B4M36DS2, BE4M36DS2: **Database Systems 2**

https://cw.fel.cvut.cz/wiki/courses/b4m36ds2/

Lecture 9

Wide Column Stores: Cassandra

Yuliia Prokop
prokoyul@fel.cvut.cz

24. 11. 2025

Authors:
Martin Svoboda (martin.svoboda@matfyz.cuni.cz)
Yuliia Prokop

Czech Technical University in Prague, Faculty of Electrical Engineering

Lecture Outline

Wide column stores

- Introduction

Apache Cassandra

- Sharding, replication
- CAP theorem
- Data model
- Cassandra query language
  - DDL statements
  - DML statements

Wide Column Stores

Data model

- Column family
  - 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 **flat sets, lists or maps** may be allowed

Apache Cassandra

Apache Cassandra

Uber


Facebook


Spotify

Netflix

Column-family database

- http://cassandra.apache.org/
- Features
  - Open-source, high availability, linear scalability, sharding (spanning multiple datacenters), peer-to-peer configurable replication, tunable consistency, MapReduce support
- Developed by **Apache Software Foundation**
  - Originally at Facebook
- Implemented in Java
- Operating systems: cross-platform
- Initial release in 2008

Key Features and Architecture

- **Decentralized Peer-to-Peer Architecture**
  - All nodes are equal; no master-slave hierarchy.
- **High Availability and Fault Tolerance:**
  - Data automatically replicated across multiple nodes.
  - Ensures uptime during node or hardware failures.
- **Linear Horizontal Scalability:**
  - Seamless addition of nodes increases capacity and performance.
  - No downtime is required for scaling operations.
- **Gossip Protocol:**
  - Nodes communicate state information periodically.
  - Maintains cluster synchronization without a central coordinator.
- **Ring Topology:**
  - Logical arrangement of nodes for efficient data distribution.
  - Simplifies scaling and data partitioning.

Data Partitioning

COUNTRY	CITY	POPULATION
USA	New York	8.000.000
USA	Los Angeles	4.000.000
FR	Paris	2.230.000
DE	Berlin	3.350.000
UK	London	9.200.000
AU	Sydney	4.900.000
DE	Nuremberg	500.000
CA	Toronto	6.200.000
CA	Montreal	4.200.000
FR	Toulouse	1.100.000
JP	Tokyo	37.430.000
IN	Mumbai	20.200.000

Partition Key

COUNTRY	CITY	POPULATION
AU	Sydney	4.900.000
CA	Toronto	6.200.000
CA	Montreal	4.200.000
DE	Berlin	3.350.000
DE	Nuremberg	500.000

