

Column Databases

UA.DETI.CBD

José Luis Oliveira / Carlos Costa

Overview

- ❖ Column-oriented datastores
 - Concept
- ❖ Google BigTable
- ❖ Cassandra
 - Data Model
 - Cassandra query language
 - DDL statements
 - DML statements

Concept

- ❖ **Store and process data by column instead of row**
- ❖ Origin in analytics and business intelligence
 - usually consist of aggregation queries
 - operating in a shared-nothing massively parallel processing architecture to build high-performance applications
- ❖ Usually described as “sparse, distributed, persistent multidimensional sorted map”
- ❖ Main inspiration for column-oriented datastores is Google’s Bigtable

Store by Column vs Row

❖ Table example:

ID	name	address	zip code	phone	city	country	age
1	Benny Smith	23 Workhaven Lane	52683	14033335568	Lethbridge	Canada	43
2	Keith Page	1411 Lillydale Drive	18529	16172235589	Woodridge	Australia	26
3	John Doe	1936 Paper Blvd.	92512	14082384788	Santa Clara	USA	33

❖ Store by row:

```
1,Benny Smith,23 Workhaven Lane,52683,14033335568,Lethbridge,Canada,43;2,Keith Page,1411 Lillydale Drive,18529,16172235589,Woodridge,Australia,26;3,John Doe,1936 Paper Blvd.,92512,14082384788,Santa Clara,USA,33;
```

❖ Store by column:

```
1,2,3;Benny Smith,Keith Page,John Doe;23 Workhaven Lane,1411 Lillydale Drive,1936 Paper Blvd.;52683,18529,92512;14033335578,16172235589,14082384788;Lethbridge,Woodridge,Santa Clara;Canada,Australia,USA;43,26,33;
```

Store by Column vs Row

- ❖ Store by row:

ID	name	address	zip code	phone
1	Benny Smith	23 Workhaven Lane	52683	14033335568
2	Keith Page	1411 Lillydale Drive	18529	16172235589
3	John Doe	1936 Paper Blvd.	92512	14082384788

- ❖ Store by column:

ID	name	address	zip code	phone
1	Benny Smith	23 Workhaven Lane	52683	14033335568
2	Keith Page	1411 Lillydale Drive	18529	16172235589
3	John Doe	1936 Paper Blvd.	92512	14082384788

Columnar databases

❖ Good for

- Queries that involve only a few columns
- Aggregation queries against vast amounts of data
- Column-wise compression

❖ Not so good

- Incremental data loading
- Online Transaction Processing (OLTP) usage
- Queries against only a few rows

(Dis)Advantages explained...

- ↑ Some queries could become really fast
 - aggregation queries
 - function over fields, e.g. average age of users
- ↑ Better data compression
 - when running the algorithms on each column (similar data)
 - accentuated as your dataset becomes larger
- ↓ Aggregation is great, but some applications need to show data for each individual record
 - columnar databases are generally not great for these types of queries
- ↓ Writing new data could take more time
 - inserting a new record into a row-oriented database is a simple write operation
 - updating many values in a columnar database could take much more time

Data Model

❖ Column family (table)

- table is a collection of similar rows (not necessarily identical)

❖ Row

- row is a collection of columns
 - should encompass a group of data that is accessed together
- associated with a unique row key

❖ Column

- column consists of a column name and column value (and possibly other metadata records)
- scalar values, but also sets, lists and maps

Usage

❖ Suitable use cases

- event logging, content management systems, blogs, ...
 - i.e., for structured flat data with similar schema
- batch processing via map reduce

❖ When not to use

- ACID transactions are required
- Complex queries: joining, ...
- Early prototypes
 - i.e., when database design may change

Representatives Technologies



HYPERTABLE^{INC}



Representatives Technologies

- ❖ Bigtable is the inspiration for column-oriented datastores
- ❖ Datastores influenced by Bigtable
 - Hypertable
 - HBase
- ❖ Cassandra is an extension of Bigtable with aspects of Amazon's Dynamo

Why Bigtable

NoSQL Google

- Internal Google database that kickstarted the NoSQL industry
- Web indexes behind search engine took too long to build
- Needed real-time access to petabytes of data
- 2006 research paper describing Bigtable
 - Led to HBase, Cassandra, and other NoSQL databases
 - Received the SIGOPS Hall of Fame Award
- Bigtable powers Gmail, Google Maps, and other services
- In 2015, Google made Bigtable available as a service

