Introduction to Apache HBase(part 1)

medium.com/@lagrang09/introduction-to-apache-hbase-755b40cc0f30

This article is aimed to be “beginners` guide” to Apache HBase.

HBase is very mature product and has extensive documentation which can provide great volume of information about it. Nothing can replace official documentation, this is source of truth:)

But for people who see documentation first time, it very hard to get a quick overview of the system capabilities and understand is it suitable for his/her task. That’s why I wrote this post.

Let’s see what you can learn from it:

- data model: how HBase stores your data, what is a table in HBase, etc.
- how to access data in HBase at client side
- how you can store multiple versions of your data and how long this version can live
- ACID semantics and what write and read guarantees HBase can provide

Each section contains links to various HBase related resources which describes one or another feature in more details. All information based on HBase 2 which was released 30 Apr 2018.

What is HBase

Let’s start with a little history. Back in 2006 Google announce paper which describes it internal distributed storage called BigTable. It was designed to store petabytes of data on thousands of servers. A Bigtable is a sparse, distributed, persistent multidimensional sorted map. The map is indexed by a row key, column key, and a timestamp; each value in the map is an uninterpreted array of bytes.
HBase is NoSQL database based on Google BigTable architecture. HBase is really more a “Data Store” than “Database” because it lacks many of the features you find in an RDBMS, such as typed columns(all data is uninterpreted raw bytes for HBase), secondary indexes, triggers, and

advanced query languages, etc(but most of this features covered by Apache Phoenix which works on top of HBase).
However, HBase has advantage over classic RDBMS: linear scaling. An RDBMS can scale well, but only up to a point—specifically, the size of a single database server—and for the best performance requires specialized hardware and storage devices. HBase clusters expanded by adding RegionServers that are hosted on commodity class servers. If a cluster expands from 10 to 20 RegionServers, for example, it increases both in terms of storage and processing capacity.
Most notable HBase features are:

- Strongly consistent reads/writes
  HBase is not an "eventually consistent" DataStore. This makes it very suitable for tasks such as high-speed counter aggregation.
- Automatic sharding
  HBase tables are distributed on the cluster via table regions, and regions are automatically splited and distributed across servers as your data grows.
- Automatic failover:
  When HBase cluster detects that some node failed, all data handled by it moved to another node.
- Designed for write intensive load
  HBase apply writes by storing new value in in-memory structure and appending "write operation" to Write-Ahead-Log(to not loss changes). No on-disk in-place data modification required.
- Effective for storing and accessing time series and key-value data
  Because HBase store data in sorted order, you can append timestamps to key and scan your data in required time range.

Data model

High level representation of data stored inside HBase is similar to relational databases. It has notion of tables, namespaces(e.g. databases), columns.
Data stored in tables which consist of rows with columns which store actual values. Tables can be logically grouped into namespaces which can

be thought as databases. Namespaces primarily used to set resource quotas or common security settings to group of tables.
Although, all said earlier is very similar to relational DBMS(RDBMS), but it has little to do with RDBMS. Difference begins from how HBase stores and consequently accesses data. Unlike RDBMS, tables in HBase has no schema and contain rows which identified by unique key(like a primary key in RDBMS). Each row can contain arbitrary count of columns with some binary data in it(HBase doesn't interpret data inside column and treat it as raw byte array). Each column has unique identifier represented by *column family* and *column qualifier*.

Column family(CF) represents a group of columns which usually accessed together(in same request) or/and have common storage properties such as compression, data encoding, caching, etc. Physically CF data stored in files which contain only columns belongs to this CF, that is why accessing columns in same CF is very efficient(HBase scan only subset of all table files to find required columns). Table can contain multiple column families, but in practice 2–3 CF is reasonable value(more CF can affect performance, see docs).

HBase can store multiple **versions** of column(optionally with TTL). Each version identified by *timestamp*. Timestamp can be set in write request to HBase. By default HBase sets timestamp at server to the current value of epoch time in milliseconds.
Versions count stored by HBase can be set per column family.
Concept of versions looks very simple at first, but can be not so obvious in some corner cases(more in following section).