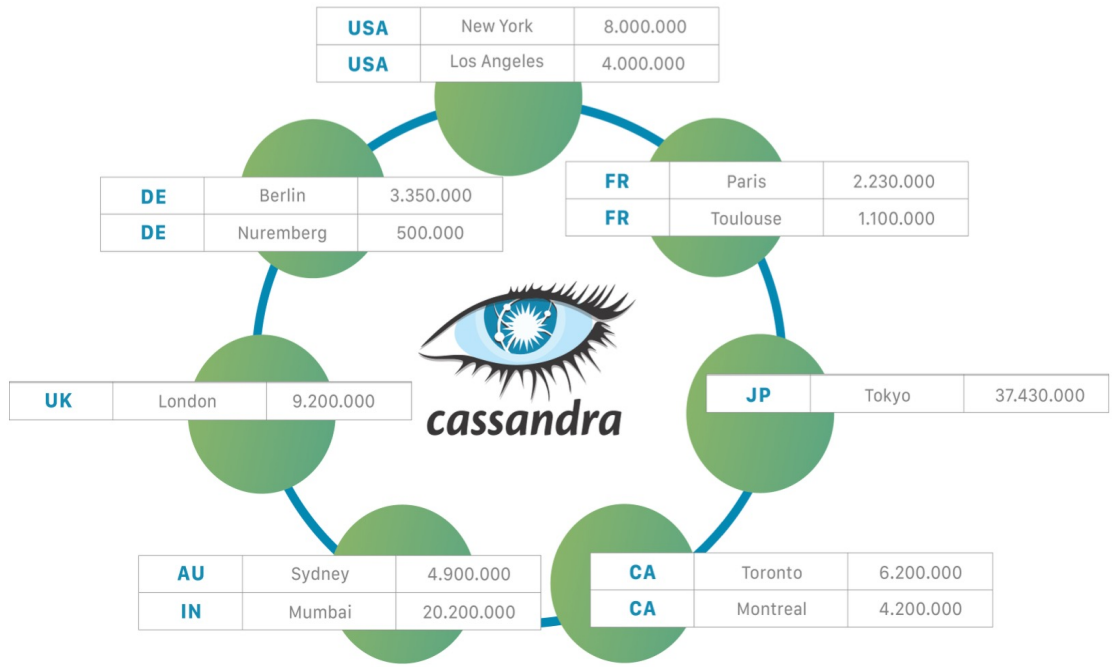
Partition Key

Partitioner

Hashing Function

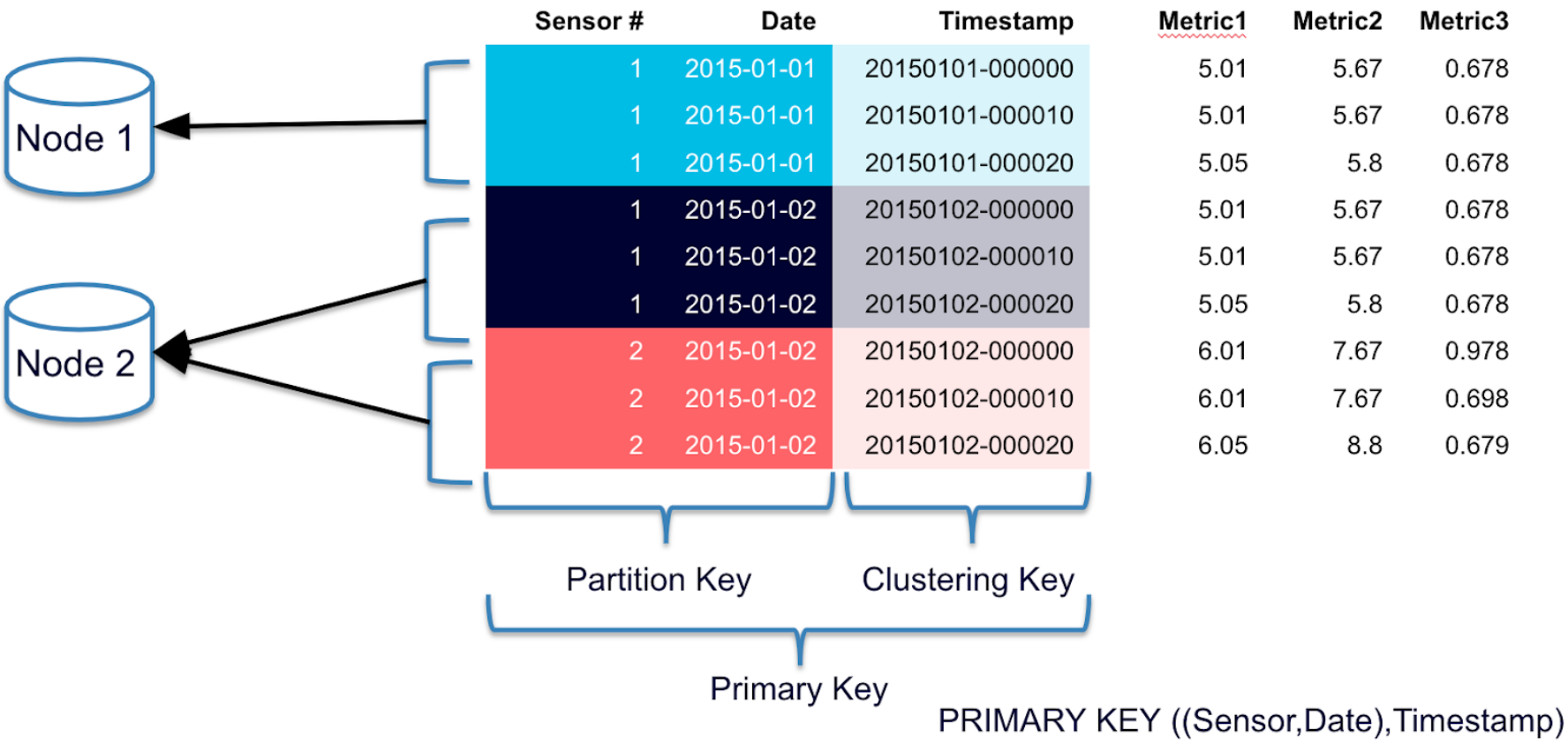
COUNTRY	CITY	POPULATION
59	Sydney	4.900.000
12	Toronto	6.200.000
12	Montreal	4.200.000
45	Berlin	3.350.000
45	Nuremberg	500.000

