Apache Hadoop: HBase

WL

Topics

Introduction

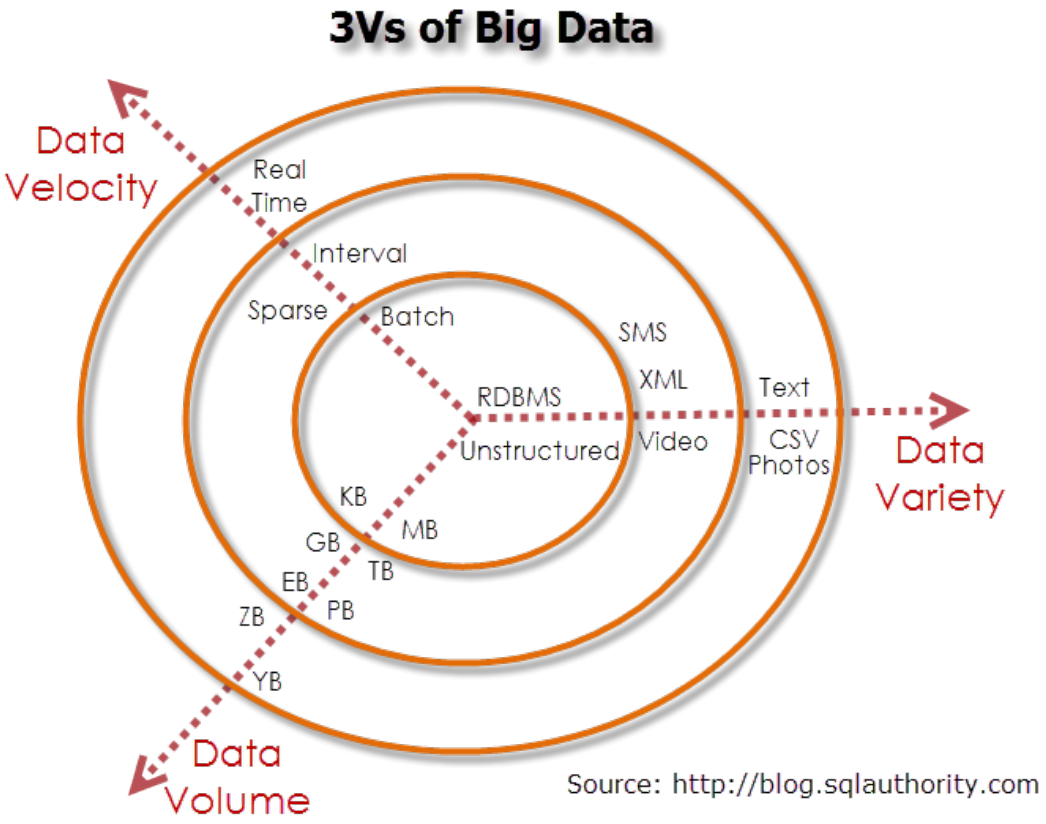
HBase Commands

Advanced Usage

What is big data ?

- Huge amount of data, also known as big data, refers to the terminology of large or complex data sets that are not adequate for processing in traditional data processing applications.