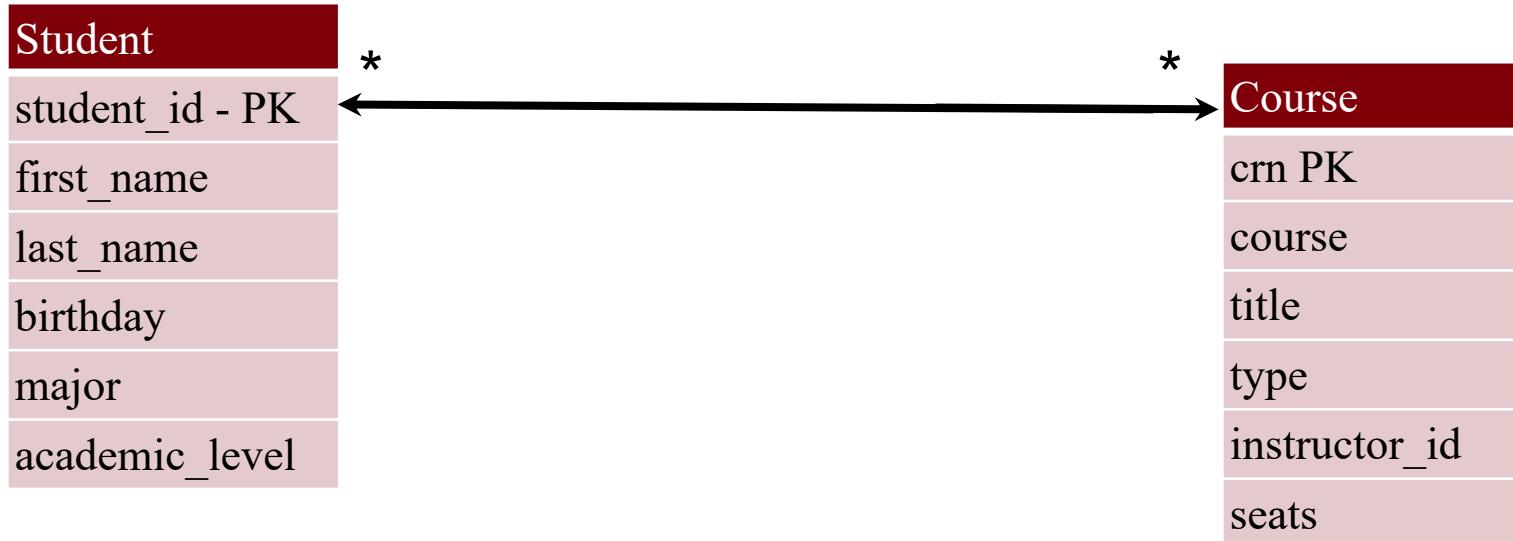


https://www.youtube.com/watch?v=1qieV-WCU_w

<https://www.usenix.org/legacy/event/osdi06/tech/chang/chang.pdf>

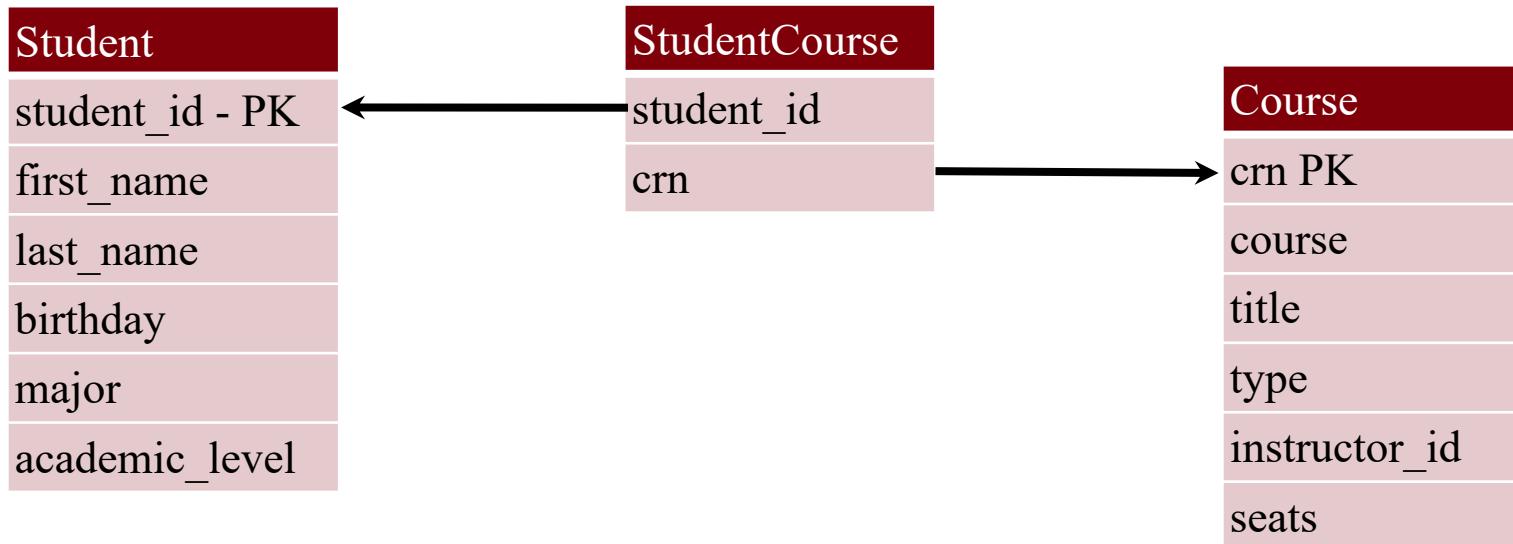
Relational vs BigTable

❖ Example:

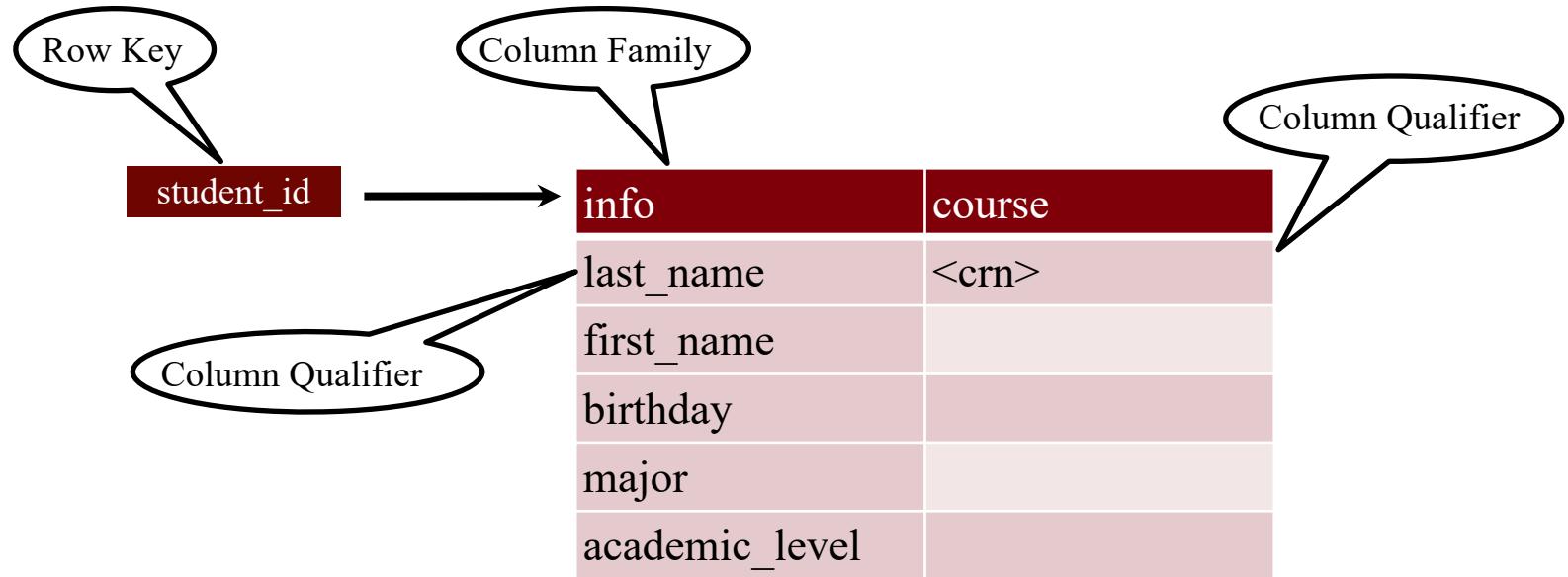


Relational vs BigTable

❖ Example:

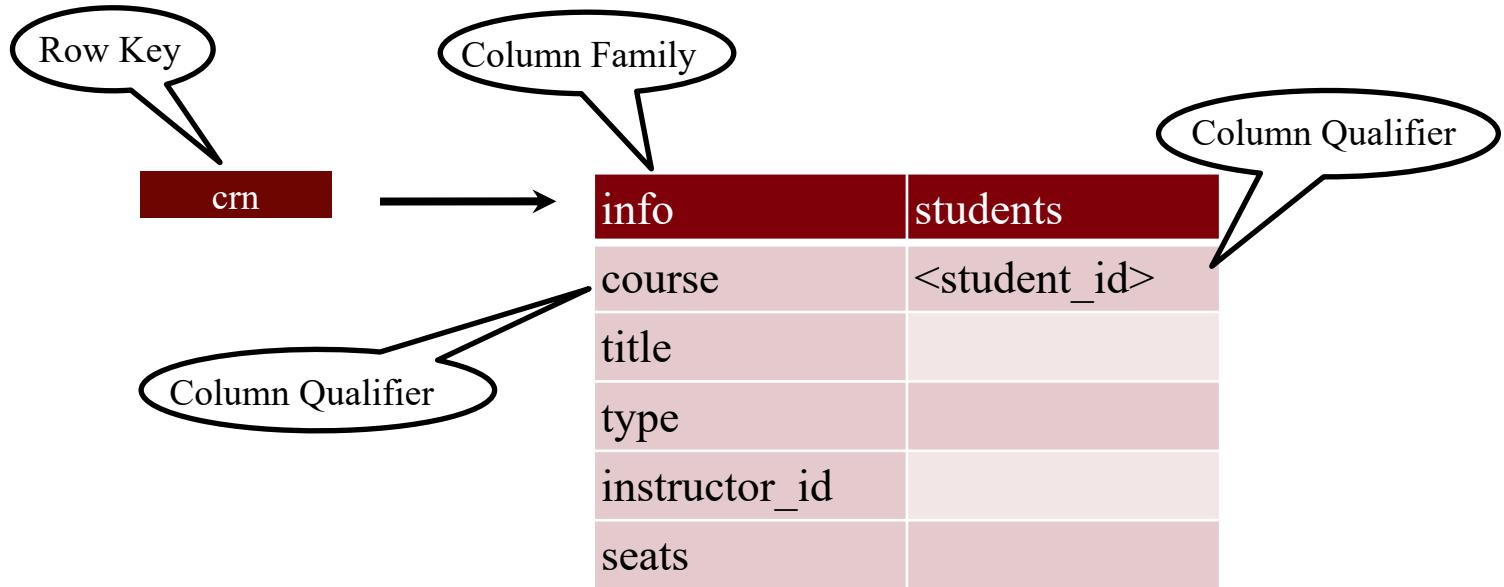


BigTable – Student table



info:first_name	info:last_name	info:major	courses:96322	courses:96320
“Sergejs”	“Melderis”	“Computer Science”	“YES”	“NO”