Combination of row key, column family, column qualifier and version is called a *cell*.
Cell contains actual data stored as raw binary(byte array). Cell content is not interpreted by HBase, except for one special case of atomic counters(see docs for more info).

Let's see example of how HBase row look like.
Suppose that table contains statistics for visit tracking of the web pages. Table consist of "info" and "stats" column families with row key which

contains ID of user which visit web page.

row key(user ID)	CF:'info', qualifier: 'user_nickname'	CF: 'stats', qualifier: 'https://web1.com'	CF: 'stats', qualifier: 'https://web2.com'	Timestamp
651	lee	205	600	1543139553000
651	lee	409	1100	1543139953000
442122	dug_1995	10	152	1543139597000
442122	dug_1995	170	302	1543149997000

More info about HBase data model can be found here.

Manipulating table data

We know that data inside HBase can organized into tables. And now we can discuss how to read, write and remove data from tables.

First set of operations used to insert, update and remove rows and columns.

- **Put**
  Put either adds new rows to a table (if the key is new) or can update existing rows (if the key already exists). Put can atomically change multiple cells in one particular row.
- **Increment**
  Special operation type which provide a way to atomically increment 64-bit value contained in some cell. Can be used to implement counter in distributed aggregation services.
- **Delete**
  Delete one or more cells associated with particular row or remove entire row. HBase does not remove data in place, and so deletes are handled by creating new markers called tombstones. These tombstones, along with the dead values, are cleaned up on major compactions.
- **CAS**
  This operation set similar to CPU compare-and-set instructions and include following operations: check-and-put or check-and-delete(recently both operations unified into one check-and-mutate). More info can be found in client docs.

Operations related to data access:

- **Get**
  Basic “read” operation. Get can return entire row or some particular row columns.
  Get operation support filtering on a server-side, for instance, by column value, by column name, by version, etc. See more in docs.
- **Scan**
  More advanced read operation. It provides a way to scan range of rows defined by start and end keys. Scan can traverse key range starting from ‘start’ key towards to ‘end’ key as well as from ‘end’ key towards to ‘start’ key. Scan operation support set of server-side filters same as get operation(because get based on scan operation).

Non-CAS mutation operation can be batched to update multiple rows in one request. Batch can mix read and write operations and result of each operation can be accessed separately. Batch operation doesn’t guarantee atomicity(see ACID section), e.g. some operations can end up with success but other fail.

Let’s see how data operation works by example. Suppose we have schema same as in “Data Model” section example. In this section we will use official Java HBase API, but this example will not be exhaustive enough to learn Java API. For more information see javadocs, HBase client docs.

Let’s create table “page_visits”. We have a few options to do this:

