The Hadoop Stack, Part 2
Introduction to HBase

CSE 40822 – Cloud Computing – Spring 2016

Prof. Douglas Thain
University of Notre Dame

Three Case Studies

- Workflow: Pig Latin
  - A dataflow language and execution system that provides an SQL-like way of composing workflows of multiple Map-Reduce jobs.
- Storage: HBase
  - A NoSQl storage system that brings a higher degree of structure to the flat-file nature of HDFS.
- Execution: Spark
  - An in-memory data analysis system that can use Hadoop as a persistence layer, enabling algorithms that are not easily expressed in Map-Reduce.

References

- Fay Chang et al, Bigtable: A Distributed Storage System for Structured Data, OSDI 2006
  - http://research.google.com/archive/bigtable.html
- Introduction to Hbase Schema Design, Amandeep Khurana, Login; Magazine, 2012.
  - https://www.usenix.org/publications/login/october-2012-volume-37-number-5/introduction-hbase-schema-design
- Apache HBase Documentation
  - http://hbase.apache.org

From HDFS to HBase

- HDFS provides us with a filesystem consisting of arbitrarily large files that can only be written once and are should be read sequentially, end to end. This works fine for sequential OLAP queries like Pig.
- OLTP workloads want to read and write individual cells in a large table. (e.g. update inventory and price as orders come in.)
- HBase implements OLTP interactions on top of HDFS by using additional storage and memory to organize the tables, and writing them back to HDFS as needed.
- HBase is an open source implementation of the BigTable design published by Google.

HBase Data Model

- 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 is defined when the table is created.
- Column families can contain multiple columns. (family:column)
- A cell is uniquely identified by (table,row,family:column).
- A cell contains an uninterpreted array of bytes and a timestamp.

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@ wels.org

Data in Tabular Form

	Name			Home		Office		Social
Key	First	Middle	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@wels.org	

New columns can be added at runtime.

Column families cannot be added at runtime.

Don’t be fooled by the picture:
Hbase is really a **sparse** table.