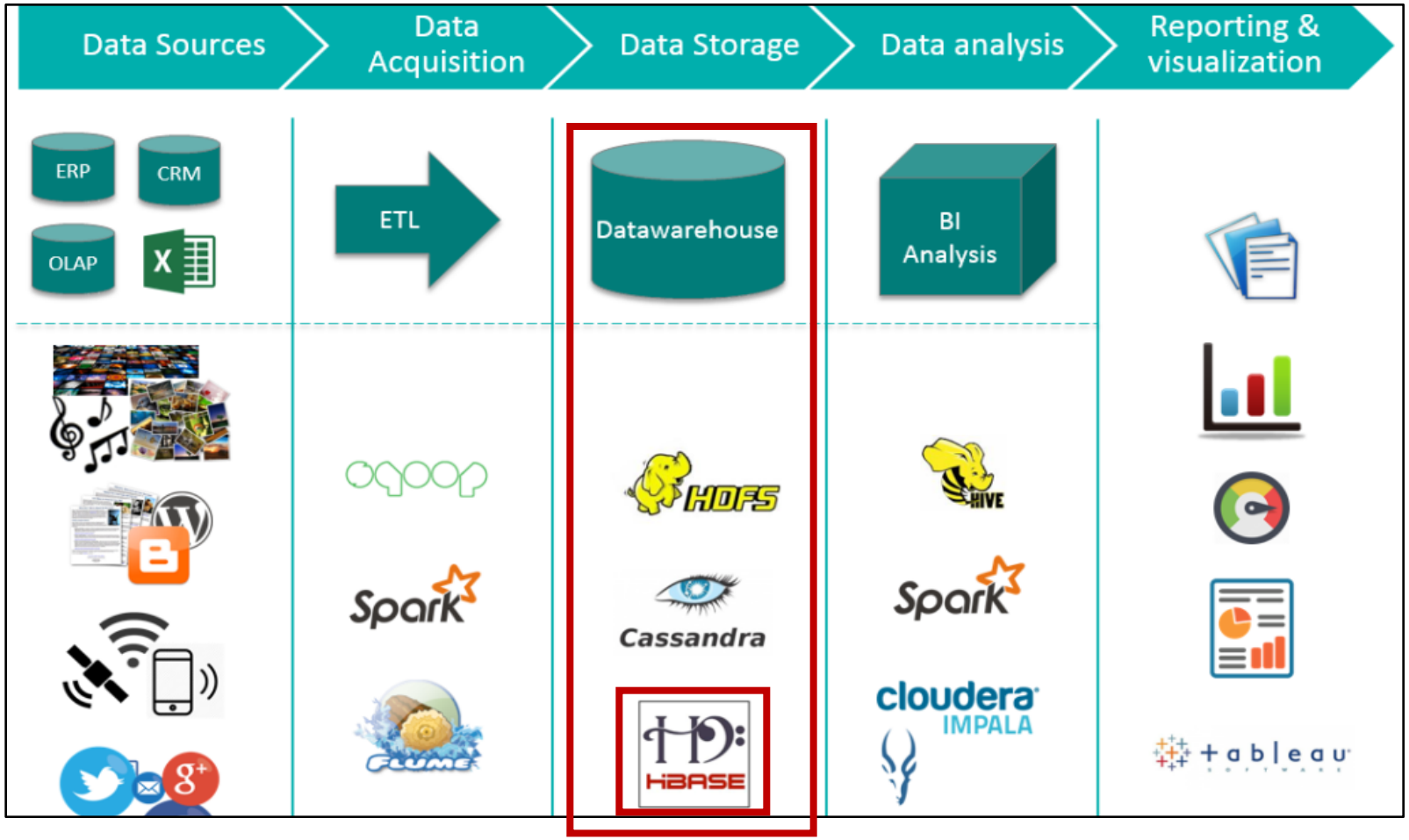
Big Data 3+1Vs



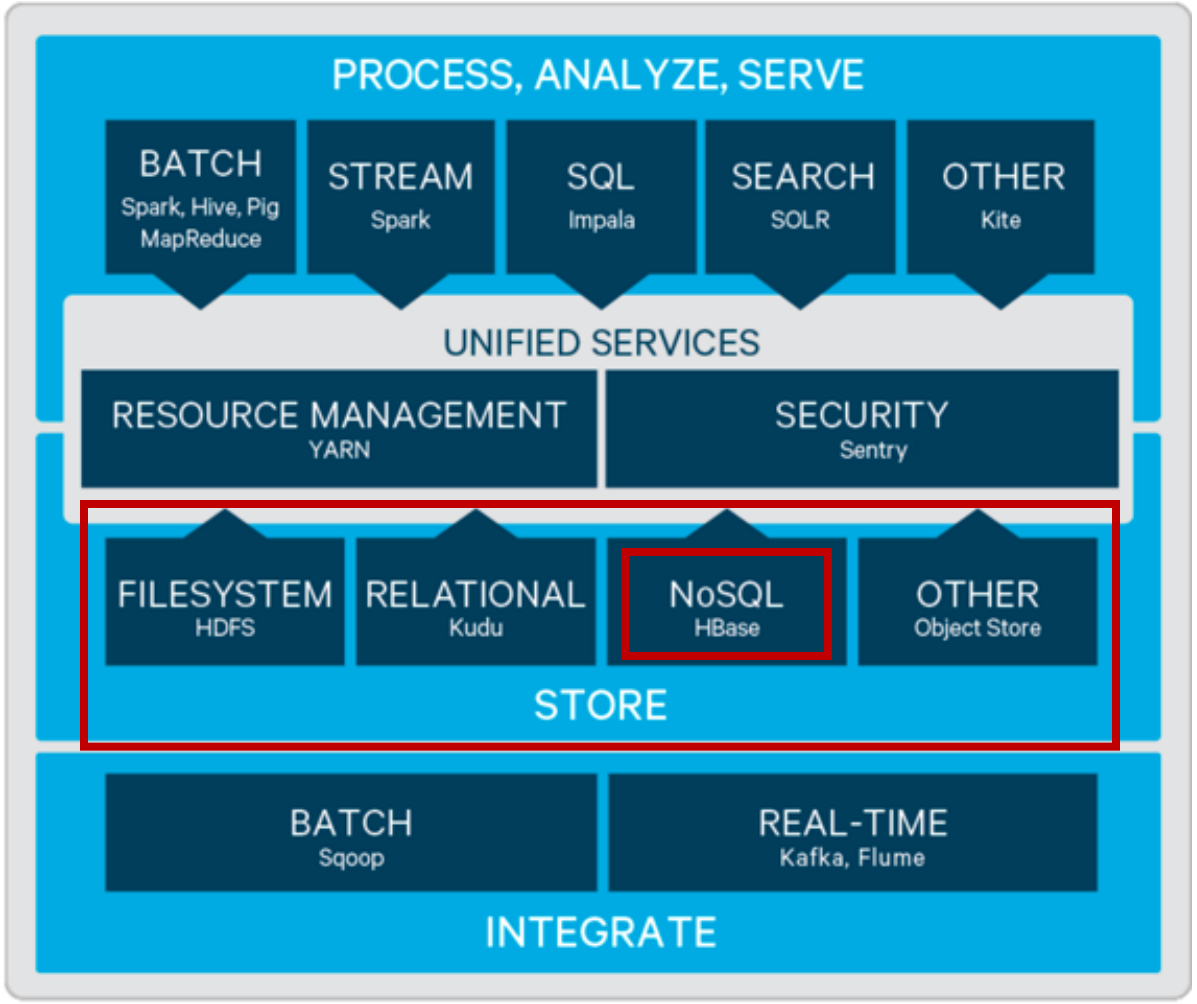
- Data Volume
- Data Variety
- Data Velocity
- Data Veracity

Modularity

ATHEMASTER



Modularity



[ref: https://docs.cloudera.com/documentation/enterprise/6/6.3/topics/cdh_intro.html](https://docs.cloudera.com/documentation/enterprise/6/6.3/topics/cdh_intro.html)

Limitations of Hadoop

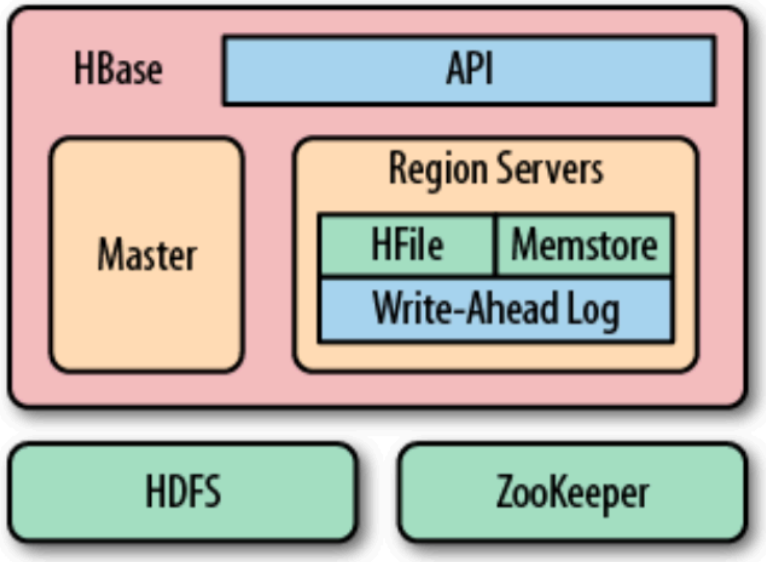
- Hadoop can perform only batch processing, and data will be accessed only in a sequential manner. That means one has to search the entire dataset even for the simplest of jobs.
- A huge dataset when processed results in another huge data set, which should also be processed sequentially. At this point, a new solution is needed to access any point of data in a single unit of time (random access).