HBase shell

```
create ‘page_visits’, {NAME => ‘info’}, {NAME => ‘stats’}
```

HAdmin

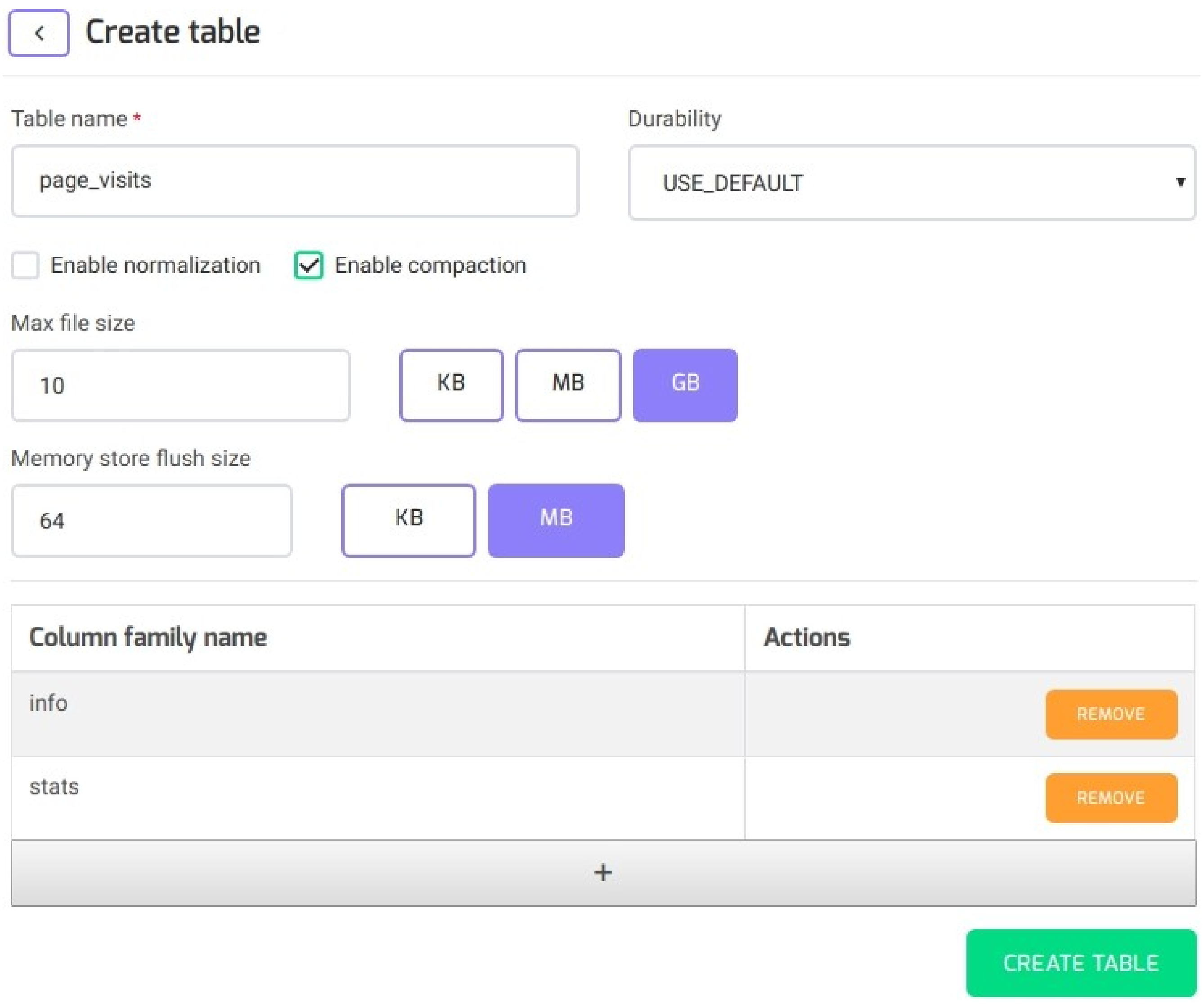

Create table

Table name * page_visits

Durability USE_DEFAULT

[ ] Enable normalization [✓] Enable compaction

Max file size 10 KB MB [✓ GB]

Memory store flush size 64 KB [✓ MB]

Column family name	Actions
info	REMOVE
stats	REMOVE

And now we ready to write some code which will show how to manipulate data in HBase. Next example is a little bit boring, but show usage of official client Java API. It contains a bunch of comments which should clarify each step.

Data versioning

As noted above, each column in HBase can have multiple version, identified by timestamp. Particular version of column can be referenced by using of *cell.* Cell is a tuple [row key, column family, column qualifier, timestamp].