BigTable – Course table

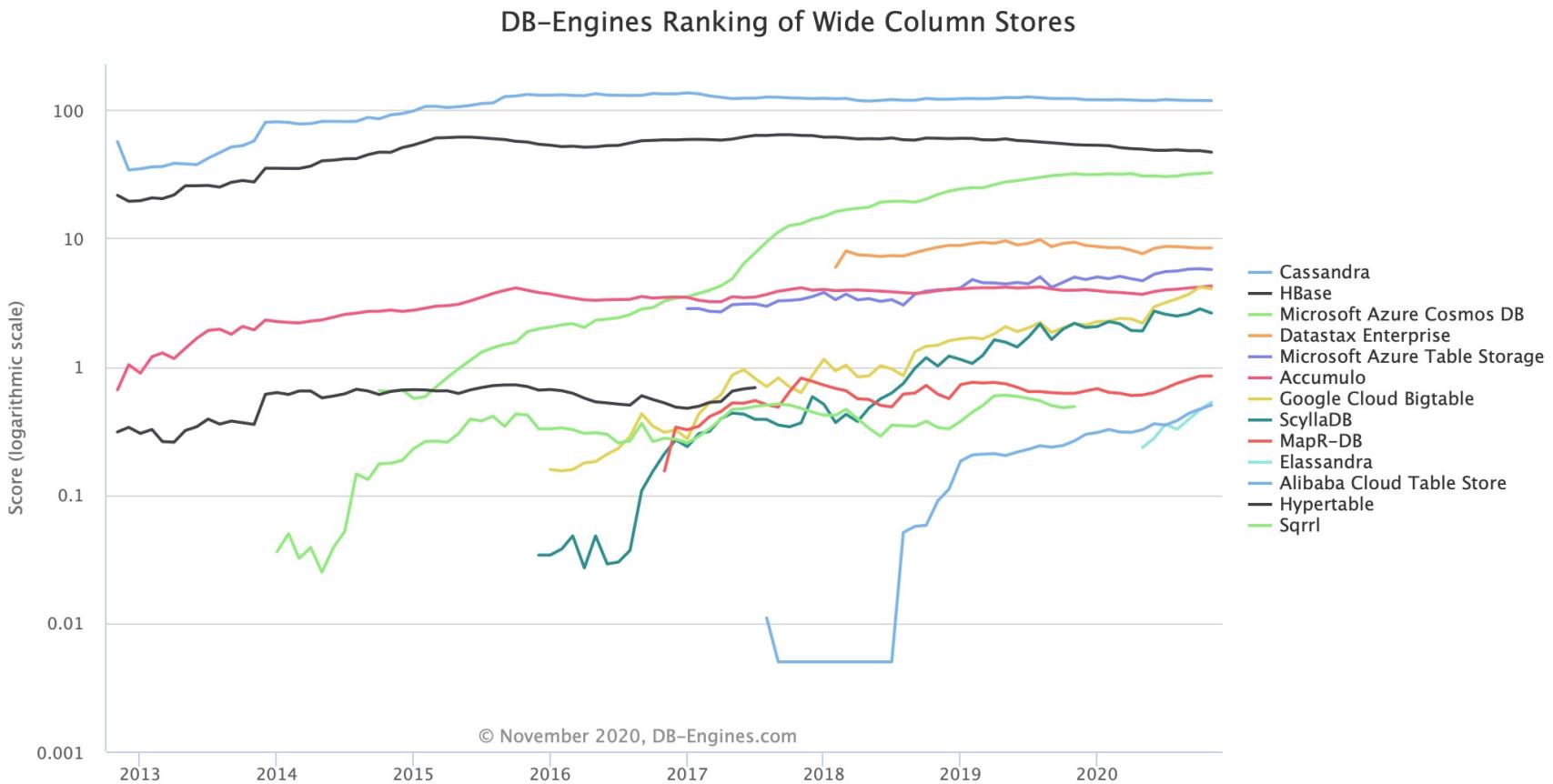


info:course	info:title	info:instructor_id	students:905514	students:905520
“96322” → “CS5204”	“Operating Systems”	“1983943”	“YES”	“YES”

Apache Cassandra



Cassandra Ranking



https://db-engines.com/en/ranking_trend/wide+column+store

Cassandra Overview

❖ Features:

- open-source, high availability, linear scalability, sharding (spanning multiple datacenters), peer-to-peer configurable replication, tunable consistency, MapReduce support

❖ Implemented in Java

❖ Cross-platform

❖ Originally developed by Facebook

- open-sourced in 2008

❖ Adopted by Twitter, Rackspace, Netflix, eBay, GitHub, Instagram, etc.

❖ Developed by Apache Software Foundation

- <http://cassandra.apache.org/>

History: Facebook Inbox Search

- ❖ Cassandra developed to address this issue
- ❖ Performance tested with more than 50TB of user messages data in a cluster of 150 nodes (2 data centers)
- ❖ Search index of all messages in 2 ways:
 - term search: search by a keyword
 - interactions search: search by a user id

Latency	Search Interactions	Search Term
Min	7.69 ms	7.78 ms
Median	15.69 ms	18.27 ms
Max	26.13 ms	44.41 ms

Cassandra vs MySQL

- ❖ > 50 GB of Data

Average Time	MySQL	Cassandra
Write	~ 300 ms	0.12 ms
Read	~ 350 ms	15 ms

* facebook data / use case

Motivations

- ❖ High Availability
- ❖ High Write Throughput
 - while not sacrificing read efficiency
- ❖ Fault Tolerance
- ❖ High and incremental scalability
- ❖ Reliability at massive scale

Clusters, Data Centers, Nodes

❖ Node

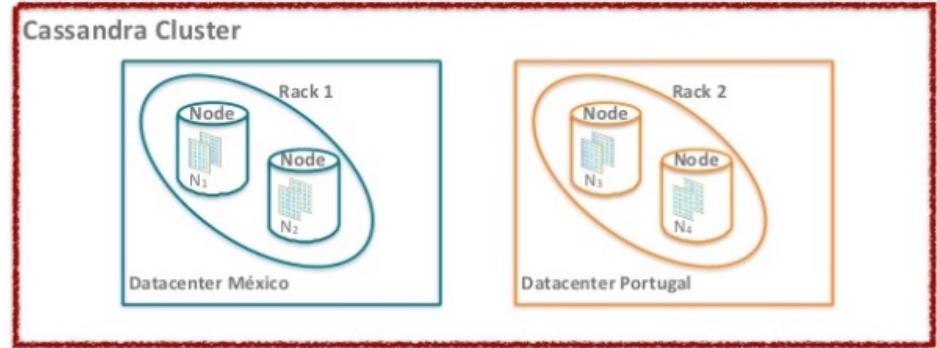
- a machine where Cassandra is running

❖ Data Center

- A collection of related nodes
- Synonymous of replication group
 - replication is set by data center
 - a grouping of nodes configured together for replication purposes
- Using separate data centers allows:
 - dedicating each data center for different processing tasks
 - satisfying requests from a data center close to client

❖ Cluster

- A cluster is a collection of data centers
 - the same data is written in all data center