Hadoop Random Access Databases

- Applications such as HBase, Cassandra, couchDB, Dynamo, and MongoDB are some of the databases that store huge amounts of data and access the data in a random manner.

HBase roles

- Master server
  - handling load balancing
  - not part of the actual data storage
  - maintains the state of the cluster
  - takes care of schema / metadata changes
- Region servers (RS)
  - all read and write requests for all regions
  - split regions
  - clients communicate directly with them to handle all data related operations.



HBase ecosystems

- SQL : Apache Phoenix
- REST Server
- HBase Java API : https://hbase.apache.org/1.2/apidocs/
- Monitor Tool : Ambari, Cloudera Manager
- Data-loaded tool : ImportTsv

Comparison

	RDB	*HDFS (hive, impala)*	*HBase*
Scalable	×	○	○
Modification	○	×	○
Volume	*GB*	*PB*	*PB*
Fixed Schema	○	○	×
primary key & foreign key	○	non-validated primary and foreign key	×

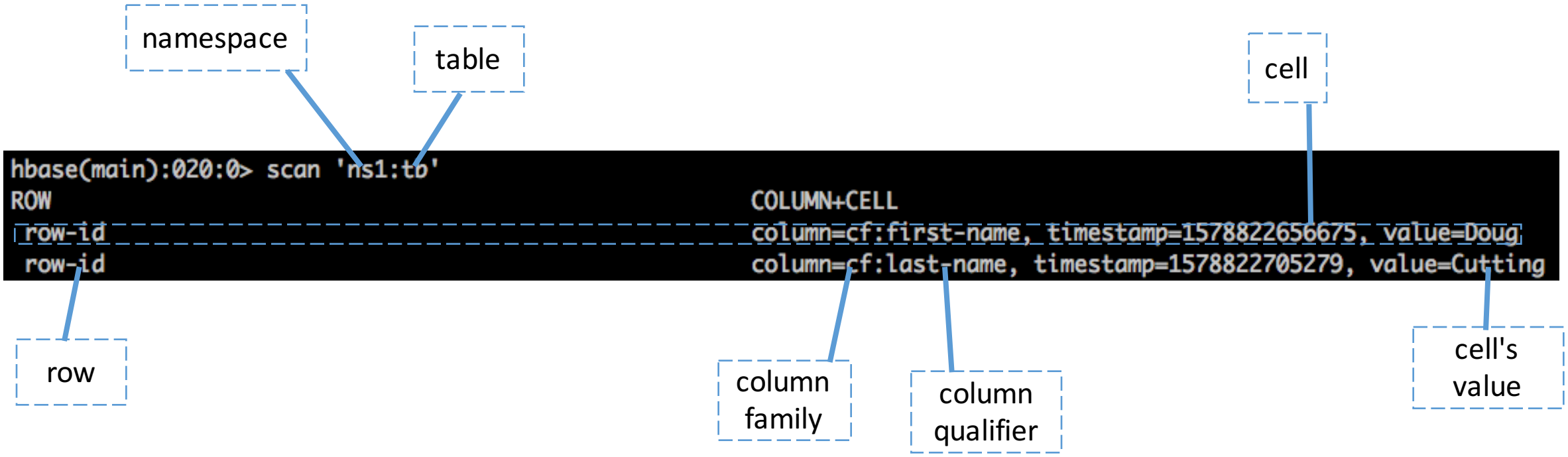
ATHEMASTER

Data Model Operations

Connect to HBase

- The ***HBase shell*** is a HBase command line tools, you can use shell commands to query the details of the data in HBase.
- If HBase environment variables are configured, you can enter the command line interface by executing hbase shell or find hbase exection located in the bin/ directory of your HBase install.
- We can start hbase shell using following command
  - > hbase shell
  - > *${hbase_home}*/bin/hbase shell

Glossary



Namespace

- A namespace is a logical grouping of tables analogous to a database in relation database systems
- There are two predefined special namespaces
  - hbase - system namespace, used to contain HBase internal tables
  - default - tables with no explicit specified namespace will automatically fall into this namespace

namespace

```
hbase(main):020:0> scan 'ns1:tb'
ROW                                                  COLUMN+CELL
 row-id                                              column=cf:first-name, timestamp=1578822656675, value=Doug
 row-id                                              column=cf:last-name, timestamp=1578822705279, value=Cutting
```

Row key

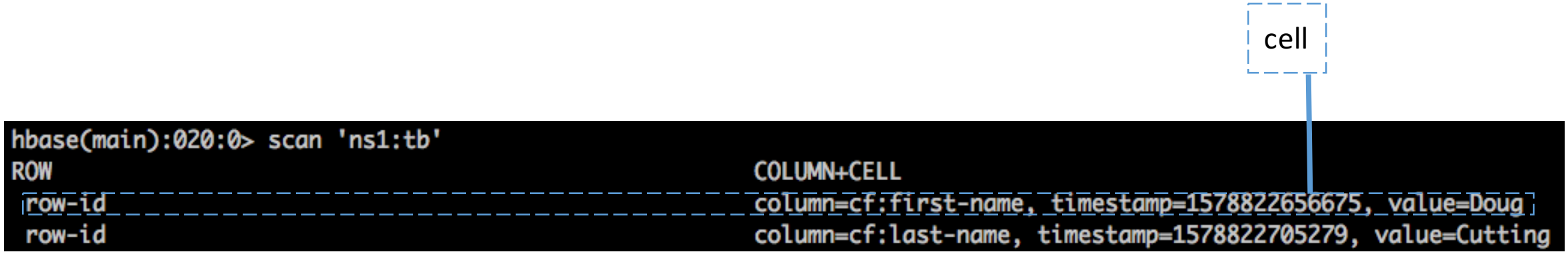
- Row keys are uninterpreted bytes. Rows are lexicographically sorted with the lowest order appearing first in a table. The empty byte array is used to denote both the start and end of a tables' namespace.

```
hbase(main):020:0> scan 'ns1:tb'
ROW                                                  COLUMN+CELL
 row-id                                              column=cf:first-name, timestamp=1578822656675, value=Doug
 row-id                                              column=cf:last-name, timestamp=1578822705279, value=Cutting
```

row key

Cells

- A {row key, column family, column qualifier, timestamp} tuple exactly specifies a cell in HBase. Cell content is uninterpreted bytes



