Introduction to HBase

What is HBase?

HBase is a distributed column-oriented data store built on top of HDFS

What is HBase?

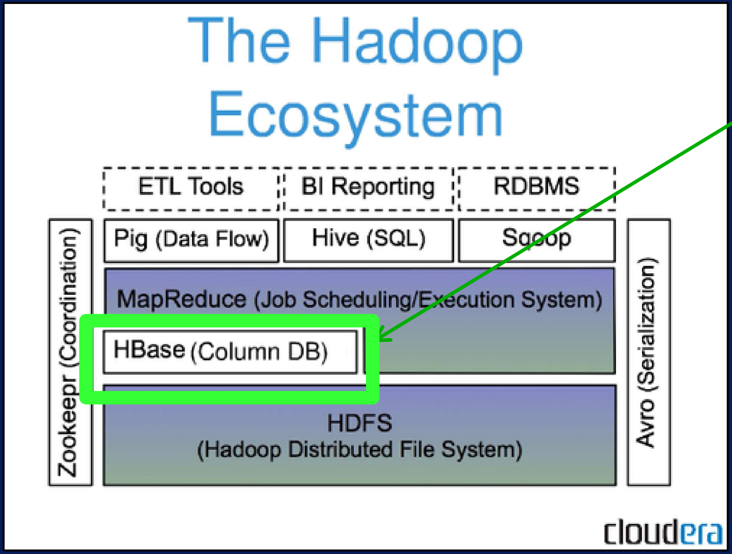
Apache HBase™ is
the Hadoop database, a
distributed, scalable, big data
store

What is HBase?

Wikipedia:

HBase is an open source, non-relational, distributed database modeled after Google's BigTable and written in Java

Hadoop Ecosystem



HBase is built on top of HDFS

HBase files are internally stored in HDFS

Why HBase when we have HDFS?

HBase vs. HDFS

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

HBase **is designed to efficiently address**

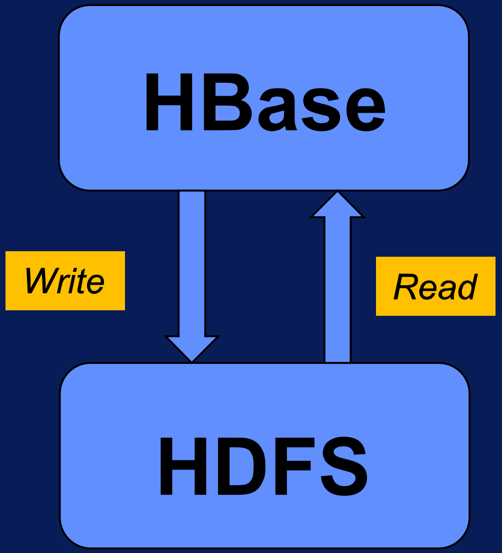
- Random Access
- Fast record lookup
- Support for record-level insertion
- Support for updates

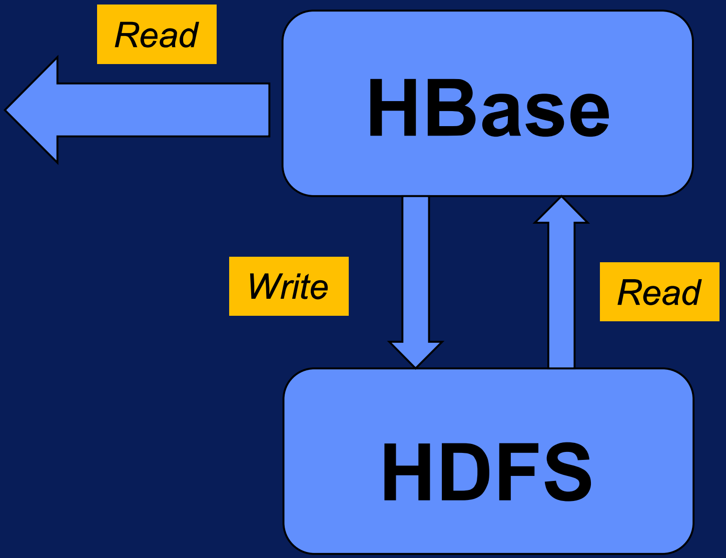
HDFS	HBase
HDFS is a distributed file system great for storing large files	HBase is a database built on top of the HDFS
HDFS does not support fast individual record lookups	HBase provides fast lookups for large tables
High latency batch processing; no concept of batch processing	Low latency access to single rows from billions of records; enables Random access
Sequential access of data only	Internally uses Hash tables and provides random access; stores the data in indexed HDFS files for faster lookups

