

Chapter5

Using Big Data for Analytics

NoSQL

Basanta Joshi, PhD

Asst. Prof., Depart of Electronics and Computer Engineering
Deputy Director, MSc in Information and Communication Engineering
Member, Laboratory for ICT Research and Development (LICT)

Institute of Engineering

basanta@ioe.edu.np

<http://www.basantajoshi.com.np>

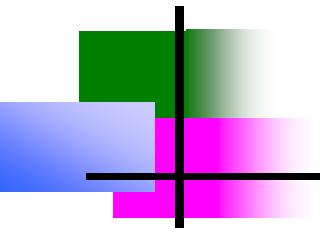
<https://scholar.google.com/citations?user=iocLiGcAAAAJ>

https://www.researchgate.net/profile/Basanta_Joshi2



CONTENTS

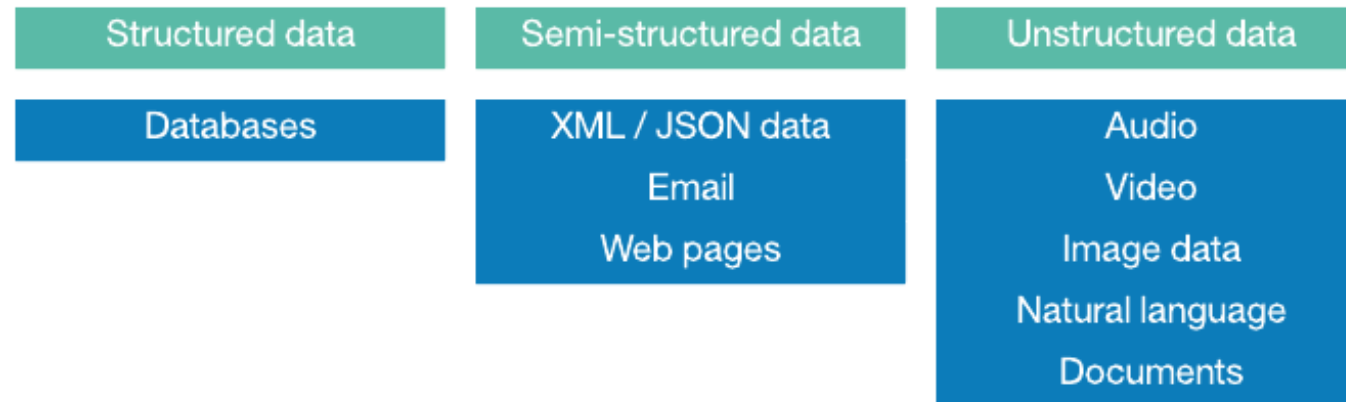
- Structured and unstructured data
- Taxonomy of NoSQL implementation
- Hbase
- Cassandra
- MongoDB



STRUCTURED AND UNSTRUCTURED DATA

Types of Data


- Structured
- Semi-structured
- Unstructured





STRUCTURED DATA

- “normal” RDBMS data
- Format is known and defined
- Example: Sales Order



[-]	[Table Icon]	Sales.SalesOrderHeader
[-]	[Folder Icon]	Columns
[Key Icon]		SalesOrderID (PK, int, not null)
[Table Icon]		RevisionNumber (tinyint, not null)
[Table Icon]		OrderDate (datetime, not null)
[Table Icon]		DueDate (datetime, not null)
[Table Icon]		ShipDate (datetime, null)
[Table Icon]		Status (tinyint, not null)
[Table Icon]		OnlineOrderFlag (Flag(bit), not null)
[Table Icon]		SalesOrderNumber (Computed, nvarchar(25), not null)
[Table Icon]		PurchaseOrderNumber (OrderNumber(nvarchar(25)), null)
[Table Icon]		AccountNumber (AccountNumber(nvarchar(15)), null)
[Key Icon]		CustomerID (FK, int, not null)
[Key Icon]		SalesPersonID (FK, int, null)
[Key Icon]		TerritoryID (FK, int, null)
[Key Icon]		BillToAddressID (FK, int, not null)
[Key Icon]		ShipToAddressID (FK, int, not null)
[Key Icon]		ShipMethodID (FK, int, not null)
[Key Icon]		CreditCardID (FK, int, null)
[Table Icon]		CreditCardApprovalCode (varchar(15), null)
[Key Icon]		CurrencyRateID (FK, int, null)
[Table Icon]		SubTotal (money, not null)
[Table Icon]		TaxAmt (money, not null)
[Table Icon]		Freight (money, not null)
[Table Icon]		TotalDue (Computed, money, not null)
[Table Icon]		Comment (nvarchar(128), null)
[Table Icon]		rowguid (uniqueidentifier, not null)
[Table Icon]		ModifiedDate (datetime, not null)



SEMI-STRUCTURED DATA

- some structure, but it is fluid
- changes in structure should not break code
- example: XML



SEMI STRUCTURED DATA

```
<SalesOrder DueDate="20120201">
  <OrderID>12</OrderID>
  <Customer>John Doe</Customer>
  <OrderDate>2012/01/15</OrderDate>
  Items
    <Item>
      <Product>Widget</Product>
      <Quantity>12</Quantity>
    </Item>
    <Item>
      <Product>Whatchamacallit</Product>
      <Quantity>2</Quantity>
    </Item>
  Items
</SalesOrder>
```



SEMI STRUCTURED DATA

```
<SalesOrder DueDate="20120201">
  <OrderID>12</OrderID>
  <Customer>John Doe</Customer>
  <OrderDate>2012/01/15</OrderDate>
  <Items>
    <Item>
      <Product>Widget</Product>
      <Quantity>12</Quantity>
    </Item>
    <Item>
      <Product>Whatchamacallit</Product>
      <Quantity>2</Quantity>
    </Item>
  </Items>
</SalesOrder>
```



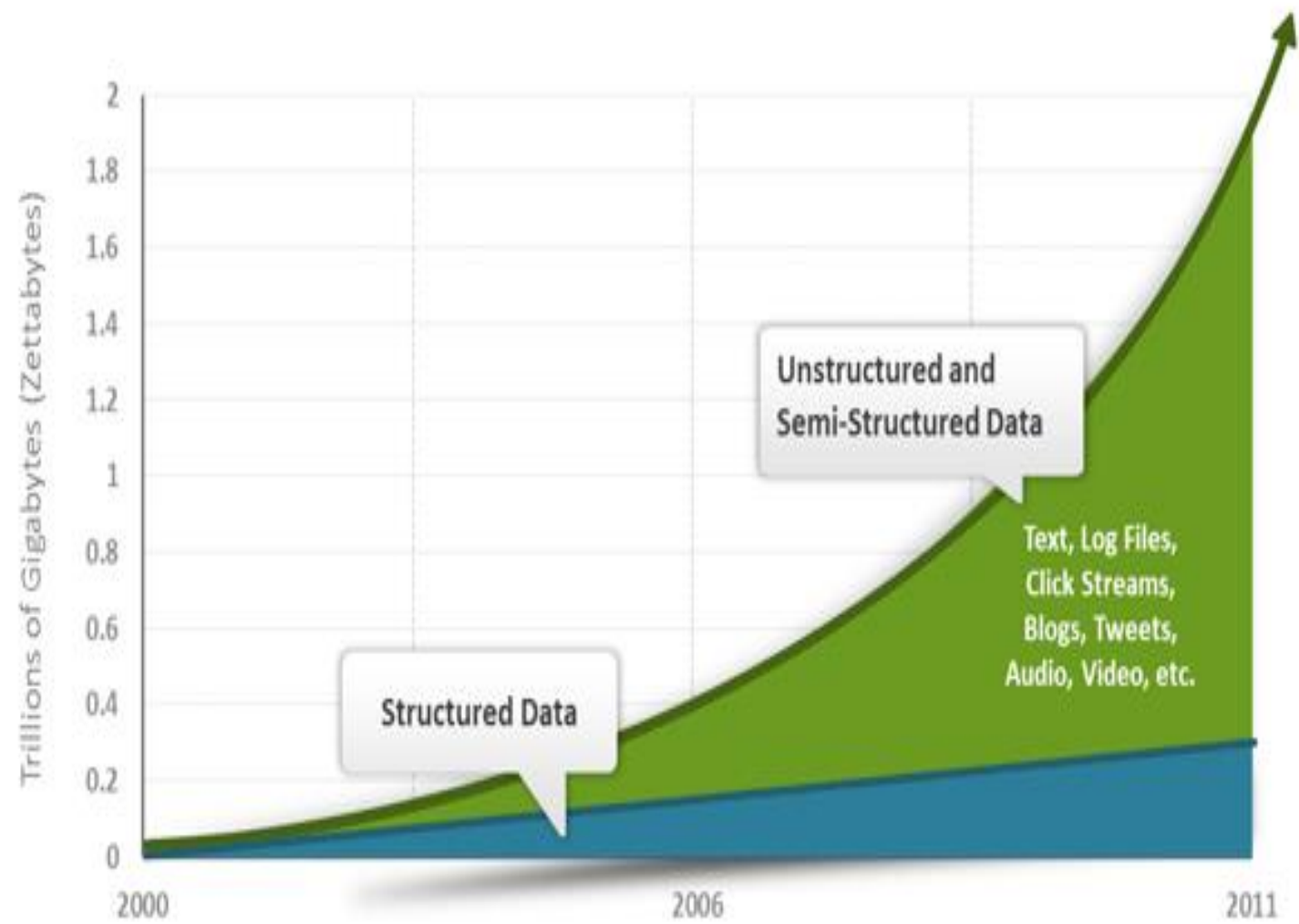

SEMI STRUCTURED DATA

```
<SalesOrder DueDate="20120201">
  <OrderID>12</OrderID>
  <Customer>John Doe</Customer>
  <OrderDate>2012/01/15</OrderDate>
  <Items>
    <Item>
      <Product>Widget</Product>
      <Quantity>12</Quantity>
    </Item>
    <Item>
      <Product>Whatchamacallit</Product>
      <Quantity>2</Quantity>
    </Item>
  </Items>
</SalesOrder>
```



UNSTRUCTURED DATA

- structure is merely encoding.
- meta data may be in the structure
- examples:
 - Audio files
 - Word Documents
 - PDF
 - Movies

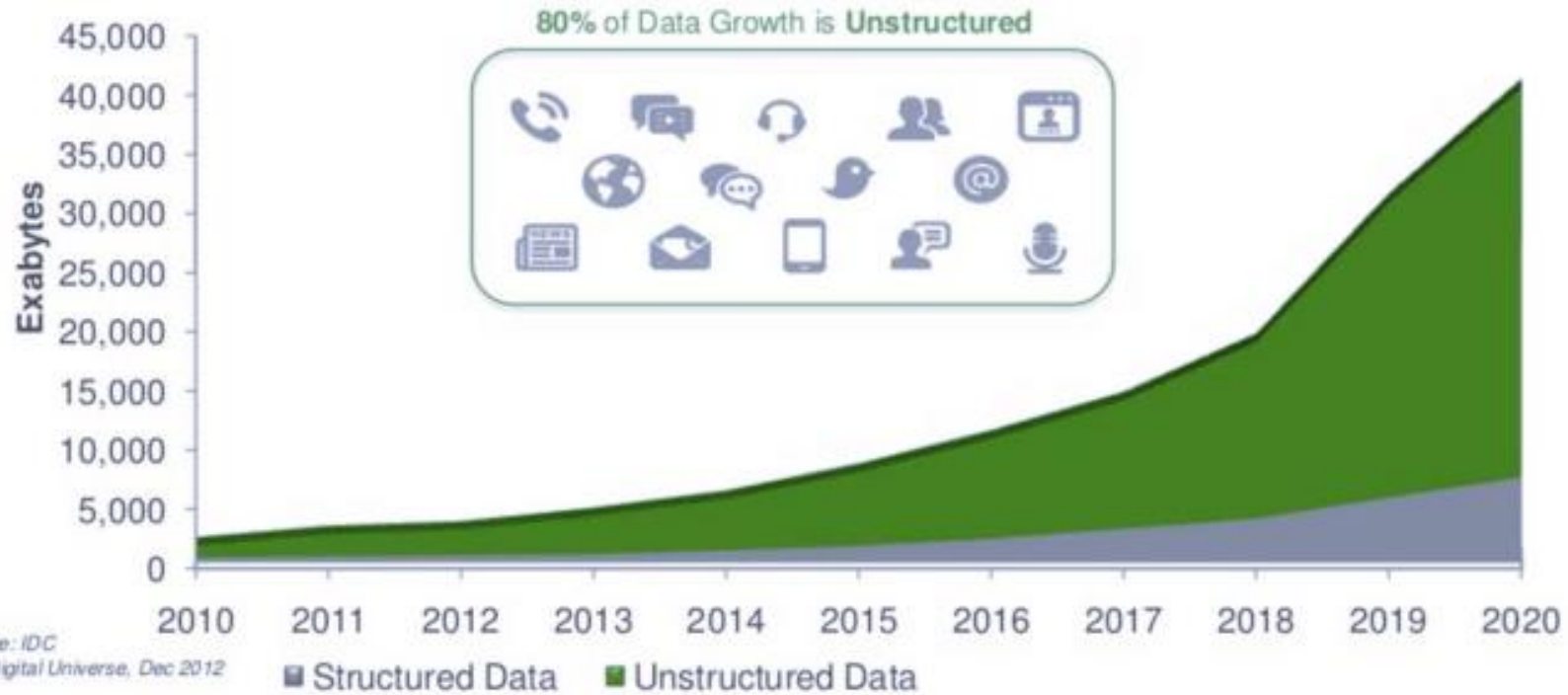


Source: IDC 2011 Digital Universe Study (<http://www.emc.com/collateral/demos/microsites/emc-digital-universe-2011/index.html>)