Column Family

- An HBase table is made of column families which are the logical and physical grouping of columns. The columns in one family are stored separately from the columns in another family. If you have data that is not often queried, assign that data to a separate column family.

```
hbase(main):020:0> scan 'ns1:tb'
ROW                                                  COLUMN+CELL
 row-id                                              column=cf:first-name, timestamp=1578822656675, value=Doug
 row-id                                              column=cf:last-name, timestamp=1578822705279, value=Cutting
```

column family

Column Qualifier

- Column qualifiers are the column names, also known as column keys. In the following case, column first-name and column last-name are the column qualifiers.

```
hbase(main):020:0> scan 'ns1:tb'
ROW                                                  COLUMN+CELL
 row-id                                              column=cf:first-name, timestamp=1578822656675, value=Doug
 row-id                                              column=cf:last-name, timestamp=1578822705279, value=Cutting
```

column qualifier

Cell value

- HBase maintains maps of Keys to Values (key -> value). Each of these mappings is called a "KeyValue". You can find a value by its key

```
hbase(main):020:0> scan 'ns1:tb'
ROW                                                  COLUMN+CELL
 row-id                                              column=cf:first-name, timestamp=1578822656675, value=Doug
 row-id                                              column=cf:last-name, timestamp=1578822705279, value=Cutting
```

cell's value

Versions

- Cells in HBase is a combination of the row, column family, and column qualifier, and contains a value and a timestamp, which represents the value's version.
- A timestamp is written alongside each value and is the identifier for a given version of a value. By default, the timestamp represents the time on the RegionServer when the data was written, but you can specify a different timestamp value when you put data into the cell.

```
hbase(main):020:0> scan 'ns1:tb'
ROW                                                  COLUMN+CELL
 row-id                                              column=cf:first-name, timestamp=1578822656675, value=Doug
 row-id                                              column=cf:last-name, timestamp=1578822705279, value=Cutting
```

timestamp (version)

Operation group name

- There are many commands in HBase shell, and commands are grouped
- Group name: general
  - status, table_help, version, whoami
- Group name: ddl
  - alter, create, describe, disable, disable_all, drop, drop_all, get_table, list, show_filters, …
- Group name: namespace
  - create_namespace, describe_namespace, list_namespace, list_namespace_tables, …
- Group name: dml
  - append, count, delete, deleteall, get, get_counter, get_splits, incr, put, scan, truncate, …

List tables in namespace

- ***List*** command can list all tables in HBase. Optional regular expression parameter could be used to filter the output. Examples:
  - hbase> list
  - hbase> list 'abc.*'
  - hbase> list 'ns:abc.*'
  - hbase> list 'ns:.*'
- We also can use command ***list_namespace_table*** to get the table list in specified namespace
  - hbase> list_namesppace_table 'default'

Create namespace

- Create namespace (command: ***create_namespace***) pass namespace name, and optionally a dictionary of namespace configuration. Examples:
  - hbase> create_namespace 'ns1'

Create table

- You can create a table using the ***create*** command, here you must specify the table name and the Column Family name. The syntax to create a table in HBase shell is shown below.
  - syntax: create '<namespace:table name>', '<column family>'
  - e.g. hbase> create 'ns1:tb-name', 'cf'
- We also can create a table with table options or multiple column families at the same time
  - hbase> create 'ns1:tb', {NAME=>'cf', VERSIONS=>5}
  - hbase> create 'table2', {NAME=>'cf'}, {NAME=>'cf2'}

Get table description

- Command ***describe*** will describe the named table. Alternatively, we can use the abbreviated '***desc***' for the same thing.
  - hbase> describe 't1' *# equal to hbase> desc 't1'*
  - hbase> describe 'ns1:t1'
- We can get all column families description of this table
  - {NAME => 'cf1', DATA_BLOCK_ENCODING => 'NONE', BLOOMFILTER => 'ROW', REPLICATION_SCOPE => '0', VERSIONS => '1', COMPRESSION => 'NONE', MIN_VERSIONS => '0', TTL => 'FOREVER', KEEP_DELETED_CELLS => 'FALSE', BLOCKSIZE => '65536', IN_MEMORY => 'false', BLOCKCACHE => 'true'}

About column family properties …

- There are some important properties in column family
  - ***VERSIONS*** : Versions can be used to store multiple, but fixed number of values for a column
  - ***TTL*** (Time To Live) : column family can set a TTL length in seconds, and HBase will automatically delete rows once the expiration time is reached
  - ***MIN_VERSIONS*** : If current cell value are older than TTL, at least MIN_VERSION latest versions will be retained. This parameter should only be set when time-to-live is enabled
  - ***KEEP_DELETED_CELL*** : The role of KEEP_DELETED_CELLS is to decide whether to clean up old data when major compaction occurs. It should be noted here that even if KEEP_DELETED_CELLS is set to True, the data will still be cleaned up due to expiration.

