

# HBase Notes

---



Giovanni  
Bonelli



Hbase è un DB NoSQL costruito su HDFS, è colonna. Basato su Google BigTable paper, dati vengono memorizzati in documenti o batch process

Perché? Salvare granigli di TBLPS, capace di gestire tante richieste, salvare dati con struttura o con schema variabili, random read and write

Perché NO? Se non è un DB, si leggono direttamente determinati file, scrivvi una volta sola o sempre in append, non ci sono modelli, l'accesso definitivo

## HBase Tables

I dati vengono salvati in tabella sotto forma di ROW, ognuna può avere colonne diverse senza uno schema preciso.

Le tabelle sono raggruppate in NAMESPACE, usati per raggruppare tabella per stante o applicazione.

Ogni riga è identificata da una ROW KEY, le row sono ordinate lessicograficamente tramite la key binariamente

Anche le colonne sono raggruppate in COLUMN FAMILIES, che impostano anche la disposizione fisica dei dati. Per questo infatti devono essere definiti in anticipo e non sono facilmente modificabili. Ogni riga ha le stesse CF, anche se sono vuote.

Le CF sono identificate dalla column QUALIFIER, una colonna specificata all'inizio per ogni CF.

Una combinazione fra ROW KEY-CF-CQ identifica una CELLA. Hbase consente più valori per ogni cella, con il timestamp

Hbase Cell Versions dunque permette di inserire valori tenui associati ad un TIMESTAMP, questo timestamp viene usato anche per il TIME-TO-LIVE

## HBase Architecture

I dati sono quindi organizzati in tabella, suddivise in REGIONS, che sono intervalli continui di ROWKEYS. Questi vengono hostati da nodi del cluster responsabili di esse detti RegionServers. Esse sono gestite dal MASTER node, i dati si collegano direttamente

alle region servers tramite RPC layer. Tutto sono coordinati da ZOOKEEPER. Vi sono altri servizi quali REST (per le API) e THrift Server.

## WRITE OPERATION

Le operazioni vengono eseguite direttamente su RegionServer. Quella di riferimento viene identificata da Zookeeper che

interroga META table, la quale contiene il mapping fra region e RS. La RegionServer salva i dati in Hfile,

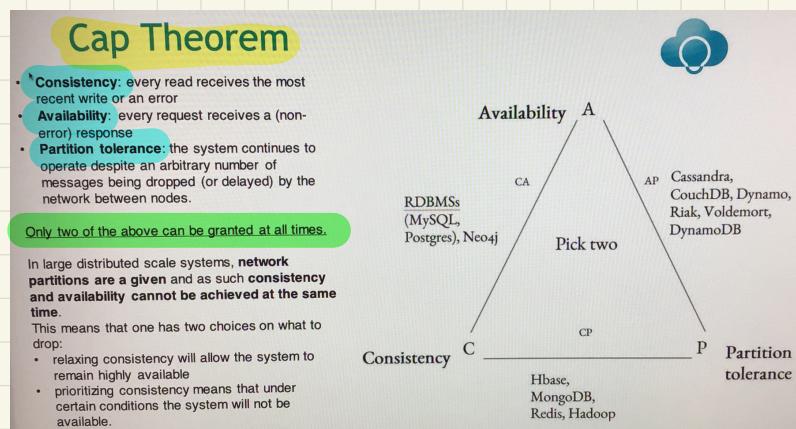
soltanto in seguito su HDFS. I cambiamenti ai dati vengono memorizzati nel WLOG su HDFS, che agisce

come WAL (write ahead log) per recuperare i dati in caso di errore. Infine una cella viene scritta nel

MEMSTORES, dei buffer di memoria da periodicamente creare nuovi Hfiles.

