UMD DATA605 - Big Data Systems
NoSQL Stores
NoSQL Taxonomy
(Apache) HBase

Dr. GP Saggese
gsaggese@umd.edu

with thanks to Prof.
Alan Sussman (UMD)
Amol Deshpande (UMD)
Oliver Kennedy (U. Buffalo)
Doug Thain (U. Notre Dame)

UMD DATA605 - Big Data Systems

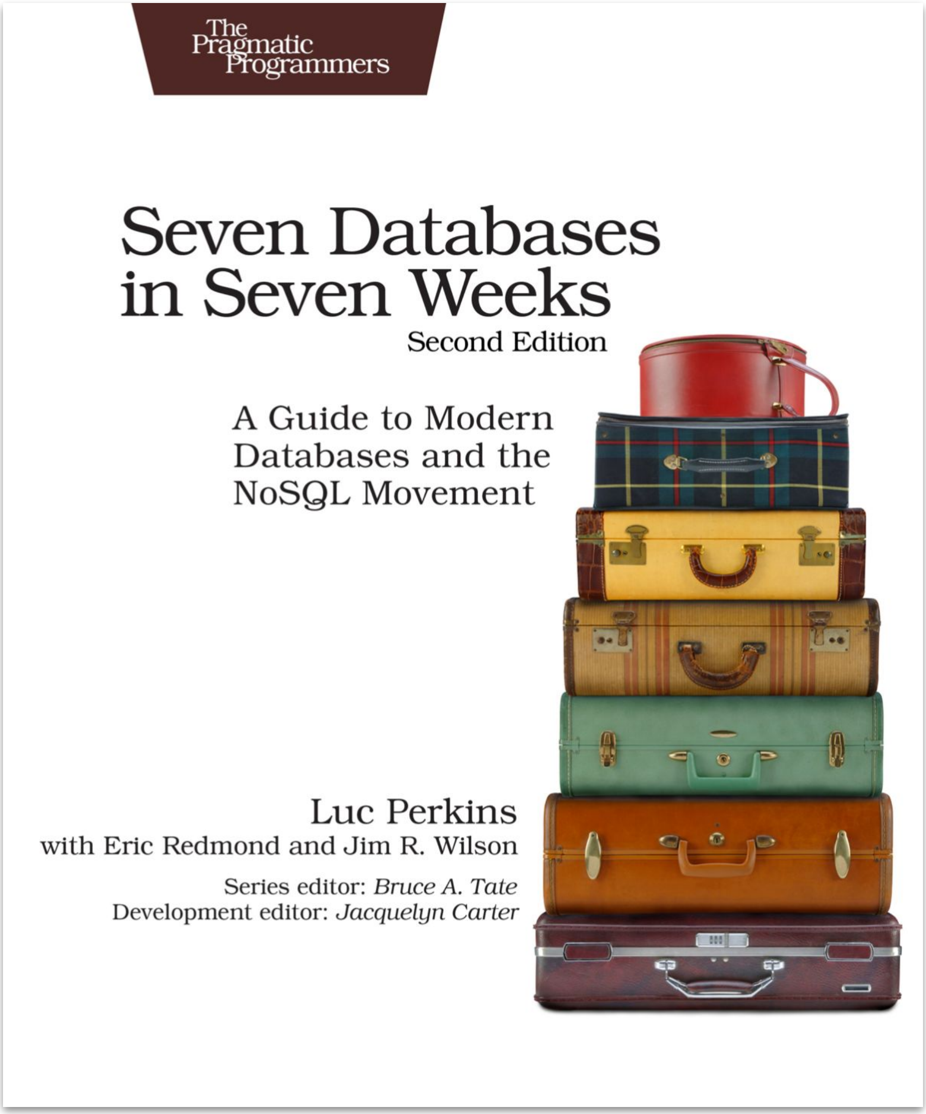
NoSQL Stores

NoSQL Taxonomy

(Apache) HBase

Resources

- Concepts in the slides
- Tutorial on MongoDB
- Class project uses MongoDB
- 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**
  - Around 2000s NoSQL “movement” started
    - New applications and constraints
    - Unclear if it stood for “No SQL”, “Not Only SQL”
- **Relational vs NoSQL stores implement different trade-offs**
  - Different DBs with different worldviews and trade-offs
  - Schema vs schema-less
  - Rich vs fast query-ability
  - Strong consistency (ACID), weak, eventual consistency
  - APIs (SQL, 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 has expanded**
  - Different use cases and demands
  - IMO Postgres and Mongo cover 99% of use cases
  - Any data scientist / engineer needs to be familiar with both
  - “What 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) Scalability
- For each drawback:
  - What is the **problem**
  - Possible **solutions** within relational DB paradigm and 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
    - Rows
    - Relationships between tables
- **Example of the app-db mismatch**:
  - App stores a simple Python map like:

```
# Store a dictionary from name (string) to tags (list of strings).
tag_dict: Dict[str, List[str]]
```

  - A relational DB needs 3 tables:
    - `(nameId, name)` to store the keys
    - `(tagId, tag)` to store the values
    - `(nameId, tagId)` to map the keys to the values
  - One could denormalize `(name, tag)`

1) App-DB Impedance Mismatch: Solutions

- **Ad-hoc mapping layer**
  - Translate objects and data structures into DB data model
  - Cons: need to write / maintain code
- **Objection-relational mapping (ORM)**
  - Technique for converting automatically data between object code and relational DB
    - E.g., SQLAlchemy for Python and SQL
    - E.g., implement a `Person` object (e.g., name, phone number, addresses) using DB
  - Cons: complex types, polymorphism, inheritance
- **NoSQL approach**
  - No schema
  - Every object can be flat or complex (e.g., nested JSON)
  - Stored objects (aka documents) can be different

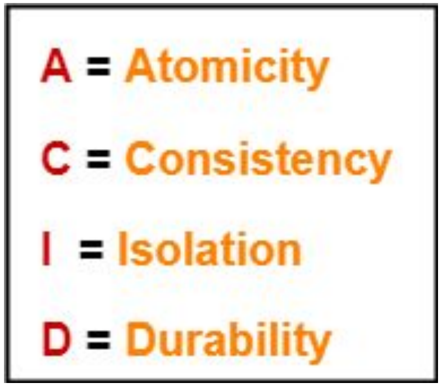
2) Schema Flexibility

- **Problem**
  - Not all applications have data that fits neatly in a schema
  - E.g., data can be nested and dishomogeneous
- **No solution within relational DB**
  - Maybe use a schema so general to accommodate all the possible cases
- **NoSQL approach**
  - E.g., MongoDB does not enforce any schema
  - Pros:
    - Application does not worry about schema when writing data
  - Cons
    - Application deals with variety of schemas when it processes the data

3) Consistency in Relational DBs

- **All systems fail**
  - Application crash
  - Application error (e.g., a scenario that was non implemented, internal error)
  - Hardware failure (e.g., ECC error, disk)
  - Power failure

- **Relational DBs enforce <u>ACID</u> properties**
- **Atomicity**
  - = transactions are all or nothing
  - Either a transaction (which can be composed of multiple statements) succeeds completely or fails
  - Atomicity needs to be guaranteed for any system failure
- **Consistency**
  - = any transaction brings the DB from one valid state to another
  - The "invariants" of the DB (e.g., constraints) must be maintained
- **Isolation**
  - = if transactions are executed *concurrently*, the result is the same as if the transactions were executed *sequentially*
- **Durability**
  - = once a transaction has been committed, the content is preserved for any system failure
  - Just record the data in non-volatile memory