Change table schema

- ***Alter*** is the command used to make changes to an existing table. Using this command, you can change the maximum number of cells of a column family, set and delete table scope operators, and delete a column family from a table.
- For example, we can changing the Maximum Number of cells of a column family
  - hbase> alter 't1', {'NAME' => 'f1', 'VERSIONS' => 5}

How to add / delete column family …

- We can use ***alter*** command to add / remove a column family
- Add new column family to an existing HBase table
  - hbase> alter 't1', 'NAME'=> 'new_cf'
  - hbase> alter 't1', {'NAME'=> 'new_cf', 'VERSIONS'=>5}
- Deleting a Column Family
  - hbase> alter 't1', 'delete' => 'cf_name'
  - hbase> alter 't1', {'NAME'=> 'cf_name', 'METHOD'=> 'delete'}
- *Note: All relative cells of column family will be removed after column family is deleted*

Drop existing tables

- By using ***drop*** command, we can remove a table from the namespace
  - e.g. hbase> drop 'ns1:table_name'
- We can use command ***drop_all*** to delete tables as well. This command is used to drop the tables matching the "regex" given in the command.
  - hbase> drop_all 'table_*'
- *Note: The table must be disabled before dropping.*
  - hbase> disable 'table_name'
  - hbase> disalbe_all 'table_*'

Challenge(1)

- step1: create your own namespace, which is named by your staff id
- step2: create a table 'tb' belonging to the above namespace, with column family 'cf'
- step3: alter the max VERSIONS to 3 of column family 'cf' in this table
- step4: add another column family 'cf2' in this table with property KEEP_DELETED_CELLS is 'TRUE' and VERSIONS is 5
- step5: describe the table to display all column family's description

Challenge(1) : Answer

- step1: create your own namespace, which is named by your staff id
  - create_namespace 'your_id'
- step2: create a table 'tb' belonging to the above namespace, with column family 'cf'
  - create table 'your_id:tb', 'cf'
- step3: alter the max VERSIONS to 3 of column family 'cf' in this table
  - alter 'your_id:tb', {NAME=>'cf', VERSIONS=>3}
- step4: add another column family 'cf2' in this table with property KEEP_DELETED_CELLS is 'TRUE' and VERSIONS is 5
  - alter 'your_id:tb', {NAME=>'cf2', KEEP_DELETED_CELLS=>'TRUE', VERSIONS=>5}
- step5: describe the table to display all column family's description
  - desc 'your_id:tb'

Using 'get' to fetch row contents

- Get row or cell contents; pass table name, row, and optionally a dictionary of column(s), timestamp, time range and versions.
  - hbase> get 'ns1:t1', 'r1'
  - hbase> get 't1', 'r1'
  - hbase> get 't1', 'r1', {TIMERANGE => [ts1, ts2]}
  - hbase> get 't1, 'r1', {COLUMN => 'c1'}
- We can get value with property VERSIONS to get old version
  - hbase> get 't1','rowkey1', {'COLUMN'=>['cf:name', 'cf:age'], 'VERSIONS'=>3}

Create data in HBase table

- To create data in Hbase table, we can use command ***put*** to put a cell 'value' at specified table/row/column and optionally timestamp coordinates.
  - hbase> put 't1', 'r1', 'c1', 'value'
  - hbase> put 'ns1:t1', 'r1', 'c1', 'value'
  - hbase> put 't1', 'r1', 'c1', 'value', ts1

we can specify a different timestamp (version) value when putting data into the cell. This value represents the time on the Region Server when the data was written by default.

Update cell value

- We can update an existing cell value using the ***put*** command. To do so, just follow the same syntax and mention your new value as shown below.
  - hbase> put 'namespace:table_name', 'row_id', 'column_family:column_qualifier', 'new value'
- The newly given value replaces the existing value, updating the row.

Questions and discussions!? (1)

- In a table with property VERSION is 3, we put 3 cell into the table
  - put 'table', 'winther.lee', 'cf:logindate', '20190305' — this cell will be removed
  - put 'table', 'winther.lee', 'cf:logindate', '20190306'
  - put 'table', 'winther.lee', 'cf:logindate', '20190307'
- Which cell will be removed after we insert following cell
  - put 'table', 'winther.lee', 'cf:logindate', '20190308'

Questions and discussions!? (2)

- In a table with property VERSION is 3, we put 3 cell into the table
  - put 'table', 'winther.lee', 'cf:logindate', '20190305', 4000
  - put 'table', 'winther.lee', 'cf:logindate', '20190306', 2000
  - put 'table', 'winther.lee', 'cf:logindate', '20190307', 1000 — this cell will be removed
- Which cell will be removed after we insert following cell
  - put 'table', 'winther.lee', 'cf:logindate', '20190308', 3000

Questions and discussions!? (3)

- In a table with property VERSION is 3, we put 3 cell into the table
  - put 'table', 'winther.lee', 'cf:logindate', '20190305', 1000
  - put 'table', 'winther.lee', 'cf:logindate', '20190306', 2000
  - put 'table', 'winther.lee', 'cf:logindate', '20190307', 3000

this cell will be removed

- Which cell will be removed after we insert following cell
  - put 'table', 'winther.lee', 'cf:logindate', '20190308', 2000

Delete

- If we want to remove some specified column, use command ***delete*** to put a delete cell value at specified table/row/column and optionally timestamp coordinates. Deletes must match the deleted cell's coordinates exactly.
  - hbase> delete 'ns1:t1', 'r1', 'c1'
  - hbase> delete 'ns1:t1', 'r1', 'c1', ts1
- We can also use ***deleteall*** command to delete all cells in a given row; pass a table name, row, and optionally a column and timestamp
  - hbase> deleteall 'ns1:t1', 'row-key'

Scan

- The scan command is used to view the data in HTable. Using the scan command, you can get the table data. If no columns are specified, all columns will be scanned. To scan all members of a column family, leave the qualifier empty as in 'col_family'. Its syntax is as follows: **(scan all data then filter)**
  - hbase> scan '<table name>'
  - hbase> scan 't1', {COLUMNS=> ['cf:c1', 'cf:c2']}
  - hbase> scan 't1', {COLUMNS=> ['cf']}

Scan

- Scanner specifications may include one or more of: TIMERANGE, FILTER, LIMIT, STARTROW, STOPROW, ROWPREFIXFILTER, TIMESTAMP, MAXLENGTH or COLUMNS, CACHE or RAW, VERSIONS, ALL_METRICS or METRICS
  - scan '<table name>'
  - scan '<table name>', {OPTIONS}
- Example given
  - scan 'ns1:tb', {COLUMN=>['cf:name'], LIMIT=>10}

Scan specifications

- Here are some scan specifications description
  - ***FILTER*** : the specified server-side filter when performing the Query
  - ***LIMIT*** : the limit of rows for this scan
  - ***COLUMNS*** : the column from the specified family with the specified qualifier
  - ***RAW*** : the scanner to return all cells (including delete markers and uncollected deleted cells)
  - ***VERSIONS*** : get up to the specified number of versions of each column
  - ***TIMERANGE*** : get versions of columns only within the specified timestamp range, [minStamp, maxStamp).

