Apache - Cassandra

Advanced Topics in NoSQL databases

nicolas.travers (at) devinci.fr

Plan

What is Cassandra?

Cassandra[1], designed initially in 2007 by *Facebook*, is now an *Apache* project since 2008. The company *Datastax* distributes it and provide several services around.

- Conceptually, it was inspired by *BigTable* from Google (column-oriented store),
- Since Cassandra v2.0, it is based on kind of *DynamoDB* (Amazon), a hash-based technic (DHT) which leads to a key-value store paradigm,
- It evolved to a *Wide-column* oriented (extended relational) $\Rightarrow$ *Document-Oriented store* (N1NF =*Non First Normal Form*),
- Pros:
  - Scalability and Fault tolerance,
  - One of the rare typed NoSQL database,
  - A simple query language inspired from SQL: `CQL`

[1] `https://github.com/apache/cassandra/tree/trunk/lib`

Model = relational + complex data types

- *Database*: Keyspace
- *Tables*: Table or Column Family[2]
- *Rows*: Row (either simple and complex values)

First sight: denormalize your schema.

Second sight: a row is a document (with nesting)

[2] ⚠This does not says that is a column-oriented store! Cassandra is document-oriented (question of storage)

Key-value Pairs and Documents

A piece of data is composed of:

- **Key**: the identifier
- **Row**: simple or a complex value but typed

Rows are typed by a schema, even for nested data.
Schema: every inserts must validate the schema

The most important thing to remind for efficiency

A row is identified by a **key**

Create your Own Cassandra Database

Create a **keyspace**:

```
CREATE KEYSPACE IF NOT EXISTS Company
    WITH REPLICATION={'class':'SimpleStrategy',
                    'replication_factor':3};
```

Create a **Column Family** / table:

```
CREATE TABLE Flight (
    idFlight INT, dateF DATE, distance INT, duration FLOAT,
    fromIATA CHAR(3), toIATA CHAR(3), pilot INT,
    copilot INT, officer INT, purser INT,
    purser2 INT,
    primary key (idFlight)
);

CREATE INDEX flight_pilot ON Flight (pilot);
```

Insert

Like in SQL

```
INSERT INTO Flight (idFlight, dateF, distance,  duration,
        fromIATA, toIATA, pilot, copilot,
        officer, purser, purser2)
    VALUES (1,'2018-10-15',344,1.3,'CDG','LCY',1,2,3,4,5);
```

Inserting JSON documents[3]:

```
INSERT INTO Flight JSON '{
    "idFlight": 1, "dateF": "2016-10-15", "distance":344,
    "duration": 1.3, "fromIATA": "CDG", "toIATA": "LCY",
    "pilot": 1, "copilot": 2, "officir": 3,
    "purser": 4, "purser2": 5}';
```

[3]to use in order to import your Dataset.

Querying language

CQL: *Cassandra Query Language*
Simplified version of SQL

```
SELECT * FROM Flight WHERE idFlight = 1;
```

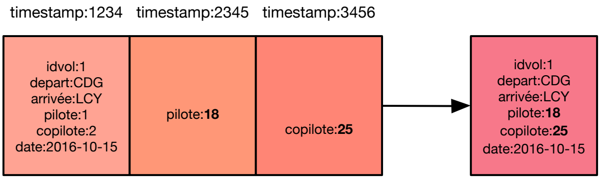
Result:

```
idFlight |      dateF | distance | duration | fromIATA | toIATA | pilot | copilo
---------------------------------------------------------------------------------
       1 | 2018-10-15|      344 |      1.3 |      CDG |    LCY |     1 |
```

Temporal Data

Each value is linked to a TIMESTAMP.
For every update a timestamp is generated and can by specified explicitly:

```
UPDATE Flight USING TIMESTAMP 2345
       SET pilot=18 WHERE idFlight = 1;
```



Temporal Data

Extract update time[4]:

```
SELECT writetime(fromIATA), writetime(pilot), writetime(copilot)
FROM Flight WHERE idFlight=1;
```

Output

```
writetime(fromIATA) | writetime(pilot) | writetime(copilot)
------------------------------------------------------------
               1234 |             2345 |               3456
```

Can be used for timely deletion:

```
DELETE pilot USING TIMESTAMP 1234 FROM Flight WHERE idFlight=1;
```

[4] Cannot be queried in the WHERE clause.