Tokens



Source https://cassandra.apache.org/_/cassandra-basics.html

Partition and Clustering Key



Sensor #	Date	Timestamp	Metric1	Metric2	Metric3
1	2015-01-01	20150101-000000	5.01	5.67	0.678
1	2015-01-01	20150101-000010	5.01	5.67	0.678
1	2015-01-01	20150101-000020	5.05	5.8	0.678
1	2015-01-02	20150102-000000	5.01	5.67	0.678
1	2015-01-02	20150102-000010	5.01	5.67	0.678
1	2015-01-02	20150102-000020	5.05	5.8	0.678
2	2015-01-02	20150102-000000	6.01	7.67	0.978
2	2015-01-02	20150102-000010	6.01	7.67	0.698
2	2015-01-02	20150102-000020	6.05	8.8	0.679

PRIMARY KEY ((Sensor,Date),Timestamp)

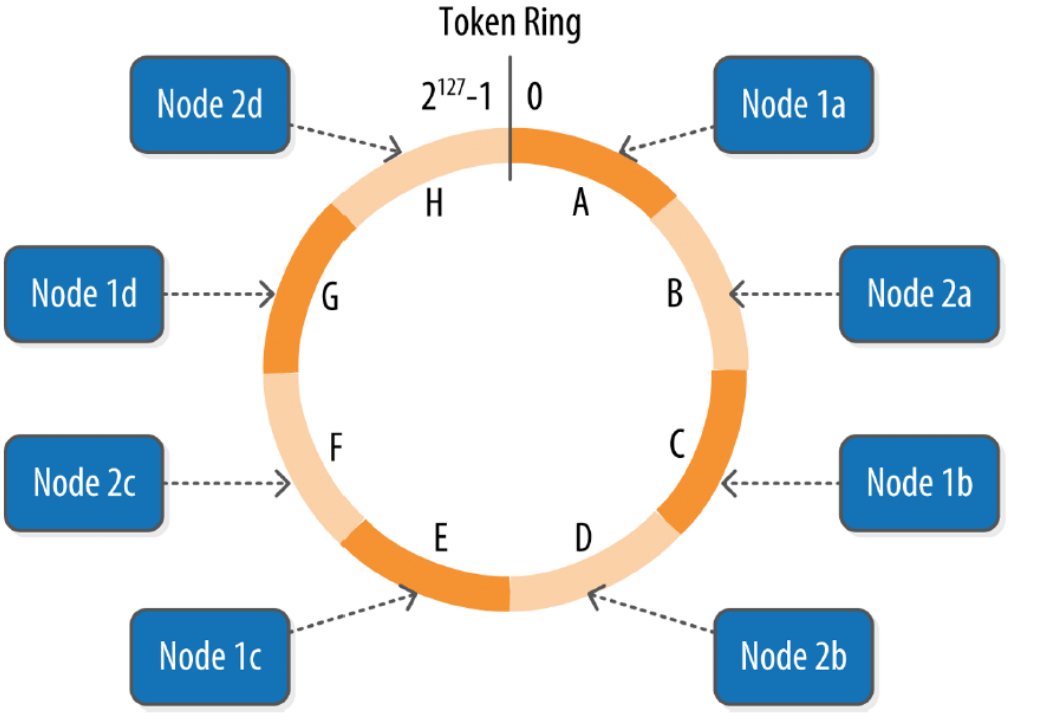
The Primary Key consists of two parts:

- **Partition Key**: (Sensor, Date) - defines which node will store the data
- **Clustering Key**: Timestamp - defines how data is sorted/organized within each partition

Source https://www.instaclustr.com/blog/cassandra-data-partitioning/

Data Partitioning

- **Data Partitioning (Sharding):**
  - **Consistent Hashing:** Distributes data based on partition key hashes.
  - **Virtual Nodes (vnodes):** Each physical node handles multiple token ranges (256 virtual nodes by default).
  - **Partition Key Design:** Ensures even data distribution and prevents hotspots.



Source: https://www.digihunch.com/2018/03/cassandra-architecture-summary/

Replication

- **Replication Strategies:**
  - **Replication Factor (RF):** Number of replicas for each piece of data.
  - **SimpleStrategy:**
    Suitable for single data center deployments.
    Replicates data to adjacent nodes in the ring.
  - **NetworkTopologyStrategy:**
    Ideal for multi-data center clusters.
    Configures replication per data center.

- **Replica Placement:**
  - Distributes replicas across different racks and data centers.
  - Enhances fault tolerance and data availability.

- **Impact on Performance and Availability:**
  - Higher RF increases data redundancy and fault tolerance.
  - Balances storage costs with reliability and access speed.

