Columnar Access with HBase

Lecture BigData Analytics

Julian M. Kunkel

julian.kunkel@googlemail.com

University of Hamburg / German Climate Computing Center (DKRZ)

2017-12-15

Disclaimer: Big Data software is constantly updated, code samples may be outdated.

Outline

1 Introduction

2 Excursion: ZooKeeper

3 Architecture

4 Accessing Data

5 Summary

Overview of HBase [29, 30]

- Column-oriented key-value database for structured data
  - Based on Google's BigTable
  - Simple data and consistency model

Row	column1	column2	...
"Hans"	bla	19	...
"Julian"	NULL	20	...

- Scalable for billion of rows with millions of columns
  - Sharding of tables: distribute keys automatically among servers
  - Stretches across data centers
- Custom query language
  - Real-time queries
  - Compression, in-memory execution
  - Bloom filters and block cache to speed up queries
- Use HDFS and supports MapReduce
- Uses ZooKeeper for configuration, notification and synchronization
- Interactive shell (invoke `hbase shell`)

Data Model [29]

- Namespace: Logical grouping of tables for quota, security
- Table: A table (namespace:table) consists of multiple rows
- Row: Consists of a row key and many columns with values
  - Key/values are binary (converted from any data type)
  - WARNING: `hbase shell` stores all data as STRING
- Column: Consists of a column family and a qualifier (cf:q)
- Column family: string with printable characters
- Cell: Combination of row, column
  - Contains value (byte array) and timestamp (last modification)
- Timestamp: versions that change upon update

Student grading table (timestamps are not shown)

Row=Matrikel	a:name	a:age	l:BigData1718	l:Analysis1 12/13	...
stud/4711	Hans	19	1.0	2.0	...
stud/4712	Julian	20	NULL	1.7	...

Main Operations [29]

Data access

- **get**: return attributes for a row
- **put**: add row or update columns
- **increment**: increment values of multiple columns
- **scan**: iterate over multiple rows (potentially filtering)
- **delete**: remove a row, column or family
  - Data is only marked for deletion, finally removed during compaction

Schema operations

- **create**: create a table, specify the column families
- **alter**: change table properties
- **describe**: retrieve table/column family properties
- **list**: list tables
- **create_namespace**: create a namespace
- **drop_namespace**: remove a namespace

Example Interactive Session

```
$ create 'student', cf=['a','b'] # a,b are the column families
0 row(s) in 0.4820 seconds
$ put 'student', 'mustermann', 'a:name', 'max mustermann' # create column on the fly
$ put 'student', 'mustermann', 'a:age', 20
# we can convert 20 to a bytearray using Bytes.toBytes(20), otherwise it is a string
$ put 'student', 'musterfrau', 'a:name', 'sabine musterfrau'
$ scan 'student'
ROW          COLUMN+CELL
 musterfrau  column=a:name, timestamp=1441899059022, value=sabine musterfrau
 mustermann  column=a:age, timestamp=1441899058957, value=20
 mustermann  column=a:name, timestamp=1441899058902, value=max mustermann
2 row(s) in 0.0470 seconds
$ get 'student','mustermann'
COLUMN       CELL
 a:age       timestamp=1441899058957, value=20
 a:name      timestamp=1441899058902, value=max mustermann
2 row(s) in 0.0310 seconds
# Increment the number of lectures attended by the student in an atomic operation
$ incr 'student', 'max mustermann', 'a:attendedClasses', 2
COUNTER VALUE = 2
# delete the table
$ disable 'student' # deactivate access to the table
$ drop 'student'
```

Inspecting Schemas

- list <NAME>: List tables with the name, regex support

```
$ list 'stud.*'
TABLE
student
```

