Complex Systems Theory and Practice

Lecture 11

Piotr Syga

2021-12-17

Introduction

- Cross- platform Framework (open-source)
- ORM with connectors to multiple SQL dialects
- Python-native with (among others) Django and Flask support

Supported dialects

- MySQL, MariaDB
- PostgreSQL
- OracleDB
- Microsoft SQL Server
- SQLite
- externals (Redshift, CreateDB, Firebird, . . .)

Connecting to the database

- import: `from sqlalchemy import create_engine`
- db object creation: `engine = create_engine("`*`dialect[+driver]://user:pass@host/dbname`*`", echo = True)`
  (we can use mock or predefined data, cf. `create_mock_engine`, `engine_from_config`)

Connecting to the database

Selected methods from Engine

- `sqlalchemy.engine.Engine.connect(close_with_result=False)` – returns a new handler to the connection
- `sqlalchemy.engine.Engine.dispose()` – rejects `sqlalchemy.pool.Pool` of the closed connections
- `sqlalchemy.engine.Engine.has_table(table_name, schema=None)`– checking for the existing tables
- `sqlalchemy.engine.Engine.execute(stmt, *multiparams, **params)` – executing SQL queries (**Note**: currently moved to `Connection.*`, kept only for compability)
- `sqlalchemy.engine.Engine.execution_options(**opt)` – creates a new `sqlalchemy.engine.Connection` object
- `sqlalchemy.engine.Engine.begin(close_with_result=False)` – starting a new transaction

Mapping

```
from sqlalchemy import Column, Integer, String
class Students(Base):
    __tablename__ = 'Students'
    indexNo = Column(Integer, primary_key = True)
    name = Column(String)
    lastName = Column(String)
    semester = Column(Integer)
    email = Column(String)
```

Queries

```
from sqlalchemy.orm import sessionmaker
Session = sessionmaker(bind = engine)
session = Session()
result = session.query(Students).filter(Students.semester>2)

for row in result:
   print ("Index number:", row.indexNO, "Name: ",row.name,
 "Last Name:",row.lastName, "Semester":, row.semester,
 "Email:",row.email)
```

Update

```
from sqlalchemy.orm import sessionmaker
Session = sessionmaker(bind = engine)
session = Session()
result = session.query(Students).filter(Students.semester = 7).
update({Students.name:"inż. "+Students.name,
 Students.semester: 0}, synchronize_session = False)
```

Delete

```
from sqlalchemy.orm import sessionmaker
Session = sessionmaker(bind=engine)
session = Session()
z = session.query(Students).get(150931)
session.delete(z)

session.query(Students).filter_by(lastName = 'Doe').
count()
```

Joins

```
from sqlalchemy.orm import sessionmaker
Session = sessionmaker(bind = engine)
session = Session()

for s, m in session.query(Students, Major).
filter(Student.indexNo == Major.student).all():
    print (...)
```