Consistency, Synchronization, and CAP Theorem

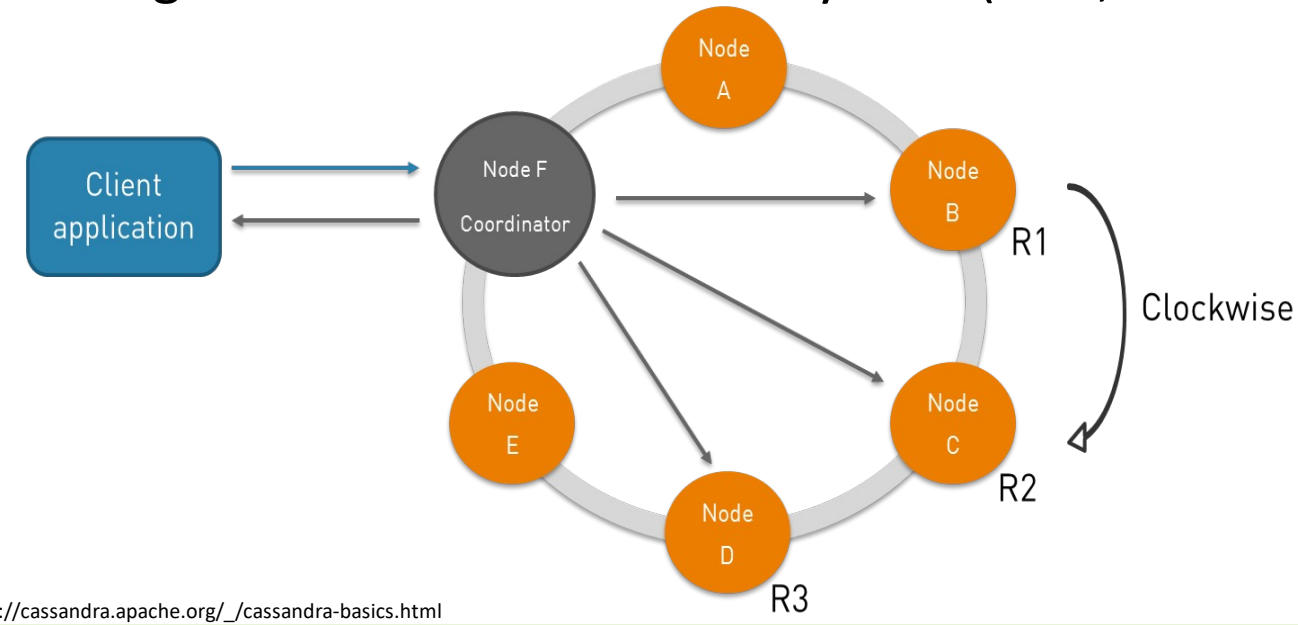
- **Tunable Consistency Levels:**
  - **Levels Available:** ANY, ONE, QUORUM, LOCAL_QUORUM, EACH_QUORUM, ALL.
  - **Per-Operation Configuration:** Set consistency levels for individual reads and writes.
- **Consistency and Synchronization Mechanisms:**
  - **Read Repair:** Corrects inconsistencies during read operations.
    - **Synchronous:** Repairs during the read.
    - **Asynchronous:** Repairs in the background.
  - **Hinted Handoff:** Temporarily stores writes for unreachable nodes.
    - Replays hints when nodes recover.
- **Anti-Entropy Repair:**
  - **Full Repair:** Comprehensive data synchronization.
  - **Incremental Repair:** Targets recently changed data.

Consistency, Synchronization, and CAP Theorem

- **Clock Synchronization and Conflict Resolution:**
  - Uses timestamps (microsecond precision) to resolve write conflicts.
  - **Last Write Wins:** Most recent write overwrites previous ones.
- **CAP Theorem Positioning:**
  - **AP System:** Prioritizes Availability and Partition Tolerance.
  - **Tunable Consistency:** Allows configurations to approach CP as needed.
- **Trade-offs and Implications:**
  - **Higher Consistency Levels:**
    - Increased data accuracy.
    - Potentially higher latency and reduced availability.
  - **Lower Consistency Levels:**
    - Enhanced performance and availability.
    - Risk of reading stale or inconsistent data.

Cassandra Request Handling Flow

- Client sends a **write** request to Coordinator (Node F)
- Coordinator calculates token positions from its token map
- Based on token, Coordinator sends write request to all three replica nodes (R1-R3)
- Coordinator waits for the answer from one, two or three replicas depending on the Tunable Consistency level (ONE, QUORUM, ALL)
- Data is replicated clockwise: B(R1), C(R2), D(R3)
- For **read** request, Coordinator queries one, two or three replica nodes (R1-R3) depending on the Tunable Consistency level (ONE, QUORUM, ALL)



Source https://cassandra.apache.org/_/cassandra-basics.html

Data Model

Database system structure

Instance → **keyspaces** → **tables** → **rows** → **columns**

- Keyspace
- Table (column family)
  - **Collection of (similar) rows**
    - Rows do not need to have exactly the same columns
  - Table schema must be specified, yet can be modified later on
- Row
  - **Collection of columns**
  - Each row is **uniquely identified** by a compulsory primary key
- Column
  - **Name-value pair** + additional data