MASSIVE GROWTH IN UNSTRUCTURED CONTENT

RECOMMIND

Worldwide Corporate Data Growth





NOSQL

- Overview of NoSQL databases
- The need of NoSQL databases
- “Battle” between SQL and NoSQL databases
- CAP Theorem
- Challenges of NoSQL databases



NOSQL: AN OVERVIEW OF NOSQL DATABASES

- the term means Not Only SQL
- It's not SQL and it's not relational. NoSQL is designed for distributed data stores for very large scale data needs.
- In a NoSQL database, there is no fixed schema and no joins. Scaling out refers to spreading the load over many commodity systems. This is the component of NoSQL that makes it an inexpensive solution for large datasets.

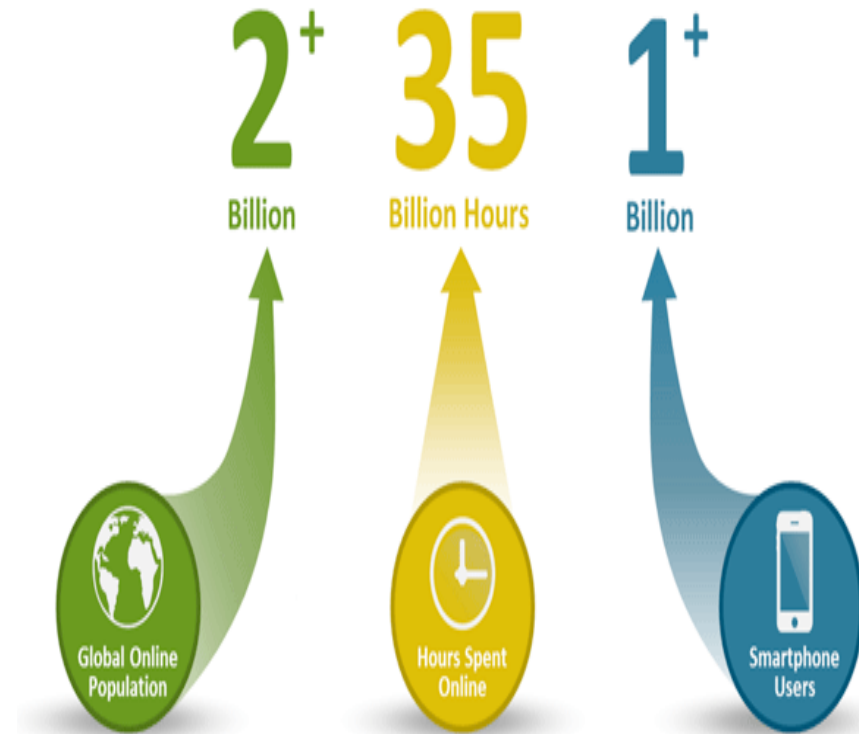


QUERYING NOSQL

- The question of how to query a NoSQL database is what most developers are interested in. After all, data stored in a huge database doesn't do anyone any good if you can't retrieve and show it to end users or web services. NoSQL databases do not provide a high level declarative query language like SQL. Instead, querying these databases is data-model specific.
- Many of the NoSQL platforms allow for RESTful interfaces to the data. Other offer query APIs. There are a couple of query tools that have been developed that attempt to query multiple NoSQL databases. These tools typically work accross a single NoSQL category.
- One example is [SPARQL](#). SPARQL is a declarative query specification designed for graph databases.
- Here is an example of a SPARQL query that retrieves the URL of a particular blogger (courtesy of [IBM](#)):
- ```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
SELECT ?url
FROM <bloggers.rdf>
WHERE {
 ?contributor foaf:name "Jon Foobar" .
 ?contributor foaf:weblog ?url .
}
```

# WHY NOSQL?

- **Three trends disrupting the database status quo—**
  - Big Data,
  - Big Users, and
  - Cloud Computing
- **Big Users** :Not that long ago, 1,000 daily users of an application was a lot and 10,000 was an extreme case.
- Today, with the growth in global Internet use, the **increased number of hours** users spend online, and the **growing popularity of smartphones and tablets**, it's not uncommon for apps to have **millions of users** a day.







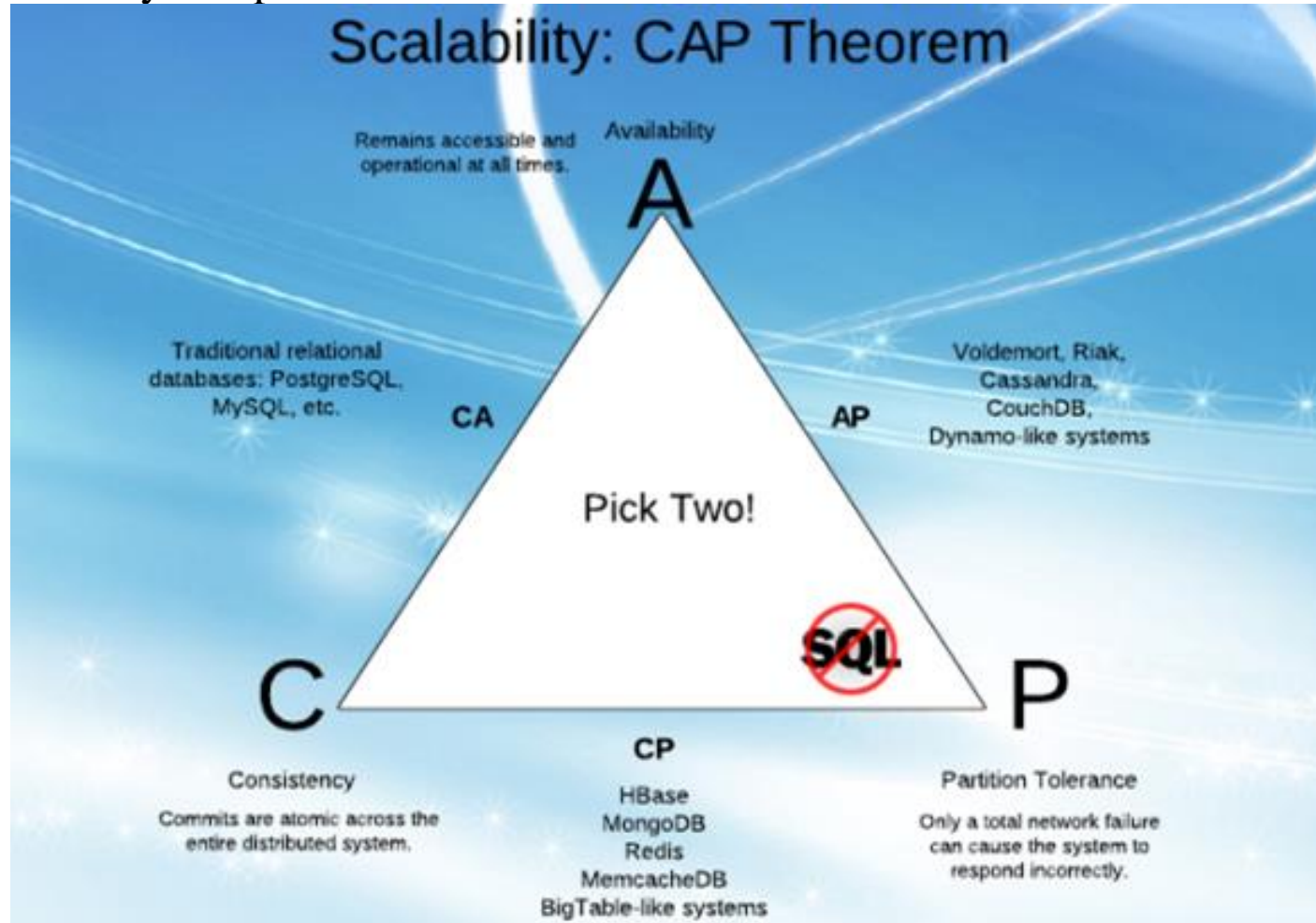
# WHY NOSQL?

---

- ***Big Data*** :Data is becoming easier to capture and access through third parties such as Facebook, D&B, and others. Personal user information, geo location data, social graphs, user-generated content, machine logging data, and sensor-generated data are just a few examples of the ever-expanding array of data being captured.
- ***Cloud Computing***: Today, most new applications (both consumer and business) use a three-tier Internet architecture, run in a public or private cloud, and support large numbers of users.

# WHAT IS CAP?

- The CAP Theorem states that it is impossible for any shared-data system to *guarantee* simultaneously all of the following three properties: consistency, availability and partition tolerance.





# WHAT IS CAP?

---

- Consistency in CAP is not the same as consistency in ACID (that would be too easy). According to CAP, consistency in a database means that whenever data is written, everyone who reads from the database will always see the latest version of the data. A database without strong consistency means that when the data is written, not everyone who reads from the database will see the new data right away; this is usually called eventual-consistency or weak consistency.
- Availability in a database according to CAP means you always can expect the database to be there and respond whenever you query it for information. High availability usually is accomplished through large numbers of physical servers acting as a single database through sharing (splitting the data between various database nodes) and replication (storing multiple copies of each piece of data on different nodes).
- Partition tolerance in a database means that the database still can be read from and written to when parts of it are completely inaccessible. Situations that would cause this include things like when the network link between a significant number of database nodes is interrupted. Partition tolerance can be achieved through some sort of mechanism whereby writes destined for unreachable nodes are sent to nodes that are still accessible. Then, when the failed nodes come back, they receive the writes they missed



# NOSQL: TAXONOMY OF IMPLEMENTATION

---

The current NoSQL world fits into 4 basic categories.

- **Key-values Stores**
- **Column Family Stores** were created to store and process very large amounts of data distributed over many machines. There are still keys but they point to multiple columns. In the case of [BigTable](#) (Google's Column Family NoSQL model), rows are identified by a row key with the data sorted and stored by this key. The columns are arranged by column family. E.g. **Cassandra**
- **Document Databases** were inspired by [Lotus Notes](#) and are similar to key-value stores. The model is basically versioned documents that are collections of other key-value collections. The semi-structured documents are stored in formats like [JSON](#) e.g. [MongoDB](#)
- **Graph Databases** are built with nodes, relationships between nodes and the properties of nodes. Instead of tables of rows and columns and the rigid structure of SQL, a flexible graph model is used which can scale across many machines.