Temporary Data

Ephemeral value updates: TTL[5]
Kind of trigger automaticaly generated to make the data disappear

```
UPDATE Flight USING TTL 3600
    SET pilot=18 WHERE idFlight = 1;
```

[5] TTL: Time To Leave

Data Types for Nesting

An attribute can be typed for nesting (XXX):

1. **SET**: set of values (unordered)
2. **LIST**: list of values (ordered)
3. **MAP**: set of key/value pairs (nested document)
4. **TYPE**: typed nested row (whole tuple)

Example of nested data (hostesses):

```
CREATE TABLE Flight (
    idFlight INT, dateF DATE, distance INT,  duration FLOAT,
    fromIATA CHAR(3), toIATA CHAR(3), pilot INT,
    copilot INT, officer INT, purser1 INT, purser2 INT,
    hostesses XXX<YYY>,
    primary key (idFlight)
);
```

hostesses SET<int>

- *Insert*:

```
INSERT INTO Flight (..., hostesses) VALUES (..., {6, 7, 8});
```

- *Updates*:

```
UPDATE Flight SET hostesses=hostesses + {9} WHERE idFlight=1;
UPDATE Flight SET hostesses=hostesses - {8} WHERE idFlight=1;
UPDATE Flight SET hostesses={10} WHERE idFlight=1;
DELETE hostesses FROM Flight WHERE idFlight=1;
```

- *Getting the set*:

```
SELECT idFlight, hostesses FROM Flight WHERE idFlight = 1;
```

```
idFlight |  hostesses
------------------
       1 | {6, 7, 8}
```

hostesses LIST<int>

- *Insert*:

```
INSERT INTO Flight (..., hostesses) VALUES (..., [6, 7, 8]);
```

- *Update*:

```
UPDATE Flight SET hostesses=hostesses + [9] WHERE idFlight=1;
UPDATE Flight SET hostesses[1]= 8 WHERE idFlight=1;
UPDATE Flight SET hostesses= [10] WHERE idFlight=1;
DELETE hostesses[0] FROM Flight WHERE idFlight = 1;
```

- *Getting the list*:

```
SELECT idFlight, hostesses FROM Flight WHERE idFlight = 1;
```

```
idFlight | hostesses
-------------------
       1 | [6, 7, 8]
```

hostesses **MAP**<text, int>

- *Insert*:

```
INSERT INTO Flight (..., hostesses) VALUES (...,
                {"h1": 6, "h2": 7,"h3": 8});
```

- *Update*:

```
UPDATE Flight SET hostesses = hostesses + {"h4": 9}
                WHERE idFlight=1;
UPDATE Flight SET hostesses["h1"] = 10 WHERE idFlight=1;
UPDATE Flight SET hostesses = {"h1": 9} WHERE idFlight=1;
DELETE hostesses["h2"] FROM Flight WHERE idFlight=1;
```

- *Getting the map*:

```
SELECT idFlight, hostesses FROM Flight WHERE idFlight = 1;
```

```
idFlight |                  hostesses
--------------------------------------
       1 | {"h1": 6, "h2": 7, "h3": 8}
```

hostess frozen<hostessType> 1/2

- First, we need to create a data type for nesting:

```
CREATE TYPE hostessType (ID int, lastname text, firstname text);
```

- Then, in the `CREATE TABLE` a "frozen" needs to be added to the type:

```
CREATE TABLE Flight (
    idFlight INT, dateF DATE, distance INT,  duration FLOAT,
    fromIATA CHAR(3), toIATA CHAR(3), pilot INT,
    copilot INT, officer INT, purser1 INT, purser2 INT,
    hostesses frozen<hostessType>,
    primary key (idFlight)
);
```

hostess **frozen<hostessType>** 2/2

- *Insert*:

```
INSERT INTO Flight (..., hostesses) VALUES (..., {"ID": 6,
        "lastname": "Walthéry", "firstname": "Natacha"});
```

- *Update*:

```
UPDATE Flight SET hostess["lastname"] = "Walter"
        WHERE idFlight = 1;
DELETE hostess FROM Flight WHERE idFlight = 1;
```

- *Nested attributes can be projected*:

```
SELECT idFlight, hostess.lastname, hostess.firstname
FROM Flight WHERE idFlight=1;
```

```
idFlight | hostess.lastname | hostess.firstname
--------------------------------------
    1 |        Walthéry |            Natacha
```

Cassandra ⇒ No Joins ⇒ Merge Data

Since data are distributed, no joins is possible between servers. It is recommended to merge data from tables in a single data structure.

- Conception based on most frequent queries[6]
- Focus on the biggest dimension in number of rows (better distribution)
- ⚠Produce redundancy

Examples of queries:

- For a given flight, give the list of hostesses
- For a given hostess, give the her flights

[6] Datastax recommend to produce for each top query a new table... Can be dangerous!

How to properly nest data?