- describe <TABLE>: List attributes of the table

```
$ describe 'student'
COLUMN FAMILIES DESCRIPTION
{NAME => 'a', BLOOMFILTER => 'ROW', VERSIONS => '1', IN_MEMORY => 'false', KEEP_DELETED_CELLS => 'FALSE', DATA_BLOCK_ENCODING => 'NONE', TTL => 'FOREVER', COMPRESSION => 'NONE', MIN_VERSIONS => '0', BLOCKCACHE => 'true', BLOCKSIZE => '65536', REPLICATION_SCOPE => '0'}
{NAME => 'b', BLOOMFILTER => 'ROW', VERSIONS => '1', IN_MEMORY => 'false', KEEP_DELETED_CELLS => 'FALSE', DATA_BLOCK_ENCODING => 'NONE', TTL => 'FOREVER', COMPRESSION => 'NONE', MIN_VERSIONS => '0', BLOCKCACHE => 'true', BLOCKSIZE => '65536', REPLICATION_SCOPE => '0'}
```

- alter: Change table settings

```
# Keep at most 5 versions for the column family 'a'
$ alter 'student', NAME => 'a', VERSIONS => 5
Updating all regions with the new schema...
0/1 regions updated.
1/1 regions updated.
```

Remove Irrelevant Responses from Scans

- Scan options allow to restrict the rows/keys/values to be retrieved
- LIMIT the number of returned rows
- COLUMNS specify the prefix of columns/families
- ROWPREFIXFILTER restricts the row names

```
# filter columns using scan properties
$ scan 'student', {COLUMNS=>['a:age','a:name'], LIMIT=>2, ROWPREFIXFILTER =>'muster'}
ROW               COLUMN+CELL
 musterfrau    column=a:name, timestamp=1449395009213, value=sabine musterfrau
 mustermann    column=a:age, timestamp=1449395005507, value=20
 mustermann    column=a:name, timestamp=1449395001724, value=max mustermann

# scan rows with keys "STARTROW" <= "ROW" < "ENDROW"
$ scan 'student', {COLUMNS=>['a:age','a:name'], STARTROW => "muster", ENDROW =>
     ↪ "mustermann"}
musterfrau     column=a:name, timestamp=1449395009213, value=sabine musterfrau
```

Client Request Filters [30]

- Filters are Java classes restricting matches; overview `show_filters`
- Filter list: combines multiple filters with AND and OR
- Compare values of one or multiple columns
  - Smaller, equal, greater, substring, prefix, ...
- Compare metadata: column family and qualifier
  - Qualifier prefix filter: Return (first few) matching columns
  - Column range filter: return a slice of columns (e.g., bb-bz)
- Compare names of rows
  - Note: it is preferable to use scan options

Example in the hbase shell [32], [33]

```
# Apply regular filters
$ scan ’student’,{ FILTER => "KeyOnlyFilter()"}
 musterfrau       column=a:name, timestamp=1449395009213, value=
 mustermann       column=a:age, timestamp=1449395005507, value=
 mustermann       column=a:name, timestamp=1449395001724, value=
# return only rows starting with muster AND columns starting with a or b AND at most 2 lines
$ scan ’student’,{ FILTER => "(PrefixFilter (’muster’)) AND MultipleColumnPrefixFilter(’a’,’b’) AND ColumnCountGetFilter(2)" }
 mustermann        column=a:age, timestamp=1449395005507, value=20
$ scan ’student’,{ FILTER => "SingleColumnValueFilter(’a’,’name’,=,’substring:sabine musterfrau’)"}
 musterfrau                          column=a:name, timestamp=1449395009213, value=sabine musterfrau
# return all students older than 19
$  scan ’student’,{ COLUMNS=>[’a:age’], FILTER => "SingleColumnValueFilter(’a’,’age’,>,’binary:19’)"}
 mustermann       column=a:age, timestamp=1449407597419, value=20
```

Consistency [29]