## READ OPERATIONS

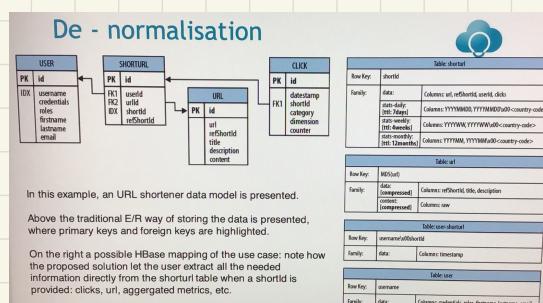
Le richieste vengono inviate a Zookeeper che indirizza la RegionServer con il META.Table che indica la regione di riferimento da considerare (la rk di interesse). Nelle fasi di lettura i Memstore e gli Hfiles vengono uniti on-the-fly per avere un qualche vantaggio della region. È invece le Cache per velocizzare queste operazioni. Per filtrare i file coinvolti in ogni regione si usano i filters Bloom. In caso di failure di un region server, i dati contenuti nel Memstore vengono persi, ma il contenuto può essere recuperato dal Wlog.



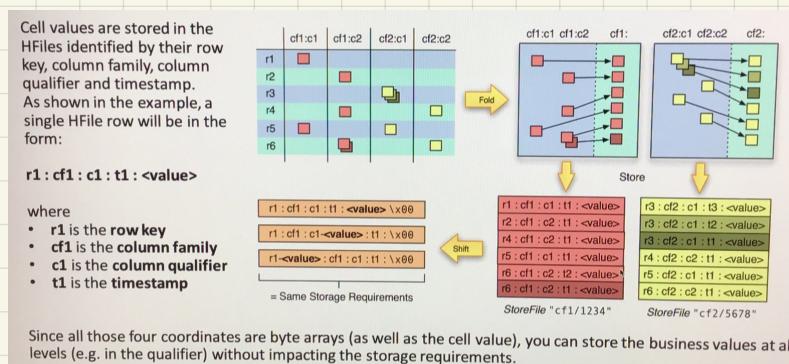
## DENORMALISATION

Spesso bisogna ripensare come i dati vengono condivisi in sistemi NoSQL per utilizzarli in modo ottimale.

Si usa quindi DDI (Denormalization, Duplication, Intelligent keys)



## STORAGE LAYER REPRESENTATION



## DIFFERENZE CON RDBMS E VANTAGGI

- I file RDBMS si basano su relazioni e normalizzazioni, HBase su denormalizzazione e pattern di accesso.
- HBase fornisce un accesso key-based ad una cella o ad un insieme sequenziale di celle.
- Le operazioni sono fatte su una riga.
- È colonnaare, le colonne vengono raggruppate in famiglie e all'interno di esse sono memorizzate sotto forma di row.

## ACCEDERE HBASE

- Si può accedere tramite JAVA API, REST (HTTP access), THrift Gateway (permette accessi da ling. varie)
- Vi sono interface SQL non Native: Phoenix, Impala, Presto, Hive

## HBase Client API



### List of basic operations

Type	Kind	Description
Get	Query	Retrieve previously stored data from a single row; can get entire row or only some cf/cells.
Scan	Query	Iterate over all or specific rows and return their data; some filters can be used.
Put	Mutation	Create or update one or more cells in a single row.
Delete	Mutation	Remove a specific cell, column, row, etc.
Increment	Mutation	Treat a column as a counter and increment its value.
Append	Mutation	Attach the given data to one or more columns in a single row.

## Delete



- Deletes work by creating **tombstone markers**. Tombstone markers are just labels with a timestamp assigned to cell versions.
- Only in the **major compaction**, the tombstones are processed to actually remove the dead values.
- When HBase does a major compaction, the tombstones are processed to actually remove the deleted cell values, together with the tombstones themselves. If the **version** you specified when deleting a row is larger than the version of any value in the row, then you can consider the complete row to be deleted.

## INTERNAL TABLE OPERATIONS

**MINOR COMPACTATION** = compatta solo alcuni file senza riunire le celle eliminate. Piccoli Hfiles vengono uniti.

**MAJOR COMPACTATION** = Tutti i file sono elati come compatibili e le celle vengono riunite. Tutti gli Hfiles di una regione vengono riscritti in un unico Nfile. Molto cariosa e pesante.

**SPLITS** = Quando una regione supera la memoria limite o è troppo ridicile viene **splitata** in 2.

**BALANCING** = Il Master si occupa di fare in modo che ogni RegionServer gestisce un numero bilanciato di Regioni.

# HBase Processes

Master → table DDL operations

Region Servers → operations READ/WRITE

Region Server con META.table → modifica le regioni con mappa region → regionserver

Sistemi esterni → Zookeeper → coordinate

Un HBase Coprocessor è un piccolo custom script dell'host.