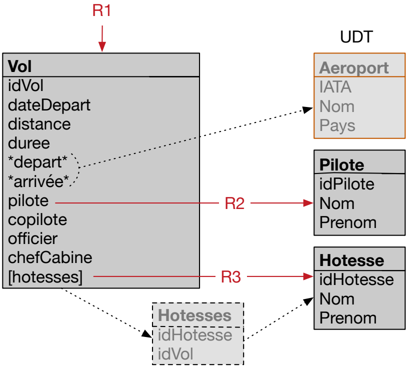
How to choose the nesting?

1. Chebotko Diagram[7]: *Query-Driven methodology*
   - Define the access pattern to the tables
   - Nest in the way that queries are applied
   - ⚠Query choice
2. Focus on the biggest dimension
   - Less updates
   - Better distribution
   - Best schema compatibility
   - ⚠Complex data model

[7] [Chebotko et al. 2015] https://pdfs.semanticscholar.org/22c6/740341ef13d3c5ee52044a4fbaad911f7322.pdf

Chebotko Diagram Example

- List []
- Set {}
- Map <>
- UDT **



1. Introduction
2. Data Model
3. CQL: Cassandra Query Language
   - Simple Queries
   - UDA ⇒ Map/Reduce
   - Sharding & Indexing
4. Scalability & Fault Tolerance

CQL 3.3

- **SELECT ...**
  - Attributes
  - For primary key and indexed attributes: DISTINCT, COUNT(*)
- **FROM**
  - A single table
- (**WHERE** ...)?
  - Detailed in next slide
- (**ORDER BY** ...)?
  - Only on primary key (ASC/DESC)
- (**LIMIT** ...)?
  - Only on primary key (ASC/DESC)
- (**ALLOW FILTERING**)?
  - Detailed in next slide

WHERE Clause

- **Primary Key = value**: Most efficient query
- **token(Primary Key)** = **value**: hashing function, gives the rows corresponding to this hashed values (not really useful)
- **attribute = value** + **INDEX**: Can be efficient (rely on statistics)
- **attribute = value** + **ALLOW FILTERING**: Totally inefficient ⇒ broadcast on all servers
- **Queries on nested data types**:
  - SET: *CONTAINS*
  - LIST: *CONTAINS*
  - MAP: *CONTAINS* / *CONTAINS KEY*
  - TYPE: att = whole nested value (BLOB)

  ⇒ *SET* & *MAP* are better to use

And what about aggregates?

`GROUP BY + COUNT/MIN/MAX/SUM/AVG`[8]

- Use Map/Reduce functions to aggregate data
  - Two phases program: filtering + clustering
  - *Map*: takes a row, produces a `key/value` in output
  - *Reduce*: takes a `key` and the list of corresponding values (from the Map phase).
- With *Cassandra*:
  - Java program: **User-Defined Aggregate Function** (`UDA`)

`https://docs.datastax.com/en/cql/3.3/cql/cql_using/useCreateUDA.html`

[8] Hard queries for the report

User-Defined Aggregate Function 1/2

Map

```
CREATE OR REPLACE FUNCTION avgState (state tuple<int,bigint>, val int)
CALLED ON NULL INPUT RETURNS tuple<int,bigint> LANGUAGE java
AS 'if (val !=null) { state.setInt(0, state.getInt(0)+1);
        state.setLong(1, state.getLong(1)+val.intValue()); }
return state;';
```

Reduce

```
CREATE OR REPLACE FUNCTION avgFinal ( state tuple<int,bigint> )
CALLED ON NULL INPUT RETURNS double LANGUAGE java
AS 'double r = 0;
        if (state.getInt(0) == 0) return null;
        r = state.getLong(1);
        r/= state.getInt(0);
        return Double.valueOf(r);';
```

User-Defined Aggregate Function 2/2

UDA

```
CREATE AGGREGATE IF NOT EXISTS average ( int )
SFUNC avgState STYPE tuple<int,bigint>
FINALFUNC avgFinal INITCOND (0,0);
```

Query

```
SELECT average (distance) FROM Flight;
```

Sharding on Primary Key ⇒ **Partitionning Key**

- By default: Partitionning is the Primary Key

```
PRIMARY KEY (idFlight);
```

- Primary keys can be composite

```
PRIMARY KEY (fromIATA, idFlight);
```

⚠Must query with both values

- Rows can be clustered (grouped according to a part of the Primary Key ⇒ Partitionning Key

```
PRIMARY KEY ( (fromIATA), idFlight);
```

  - Flights' rows from a given IATA are placed on a single server. All queries on "`fromIATA`" are optimized,
  - Locally, all data are ordered according to "`idFlight`"
    ⚠Sorted files are called `sstables`

Indexing

Some secondary indexes can be created

```
CREATE INDEX idflights ON Flight (idFlight);
```