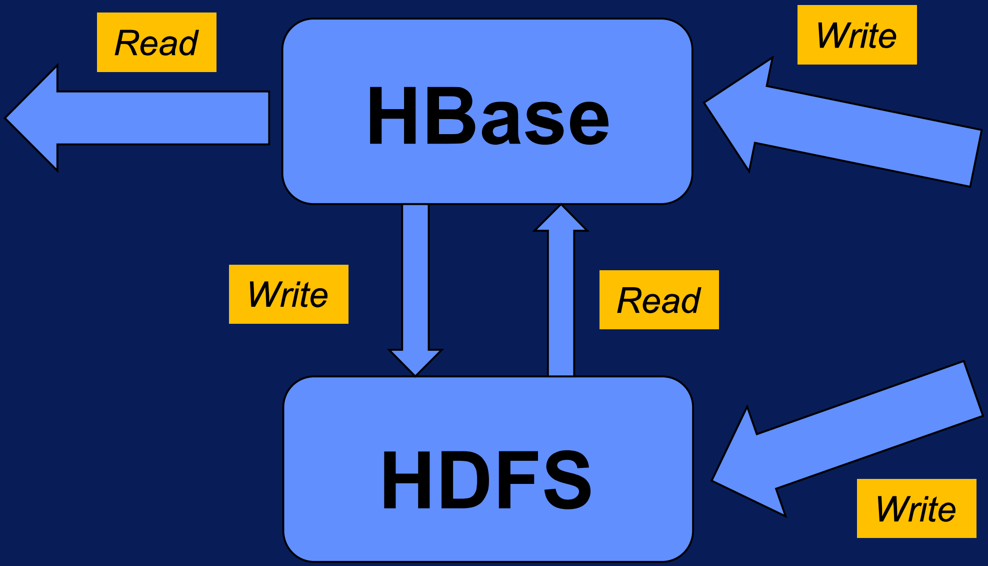
HDFS	HBase
HDFS is a distributed file system great for storing large files	HBase is a database built on top of the HDFS
HDFS does not support fast individual record lookups	HBase provides fast lookups for large tables
High latency batch processing; no concept of batch processing	Low latency access to single rows from billions of records; enables Random access
Sequential access of data only	Internally uses Hash tables and provides random access; stores the data in indexed HDFS files for faster lookups

HDFS	HBase
HDFS is a distributed file system great for storing large files	HBase is a database built on top of the HDFS
HDFS does not support fast individual record lookups	HBase provides fast lookups for large tables
High latency batch processing; no concept of batch processing	Low latency access to single rows from billions of records; enables Random access
Sequential access of data only	Internally uses Hash tables and provides random access; stores the data in indexed HDFS files for faster lookups

HDFS	HBase
HDFS is a distributed file system great for storing large files	HBase is a database built on top of the HDFS
HDFS does not support fast individual record lookups	HBase provides fast lookups for large tables
High latency batch processing; no concept of batch processing	Low latency access to single rows from billions of records; enables Random access
Sequential access of data only	Internally uses Hash tables and provides random access; stores the data in indexed HDFS files for faster lookups


HBase
Write
Read
HDFS


Read
HBase
Write
Read
HDFS


Read
HBase
Write
Write
Read
HDFS
Write

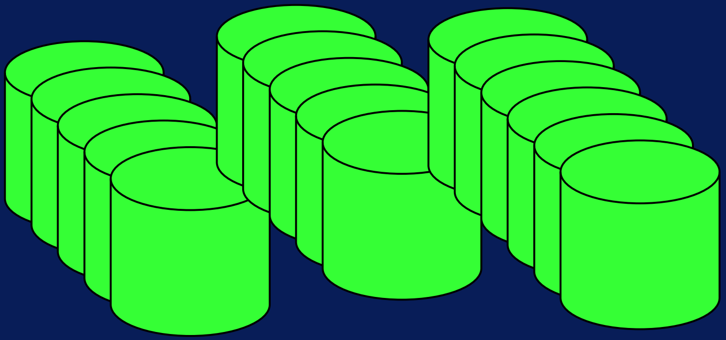
When to use Hbase?

when random, real-time read/write access to Big Data is needed

billions of rows x millions of columns

Web indexing

Satellite imagery



When to use Hbase?

Need to perform many thousands of operations per second on multiple TB/PB of data

Access patterns are well-know and simple

HBase is

Distributed column-oriented database built on top of the Hadoop file system

Horizontally scalable

HBase is NOT a Relational Database!

HBase is NOT a Relational Database!

HBase is sparse,

Lots of NULL
empty values

HBase is NOT a Relational Database!

HBase is sparse,
distributed,

“Share-nothing” architecture

HBase is NOT a Relational Database!

HBase is sparse,
distributed,
persistent,

HBase is NOT a Relational Database!

HBase is sparse,
distributed,
persistent,
multi-dimensional

HBase is NOT a Relational Database!

HBase is sparse,
distributed,
persistent,
multi-dimensional
sorted map or Key/value store

HBase Data Model

HBase Data Model

Based on Google’s BigTable

BigTable is a distributed storage system for managing structured data designed to scale to a very large size

Why BigTable?