3) Consistency in Distributed DB

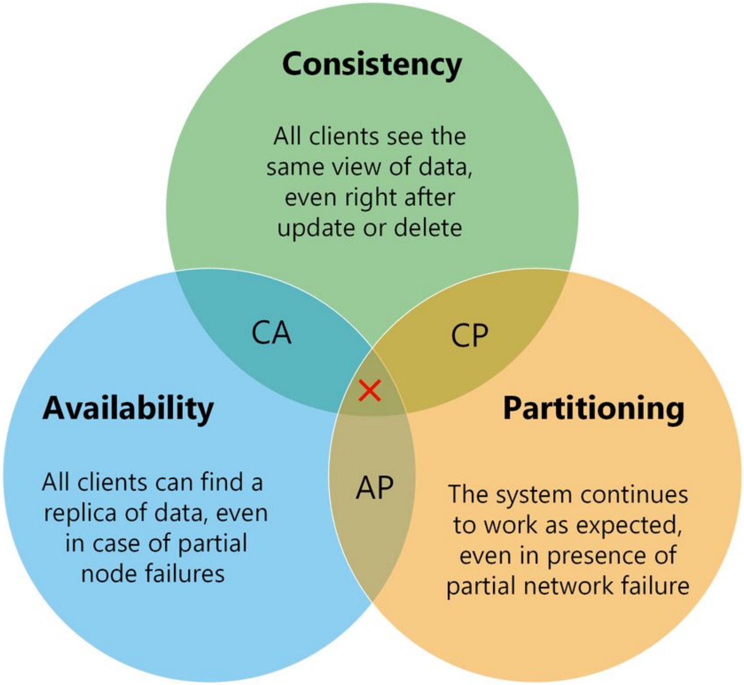
- When data scales up or number of clients increases -> distributed setup to achieve:
  - performance (e.g., transaction per seconds)
  - availability (guarantees a certain up-time)
  - fault-tolerance (can recover from faults)

- **Achieving ACID consistency** is:
  - non-easy in a single DB server setup
  - impossible in a distributed DB server setup due to CAP theorem
    - Even weak consistency is difficult to achieve

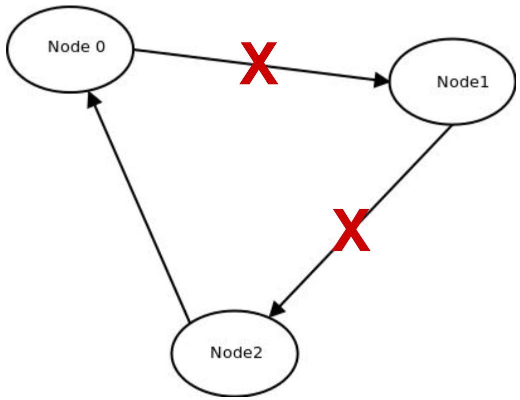
CAP Theorem

- **CAP theorem**: Any *distributed* DB can have *at most two* of the following three properties
  - **Consistent**: writes are atomic and subsequent reads retrieve the new value
  - **Available**: a value is returned as long as a single server is running
  - **Partition tolerant**: the system still works even if communication is temporary lost (i.e., the network is partitioned)
- Originally a conjecture (Eric Brewer), but made formal later (Gilbert, Lynch, 2002)

- **CAP corollary**: Network partitions cannot be prevented in large-scale distributed system, so either sacrifice:
  - Availability (i.e., go down): e.g., banking system
  - Consistency (i.e., different views of the system): e.g., social network
- Minimize probability of failures using redundancy and fault-tolerance



CAP Theorem: Intuition



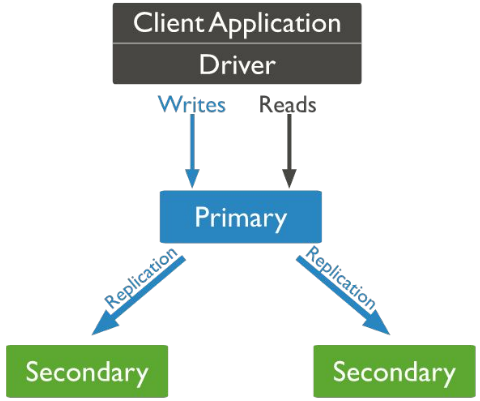
- **Example of network partition**
- Imagine there are 2 DB replicas (Node1, Node2)
- A network partition happens
  - DB servers (Node1, Node2) can't communicate with each other
  - Users (Node 0) can access only one of them (Node2)
  - Reads: the user can access the data of the server in the same partition
  - Writes: data can't be updated since multiple users might be updating the data at the same data, leading to inconsistency

- **CAP theorem**: one needs to sacrifice consistency or availability
- Available but not consistent
  - Let updates happen on the accessible replica at cost of inconsistency
  - Sometimes inconsistency is fine (e.g., social networking)
- Consistent but not available
  - Stop the service (no availability) to avoid inconsistency
  - Sometimes inconsistency is not acceptable (e.g., a banking system)

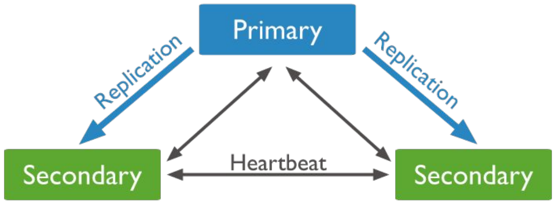
Replication Schemes

- **Replication schemes**: how to organize multiple servers implementing a distributed DB

- **Primary-secondary replication**
  - Aka "master-slave replication"
  - Application only communicate with primary
  - Replicas cannot update local data, but require primary node to perform update
  - Single-point of failure
- **Update-anywhere replication**
  - Aka "multi-master replication"
  - Every replica can update a data item, which is then propagated (synchronously or asynchronously) to the other replicas
- **Quorum**
  - Let N be the total number of replicas
  - When writing, we make sure to write to W replicas
  - When reading, we read from R replicas and pick the latest update (using timestamps)



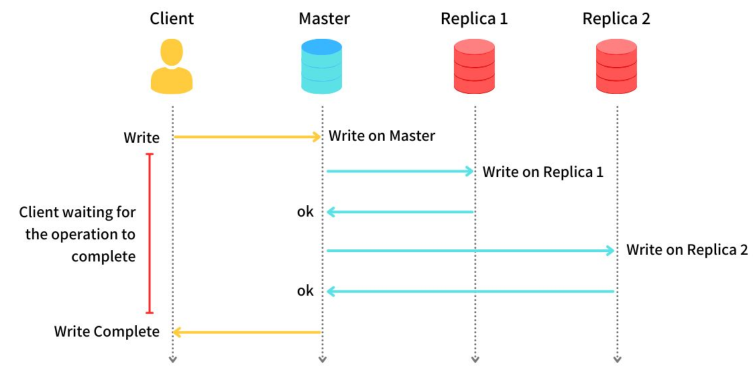
Primary-secondary replication



Update-anywhere replication

Synchronous Replication

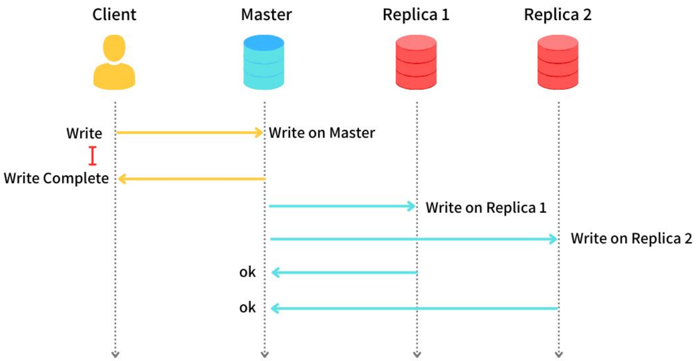
- **Synchronous replication**: updates are propagated to other replicas as part of a single transaction
- Implementations
  - **2-Phase Commit (2PC)**: original proposal for doing this
    - Single point of failure
    - Can't handle primary server failure
  - **Paxos**: more widely used today
    - Doesn't require a primary
    - More fault tolerant
  - Both solutions are complex / expensive
- CAP theorem: still only two among Consistency, Availability, fail in case of network Partition
  - Many systems use relaxed / loose consistency models