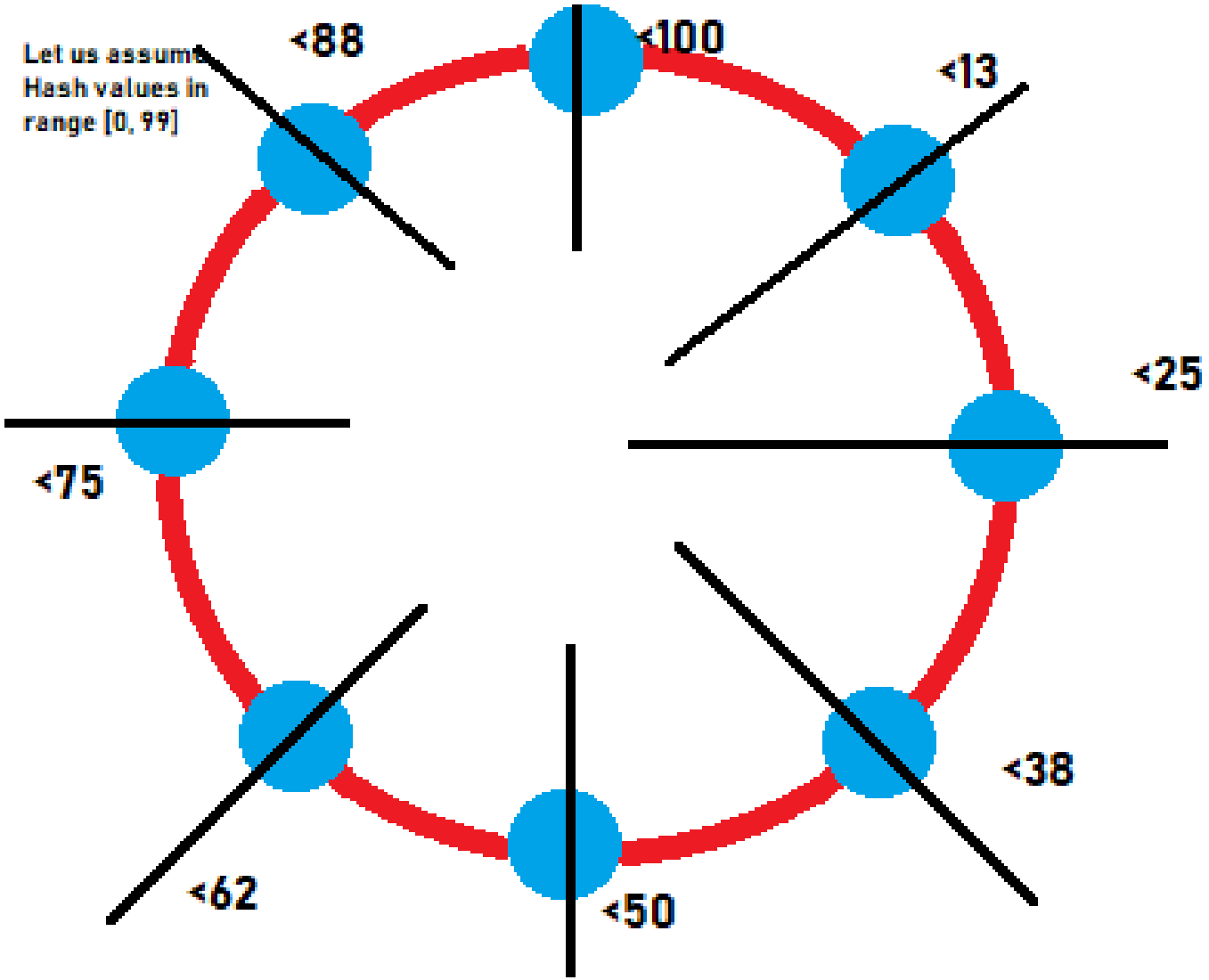
Main properties

- NOSQL columnstore database
- runs on Java Virtual Machine and requires access to a local disk
- each row in a column is stored as a key–value pair
- designed for simplicity and horizontal scaling
- peer-to-peer architecture
- user defined limits of availability and consistency
- fast writes and reads
- supports transactions
- based on tables with flexible schema
- CQL similar to SQL dialects
- *query first* approach
- allows no JOINs
- utilizes Bloom filters(with specified false positive probability), column and table compression and dynamic gossiping