Challenge - RDBMS performance for very large scale analytic processing

Large scale analytic processing

- Big queries – typically range or table scans
- Big databases (100s of TB/PB/ZB etc)

Bigtable

Similar to a database

NOT a full relational data model

Data indexed using row and column names

Uses MapReduce

A Bigtable table is:

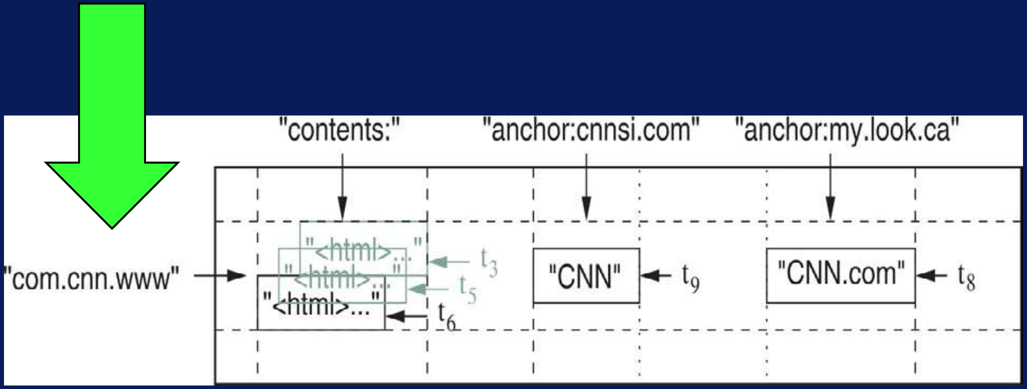
- Sparse
- Distributed
- Persistent
- Multidimensional
- Sorted map

The data in the tables is organized into three dimensions

Rows, Columns, Timestamps

(row:string, column:string, time:int64) → string

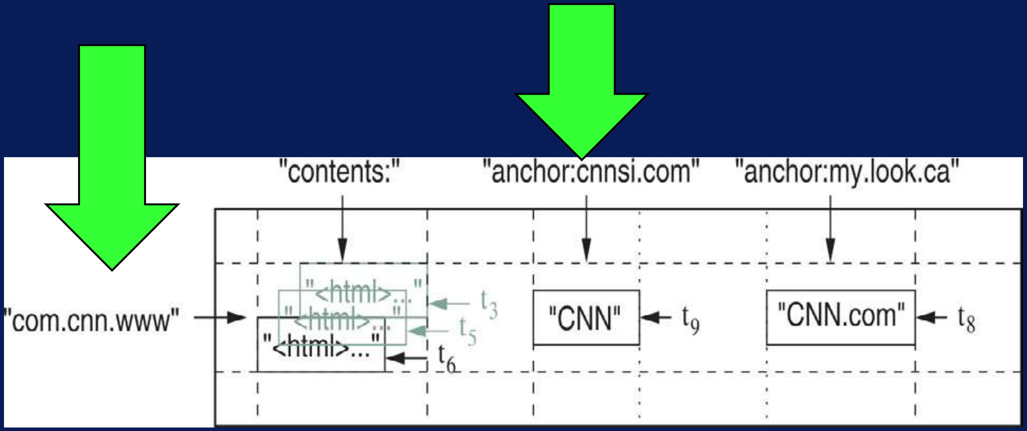
A cell is the storage referenced by a particular **row key,**



row:string, column:string, time:int64

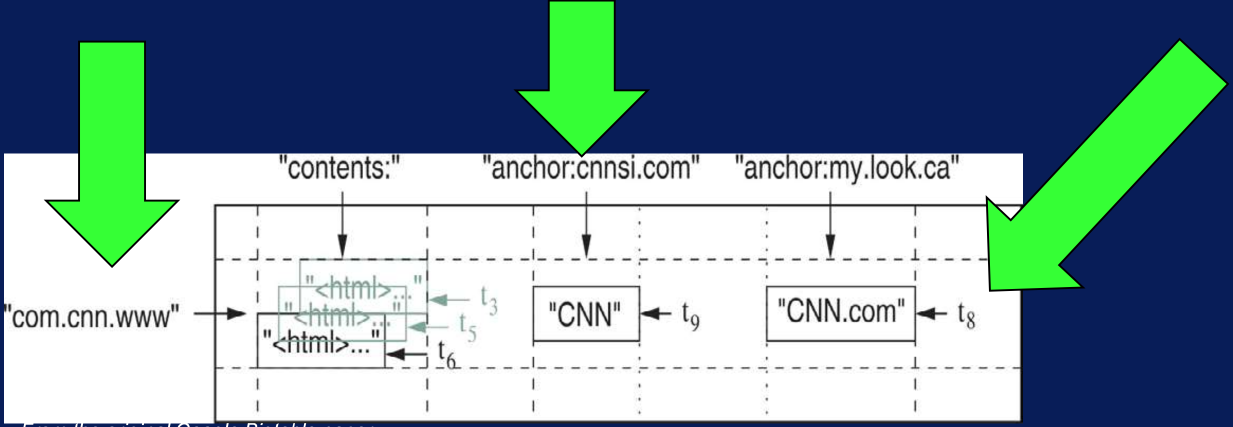
A cell is the storage referenced by a particular

row key, column key



row:string, column:string, time:int64

A cell is the storage referenced by a particular row key, column key and timestamp



