Julian Kunkel

Columnar Access & Document Storage

Learning Objectives

- Create a Columnar Data Model (for HBase) for a given use case
- Justify the reasons and implications behind the HBase storage format
- Describe how HBase interacts with Hive and Hadoop
- Describe the features and namespace handling in Zookeeper
- Create a Document Data Model (for MongDB) for a given use case
- Provide example data (JSON) for the MongoDB data model and the queries
- Sketch the mapping of keys to servers in MongoDB and HBase
- Select and justify a suitable shard key for a simple use case

Outline

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 consists of rows; name: (namespace:table)
- 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; group by requirement
- Cell: Combination of row, column
  - Contains value (byte array) and timestamp (last modification)
- Versioning: upon update update timestamp, can keep multiple versions

Table: 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 (flexible columns!)
- **alter**: change table properties
- **describe**: retrieve table/column family properties
- **list**: list tables
- **create_namespace**: create 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

```
1  # filter columns using scan properties
2  $ scan 'student', {COLUMNS=>['a:age','a:name'], LIMIT=>2, ROWPREFIXFILTER =>'muster'}
3  ROW            COLUMN+CELL
4   musterfrau    column=a:name, timestamp=1449395009213, value=sabine musterfrau
5   mustermann    column=a:age, timestamp=1449395005507, value=20
6   mustermann    column=a:name, timestamp=1449395001724, value=max mustermann
7
8  # scan rows with keys "STARTROW" <= "ROW" < "ENDROW"
9  $ scan 'student', {COLUMNS=>['a:age','a:name'], STARTROW => "muster", ENDROW => "mustermann"}
10 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)
  - Columns of 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 old version but between subsequent gets the version never decrease
    - **No time travel**
  - Any row returned must be consistent (isolates ongoing column mutations)