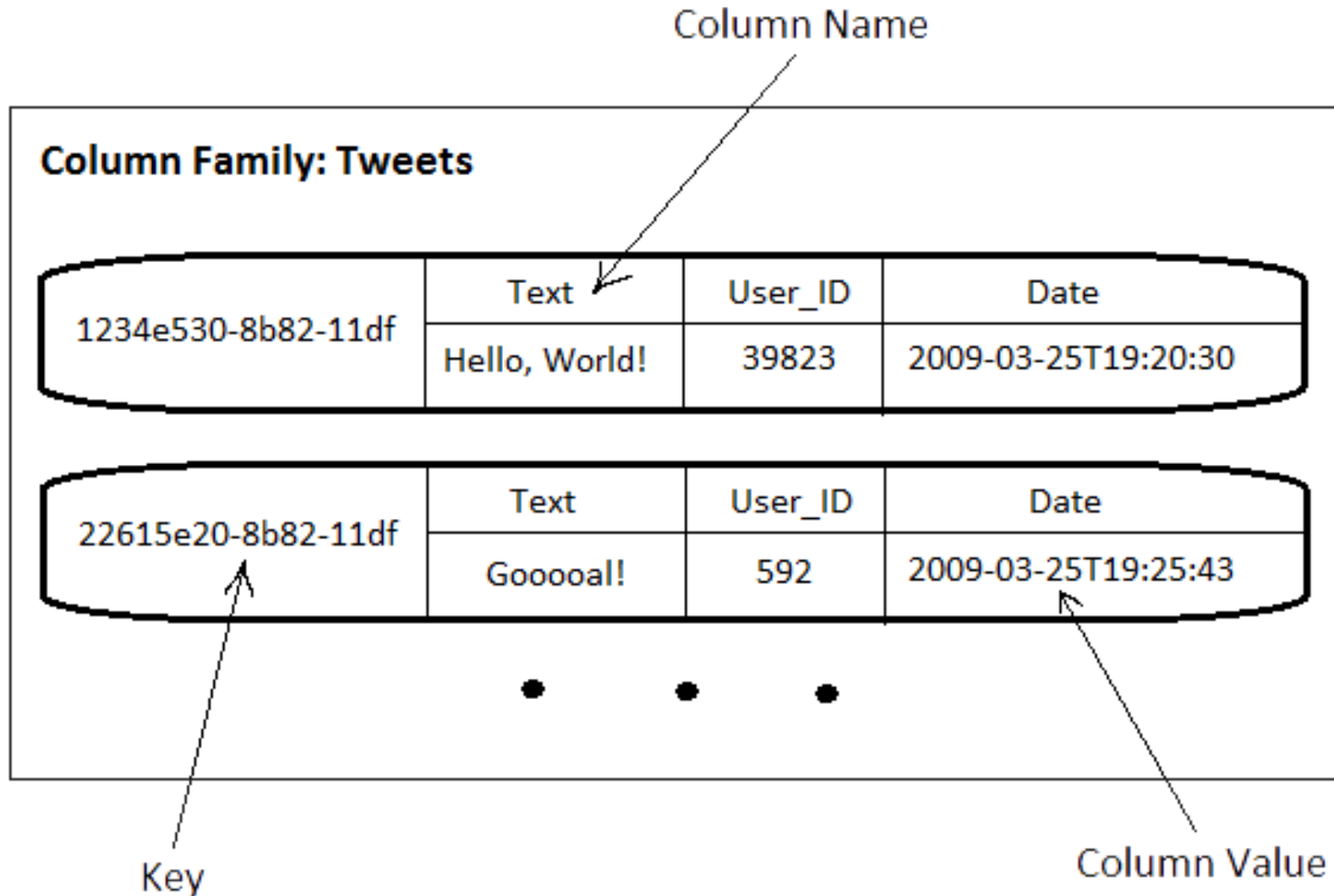
# KEY-VALUES STORES

---

- The Key-Value database is a very simple structure based on Amazon's Dynamo DB. Data is indexed and queried based on its key. Key-value stores provide consistent hashing so they can scale incrementally as your data scales. They communicate node structure through a gossip-based membership protocol to keep all the nodes synchronized. If you are looking to scale very large sets of low complexity data, key-value stores are the best option.
- **Examples**
  - [Riak](#)
  - [Voldemort](#)
  - Tokyo Cabinet

| key       | value |
|-----------|-------|
| firstName | Bugs  |
| lastName  | Bunny |
| location  | Earth |

# COLUMN FAMILY STORES





# COLUMN FAMILY STORES

---

- These data stores are based on Google's BigTable implementation.
- may look similar to relational databases on the surface but under the hood a lot has changed.
- A column family database can have different columns on each row so is not relational and doesn't have what qualifies in an RDBMS as a table.
- The only key concepts in a column family database are columns, column families and super columns.
- Column families define how the data is structured on disk. A column by itself is just a key-value pair that exists in a column family.
- A super column is like a catalogue or a collection of other columns except for other super columns.
- Column family databases are still extremely scalable but less-so than key-value stores. However, they work better with more complex data sets.
- **Examples of Column family databases are:**
  - [Apache HBase](#)
  - [Hypertable](#)
  - [Cassandra](#)



# COLUMN FAMILY STORES

---

- Equivalent to “relational table” – “set of rows” identified by key
- Concept starts with columns
- Data is organized in the columns
- Columns are stored contiguously
- Columns tend to have similar data
- A row can have as many columns as needed
- Columns are essentially keys, that can let you lookup values in the rows
- Columns can be added any time, NULL don't exist
- Unused columns in a row does not occupy storage
- Write Data to Append Only file, an extremely efficient and makes write are significantly faster then write
- Built in control about replication and distribution
- HBase :From CAP Theorem
  - Partition Tolerance
  - Consistency
- Cassandra :From CAP Theorem
  - Partition Tolerance
  - Availability Note: Tradeoff between availability and consistency is tunable





# DOCUMENT DATABASES

---

- A document database is not a new idea. It was used to power one of the more prominent communication platforms of the 90's and still in service today, Lotus Notes now called Lotus Domino. APIs for document DBs use Restful web services and JSON for message structure making them easy to move data in and out.
- A document database has a fairly simple data model based on collections of key-value pairs.
- A typical record in a document database would look like this:
  - { "Subject": "I like Plankton"  
"Author": "Rusty"  
"PostedDate": "5/23/2006"  
"Tags": ["plankton", "baseball", "decisions"]  
"Body": "I decided today that I don't like baseball. I like plankton." }
- Document databases improve on handling more complex structures but are slightly less scalable than column family databases.
- **Examples of document databases are:**
  - [CouchDB](#)
  - [MongoDB](#)

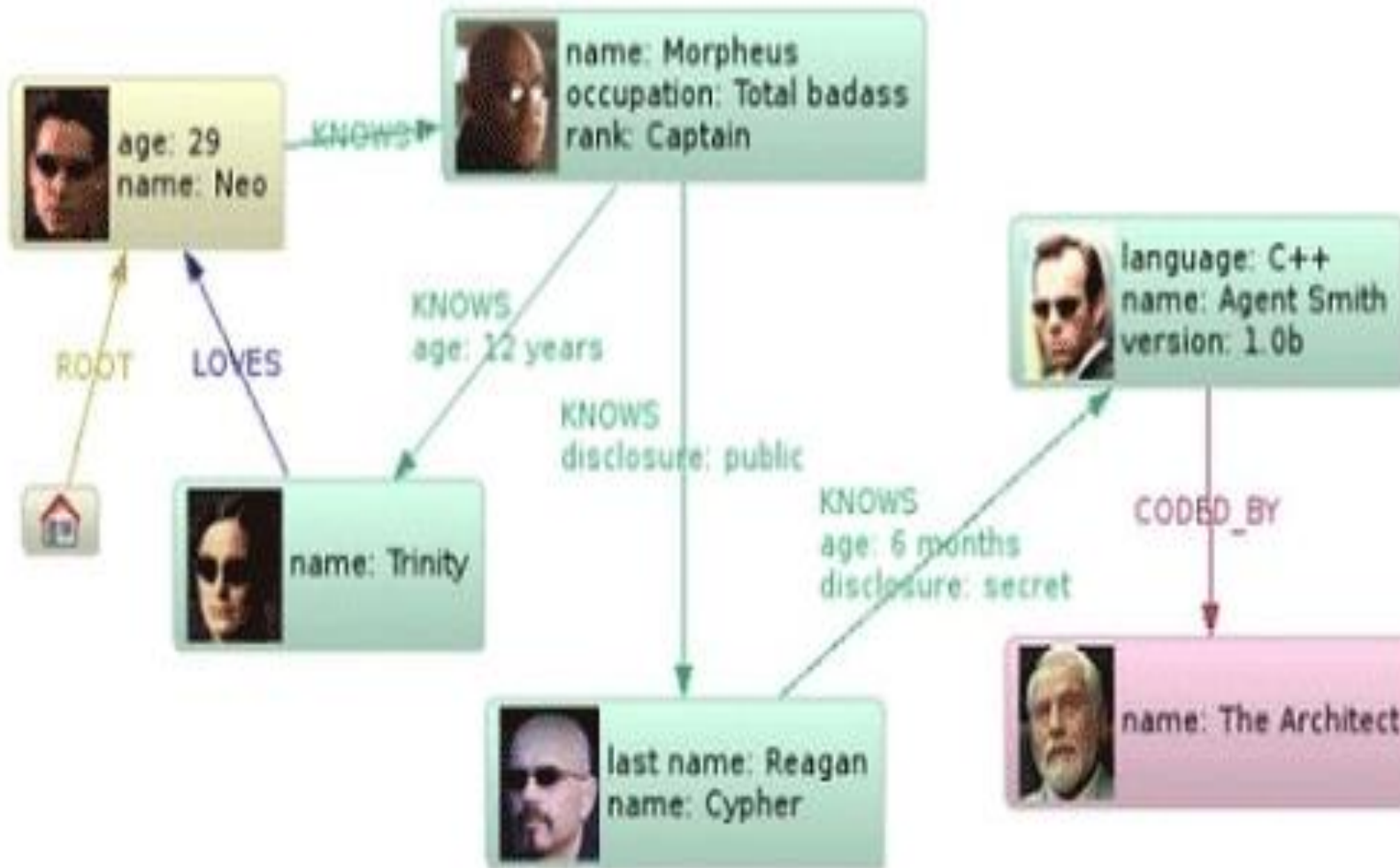


# DOCUMENT DATABASES

---

- Documents are key-value pair
- document can also be stored in JSON format
- Because of JSON document considered as object
- JSON documents are used as Key-Value pairs
- Document can have any set of keys
- Any key can associate with any arbitrarily complex value, that is itself a JSON document
- Documents are added with different sets of keys
  - Missing keys
  - Extra keys
  - Add keys in future when in need
  - Application must know that certain key present
  - Queries are made on Keys
  - Index are set to keys to make search efficient

# GRAPH DATABASES





# GRAPH DATABASES

---

- Graph databases take document databases to the extreme by introducing the concept of type relationships between documents or nodes. The most common example is the relationship between people on a social network such as Facebook. The idea is inspired by the graph theory work by Leonhard Euler, the 18th century mathematician. Key/Value stores used key-value pairs as their modeling units. Column Family databases use the tuple with attributes to model the data store. A graph database is a big dense network structure.
- While it could take an RDBMS hours to sift through a huge linked list of people, a graph database uses sophisticated shortest path algorithms to make data queries more efficient. Although slower than its other NoSQL counterparts, a graph database can have the most complex structure of them all and still traverse billions of nodes and relationships with light speed.
- **Examples of Graph Databases are:**
  - [AllegroGraph](#)
  - [Sones](#)
  - [Neo4j](#)