From the original Google Bigtable paper

row:string, column:string, time:int64

Rows in Big Table

Rows in Bigtable

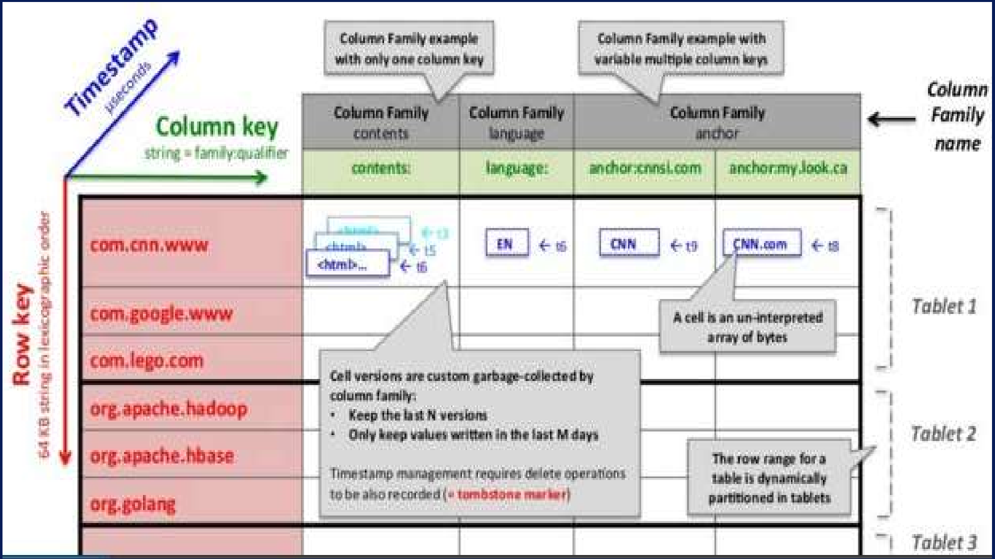
Table is collection of rows

Bigtable Tablets

Several rows are grouped in tablets

Tablets are distributed

Stored close to each other


Timestamp
μseconds
Column key
string = family:qualifier
Row key
64 KB string in lexicographic order
Column Family example with only one column key
Column Family example with variable multiple column keys
Column Family contents
Column Family language
Column Family anchor
Column Family name
contents:
language:
anchor:cnnsi.com
anchor:my.look.ca
com.cnn.www
com.google.www
com.lego.com
org.apache.hadoop
org.apache.hbase
org.golang
<html>...
← t3
← t5
← t6
EN ← t6
CNN ← t9
CNN.com ← t8
A cell is an un-interpreted array of bytes
Cell versions are custom garbage-collected by column family:
• Keep the last N versions
• Only keep values written in the last M days
Timestamp management requires delete operations to be also recorded (= tombstone marker)
The row range for a table is dynamically partitioned in tablets
Tablet 1
Tablet 2
Tablet 3

Big Table Columns

Column-oriented

The most basic unit in HBase is a column

HBase Columns

Columns could have multiple versions

Distinct values contained in a separate cells

Columns

One or more columns form a row addressed uniquely by row key

A table is a collection of rows

Column name

A column key is named with syntax → family:qualifier

Columns

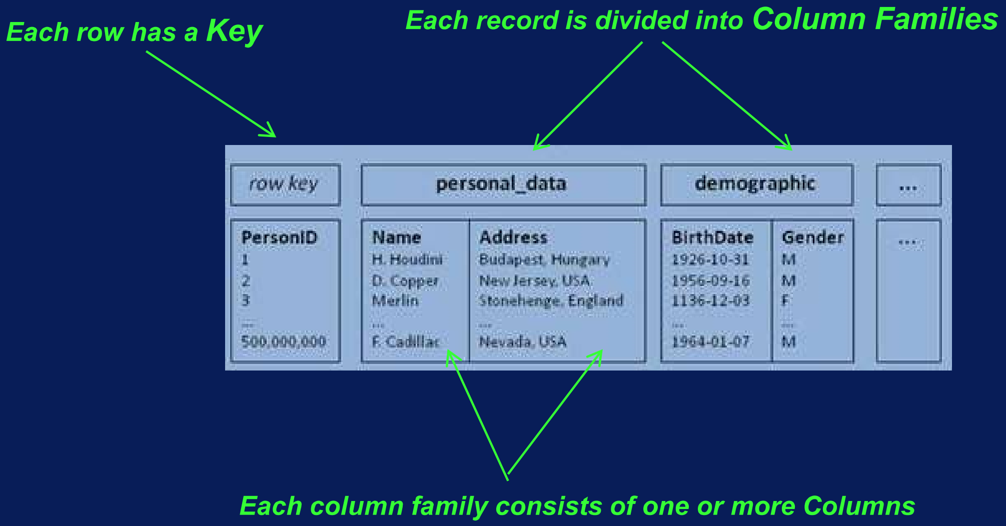
Column keys are grouped into column families