Synchronous Replication

Asynchronous Replication

- **Asynchronous replication**
  - The primary node propagates updates to replicas
  - The transaction is completed before replicas are updated (even if there are failures)
  - Commits are quick at cost of consistency
- **Eventual consistency**
  - Popularized by AWS DynamoDB
  - Cannot provide guarantees about what different clients will see, in which order they will see updates, etc.
  - Guarantees provided only on the eventual outcome
  - “Eventual” can mean after the server or network is fixed
- **“Freshness” property**
  - Under asynchronous updates, a read from a replica may not get the latest version of a data item
  - User can request a version with a certain "freshness"
    - E.g., “data from not more than 10 minutes ago”
    - E.g., it's ok to show price for an airplane ticket that is few minutes old
  - Replicas version their data with timestamps
  - If local replica has fresh data, uses it, otherwise send request to primary node



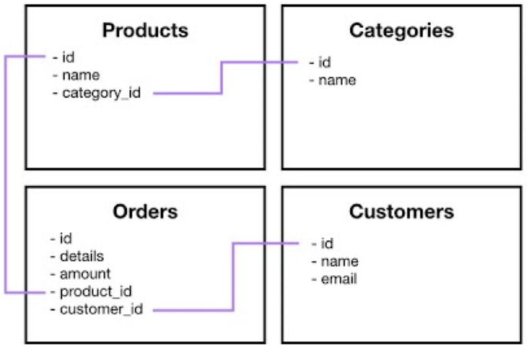
Asynchronous Replication

4) Scalability Issues with RDMS

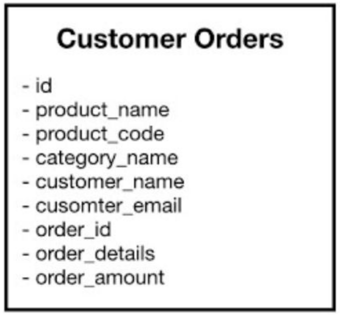
- Sources of relational DB scalability issues
- **Locking data**
  - The DB engine needs to lock rows and tables to ensure ACID properties
  - When DB locked:
    - higher latency ->
    - less updates per second ->
    - slower application
- **Scaling out**
  - Requires replicating data over multiple servers
  - Application becomes even slower
    - Network delays
    - To enforce DB consistency, locks are applied across networks
    - Overhead of replica consistency (2PC, Paxos)

Scalability Issues with RDMS: Solutions

- **Table denormalization**
  - = approach used to increase relational DB performance by adding redundant data
  - Pros:
    - Reads become faster
      - Lock only one table, instead of multiple ones, reducing resource contention
      - No need for joins
  - Cons:
    - Writes become slower
      - There is more data to update
      - E.g., to update a *category name*, need to do a scan
    - If we join the tables, we lose relations between tables (this is the main reason of using a relational DB!)
- **Relax consistency**
  - Give up on part of ACID
  - Make definition of consistency weaker (e.g., eventual consistency)
- **NoSQL**



Normalized data



Denormalized data

NoSQL Stores

- Geared toward the use case of **large-scale web applications**
  - MongoDB started at DoubleClick working in AdTech
  - Need real-time access with a few ms latencies
    - E.g., Facebook, 4ms for reads to get snappy UI
  - Don't need ACID properties
- Solve problems with using relational databases
  - 1) Application-DB impedance mismatch
  - 2) Schema flexibility
  - 3) Consistency in distributed set-up
  - 4) Scalability
- If you want to really scale, you must give up something
  - Give up consistency
  - Give up joins
    - Most NoSQL stores don't allow server-side joins
    - Instead require data to be denormalized and duplicated
  - Only allow restricted transactions
    - Most NoSQL stores will only allow one object transactions
    - E.g., one document / key

Relational DB vs MongoDB

How MongoDB solves the four RDBM problems

1) **Application-DB impedance mismatch**
   - Store data as nested objects

2) **Schema flexibility**
   - No schema, no tables, no rows, no columns, no relationships between tables

3) **Consistency in replicated set-up**
   - Application decides consistency level
     - Synchronous: wait until primary and secondary servers are updated
     - Quorum synchronous: wait until the majority of secondary servers are updated
     - Asynchronous, eventual: wait until only the primary is updated
     - "Fire and forget": not even wait until the primary persisted the data

4) **Scalability**
   - Updating data means locking only one document, and not entire collection
   - Sharding: use more machines to do collectively do more work

