Big Data Technologies

Jnaneshwar Bohara

Chapter 4: NoSQL

Structured and Unstructured Data

- Structured data

  Structured data refers to any data that resides in a fixed field within a record or file. This includes data contained in relational databases and spreadsheets.

- Unstructured data

  Unstructured data is all those things that can't be so readily classified and fit into a neat box: photos and graphic images, videos, streaming instrument data, webpages, PDF files, PowerPoint presentations, emails, blog entries, wikis and word processing documents.

Structured vs unstructured data

- Structured data : information in “tables”

Employee	Manager	Salary
Smith	Jones	50000
Chang	Smith	60000
Ivy	Smith	50000

Typically allows numerical range and exact match (for text) queries, e.g.,
Salary < 60000 AND Manager = Smith.

Unstructured data

- Typically refers to free text

- Allows
  - Keyword-based queries including operators
  - More sophisticated “concept” queries, e.g.,
    - find all web pages dealing with *drug abuse*

Features of “unstructured” data

- Does not reside in traditional databases and data warehouses
- May have an internal structure, but does not fit a relational data model
- Generated by both humans and machines
  - Textual and multimedia content
  - Machine-to-machine communication
- Examples include
  - Personal messaging – email, instant messages, tweets, chat
  - Business documents – business reports, presentations, survey responses
  - Web content – web pages, blogs, wikis, audio files, photos, videos
  - Sensor output – satellite imagery, geolocation data, scanner transactions

Semi-structured data

- In fact almost no data is "unstructured"
  - E.g., this slide has distinctly identified zones such as the *Title* and *Bullets*
- Facilitates "semi-structured" search such as
  - *Title* contains <u>data</u> AND *Bullets* contain <u>search</u>

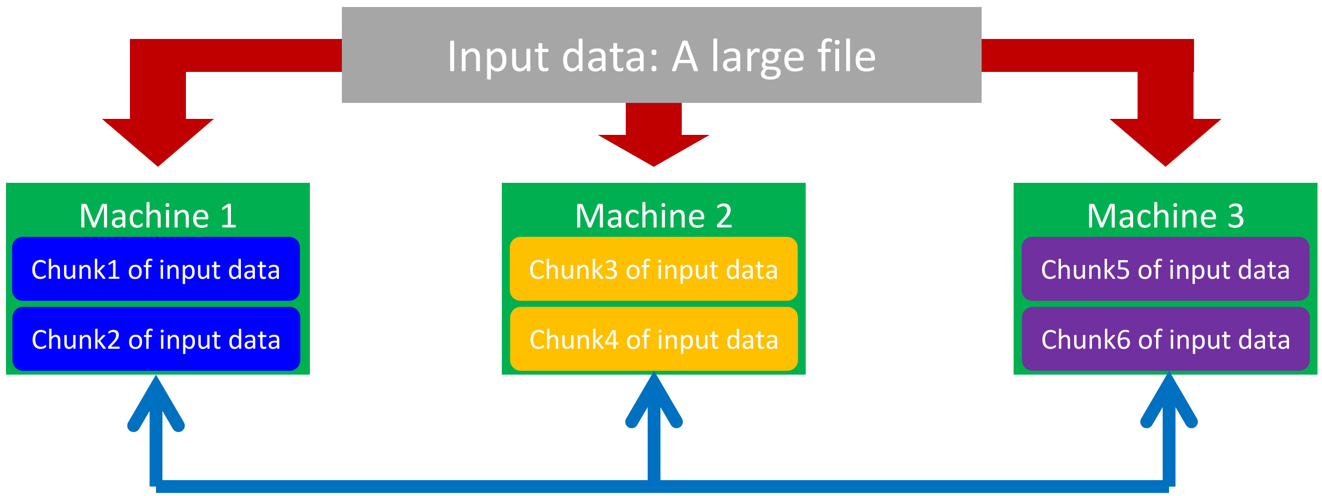
    … to say nothing of linguistic structure

Scaling Traditional Databases

- Traditional RDBMSs can be either scaled:
  - Vertically (or Up)
    - Can be achieved by hardware upgrades (e.g., faster CPU, more memory, or larger disk)
    - Limited by the amount of CPU, RAM and disk that can be configured on a single machine
  - Horizontally (or Out)
    - Can be achieved by adding more machines
    - Requires database *sharding* and probably *replication*
    - Limited by the Read-to-Write ratio and communication overhead

Why Sharding Data?

- Data is typically *sharded* (or *striped*) to allow for concurrent/parallel accesses



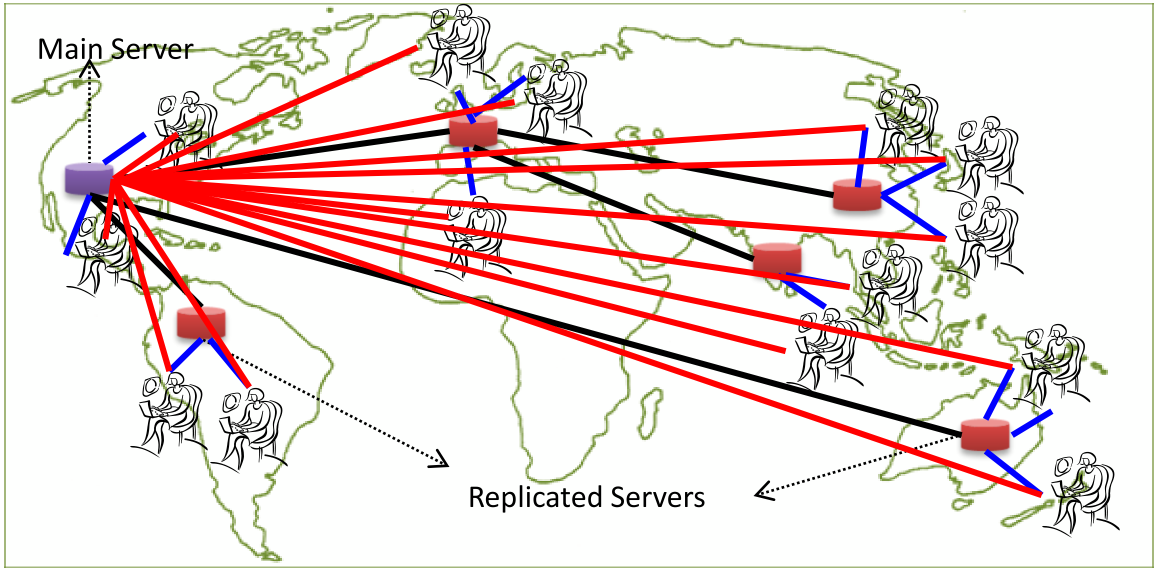
E.g., Chunks 1, 3 and 5 can be accessed in parallel

Why Replicating Data?

- Replicating data across servers helps in:
  - Avoiding performance bottlenecks
  - Avoiding single point of failures
  - And, hence, enhancing scalability and availability

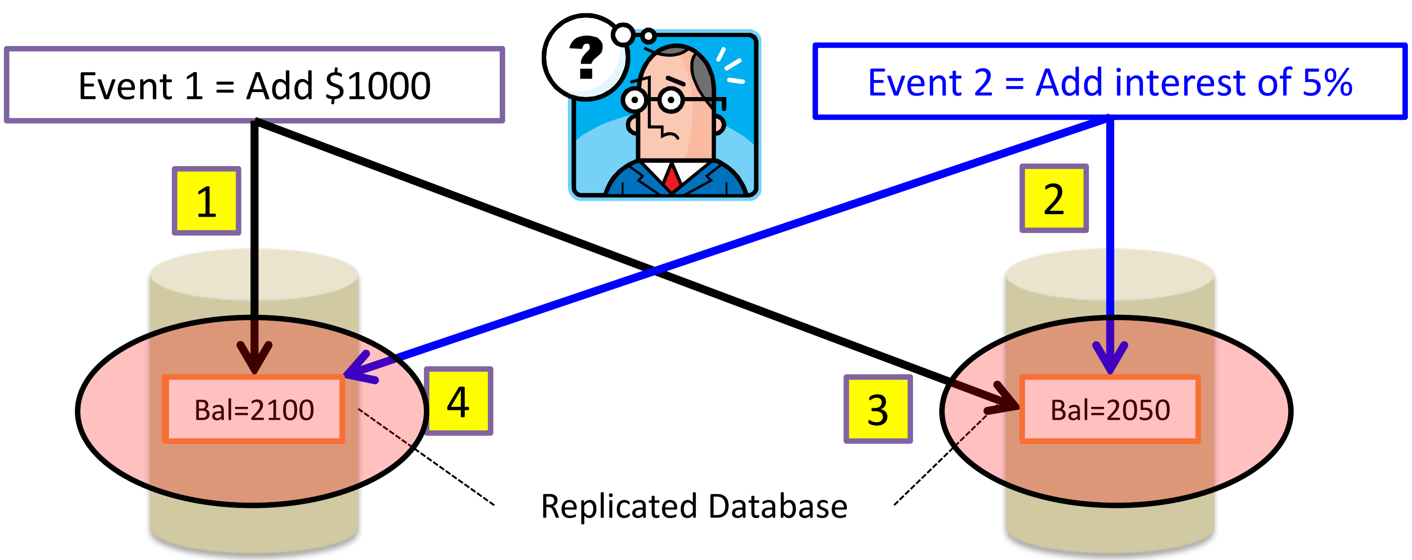
Why Replicating Data?

- Replicating data across servers helps in:
  - Avoiding performance bottlenecks
  - Avoiding single point of failures
  - And, hence, enhancing scalability and availability



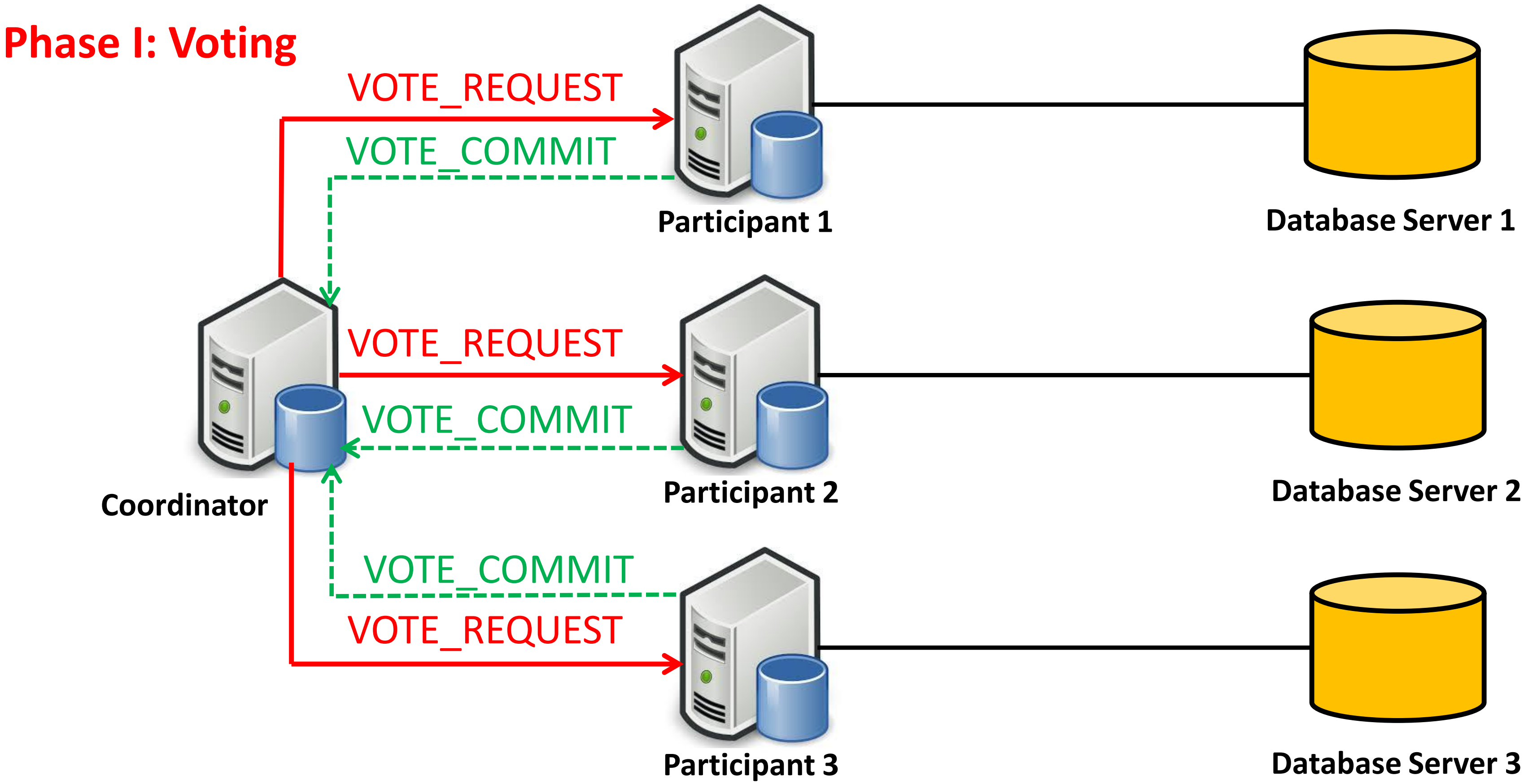
But, Consistency Becomes a Challenge

- An example:
  - In an e-commerce application, the bank database has been replicated across two servers
  - Maintaining consistency of replicated data is a challenge