Column Families

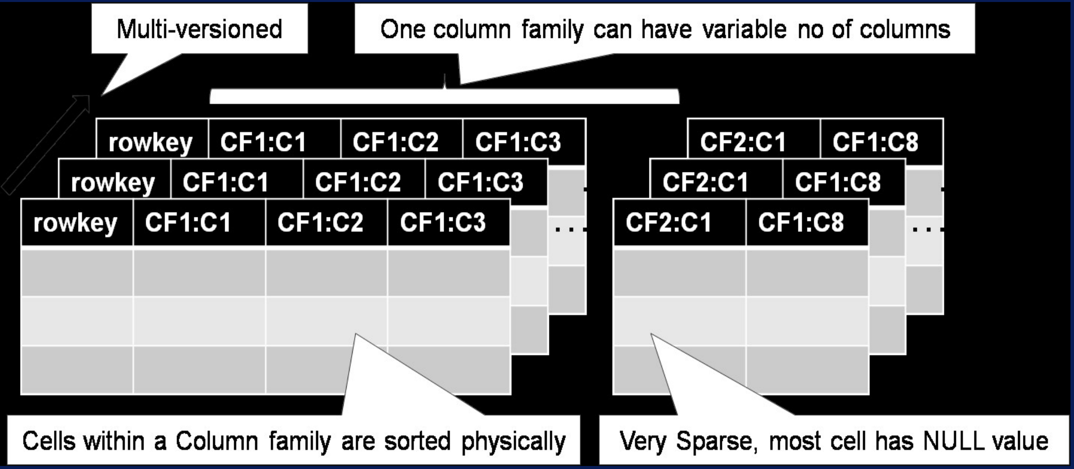
Semantical boundaries between data

Column families and columns are stored together in the same low-level storage file -> Hfile

HBase: Keys and Column Families


Each row has a Key
Each record is divided into Column Families
Each column family consists of one or more Columns

row key	personal_data		demographic		...
PersonID	Name	Address	BirthDate	Gender	...
1	H. Houdini	Budapest, Hungary	1926-10-31	M	
2	D. Copper	New Jersey, USA	1956-09-16	M	
3	Merlin	Stonehenge, England	1136-12-03	F	
...	...	...	...	...	
500,000,000	F. Cadillac	Nevada, USA	1964-01-07	M	


Multi-versioned
One column family can have variable no of columns
rowkey
CF1:C1
CF1:C2
CF1:C3
CF2:C1
CF1:C8
Cells within a Column family are sorted physically
Very Sparse, most cell has NULL value

New chunk
Big Table Timestamps

Timestamps

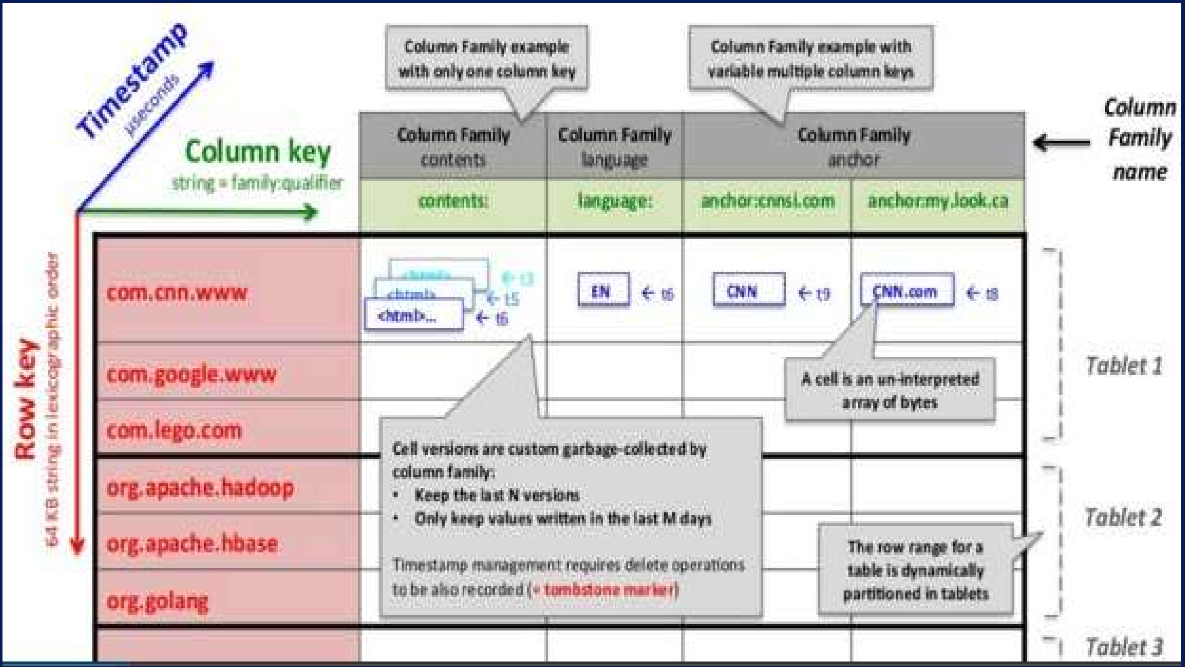
A cell can hold multiple versions of the data