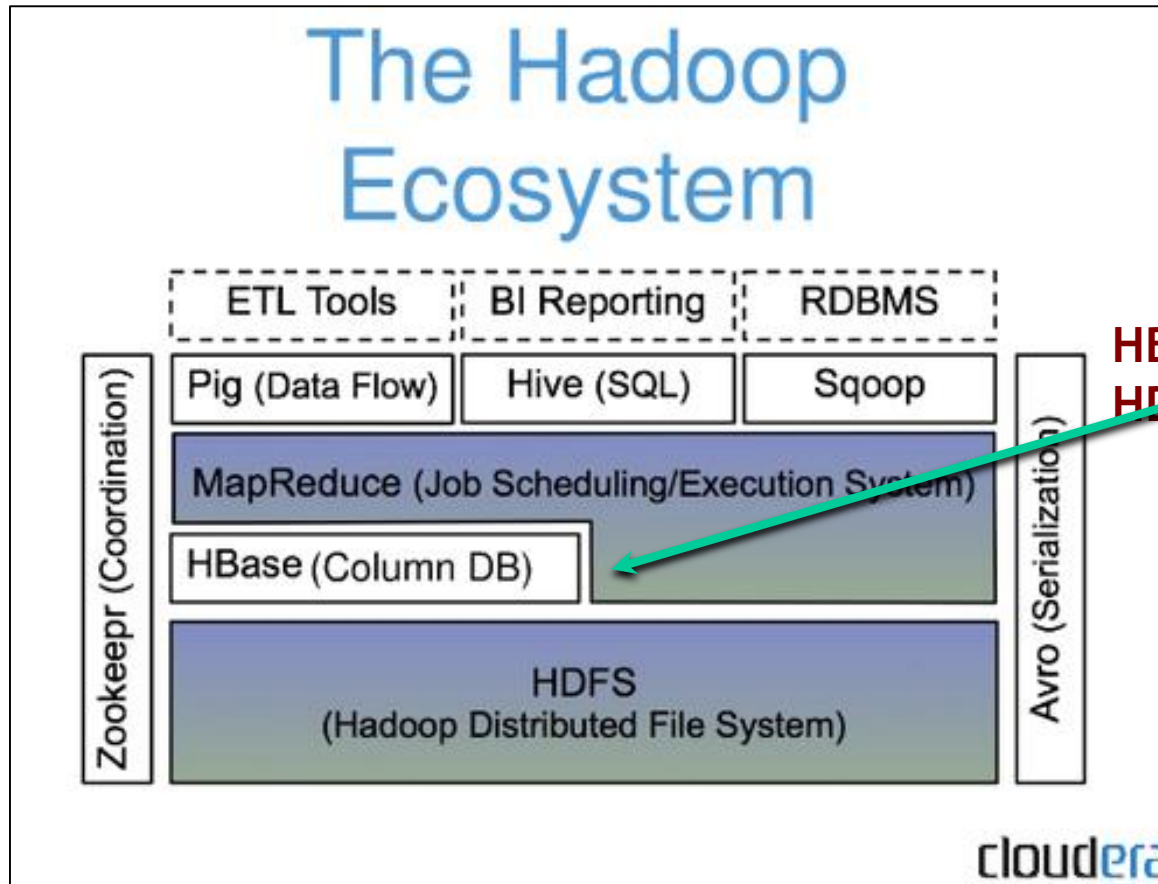
# HBASE: OVERVIEW

---

- 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



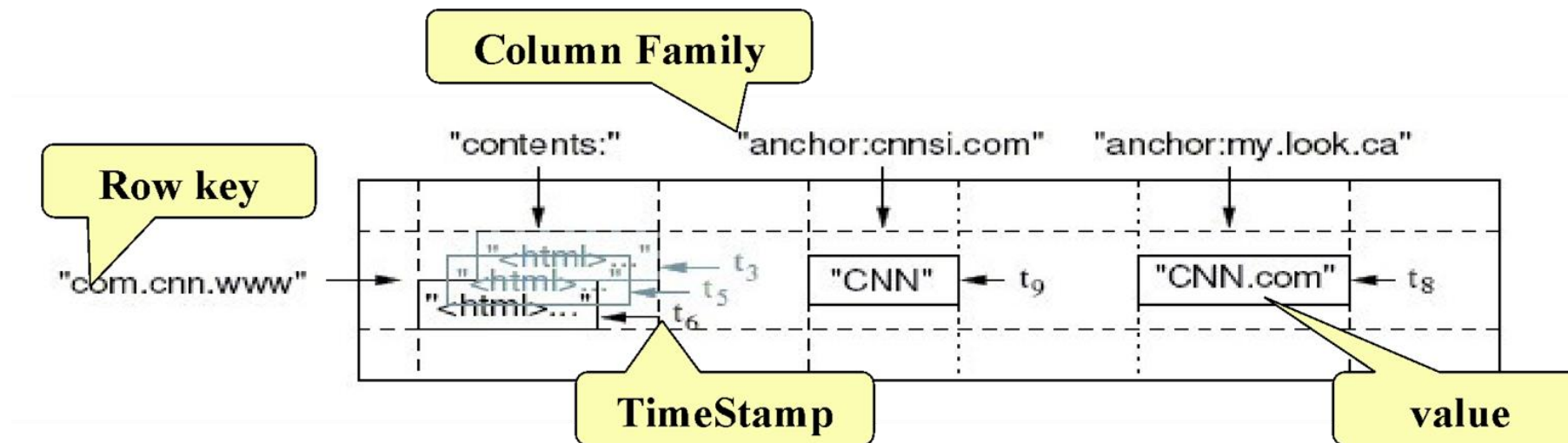
# HBASE VS. HDFS (CONT'D)

---

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

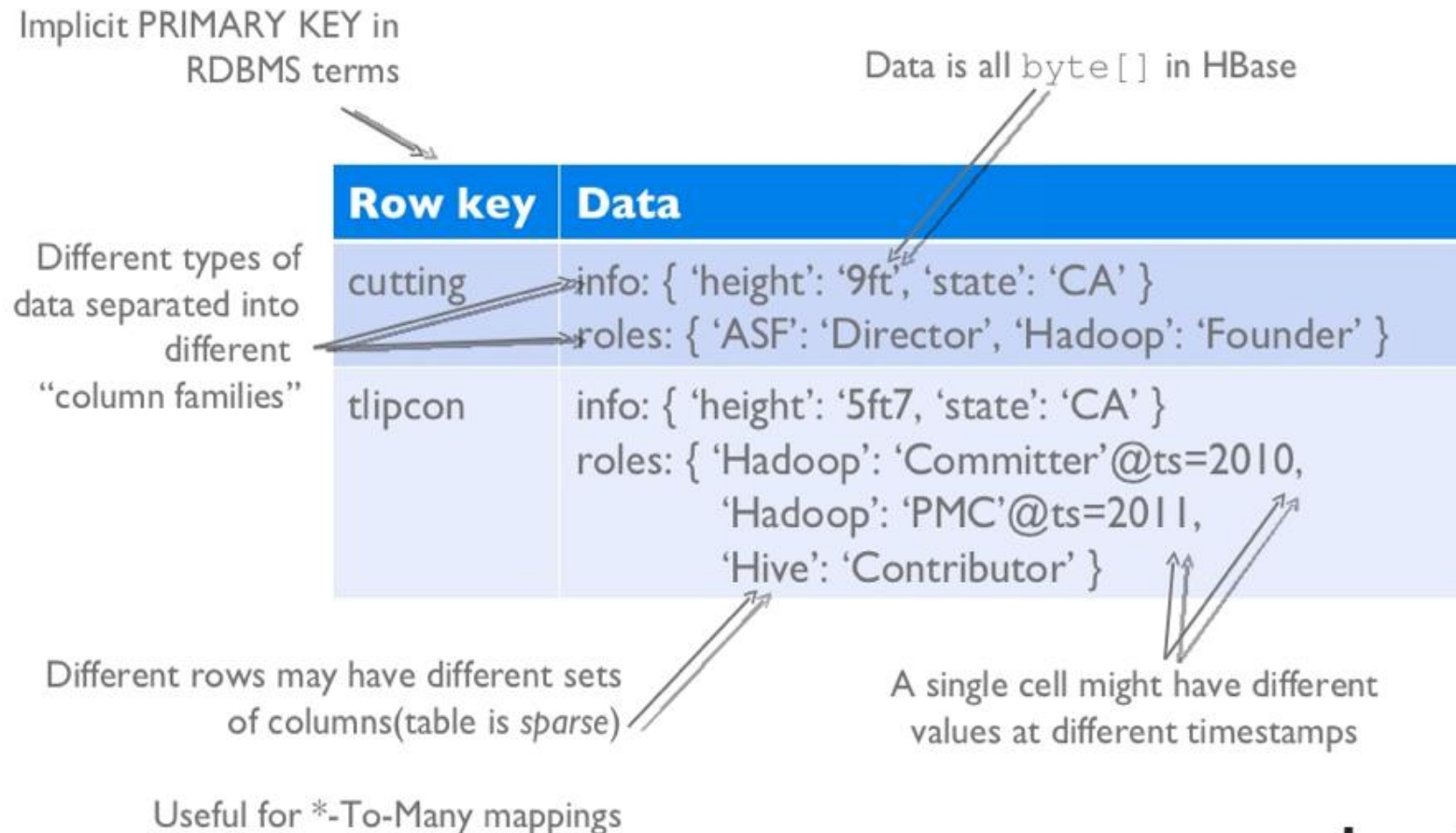
# HBASE DATA MODEL

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





# HBASE LOGICAL VIEW



# HBASE: KEYS AND COLUMN FAMILIES

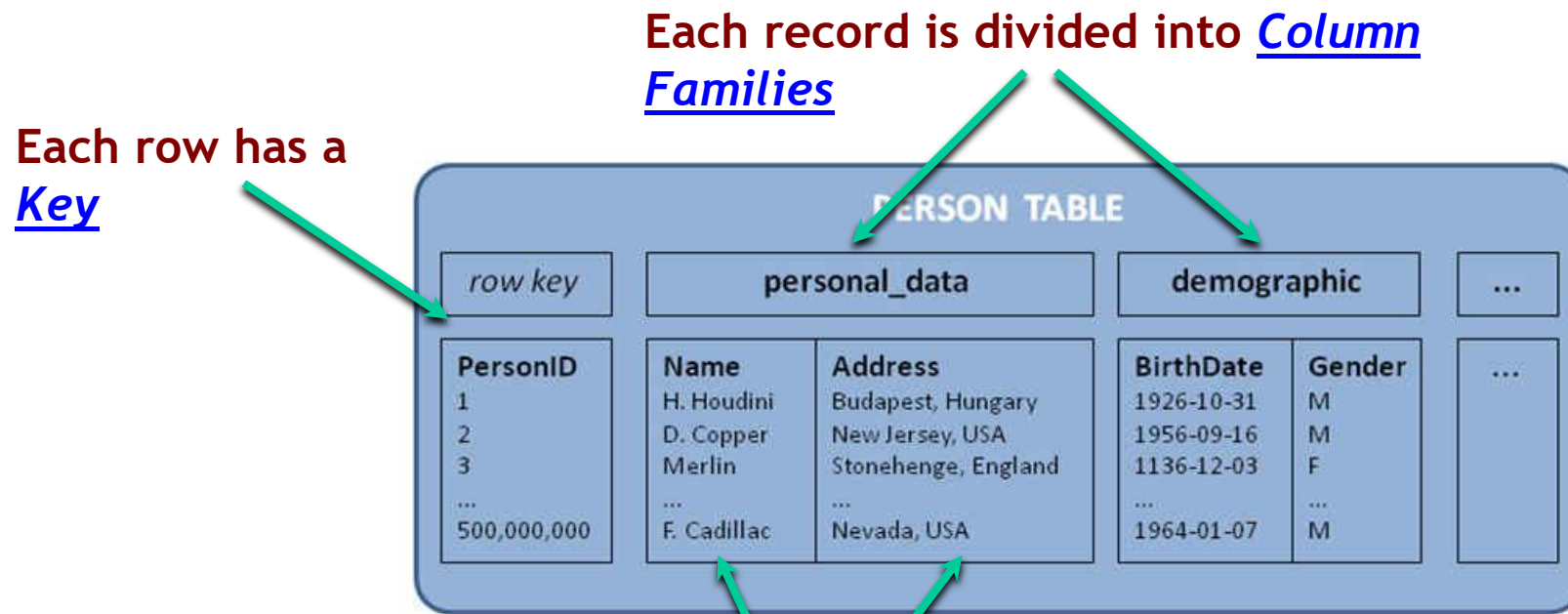


Figure 2. Census Data in Column Families

Each column family consists of one or more Columns

# HBASE: KEYS AND COLUMN FAMILIES

- **Key**
  - Byte array
  - Serves as the primary key for the table
  - Indexed for 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 "Contents" "anchor"

| 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:cnn.com" "CNN"        |
|                  | t13        |                    | "anchor:my.look.ca" "CNN.com" |
|                  | t6         | "<html>..."        |                               |
|                  | t5         | "<html>..."        |                               |
|                  | t3         | "<html>..."        |                               |