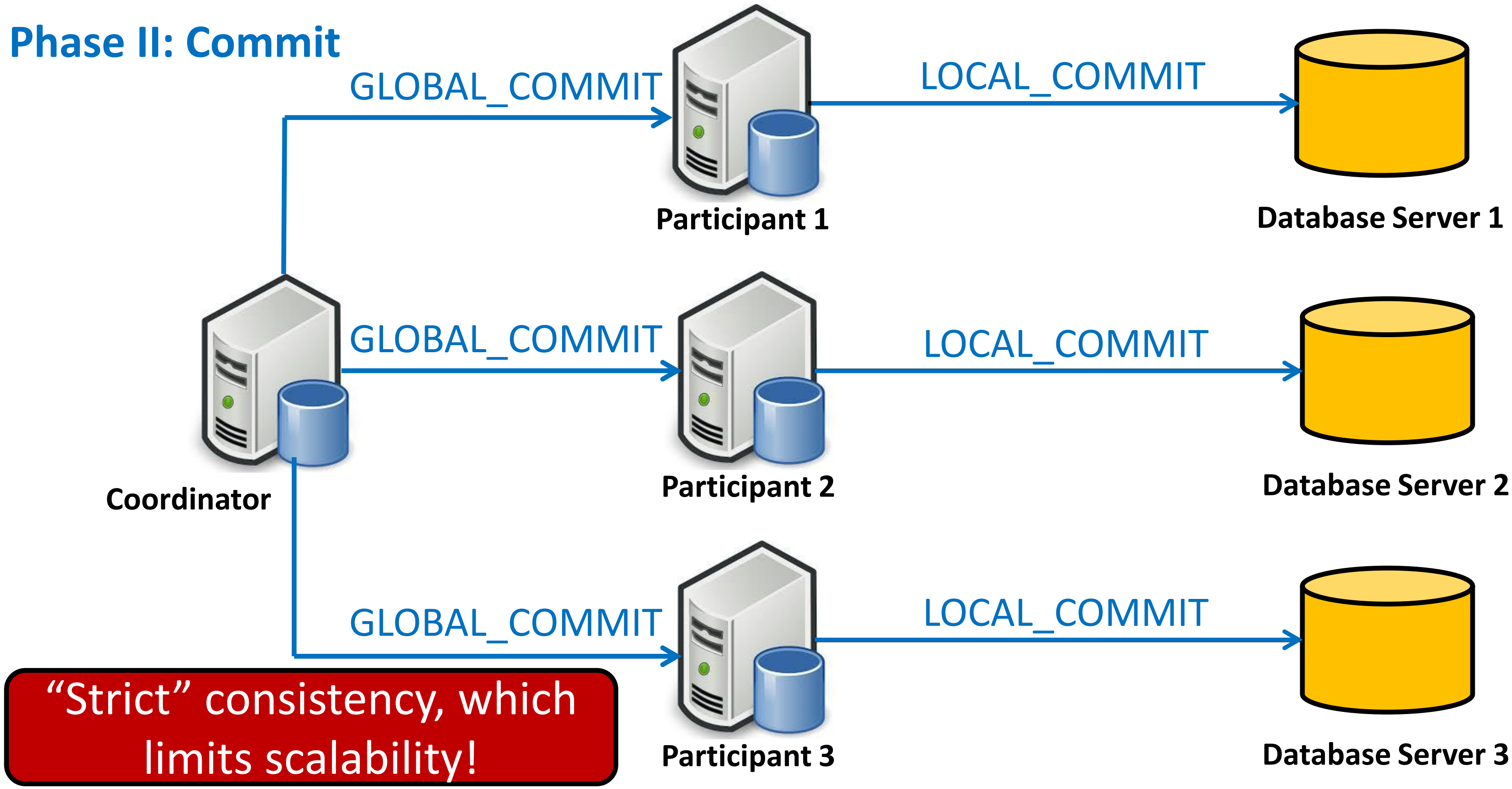
The Two-Phase Commit Protocol

- The two-phase commit protocol (2PC) can be used to ensure atomicity and consistency



The Two-Phase Commit Protocol

- The two-phase commit protocol (2PC) can be used to ensure atomicity and consistency



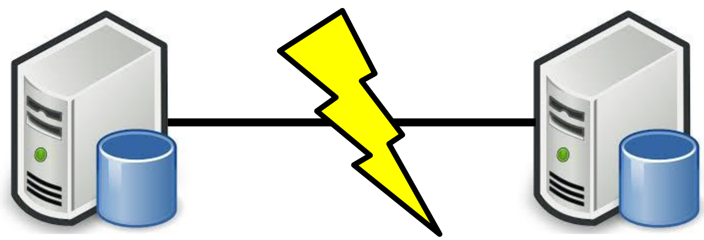
The CAP Theorem

- The limitations of distributed databases can be described in the so called the CAP theorem
  - **C**onsistency: every node always sees the same data at any given instance (i.e., strict consistency)
  - **A**vailability: the system continues to operate, even if nodes in a cluster crash, or some hardware or software parts are down due to upgrades
  - **P**artition Tolerance: the system continues to operate in the presence of network partitions

CAP theorem: any distributed database with shared data, can have ***at most two*** of the three desirable properties, C, A or P

The CAP Theorem (*Cont'd*)

- Let us assume two nodes on opposite sides of a network partition:

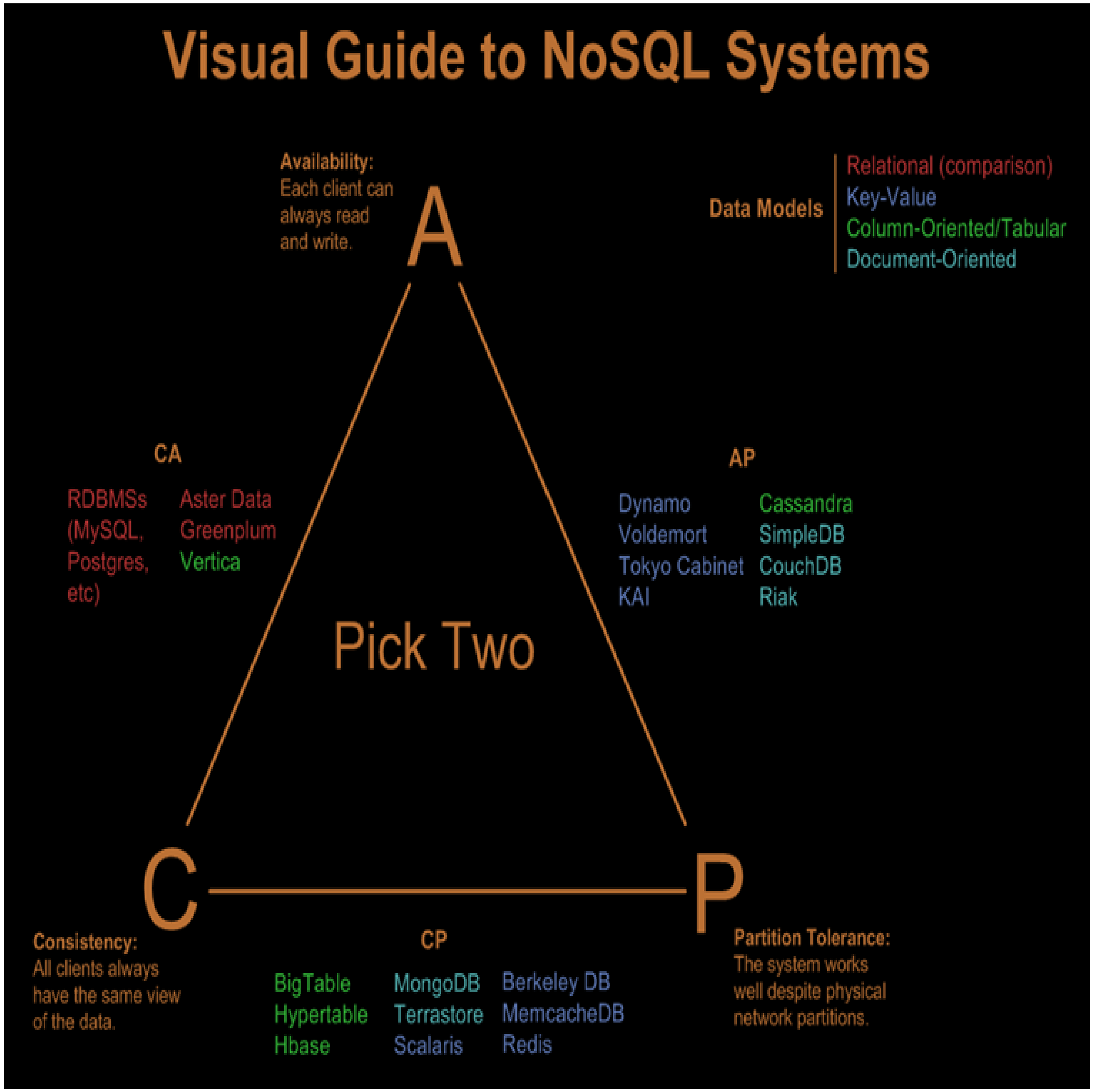


- Availability + Partition Tolerance forfeit Consistency

- Consistency + Partition Tolerance entails that one side of the partition must act as if it is unavailable, thus forfeiting Availability

- Consistency + Availability is only possible if there is no network partition, thereby forfeiting Partition Tolerance

CAP Theorem



Visual Guide to NoSQL Systems
Availability: Each client can always read and write.
A
Data Models
Relational (comparison)
Key-Value
Column-Oriented/Tabular
Document-Oriented
CA
RDBMSs (MySQL, Postgres, etc)
Aster Data
Greenplum
Vertica
AP
Dynamo
Voldemort
Tokyo Cabinet
KAI
Cassandra
SimpleDB
CouchDB
Riak
Pick Two
C
P
Consistency: All clients always have the same view of the data.
CP
BigTable
Hypertable
Hbase
MongoDB
Terrastore
Scalaris
Berkeley DB
MemcacheDB
Redis
Partition Tolerance: The system works well despite physical network partitions.