Timestamps

Timestamps set by Bigtable or client applications

Timestamps

Data is stored so that new data are fastest to read


Timestamp
μseconds
Column key
string = family:qualifier
Row key
64 KB string in lexicographic order
Column Family example with only one column key
Column Family example with variable multiple column keys
Column Family contents
Column Family language
Column Family anchor
Column Family name
contents:
language:
anchor:cnnsi.com
anchor:my.look.ca
com.cnn.www
com.google.www
com.lego.com
org.apache.hadoop
org.apache.hbase
org.golang
<html> ← t3
<html> ← t5
<html>... ← t6
EN ← t6
CNN ← t9
CNN.com ← t8
A cell is an un-interpreted array of bytes
Cell versions are custom garbage-collected by column family:
• Keep the last N versions
• Only keep values written in the last M days
Timestamp management requires delete operations to be also recorded (= tombstone marker)
The row range for a table is dynamically partitioned in tablets
Tablet 1
Tablet 2
Tablet 3

From Bigtable to HBase

- It is open source
- Good integration for the Hadoop
- *No real indexes*
- *Automatic partitioning*

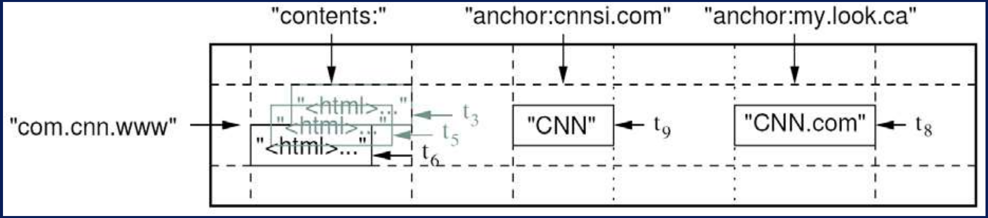
From Bigtable to HBase

- *Scale linearly and automatically with new nodes*
- *Commodity hardware*
- *Fault tolerance*
- *Batch processing*

HBase Data Model

Map indexed by a row key, column key, and a timestamp

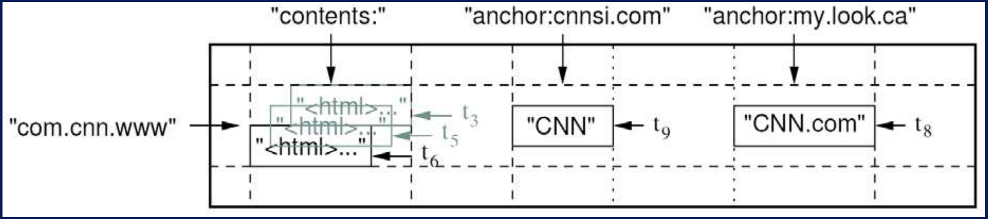
(row:string, column:string, time:int64) $\rightarrow$ uninterpreted byte array



Data Model

Supports lookups, inserts, deletes

- Single row transactions only



Rows and Columns Summary

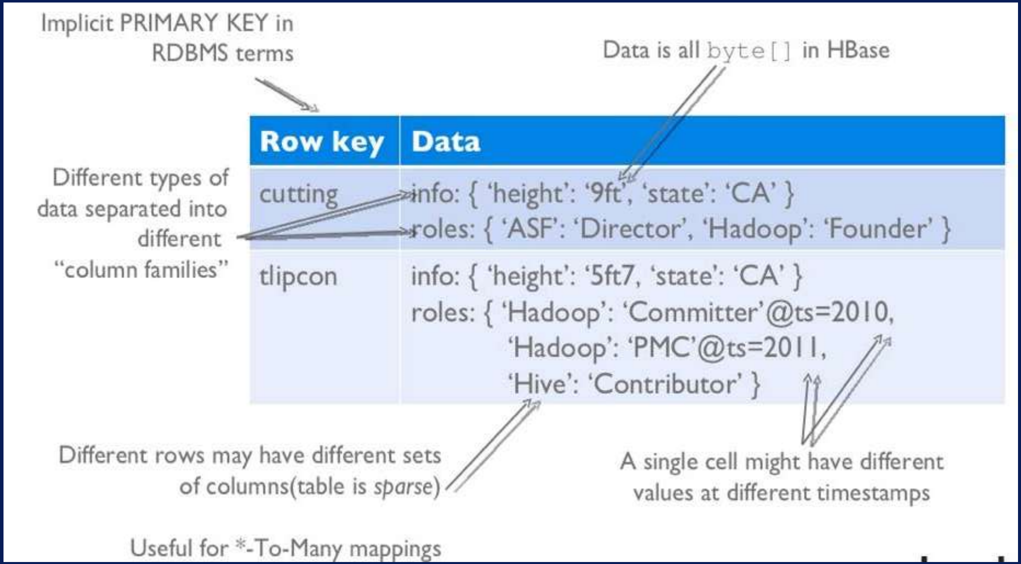
Rows maintained in sorted lexicographic order

- Efficient row scans
- Row ranges dynamically partitioned into tablets

Columns grouped into column families

- Column key = *family:qualifier*
- Column families - locality indications
- Boundless number of columns

HBase Example



Implicit PRIMARY KEY in RDBMS terms

Data is all `byte[]` in HBase

Row key	Data

cutting	info: { 'height': '9ft', 'state': 'CA' }
	roles: { 'ASF': 'Director', 'Hadoop': 'Founder' }