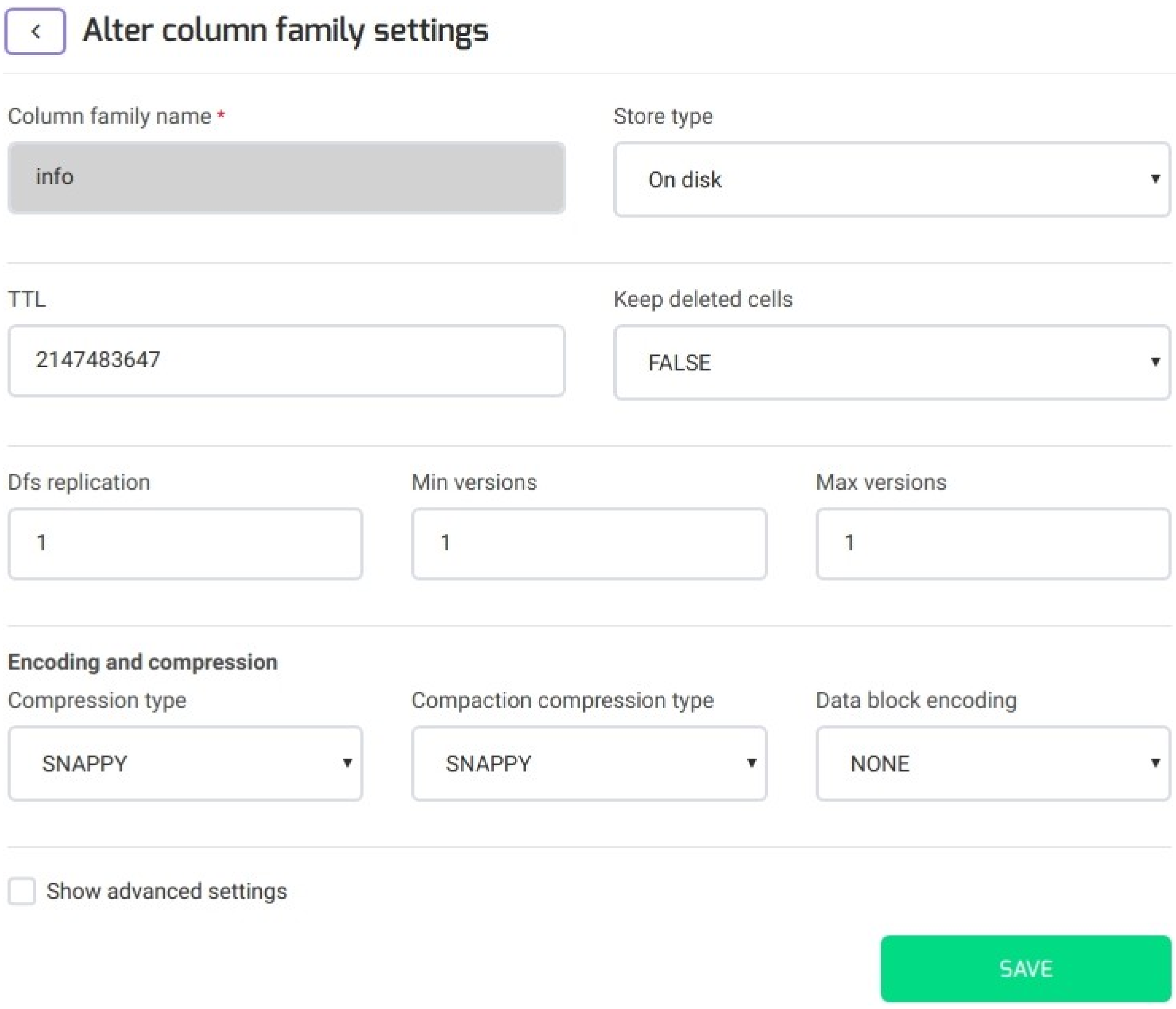
Recent versions of HBase store 1 version of data by default. You can set max versions count per column family by using:

HBase shell:

```
alter ‘your_table_name′, NAME => ‘multi_version_cf′, VERSIONS => 3
```

Java API(see Admin interface)

HAdmin



Alter column family settings

Column family name *	info
Store type	On disk
TTL	2147483647
Keep deleted cells	FALSE
Dfs replication	1
Min versions	1
Max versions	1

Encoding and compression

Compression type	SNAPPY
Compaction compression type	SNAPPY
Data block encoding	NONE

☐ Show advanced settings

Timestamp can be set in write request to HBase, by default HBase set timestamp at server to current value of epoch time in milliseconds. Timestamp can be any positive 64-bit value, not strictly increasing in time.
Get API returns only latest cell version(with greatest timestamp), not the last written cell. Get request allows to configure timestamp range per column family. Such get request returns cells which fall into this range.

The most interesting part of versioning is how it interacts with deletion. HBase API allow to remove cells:

- with a timestamp less than or equal to some value, or
- with specific version

Because remove operations don’t actually remove data in-place(see tombstone marker in Data Model section), this can lead to non-obvious behavior. For instance, table contains 3 cells c1, c2, c3 with increasing timestamps and table configured to stores maximum 2 versions. In this case, get request returns c2 and c3 cells(HBase returns two newest versions).