- Row keys cannot be changed
- Strong consistency of reads and writes
- Mutations are typically atomic (no partial succeed)
  - Multiple column families of one row can be changed atomically
  - Order of concurrent mutations not defined
  - Successful operations are made durable
- Mutations of multiple rows are not atomic (need more than one API call)
- The tuple (row, column, version) specifies the cell
  - Normally version is the timestamp, but can be changed
  - The last mutation to a cell defines the content
  - Any order of versions can be written (max number of versions defined by cf)
- Get and scan return recent versions but maybe not the newest
  - A get may return an old version but between subsequent gets the version may never decrease (no time travel)
  - Any row returned must be consistent (isolates ongoing column mutations)
  - A scan must return all mutations completed before it started
    - It MAY contain later changes
  - Content read is guaranteed to be durable
- Deletes masks (hides) newer puts until compaction is done

Co-Processors [43]

- Coprocessor concept allow to compute functions based on column values
- Similar to database triggers
- Hooks are executed on the RegionServers implemented in observers
- Can be used for secondary indexing, complex filtering and access control
- Scope for the execution
  - All tables (system coprocessors)
  - On a table (table coprocessor)
- Observer intercepts method invocation and allows manipulation
  - RegionObserver: intercepts data access routines on RegionServer/table
  - WALObserver: intercepts write-ahead log, one per RegionServer
  - MasterObserver: intercepts schema operations
- Currently must be implemented in Java
- Can be loaded from the hbase shell

1 Introduction

2 Excursion: ZooKeeper

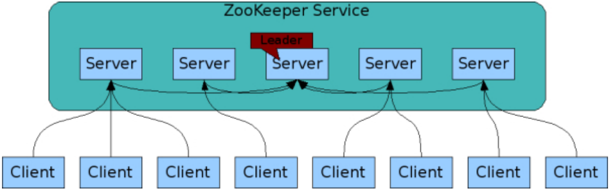
3 Architecture

4 Accessing Data

5 Summary

Zookeeper Overview [39, 40]

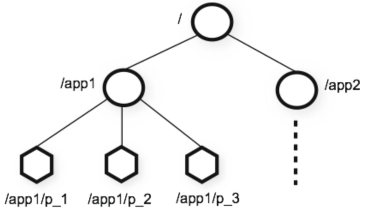
- Centralized service providing
  - Configuration information (e.g., service discovery)
  - Distributed synchronization (e.g., locking)
  - Group management (e.g., nodes belonging to a service)
- Simple: Uses a hierarchical namespace for coordination
- Strictly ordered access semantics
- Distributed and reliable using replication
- Scalable: A client can connect to any server



Source: ZooKeeper Service [40]

Hierarchical Namespace [40]

- Similar to file systems but kept in main memory
- znodes represent both file and directory



Source: ZooKeeper's Hierarchical Namespace [40]

Nodes

- Contain a stat structure: version numbers, ACL changes, timestamps
- Additional application data is read together with stats
- Watch can be set on a node: triggered once when a znode changes
- Ephemeral nodes: are automatically removed once the session that created them terminates (e.g., server crashes)

Consistency Guarantees

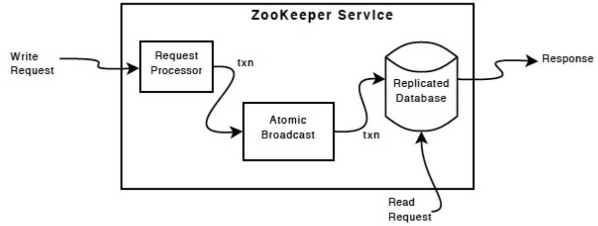
- Atomicity: no partial results
- Single System Image: same data regardless to the server connected
- Reliability: an update is persisted
- Timeliness: a client's view can lack behind only a certain time
- Optional: sequential consistency
  - Updates are applied in the order they are performed
  - Note: znodes need to be marked as sequential, if this is needed

Reliability: Server failures are tolerated

- Quorum: Reliable as long as $ceil(N/2)$ nodes are available
- Uses Paxos consensus protocols with atomic message transfer

Architecture: Updating Data [40]