Data Model

Column values

- Empty value
  - `null`
- Atomic values
  - **Native data types** such as text, integers, date, timestamp, …
  - **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 also permitted, however, currently only in a frozen mode

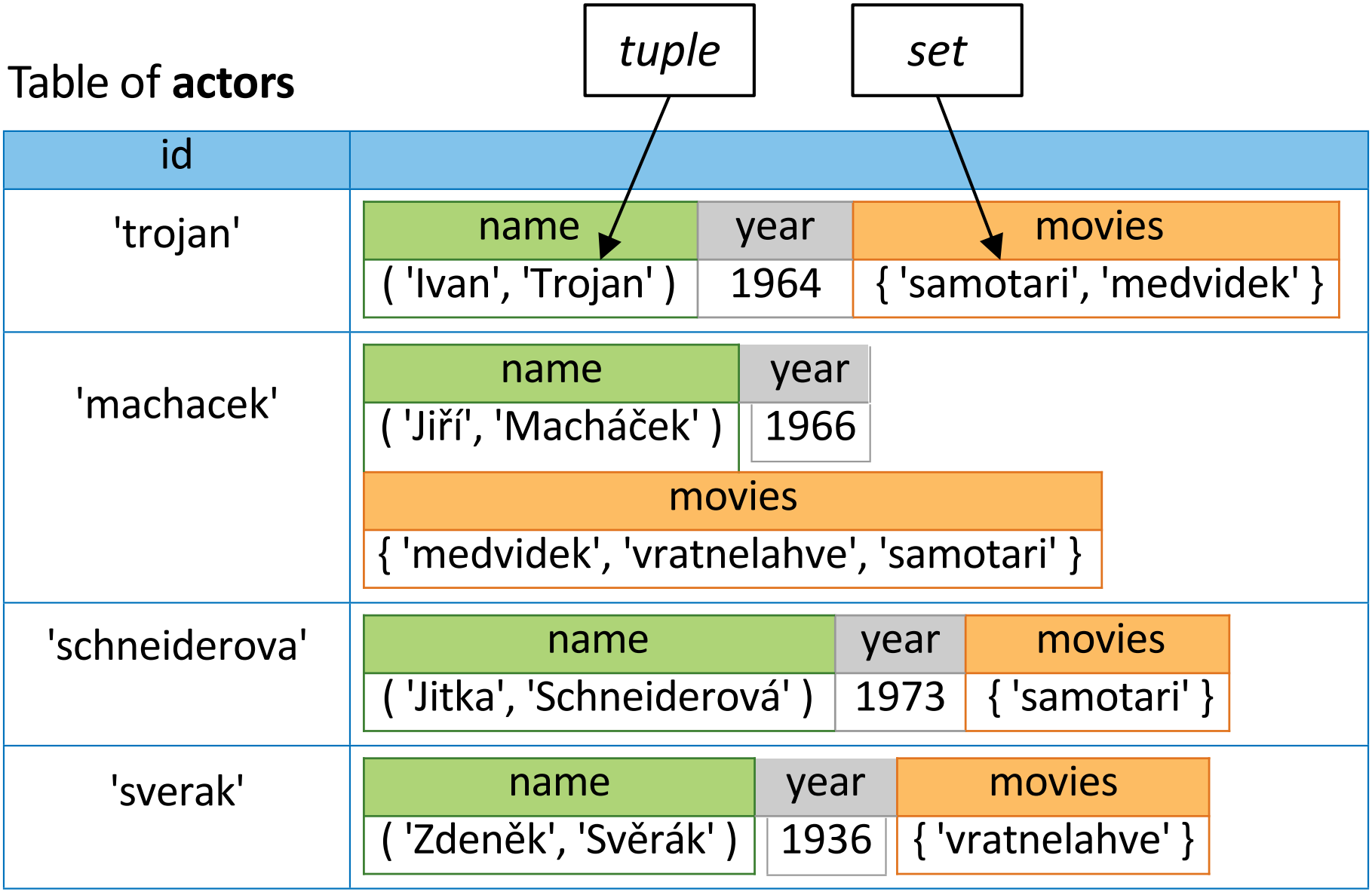
Data Model

Collections

- **List** = **ordered collection of values**
  - This order is based on positions
  - Values do not need to be unique
- **Set** = **collection of unique values**
  - Values are internally ordered based on hash values
- **Map** = **collection of key-value pairs**
  - Keys must be unique
  - Pairs are internally ordered based on keys