After that, we execute delete request for c2, c3 cells and call same get request as before. Because c1 cell still stored on disk and more newest versions removed, in result we see that c1 again.

But we have one more possible result. Because HBase not modify data in place and all changes(insert, delete) is a new record on disk, it needs to be periodically clean up of on-disk data. This is called compaction. Let’s see how compaction can affect query results.

Consider following situation:

- compaction will remove c1, because table configured to stores only 2 versions of data(and we have it, c2 and c3)
- after that you delete c2 and c3
- at this point, your query returns empty result

Since HBase 2.0.0 this behaviour can be changed by explicitly setting new version behavior. This will prevent such unpredictable behaviour, but can affect performance. More info about versions in HBase can be found in docs.

Time-To-Live

HBase can handle stale data by removing it after some time. This feature is called Time-To-Live(TTL).
TTL can be configured on column family or on particular cell. There are notable differences in handling TTL for cell/CF:

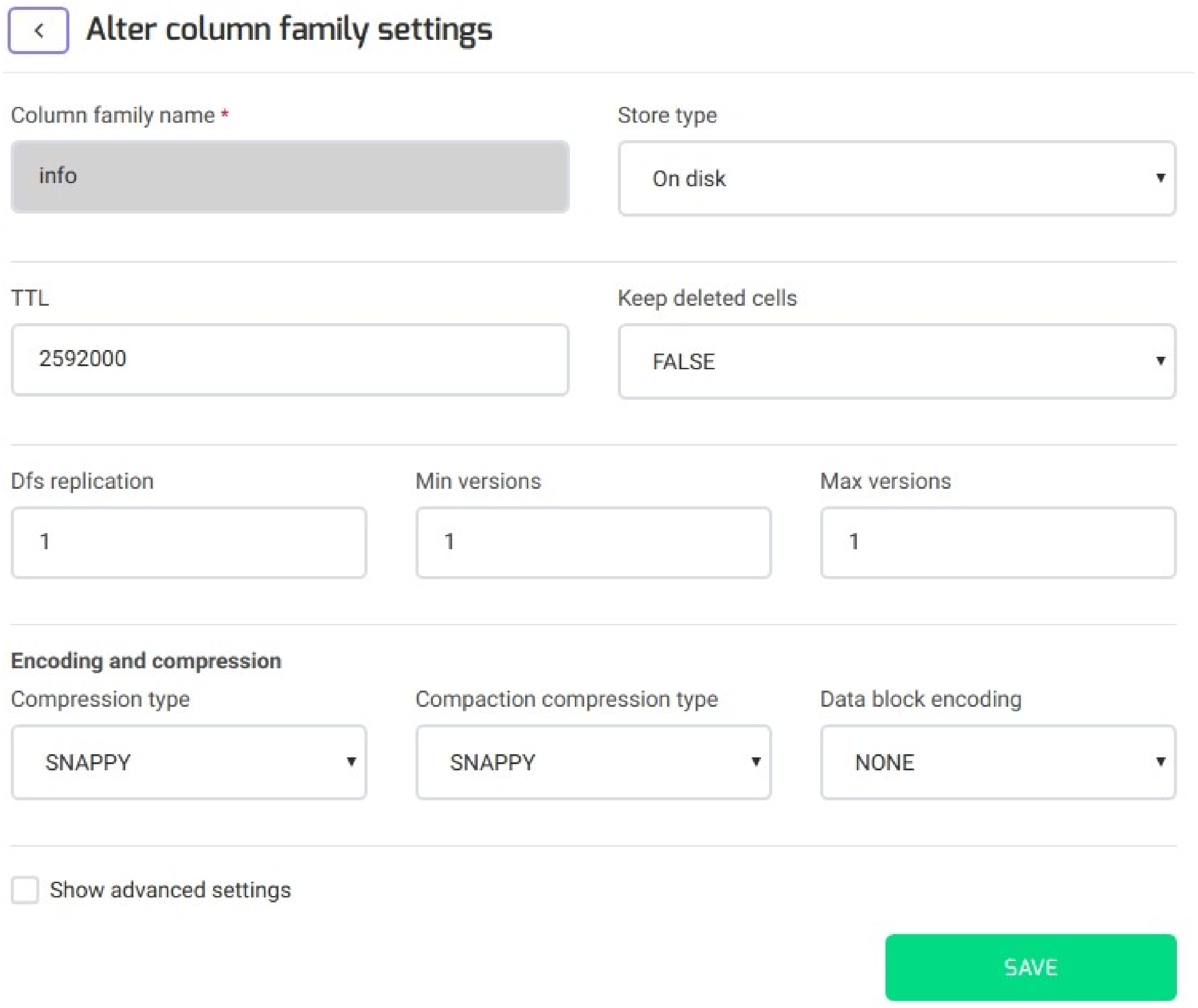
- column family TTLs are expressed in units of seconds
- cell TTLs are expressed in units of milliseconds
- cell TTLs can not exceed a column family level TTL setting