  - A scan returns all mutations completed before started and **may** contain later changes
  - Content read is guaranteed to be durable
- Deletes masks (hides) newer puts until compaction is done

Co-Processors [43]

- Run custom code on Region Server
- 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

Outline

Zookeeper Overview [39, 40]

- Centralized service for coordination 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

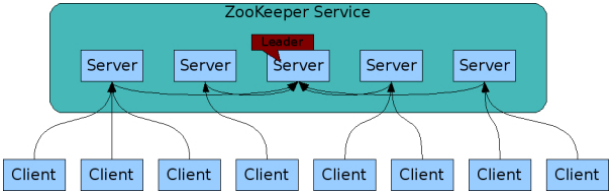


Figure: Source: ZooKeeper Service [40]

Hierarchical Namespace [40]

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

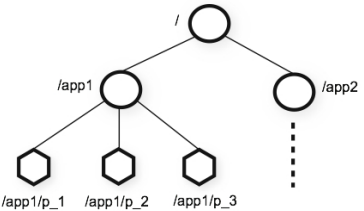


Figure: Source: ZooKeeper's Hierarchical Namespace [40]

Nodes

- Contain metadata (query with stat()): 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

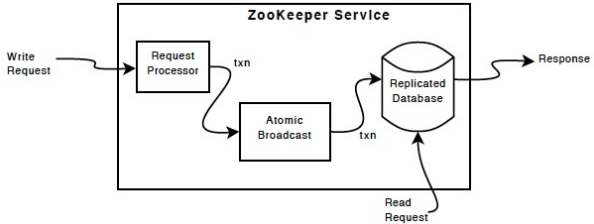


Figure: Source: ZooKeeper Components [40]

Outline

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*, special non splittable table
  - 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

Sharding of a Table into Regions

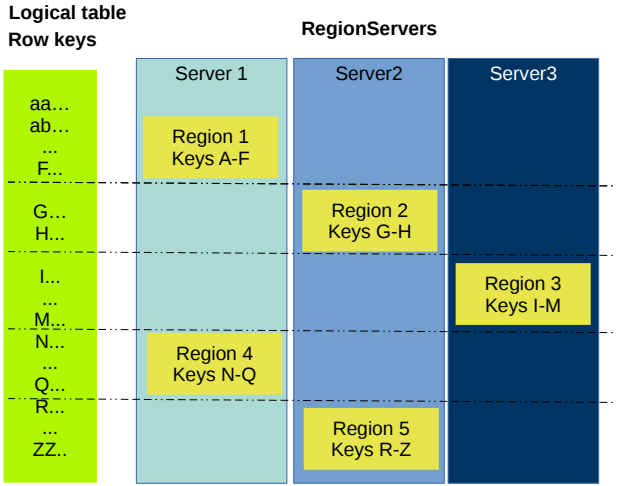


Figure: Distribution of keys to servers, values are stored with the row

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*

Figure: Record format. Source: [36]

Storage Format

- Strings are highly redundant (due to sorting by key)
- Encoding can optimize storage space

Key Len	Val Len	Key	Value
24	...	RowKey:family:qualifier0	...
24	...	RowKey:family:qualifier1	...
24	...	RowKey:family:qualifierN	...
25	...	RowKey2:family:qualifier1	...
25	...	RowKey2:family:qualifier2	...
...	...	...	...

No Encoding

Key Len, Value Len
Key Bytes, Val Bytes
(since the keys are sorted, each row looks very similar)

Prefix Encoding

Extra column that contains "common length" bytes equals in the previous row

Store just the differences

Key Len	Val Len	Prefix Len	Key	Value
24	...	0	RowKey:family:qualifier0	...
1	...	23	1	...
1	...	23	N	...
19	...	6	2:family:qualifier1	...
1	...	24	2	...
...	...	...	...	...

Figure: Storage format. Source: [45]

Storage Format [30]

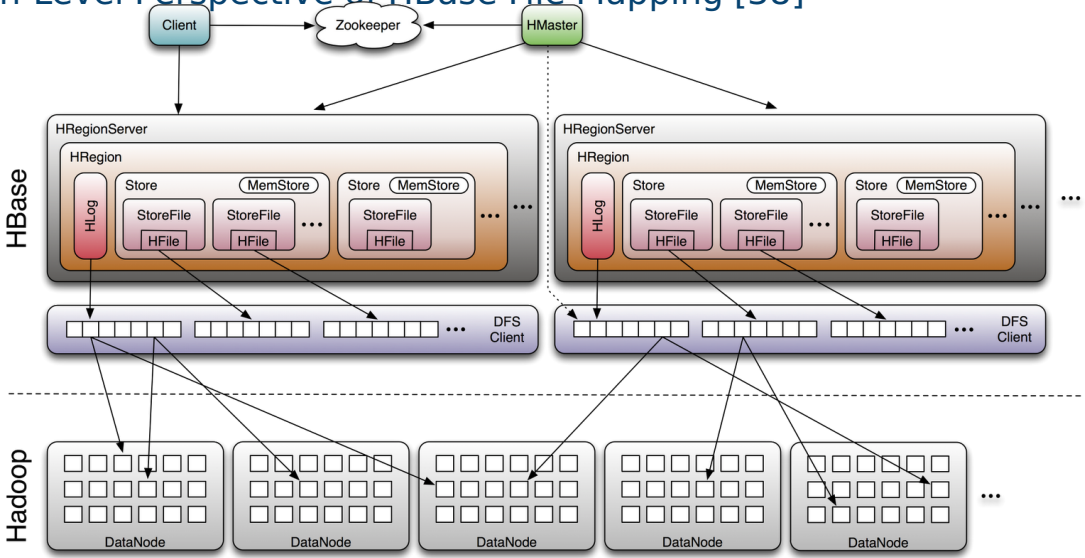