- Writes are serialized to storage before applied to the in-memory db
- Writes are processed by the agreement protocol Paxos
  - All writes are forwarded to the leader server
  - Other servers receive message proposals and agree upon delivery
  - Leader calculates when to apply the write and creates a transaction



Source: ZooKeeper Components [40]

1 Introduction

2 Excursion: ZooKeeper

3 Architecture

4 Accessing Data

5 Summary

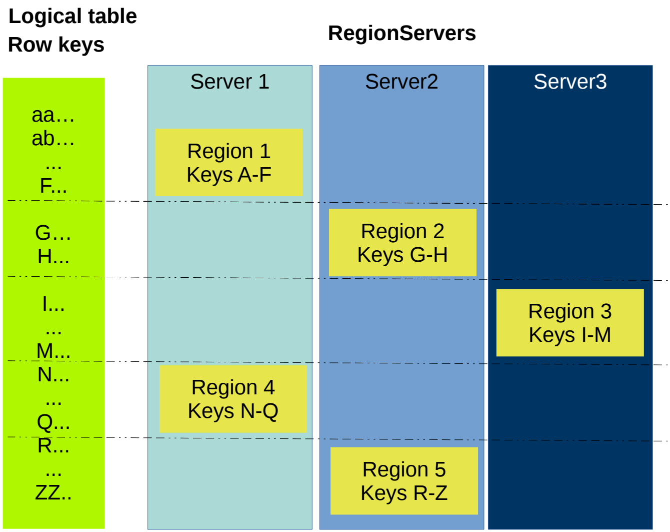
Distribution of Data [30]

- HBase uses HDFS as backend to store data
  - Utilize replication and place servers close to data
- Server (RegionServer) manage key ranges on a per table bases
  - Buffer I/O to multiple files on HDFS
  - Performs computation (and data filtering)
- Regions: base element for availability and distribution of tables
  - One **store** object per ColumnFamily
  - One Memstore for each **store** to write data to files
  - Multiple StoreFiles (HFile format) for each store (each sorted)
- Catalog Table *HBase:meta* (not splittable)
  - Contains a list of all regions $< table >, < regionstartkey >, < regionid >$

Table splitting

- Upon initialization of a table only one region is created
- Auto-Splitting: Based on a policy, a region is split into two
  - Typical policy: split when the region is sufficiently large
  - Benefit: increases parallelism, automatic scale-out
- Manual splitting can be triggered

Sharding of a Table into Regions



Distribution of keys to servers, values are stored with the row

Storage Format [30]

HFile format [35]

- Cell data is kept in store files on HDFS
- Sorted by row key
- Multi-layered index with bloom filters and snapshot support
- Append only, deletion writes key type with tombstone marker
- Compaction process merges multiple store files

Row Length *short*	**Row Key** *byte[]*	Family Length *byte*	Column **Family** byte[]	Column **Qualifier** *byte[]*	**Timestamp** *long*	**Key Type** *byte*

Record format. Source: [36]

Storage Format [30]

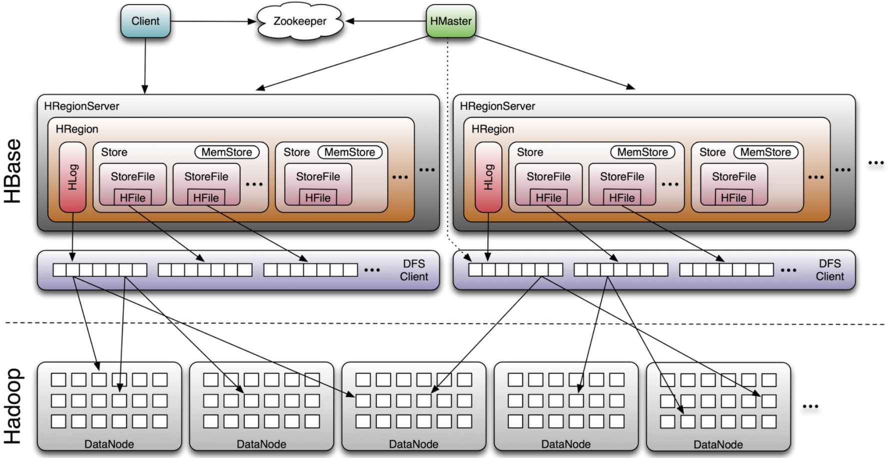
- Write Ahead Log (WAL) – stored as sequence file
  - Record all planned data changes before doing them
  - Ensure durability by enabling replay when server crashes
- Medium-sized Objects (MOB)
  - HBase is optimzed for values $\leq 100KB$
    - Larger objects degrade performance for splits, compaction
  - MOBs are stored in separate files on HDFS and referenced by HFiles
  - Example: Add support for MOB to the column family pic

```
alter 'stud', {NAME => 'pic', IS_MOB => true, MOB_THRESHOLD => 102400}
```