- On MAP nested data types[9]:
  - By default find values in the nested document. Query:

    ```
    WHERE hostesses CONTAINS 8
    ```

  - Searching for existence of keys:

    ```
    CREATE INDEX existingHostesses ON Flight (keys(hostesses));
    ```

    Query:

    ```
    WHERE hostesses CONTAINS KEY "h1"
    ```

- Only one index per attribute

[9] Complex queries for the report

1. Introduction
2. Data Model
3. CQL: Cassandra Query Language
4. Scalability & Fault Tolerance
   - Distribution & Replication
   - Consistency

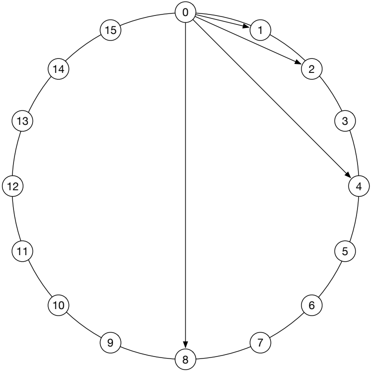
Distribution & Replication

DHT: Distributed Hash Table

- Distribution:
  - A single ring of 2^{64} nodes,
  - Routing by jumps (hash table).
- Replication:
  - Fault Tolerance,
  - A data is duplicated 3 times (locally and 2 previous nodes),
  - *Replication Factor* (3 by default),
  - Replicates can answer to queries $\Rightarrow$ Consistency?

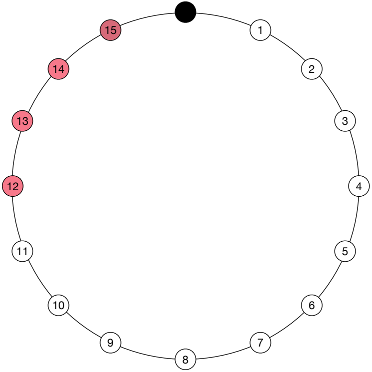
DHT

- Distribution
- Scalability
- Replication
- Elasticity
- Decentralized



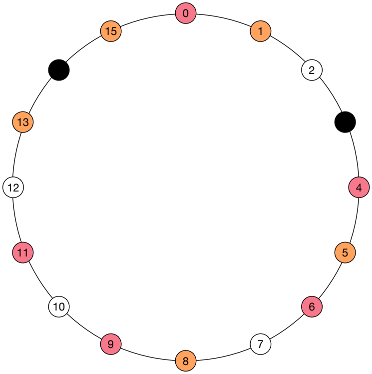
DHT

- Distribution
- Scalability
- Replication
- Elasticity
- Decentralized



DHT

- Distribution
- Scalability
- Replication
- Elasticity
- Decentralized



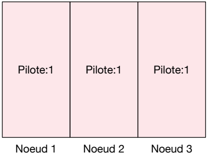
Consistency: Quorum

R + W > N

- **N**: Replication rate
- **W**: Minimum Nb of aknowledged writes
- **R**: Minimum Nb of data to be read in a query
- Parallel Writes and Reads
- If W + R <= N ⇒ Eventual consistency

Example

- Let replication rate N=3



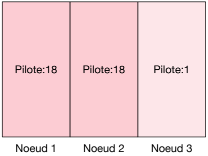
Consistency: Quorum

R + W > N

- **N**: Replication rate
- **W**: Minimum Nb of aknowledged writes
- **R**: Minimum Nb of data to be read in a query
- Parallel Writes and Reads
- If W + R <= N ⇒ Eventual consistency

Example

- Let replication rate N=3
- Update: pilot=18
- 2 acknowledgements (**W=2**)
- 2 parallel reads (**R=2**)



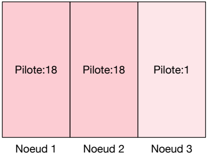
Consistency: Quorum

R + W > N

- **N**: Replication rate
- **W**: Minimum Nb of aknowledged writes
- **R**: Minimum Nb of data to be read in a query
- Parallel Writes and Reads
- If W + R <= N ⇒ Eventual consistency

Example

- Let replication rate N=3
- Update: pilot=18
- 2 acknowledgements (**W=2**)
- 2 parallel reads (**R=2**)
- 2+2>3 ⇒ returns: 18



Consistency: Management

```
CONSISTENCY <level>;
```

level: ANY, ONE, TWO, THREE, QUORUM, ALL

- **ONE/ANY**: by default, at least 1 aknowledgement
  $\Rightarrow$ *Good compromise*
- **QUORUM**: Quorum management
  $\Rightarrow$ *More reads but better consistency*
- **ALL**: Every replicates must be aknowledged
  $\Rightarrow$ *System with few writes*