Column named "apache.com"



# HBASE: KEYS AND COLUMN FAMILIES

Version number for each row

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

| Row key          | Time Stamp | Column "content s." | Column "anchor:"    |           |
|------------------|------------|---------------------|---------------------|-----------|
| "com.apache.www" | t12        | "<html> ..."        |                     | value     |
|                  | t11        | "<html> ..."        |                     |           |
|                  | t10        |                     | "anchor:apache.com" | "APACHE"  |
| "com.cnn.www"    | t15        |                     | "anchor:cnn.com"    | "CNN"     |
|                  | t13        |                     | "anchor:my.look.ca" | "CNN.com" |
|                  | t6         | "<html> ..."        |                     |           |
|                  | t5         | "<html> ..."        |                     |           |
|                  | t3         | "<html> ..."        |                     |           |

# HBASE 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

“Roles” column family has different columns in different cells



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

# HBASE DATA MODEL

- 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 s

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' }<br>roles: { 'ASF': 'Director', 'Hadoop': 'Founder' }                                               |
| tlipcon | info: { 'height': '5ft7', 'state': 'CA' }<br>roles: { 'Hadoop': 'Committer'@ts=2010,<br>'Hadoop': 'PMC'@ts=2011,<br>'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*

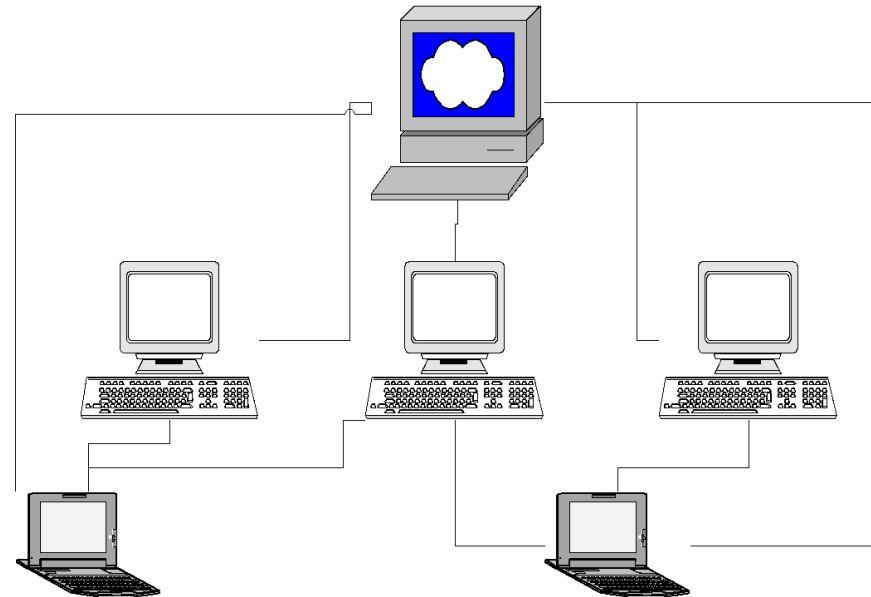


---

# HBASE ARCHITECTURE

# THREE MAJOR COMPONENTS

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



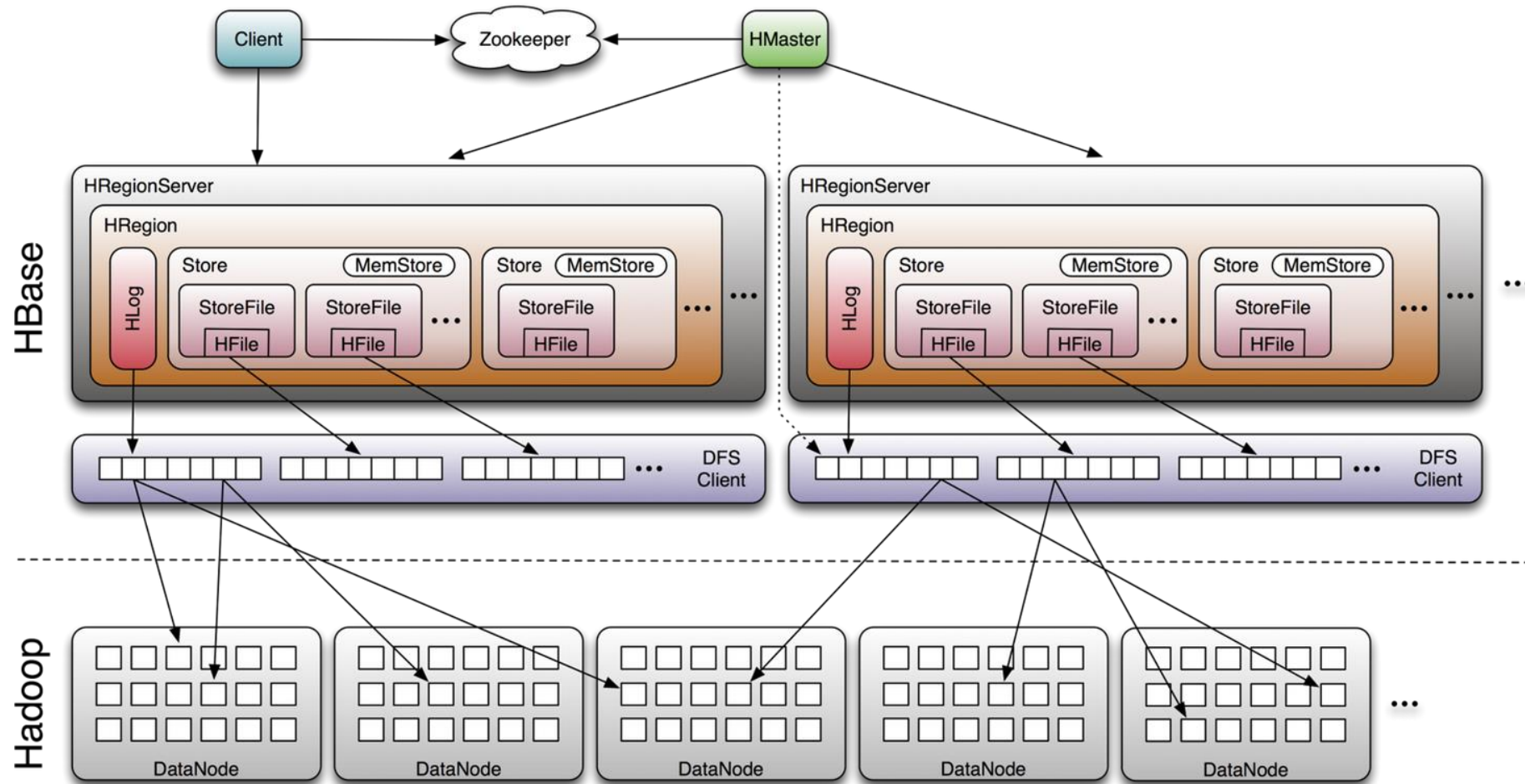


# HBASE COMPONENTS

---

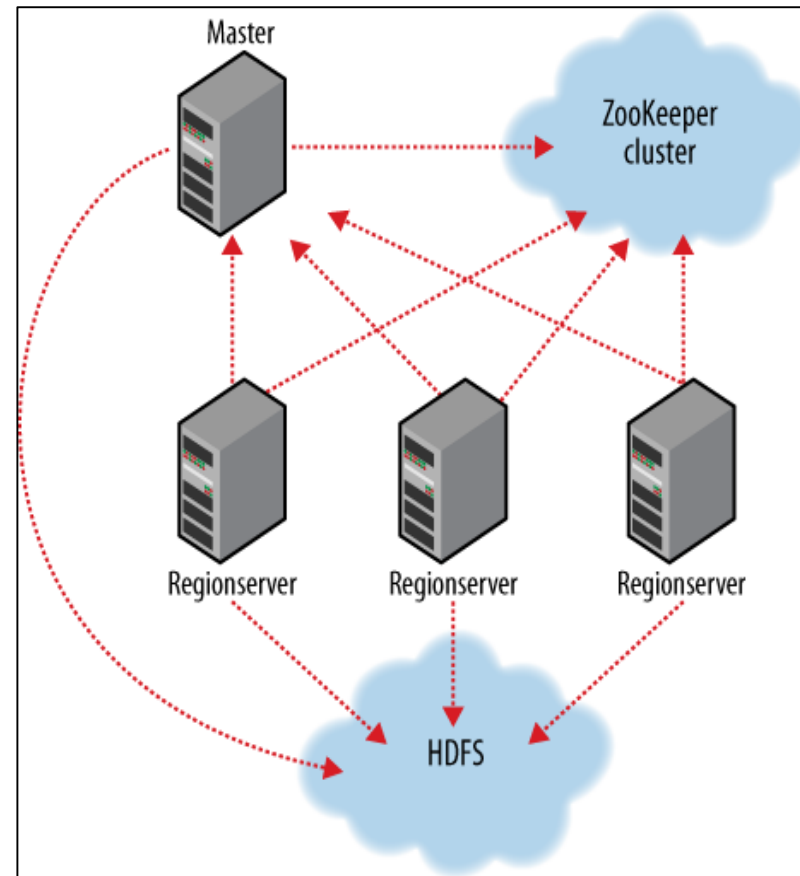
- **Region**
  - A subset of a table's rows, like horizontal range partitioning
  - Automatically done
- **RegionServer (many slaves)**
  - Manages data regions
  - Serves data for reads and writes (*using a log*)
- **Master**
  - Responsible for coordinating the slaves
  - Assigns regions, detects failures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

# BIG PICTURE



# 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 Example r.getValue(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: **SCAN()**

---

```
HTable htable = ... // instantiate HTable

Scan scan = new Scan();
scan.addColumn(Bytes.toBytes("cf"), Bytes.toBytes("attr"));
scan.setStartRow(Bytes.toBytes("row")); // start key is inclusive
scan.setStopRow(Bytes.toBytes("row" + (char)0)); // stop key is exclusive
ResultScanner rs = htable.getScanner(scan);
try {
 for (Result r = rs.next(); r != null; r = rs.next()) {
 // process result...
 } finally {
 rs.close(); // always close the ResultScanner!
 }
}
```



# GET()

Select value from table where  
key='com.apache.www' AND  
label='anchor:apache.com'

---

| Row key          | Time Stamp | Column "anchor:"    |                 |
|------------------|------------|---------------------|-----------------|
| "com.apache.www" | t12        |                     |                 |
|                  | t11        |                     |                 |
|                  | t10        | "anchor:apache.com" | <b>"APACHE"</b> |
| "com.cnn.www"    | t9         | "anchor:cnnsi.com"  | "CNN"           |
|                  | t8         | "anchor:my.look.ca" | "CNN.com"       |
|                  | t6         |                     |                 |
|                  | t5         |                     |                 |
|                  | t3         |                     |                 |



# SCAN()

Select value from table  
where  
anchor='cnnsi.com'

| Row key          | Time Stamp | Column "anchor:"    |           |
|------------------|------------|---------------------|-----------|
| "com.apache.www" | t12        |                     |           |
|                  | t11        |                     |           |
|                  | t10        | "anchor:apache.com" | "APACHE"  |
| "com.cnn.www"    | t9         | "anchor:cnnsi.com"  | "CNN"     |
|                  | t8         | "anchor:my.look.ca" | "CNN.com" |
|                  | t6         |                     |           |
|                  | t5         |                     |           |
|                  | t3         |                     |           |