- Write Ahead Log (WAL) – stored as sequence file
  - Record all planned data changes before doing them
  - Ensure durability by enabling replay when server crashes
- Medium-sized Objects (MOB)
  - HBase is optimzed for values $\leq 100KB$
    - Larger objects degrade performance for splits, compaction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

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

Architecture Components and Responsibilities [30]

- Master
  - Monitors RegionServer and manages hbase:meta table
  - LoadBalancer transfers regions, CatalogJanitor garbage clean
  - Typically runs on HDFS NameNode
- RegionServer
  - Hosts a subsequent span of keys (Region) for tables
  - Executes Client Request Filters
  - Runs periodic compaction
  - 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
  - Typically runs on HDFS DataNode
- 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 [38]



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

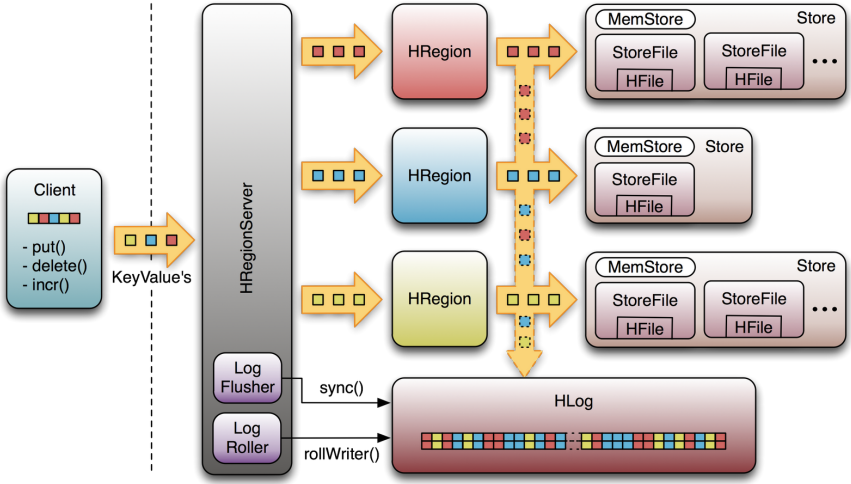


Figure: Write-path: Updates of rows 1) trigger writes to WAL, 2) modify the memstore, 3) batch modifications are issued to HFiles. Source: [39]

Caching of Data [30]

- MemStore caches writes and batches them
  - Exists per Region (and ColumnFamily), 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 the split transaction completed
    - 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

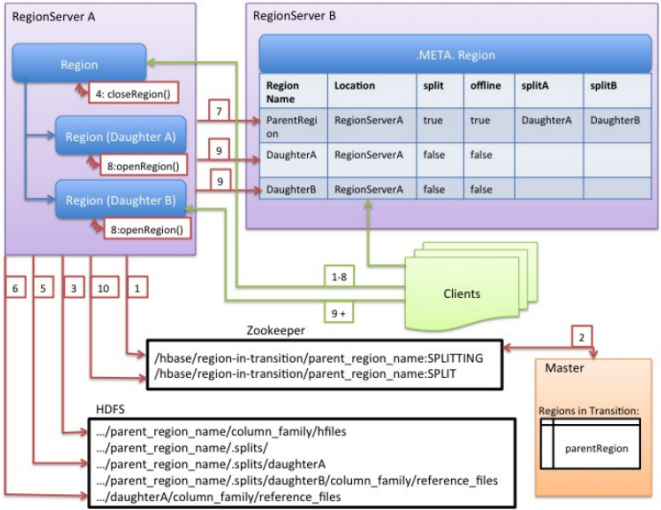


Figure: Source: RegionServer Split Process [30]

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)

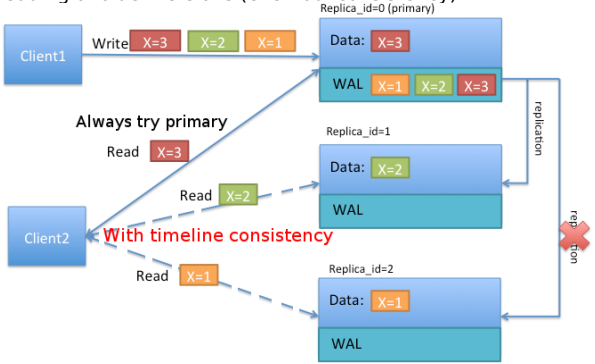


Figure: Source: Timeline Consistency [30]

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

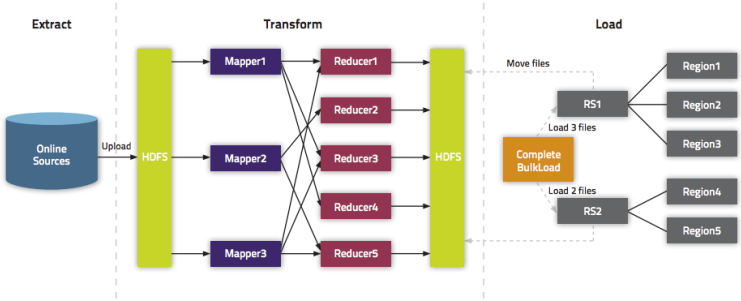


Figure: Source: [31]

Bulk Loading (2) [31]

Transform step

- Either replace complete dataset or incremental loading (update)
- Bypasses the normal write path (especially, the 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
  - Note: no replication of data is performed

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