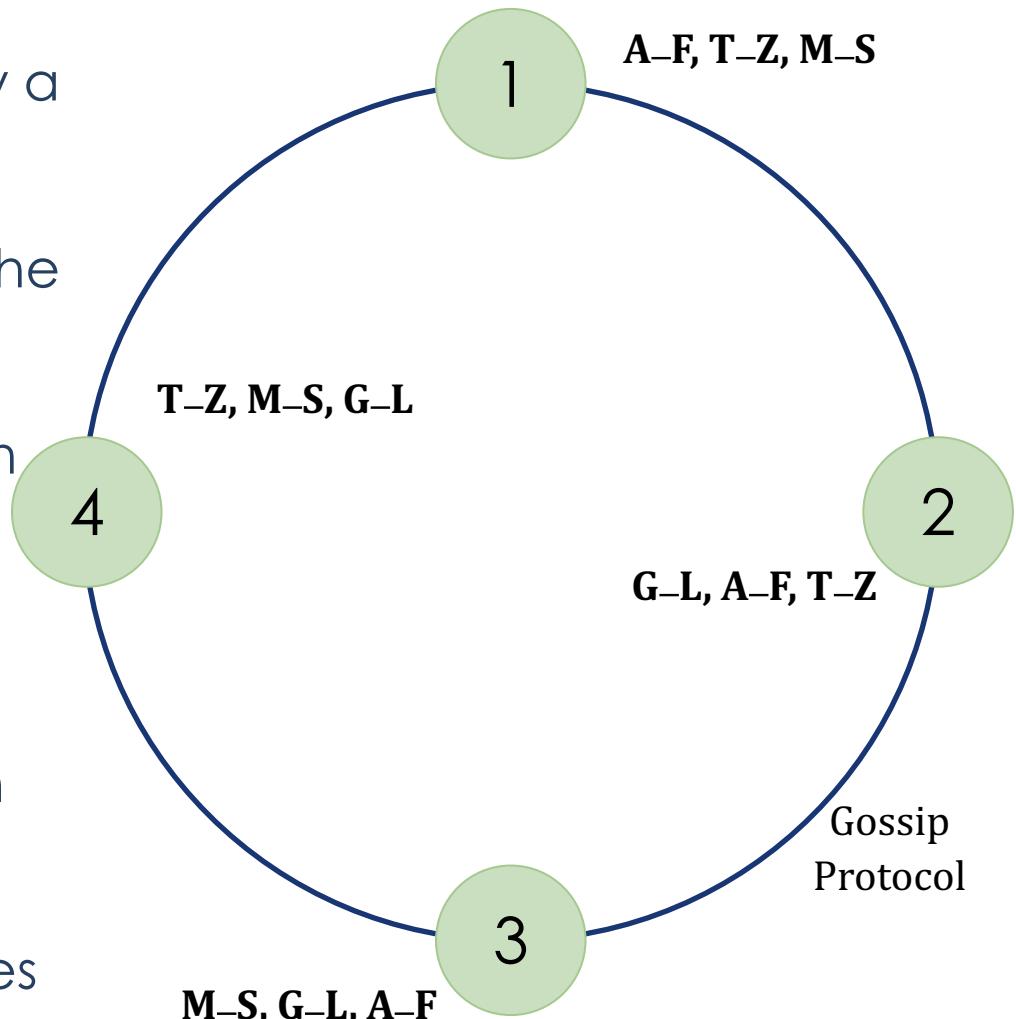
Cassandra – System Architecture

- ❖ Cluster Membership
 - how nodes are added, deleted to the cluster
- ❖ Partitioning
 - how data is partitioned across nodes
 - nodes are logically structured in Ring Topology
 - hashed value of key associated with data partition is used to assign it to a node in the ring
- ❖ Replication
 - how data is duplicated across nodes
 - each data item is replicated at N (replication factor) nodes

Note: Those topics will be discussed more in detail in posterior lessons. In this phase, we will be only focusing in the data model.

Network Nodes Topology

- ❖ Cluster Data managed by a ring of nodes
- ❖ Each node has a part of the database
- ❖ Rows distribution based on primary key
 - row lookups are fast
- ❖ Multiple nodes have the same data to ensure both availability and durability
- ❖ No master node – all nodes can perform all operations



Data Model (*keyspaces* → **tables** → **rows** → **columns**)

❖ Keyspace

- a **namespace** that defines data replication on nodes
- a cluster contains one keyspace per node

❖ Table (column family)

- collection of (similar) rows
- a multi dimensional map indexed by key (row key)
- table schema must be specified but can be modified later
- 2 types: simple or super (nested Column Families)

❖ Row

- collection of columns
- rows in a table do not need to have the same columns
- each row is **uniquely identified** by a **primary key**

❖ Column

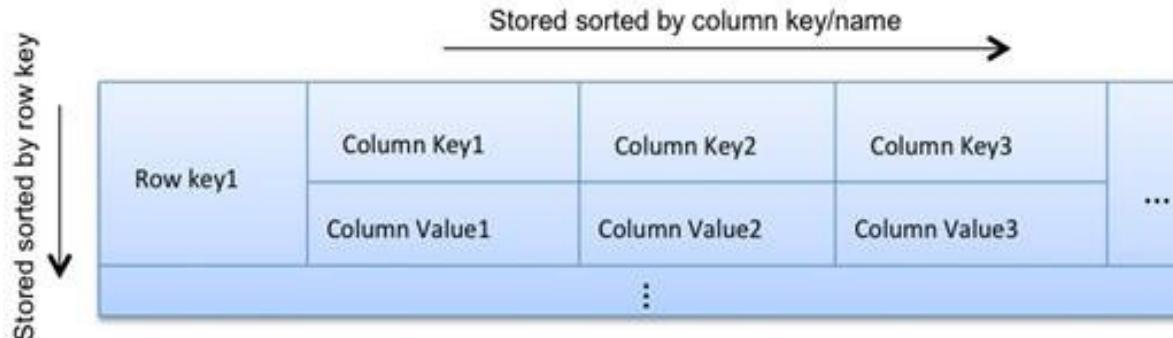
- name-value pair + additional data

Data Model

Relational Model	Cassandra Model
Database	Keyspace
Table	Column Family (CF)
Primary key	Row key
Column name	Column name/key
Column value	Column value

A nested sorted map is an accurate analogy for each column family:

```
Map<RowKey, SortedMap<ColumnKey, ColumnValue>>
```