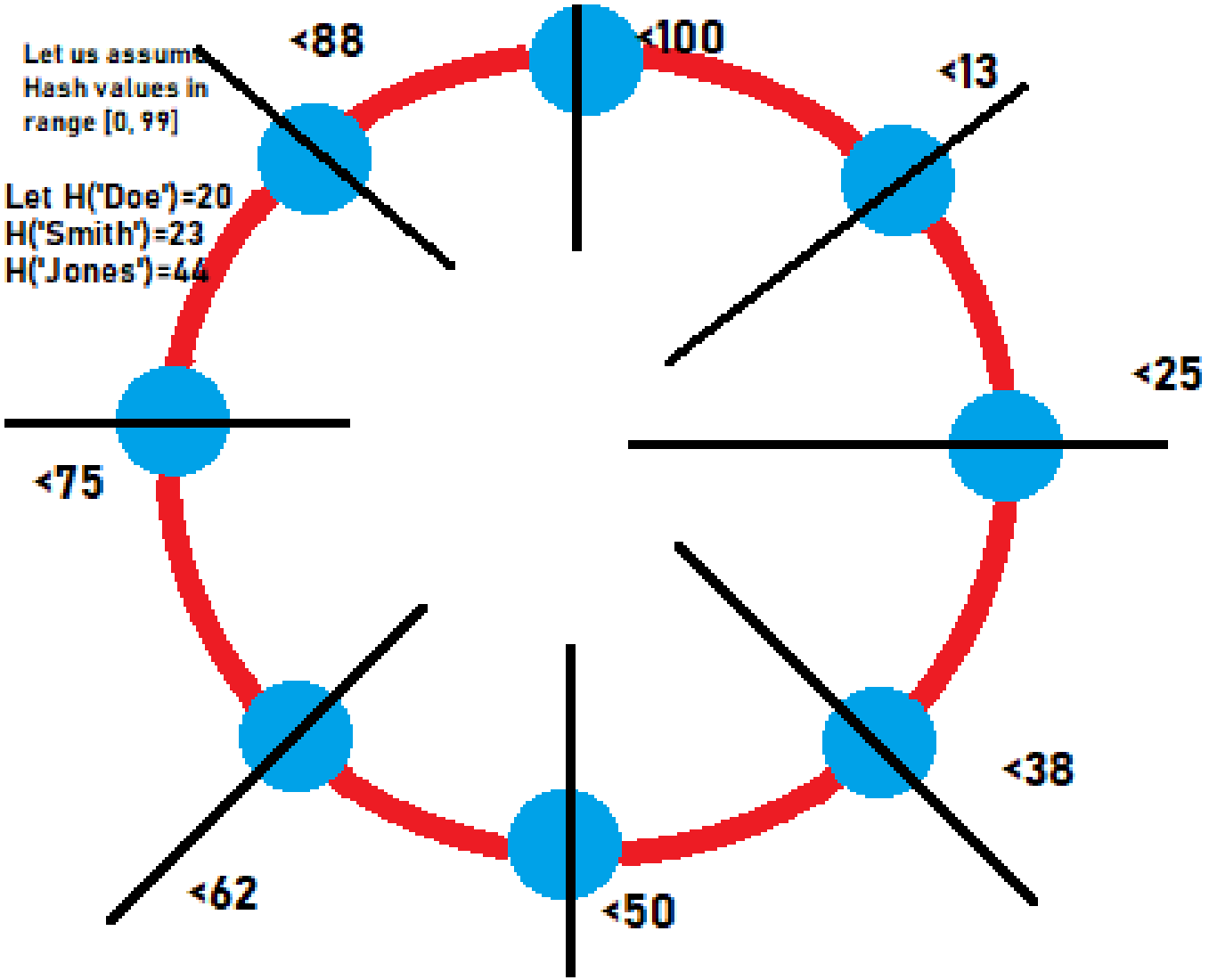
Ring Architecture

- a single keyspace ('equivalent' to SQL database) may be kept on multiple nodes
- data distribution is performed with respect to table's *partition key*
- 64bit Hash function assigns data specific cluster or a node
- each node is assigned token range up to some threshold, in an ordered way in a modulo ring
- each range may be divided into virtual nodes (multiple sections inside one base range)
- the ranges and the number of virtual nodes do not have to be equal
- by default all nodes have 256 virtual nodes