```
1 $ 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]

```
1  public static class MyMapper extends TableMapper<Text, Text> {
2    public void map(ImmutableBytesWritable row, Result value, Context context) throws InterruptedException,
         ↪ IOException {
3      // process data for the row from the Result instance.
4  }  }
5  Configuration config = HBaseConfiguration.create();
6  Job job = new Job(config, "ExampleRead");
7  job.setJarByClass(MyReadJob.class);     // class that contains mapper
8  Scan scan = new Scan();
9  scan.setCaching(500);        // the default 1 is be bad for MapReduce jobs
10 scan.setCacheBlocks(false);  // don't set to true for MR jobs
11 // set other scan attrs ...
12 TableMapReduceUtil.initTableMapperJob(
13   tableName,        // input HBase table name
14   scan,             // Scan instance controls column family and attribute selection
15   MyMapper.class,   // mapper
16   null,             // mapper output key
17   null,             // mapper output value
18   job);
19 job.setOutputFormatClass(NullOutputFormat.class);   // because we aren't emitting anything from the mapper
       ↪ but storing data in HBase
20 if (! job.waitForCompletion(true) ) {
21   throw new IOException("error with job!");
22 }
```

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
- HBase stores data either as string or binary

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

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

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

- Example:

```
1 INSERT OVERWRITE TABLE hbase_table SELECT map(key, value), row FROM tbl WHERE row>100;
```

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 inserts of 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

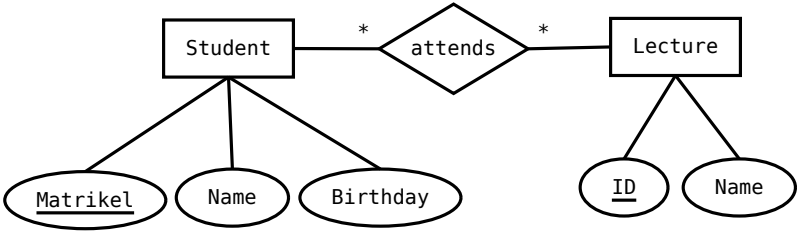


Figure: Our student lecture example

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

Outline

Data Model

- Documents contain semi-structured data (JSON, XML)
- Each document can contain data with other structures
- Addressing to lookup documents are implementation-specific
  - E.g., bucket/document key, (sub) collections, hierarchical namespace
- References between documents are possible
- Example technology: **MongoDB**, Couchbase, DocumentDB

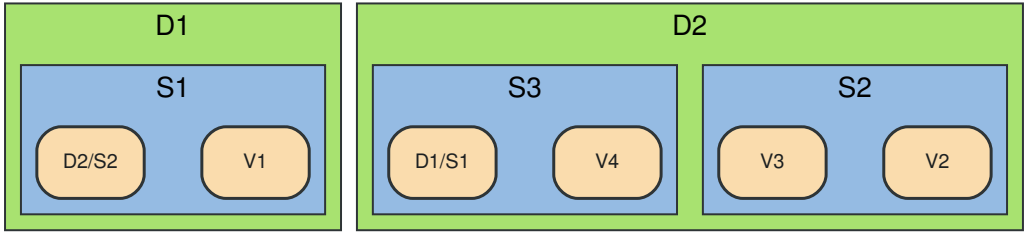


Figure: Source: Document Store. The Neo4j Manual v2.2.5 [33]

D=Document, S=Subdocument, V=Value, X/Y=reference to a subdocument in another document

MongoDB [11]

- Open-source document database
- High-performant and horizontally scalable for clusters
- Interfaces: Interactive mongo shell, REST, C, Python, ...
  - Connector for Hadoop for reading/writing to MongoDB

Data Model

- Database: As usual, defines permissions
- Document: BSON object (binary JSON) – Consisting of subdocuments
  - Primary key: _id field (manually set or automatically filled)

```
1  "_id" : ObjectId("43459bc2341bc14b1b41b124"),
2  "students" : [ # subdocument
3    { "name" : "Julian", "id" : 4711, "birth" : ISODate("2000-10-01")},
4    { "name" : "Hans",   "id" : 4712, "birth", ... }  ]
```

- Collection: Like a table of documents
  - Addressing: Collection name, document _id field (choose appropriately)
  - Documents can have individual schemas
  - Support for indexes on fields (and compound fields)
- Document references via object ids

Object ID

- Object ID must be unique
- By default ObjectIDs are 12 bytes and automatically generated
  - Example: ObjectId("43459bjebc2341bc14b1b41b124")
  - 4-byte time value representing the seconds since the Unix epoch
  - 3-byte machine identifier
  - 2-byte process id
  - 3-byte counter, starting with a random value
- If a manual ID is chosen, conflicts must be prevented
- By default data is distributed on the object ID into shards...