UMD DATA605 - Big Data Systems

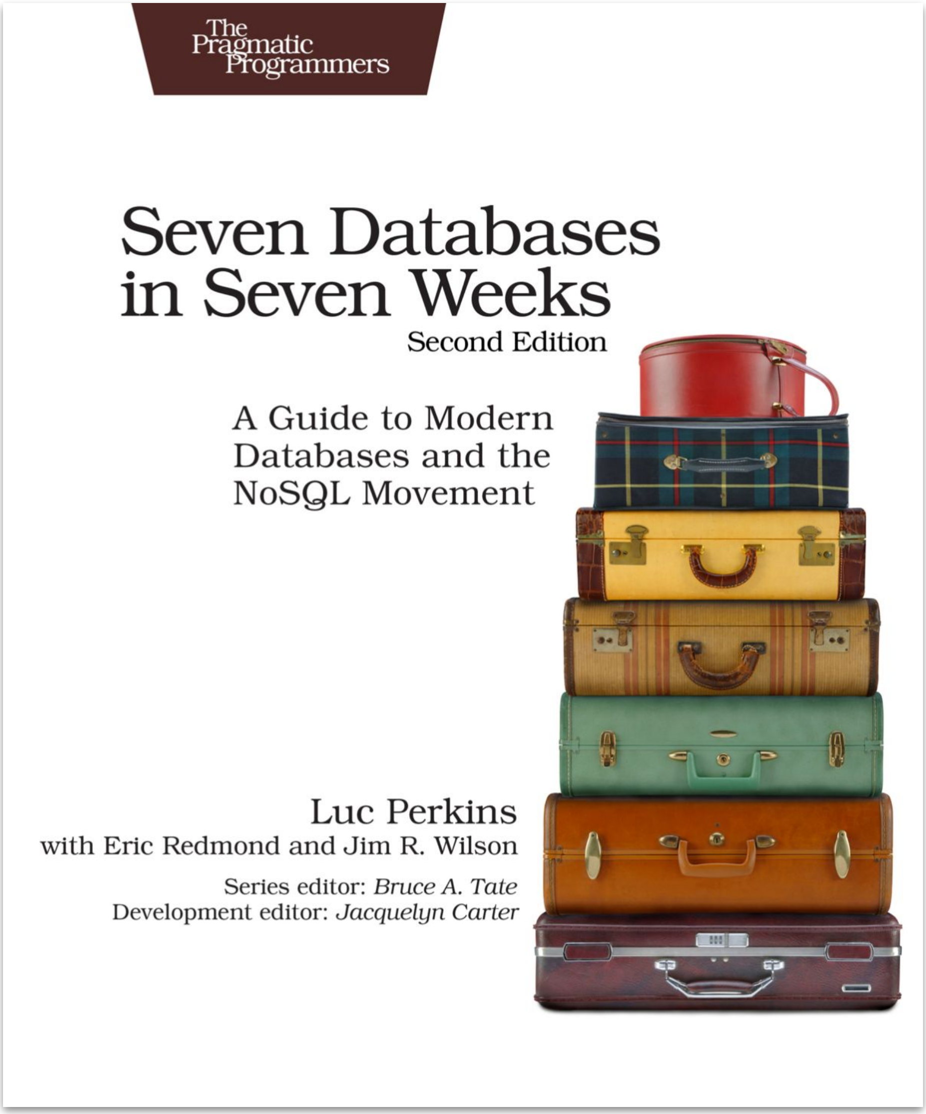
NoSQL Stores

NoSQL Taxonomy

(Apache) HBase

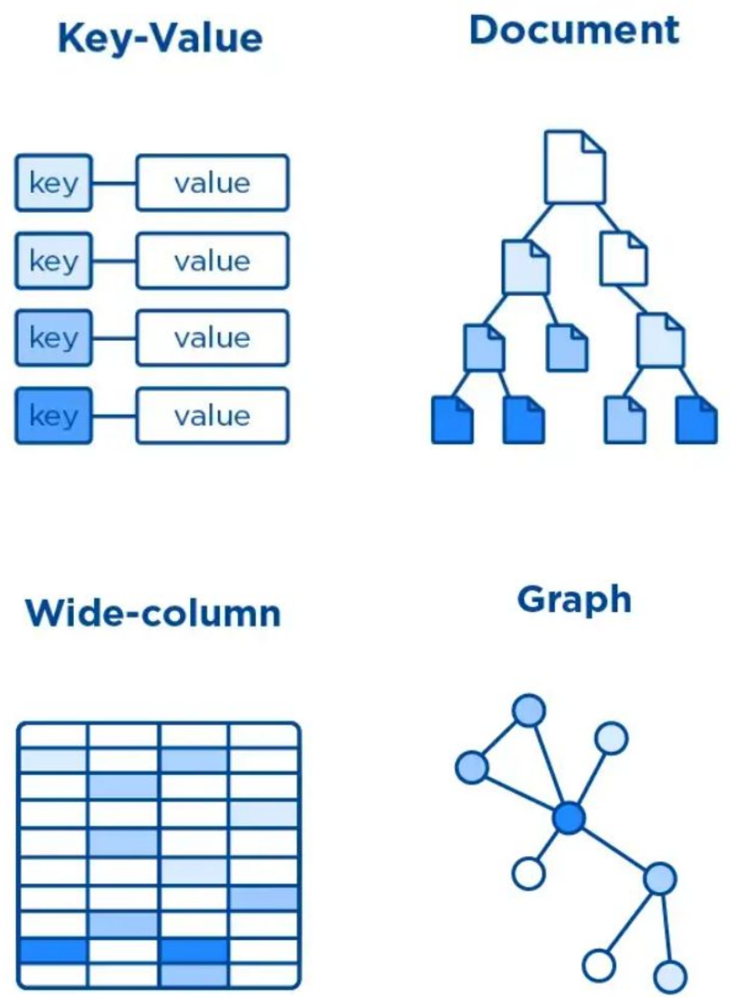
Resources

- Concepts in the slides
- Silbershatz Chap 23.6
- Mastery:
  - Seven Databases in Seven Weeks, 2e



DB Taxonomy

- **At least five DB genres**
  - Relational
  - Key-value
  - Document
  - Columnar
  - Graph
- **Criteria to differentiate DBs**
  - Data model
  - Trade-off with respect to CAP theorem
  - Querying capability
  - Replication scheme

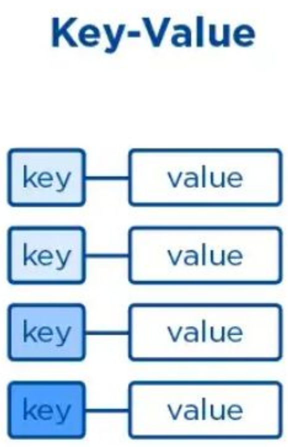


Relational DB

- E.g., *Postgres*, MySQL, Oracle, SQLite
- **Data model**
  - Based on set-theory and relational algebra
  - Data as two dimensional tables with rows and columns
  - Many attribute types (e.g., numeric, strings, dates, arrays, blobs)
  - Attribute types are strictly enforced
  - SQL query language
  - ACID consistency
- **Application**
  - Any relational tabular data
- **Good for**
  - Useful when layout of data is known, but not the data access pattern
  - Complexity upfront (for schema) to achieve query flexibility
  - Used when data is regular
- **Not so good for**
  - When data is hierarchical (not a nice row in one or more tables)
  - When data structure is variable (record-to-record variation)

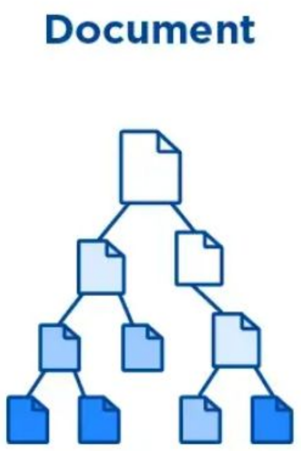
Key-Value Store

- E.g., Redis, DynamoDB, *Git*, AWS S3, filesystem
- **Data model**
  - Map simple keys (e.g., strings) to more complex values (e.g., it can be anything, binary blob)
  - Support get, put, and delete operations on a primary key
- **Application**
  - Caching data
  - Store users' session data in a web application
- **Good for**
  - Useful when data is not "related" (e.g., no joins)
  - Lookups are fast
  - Easy to scale horizontally using partitioning scheme
- **Not so good for**
  - Lacking secondary indexes and scanning capabilities
  - Not great if data queries are needed



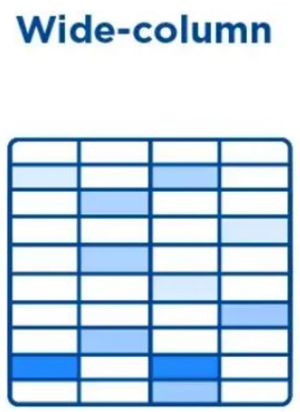
Document Store

- E.g., *MongoDB*, CouchDB
- **Data model**
  - Each document has a unique ID (e.g., hash)
  - Allow for any number of fields per document, even nested
    - E.g., JSON, XML value
  - Since documents are not related, it's easy to shard and replicate over distributed servers
- **Application**
  - Any semi-structured data
- **Good for**
  - When you don't know how your data will look like
  - Map well to OOP models (less impedance mismatch between application and DB)
- **Not so good for**
  - Complex join queries on normalized data not possible
  - Denormalized form is the norm



Columnar Store

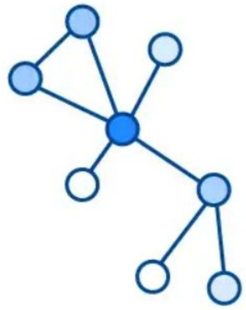
- E.g., *HBase*, Cassandra, *Parquet*
- Data model
  - Data is stored by columns, instead of rows like in relational DBs
  - Share similarities with both key-value and relational DBs
    - Keys are used to query values, like key-value stores
    - Values are groups of zero or more columns, like relational stores
- Application
  - E.g., storing web-pages
- Good for
  - Horizontal scalability
  - Enable compression and versioning
  - Tables can be sparse without extra storage cost
  - Columns are inexpensive to add
- Not so good for
  - You need to design the schema based on how you plan to query the data
  - No native joins, applications need to handle join