Data Model – Example

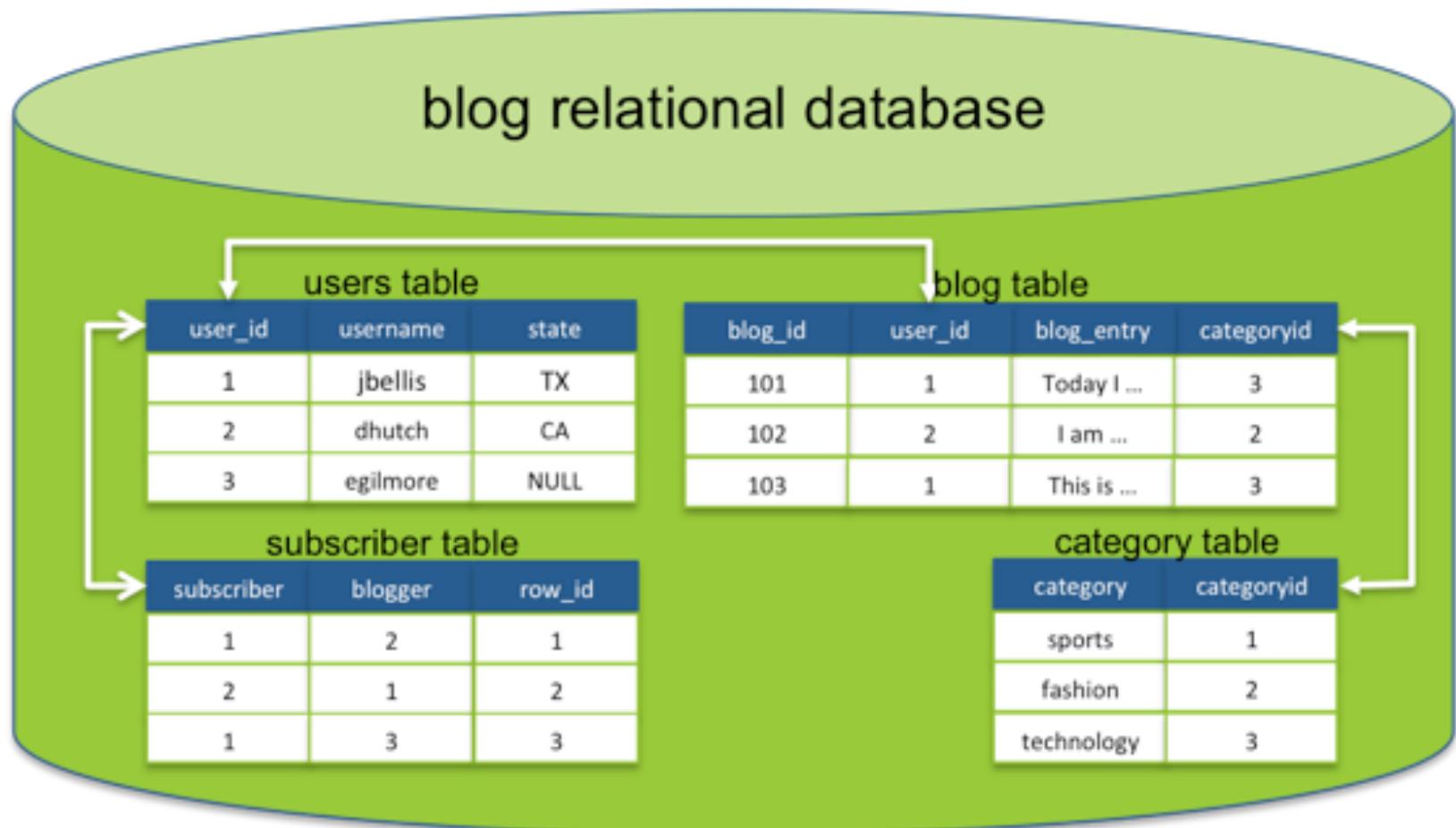
actors

id			
trojan	name (Ivan, Trojan)	year 1964	movies { samotari, medvidek }
machacek	name (Jiří, Macháček)	year 1966	movies { medvidek, vratnelahve, samotari }
schneiderova	name (Jitka, Schneiderová)	year 1973	movies { samotari }
sverak	name (Zdeněk, Svérák)	year 1936	movies { vratnelahve }

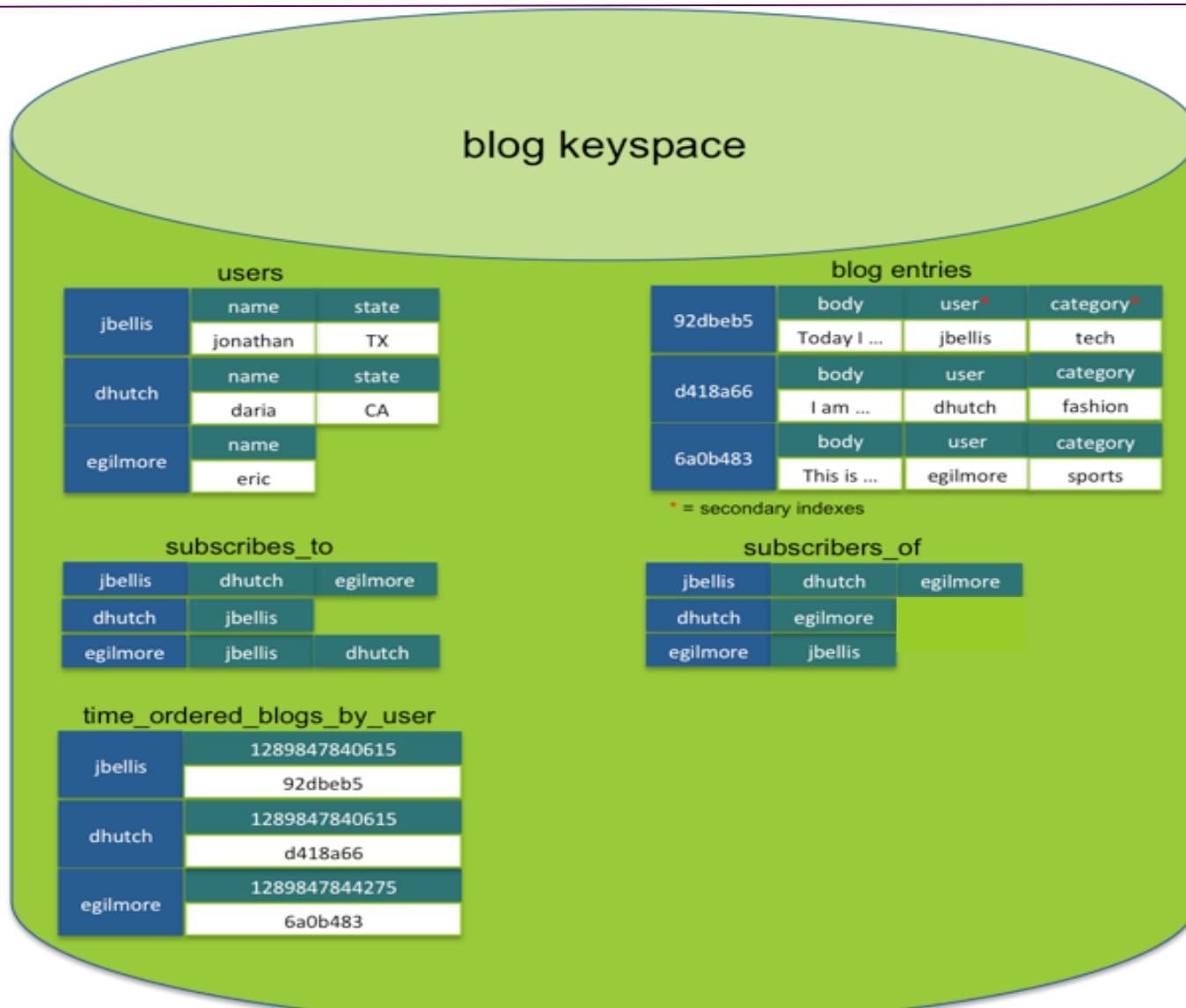
movies

id				
samotari	title Samotáři	year 2000	actors null	genres [comedy, drama]
medvidek	title Medvídek	director (Jan, Hřebejk)	year 2007	actors { trojan: Ivan, machacek: Jirka }
vratnelahve	title Vratné lahve	year 2006	actors { machacek: Robert Landa }	
zelary	title Želary	year 2003	actors { }	genres [romance, drama]

Cassandra vs Relational Example



Cassandra vs Relational Example



Data Model – Column Values

- ❖ Empty value
 - null
- ❖ Atomic value
 - **native data types**
 - texts, integers, dates, ...
 - **tuples**
 - tuple of anonymous fields, each of any type (even different)
 - **user defined types (UDT)**
 - set of named fields of any type
- ❖ Collections
 - **lists, sets, and maps**
 - nested tuples, UDTs, or collections are allowed, but currently only in frozen mode (such elements are serialized when stored)

Data Model – Additional Data

- ❖ Associated with the whole column in case of atomic values, or every element of a collection
- ❖ **Time-to-live** (TTL)
 - after a certain amount of time (number of seconds) a given value is automatically deleted
- ❖ **Timestamp** (writetime)
 - timestamp of the last value modification
 - assigned automatically or manually as well
- ❖ Both elements can be queried
 - but not in case of collections and their elements

Cassandra API

❖ CQLSH

- interactive command line shell
- bin/cqlsh
- uses CQL (Cassandra Query Language)

❖ Client drivers

- provided by the community
- available for various languages
 - Java, Python, Ruby, PHP, C++, Scala, Erlang, ...

Cassandra Query Language (CQL)

- ❖ Declarative query language inspired by SQL

- <https://cassandra.apache.org/doc/latest/cql/>
 - <http://docs.datastax.com/en/dse/6.8/cql/>

- ❖ **DDL statements (Data Definition Lang.)**

CREATE KEYSPACE – creates a new keyspace

CREATE TABLE – creates a new table

...

- ❖ **DML statements (Data Manipulation Lang.)**

SELECT – selects and projects rows from a single table

INSERT – inserts rows into a table

UPDATE – updates columns of rows in a table

DELETE – removes rows from a table

...