# OPERATIONS ON REGIONS: PUT()

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

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

**Implicit version  
number  
(timestamp)**



```
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);
```

**Explicit version  
number**





# OPERATIONS ON REGIONS:

## DELETE()

---

- Marking table cells as deleted
- **Multiple levels**
  - Can mark an entire column family as deleted
  - Can make all column families of a given row as deleted

- All operations are logged by the RegionServers
- The log is flushed periodically



# HBASE: JOINS

---

- HBase does not support joins
- Can be done in the application layer
  - Using scan() and get() operations



# ALTERING A TABLE

---

```
Configuration config = HBaseConfiguration.create();
HBaseAdmin admin = new HBaseAdmin(conf);
String table = "myTable";

admin.disableTable(table);

HColumnDescriptor cf1 = ...;
admin.addColumn(table, cf1); // adding new ColumnFamily
HColumnDescriptor cf2 = ...;
admin.modifyColumn(table, cf2); // modifying existing ColumnFamily

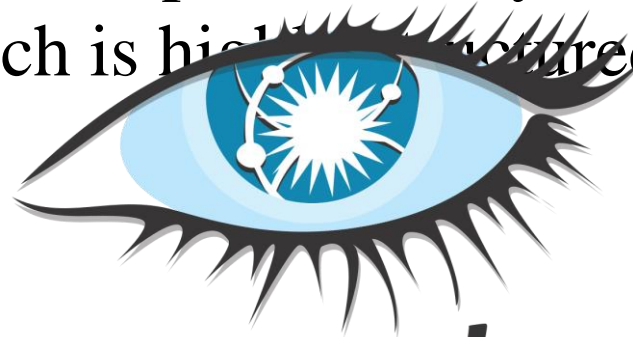
admin.enableTable(table);
```

Disable the table before changing the schema

6.1. Schema Creation

# CASSANDRA: DATA MODEL

A table in Cassandra is a distributed multi dimensional map indexed by a key. The value is an object which is highly structured.



***cassandra***

Cassandra exposes two kinds of columns families

Simple column families

Super column families





# STRUCTURE

---

- keyspace
  - settings (eg, partitioner)
- column family
  - settings (eg, comparator, type [Std])
- column
  - name
- value
- clock



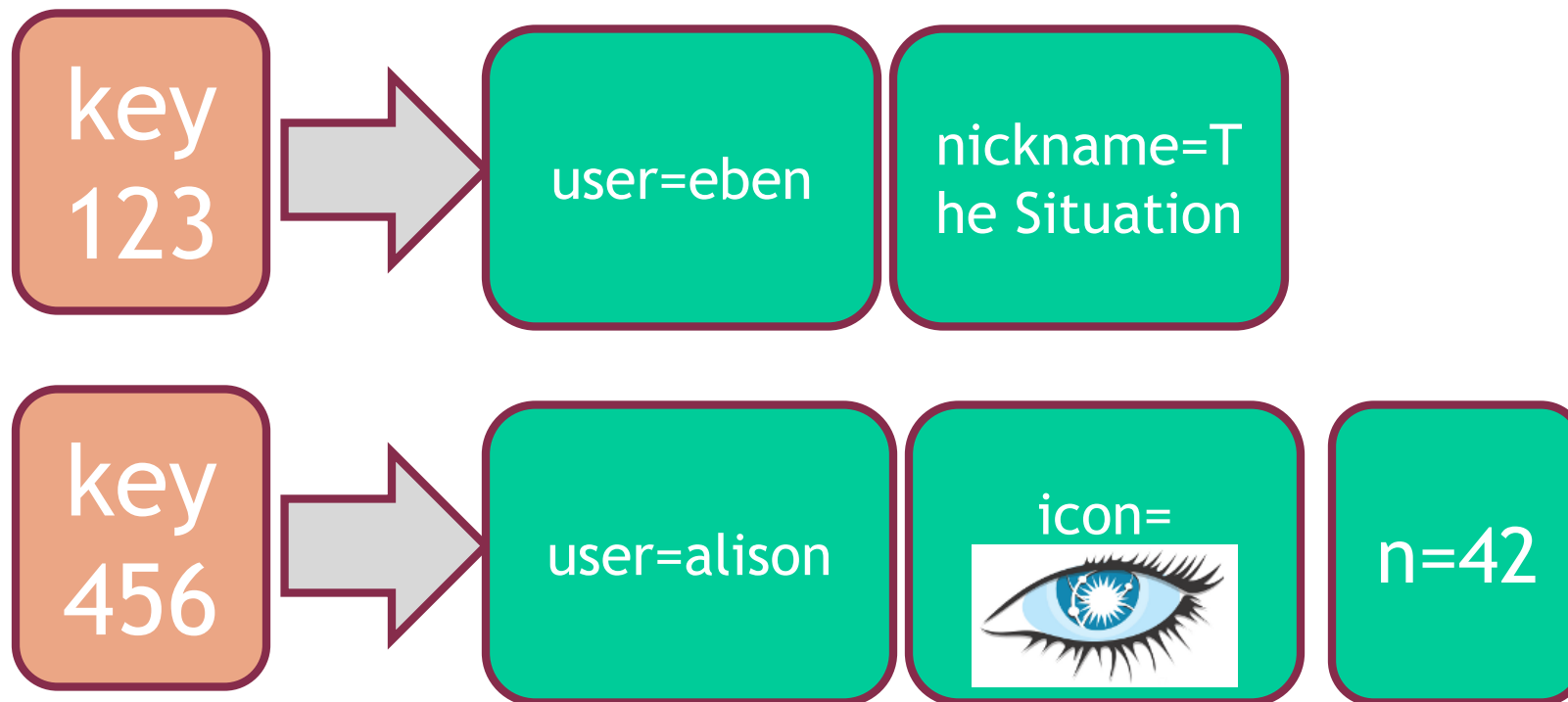
# COLUMN FAMILY

---

- group records of *similar* kind
- not *same* kind, because CFs are **sparse tables**
- ex:
  - User
  - Address
  - Tweet
  - PointOfInterest
  - HotelRoom

# CASSANDRA

- **THINK OF CASSANDRA AS ROW-ORIENTED**
  - each row is uniquely identifiable by key
  - rows group columns and super columns
- COLUMN FAMILY



# JSON-LIKE NOTATION

---

```
User {
 123 : { email: alison@foo.com,
 i  },
 456 : { email: eben@bar.com,
 location: The Danger Zone}
}
```



## 0.6 EXAMPLE

---

```
$cassandra -f
```

```
$bin/cassandra-cli
```

```
cassandra> connect localhost/9160
```

```
cassandra> set Keyspace1.Standard1['eben']['age']='29'
```

```
cassandra> set
```

```
 Keyspace1.Standard1['eben']['email']='e@e.com'
```

```
cassandra> get Keyspace1.Standard1['eben']['age']
```

```
=> (column=6e616d65, value=39,
 timestamp=1282170655390000)
```



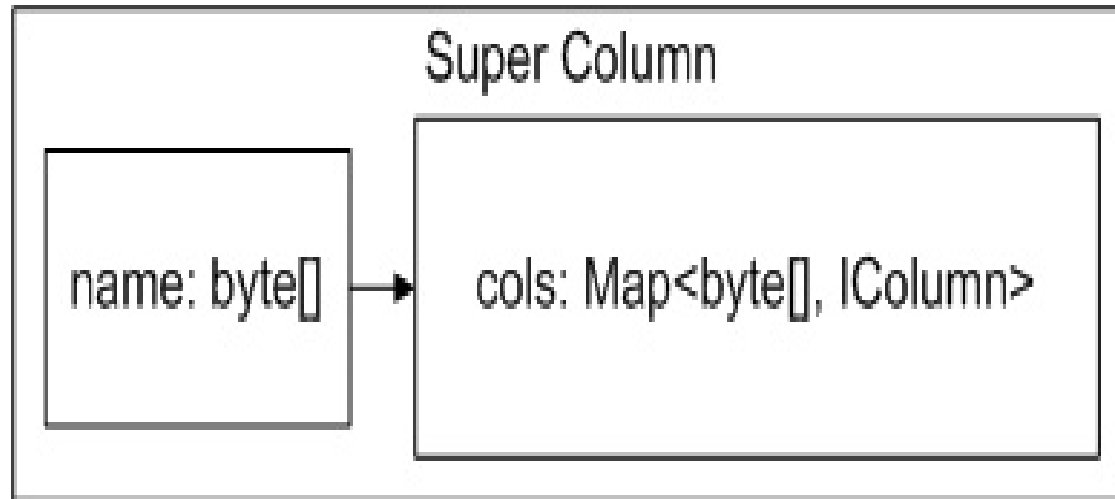
# A COLUMN HAS 3 PARTS

---

1. name
  - byte[]
  - determines sort order
  - used in queries
  - indexed
2. value
  - byte[]
  - *you don't query on column values*
3. timestamp
  - long (clock)
  - last write wins conflict resolution

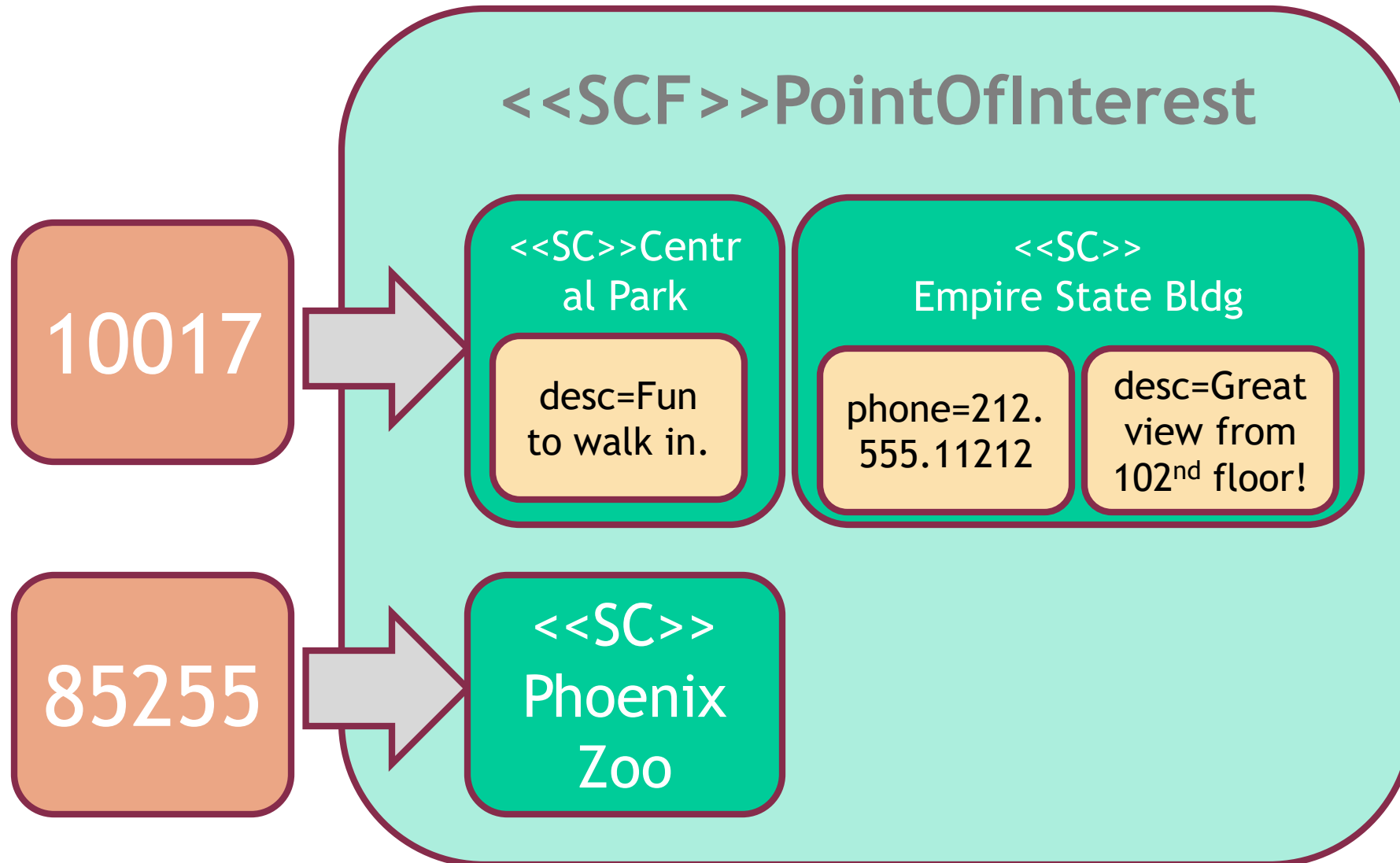
# SUPER COLUMN

- super columns group columns under a common name



- sub-column names in a SCF are *not* indexed
  - top level columns (SCF Name) are *always* indexed
- often used for **denormalizing** data from standard CFs