Large-Scale Databases

- When companies such as Google and Amazon were designing large-scale databases, 24/7 Availability was a key
  - A few minutes of downtime means lost revenue
- When *horizontally* scaling databases to 1000s of machines, the likelihood of a node or a network failure increases tremendously
- Therefore, in order to have strong guarantees on Availability and Partition Tolerance, they had to sacrifice "strict" Consistency (*implied by the CAP theorem*)

Trading-Off Consistency

- Maintaining consistency should balance between the strictness of consistency versus availability/scalability
  - Good-enough consistency ***depends on your application***

Trading-Off Consistency

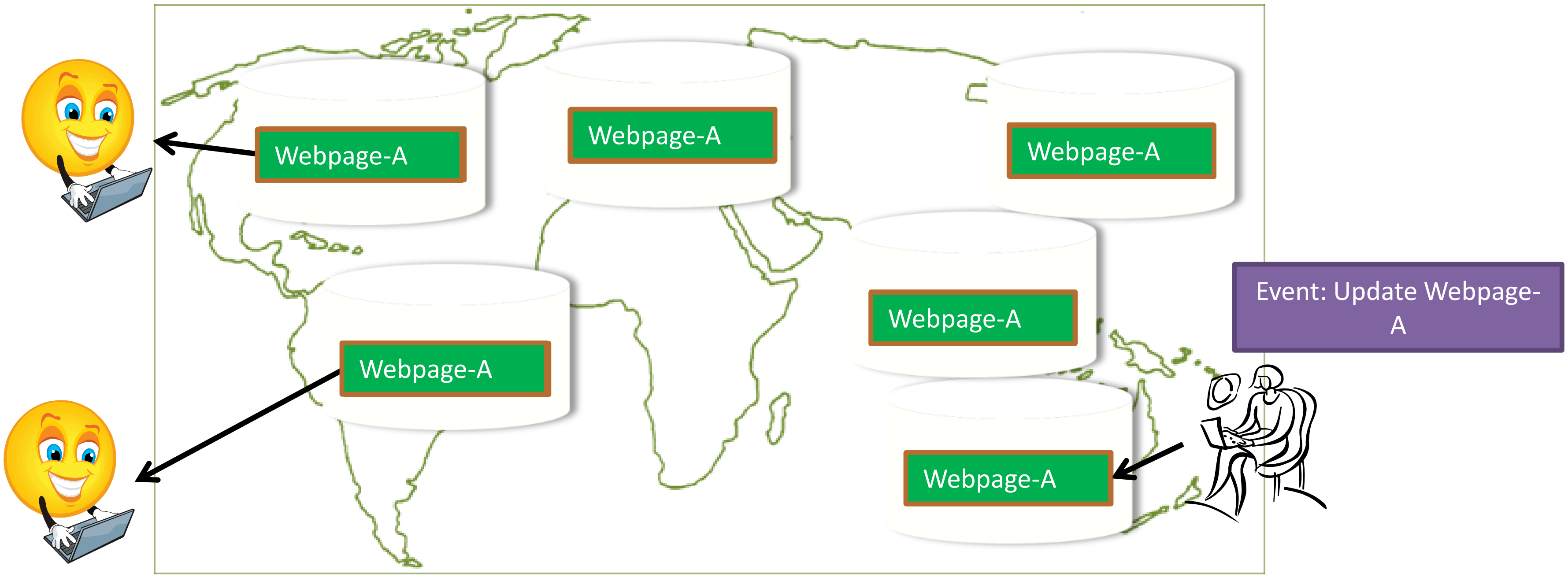
- Maintaining consistency should balance between the strictness of consistency versus availability/scalability
  - Good-enough consistency ***depends on your application***

Eventual Consistency

- A database is termed as *Eventually Consistent* if:
  - All replicas will *gradually* become consistent in the absence of updates

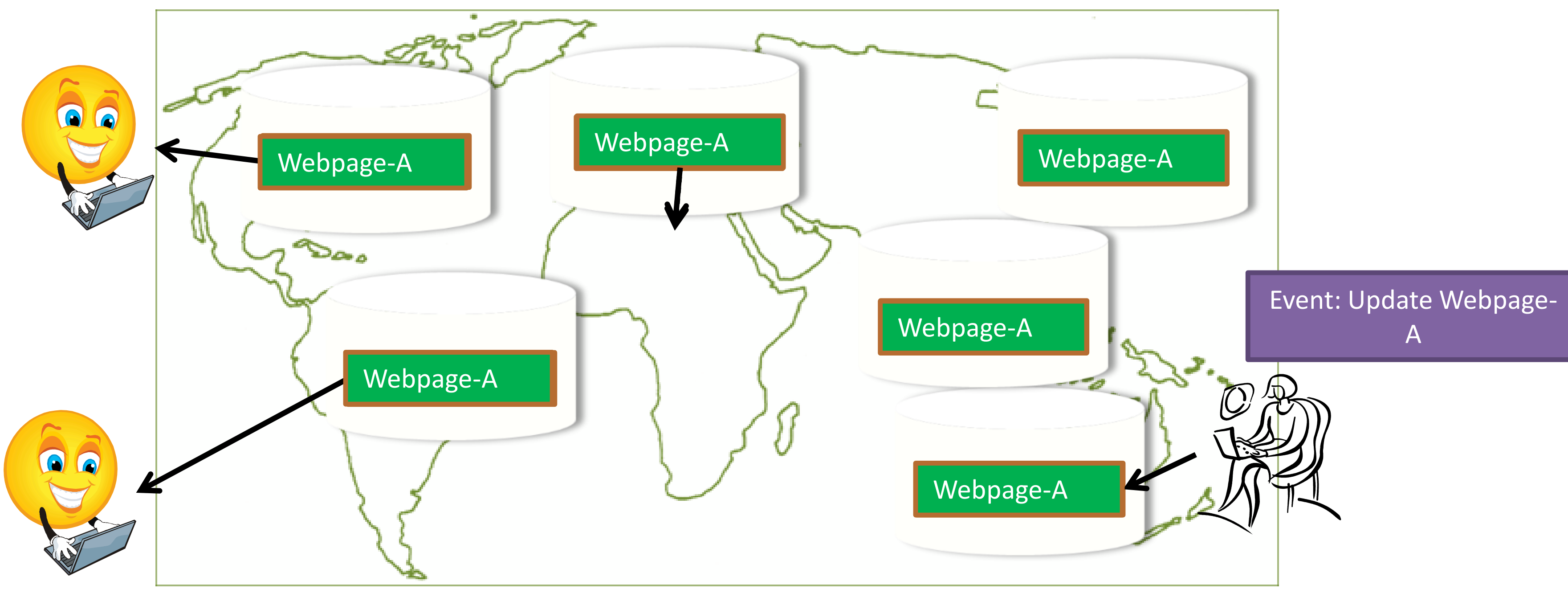
Eventual Consistency

- A database is termed as *Eventually Consistent* if:
  - All replicas will *gradually* become consistent in the absence of updates



Eventual Consistency: A Main Challenge

- But, what if the client accesses the data from different replicas?



Protocols like Read Your Own Writes (RYOW) can be applied!

The BASE Properties

- The CAP theorem proves that it is impossible to guarantee strict Consistency and Availability while being able to tolerate network partitions

- This resulted in databases with relaxed ACID guarantees

- In particular, such databases apply the BASE properties:
  - **B**asically **A**vailable: the system guarantees Availability
  - **S**oft-State: the state of the system may change over time
  - **E**ventual Consistency: the system will *eventually* become consistent

What is NoSQL?

- Class of database management systems (DBMS)
- "*Not only SQL*"
  - Does not use SQL as querying language
  - Distributed, fault-tolerant architecture
  - No fixed schema (formally described structure)
  - No joins (typical in databases operated with SQL)
    - Expensive operation for combining records from two or more tables into one set
    - Joins require strong consistency and fixed schemas

  Lack of these makes NoSQL databases more flexible
- It's not a replacement for a RDBMS but compliments it

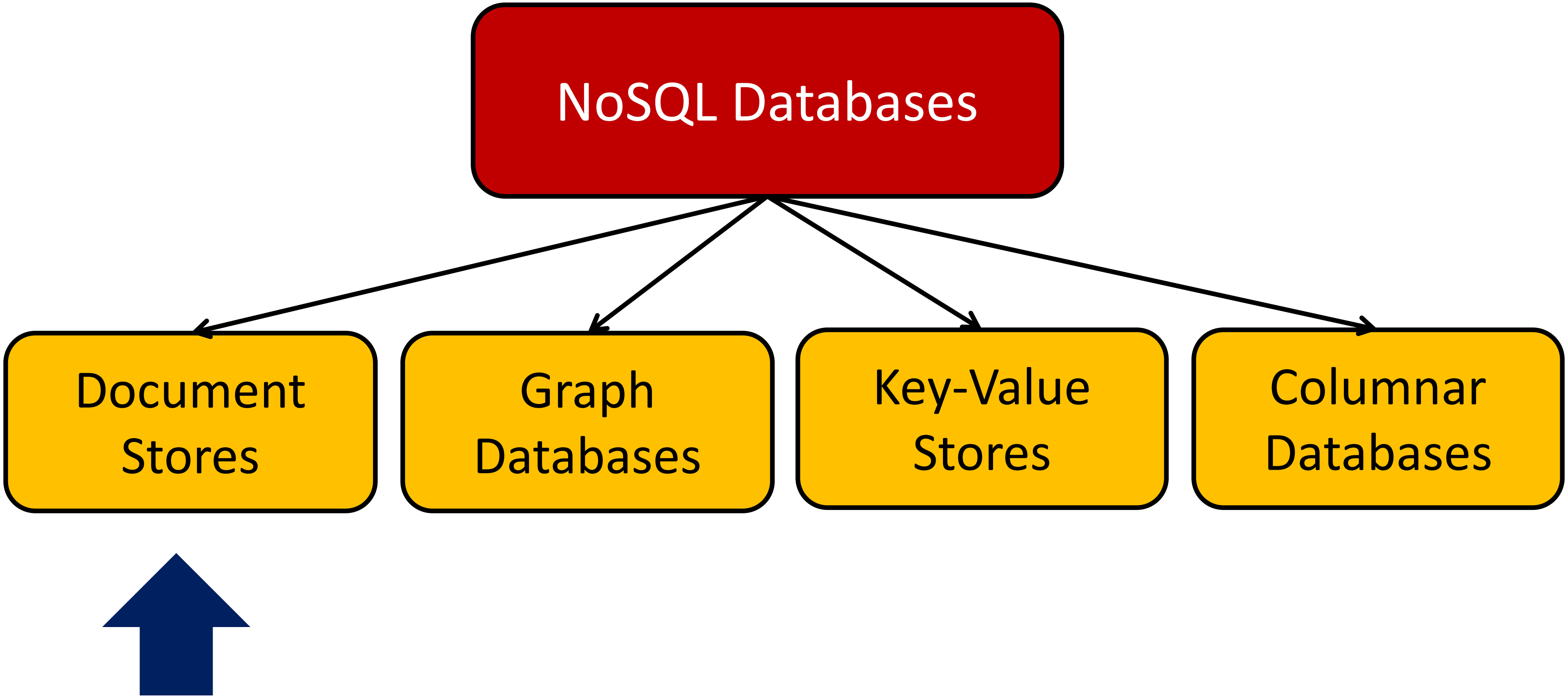
NoSQL Databases

- To this end, a new class of databases emerged, which mainly follow the BASE properties
  - These were dubbed as NoSQL databases
  - E.g., Amazon's Dynamo and Google's Bigtable

- Main characteristics of NoSQL databases include:
  - No strict schema requirements
  - No strict adherence to ACID properties
  - Consistency is traded in favor of Availability

Types of NoSQL Databases

- Here is a limited taxonomy of NoSQL databases:

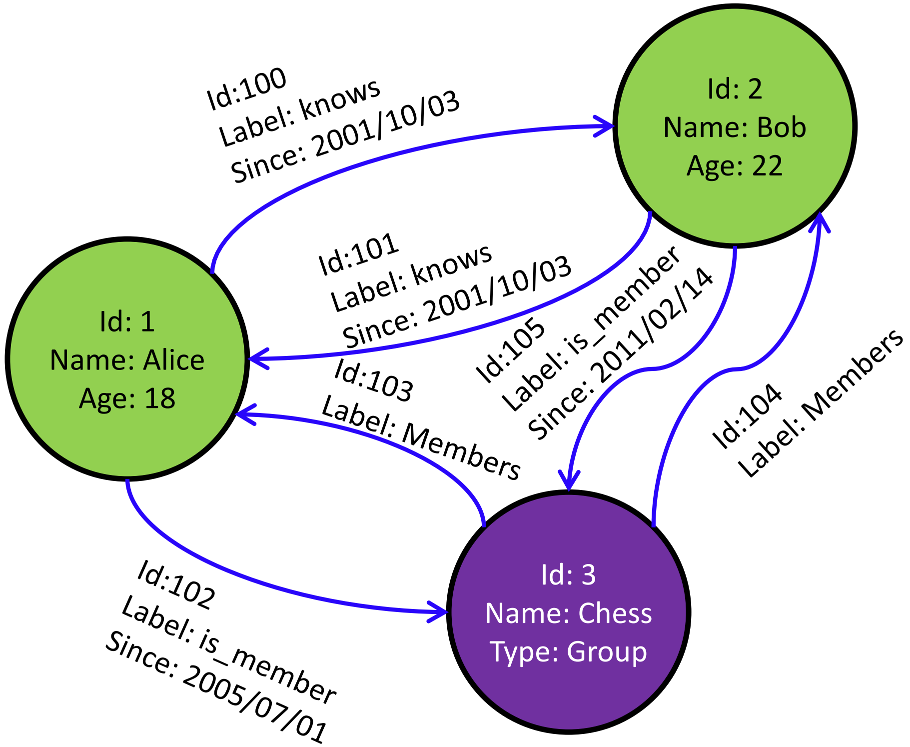


Document Stores

- Documents are stored in some standard format or encoding (e.g., XML, JSON, PDF or Office Documents)
  - These are typically referred to as Binary Large Objects (BLOBs)
- Documents can be indexed
  - This allows document stores to outperform traditional file systems
- E.g., MongoDB and CouchDB (both can be queried using MapReduce)

Graph Databases

- Data are represented as vertices and edges



- Graph databases are powerful for graph-like queries (e.g., find the shortest path between two elements)

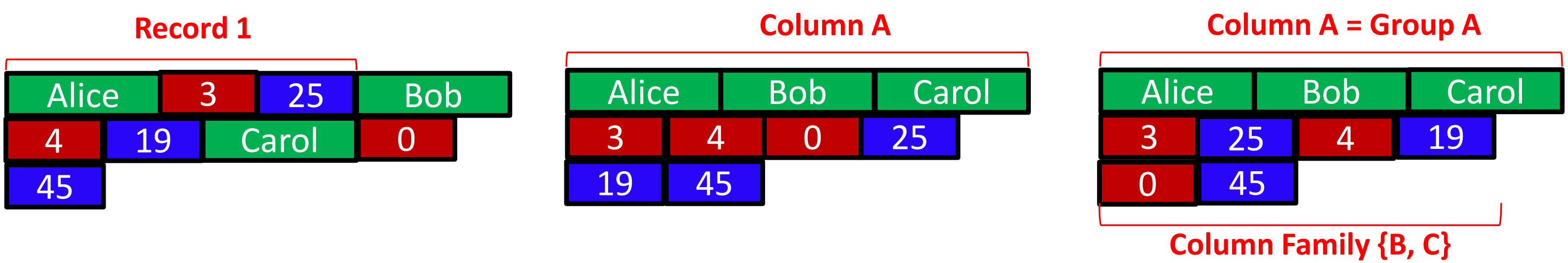
- E.g., Neo4j and VertexDB

Key-Value Stores

- Keys are mapped to (possibly) more complex value (e.g., lists)
- Keys can be stored in a hash table and can be distributed easily
- Such stores typically support regular CRUD (create, read, update, and delete) operations
  - That is, no joins and aggregate functions
- E.g., Amazon DynamoDB and Redis

Columnar Databases

- Columnar databases are a hybrid of RDBMSs and Key-Value stores
  - Values are stored in groups of zero or more columns, but in Column-Order (as opposed to Row-Order)



Row-Order ***Columnar (or Column-Order)*** ***Columnar with Locality Groups***

  - Values are queried by matching keys

- E.g., Hbase, Cassandra

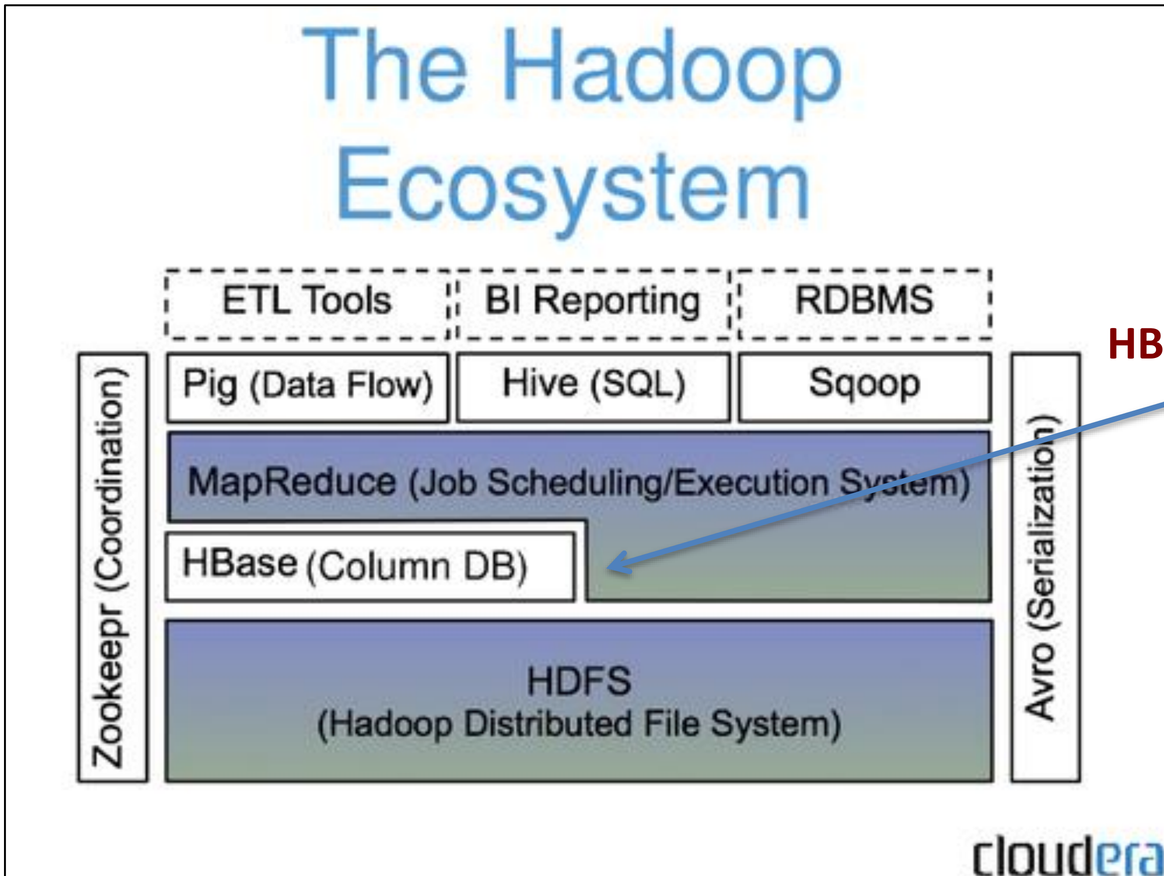
NoSQL

- Key-value
- Graph database
- Document-oriented
- Column family

HBase: Overview

- **HBase is a distributed column-oriented data store built on top of HDFS**
- **HBase is an Apache open source project whose goal is to provide storage for the Hadoop Distributed Computing**
- **Data is logically organized into tables, rows and columns**

HBase: Part of Hadoop's Ecosystem



HBase is built on top of HDFS

HBase files are internally stored in HDFS

HBase vs. HDFS

- Both are distributed systems that scale to hundreds or thousands of nodes

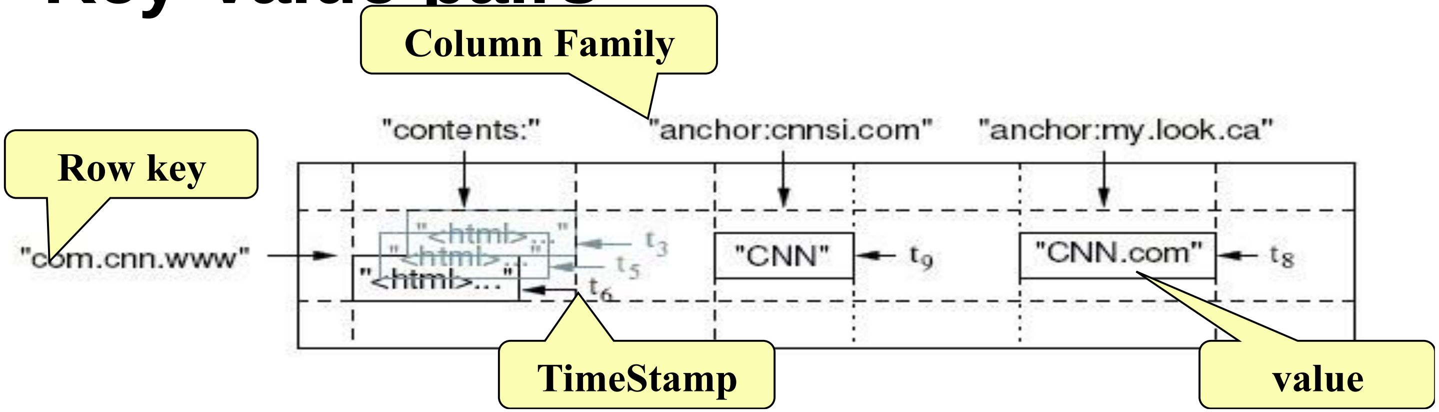
- ***HDFS*** is good for batch processing (scans over big files)
  - Not good for record lookup
  - Not good for incremental addition of small batches
  - Not good for updates

HBase vs. HDFS (Cont'd)

- ***HBase*** is designed to efficiently address the above points
  - Fast record lookup
  - Support for record-level insertion
  - Support for updates (not in place)
- HBase updates are done by creating new versions of values

HBase Data Model

- **HBase is based on Google's Bigtable model**
  - **Key-Value pairs**



HBase Logical View

Implicit PRIMARY KEY in RDBMS terms

Data is all `byte[]` in HBase

Different types of data separated into different "column families"

Row key	Data
cutting	info: { 'height': '9ft', 'state': 'CA' } roles: { 'ASF': 'Director', 'Hadoop': 'Founder' }
tlipcon	info: { 'height': '5ft7, 'state': 'CA' } roles: { 'Hadoop': 'Committer'@ts=2010, 'Hadoop': 'PMC'@ts=2011, 'Hive': 'Contributor' }

Different rows may have different sets of columns(table is *sparse*)

A single cell might have different values at different timestamps

Useful for *-To-Many mappings

HBase: Keys and Column Families

Each record is divided into *Column Families*

Each row has a *Key*

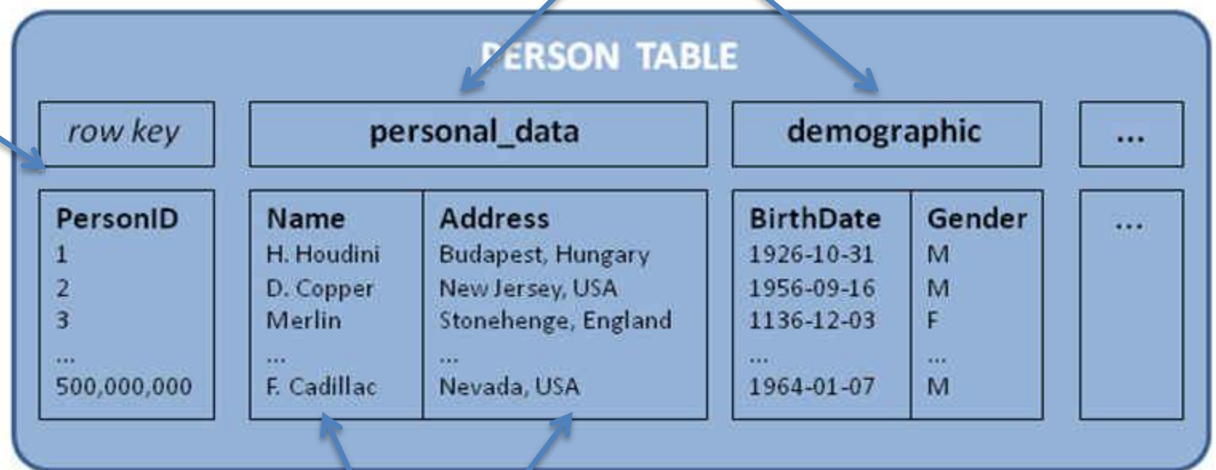


Figure 2: Census Data in Column Families

Each column family consists of one or more *Columns*

- **Key**
  - Byte array
  - Serves as the primary key for the table
  - Indexed far fast lookup
- **Column Family**
  - Has a name (string)
  - Contains one or more related columns
- **Column**
  - Belongs to one column family
  - Included inside the row
    - ***familyName:columnName***