Architecture Components and Responsibilities [30]

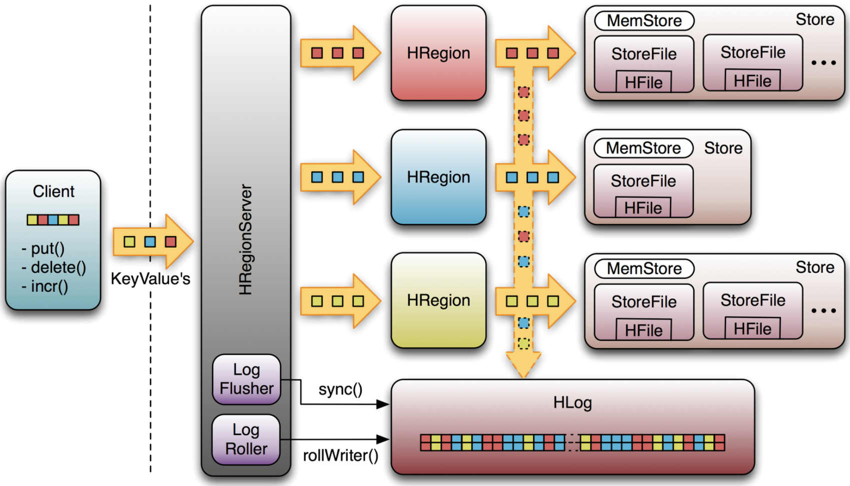
- Master
  - Monitor RegionServer
  - Runs LoadBalancer to transfer regions between servers
  - CatalogJanitor: check and clean the meta table
  - Typically runs on HDFS NameNode
- RegionServer
  - Hosts a subsequent span of keys (Region) for tables
  - Executes Client Request Filters
  - Runs periodic compaction
  - Typically runs on HDFS DataNode
  - Memstore: accumulates all writes
    - If filled, data is flushed to new store files
    - Multiple smaller files can be compacted into fewer
    - After flushes/compaction the region may be split
- Client
  - Identify location of *HBase:meta* from ZooKeeper
  - Query *HBase:meta* for identifying the RegionServers
  - May use Client Request Filters

High-Level Perspective of HBase File Mapping



Mapping of logical files to file blocks. Source: [38]

Write-Path and the Write-Ahead Log [39]



Write-path: Updates of rows 1) trigger writes to WAL, 2) modify the memstore, 3) batch modifications are issued to HFiles. Source: [39]

Caching of Data [30]

- MemStore caches writes and batches them
  - Exists per Region, sorts rows by key upon write
- BlockCache keeps data read in block-level granularity
  - One shared pool per RegionServer
- Access to rows/values is cached via LRU or BucketCache
- Cached data can be compressed in memory
- LRU keeps data in Java heap
- LRU eviction priority changes with access pattern and setup
  1. Single access priority: when a block is loaded into memory
  2. Multi access priority: block was repeatedly accessed
  3. Highest priority: in-memory, configurable in the ColumnFamily
- BucketCache is a two tier cache with L1 LRU (memory) and L2 in file
- CombinedCache: data in BucketCache, indices/bloom in LRU