Nested Data Representation

```
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

GET People:101

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

People:101:Name: {First="Florian", Middle="Garfield", Last="Krepsbach"}

GET People:101:Name:First

People:101:Name:First="Florian"

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

Consistency Model

- Atomicity: Entire rows are updated atomically or not at all.
- 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, and may include updates since it started.
- Isolation: Not guaranteed outside a single row.
- Durability: All successful writes have been made durable on disk.

The Implementation

- I'll give you the idea in several steps:
  - Idea 1: Put an entire table in one file. (It doesn't work.)
  - Idea 2: Log + one file. (Better, but doesn't scale to large data.)
  - Idea 3: Log + one file per column family. (Getting better!)
  - Idea 4: Partition table into regions by key.

- Keep in mind that "one file" means one **gigantic** file stored in HDFS. We don't have to worry about the details of how the file is split into blocks.

Idea 1: Put the Table in a Single File

File “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”}},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
  - SCAN
  - GET
  - PUT

Variable-Length Data is **Fundamentally** Hard!

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, just 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"}},102: {Timestamp: T593;Name: { First="Marilyn", Last="Tollerud"},Home: { Phone="555-1213" },Office: { Phone="666-1213" }}, . . .

?

Fun reading on the topic:
http://www.joelonsoftware.com/articles/fog0000000319.html

Idea 2: One Tablet + Transaction Log

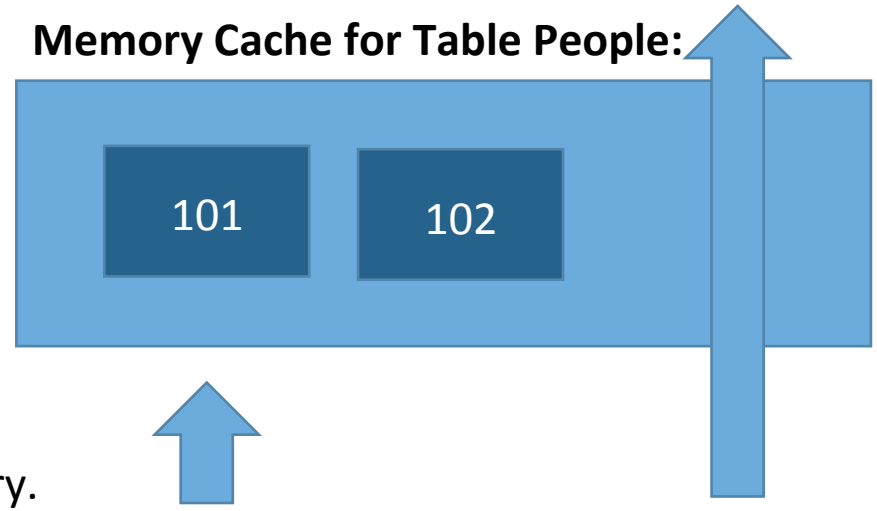
Table for People

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"}},102: {Timestamp: T593;Name: { First="Marilyn", Last="Tollerud"},Home: { Phone="555-1213" },Office: { Phone="666-1213" }}, . . .

Transaction Log for Table People:

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