Operations for the Data

- Documents: insert (create), find (read), update, delete (CRUD)
  - Sort, aggregate: use accumulators to aggregate fields
- Collections: create and drop (remove) collections
  - Automatically created when the first document is inserted
- Schemas via document validation
  - When creating a collection, a validator can be defined
  - It is checked upon insert/update
  - Triggers an action: warning or reject the changes

Semantics [14]

- The _id field is always created. You can also define a (unique) id as a string!
- Atomicity on document level: Changes only one document at a time
  - All fields that must be updated together must be part of one doc
- Durability: Flexibly; users can define a “write concern”
- Concurrency: read/write exclusive locks are used internally
- Bulk operations are supported

Query Documents [13]

- Operations select the document to operate by defining documents
  - Example: Lookup of a document using find(<query>, <projection>)
    - Query: Defines the relevant documents (filtering)
    - Projection: The matched elements from JSON, e.g., returns the first array element
- Properties of a query (filter) document restrict the query:
  - Select all: {} (empty JSON)
  - Select documents with the key and value: { key : value }
  - Comparators: $eq, $lt, $ne (not equal)
  - Compare with sets: $in and $nin; { key : { $in: [value1, value2] } }
  - Logical: $and, $or, $not, $nor, $exists; { $or : [key : val, alt key, alt val] ; }
  - Text search: $text, $regex, $where (JavaScript expression)
  - Geospatial query operators: $geoWithin, $near, $minDistance
- Subdocuments can be addressed using the dot notation
  - Example query document: { "students.age" : { $gt : 15} }

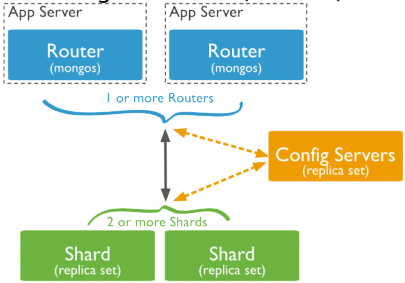
Outline

Architecture

- Shard: MongoDB server or *replica set* responsible for a set of data
- Replica set: Server cluster implementing master-slave replication/failover

Components

- Config server: Replica set stores metadata and replication information
- Mongos: Query router between client and replica set
- Mongod: MongoDB shard server providing storage space
- Balancer migrates data (chunks) between the servers



Source: Reference [14]

Accessing Sharded Data [14]

- Sharding (and options) are set on the collection level

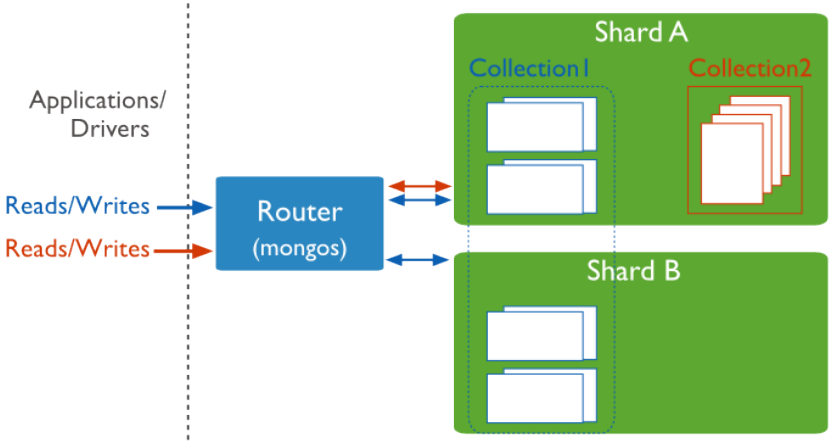


Figure: Source: Reference [14]

Partitioning of Data (One Collection) [14]