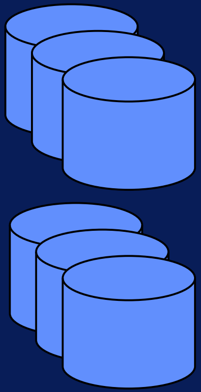
tlipcon	info: { 'height': '5ft7, 'state': 'CA' }
	roles: { 'Hadoop': 'Committer'@ts=2010,
	'Hadoop': 'PMC'@ts=2011,
	'Hive': 'Contributor' }

Different types of data separated into different "column families"

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

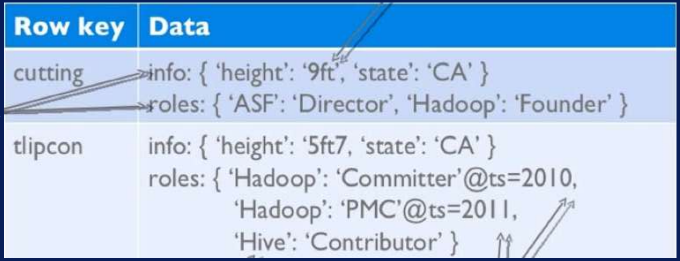


HBase Data Model

HBase schema consists of several *Tables*

Each table consists of a set of *Column Families*

- Columns are not part of the schema



Row key	Data

cutting	info: { 'height': '9ft', 'state': 'CA' }
	roles: { 'ASF': 'Director', 'Hadoop': 'Founder' }

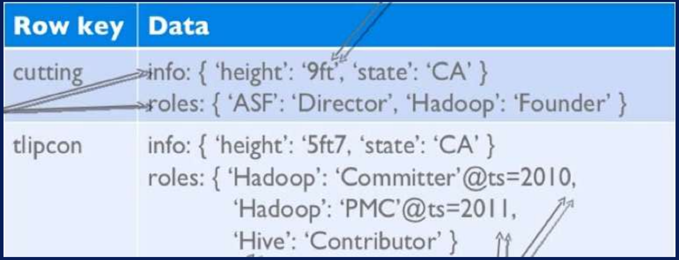
tlipcon	info: { 'height': '5ft7, 'state': 'CA' }
	roles: { 'Hadoop': 'Committer'@ts=2010,
	'Hadoop': 'PMC'@ts=2011,
	'Hive': 'Contributor' }

"Roles" column family has different columns in different cells

HBase Data Model

HBase has *Dynamic Columns*

- Column names are encoded inside the cells
- Different cells can have different columns



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' }

"Roles" column family has different columns in different cells

CAP Theorem for HBase

HBase provides

Consistency and

Partition Tolerance but is

NOT always Available

Accessing HBase

HBase Access

- HBase Shell
- Native Java API
- C/C++ HBase client
- Thrift Server
- REST
- Spark

HBase Shell

Provides interactive commands for manipulating database

- Create/delete tables
- Insert/update/read from tables
- Manage regions

Basic HBase Operations

Get – Retrieves a row of data based on the row key

Put - Inserts a row with data based on the row key

Scan - Finds all matching rows based on the row key

HBase API Operations

get(row) put(row,Map<column,value>)

scan(key range, filter)

increment(row, columns)

Check and Put, delete etc.

Quote all names

- Table and column names
- Single quotes for text

```
hbase> get 'timestamp1', 'RowId‘
```

Double quotes for binary

```
hbase> get 't1', "key\x03\x3f\xcd“
```

Specifying parameters

```
hbase> get 'UserTable', 'userId1', {COLUMN => 'address:str'}
```

HBase Shell Commands

General

Data Definition Language (DDL)

Data Manipulation Language (DML)

Cluster administration

HBase Shell Exercise

Start the QuickStart VM

Open a terminal

At the prompt – type in:

```
[cloudera@quickstart~]$ hbase shell
```

Learn more about each command

– hbase> help "<command>"

Display cluster's status via status command

```
hbase> status
hbase> status 'detailed'
```

Create HBase table

```
hbase> create ‘Movies’,{NAME=>’info’},{Name=>’director’}

hbase> put ‘Movies’,’1’,’info:title’,’Godfather’
hbase> put ‘Movies’,’1’,’info:star’,’Marlon Brando’
hbase> put ‘Movies’,’1’,’info:star’,’Al Pacino’
hbase> put ‘Movies’,’1’,’info:type’,’Crime’
hbase> put ‘Movies’,’1’,’info:type’,’Drama’
```

Create HBase table

```
hbase> create ‘Movies’,{NAME=>’info’},{Name=>’director’}

hbase> put ‘Movies’,’1’,’info:title’,’Godfather’
hbase> put ‘Movies’,’1’,’info:star’,’Marlon Brando’
hbase> put ‘Movies’,’1’,’info:star’,’Al Pacino’
hbase> put ‘Movies’,’1’,’info:movietype’,’Crime’
hbase> put ‘Movies’,’1’,’info:movietype’,’Drama’
hbase> put ‘Movies’,’1’,’director:First’,’Francis’
hbase> put ‘Movies’,’1’,’director:Middle’,’Ford’
hbase> put ‘Movies’,’1’,’director:Last’,’Copola’
```