Keyspace

❖ CREATE KEYSPACE

`CREATE KEYSPACE [IF NOT EXISTS] keyspace_name WITH options`

❖ Replication option is mandatory

- SimpleStrategy
 - one replication factor for the whole cluster
- NetworkTopologyStrategy
 - individual replication factor for each data center

```
CREATE KEYSPACE Excelsior
    WITH replication = {'class': 'SimpleStrategy', 'replication_factor' : 3};

CREATE KEYSPACE Excalibur
    WITH replication = {'class': 'NetworkTopologyStrategy', 'DC1' : 1, 'DC2' : 3}
        AND durable_writes = false;
```

Keyspace (cont)

- ❖ USE KEYSPACE

USE KEYSPACE

- ❖ DROP KEYSPACE

DROP KEYSPACE [IF EXISTS]

- ❖ ALTER KEYSPACE

ALTER KEYSPACE keyspace_name WITH options

```
ALTER KEYSPACE Excelsior
    WITH replication = {'class': 'SimpleStrategy', 'replication_factor' : 4};
```

Native Data Types

type	constants supported	description
ascii	string	ASCII character string
bigint	integer	64-bit signed long
blob	blob	Arbitrary bytes (no validation)
boolean	boolean	Either true or false
counter	integer	Counter column (64-bit signed value). See Counters for details
date	integer, string	A date (with no corresponding time value). See Working with dates below for details
decimal	integer, float	Variable-precision decimal
double	float	64-bit IEEE-754 floating point
duration	duration,	A duration with nanosecond precision. See Working with durations below for details
float	integer, float	32-bit IEEE-754 floating point
inet	string	An IP address, either IPv4 (4 bytes long) or IPv6 (16 bytes long). Note that there is no <code>inet</code> constant, IP address should be input as strings
int	integer	32-bit signed int
smallint	integer	16-bit signed int
text	string	UTF8 encoded string
time	integer, string	A time (with no corresponding date value) with nanosecond precision. See Working with times below for details
timestamp	integer, string	A timestamp (date and time) with millisecond precision. See Working with timestamps below for details
timeuuid	uuid	Version 1 UUID, generally used as a “conflict-free” timestamp. Also see Timeuuid functions
tinyint	integer	8-bit signed int
uuid	uuid	A UUID (of any version)
varchar	string	UTF8 encoded string
varint	integer	Arbitrary-precision integer

Table – Create Statement

❖ CREATE TABLE

- creates a new table within the current keyspace
- each table must have one primary key specified

```
CREATE TABLE [ IF NOT EXISTS ] table_name
    '(' column_definition ( ',' column_definition )*
        [ ',' PRIMARY KEY '(' primary_key ')' ]
    ')'
    [ WITH table_options ]

column_definition ::= column_name cql_type [ STATIC ] [ PRIMARY KEY]
primary_key ::= partition_key [ ',' clustering_columns ]
    partition_key ::= column_name | '(' column_name ( ',' column_name )* ')'
    clustering_columns ::= column_name ( ',' column_name )*
table_options ::= COMPACT STORAGE [ AND table_options ] | CLUSTERING
    ORDER BY '(' clustering_order ')'
    [ AND table_options ] | options
clustering_order ::= column_name (ASC | DESC) ( ',' column_name (ASC | DESC) )*
```

Table Primary Key

Primary key has two parts:

- ❖ Compulsory **partition key**
 - single column or multiple columns
 - describes how table rows are distributed among partitions
- ❖ Optional **clustering columns**
 - defines the clustering order, i.e. how table rows are locally stored within a partition

```
CREATE TABLE groups (
    groupname text,
    username text,
    email text,
    age int,
    PRIMARY KEY (groupname, username)
)
```

PRIMARY KEY has two components: **groupname**, which is the **partitioning key**, and **username**, which is called the **clustering key**. This will give us one partition per groupname. Within a particular partition (group), rows will be ordered by username.

Keys Roles

❖ Partition Key

- responsible for data partitioning across database nodes

❖ Clustering Key

- responsible for data sorting within the partition

❖ Primary Key

- **Partition Key + Clustering Key**
- is equivalent to the **Partition Key** in a single-field-key table

❖ Composite Key

- partition and clustering key can have multiple-columns

```
create table mytable (
    k_part_one text,
    k_part_two int,
    k_clust_one text,
    k_clust_two int,
    k_clust_three uuid,
    data text,
    PRIMARY KEY((k_part_one,k_part_two), k_clust_one, k_clust_two, k_clust_three)
);
```

Data Storage and the Keys Roles

```
CREATE TABLE number0fRequests (
    cluster text,
    date text,
    datacenter text,
    hour int,
    minute int,
    number0fRequests int,
    PRIMARY KEY ((cluster, date), datacenter, hour, minute))
```

- ❖ cluster and date fields define the partition (node) where the row is stored
- ❖ Inside the partition, every row will be stored like this:

```
{datacenter: US_WEST_COAST {hour: 0 {minute: 0 {number0fRequests: 130}} {minute: 1 {number0fRequests: 125}} ...
    {minute: 59 {number0fRequests: 97}}}
    {hour: 1 {minute: 0 ...}}
```

Create Table Examples

```
CREATE TABLE postsbyuser (
    userid bigint,
    posttime timestamp,
    postid uuid,
    postcontent text,
    PRIMARY KEY ((userid), posttime)
) WITH CLUSTERING ORDER BY (posttime DESC);
```

```
CREATE TABLE timeline (
    userid uuid,
    posted_month int,
    posted_time uuid,
    body text,
    posted_by text,
    PRIMARY KEY (userid, posted_month, posted_time)
) WITH compaction = { 'class' : 'LeveledCompactionStrategy' };
```

```
CREATE TABLE movies (
    id TEXT,
    title TEXT,
    director TUPLE<TEXT, TEXT>,
    year SMALLINT,
    actors MAP<TEXT, TEXT>,
    genres LIST<TEXT>,
    countries SET<TEXT>,
    PRIMARY KEY (id)
)
```

Table – Other Statements

❖ DROP TABLE

`DROP TABLE [IF EXISTS] table_name`

❖ TRUNCATE TABLE

- preserves a table but removes all data it contains

`TRUNCATE [TABLE] table_name`

❖ ALTER TABLE

`ALTER [TABLE] table_name alter_table_instruction`

`alter_table_instruction ::= ADD column_name cql_type (',' column_name cql_type)* | DROP column_name (column_name)* | WITH options`

```
ALTER TABLE addamsFamily ADD gravesite varchar;
```

Select Statement

- ❖ Reads one or more columns for 1+ rows in a table

```
SELECT [ JSON | DISTINCT ] ( select_clause | '*' )
FROM table_name
[ WHERE where_clause ]
[ GROUP BY group_by_clause ]
[ ORDER BY ordering_clause ]
[ PER PARTITION LIMIT (integer | bind_marker) ]
[ LIMIT (integer | bind_marker) ]
[ ALLOW FILTERING ]
```

- ❖ Clauses

SELECT – columns or values to appear in the result

FROM – single table to be queried

WHERE – filtering conditions to be applied on table rows

GROUP BY – columns used for grouping of rows

ORDER BY – criteria defining the order of rows in the result

LIMIT – number of rows to be included in the result

Select – FROM Clause

- ❖ Defines a **single table** to be queried
 - from the current / specified keyspace
 - joining of multiple tables is not possible
- ❖ Supports:
 - **distinct** to remove duplicate rows
 - (user-defined) **aggregate functions**
 - * to select all columns; and attributes **alias (AS)**
 - WRITETIME (**timestamp**) and TTL (**time-to-live**) of a column
 - cannot be used in WHERE clause

```
SELECT name, occupation FROM users WHERE userid IN (199, 200, 207);
SELECT JSON name, occupation FROM users WHERE userid = 199;
SELECT name AS user_name, occupation AS user_occupation FROM users;
```

```
SELECT time, value
FROM events
WHERE event_type = 'myEvent'
  AND time > '2011-02-03'
  AND time <= '2012-01-01'
```

```
SELECT COUNT (*) AS user_count FROM users;
```

videoname	ttl(videoname)	writetime(videoname)
Ondas gigantes na barra!	null	1509294890781000
Aviões de papel!	null	1509294888607000