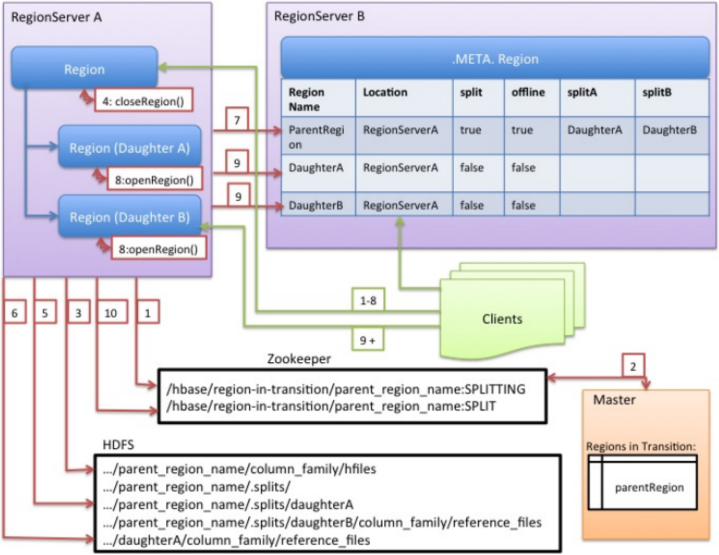
Implications of the Storage Schema

- Row keys and data
  - Rows are distributed across RegionServers based on the key
  - The key-prefix of rows close together is similar
    - With reversed URLs, de.dkrz.www/x is close to de.dkrz.internal/y
  - Different access patterns should be handled by different column families
  - Rows are always sorted by the row key and stored in that order
  - Similar keys are in the same HDFS file/block
  - Wrong insertion order creates additional HFiles!
- Column family: string with printable characters
  - Tunings and storage options are made on this level
  - All cf members are stored together and managed by a MemStore
- Reading data
  - MemStore and store files must be checked for newest version
  - Requires to scan through all HFiles (uses BloomFilters)

Splitting of Regions [30]

1. The memstore triggers splitting based on the policy
   - Identify the split point in the region to split into half
2. Notify Zookeeper about the new split and create a znode
   - The master knows this by watching for the znode
3. Create .splits subdirectory in HDFS
4. Close the parent region and mark it as offline
   - Clients cannot access regions but will retry access with some delay
5. Create two new region directories for daughter regions.
   Create *reference files* linking to the bottom and top part per store file
6. Create new region directory in HDFS and move all daugther reference files
7. Send a put request to the meta table, setting parent offline and adding new daugthers
8. Open daugthers
9. Add daugthers to meta table and be responsible for hosting them. They are now online
   - Clients will now learn about the new regions from the meta table
10. Update the znode in Zookeeper
    - The master now learns that split transaction complete
    - The LoadBalancer can re-assign the daughter regions to other region servers
11. Gradually move data from parent store files to daugther reference files during compaction
    - If all data is moved, delete the parent region

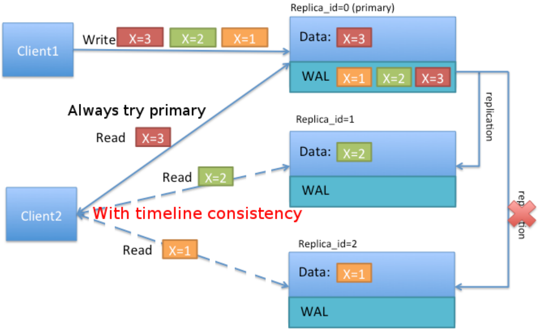
Splitting of Regions



Source: RegionServer Split Process [30]

Tunable Semantics: Reduce Guarantees

- Durability can be weakened by flushing data only periodically
- Visibility of each read can be changed [30]
  - Normally strong consistency accesses only from primary replica
  - Timeline consistency enables use of other replicas, if timeout
    - May cause reading of older versions (eventual consistency)