Create HBase table

```
hbase> create ‘Movies’,{NAME=>’info’},{Name=>’director’}

hbase> put ‘Movies’,’2’,’info:title’,’Pulp Fiction’
hbase> put ‘Movies’,’2’,’info:star’,’John Travolta’
hbase> put ‘Movies’,’2’,’info:star’,’Uma Thurman’
hbase> put ‘Movies’,’2’,’info:star’,’Samuel Jackson’
hbase> put ‘Movies’,’2’,’info:movietype’,’Drama’
hbase> put ‘Movies’,’2’,’director:First’,’Quentin’
hbase> put ‘Movies’,’2’,’director:Last’,’Tarantino’
```

Verify your data

```
hbase> get ‘Movies’, ‘1’
hbase> get ‘Movies’, ‘2’
```

Change data

```
hbase> put ‘Movies’,’2’,’info:star’,’Samuel
L. Jackson’

hbase> get ‘Movies’, ‘2’
hbase> scan ‘Movies’
```

Delete data

```
hbase> delete ‘Movies’,’1’,’info:star’
hbase>disable ‘Movies’
Hbse>drop ‘Movies’
```

HUE Interface Tutorial

HBase Shell Exercise

Verify that cluster is running

```
$ hbase shell
> list
```

HBase Shell Exercise

```
$ hbase shell
> create 'test', 'data'
```

Create a simple table

HBase Shell Exercise

```
> put 'test', 'row1', 'data:1', 'value1'
```

Populate table with records

HBase Shell Exercise

```
> put 'test', 'row2', 'data:2', 'value2'
```

HBase Shell Exercise

```
> put 'test', 'row3', 'data:3', 'value3'
```

HBase Shell Exercise

```
>scan 'test'
```

Retrieve rows by scanning the entire table

HBase Shell Exercise

```
> get 'test', 'row2', 'data:2', 'value2'
```

HBase Shell Exercise

```
> delete 'test', 'row2', 'data:2',
  'value2'
```

HBase Shell Exercise

```
>scan 'test'
```

Retrieve rows by scanning the entire table

HBase Shell Exercise

```
>disable ‘test’
```

Puts the table “off-line”

Must disable before dropping the table

HBase Shell Exercise

```
>scan 'test‘
>disable ‘test’
>drop ‘test’
```

HBase Shell Exercise

```
>scan 'test‘
>disable ‘test’
>drop ‘test’
>list
```

HBase in Conclusion

HBase vs. RDBMS Revisited

HBase	vs.	RDBMS
Column-oriented		Row oriented
Flexible schema, add columns on the fly		Fixed schema
Good with sparse tables		Not optimized for sparse tables
No query language		SQL
Wide tables		Narrow tables
Joins using MapReduce		Natively performs joins
Tight integration with MapReduce		Minimal if any integration with MapReduce
Horizontal scalability – just add hardware		Hard to shard and scale
De-normilized		Normalized
No transactions		Transactional
Semi-structured & structured data		Structured data

When to consider using HBase?

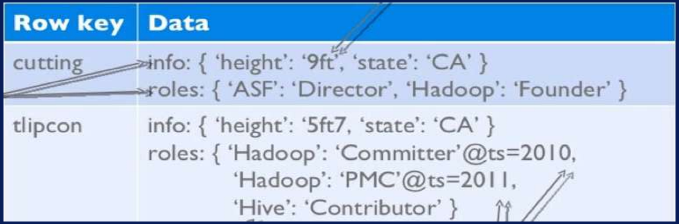
Hundreds of millions or billions of rows

Not optimized for classic transactional applications or relational analytics

When to consider using HBase?

If your application has a variable schema where each row is slightly different

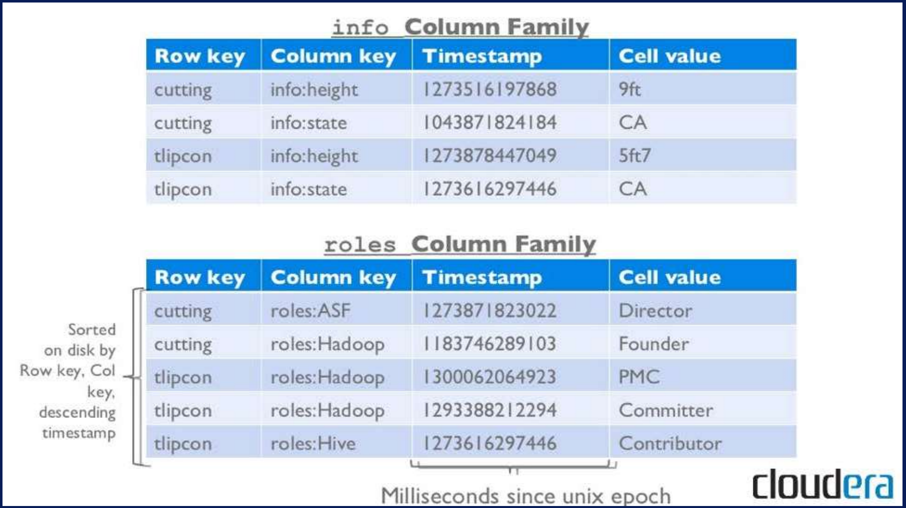
Example



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