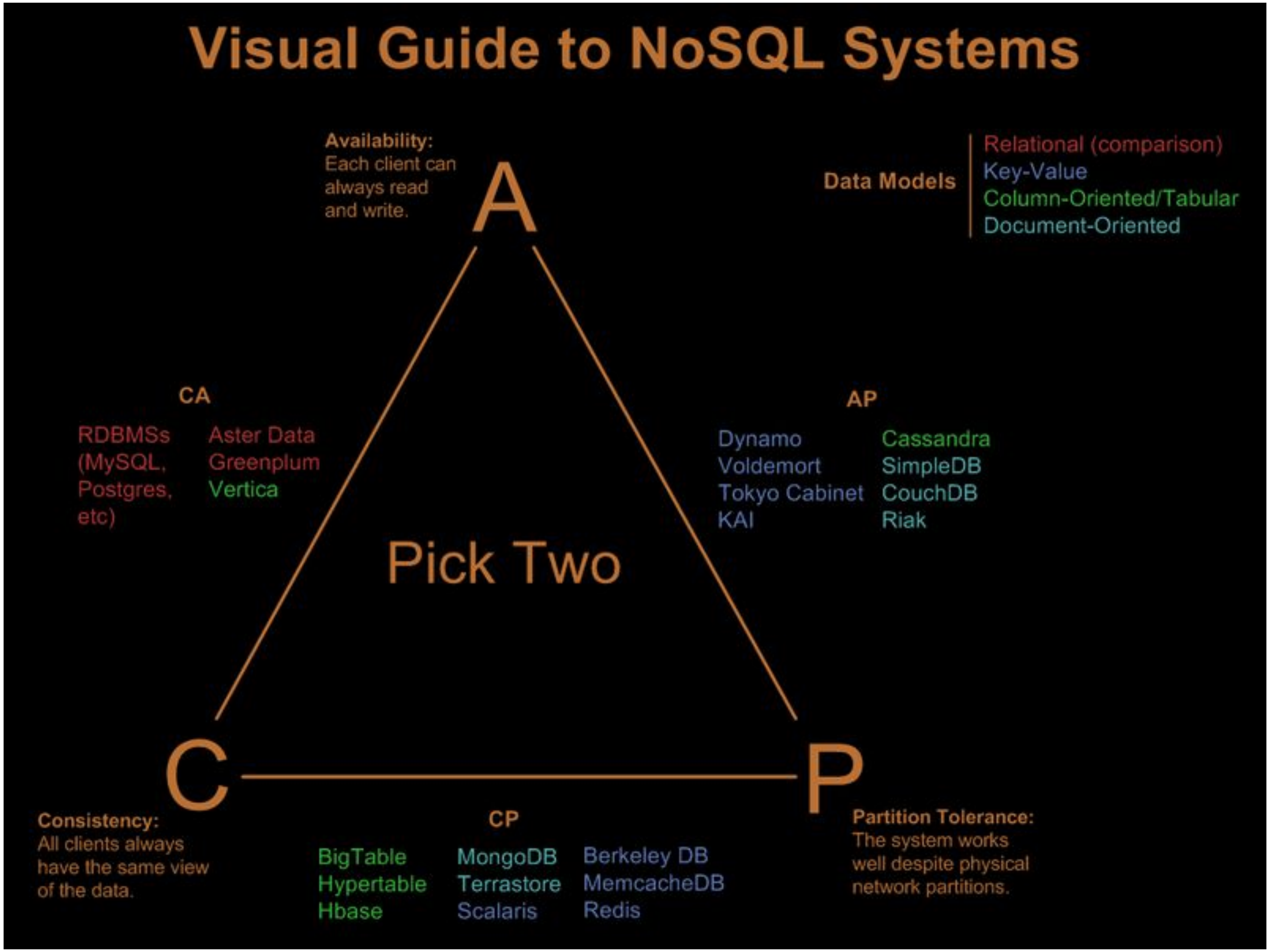
Graph DB

- E.g., *Neo4J*, GraphX
- **Data model**
  - Highly interconnected data, storing nodes and relationships between nodes
  - Both nodes and edges have properties (i.e,. key-value pairs)
  - Queries involve traversing nodes and relationships to find relevant data
- **Applications**
  - Social data
  - Recommendation engines
  - Geographical data
- **Good for**
  - Perfect for "networked data", which is difficult to model with relational model
  - Good match for OO systems
- **Not so good for**
  - Don't scale well, since it's difficult to partition graph on different nodes
    - Store the graph in the graph DB and the relations in a key-value store

Graph



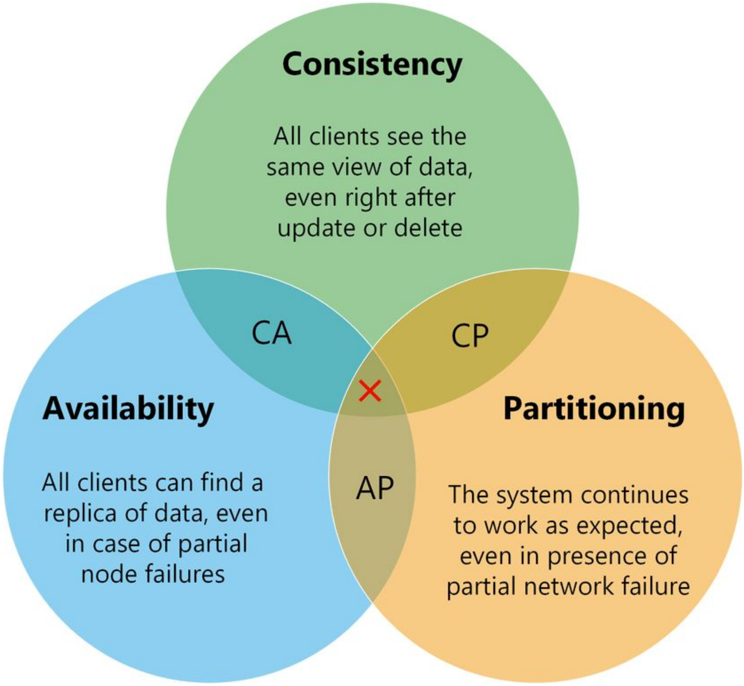
Taxonomy by CAP



From http://blog.nahurst.com/visual-guide-to-nosql-systems

Taxonomy by CAP

- **CA (Consistent, Available) systems**
  - Have trouble with partitions and typically deal with it with replication
  - E.g.,
    - Traditional RDBMSs like PostgreSQL, MySQL
- **CP (Consistent, Partition-Tolerant) systems**
  - Have trouble with availability while keeping data consistent across partitioned nodes
  - E.g.,
    - BigTable (column-oriented/tabular)
    - HBase (column-oriented/tabular)
    - MongoDB (document-oriented)
    - Redis (key-value)
    - MemcacheDB (key-value)
    - Berkeley DB (key-value)
- **AP (Available, Partition-Tolerant) systems**
  - Achieve "eventual consistency" through replication and verification
  - E.g.,
    - Dynamo (key-value)
    - Cassandra (column-oriented/tabular)
    - CouchDB (document-oriented)