Column family named "anchor"

Column family named "Contents"

Column named "apache.com"

Row key	Time Stamp	Column "contents:"	Column "anchor:"	
"com.apache.www"	t12	"<html> …"		
	t11	"<html> …"		
	t10		"anchor:apache.com"	"APACHE"
"com.cnn.www"	t15		"anchor:cnnsi.com"	"CNN"
	t13		"anchor:my.look.ca"	"CNN.com"
	t6	"<html> …"		
	t5	"<html> …"		
	t3	"<html> …"		

- **Version Number**
  - Unique within each key
  - By default→ System's timestamp
  - Data type is Long
- **Value (Cell)**
  - Byte array

Version number for each row

Row key	Time Stamp	Column "contents:"	Column "anchor:"	
"com.apache.www"	t12	"<html> …"		
	t11	"<html> …"		
	t10		"anchor:apache.com"	"APACHE"
"com.cnn.www"	t15		"anchor:cnnsi.com"	"CNN"
	t13		"anchor:my.look.ca"	"CNN.com"
	t6	"<html> …"		
	t5	"<html> …"		
	t3	"<html> …"		

value

Notes on Data Model

- HBase schema consists of several ***Tables***
- Each table consists of a set of ***Column Families***
  - Columns are not part of the schema
- HBase has ***Dynamic Columns***
  - Because column names are encoded inside the cells
  - Different cells can have different columns

"Roles" column family has different columns in different cells

Row key	Data
cutting	info: { 'height': '9ft', 'state': 'CA' } roles: { 'ASF': 'Director', 'Hadoop': 'Founder' }
tlipcon	info: { 'height': '5ft7, 'state': 'CA' } roles: { 'Hadoop': 'Committer'@ts=2010, 'Hadoop': 'PMC'@ts=2011, 'Hive': 'Contributor' }

Notes on Data Model (Cont'd)

- The ***version number*** can be user-supplied
  - Even does not have to be inserted in increasing order
  - Version number are unique within each key
- Table can be very sparse
  - Many cells are empty
- ***Keys*** are indexed as the primary key

Has two columns
[cnnsi.com & my.look.ca]

Row Key	**Time Stamp**	**ColumnFamily** `contents`	**ColumnFamily** `anchor`
"com.cnn.www"	t9		anchor:cnnsi.com = "CNN"
"com.cnn.www"	t8		anchor:my.look.ca = "CNN.com"
"com.cnn.www"	t6	contents:html = "<html>..."	
"com.cnn.www"	t5	contents:html = "<html>..."	
"com.cnn.www"	t3	contents:html = "<html>..."	

HBase Physical Model

- Each column family is stored in a separate file (called ***HTables***)
- Key & Version numbers are replicated with each column family
- Empty cells are not stored

HBase maintains a multi-level index on values:
<key, column family, column name, timestamp>

Table 5.3. ColumnFamily `contents`

Row Key	Time Stamp	ColumnFamily "contents:"
"com.cnn.www"	t6	`contents:html` = "<html>..."
"com.cnn.www"	t5	`contents:html` = "<html>..."
"com.cnn.www"	t3	`contents:html` = "<html>..."

Table 5.2. ColumnFamily `anchor`

Row Key	Time Stamp	Column Family `anchor`
"com.cnn.www"	t9	`anchor:cnnsi.com` = "CNN"
"com.cnn.www"	t8	`anchor:my.look.ca` = "CNN.com"

Example

Row key	Data
cutting	info: { 'height': '9ft', 'state': 'CA' } roles: { 'ASF': 'Director', 'Hadoop': 'Founder' }
tlipcon	info: { 'height': '5ft7, 'state': 'CA' } roles: { 'Hadoop': 'Committer'@ts=2010, 'Hadoop': 'PMC'@ts=2011, 'Hive': 'Contributor' }

`info` **Column Family**

Row key	Column key	Timestamp	Cell value
cutting	info:height	1273516197868	9ft
cutting	info:state	1043871824184	CA
tlipcon	info:height	1273878447049	5ft7
tlipcon	info:state	1273616297446	CA

`roles` **Column Family**

Row key	Column key	Timestamp	Cell value
cutting	roles:ASF	1273871823022	Director
cutting	roles:Hadoop	1183746289103	Founder
tlipcon	roles:Hadoop	1300062064923	PMC
tlipcon	roles:Hadoop	1293388212294	Committer
tlipcon	roles:Hive	1273616297446	Contributor

Sorted on disk by Row key, Col key, descending timestamp

Milliseconds since unix epoch

cloudera

HBase Regions

- Each HTable (column family) is partitioned horizontally into ***regions***
  - Regions are counterpart to HDFS blocks

Table 5.3. ColumnFamily `contents`

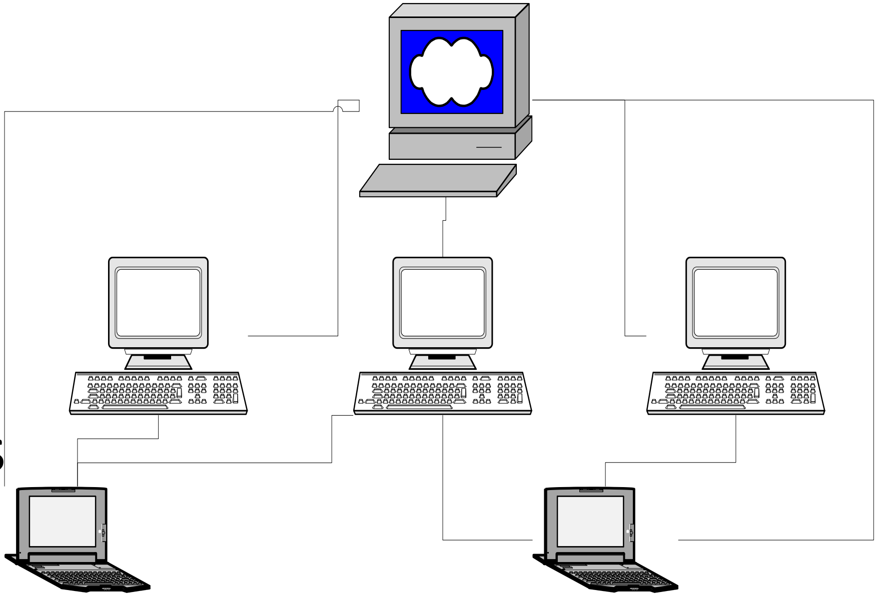
Row Key	Time Stamp	ColumnFamily "contents:"
"com.cnn.www"	t6	contents:html = "<html>..."
"com.cnn.www"	t5	contents:html = "<html>..."
"com.cnn.www"	t3	contents:html = "<html>..."

Each will be one region

HBase Architecture

Three Major Components

- The HBaseMaster
  - One master
- The HRegionServer
  - Many region servers
- The HBase client

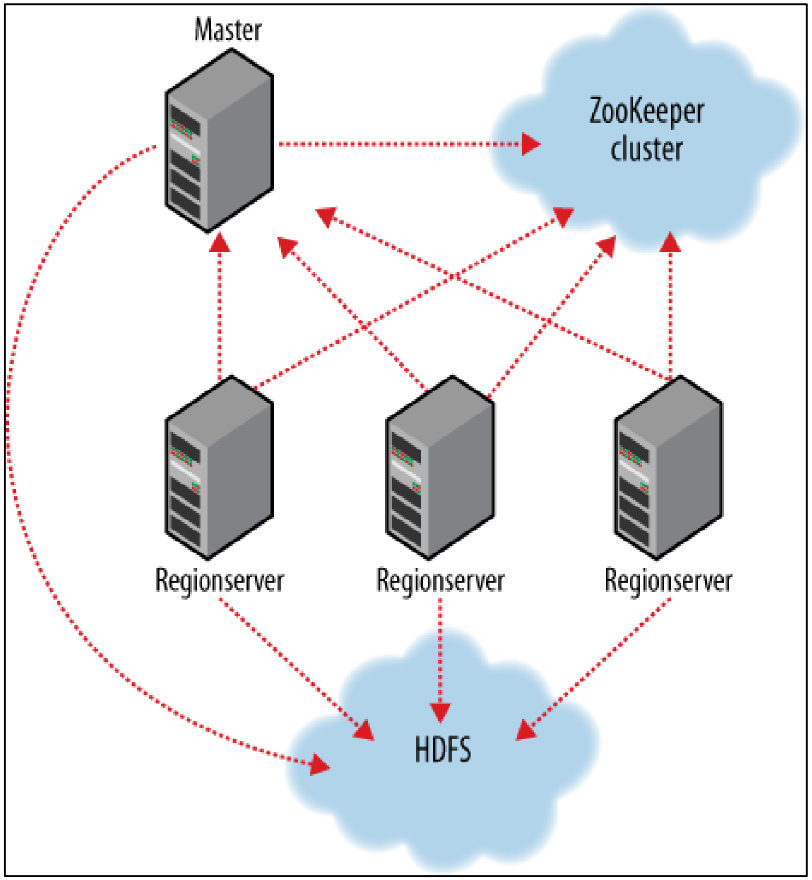


HBase Components

- **Region**
  - A subset of a table's rows, like horizontal range partitioning
  - Automatically done
- **RegionServer (many slaves)**
  - Manages data regions
  - Serves data for reads and writes (***using a log***)
- **Master**
  - Responsible for coordinating the slaves
  - Assigns regions, detects failures
  - Admin functions

ZooKeeper

- HBase depends on ZooKeeper
- By default HBase manages the ZooKeeper instance
  - E.g., starts and stops ZooKeeper
- HMaster and HRegionServers register themselves with ZooKeeper



Creating a Table

```
HBaseAdmin admin= new HBaseAdmin(config);
HColumnDescriptor []column;
column= new HColumnDescriptor[2];
column[0]=new
   HColumnDescriptor("columnFamily1:");
column[1]=new
   HColumnDescriptor("columnFamily2:");
HTableDescriptor desc= new HTableDescriptor(Bytes.toBytes("MyTable"));
desc.addFamily(column[0]);
desc.addFamily(column[1]);
admin.createTable(desc);
```

Operations On Regions: Get()

- Given a key → return corresponding record
- For each value return the highest version

```
Get get = new Get(Bytes.toBytes("row1"));
Result r = htable.get(get);
5.8.1.2. Default Get Exampleue(Bytes.toBytes("cf"), Bytes.toBytes("attr")); // returns current version of value
```

- Can control the number of versions you want

```
Get get = new Get(Bytes.toBytes("row1"));
get.setMaxVersions(3);  // will return last 3 versions of row
Result r = htable.get(get);
byte[] b = r.getValue(Bytes.toBytes("cf"), Bytes.toBytes("attr"));  // returns current version of value
List<KeyValue> kv = r.getColumn(Bytes.toBytes("cf"), Bytes.toBytes("attr"));  // returns all versions of
```

Operations On Regions: Put()