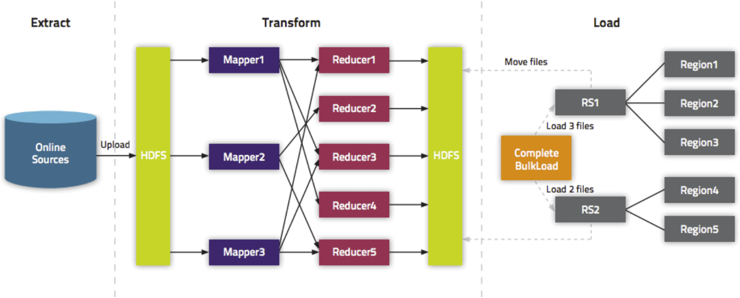


Source: Timeline Consistency [30]

Bulk Loading [31]

General process (ETL)

1. Extract data (and usually import it into HDFS)
2. Transform data into HFiles using MapReduce
3. Load files into HBase by informing the RegionServer



Source: [31]

Bulk Loading (2) [31]

Transform step

- Either replace complete dataset or incremental loading (update)
- Bypasses the normal write path (WAL)
- Create one reduce job per Region

Alternatives

- Original dataset loading
  - Replaces data in the table with all data
  - You have to specify key mappings/splits when creating the table
  - Hbase ships with importtsv mapreduce job to perform the import as strings
  - Importtsv replaces the existing files with converted HFiles from the CSV
- Incremental loading
  - Triggers minor compaction
  - No replication of data!

Support for MapReduce [30]

- HBase can be a data source and/or data sink
  - At least (# of regions) mapper jobs are run
  - Java: TableInputFormat / Output, MultiTableOutputFormat
  - One table can be natively read with MR task, multiple explicitly
- HRegionPartitioner for load-balancing output
  - Each reducer stores data to a single region
- Tool for accessing table: HBase-server-VERSION.jar

```
$ hadoop jar ${HBase_HOME}/HBase-server-VERSION.jar <Command> <ARGS>
```

Operations:

  - Copy table
  - Export/Import HDFS to HBase
  - Several file format importers
  - Rowcounter

MapReduce Example Reading from one Table [30]

```java
public static class MyMapper extends TableMapper<Text, Text> {
  public void map(ImmutableBytesWritable row, Result value, Context context) throws InterruptedException, IOException {
    // process data for the row from the Result instance.
   }
}

Configuration config = HBaseConfiguration.create();
Job job = new Job(config, "ExampleRead");
job.setJarByClass(MyReadJob.class);     // class that contains mapper
Scan scan = new Scan();
scan.setCaching(500);        // the default 1 is be bad for MapReduce jobs
scan.setCacheBlocks(false);  // don't set to true for MR jobs
// set other scan attrs ...
TableMapReduceUtil.initTableMapperJob(
  tableName,        // input HBase table name
  scan,             // Scan instance controls column family and attribute selection
  MyMapper.class,   // mapper
  null,             // mapper output key
  null,             // mapper output value
  job);
job.setOutputFormatClass(NullOutputFormat.class);   // because we aren't emitting anything from the mapper but storing data in HBase
if (! job.waitForCompletion(true) ) {
  throw new IOException("error with job!");
}
```

HBase Support in Hive [42]

- HiveQL statements access HBase tables using SerDe
- Row key and columns are mapped in a flexible way
- Preferably: Use row key as table key for relational model
- Supported storage types: string or binary

```
CREATE TABLE hbase_table(key int, value string)
STORED BY 'org.apache.hadoop.hive.hbase.HBaseStorageHandler'
WITH SERDEPROPERTIES ("hbase.columns.mapping" = ":key,cf1:myval#binary")
TBLPROPERTIES ("hbase.table.name" = "xyz");
```