You can set TTL value from:

HBase shell

```
create ‘test_table’, {NAME => ‘cf1’, VERSIONS => 1, TTL => 2592000}
```

HAdmin



Alter column family settings

Column family name *	info
Store type	On disk
TTL	2592000
Keep deleted cells	FALSE
Dfs replication	1
Min versions	1
Max versions	1

Encoding and compression

Compression type	SNAPPY
Compaction compression type	SNAPPY
Data block encoding	NONE

More info about TTL see in docs.

ACID

In this section we discuss ACID semantics in common and how HBase conforms to it. Who is familiar with ACID terminology can freely skip next paragraph and go directly to HBase ACID part.
ACID is a set of guarantees provided by database during transaction processing. This is acronym for “Atomicity, Consistency, Isolation and Durability”. Let’s discuss each component of ACID:

- **Atomicity** property guarantee that all changes inside transaction will all successfully applied or none in case of failure.
- **Consistency** is most ambiguous part of ACID and this property is mostly application level which cannot be validated by database. It means that some data is consistent with other according application requirements. In reality, this property is not a part of ACID.
- **Isolation** mostly prevent concurrency problems which arise when several clients simultaneously changes same data. For example, isolation can guarantee that transaction will not see changes made by other transactions until it finished or data accessed(updated) by one transaction cannot be accessed by another concurrent transaction.
- **Durability** means that data written by successful transaction will be persisted to stable storage and not be lost in case of server/database failures. Non-replicated database doesn't covers failures related to liveness of stable storage. To protect yourself from this type of failures, use replication which will write data to replicas before transaction commit.
  ACID is not a strongly defined and most databases which claims itself as ACID compliant provide different guarantees.

HBase is not an ACID compliant database, however, it does guarantee some specific properties. Next section contains only brief review of HBase ACID properties with few examples to ease understanding. Full version can be found in documentation).

Atomicity:

- All mutations are atomic within a same row(even for multiple column families). Any put/increment/delete operation for same row will either wholly succeed or fail. No distributed transaction involved here: all data inside row handled by one HBase server in time.
- The checkAndPut API happens atomically like the typical compareAndSet (CAS) operation found in many hardware architectures.

Consistency:

- A scan is not a consistent view of a table(scans do not exhibit snapshot isolation). A scan will always reflect a view of the data at least as new as the beginning of the scan, e.g. you can see changes made by transactions started after beginning of a scan.
- All rows returned via read API will consist of a complete row that existed at some point in the table's history.
  For instance, few clients concurrently execute put requests p1,…, p5 for same row *r1* in some random order.
  Suppose, all operation p1-p5 change same set of columns. Read API can see *r1* only in one of 5 possible states p1-p5, not in some state which interleave changes from different put requests.
  Let's dive into detailed explanation by code example.

Visibility:

> When a client receives a "success" response for any mutation, that mutation is immediately visible to both that client and any other client which was notified by client from which originates "success" mutation. In other words, you read your own writes and others can read your writes if they know when you will finish your mutations.

Durability:

- All visible data is durable data(always on disk).
- Any operation that returns a "success" code is durable.

Conclusion

This is the end of first part of this blog post. In this part we review basis of Apache HBase data model, explain ACID semantics and see how to access data using Java API.

In next part we consider following topics:

- HBase cluster components and how they interact with each other
- how HBase stores data on disk
- replication
- few real world examples