- Insert a new record (with a new key), Or
- Insert a record for an existing key

Implicit version number (timestamp)

```
Put put = new Put(Bytes.toBytes(row));
put.add(Bytes.toBytes("cf"), Bytes.toBytes("attr1"), Bytes.toBytes( data));
htable.put(put);
```

Explicit version number

```
Put put = new Put( Bytes.toBytes(row));
long explicitTimeInMs = 555;  // just an example
put.add(Bytes.toBytes("cf"), Bytes.toBytes("attr1"), explicitTimeInMs, Bytes.toBytes(data));
htable.put(put);
```

HBase vs. HDFS

	Plain HDFS/MR	HBase
Write pattern	Append-only	Random write, bulk incremental
Read pattern	Full table scan, partition table scan	Random read, small range scan, or table scan
Hive (SQL) performance	Very good	4-5x slower
Structured storage	Do-it-yourself / TSV / SequenceFile / Avro / ?	Sparse column-family data model
Max data size	30+ PB	~1PB

HBase vs. RDBMS

	RDBMS	HBase
Data layout	Row-oriented	Column-family-oriented
Transactions	Multi-row ACID	Single row only
Query language	SQL	get/put/scan/etc *
Security	Authentication/Authorization	Work in progress
Indexes	On arbitrary columns	Row-key only
Max data size	TBs	~1PB
Read/write throughput limits	1000s queries/second	Millions of queries/second

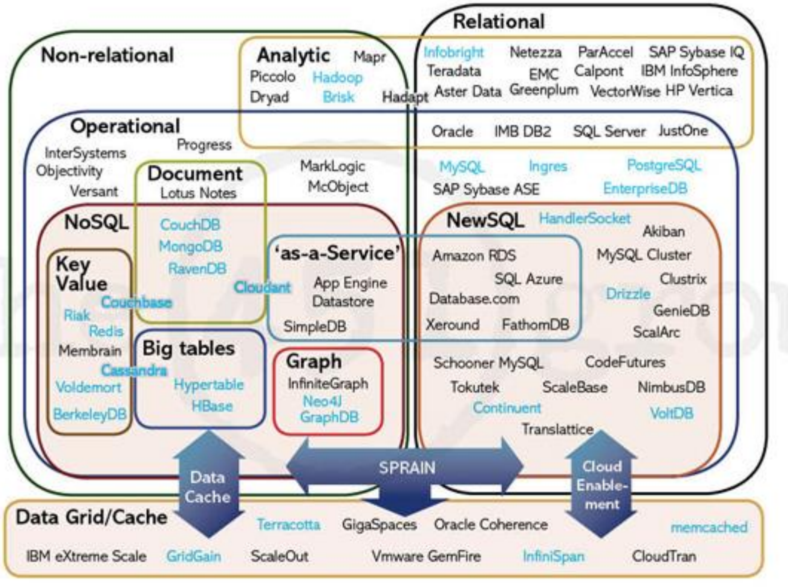
When to use HBase

- You need random write, random read, or both (*but not neither*)
- You need to do many thousands of operations per second on multiple TB of data
- Your access patterns are well-known and simple

Cassandra - A Decentralized Structured Storage System

What is Cassandra?

- Open-source database management system (DBMS)
- Several key features of Cassandra differentiate it from other similar systems



Cassandra

- Originally developed at Facebook
- Follows the BigTable data model: column-oriented
- Uses the Dynamo Eventual Consistency model
- Written in Java
- Open-sourced and exists within the Apache family
- Uses Apache Thrift as it’s API

History of Cassandra

- Cassandra was created to power the Facebook Inbox Search
- Facebook open-sourced Cassandra in 2008 and became an Apache Incubator project
- In 2010, Cassandra graduated to a top-level project, regular update and releases followed

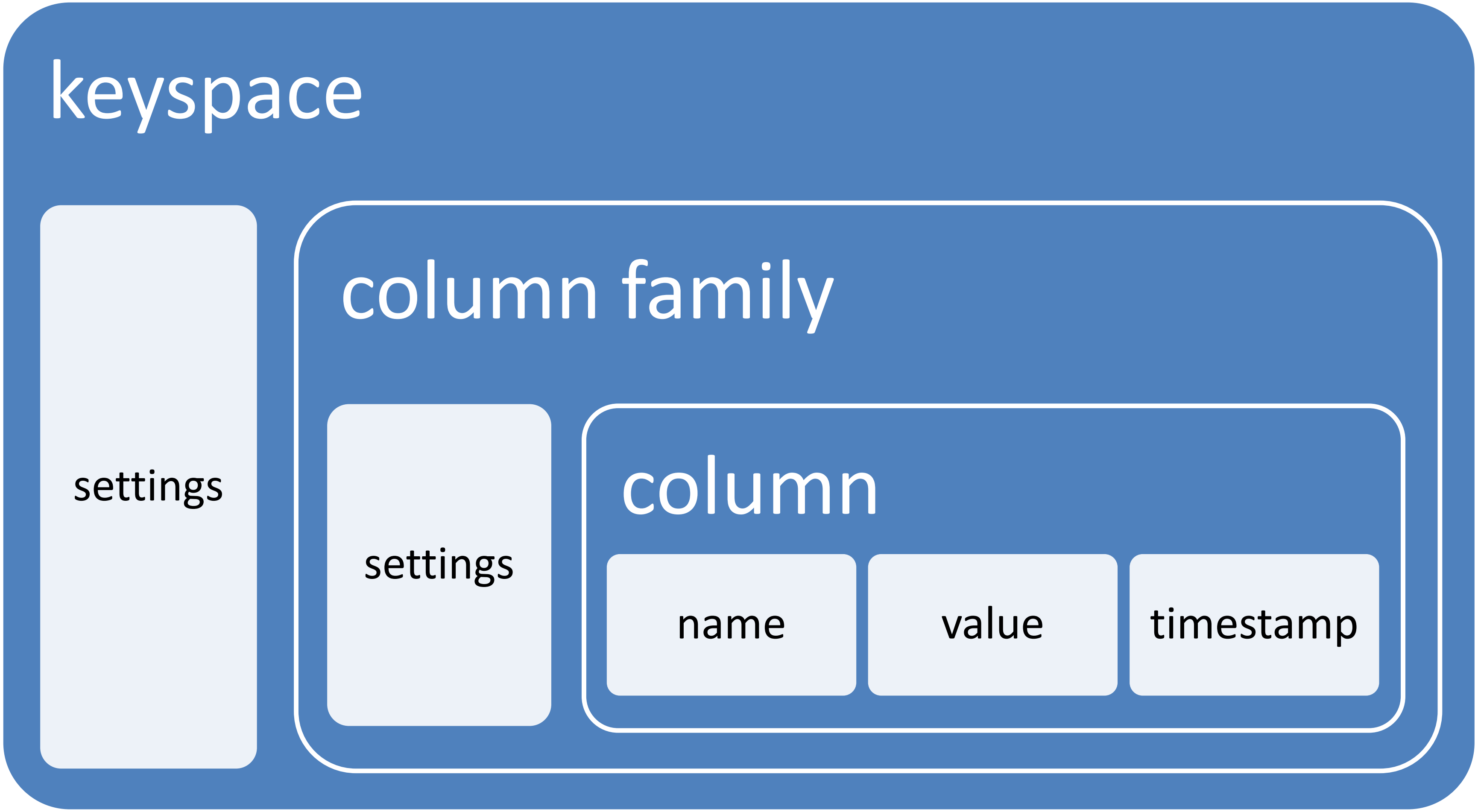
Reasons for Choosing Cassandra

- Value availability over consistency
- Require high write-throughput
- High scalability required
- No single point of failure

Data Model

- **Table is a multi dimensional map indexed by key (row key).**
- **Columns are grouped into Column Families.**
- **2 Types of Column Families**
  - **Simple**
  - **Super (nested Column Families)**
- **Each Column has**
  - **Name**
  - **Value**
  - **Timestamp**

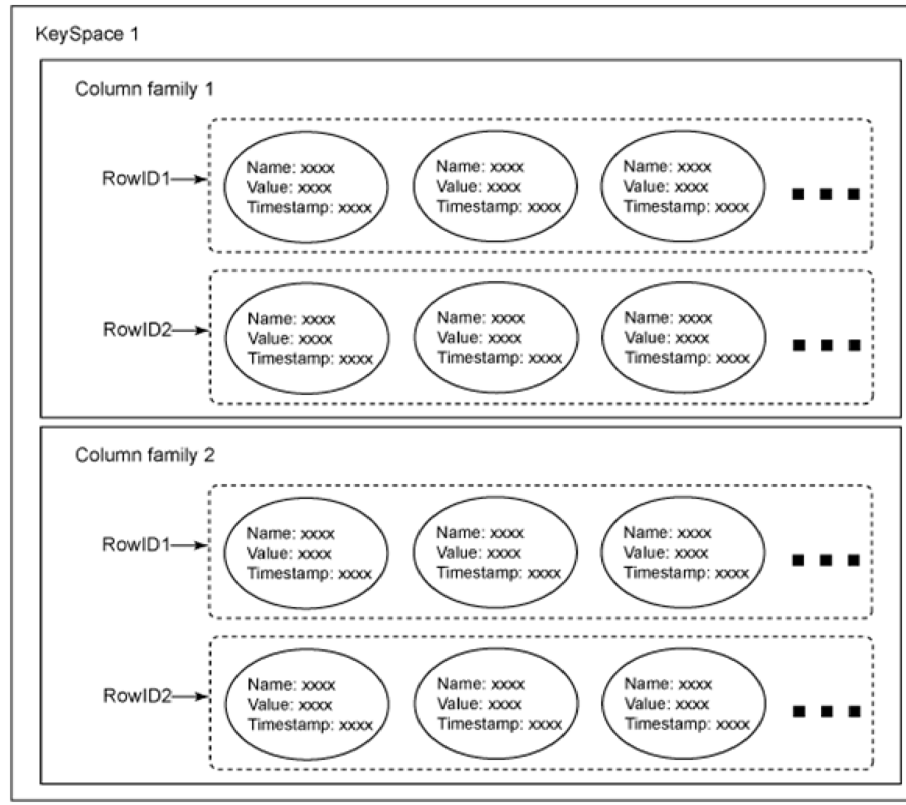
Data Model



* Figure taken from Eben Hewitt's (author of Oreilly's Cassandra book) slides.

