cached in memory
- Reads must consult both memory and disk

Memory Cache for Table People

101

102

GET People:101

GET People:103

PUT People:101:Office:Phone = “555-3434”

```
101: {
    Timestamp: T403;
```

Idea 2 Requires Periodic Table Update

101: {Timestamp: T403;Name: {First="Florian", Middle="Garfield", Last="Krepsbach"},Home: {Phone="555-1212", Email="florian@wobegon.org"},Office: {Phone="666-1212", Email="fk@phc.com"}}, 102: {Timestamp: T593;Name: { First="Marilyn", Last="Tollerud"},Home: { Phone="555-1213" },Office: { Phone="666-1213" }}, . . .

Table for People on Disk (Old)

PUT 101:Office:Phone = "555-3434" PUT 102:Home:Email = mt@yahoo.com . . .

WAL for Table People:

101: {Timestamp: T403;Name: {First="Florian", Middle="Garfield", Last="Krepsbach"},Home: {Phone="555-1212", Email="florian@wobegon.org"},Office: {Phone="**555-3434**", Email="fk@phc.com"}},102: {Timestamp: T593;Name: { First="Marilyn", Last="Tollerud"},Home: { Phone="555-1213", **Email="my@yahoo.com"** }, . . .

Table for People on Disk (New)

Idea 3: Partition by Column Family

Data for Column Family Name

Tables for People on Disk (Old)

PUT 101:Office:Phone = "555-3434" PUT 102:Home:Email = mt@yahoo.com

. . . .

WAL for Table People

Tables for People on Disk (New)

- Write out a new copy of the table, with all of the changes applied
- Delete the log and memory cache
- Start over

Data for Column Family Home

Data for Column Family Office

Data for Column Family Home (Changed)

Data for Column Family Office (Changed)

Data for Column Family Name

Idea 4: Split Into Regions

Region 1: Keys 100-200

Region 2: Keys 100-200

Region 3: Keys 100-200

Region 4: Keys 100-200

Region Server

Region Master

Region Server

Region Server

Region Server

Transaction Log

Memory Cache

Table

Table

Final HBase Data Layout