Scan deleted rows ?!

ATHEMASTER

- "raw" scan options returns all deleted rows and the delete markers.
  - scan '<table name>', {RAW=>true}

```
hbase(main):095:0> put 'ns1:tb', 'removed-key', 'cf:name', 'winther.lee'
0 row(s) in 0.0240 seconds

hbase(main):096:0> deleteall 'ns1:tb', 'removed-key'
0 row(s) in 0.0180 seconds

hbase(main):097:0> scan 'ns1:tb'
ROW                                                 COLUMN+CELL
0 row(s) in 0.0260 seconds

hbase(main):098:0> scan 'ns1:tb', {RAW=>true}
ROW                                                 COLUMN+CELL
 removed-key                                        column=cf:, timestamp=1579015899683, type=DeleteFamily
 removed-key                                        column=cf:name, timestamp=1579015886549, value=winther.lee
1 row(s) in 0.0310 seconds
```

delete rowkey 'removed-key'

get no data

get data with RAW = true

How to remove cell permanently?

- When you delete the cell in HBase, the data is not actually deleted but a tombstone marker is set, making the deleted cells invisible. HBase deleted are actually removed during compactions.
  - hbase> flush '<table>' # flush all cache from memstore
  - hbase> major_compact '<table>' # remove all delete markers cell
- ***Note: KEEP_DELETED_CELL,*** The role of KEEP_DELETED_CELLS is to decide whether to clean up old data when major compaction occurs. It should be noted here that even if KEEP_DELETED_CELLS is set to True, the data will still be cleaned up due to expiration.

Challenge(2)

- step1: put data tag101, tag102, tag103 into table ${id}:tb with rowkey is 'user-id', column is 'cf:tags', then use command get to display the result, including old data.
  - e.g. hbase> put '<table>', 'user-id', 'id:tb', 'cf:tags', 'tag101'
  - e.g. hbase> get '<table>', 'user-id', {COLUMN=>'cf:tags', VERSIONS=>5}
- step2: put data tag104 into table ${id}:tb and column 'cf:tags', then use command get to display the result
- step3: remove row 'user-id', then use scan command with/without RAW property is true to fetch the results
  - e.g. hbase> delete '<table>', 'row1', 'cf:tags'
  - e.g. hbase> scan '<table>'
  - e.g. hbase> scan '<table>', {RAW=>true}

Challenge(2)

- step4: flush and compact hbase table, then scan again
  - e.g. hbase> flush '<table>'
  - e.g. hbase> major_comact '<table>'
  - e.g. hbase> scan '<table>', {RAW=>true}
- step5: do step1-4 again in column family cf2 again to observe the difference

Challenge(2) - Answer

- step1:
  - put '$id:tb', 'user-id', 'cf:tags', 'tag101'
  - put '$id:tb', 'user-id', 'cf:tags', 'tag102'
  - put '$id:tb', 'user-id', 'cf:tags', 'tag103'
  - get '$id:tb', 'user-id', {COLUMN=>'cf:tags', VERSIONS=>5}

```
hbase(main):016:0> get '1001:tb', 'user-id', {COLUMN=>['cf:tags'], VERSIONS=>5}
COLUMN                                 CELL
 cf:tags                               timestamp=1579097972770, value=tag103
 cf:tags                               timestamp=1579097970517, value=tag102
 cf:tags                               timestamp=1579097968637, value=tag101
3 row(s) in 0.0130 seconds
```

Challenge(2) - Answer

- step2:
  - put '1001:tb', 'user-id', 'cf:tags', 'tag104'
  - get '1001:tb', 'user-id', {COLUMN=>['cf:tags'], VERSIONS=>5}

```
hbase(main):018:0> get '1001:tb', 'user-id', {COLUMN=>['cf:tags'], VERSIONS=>5}
COLUMN                                 CELL
 cf:tags                               timestamp=1579098091085, value=tag104
 cf:tags                               timestamp=1579097972770, value=tag103
 cf:tags                               timestamp=1579097970517, value=tag102
3 row(s) in 0.0080 seconds
```

- step3: deleteall '1001:tb', 'user-id'

Challenge(2) - Answer

- step3:
  - deleteall '1001:tb', 'user-id'
  - scan '1001:tb'

```
hbase(main):027:0> scan '1001:tb'
ROW                                    COLUMN+CELL
0 row(s) in 0.0030 seconds
```

  - scan '1001:tb', {RAW=> true}

```
hbase(main):029:0> scan '1001:tb', {RAW=> true}
ROW                                    COLUMN+CELL
 user-id                               column=cf:, timestamp=1579098458734, type=DeleteFamily
 user-id                               column=cf:tags, timestamp=1579098091085, value=tag104
 user-id                               column=cf2:, timestamp=1579098458734, type=DeleteFamily
1 row(s) in 0.0160 seconds
```

  - scan '1001:tb', {RAW=>true, VERSIONS=>5}

```
hbase(main):030:0> scan '1001:tb', {RAW=> true, VERSIONS=>5}
ROW                                    COLUMN+CELL
 user-id                               column=cf:, timestamp=1579098458734, type=DeleteFamily
 user-id                               column=cf:tags, timestamp=1579098091085, value=tag104
 user-id                               column=cf:tags, timestamp=1579097972770, value=tag103
 user-id                               column=cf:tags, timestamp=1579097970517, value=tag102
 user-id                               column=cf:tags, timestamp=1579097968637, value=tag101
 user-id                               column=cf2:, timestamp=1579098458734, type=DeleteFamily
1 row(s) in 0.0070 seconds
```

Challenge(2) - Answer

- step4:
  - flush '1001:tb'
  - major_compact '1001:tb'
  - scan '1001:tb', {RAW=>true, VERSIONS=>5}

```
hbase(main):033:0> scan '1001:tb', {RAW=>true, VERSIONS=>5}
ROW                                   COLUMN+CELL
0 row(s) in 0.0100 seconds
```