Cassandra’s Data Model

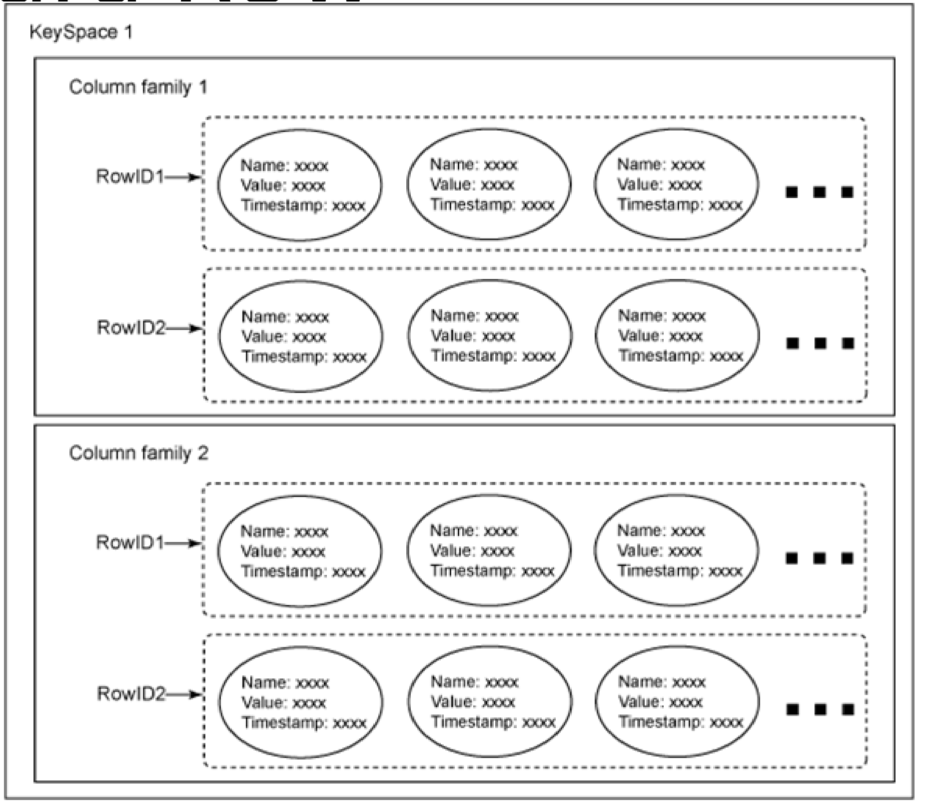
Key-Value Model

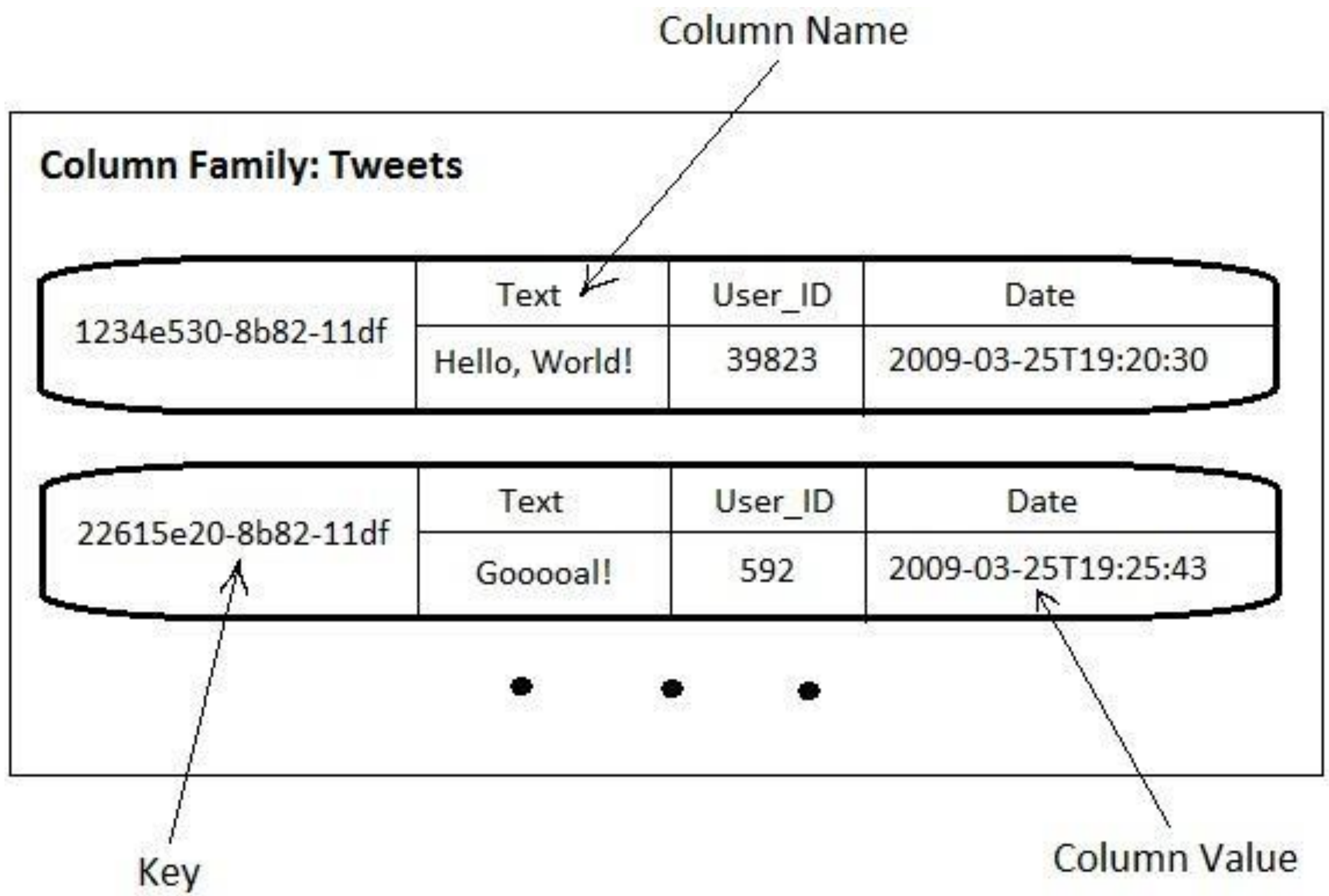


- Cassandra is a column oriented NoSQL system
- Column families: sets of key-value pairs
    - column family as a table and key-value pairs as a row (using relational database analogy)
- A row is a collection of columns labeled with a name

Cassandra Row

- the value of a row is itself a sequence of key-value pairs
- such nested key-value pairs are *columns*
- key = column name
- a row must contain at least 1 column

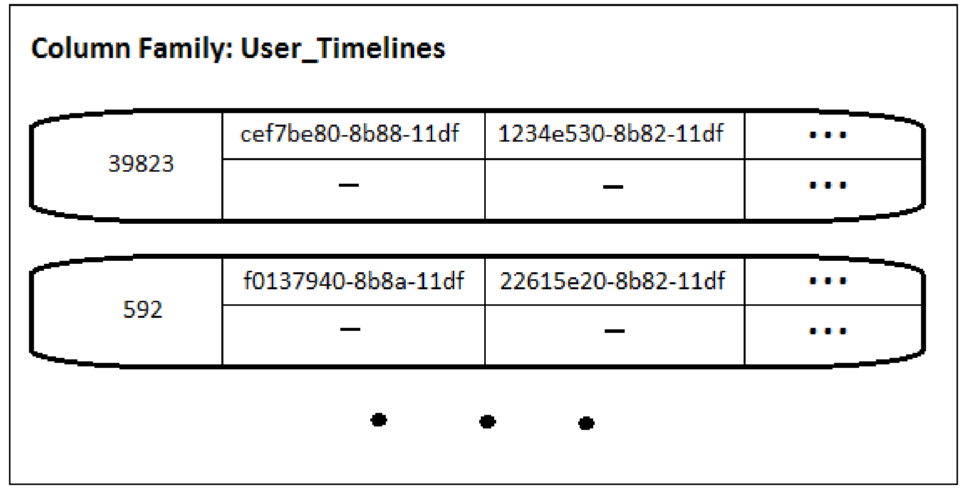




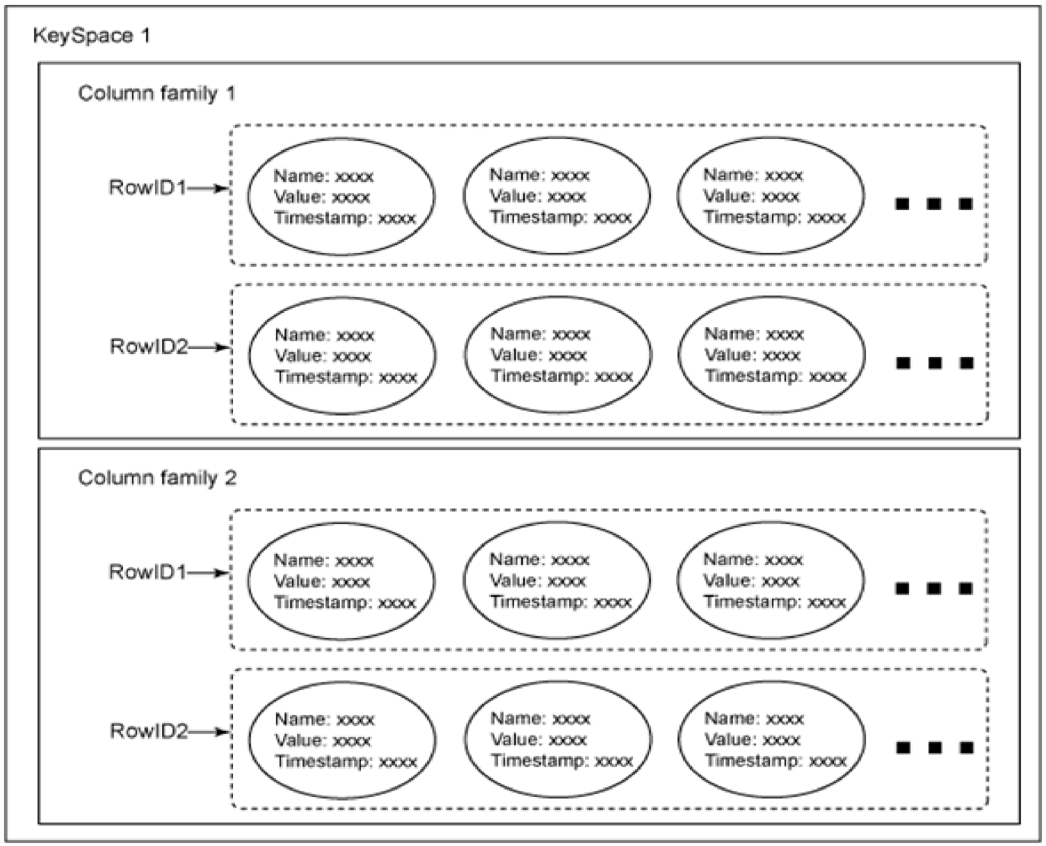
Example of Columns

Column names storing values

- key: User ID
- column names store tweet ID values
- values of all column names are set to “-” (empty byte array) as they are not used



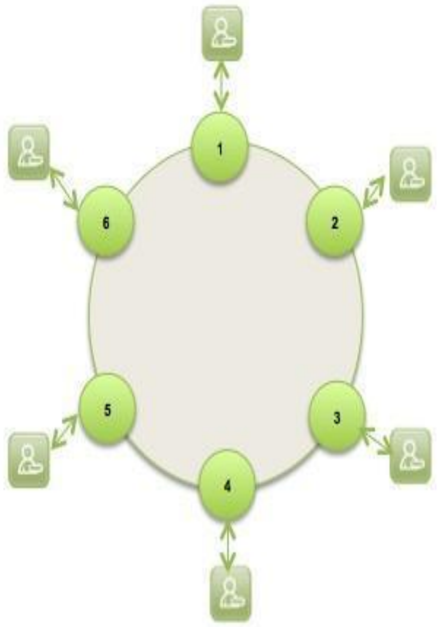
Key Space



- A Key Space is a group of column families together. It is only a logical grouping of column families and provides an isolated scope for names