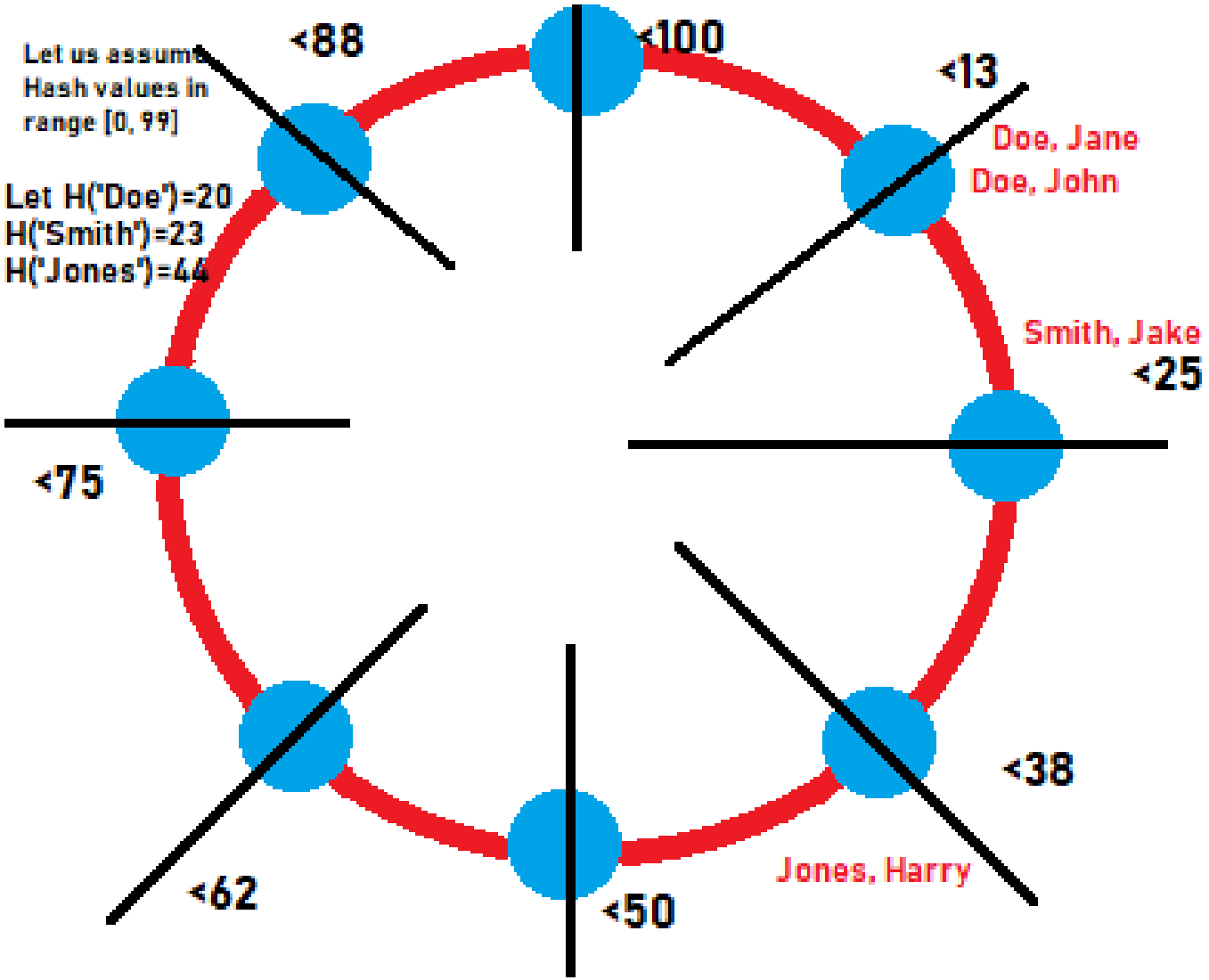
Ring Architecture



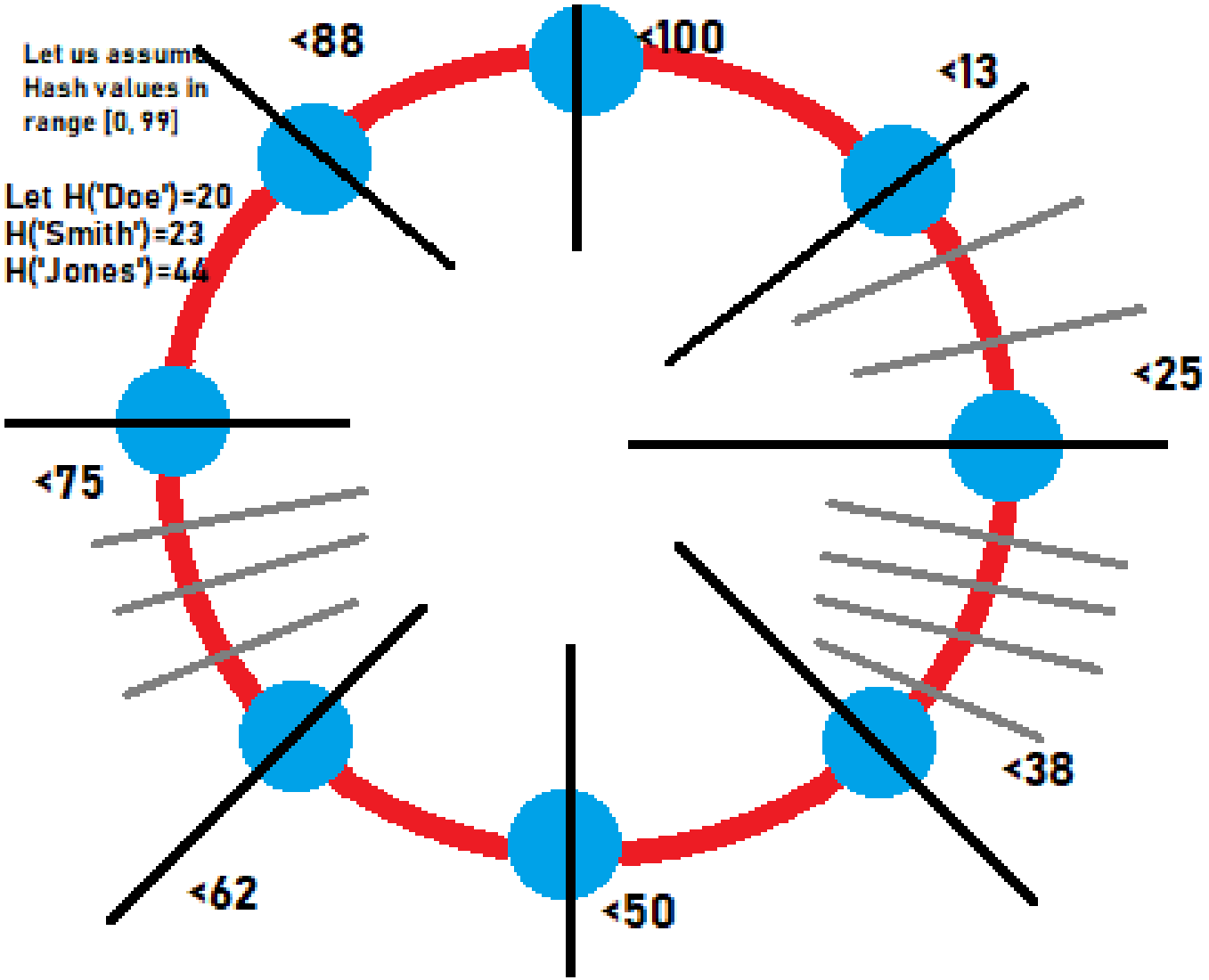