Select – WHERE Clause

- ❖ **Similar syntaxes:** CQL and SQL
- ❖ **Several differences** due the fact that Cassandra is dealing with distributed data and aims to prevent inefficient queries
 - rows are spread around the cluster based on the hash of the partition keys
 - clustering key columns are used to cluster the data of a partition, allowing a very efficient retrieval of rows
- ❖ Partition key, clustering and normal columns support different sets of restrictions within the WHERE clause

Select – WHERE Clause

❖ Partition key columns support only = and IN

- all primary key columns must be used (restricted), unless secondary index exist
 - all columns are needed to compute the hash that will allow it to locate the nodes containing the partition

❖ Clustering columns supports:

- comparisons =, <, <=, =>, >
- IN, returns true if the actual value is one of the enumerated
- CONTAINS* and CONTAINS KEY**
 - used on collections* (lists, sets, and maps) / maps**
 - returns true if a collection contains a given element, or, when the query is using a secondary index

Select – WHERE: Examples 1

```
CREATE TABLE numberOfRequests (
    cluster text,
    date text,
    datacenter text,
    hour int,
    minute int,
    numberOfRequests int,
    PRIMARY KEY ((cluster, date), datacenter, hour, minute))
```

```
/* Data will be stored like this:
{datacenter: US_WEST_COAST {hour: 0 {minute: 0 {numberOfRequests: 130}} {minute: 1 {numberOfRequests: 125}} ...
    {minute: 59 {numberOfRequests: 97}}}
    {hour: 1 {minute: 0 ...}}
```

```
*/
```

```
SELECT * FROM numberOfRequests
WHERE cluster = 'cluster1'
AND datacenter = 'US_WEST_COAST'
AND hour = 14
AND minute = 00;
```



```
SELECT * FROM numberOfRequests
WHERE cluster = 'cluster1'
AND date = '2015-06-05'
AND datacenter = 'US_WEST_COAST'
AND hour = 14
AND minute = 00;
```



```
SELECT * FROM numberOfRequests
WHERE cluster = 'cluster1'
AND date = '2015-06-05'
AND hour = 14
AND minute = 00;
```



```
SELECT * FROM numberOfRequests
WHERE cluster = 'cluster1'
AND date = '2015-06-05'
AND datacenter = 'US_WEST_COAST'
AND hour IN (14, 15)
AND minute = 0;
```



```
SELECT * FROM numberOfRequests
WHERE cluster = 'cluster1'
AND date = '2015-06-05'
AND datacenter = 'US_WEST_COAST'
AND (hour, minute) IN ((14, 0), (15, 0));
```

-- multi-column IN restrictions can be applied to any set of clustering columns.

```
SELECT * FROM numberOfRequests
WHERE cluster = 'cluster1'
AND date = '2015-06-05'
AND (datacenter, hour) IN (('US_WEST_COAST', 14), ('US_EAST_COAST', 17))
AND minute = 0;
```

Select – WHERE: Examples 2

, >, >=, <= and < restrictions

Single column slice restrictions are allowed only on the last clustering column being restricted.
Multi-column slice restrictions are allowed on the last set of clustering columns being restricted.

```
-- OK
SELECT * FROM number0fRequests
  WHERE cluster = 'cluster1'
    AND date = '2015-06-05'
    AND datacenter = 'US_WEST_COAST'
    AND hour= 12
    AND minute >= 0 AND minute <= 30;
```

```
-- OK
SELECT * FROM number0fRequests
  WHERE cluster = 'cluster1'
    AND date = '2015-06-05'
    AND datacenter = 'US_WEST_COAST'
    AND hour >= 12;
```

```
-- OK
SELECT * FROM number0fRequests
  WHERE cluster = 'cluster1'
    AND date = '2015-06-05'
    AND datacenter > 'US';
```

```
-- NOK
SELECT * FROM number0fRequests
  WHERE cluster = 'cluster1'
    AND date = '2015-06-05'
    AND datacenter = 'US_WEST_COAST'
    AND hour >= 12 AND minute = 0;
```

```
CREATE TABLE number0fRequests (
  cluster text,
  date text,
  datacenter text,
  hour int,
  minute int,
  number0fRequests int,
  PRIMARY KEY ((cluster, date), datacenter, hour, minute))
```

```
-- OK
SELECT * FROM number0fRequests
  WHERE cluster = 'cluster1'
    AND date = '2015-06-05'
    AND datacenter = 'US_WEST_COAST'
    AND (hour, minute) >= (12, 0) AND (hour, minute) <= (14, 0)
```

```
-- OK
SELECT * FROM number0fRequests
  WHERE cluster = 'cluster1'
    AND date = '2015-06-05'
    AND datacenter = 'US_WEST_COAST'
    AND (hour, minute) >= (12, 30) AND (hour) <= (14)
```

```
-- NOK: the restrictions must start with the same column
SELECT * FROM number0fRequests
  WHERE cluster = 'cluster1'
    AND date = '2015-06-05'
    AND datacenter = 'US_WEST_COAST'
    AND (hour, minute) >= (12, 30) AND (minute) <= (30)
```

Select – WHERE: Secondary Index

- ❖ Direct queries on secondary indices support only =, **CONTAINS** or **CONTAINS KEY** restrictions

```
CREATE TABLE contacts (
    id int PRIMARY KEY,
    firstName text,
    lastName text,
    phones map<text, text>,
    emails set<text>
);
```

```
select * from contacts
where firstname = 'Maria'; X
```



```
/*
   Solution: Secondary Index
*/
CREATE INDEX ON contacts (firstName);
-- Using the keys function to index the map keys
CREATE INDEX ON contacts (keys(phones));
CREATE INDEX ON contacts (emails);
```



```
SELECT * FROM contacts WHERE firstname = 'Benjamin';
SELECT * FROM contacts WHERE phones CONTAINS KEY 'office';
SELECT * FROM contacts WHERE emails CONTAINS 'Benjamin@oops.com'; V
```

Select – GROUP BY, ORDER BY and LIMIT

❖ GROUP BY clause

- groups rows of a table according to certain columns
- only groupings induced by primary key columns are allowed
- when a non-grouping column is selected without an aggregate function, the first value encountered is always returned

❖ ORDER BY clause

- defines the order (ASC or DESC) of returned rows
- partition key must be restricted (= or IN)
- only orderings induced by clustering columns are allowed!

❖ LIMIT clause

- limits the number of rows returned in the query result

Select – Group and Order By Examples

```
CREATE TABLE contacts (
    type text,
    id int,
    firstName text,
    lastName text,
    phones map<text, text>,
    emails set<text>,
    PRIMARY KEY(type, id)
);
```

```
select * from contacts
where id = 33
order by id;
```

```
select * from contacts
where type = 'personal'
and id = 33
order by id;
```

```
select * from contacts
where type = 'personal'
group by type;
```

```
select * from contacts
where type = 'personal'
group by id;
```

```
select count(*) from contacts
group by type;
```

```
select count(*) from contacts
group by id;
```

only the first
record is returned

Select – User Defined Functions

❖ User-Defined Functions (UDF)

- allow the execution of user-provided code (Java or JavaScript)
- Statements: CREATE (or REPLACE) /DROP FUNCTION

```
CREATE FUNCTION IF NOT EXISTS akeyspace.fname(someArg int)
    CALLED ON NULL INPUT
    RETURNS text
    LANGUAGE java
    AS $$
        // some Java code
    $$;
```

```
CREATE FUNCTION IF NOT EXISTS div (n counter, d counter)
    CALLED ON NULL INPUT
    RETURNS double
    LANGUAGE java AS '
        return Double.valueOf(n/d);
    ';

select rating_counter, div(rating_total, rating_counter)
from ...
```

Select – Aggregates

❖ Native

- COUNT(column), MIN(column), MAX(column), SUM(column) and AVG(column)

❖ User-Defined Aggregate Function (UDA)