ENDPOINT COPROCESSOR → Simile ad una stored procedure. Deve essere invocata tramite PROTOBUF specifico.

OBSERVER COPROCESSOR → Simile ai trigger. Si esegue ad una azione, disponibile <sup>4 de</sup> per ogni action.

## Observer Coprocessor – Possible Triggers

- Region Server Observer
  - preStopRegionServer
  - preExecuteProcedures
  - postExecuteProcedures
  - preClearCompactionQueues
  - postClearCompactionQueues
- Region Observer
  - preGetOp
  - postGetOp
  - prePut
  - postPut
- WAL Observer
  - preWALRoll
  - preWALWrite
  - postWALRoll
  - postWALWrite

- Master Observer
  - Runs in HBase master
  - Create, Delete, Modify table
  - Clone, Restore, Delete Snapshots
  - Region splits and many more

## Filters

- Similar to Observer Coprocessors for get/scan
- Can filter on any part of a row
- Many built-in filters available
- Can deploy custom filters
- Support for push-down predicates
- Different API
- Came earlier and much more limited

Perché usarli?

Per controllare il comportamento di RegionServer o di operazioni, filtri o aggregazioni, ridurre la pressione dato che per analisi di dati non complessi (Phoenix)

## DYNAMIC VS STATIC

Description	Static Loading	Dynamic Loading
Changes to hbase-site.xml	Yes	No
Restart region servers	Yes	No
Jar location	Local filesystem	HDFS
Coprocessor availability	Global	Per table
Loaded via hbase shell	No	Yes
Loaded via java API	No	Yes
Read permissions	HBase	HBase
Management complexity	High	Low

## Deployment Guidelines

- Enable user coprocessors
  - Set `hbase.coprocessor.user.enabled` to true
  - Phoenix coprocessors are treated as user coprocessor
- Set `hbase.coprocessor.enabled` to true
  - Keeps system coprocessors enabled (AccessController)
- Enable coprocessor white listing (HBASE-16700)
  - Set `hbase.coprocessor.region.whitelist.paths`
  - Specify each directory individually
  - Wildcards won't include subdirectories (documentation says otherwise)
  - Entire filesystem, i.e., `hdfs://Test-Laptop`, won't work (documentation says otherwise)

## RECAP

- Coprocessors are necessary
  - Phoenix
  - HBase security
- User coprocessors are dangerous
  - Write defensive code
  - Be careful with deployment
- Make use of HBASE-16700
- Cleanup can be messy
  - HBASE-14190 – Assign system tables ahead of user region assignment

## Schema DESIGN

This article covered the basics of HBase schema design. I started with a description of the data model and went on to discuss some of the factors to think about while designing HBase tables. There is much more to explore and learn in HBase table design which can be built on top of these fundamentals. The key takeaways from this article are:

- ◆ Row keys are the single most important aspect of an HBase table design and determine how your application will interact with the HBase tables. They also affect the performance you can extract out of HBase.
- ◆ HBase tables are flexible, and you can store anything in the form of `byte[]`.
- ◆ Store everything with similar access patterns in the same column family.
- ◆ Indexing is only done for the `Keys`. Use this to your advantage.
- ◆ Tall tables can potentially allow you faster and simpler operations, but you trade off atomicity. Wide tables, where each row has lots of columns, allow for atomicity at the row level.
- ◆ Think how you can accomplish your access patterns in single API calls rather than multiple API calls. HBase does not have cross-row transactions, and you want to avoid building that logic in your client code.
- ◆ Hashing allows for fixed length keys and better distribution but takes away the ordering implied by using strings as keys.
- ◆ Column qualifiers can be used to store data just like the cells themselves.
- ◆ The length of the column qualifiers impact the storage footprint since you can put data in them. Length also affects the disk and network I/O cost when the data is accessed. Be concise.
- ◆ The length of the column family name impacts the size of data sent over the wire to the client (in KeyValue objects). Be concise.