- Shard key: Immutable field(s) in every collection document
  - Either by hashing of fields or by distributing ranges
  - Performance relevant: Select an appropriate shard key
- Chunk: A contiguous range of shard key values
  - Chunks are automatically split and migrated between shards

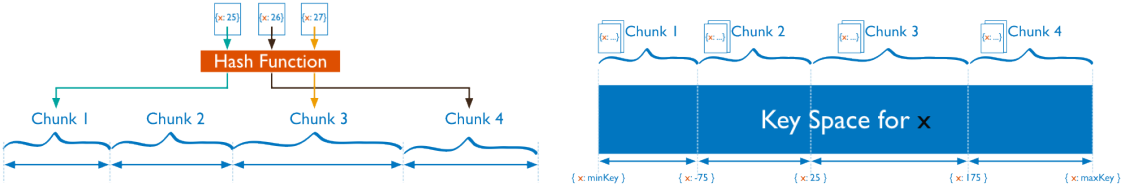
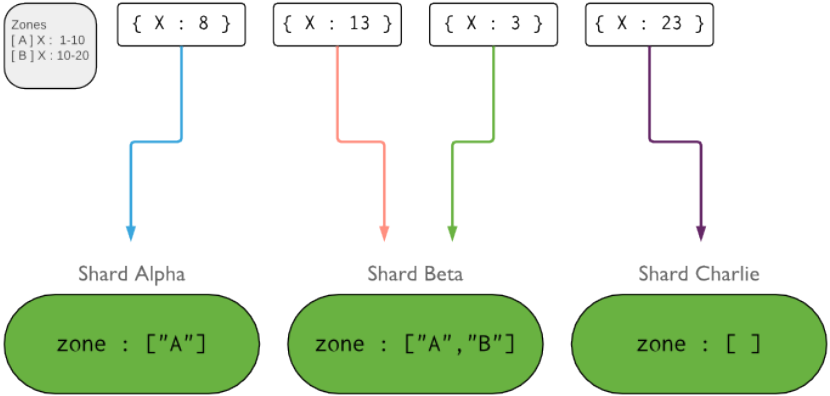


Figure: Hash and ranged sharding – Source: Reference [14]

- Internal processing of queries
  - Broadcast (scatter-gather) necessary if the query filter does not contain the shard key
  - If shard key is part of the query, only the subset of servers is contacted

Zones [14,18]

- Goals: Improve locality of data, distribute data across data centres
- Zone: Group documents based on the value of the shard key
- Create a tag for shards (e.g., Frankfurt-DC) matching a key range
- A shard (server/replica set) may be assigned to multiple zones
- Migration of chunks is done only within its origin zone



Optimal Selection of Shard Keys

Query handling

- Mongos broadcasts query to all shards if cannot identify subset using shard key

Guidelines [46]

- All inserts, updates, and deletes would each be **distributed uniformly** across all of the shards in the cluster
- All queries would be uniformly distributed across all of the shards in the cluster
- All operations would only target the shards of interest: an update or delete would **never be sent to a shard** which didn't own the data being modified
- A query isn't sent to a shard which holds none of the data being queried

Group Work

- Think about the selection of shard keys (MongoDB [44]) / row keys (HBase)
- Use Case
  - Global registry for cars
  - Data: Car registration data (owner ID, country, plate ID, car type, car color, ...)
  - An owner can have multiple cars ...
  - Write pattern (less frequent than queries though)
    - Inserting new data set
    - Deleting a data set
    - Rarely: bulk insertion of all plate IDs for a given country
  - Query pattern
    - Retrieve data for a given "country" + "plate ID" OR by "owner"
    - Rarely: search for cars with given properties (in a country or globally)
- Time: 10 min
- Organization: breakout groups - please use your mic or chat

Solution: HBase

Data Model

- Depends on the importance of the two query patterns
- Could store two rows per entry to optimize both queries
  - One per owner; Row ID = owner ID
    - Columns: for all plate IDs = 1 that the owner owns
  - One per car: Row ID = ("plate ID", country)
    - Store plate ID first to prevent one zone per country
    - Could prefix with hash (to N zones) to distribute it even further
    - Columns: features
- Drawbacks
  - Searching for a car with given properties is expensive (bloom filters help though)
  - Full table scan could be prevented, if the row sorting order would be (country, plateID)
- One column family for owner and one per car