Challenge(2) - Answer

- step5:
  - get '1001:tb', user-id', {COLUMN=>['cf2:tags'], VERSIONS=>5} will return 4 rows (cf2's VERSIONS property is 5

```
hbase(main):041:0> get '1001:tb', 'user-id', {COLUMN=>['cf2:tags'], VERSIONS=>5}
COLUMN                                CELL
 cf2:tags                             timestamp=1579099030739, value=tag104
 cf2:tags                             timestamp=1579099029420, value=tag103
 cf2:tags                             timestamp=1579099028220, value=tag102
 cf2:tags                             timestamp=1579099026394, value=tag101
4 row(s) in 0.0090 seconds
```

Challenge(2) - Answer

- step5:
  - scan '1001:tb', {RAW=>true, VERSIONS=>5} will return all deleted cells because we set cf2 KEEP_DELETED_CELLS property is true

```
hbase(main):001:0> flush '1001:tb'
0 row(s) in 0.3900 seconds

hbase(main):002:0> major_compact '1001:tb'
0 row(s) in 0.2320 seconds

hbase(main):003:0> scan '1001:tb', {RAW=>true, VERSIONS=>5}
ROW                                   COLUMN+CELL
 user-id                              column=cf2:, timestamp=1579099229872, type=DeleteFamily
 user-id                              column=cf2:tags, timestamp=1579099030739, value=tag104
 user-id                              column=cf2:tags, timestamp=1579099029420, value=tag103
 user-id                              column=cf2:tags, timestamp=1579099028220, value=tag102
 user-id                              column=cf2:tags, timestamp=1579099026394, value=tag101
1 row(s) in 0.0490 seconds
```

Scan with TIMERANGE & TIMESTAMP

- Represents an interval of version timestamps. Presumes timestamps between *INITIAL_MIN_TIMESTAMP* and *INITIAL_MAX_TIMESTAMP* only. Evaluated according to *minStamp <= timestamp < maxStamp* or [minStamp, maxStamp) in interval notation
  - hbase> scan '<table name>', {TIMERANGE => [1303668804, 1303668904]}
  - hbase> scan '<table name>', {TIMESTAMP => 1303668804}

Scan with STARTROW, STOPROW, ROWPREFIXFILTER

- We can scan table with option STARTROW and STOPROW
  - STARTROW : row to start scanner at or after
  - STOPROW : row to end at (exclusive)
  - e.g. hbase> scan 'table', {STARTROW=>'row1', STOPROW=>'row2'}
- We can use rowprefixfilter so the result set only contains rows where the rowKey starts with the specified prefix.
  - This will overwrite any startrow/stoprow settings
  - e.g. hbase> scan 'table', {ROWPREFIXFILTER=>'r'}
    Note: rowprefixfilter can safely be used with other filter

Scan with FILTER

- Provides row-level filters applied to HRegion scan results during calls to ResultScanner.next(). We can use '***show_filters***' instruction to list all available filter in HBase shell. See following page for more filter detail
  - hbase> show_filters

- ColumnPrefixFilter
- TimestampsFilter
- PageFilter
- MultipleColumnPrefixFilter
- FamilyFilter
- FirstKeyOnlyFilter
- ColumnPaginationFilter
- SingleColumnValueFilter
- RowFilter
- QualifierFilter
- ColumnRangeFilter
- ValueFilter
- PrefixFilter
- SingleColumnValueExcludeFilter
- ColumnCountGetFilter
- InclusiveStopFilter
- DependentColumnFilter
- KeyOnlyFilter

Filter language

- Filter Language allows you to perform server-side filtering when accessing HBase over Thrift or in the HBase shell.
- You specify a filter as a string, which is parsed on the server to construct the filter. A simple filter expression is expressed as a string:
  - "FilterName (argument, argument,... , argument)"
- Example given:
  - hbase> scan 'table', {FILTER => "PrefixFilter('ro')"}

Filter language Example

- ***KeyOnlyFilter***
  - ***example given: {FILTER=>"KeyOnlyFilter()"}***
  - This filter doesn't take any arguments. It returns solely the key part of every key-value
- ***PrefixFilter***
  - ***example given: {FILTER=>"PrefixFilter("A")"}***
  - This filter takes one argument as a prefix of a row key. It returns solely those key-values present in the very row that starts with the specified row prefix
- ***ValueFilter***
  - ***example given: {FILTER=>"ValueFilter(=, 'substring:view')"}***
  - This filter takes a compare operator and a comparator. It compares every value with the comparator using the compare operator and if the comparison returns true, it returns that key-value.

Filter language Example

- ***ColumnPrefixFilter***
  - ***example given: {FILTER=>"ColumnPrefixFilter('name')"}***
  - This filter takes one argument as column prefix
- ***SingleColumnValueFilter***
  - ***example given: {FILTER=>"SingleColumnValueFilter('cf', 'name', =, 'substring:lee')" }***
  - This filter as an argument takes a column family, a qualifier, a compare operator and a comparator.

Compound Filters and Operators

- You can combine multiple operators to create a hierarchy of filters, such as the following example:
  - (Filter1 AND Filter2) OR (Filter3 AND Filter4)
- Binary Operators
  - AND : If the AND operator is used, the key-value must satisfy both filters.
  - OR : If the OR operator is used, the key-value must satisfy at least one of the filters.
- Example Given:
  - scan 'table', {FILTER => "PrefixFilter('ro') AND ValueFilter(>=,'binary:value1')"}

Filter compare operator

- The following compare operators are provided:
  - LESS (<)
  - LESS_OR_EQUAL (<=)
  - EQUAL (=)
  - NOT_EQUAL (!=)
  - GREATER_OR_EQUAL (>=)
  - GREATER (>)
  - NO_OP (no operation)
- The client should use the symbols (<, <=, =, !=, >, >=) to express compare operators.