Memory Cache for Table People:



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

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

GET People:101

GET People:103

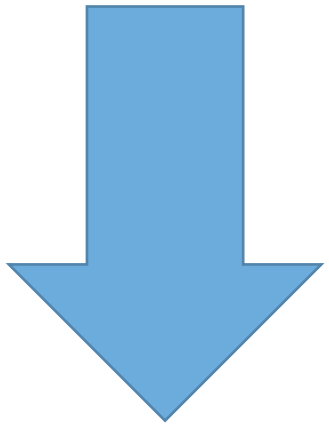
Idea 2 Requires Periodic Log Compression

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" }}, . . .

Transaction Log 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, and 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

Tablets for People on Disk: (Old)

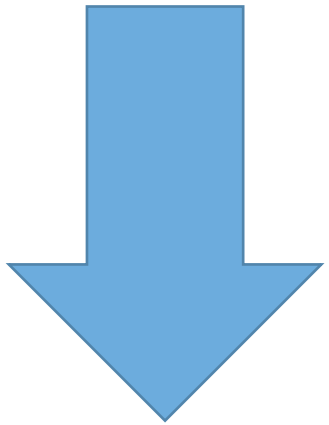
Data for
Column Family
Name

Data for
Column Family
Home

Data for
Column Family
Office

Transaction Log 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, and start over.

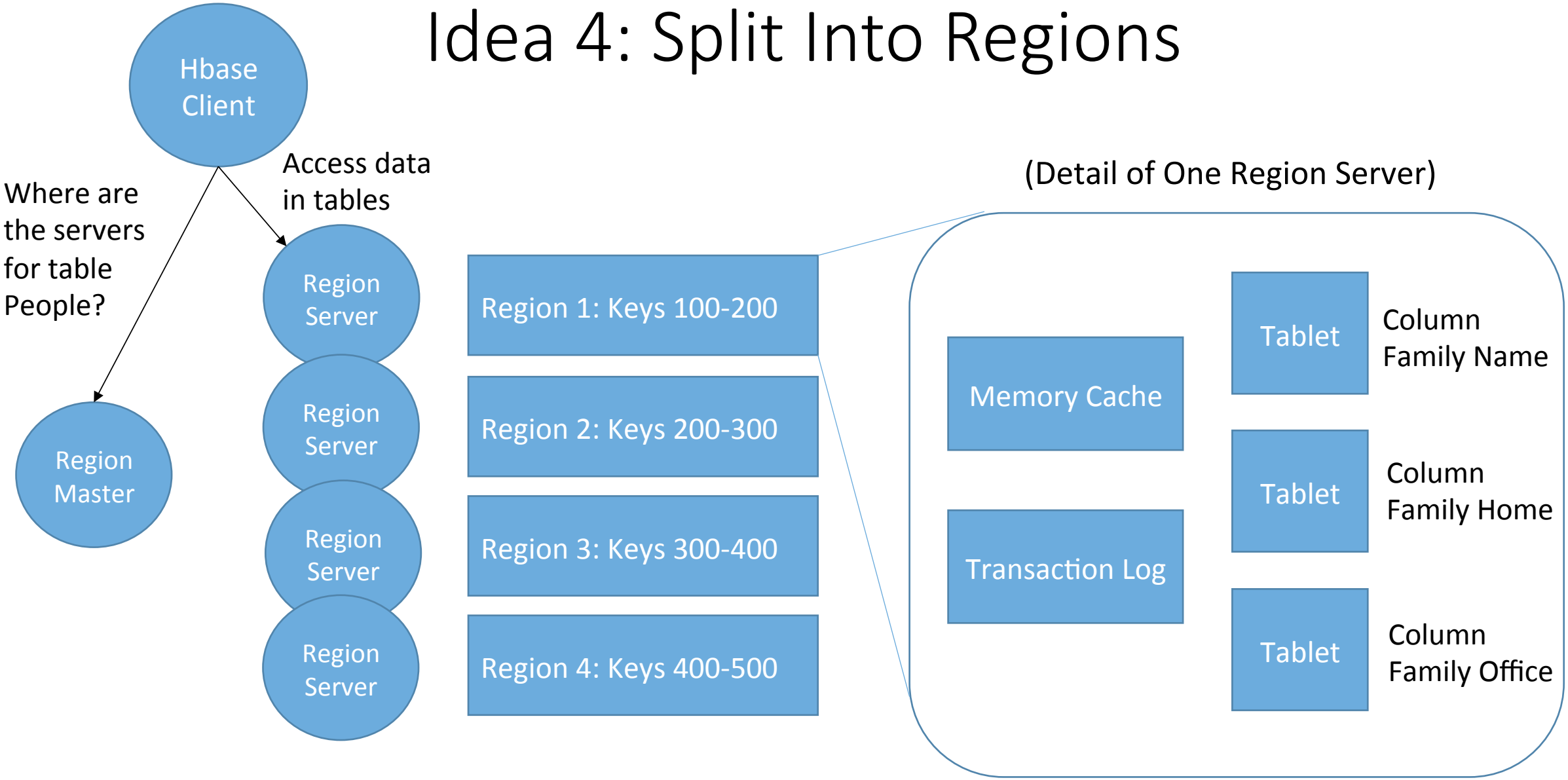
Tablets for People on Disk: (New)

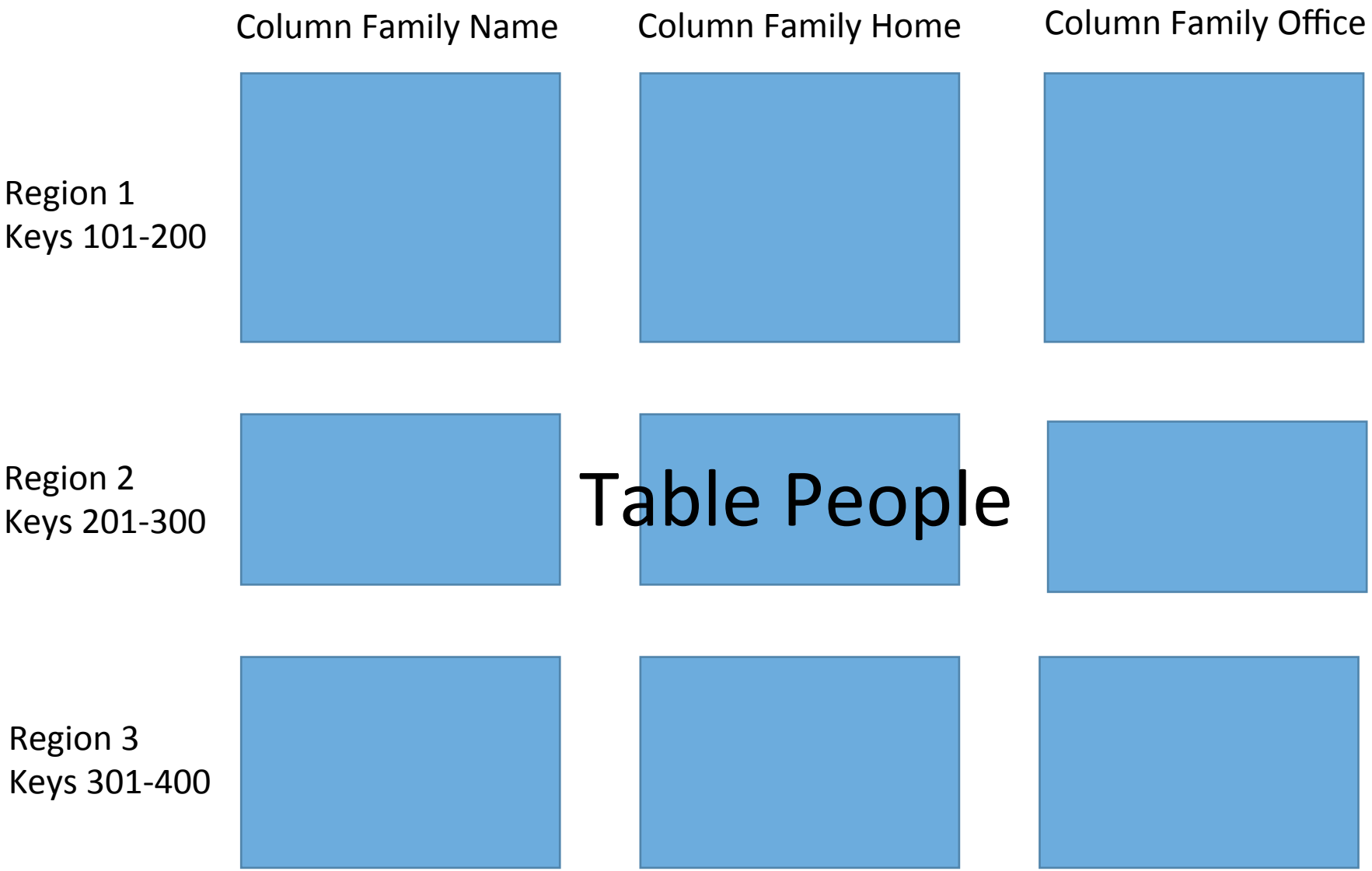
Data for
Column Family
Name

Data for
Column Family
Home (Changed)

Data for
Column Family
Office (Changed)

Idea 4: Split Into Regions




Column Family Name
Column Family Home
Column Family Office
Region 1
Keys 101-200
Region 2
Keys 201-300
Region 3
Keys 301-400
Table People

The consistency model is tightly coupled to the scalability of the implementation!

Consistency Model

- Atomicity: Entire rows are updated atomically or not at all.
- 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, and may include updates since it started.
- Isolation: Not guaranteed outside a single row.
- Durability: All successful writes have been made durable on disk.

Questions for Discussion

- What are the tradeoffs between having a very tall vertical table versus having a very wide horizontal table?
- Suppose a table starts off small, and then a large number of rows are added to the table. What happens and what should the system do?
- Using the People example, come up with some examples of simple queries that are very fast, and some that are very slow. Is there anything that can be done about the slow queries?
- Would it be possible to have SCAN return a consistent snapshot of the system? How would that work?