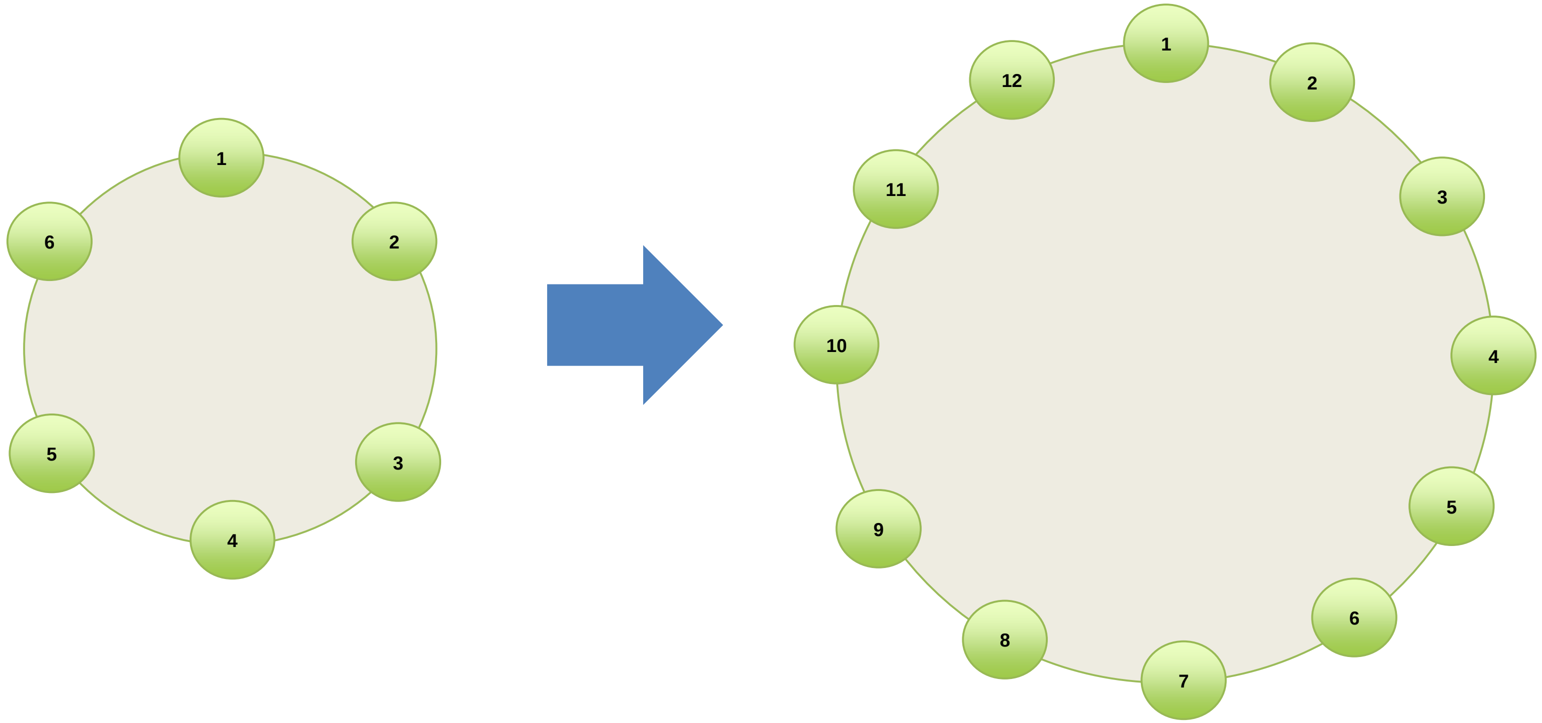
Cassandra Architecture Overview

- Cassandra was designed with the understanding that system/ hardware failures can and do occur
- Peer-to-peer, distributed system
- All nodes are the same
- Data partitioned among all nodes in the cluster
- Custom data replication to ensure fault tolerance
- Read/Write-anywhere design
- Google BigTable - data model
  - Column Families
  - Memtables
  - SSTables
- Amazon Dynamo - distributed systems technologies
  - Consistent hashing
  - Partitioning
  - Replication
  - One-hop routing



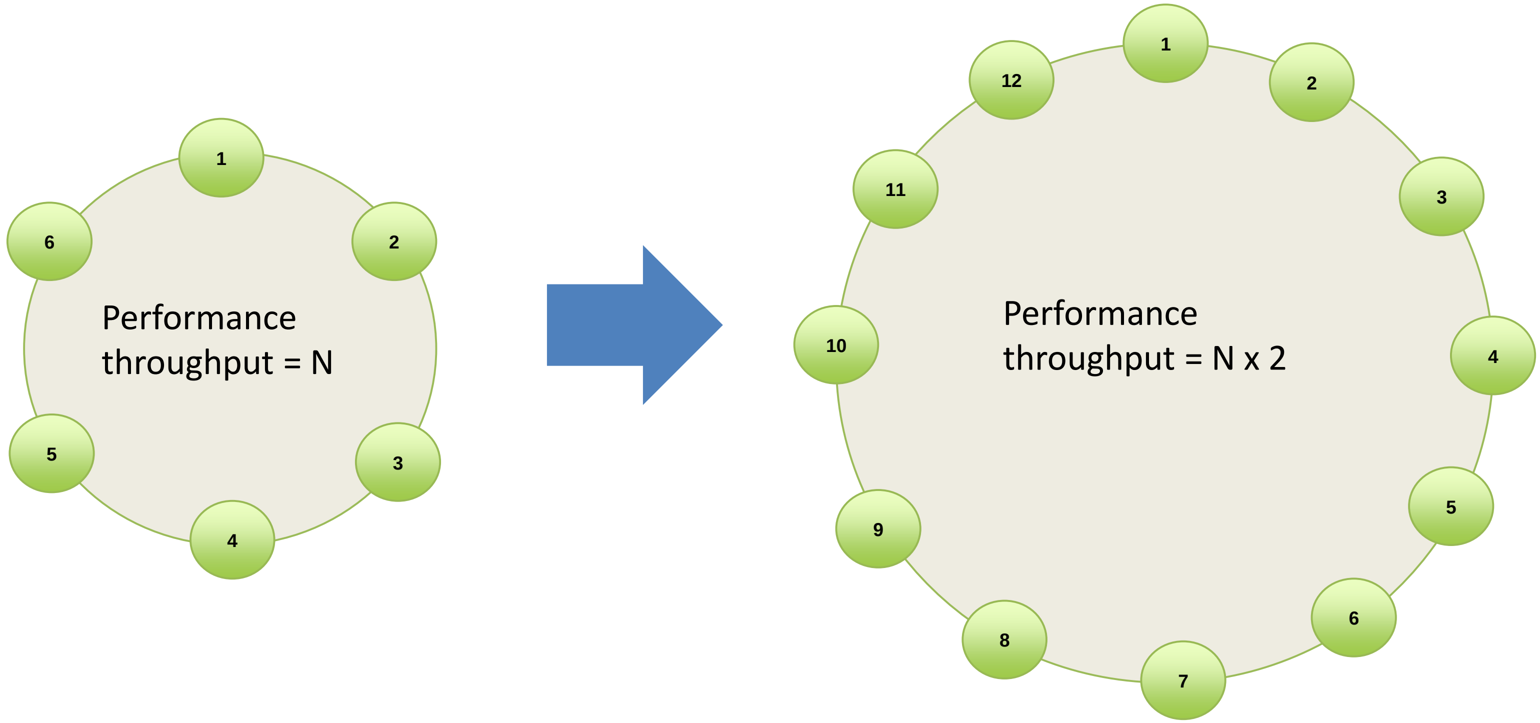
Transparent Elasticity

Nodes can be added and removed from Cassandra online, with no downtime being experienced.



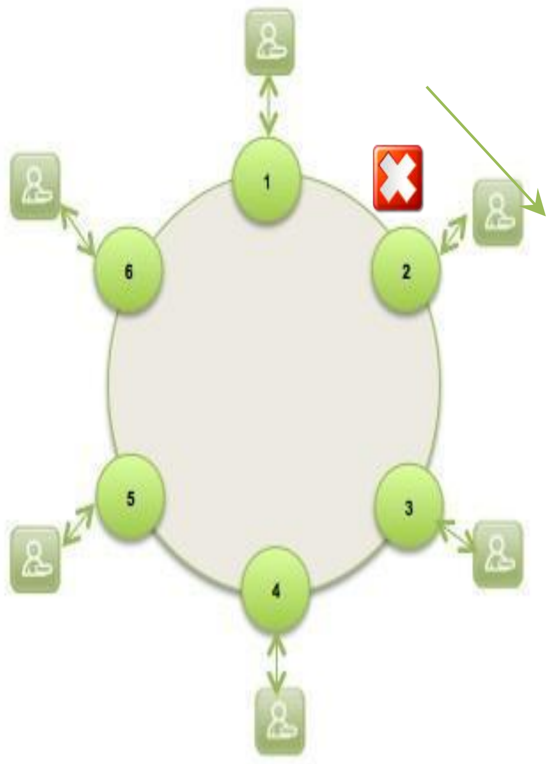
Transparent Scalability

Addition of Cassandra nodes increases performance linearly and ability to manage TB's-PB's of data.



High Availability

Cassandra, with its peer-to-peer architecture has no single point of failure.



Facebook Inbox Search

- **Cassandra developed to address this problem.**
- **50+TB of user messages data in 150 node cluster on which Cassandra is tested.**
- **Search user index of all messages in 2 ways.**
  - **Term search : search by a key word**
  - **Interactions search : search by a user id**

Latency Stat	Search Interactions	Term Search
Min	7.69 ms	7.78 ms
Median	15.69 ms	18.27 ms
Max	26.13 ms	44.41 ms

Cassandra Advantages

- perfect for time-series data
- high performance
- Decentralization
- nearly linear scalability
- replication support
- no single points of failure
- MapReduce support

Cassandra Weaknesses

- no referential integrity
  - no concept of JOIN
- querying options for retrieving data are limited
- sorting data is a design decision
  - no GROUP BY
- no support for atomic operations
  - if operation fails, changes can still occur
- first think about queries, then about data model

Cassandra: Points to Consider

- Cassandra is designed as a distributed database management system
  - use it when you have a lot of data spread across multiple servers
- Cassandra write performance is always excellent, but read performance depends on write patterns
  - it is important to spend enough time to design proper schema around the query pattern
- having a high-level understanding of some internals is a plus
  - ensures a design of a strong application built atop Cassandra

What is MongoDB?

- It is a NoSQL database
- It should be viewed as an alternative to relational databases
- It is a documented-oriented database
- It uses JSON format

The Basics

- Within a MongoDB instance you can have zero or more databases
- A database can have zero or more ‘collections’.
- Collections are made up of zero or more ‘documents’.
- A document is made up of one or more ‘fields’.
- ‘Indexes’ in MongoDB function much like their RDBMS counterparts.

MongoDB vs. RDBMS

- Collection vs. table
- Document vs. row
- Field vs. column
- Collection isn't strict about what goes in it (it's schema-less)

Why use MongoDB

- Simple queries
- Makes sense with most web applications
- Easier and faster the integration of data
- Not well suited for large-scale business applications b/c it does not support transactions.

Document store

RDBMS		MongoDB
Database	⇨	Database
Table, View	⇨	Collection
Row	⇨	Document (JSON, BSON)
Column	⇨	Field
Index	⇨	Index
Join	⇨	Embedded Document
Foreign Key	⇨	Reference
Partition	⇨	Shard

Document store

RDBMS		MongoDB
Database	⇨	Database
Table, View	⇨	Collection
Row	⇨	Document (
Column	⇨	Field
Index	⇨	Index
Join	⇨	Embedded D
Foreign Key	⇨	Reference
Partition	⇨	Shard

```
> db.user.findOne({age:39})
{
        "_id" : ObjectId("5114e0bd42…"),
        "first" : "John",
        "last" : "Doe",
        "age" : 39,
       "interests" : [
               "Reading",
               "Mountain Biking ]
       "favorites": {
              "color": "Blue",
              "sport": "Soccer"}
}
```

CRUD

- Create
  - db.collection.insert(<document>)
  - db.collection.save(<document>)
  - db.collection.update(<query>, <update>, { upsert: true })
- Read
  - db.collection.find(<query>, <projection>)
  - db.collection.findOne(<query>, <projection>)
- Update
  - db.collection.update(<query>, <update>, <options>)
- Delete
  - db.collection.remove(<query>, <justOne>)

CRUD example

```
> db.user.insert({
      first: "John",
      last : "Doe",
      age: 39
})
```

```
> db.user.find ()
{
      "_id" : ObjectId("51…"),
      "first" : "John",
      "last" : "Doe",
      "age" : 39
}
```

```
> db.user.update(
      {"_id" : ObjectId("51…")},
      {
            $set: {
                  age: 40,
                  salary: 7000}
      }
)
```

```
> db.user.remove({
      "first": /^J/
})
```

Features

- Document-Oriented storage
- Full Index Support
- Replication & High Availability
- Auto-Sharding
- Querying
- Fast In-Place Updates
- Map/Reduce

Agile

Scalable

In Produc

10gen | the MongoDB company

Firebase

sourceforge

Disney

SAP

Doodle

theguardian

bit.ly

SavingStar

foursquare

CollegeHumor

IGN

craigslist

High Five poker

the national archives

github SOCIAL CODING

The New York Times

http://www.mongodb.org/about/production-deployments/