# SUPER COLUMN FAMILY





# SUPER COLUMN FAMILY

PointOfInterest {

key: 85255 {

Phoenix Zoo { phone: 480-555-5555, desc: They have animals here. },

Spring Training { phone: 623-333-3333, desc: Fun for baseball fans. },  
}, //end phx

key: 10019 {

Central Park { desc: Walk around. It's pretty. },

Empire State Building { phone: 212-777-7777,

desc: Great view from 102nd floor. }

} //end nyc

}

super column family

column

key

super column

flexible schema



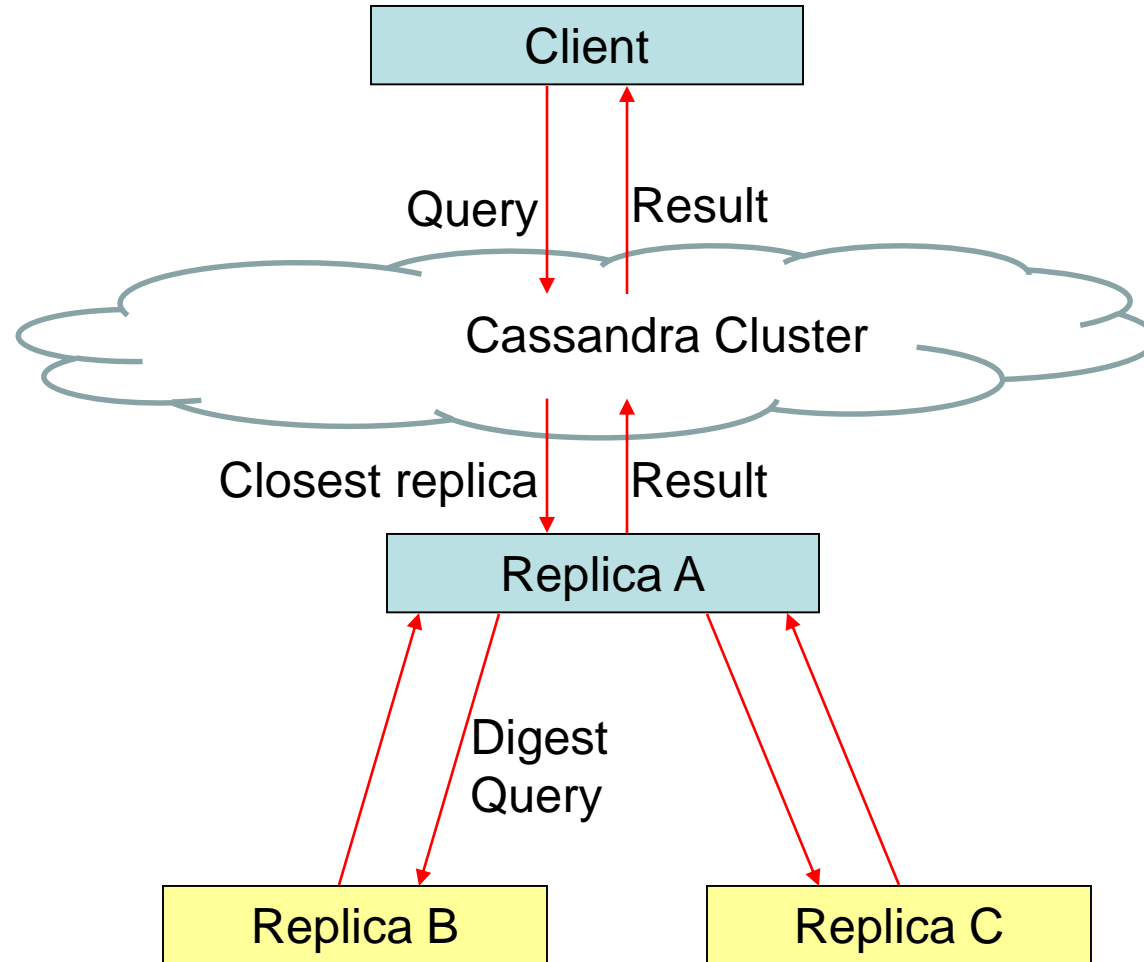
# API

---

The Cassandra API consists of the following three simple methods.

- `insert(table; key; rowMutation)`
- `get(table; key; columnName)`
- `delete(table; key; columnName)`

# ARCHITECTURE



- Built from the start to solve the scaling problem
- Consistency, Availability, Partitioning
  - (can't have it all)
- Configurable to fit requirements
- *MongoDB is an open source, document-oriented database designed with both scalability and developer agility in mind.*
- *Instead of storing data in tables and rows like as relational database, in MongoDB store JSON-like documents with dynamic schemas(schema-free, schemaless).*

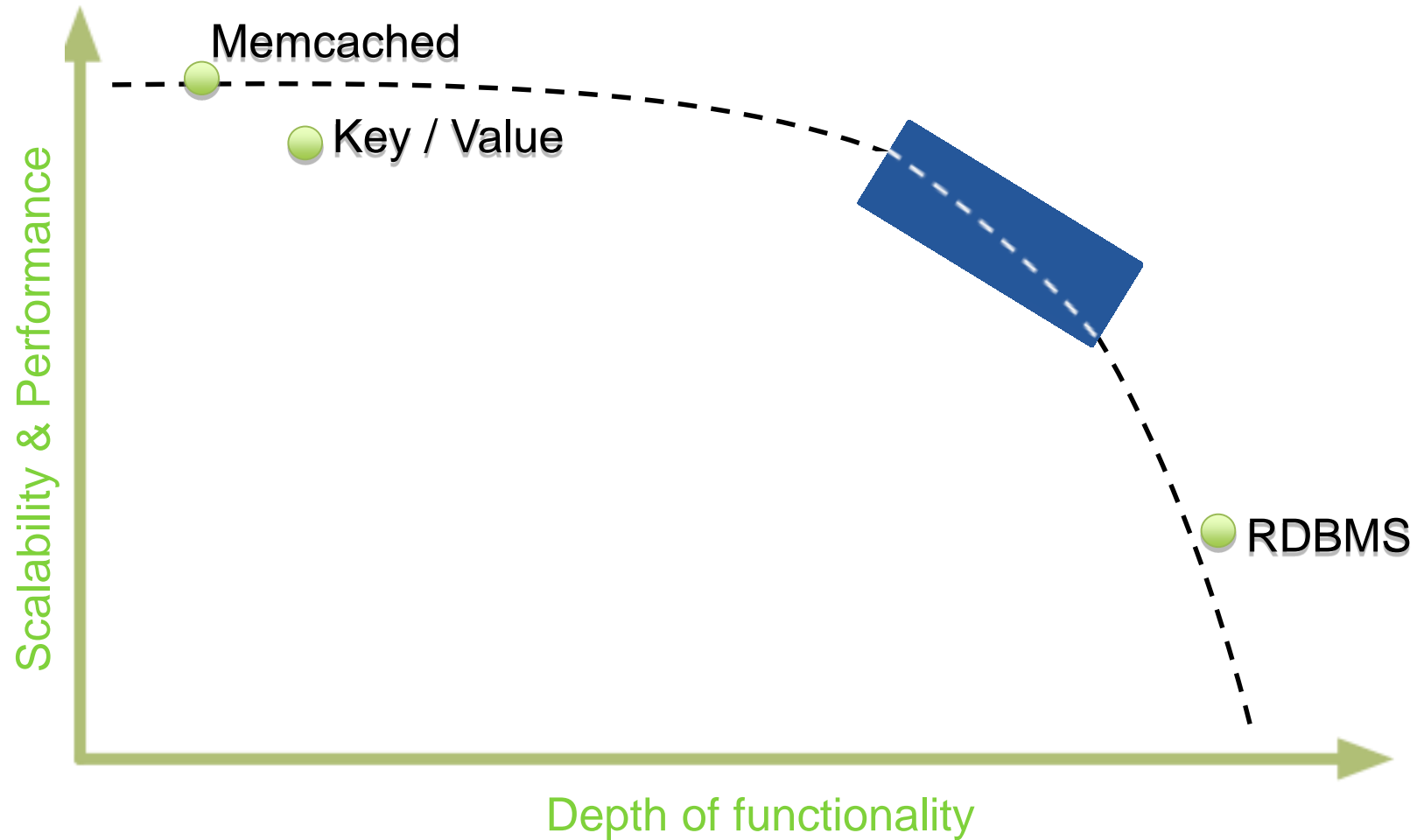


# MangoDB (Introduction)

---

- **Written in:** C++
- **Main point:** Retains some friendly properties of SQL
- **License:** AGPL (Drivers: Apache)
- Master/slave replication (auto failover with replica sets)
- Sharding built-in
- Queries are javascript expressions
- Run arbitrary javascript functions server-side
- Better update-in-place than CouchDB
- Uses memory mapped files for data storage
- An empty database takes up 192Mb
- GridFS to store big data + metadata (not actually an FS)

# AS SIMPLE AS POSSIBLE, BUT NO SIMPLER





# Data Model

---

- **Data model:** Using BSON (binary JSON), developers can easily map to modern object-oriented languages without a complicated ORM layer.
- BSON is a binary format in which zero or more key/value pairs are stored as a single entity.
- lightweight, traversable, efficient.

|                                                |   |                                                                                                                                                    |
|------------------------------------------------|---|----------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>{"hello": "world"}</code>                | → | <code>"\x16\x00\x00\x00\x02hello\x00\x06\x00\x00\x00world\x00\x00"</code>                                                                          |
| <code>{"BSON": ["awesome", 5.05, 1986]}</code> | → | <code>"1\x00\x00\x00\x04BSON\x00&amp;\x00\x00\x00\x020\x00\x08\x00\x00\x00awesome\x00\x011\x00333333\x14@\x102\x00\xc2\x07\x00\x00\x00\x00"</code> |