- Hive map with string key can be used to access arbitrary columns

```
# use a map, all column names starting with cf are keys in the map
# without hbase.table.name, table name is expected to match hbase tbl
CREATE TABLE hbase_table(row_key int, value map<string,int>)
STORED BY 'org.apache.hadoop.hive.hbase.HBaseStorageHandler'
WITH SERDEPROPERTIES ( "hbase.columns.mapping" = ":key,cf:" );
```

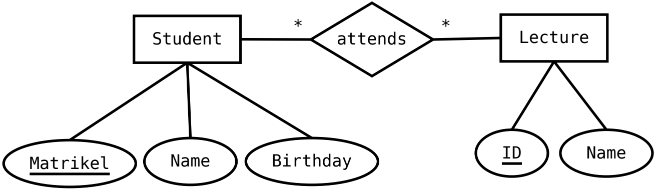
- HBase composite keys can be used as struct (terminator must be set)

```
CREATE EXTERNAL TABLE delimated(key struct<f1:string, f2:string>, value string)
ROW FORMAT DELIMITED COLLECTION ITEMS TERMINATED BY '~'
STORED BY 'org.apache.hadoop.hive.hbase.HBaseStorageHandler'
WITH SERDEPROPERTIES ('hbase.columns.mapping'=':key,f:c1');
```

Schema Design Guidelines [29]

- Keep the cardinality of column families small
- Prevent hotspotting in row key design
  - As rows with related keys are stored together, this may cause bottlenecks
  - Salting (adding a prefix randomly), increases write but decreases reads
  - Hashing: Add a hash value as prefix
  - Reversing the key
- Prevent writes on monotonically increasing row keys
  - Timestamps or sequences should not be the row key
- Reduce size of row, column familiy and attribute names
  - Goal: save network bandwith and memory for cell coordinates
  - Example: table student should be abbreviated st
  - Use binary representations instead of strings
- Finding the most recent version of a row
  - Use <orignal key><ReverseTimestamp> as key
  - Scan for <orignal key> will return the newest key

Example Mapping of an Entity Relationship Diagram



Our student lecture example

Possible mapping (uses short names)

- Table students (st)
  - Row key: reverse matrikel(mr) ⇒Avoid re-partitioning
  - Columns: Name(n), birthday(bd), attends as columns for each <lecture id>
- Table lecture (lc)
  - Row key: ID (e.g., year-abbreviation)
  - Columns: Name (n), attendees columns for each <matrikel>
- We may add tables to map names to lecture/student IDs

Summary

- HBase is a wide-columnar storage
- Data model: key (row), columnfamiliy:column, values
- Main operations: put, get, scan, increment
- Strong consistency model returns newest version
- Sharding distributes keys (rows) across servers
- HFile format appends modifications
- Automatic region splitting increases concurrency (but complex)
- Schema design can be tricky
- ZooKeeper manages service configuration and coordinates applications

Bibliography

10 Wikipedia

29 http://HBase.apache.org/

30 http://HBase.apache.org/book.html

31 http://blog.cloudera.com/blog/2013/09/how-to-use-hbase-bulk-loading-and-why/

32 http://www.hadooptpoint.com/filters-in-hbase-shell/

33 http://www.cloudera.com/content/www/en-us/documentation/enterprise/latest/topics/admin_hbase_filtering.html

34 http://www.myhadoopexamples.com/2015/06/19/hbase-shell-commands-in-practice/

35 http://de.slideshare.net/Hadoop_Summit/hbase-storage-internals

36 http://blog.cloudera.com/blog/2012/06/hbase-io-hfile-input-output/

37 http://www.larsgeorge.com/2010/01/hbase-architecture-101-write-ahead-log.html

38 http://www.larsgeorge.com/2009/10/hbase-architecture-101-storage.html

39 https://zookeeper.apache.org/

40 https://zookeeper.apache.org/doc/trunk/zookeeperOver.html

41 http://zookeeper.apache.org/doc/trunk/zookeeperProgrammers.html

42 https://cwiki.apache.org/confluence/display/Hive/HBaseIntegration

43 https://blogs.apache.org/hbase/entry/coprocessor_introduction