Ring Architecture



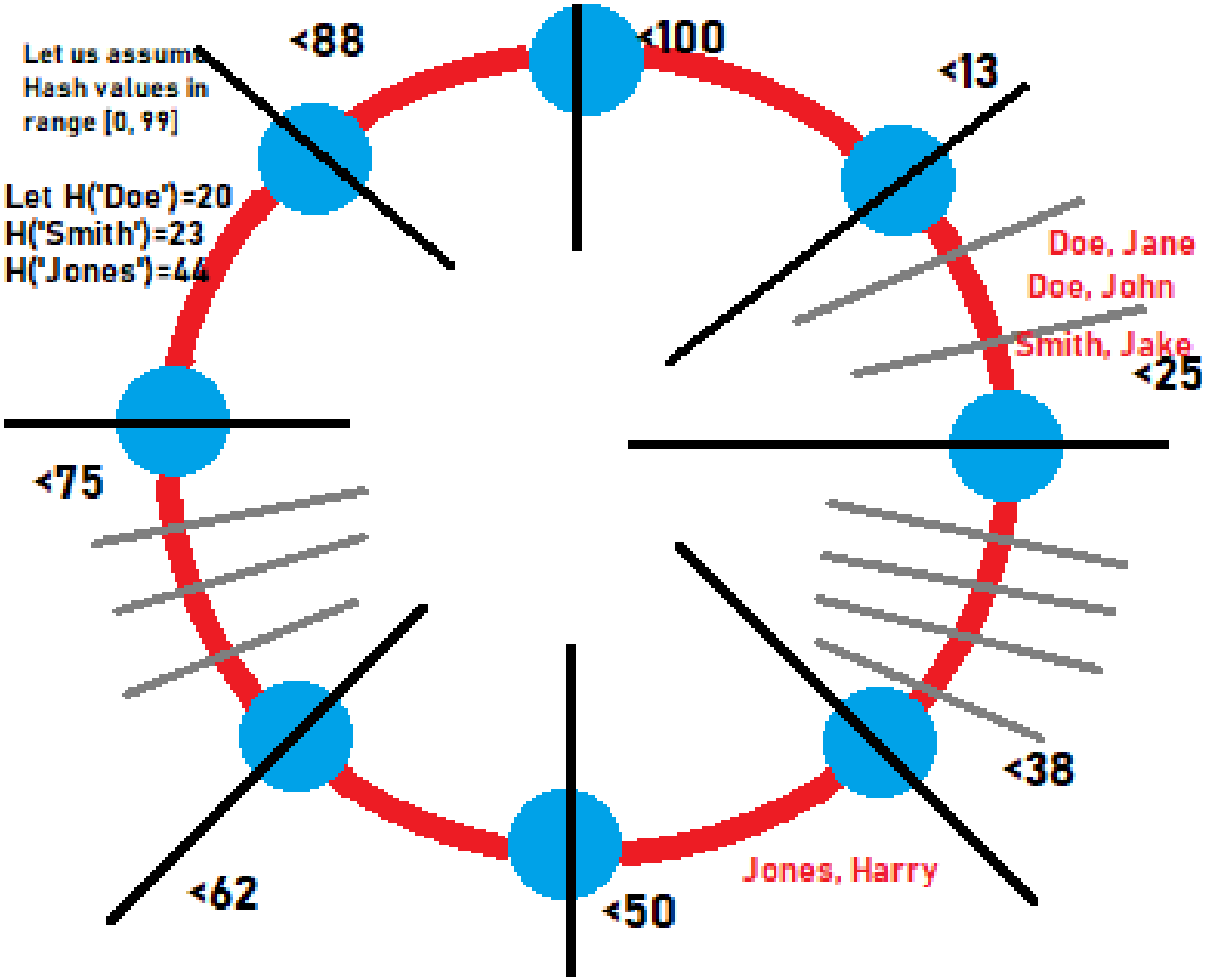
Ring Architecture



Ring Architecture



Ring Architecture



Ring Architecture

Organization of the nodes:

- physical nodes are organized into racks
- several racks may be connected (e.g., in the same localization)
- multiple racks, not necessarily directly connected, create a data center
- `nodetool status` displays information on nodes - their up or down status, rack and data center assignments
- the assignment for a specific rack and data center is stored in `cassandra-rackdc.properties` file

Replicas

- In order to prevent data loss on node failure and to speed up the response, data is stored in multiple nodes.
- Replication factor property specifies how many replicas we want to keep (by default 3).
- Basic replication strategies include: Simple strategy, local strategy and network topology strategy
- `SimpleStrategy` (recommended for single data center) – moved clockwise on the ring and decides the nodes at random
- `LocalStrategy` (is used only for internal keystores) – distributes among nodes in a single cluster
- `NetworkTopologyStrategy` – keeps data spread and allows to specify number of replicas kept in each data center

Replicas

- Since data is kept on multiple nodes, we have to assure consistency.
- Whenever data is requested a consistency parameter is read to determine the policy on the number of nodes that have to return the answer.
- Consistency levels:
  - `ANY` (returns successful write if at least one node is written)
  - `(local) <number>` e.g., `TWO` (returns when selected number of nodes are written to/read from – in case of different values, decides based on the newest and repairs older values)
  - `(local) QUORUM` (waits until majority responds, the newest value is selected and older are fixed)
- `CONSISTENCY` command checks current consistency level (it is set globally for a session), to change it – `CONSISTENCY <type>`, e.g., `CONSISTENCY QUORUM`

Snitches

- Snitches are responsible for determining which node is located in which rack and data center.
- A snitch assigns user for connection to a given node based on the query, their location and information about the delays of the nodes.
- Several snitch types are supported by Cassandra, including multiple vendor–specific ones.
- *Simple snitch* – when all nodes are on a single server
- *Property File Snitch* – each node has a file with IPs, data center and rack of each node
- *Gossiping Property File Snitch* – each node keeps only its own data center and rack information in `cassandra-rackdc.properties` file, information spread via gossiping protocol
- Snitch configuration is kept in `cassandra.yaml`
- A specific snitch time that works simultaneously with others is `Dynamic Snitch` that keeps track of the location of the replicas and calculates delays to those nodes

Basics

- Cassandra provides a simple query language that is based on SQL3.
- The data is stored vertically, inside of keyed tables that are kept in a keyspace.
- Each table has to have a primary key that consists of (possibly complex) partition key and optional clustering columns.
- Each column has a single (possibly complex) data type.
- In a row, inside a fixed non-key column, there can be stored an instance of a given data type, a `null` or nothing at all.
- Due to time-efficiency there are significant restrictions in the possibilities of a query (no JOINs, restricted WHERE and ORDER BY), hence tables are created with a specific query in mind, leading to data redundancy.

Basics

indexNo	lastName	firstName	major
200831	{lastName:Doe}	{firstName:Jane}	{major: BDA}
200832	{lastName:Doe}	{firstName:John}	{major: BDA}
200833	{lastName:Jonson}	{firstName:John}	{major: BDA}
200999	{lastName:Smith}	{firstName:Frank}	{major: BDA}
200840	{lastName:Smithy}	{firstName:Paul}	{major: CS}
220832	{lastName:West}	{firstName:Eve}	{major: OP}

Basics

indexNo	lastName	firstName	major
200831	{lastName:Doe}*2	{firstName:Jane}	{major: BDA}*4
200832		{firstName:John}*2	
200833	{lastName:Jonson}		
200999	{lastName:Smith}	{firstName:Frank}	
200840	{lastName:Smithy}	{firstName:Paul}	{major: CS}
220832	{lastName:West}	{firstName:Eve}	{major: OP}

Data types

Simple types:

- logical – Boolean
- numerical – (big)int, decimal, double, float, varint
- alphabetic – ascii, inet, text, varchar

Special types:

- blob
- counter
- timestamp
- uuid and timeuuid
- user defined types (`CREATE TYPE`, `ALTER TYPE`)

Collection types:

- list<type>
- set<type>
- map<type, type>

Data types

```
CREATE TYPE address (
city text,
postalCode text,
street text,
building int,
flat int,
phone set<int>
);

ALTER TYPE address ADD email text;
```

Data types

- uuid – a (pseudo)random, unique, identifier with a preset length, we can generate a new one using `uuid()`
- timeuuid – a (pseudo)random, unique, identifier with a preset length, that is partially dependent on the time of creation, i.e., IDs created after each other are similar and may be ordered, we can generate a new one using `now()`
- counter – a non-settable type, that may be modified only via updates, in particular cannot be initialized, we start with update column=column+1, so that counter column gets value 1, later on we increment or decrement the value

Keyspaces

- Keyspaces are containers that keep the data-filled tables.
- Since no JOINs are possible between tables, keyspaces are kept for managing the settings and parameters.
- `CREATE KEYSPACE` <name> <options> (e.g., `CREATE KEYSPACE` csKS `WITH` replication={'class' : 'SimpleStrategy', 'replication_factor': '3'} `AND` 'durable_writes'='false');
- `DESCRIBE KEYSPACES` shows available keyspaces, `USE` csKS; `DESCRIBE TABLES`; describes tables in a given keyspace
- To remove a keyspace we use `DROP` <ksName>.

Tables

- CQL uses tables similar tables to the ones known from relational databases.
- We can create a new table using `CREATE TABLE` <name> (field1 type1 `PRIMARY KEY`, field2 type2, ...);
- Each column has to be of a single data type.
- Check constraints and triggers are not supported.
- We change the tables using `ALTER TABLE`, we remove a table using `DROP TABLE`, a single command for deleting all rows in a table is also available – `TRUNCATE` <name>.
- We use `DESCRIBE` <tablename> in order to get information (columns, CREATE statement, compression parameters, Bloom filter FPR, TTL, index intervals, . . .)
- Key specification may include multicolumn keys – `CREATE TABLE` <name> (field1 type1, field2 type2, ..., `PRIMARY KEY`(<partition key>, <clustering column>));
- The partition key may be complex, e.g., (lastName, firstName) with the order determining the clustering.

Tables

Examples

`CREATE TABLE` StudendsByID (indexNo int `PRIMARY KEY`, lastName text, firstName text, major text);

`CREATE TABLE` StudendsByName (lastName text, firstName text, startingYear int, email text, indexNo int, major text, `PRIMARY KEY`((lastName, firstName), startingYear)) WITH bloom_filter_fp_chance = 0.03;

`ALTER TABLE` StudendsByName `ADD` tutor text;

`DESCRIBE` StudendsByName;

`DROP TABLE` StudendsByID;

Selects

- Selects follow the same syntax as in SQL however there are significant constraints.
- The select query, by default, may not condition the selection (no `WHERE` clause) or has to use the whole partition key in the condition.
- In order to allow conditions on a different column, we have to use `ALLOW FILTERING`, use a different table, a view or create a secondary index.
- We can use writetime(column) to get the time of the last update of each row in the column.
- The results may be `ORDER`ed BY one or many clustering columns.
- We can `GROUP BY` by one or more clustering column.
- We can use standard aggregate functions – `COUNT`, `SUM`, `MIN`, `MAX`, `AVG` or create our own.

Selects

Examples

`SELECT * FROM StudendsByName;`

`SELECT lastName, writetime(lastName) FROM StudendsByName;`

`SELECT * FROM StudendsByName WHERE lastName='Doe';` — will result in an error

`SELECT * FROM StudendsByName WHERE lastName='Doe' AND firstName='Jane';`

`SELECT * FROM StudendsByName ORDER BY startingYear;`

`SELECT MIN(startingYear) FROM StudendsByName;`

Creating aggregate functions

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

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

CREATE AGGREGATE mean(int)
SFUNC avgState
STYPE tuple<int,bigint>
FINALFUNC avgFinal
INITCOND (0,0);
```