# Supported Languages







---

# REPRESENTING & QUERYING DATA

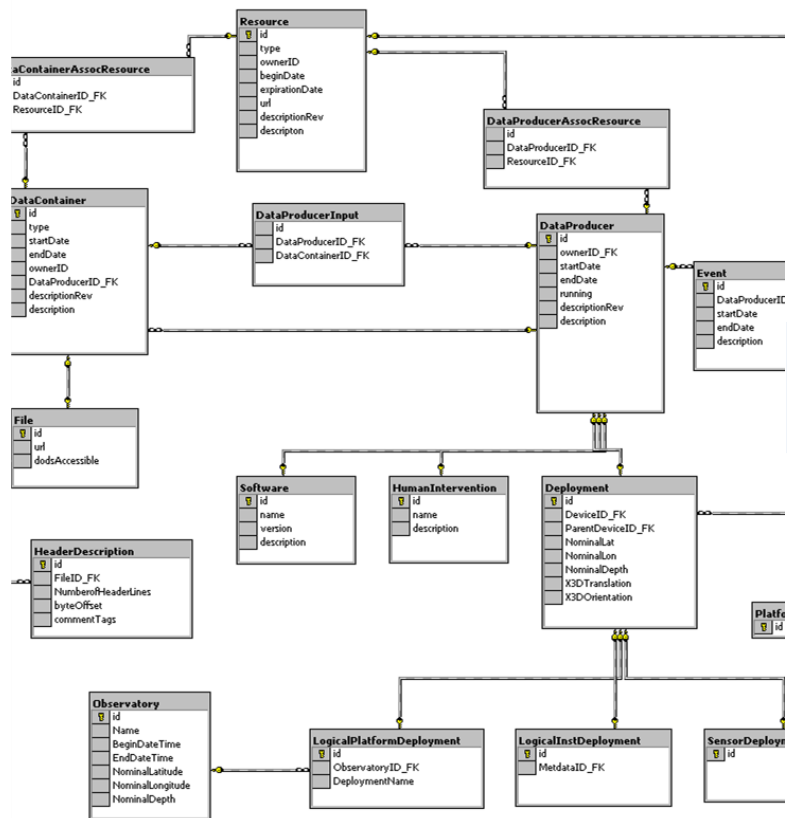


# TERMINOLOGY

---

| <b>RDBMS</b>  | <b>MongoDB</b>                 |
|---------------|--------------------------------|
| <b>Table</b>  | <b>Collection</b>              |
| <b>Row(s)</b> | <b>JSON Document</b>           |
| <b>Index</b>  | <b>Index</b>                   |
| <b>Join</b>   | <b>Embedding &amp; Linking</b> |
|               |                                |
|               |                                |

# TABLES TO COLLECTIONS OF JSON DOCUMENTS



```
{
 title: 'MongoDB',
 contributors:
 [
 { name: 'Eliot Horowitz',
 email: 'eliot@10gen.com' },
 { name: 'Dwight Merriman',
 email: 'dwight@10gen.com' }
],
 model:
 {
 relational: false,
 awesome: true
 }
}
```



# DOCUMENTS

---

- Collections contain documents
- Documents can contain other documents  
and/or
- Documents can reference other documents

## Flexible Schema

- Flexible/powerful ability to relate data



# DOCUMENTS

---

```
var p = { author: "roger",
 date: new Date(),
 title: "Spirited Away",
 avgRating: 9.834,
 tags: ["Tezuka", "Manga"]}
```

```
> db.posts.save(p)
```



Collection

# LINKED VS EMBEDDED DOCUMENTS

| EMBEDDED                                                                                                                                                                                                                                                                                                              | LINKED                                                                                                                                                                                                                                                                                                                                                                                                                                                  |
|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <pre>{   "_id": "61829c46914595254881d99c",   "text": "This is the text of the post document",   "comments": [     {       "text": "This is the text of the first comment",       "author": "author"     },     {       "text": "This is the text of the second comment",       "author": "author2"     }   ] }</pre> | <pre>{   "_id": "61829c46914595254881d99c",   "text": "This is the text of the post document" }  [   {     "_id": "61829c46914595254881d99c",     "text": "This is the text of the first comment",     "author": "author",     "postId": "61829c46914595254881d99c"   },   {     "_id": "61829c46914595254881d99c",     "text": "This is the text of the second comment",     "author": "author2",     "postId": "61829c46914595254881d99c"   } ]</pre> |

<https://medium.com/geekculture/mongodb-relationships-embedding-vs-linking-pro-and-cons-5f7583e655ab>



# QUERYING

---

```
>db.posts.find()
```

```
{ _id : ObjectId("4c4ba5c0672c685e5e8aabf3"),
 author : "roger",
 date : "Sat Jul 24 2010 19:47:11 GMT-0700 (PDT)",
 text : "Spirited Away",
 tags : ["Tezuka", "Manga"] }
```

Note:

- `_id` is unique, but can be anything you'd like



# QUERY OPERATORS

---

- Conditional Operators

- \$all, \$exists, \$mod, \$ne, \$in, \$nin, \$nor, \$or, \$size, \$type
- \$lt, \$lte, \$gt, \$gte

// find posts with any tags

```
> db.posts.find({tags: {$exists: true }})
```

// find posts matching a regular expression

```
> db.posts.find({author: /^rog*/i })
```

// count posts by author

```
> db.posts.find({author: 'roger'}).count()
```





# ATOMIC OPERATIONS

---

- \$set, \$unset, \$inc, \$push, \$pushAll, \$pull, \$pullAll, \$bit

```
> comment = { author: "fred",
 date: new Date(),
 text: "Best Movie Ever" }
```

```
> db.posts.update({ _id: "..." },
 $push: { comments: comment });
```



# INDEXES

---

Create index on any Field in Document

```
> db.posts.ensureIndex({author: 1})
```

// Index nested documents

```
> db.posts.ensureIndex("comments.author":1)
```

```
> db.posts.find({ 'comments.author': 'Fred' })
```

// Index on tags (array values)

```
> db.posts.ensureIndex(tags: 1)
```

```
> db.posts.find({ tags: 'Manga' })
```

// geospatial index

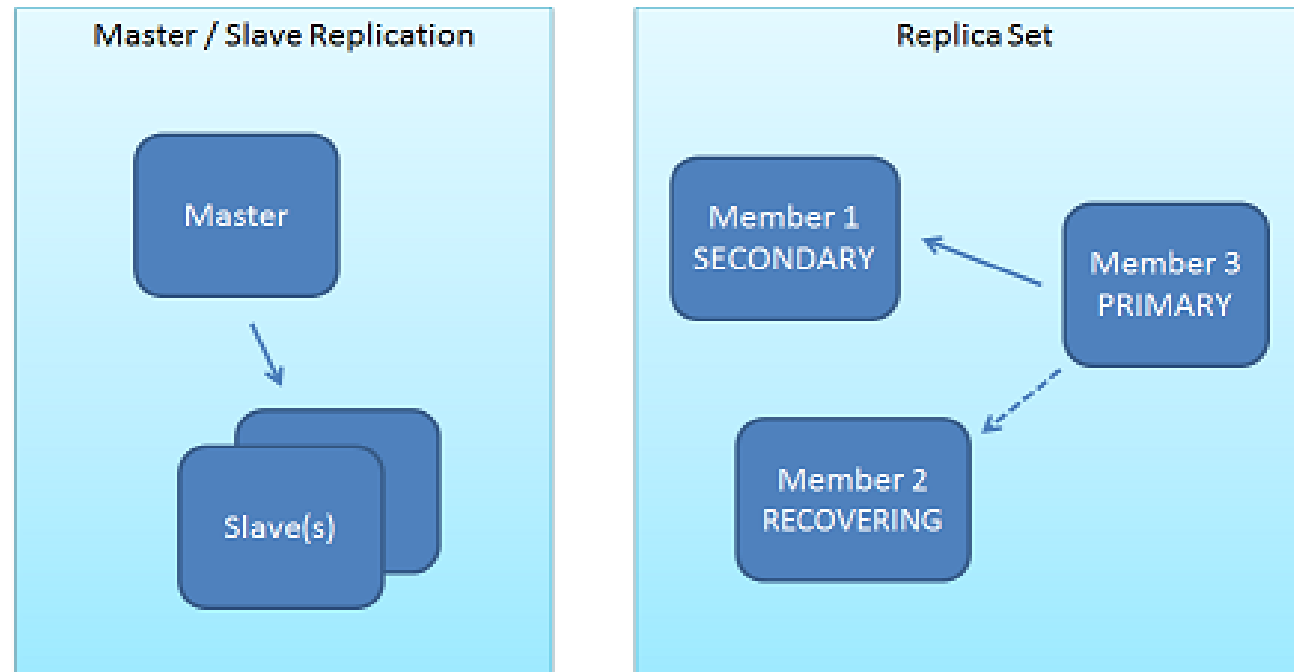
```
> db.posts.ensureIndex({ "author.location": "2d" })
```

```
> db.posts.find("author.location" : { $near : [22,42] })
```

# Architecture (MongoDB)

## 1. Replication

- Replica Sets and Master-Slave
- replica sets are a functional superset of master/slave.





# Architecture (Write process)

---

- *All write operation go through primary, which applies the write operation.*
- *write operation than records the operations on primary's operation log "oplog"*
- *Secondary are continuously replicating the oplog and applying the operations to themselves in a asynchronous process.*



# Why replica sets

---

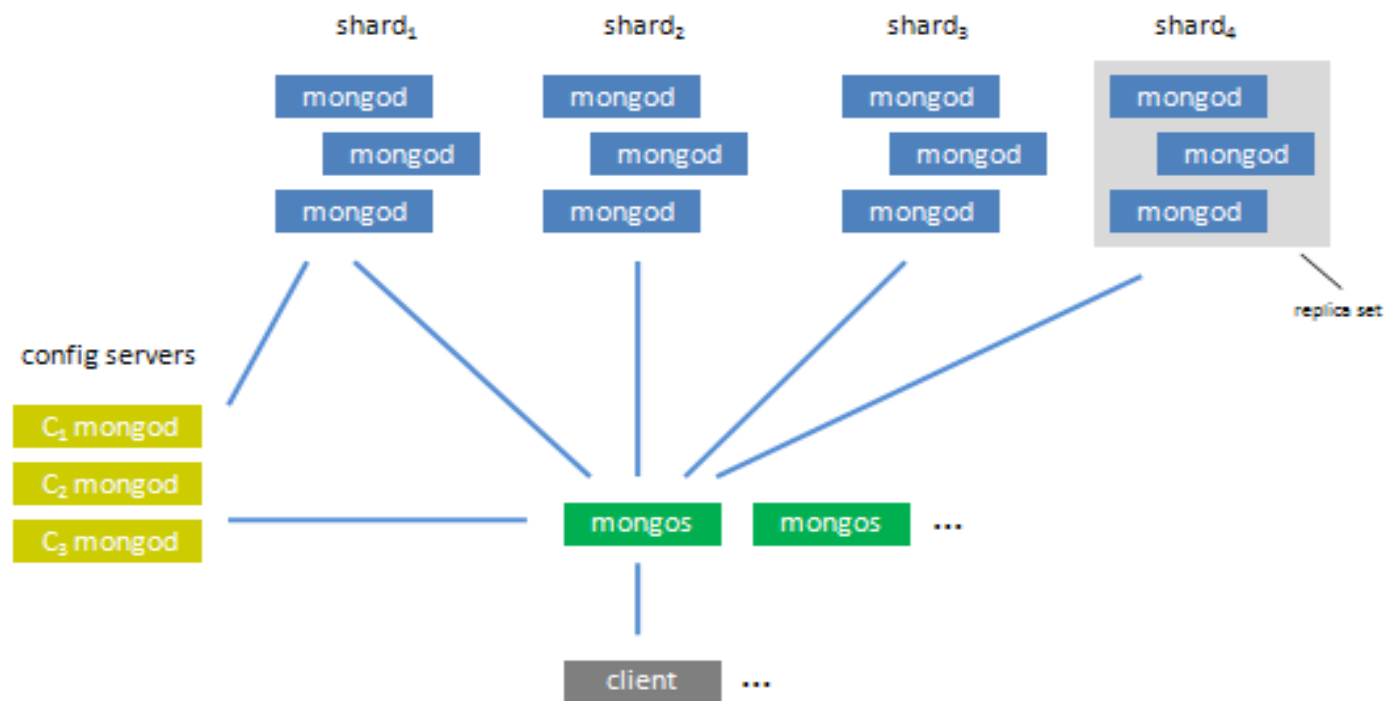
- Data Redundancy
- Automated Failover
- Read Scaling
- Maintenance
- Disaster Recovery(delayed secondary)

# Architecture

## 2. Sharding

Sharding is the partitioning of data among multiple machines in an order-preserving

r





# AGGREGATION/BATCH DATA PROCESSING

---

- Map/Reduce can be used for batch data processing
  - Currently being used for totaling, averaging, etc
  - Map/Reduce is a big hammer
- Simple aggregate functions available
- Aggregation Framework: Simple, Fast
  - No Javascript Needed, runs natively on server
  - Filter or Select Only Matching Sub-documents or Arrays via new operators
- MongoDB Hadoop Connector
  - Useful for Hadoop Integration
  - Massive Batch Processing Jobs