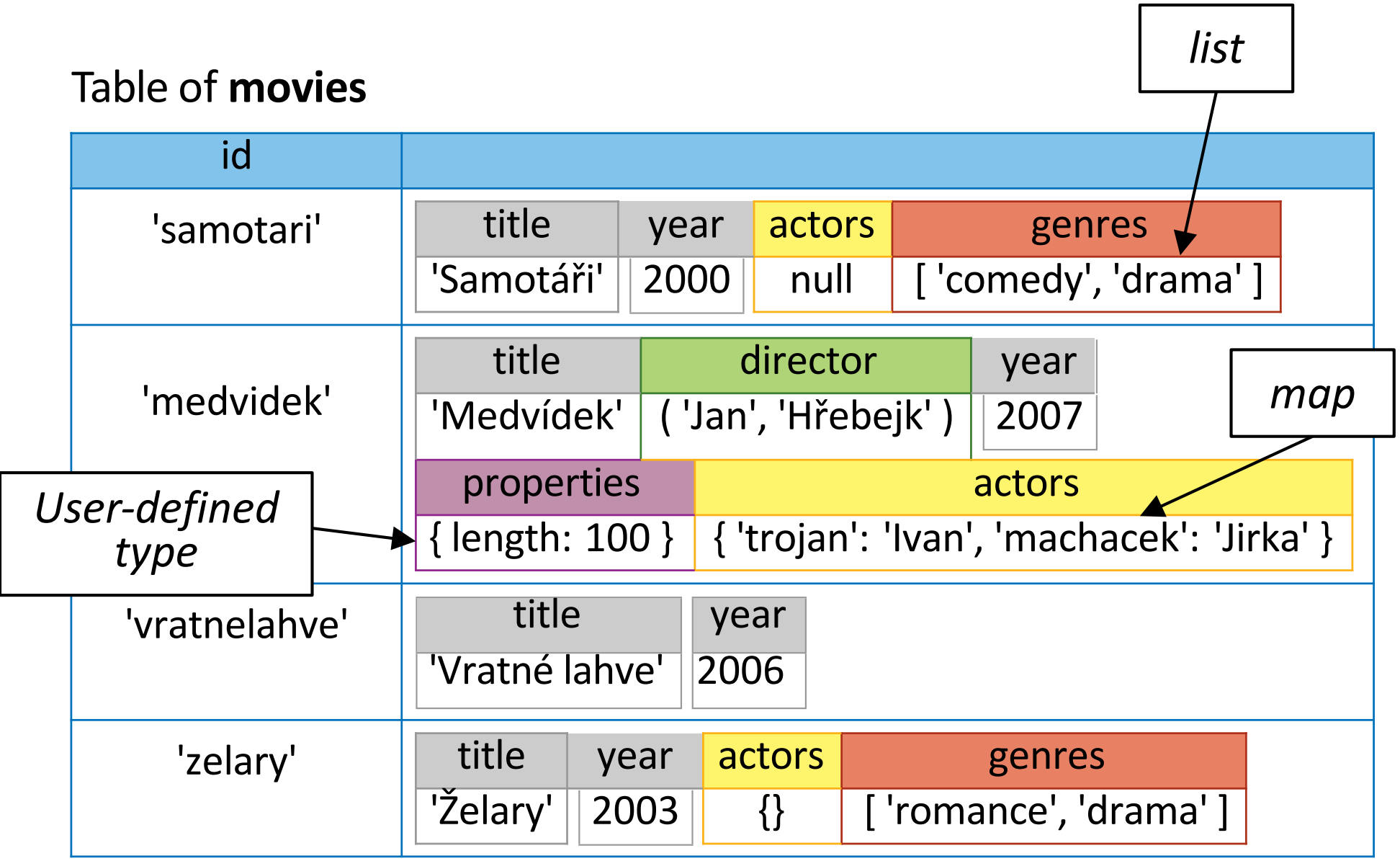
Sample Data

Table of **actors**



Sample Data

Table of **movies**



Data Model

Additional data associated with…

the whole column in case of atomic values, or
each individual element of a collection

- **Time-to-live** (TTL)
  - After a certain period of time (number of seconds) a given column / element is automatically deleted
- **Timestamp** (writetime)
  - Timestamp of the last modification
  - Assigned automatically or manually as well
- Both the records can be queried
  - Limited support for collections and their elements, depending on the way of collection storage (frozen/non-frozen)

Cassandra API

CQLSH

- **Interactive command line shell**
- `bin/cqlsh` (or via Docker container)
- Uses **CQL** (*Cassandra Query Language*)

Client drivers

- Officially supported by DataStax and community
- Available for various languages
  - Java, Python, Ruby, PHP, C++, Scala, Erlang, …