- creation of custom aggregate functions

```
CREATE TABLE team_average (
    team_name text,
    cyclist_name text,
    cyclist_time_sec int,
    race_title text,
    PRIMARY KEY (team_name, race_title, cyclist_name)
);
```

1

```
-- UDA: calculate the average
--      value in the column
CREATE AGGREGATE average(int)
SFUNC avgState
STYPE tuple<int,bigint>
FINALFUNC avgFinal
INITCOND (0,0);
```

4

```
-- Test the function using a select statement
SELECT average(cyclist_time_sec) FROM team_average
WHERE team_name='UnitedHealthCare'
AND race_title='Amgen Tour';
```

5

```
-- UDF: adds all the race times together and counts the number of entries.
CREATE OR REPLACE FUNCTION avgState ( state tuple<int,bigint>, val int )
CALLED ON NULL INPUT
RETURNS tuple<int,bigint>
LANGUAGE java AS
$$ if (val !=null) {
    state.setInt(0, state.getInt(0)+1);
    state.setLong(1, state.getLong(1)+val.intValue());
}
return state; $$
```

2

```
-- UDF: computes the average of the values passed to it from the state function
CREATE OR REPLACE FUNCTION avgFinal ( state tuple<int,bigint> )
CALLED ON NULL INPUT
RETURNS double
LANGUAGE java AS
$$ double r = 0;
if (state.getInt(0) == 0) return null;
r = state.getLong(1);
r/= state.getInt(0);
return Double.valueOf(r); $$
```

3

Select – ALLOW FILTERING

- ❖ Option used to explicitly allow (some) queries that require filtering
- ❖ By default, only non-filtering queries are allowed
 - i.e. queries where the number of rows read ~ the number of rows returned
 - such queries have predictable performance
 - execution time that is proportional to the amount of data returned

```
CREATE TABLE users (
    username text PRIMARY KEY,
    firstname text,
    lastname text,
    birth_year int,
    country text
)
CREATE INDEX ON users(birth_year);
```

```
SELECT * FROM users;
SELECT * FROM users WHERE birth_year = 1981;
```

```
SELECT * FROM users WHERE birth_year = 1981 AND country = 'FR';
```

```
SELECT * FROM users WHERE birth_year = 1981 AND country = 'FR' ALLOW FILTERING;
```

Insert Statement

- ❖ Inserts a new row into a given table
 - if the primary key already exists, the row is updated
 - IF NOT EXISTS condition to only insert if row does not exist
- ❖ Writes one or more columns for a given row
- ❖ At least primary key columns must be specified

```
INSERT INTO NerdMovies (movie, director, main_actor, year)
    VALUES ('Serenity', 'Joss Whedon', 'Nathan Fillion', 2005)
    USING TTL 86400;
```

```
INSERT INTO NerdMovies JSON '{"movie": "Serenity",
    "director": "Joss Whedon",
    "year": 2005}';
```

```
CREATE TABLE movies (
    id TEXT,
    title TEXT,
    director TUPLE<TEXT, TEXT>,
    year SMALLINT,
    actors MAP<TEXT, TEXT>,
    genres LIST<TEXT>,
    countries SET<TEXT>,
    PRIMARY KEY (id)
)
```

```
INSERT INTO movies (id, title, director, year, actors, genres)
VALUES (
    'stesti',
    'Štěsti',
    ('Bohdan', 'Sláma'),
    2005,
    { 'vilhelmova': 'Monika', 'liska': 'Toník' },
    [ 'comedy', 'drama' ]
)
USING TTL 86400
```

Update Statement

- ❖ Updates existing rows within a given table
 - when a row with a given primary key does not yet exist, it is inserted
- ❖ All primary key columns must be specified in the WHERE clause

```
UPDATE NerdMovies USING TTL 400
    SET director = 'Joss Whedon',
        main_actor = 'Nathan Fillion',
        year       = 2005
    WHERE movie = 'Serenity';

UPDATE UserActions
    SET total = total + 2
    WHERE user = B70DE1D0-9908-4AE3-BE34-5573E5B09F14
        AND action = 'click';
```

Update Statement – More Examples

```
UPDATE movies
SET
  year = 2006,
  director = ('Jan', 'Svěrák'),
  actors = { 'machacek': 'Robert Landa', 'sverak': 'Josef Tkaloun' },
  genres = [ 'comedy' ],
  countries = { 'CZ' }
WHERE id = 'vratnelahve'
```

```
UPDATE movies
SET
  actors = actors + { 'vilhelmova': 'Helenka' },
  genres = [ 'drama' ] + genres,
  countries = countries + { 'SK' }
WHERE id = 'vratnelahve'
```

```
UPDATE movies
SET
  actors['vilhelmova'] = 'Helenka',
  genres[1] = 'comedy'
WHERE id = 'vratnelahve'
```

```
CREATE TABLE movies (
  id TEXT,
  title TEXT,
  director TUPLE<TEXT, TEXT>,
  year SMALLINT,
  actors MAP<TEXT, TEXT>,
  genres LIST<TEXT>,
  countries SET<TEXT>,
  PRIMARY KEY (id)
)
```

Insert and Update Parameters

❖ TTL: time-to-live

- 0 or null or simply missing for persistent values
- if set, the inserted values are automatically removed from the database after the specified time.

❖ TIMESTAMP: writetime

- only newly inserted / updated values are really affected
- if not specified, it will be used the current time (in microseconds)

```
UPDATE user USING TTL 3600 SET last_name = 'McDonald'  
WHERE first_name = 'Mary';  
  
SELECT first_name, last_name, TTL(last_name)  
FROM user WHERE first_name ='Mary';  
  
first_name | last_name | ttl(last_name)  
-----+-----+-----  
 Mary | McDonald | 3588
```

```
INSERT INTO cycling.comments (  
    id,  
    created_at)  
values (  
    123456,  
    toTimeStamp(now()));
```

Delete Statement

- ❖ Removes existent rows / columns / collection elements from a given table
- ❖ **WHERE** clause specifies which rows are to be deleted
- ❖ Multiple rows may be deleted with one statement by using an **IN** operator.
- ❖ A range of rows may be deleted using an inequality operator (such as \geq)

```
DELETE FROM NerdMovies USING TIMESTAMP 1240003134
WHERE movie = 'Serenity';
```

```
DELETE phone FROM Users
WHERE userid IN (C73DE1D3-AF08-40F3-B124-3FF3E5109F22, B70DE1D0-9908-4AE3-BE34-5573E5B09F14);
```

Java Driver

❖ Driver:

- Datastax
 - <https://github.com/datastax/java-driver>
 - com.datastax.driver.core
- required JARs
 - cassandra-driver-core-...
 - slf4j-api-...
 - guava-...
 - metrics-core-...
 - netty-all-...

❖ Documentation

- <http://docs.datastax.com/en/developer/java-driver>
- <http://docs.datastax.com/en/drivers/java/>
- https://www.tutorialspoint.com/cassandra/cassandra_installation.htm

Summary

- ❖ Column-oriented databases
- ❖ Google BigTable
- ❖ Apache Cassandra
 - Data Model
 - Cassandra query language
 - DDL statements
 - DML statements
 - CRUD operations
 - UDF
- ❖ Cassandra Java driver

Resources

- ❖ Martin Kleppmann, ***Designing Data-Intensive Applications***, O'Reilly Media, Inc., 2017.
- ❖ Eben Hewitt, Jeff Carpenter, ***Cassandra: The Definitive Guide***, 2nd Edition - Distributed Data at Web Scale, O'Reilly Media, Inc., 2016.
- ❖ Christof Strauch. ***NoSQL Databases***, 2009.
- ❖ Cassandra Tutorials/Documentation:
 - <https://www.tutorialspoint.com/cassandra/>
 - <https://www.datastax.com/resources/tutorials>
 - <https://teddyma.gitbooks.io/learn cassandra/>
 - <https://cassandra.apache.org/doc/latest/>
 - <http://docs.datastax.com/en/dse/5.1/cql/>
 - <https://cassandra.apache.org/doc/latest/cql/>