UMD DATA605 - Big Data Systems

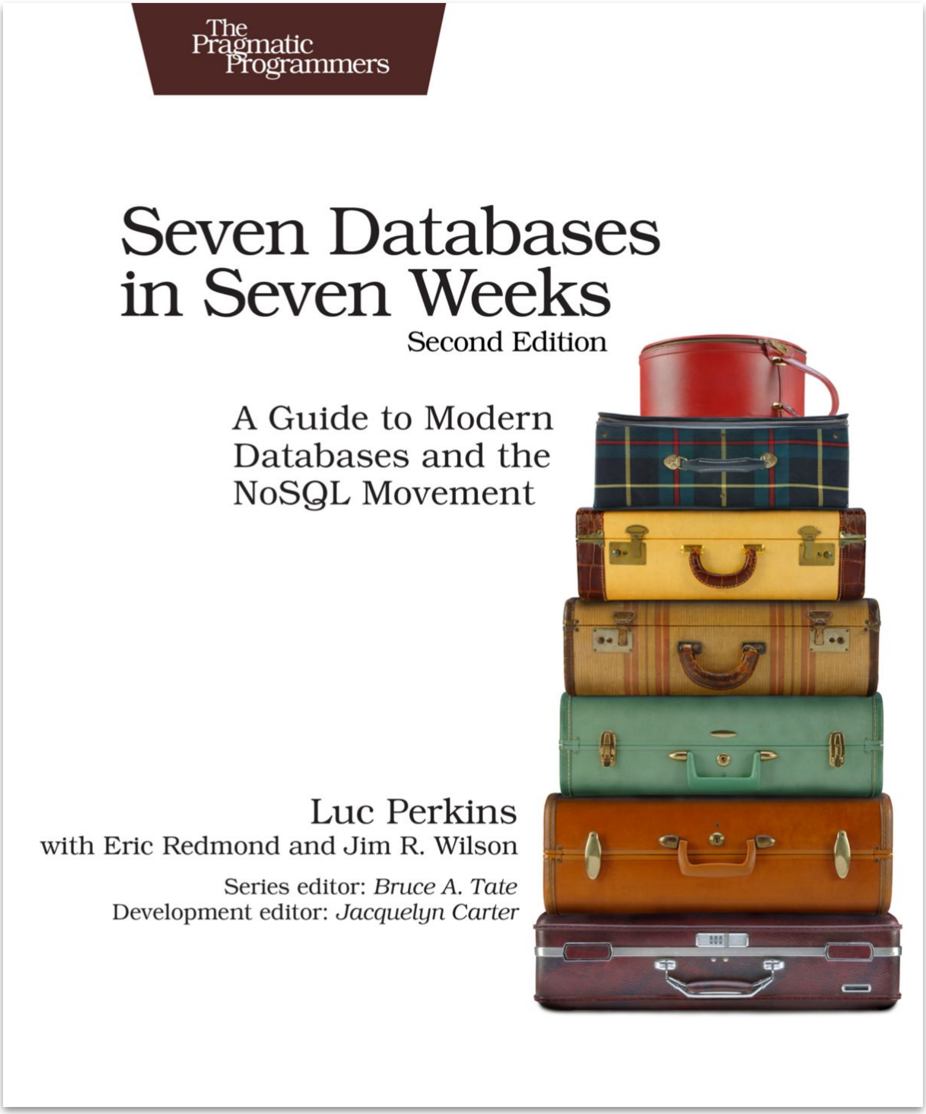
NoSQL Stores

NoSQL Taxonomy

(Apache) HBase

Resources

- Content in slides
- Web
  - [2006, BigTable paper]
  - [https://hbase.apache.org/]
  - [https://github.com/apache/hbase]
- Good overview:
  - [Seven Databases in Seven Weeks, 2e]



(Apache) HBase

- HBase = **H**adoop Data**Base**
  - Support very large tables on clusters of commodity hardware
  - Column oriented DB
  - Part of Apache Hadoop ecosystem
  - Use Hadoop filesystem (HDFS)
    - HDFS modeled after Google File System (GFS)
    - HBase based on Google BigTable
    - Google BigTable runs on GFS, HBase runs on HDFS
  - Used at Google, Airbnb, eBay
- **When to use HBase**
  - For large DBs (e.g., at least many 100 GBs or TBs)
  - When having at least 5 nodes in production
- **Applications**
  - Large-scale online analytics
  - Heavy-duty logging
  - Search systems (e.g., Internet search)
  - Facebook Messages (based on Cassandra)
  - Twitter metrics monitoring

HBase: Features

- Data versioning
- Data compression
- Garbage collection (for expired data)
- In-memory tables
- Atomicity (at row level)
- Strong consistency guarantees
- Fault tolerant (for machines and network)
  - Write-ahead logging
    - Write data to an in-memory log before it's written to disk
  - Distributed configuration
    - Nodes can rely on each other rather than on a centralized source

From HDFS to HBase

- **Different types of workloads for DB backends**
  - OLAP (**O**n-**L**ine **A**nalytical **P**rocessing)
    - Read continuously large amount of data and process it
    - E.g., analyze item purchases over time
  - OLTP (**O**n-**L**ine **T**ransactional **P**rocessing)
    - Read and write individual data items in a large table
    - E.g., update inventory and price as orders come in
- **Hadoop FileSystem (HDFS) supports OLAP workloads**
  - Provide a filesystem consisting of arbitrarily large files
  - Data should be read sequentially, end-to-end
  - Rarely updated
- **HBase supports OLTP interactions**
  - Built on top of HDFS
  - Use additional storage and memory to organize the tables
  - Write tables back to HDFS as needed

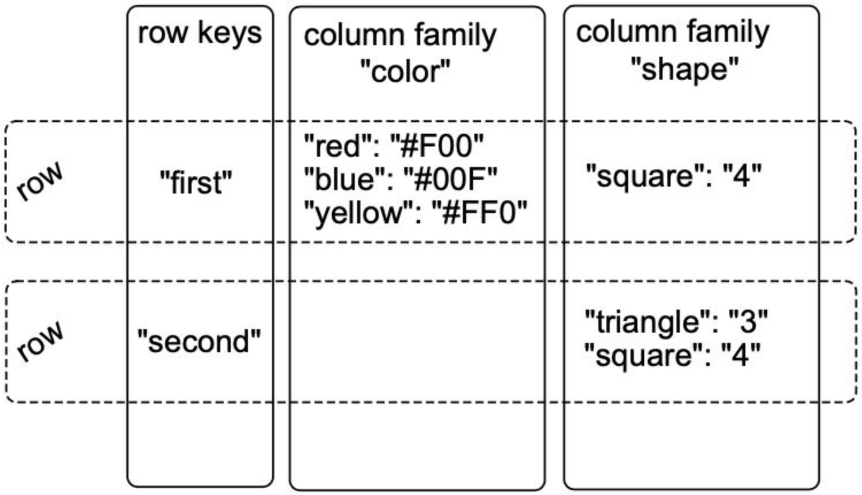
HBase Data Model

- **Warning**: HBase uses names similar to relational DB concepts, but with different meanings
- A **database** consists of multiple tables
- Each **table** consists of multiple rows, sorted by row key
- Each row contains a **row key** and one or more column families
- Each **column family**
  - can contain multiple **columns** (family:column)
  - is defined when the table is created
- A **cell**
  - is uniquely identified by (table, row, family:column)
  - contains **metadata** (e.g., timestamp) and an uninterpreted array of bytes (blob)
- Versioning
  - New values don't overwrite the old ones
  - \`put()\` and \`get()\` allow to specify a timestamp (otherwise uses current time)

```
# HBase Database: from table name to Table.
Database = Dict[str, Table]

# HBase Table.
table: Table = {
  # Row key
  'row1': {
    # (column family, column) -> value
    'cf1:col1': 'value1',
    'cf1:col2': 'value2',
    'cf2:col1': 'value3'
  },
  'row2': {
    ... # More row data
  }
}

database = {'table1': table}

# Querying data.
(value, metadata) = \
     table['row1']['cf1:col1']
```

Example 1

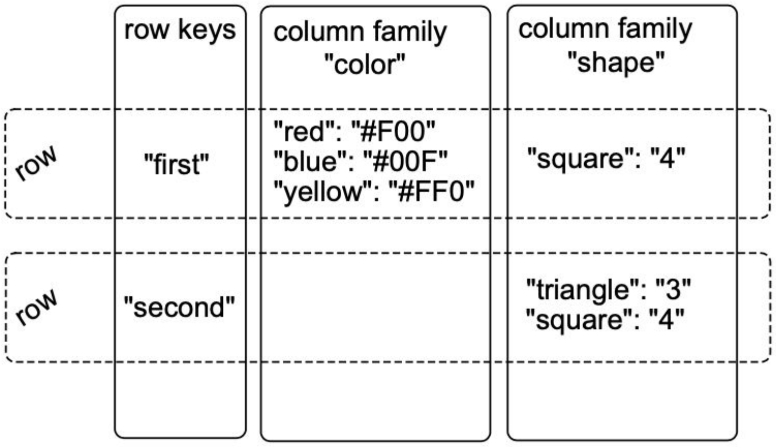
- Table with:
  - 2 column families (“color” and “shape”)
  - 2 rows (“first” and “second”)
- The row “first” has:
  - 3 columns in the column family “color” (“red”, “blue”, “yellow”)
  - 1 column in the column family “shape”
- The row “second” has:
  - no columns in “color”
  - 2 columns in the column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




```
table = {
  'first': {
    # (column family, column) -> value
    'color': {'red': '#F00',
              'blue': '#00F',
              'yellow': '#FF0'}
    'shape': {'square': 4}
  },
  'second': {
    'shape': {'triangle': 3,
              'square': 4}
  }
}
```

Why all this convoluted stuff?