Query Language

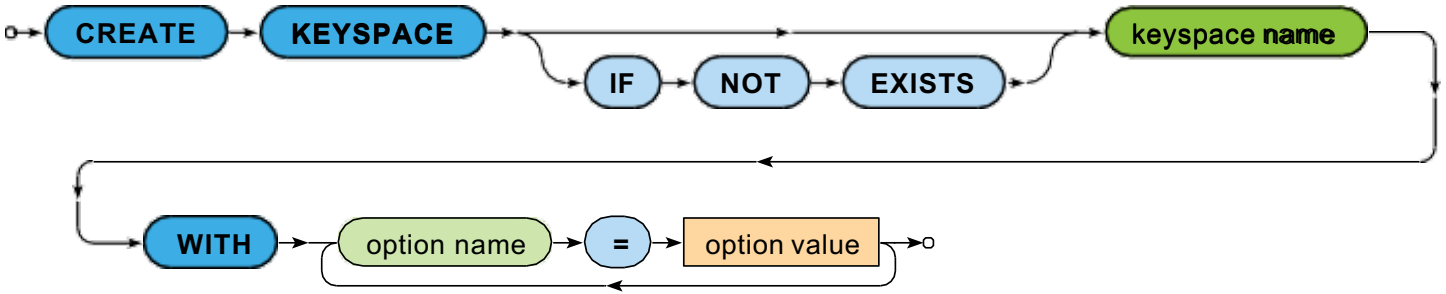
CQL = **Cassandra Query Language**

- Declarative query language
  - Inspired by SQL
- **DDL statements**
  - CREATE KEYSPACE – creates a new keyspace
  - CREATE TABLE – creates a new table
  - …
- **DML statements**
  - 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
  - …

DDL Statements

Keyspaces

CREATE KEYSPACE



- **Creates a new keyspace**
- **Replication option** is mandatory
  - SimpleStrategy

    (single data centre, development/testing only)
  - NetworkTopologyStrategy

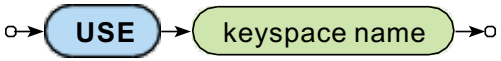
    (individual replication factor for each data center)

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

Keyspaces

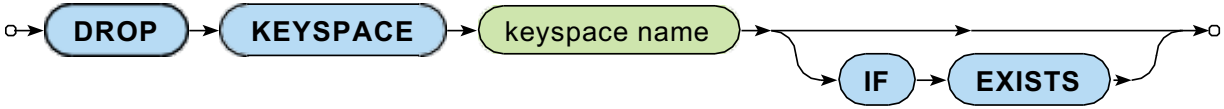
USE

- Changes the current keyspace



DROP KEYSPACE

- Removes a keyspace, all its tables, data etc.