Comparator

- The general syntax of a comparator is ComparatorType:ComparatorValue
- The ComparatorType for the various comparators is as follows:
  - BinaryComparator : binary
  - BinaryPrefixComparator : binaryprefix
  - RegexStringComparator : regexstring
  - SubStringComparator : substring
- Example given
  - hbase> scan 'table', {FILTER => "ValueFilter(=,'substring:val')"}

Some comparator value example

- ***binary:abc*** will match everything that is lexicographically greater than "abc"
- ***binaryprefix:abc*** will match everything whose first 3 characters are lexicographically equal to "abc"
- ***regexstring:ab*yz*** will match everything that doesn't begin with "ab" and ends with "yz"
- ***substring:abc123*** will match everything that begins with the substring "abc123"

- hbase> scan 'table', {FILTER => "ValueFilter(=,'substring:value')"}
  # this scan will return all cells that begin with the substring "value"

Filter objects

- We can use Filter objects directly to filter the results. See the following page to get the Filter list and usage detail:
  - https://hbase.apache.org/1.2/apidocs/org/apache/hadoop/hbase/filter/package-summary.html
- Example given
  - hbase> scan 'table', {FILTER=>ValueFilter.new(CompareFilter::CompareOp.valueOf('EQUAL'), BinaryComparator.new(Bytes.toBytes('value1')))}

Challenge(3)

- Please scan following data from table 'ts:tags'
  - Q1: fetch 5 rows and only take column 'cf:tags' and 'cf:age' value information
  - Q3: find people whose row-id start with "M"
  - Q2: find people (rows) that column "cf:tags" contains tag "Taishin"
  - Q4: find people whose age is less than "40"
  - Q5: find people whose age is "50" and gender is "male"

Challenge(3) : Answer

- Q1: fetch 5 rows and only take column 'cf:tags' and 'cf:age' value information
  - scan 'ts:tags', {COLUMNS=>['cf:tags', 'cf:age'], LIMIT=> 5}
- Q2: find people whose row-id start with "M"
  - scan 'ts:tags', {FILTER=>"PrefixFilter('M')"}
- Q3: find people (rows) that column "cf:tags" contains tag "Taishin"
  - scan 'ts:tags', {COLUMN=>['cf:tags'], FILTER=>"ValueFilter(=, 'substring:Taishin')"}
- Q4: find people whose age is less than "40"
  - scan 'ts:tag', {FILTER=>"ValueFilter(=,'binary:40')"}
- Q5: find people whose age is "50" and gender is "male"
  - scan 'ts:tags', {FILTER=>"SingleColumnValueFilter('cf', 'age', =, 'binary:50') AND SingleColumnValueFilter('cf', 'gender', =, 'binary:male')"}

Advanced Usage

HBase performance

unit : operation / second

load process (node, operations/second)

1, 15,617.98
2, 23,373.93
4, 38,991.82
8, 74,405.64
16, 143,553.41
32, 296,857.36

5 nodes AWS i3 with SSD

update heavy: 18,763 (session store, recording recent action)
read mostly: 13,381 (photo tagging, data labeling)
read only: 13,372 (user profile, newsfeed cache)
read latest: 22,069 (user status)
short ranges: 2,508
read-modify-write: 12,843 (activity store, user databases)

https://www.datastax.com/products/compare/nosql-performance-benchmarks
https://www.intellectsoft.net/blog/hbase-vs-cassandra/

Supported Datatypes

- HBase supports a "bytes-in/bytes-out" interface via Put and Result, so anything that can be converted to an array of bytes can be stored as a value. Input could be strings, numbers, complex objects, or even images as long as they can rendered as bytes.

Joins

- Whether HBase supports joins is a common question on the dist-list, and there is a simple answer: it doesn't, at not least in the way that RDBMS' support them (e.g., with equi-joins or outer-joins in SQL). As has been illustrated in this chapter, the read data model operations in HBase are Get and Scan. However, that doesn't mean that equivalent join functionality can't be supported in your application, but you have to do it yourself.

Internal Table Operations

- HBase scalability is based on its ability to regroup data into bigger files and spread a table across many servers. To reach this goal, HBase has three main mechanisms: ***compactions***, ***splits***, and ***balancing***.

Compaction

- Minor compaction
  - A compaction is called minor when HBase elects only some of the HFiles to be compacted but not all
- Major compaction **table 級別**
  - We call it a major compaction when all the files are elected to be compacted together. A major compaction works like a minor one except that the delete markers can be removed after they are applied to all the related cells and all extra versions of the same cell will also be dropped.

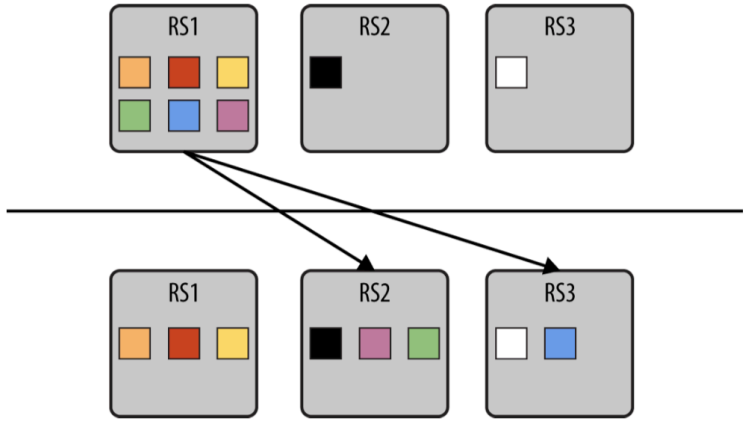
Splits (Auto-Sharding)

- Split is a very important function in Hbase. Hbase achieves load balancing by distributing data to a certain number of regions. A table is assigned to one or more regions, and these regions are assigned to one or more region Servers. In the automatic split strategy, when a region reaches a certain size, it will automatically split into two regions.

Balancing

ATHEMASTER

- Regions get split, servers might fail, and new servers might join the cluster, so at some point the load may no longer be well distributed across all your RegionServers. To help maintain a good distribution on the cluster, every five minutes (default configured schedule time), the HBase Master will run a load balancer to ensure that all the RegionServers are managing and serving a similar number of regions.



台新金控
Taishin Holdings

ATHEMASTER

課程名稱：【Hadoop從入門到進階(客群經營處包班)】
Hbase 實戰

課程講師：李偉僑

填寫期限：2020/01/16(四) 23:00截止

5 高 ⟷ 1 低

Reducing paper consumption !
Making the most of your time !

誠信 承諾 創新 合作

ATHEMASTER

Q & A

ATHEMASTER

THANK YOU

- THE PATHFINDER TO OPENSOURCE -

www.athemaster.com
info@athemaster.com
FB:athemaster