Solution: MongDB

Data Model

- Create collection owner with one document per owner with set of car IDs owned
  - Sharding key = _id : owner ID
  - Could also use typical object ID
  - Automatically hashed by MongoDB
- Create collection cars with one document per car
  - Sharding key = _id = country, plate ID
  - Automatically hashed by MongoDB
- Drawbacks
  - Broadcast is triggered when searching for given properties (with index propably OK)
  - Must create index of relevant search properties

Outline

MongoDB Shell [12]

Start by invoking the `mongo` command

Commands

- `<X>.help`: Show help for obj X
- `show collections`: Print the existing collections
- `db.<COL>`: Access the collection COL
- Collection operations:
  - `find(query)`: Search for a document with properties according to doc
  - `insert(query)`: Insert
  - `update(query,update)`: Update
  - `remove(query)`: Delete all matching documents
  - `drop()`: Remove the collection discarding all data
  - `createIndex(doc)`: Create an index for all listed fields
  - `sort(doc)`: Sort documents based on the keys in the doc
  - `aggregate(doc)`: Use accumulators
  - `explain()`: Describe the operations to perform

Examples

```
# Bulk insert some values into the collection uni (to be created)
var bulk = db.uni.initializeUnorderedBulkOp();
bulk.insert({"_id": "4711", "name": "Julian", "gender": "male", "major": "computer science", "birth": ISODate("2000-10-01")})
bulk.insert({"_id": "4712", "name": "Hans", "gender": "male", "major": "computer science", "birth": ISODate("2000-10-01")})
bulk.execute()
# BulkWriteResult({ "writeErrors" : [ ],    "writeConcernErrors" : [ ], "nInserted" : 2, "nUpserted" : 0,  "nMatched" : 0, "nModified" : 0, "nRemoved" :
       ↪ 0,"upserted" : [ ] })

# Create an index on the student's name
db.uni.createIndex( { "name": 1 } )

# Return the first 10 student names
db.uni.find( {}, {"name" : 1} ).limit(10)
#{ "_id" : "4711", "name" : "Julian" }
#{ "_id" : "4712", "name" : "Hans" }

# Return the student birth data where the name matches Hans
db.uni.find( { "name" : "Hans" }, {"birth" : 1} )
# { "_id" : "4712", "birth" : ISODate("2000-10-01T00:00:00Z") }

# Update the student, adding an address to all students with name Julian
db.uni.update ( {"name" : "Julian" }, {$set : { "address" : { "plz" : 4711, "city" : "Hamburg" } } }, {multi: true} )
# WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })

# Aggregate to count the number of male and female computer science students
# The match stage filters the documents first
# The _id field indicates the field to use for grouping, here gender
db.uni.aggregate( [ { $match: { "major": "computer science"} },
                    { $group: { "_id": "$gender", "count": { $sum: 1 } } }   ] )
# Returns: { "_id" : "male", "count" : 2 }

db.uni.drop() # remove collection
```

Python [17]

```python
import pymongo
from bson.objectid import ObjectId # internal object IDs

# Establish a connection
client = pymongo.MongoClient('localhost', 27017)
db = client.test # access test database

# print collections
db.collection_names(include_system_collections=False)
# ['uni']
uni = db.uni # access uni collection

print(uni.find_one({"name": "Julian"}))
# {'_id': '4711', 'name': 'Julian', 'gender': 'male', 'birth': datetime.datetime(2000, 10, 1, 0, 0),
    ↪ 'major': 'computer science'}

# Insert a student, we don't care about the id here
print(uni.insert_one({"name" : "Fritz"}).inserted_id)
# 58495ad0e91ebf67ae7f197d

# We can also use strings as the ID...
print(uni.insert_one({"_id": "Fritz", "name" : "Fritz"}).inserted_id)
```

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 highly complex architecture)
- Schema design can be tricky
- ZooKeeper manages service configuration and coordinates applications
- The document object model stores documents with subdocuments
  - Relations by embedding data as subdocument OR object reference
- MongoDB is a document object storage for JSON-like data
  - Query filtering (SQL where clause) via JSON documents
  - Scalable on a cluster via sharding of documents

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

44 https://docs.mongodb.com/manual/core/sharding-choose-a-shard-key

45 https://blog.cloudera.com/apache-hbase-i-o-hfile/

46 https://www.mongodb.com/blog/post/on-selecting-a-shard-key-for-mongodb