- **Intuition**: 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 string and an article text

HBase data model

- Table name -> `wikipedia`
- Row -> entire wiki page
- Row keys -> wiki identifier (e.g., title, URL, path)
- Column family -> `text`
- Column -> not defined, ‘’ (empty)
- Cell value -> article text

Add data

- Columns don't need to be predefined when creating a table
- The column is defined as `text`

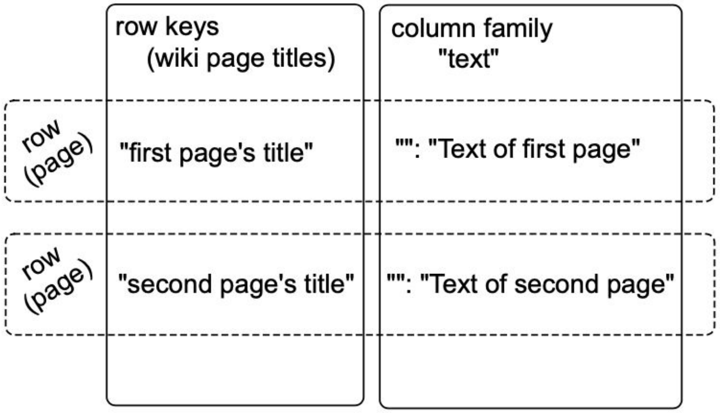
```
> put 'wikipedia', 'Home', 'text', 'Welcome!'
```

Query data

- Specify the table name, the row key, and optionally a list of columns

```
> get 'wikipedia', 'Home', 'text'
text: timestamp=1295774833226, value=Welcome!
```

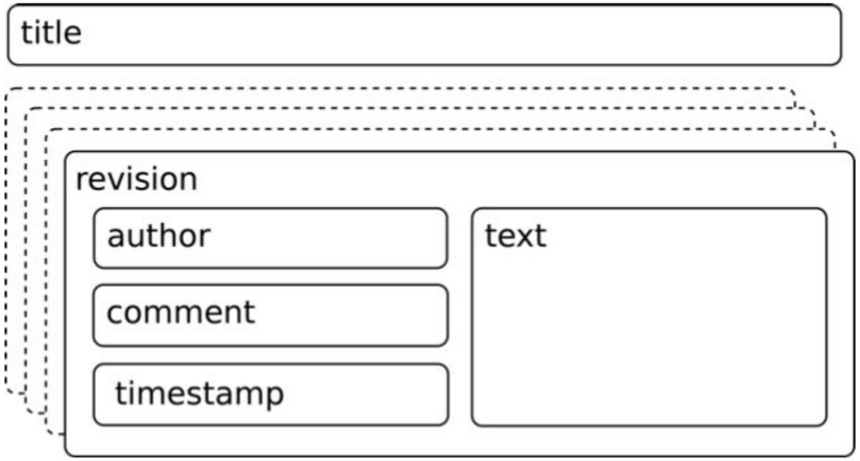
- HBase returns the timestamp (ms since the epoch 01-01-1970 UTC)



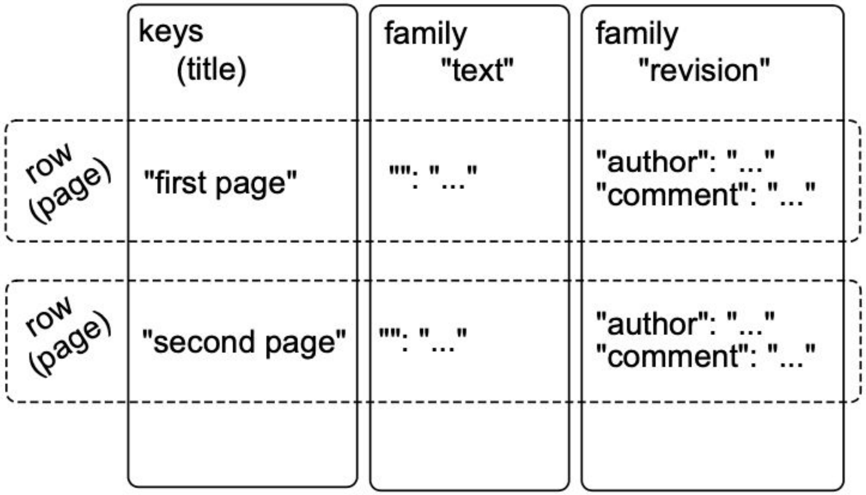

```
wikipedia_table = {
  # wiki id.
  'Home': {
    # Column family, column -> value
    'text': 'Welcome to the wiki!',
  },
  'Welcome page': {
    … # More row data
  }
}

Database = Dict[str, Table]
database: Database = {'wikipedia':
wiki_table}


(queried_value, metadata) = \
     wiki_table['Home']['text']
```

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
    - 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.org

- 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”}
     },
     102: {
          Timestamp: T593;
          Name: {First=“Marilyn”, Last=“Tollerud”},
          Home: {Phone=“555-1213”},
          Office: {Phone=“666-1213”}
     },
     …
}
```

Nested Data Representation

	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.org

```
GET People: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”}
    }

GET People:101:Name
    {First=“Florian”, Middle=“Garfield”, Last=“Krepsbach”}

GET People:101:Name:First
    “Florian”
```

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 (expensive)

- **Why 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

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
    - Check the timestamp to be sure!
  – A SCAN
    - must include all data written prior to the scan
    - may include updates since it started
- **Isolation**
  – Not guaranteed outside a single 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
- **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 using that to set some bits to 1
  - To test if we have seen a blob, compute the bits and check
    - If all bits are 0s, then for sure we didn't see it
    - If all bits are 1s, then we might have seen that blob
- **Bloom filter complexity**
  - Use a constant amount of space
  - Have false positives
  - O(1)

Write-Ahead Log (WAL)

- HBase uses WAL
  - A technique to provide atomicity and durability, protecting against node failures
  - Equivalent to journaling in file system

- **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 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, 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

HBase Implementation

- HBase is backed by HDFS
  - Store each table (e.g., Wikipedia) in one file
  - "One file" means one gigantic file stored in HDFS
  - Not to worry about the details of how the file is split into blocks
- 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
  - Idea 2: One file + WAL
    - Better, but doesn't scale to large data
  - Idea 3: One file per column family + WAL
    - Getting better!
  - 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

File “People”

```
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”}
     },
     …
}
```

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

Variable-Length Data

`SQL Table: People(ID: Integer, FirstName: CHAR[20], LastName: Char[20], Phone: CHAR[8])`

`UPDATE People SET Phone="555-3434" WHERE ID=403;`

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

- Each row is exactly 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

`HBase Table: People(ID, Name, Home, Office)`

`PUT People, 403, Home:Phone, 555-3434`

```
{
     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"}
     },
...
```

Need to use pointers

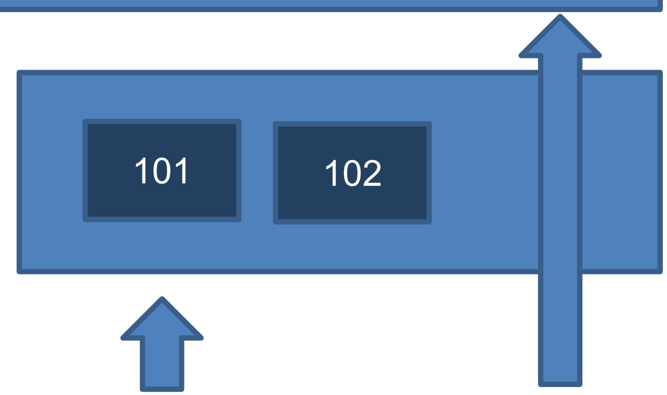
Idea 2: One Table + WAL

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"}
     },
     …
```

WAL for Table People

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



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

GET People:101

GET People:103

PUT People:101:Office:Phone = "555-3434"

Idea 2 Requires Periodic Table Update

Table for People on Disk (Old)

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" }}, . . .

WAL for Table People:

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

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

Table for People on Disk (New)

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"** }, . . .

Idea 3: Partition by Column Family

- Same scheme as before but split by column family

Tables for People on Disk (Old)

Data for Column Family Name

Data for Column Family Home

Data for Column Family Office

WAL for Table People

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

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

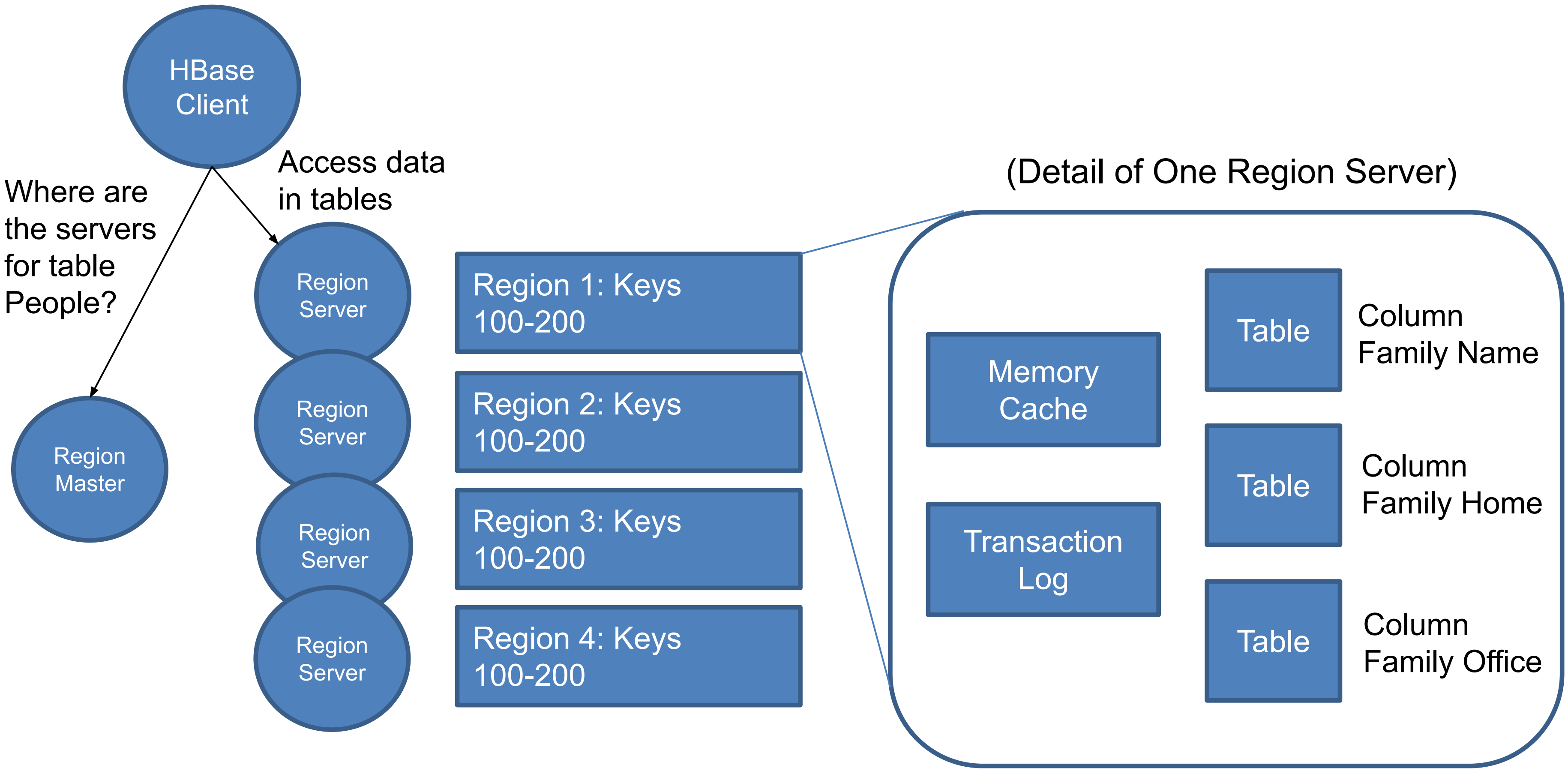
Tables for People on Disk (New)

Data for Column Family Name

Data for Column Family Home (Changed)

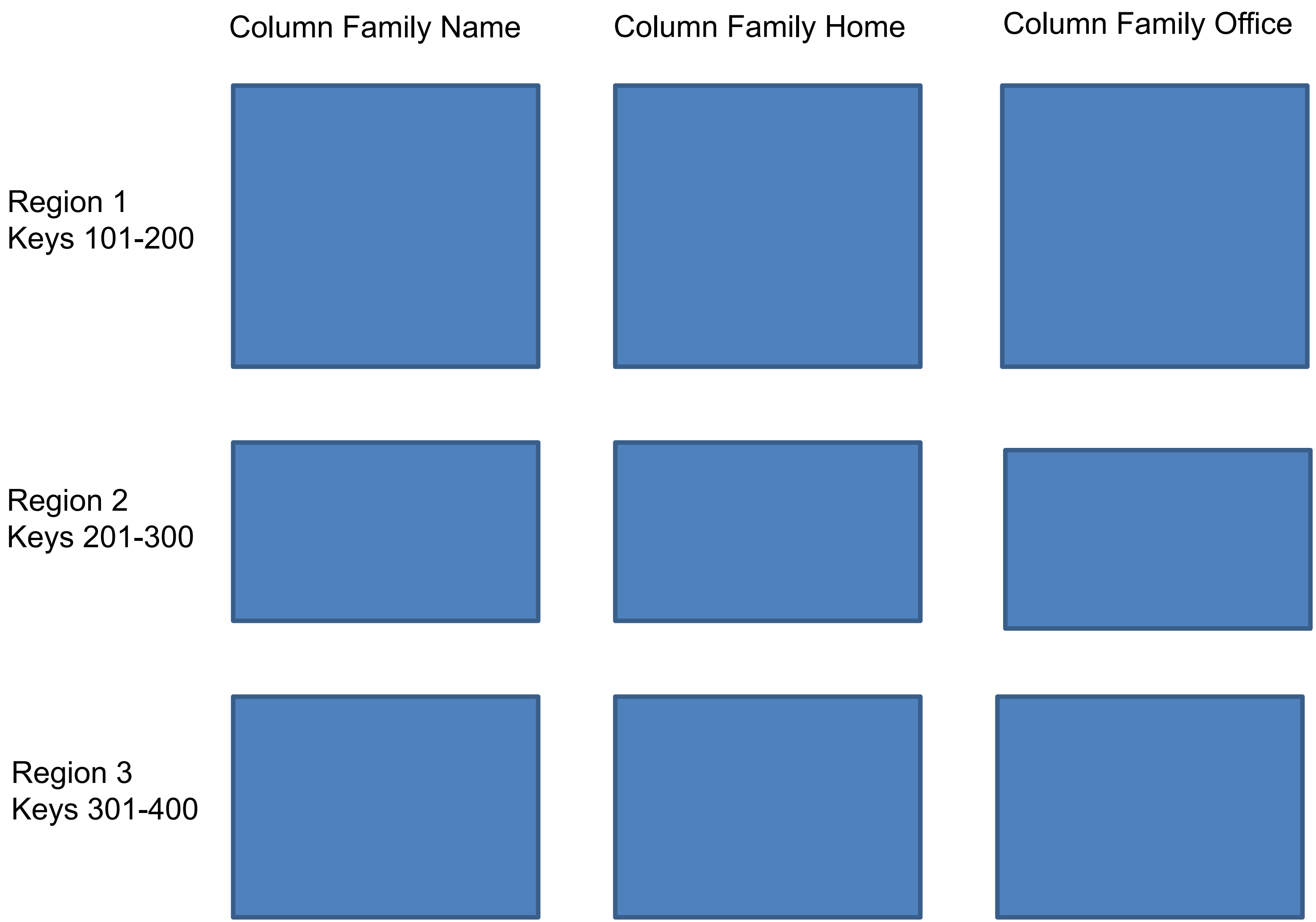
Data for Column Family Office (Changed)

Idea 4: Split Into Regions



Final HBase Data Layout

Table People



Backup Slides

BASE Consistency

- Basically Available
- Soft state
  - Each replica can have a different state after partitioning
- Eventually consistent
  - Once the partitioning is resolved, all replicas will become eventually consistent
  - E.g., merge updates in a meaningful way (e.g., by timestamp)