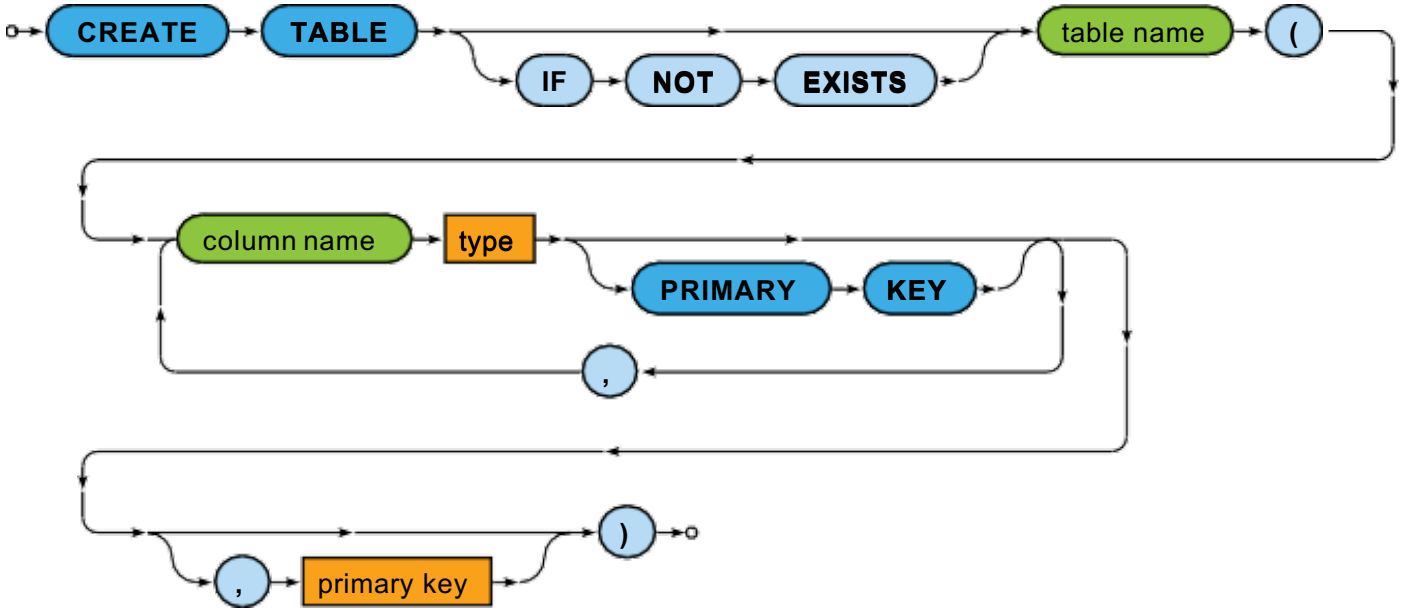
ALTER KEYSPACE

- Modifies options of an existing keyspace

Tables

CREATE TABLE

- **Creates a new table** within the current keyspace
- Each table must have exactly one **primary key** specified



- None of the columns is compulsory (except the primary key)
- The primary key consists of **partition** key and (optionally) **clustering** key

Tables

Examples: tables for **actors and movies**

```
CREATE TABLE IF NOT EXISTS actors  (
    id TEXT PRIMARY KEY,
    name TUPLE<TEXT, TEXT>,
    year SMALLINT,
    movies SET<TEXT>
)
```

```
CREATE TABLE IF NOT EXISTS movies (
    id TEXT,
    title TEXT,
    director TUPLE<TEXT, TEXT>,
    year SMALLINT,
    actors MAP<TEXT, TEXT>,
    genres LIST<TEXT>,
    countries SET<TEXT>,
    properties details,
    PRIMARY KEY (id)
)
```

Primary Keys

Primary keys have two parts

- Compulsory **partition key**
  - At least one column
  - Defines how individual rows are distributed between shards
- Optional **clustering columns**
  - Defines the order in which individual rows are locally stored by each shard

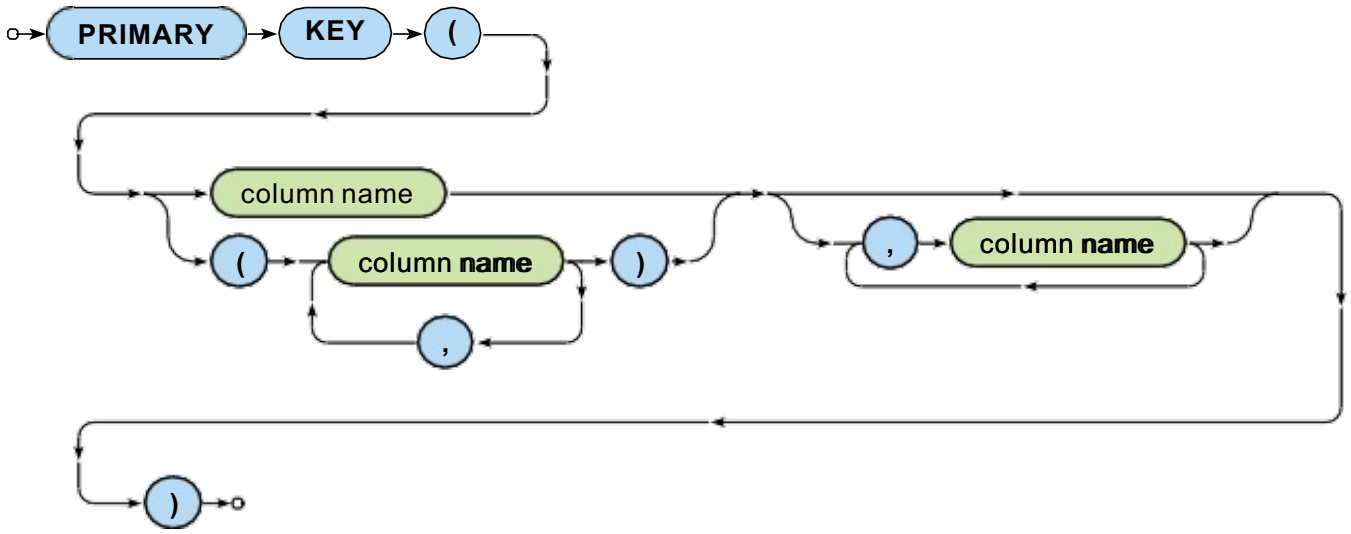
Primary Key Types:

- **Simple** primary key:
  - Can consist of a single column that becomes the partition key
  - May have no clustering columns
- **Composite** primary key:
  - Can have a composite partition key consisting of multiple columns
  - May include clustering columns

Primary Keys

Table-level primary key definition

- The first column / all columns in the embedded parentheses become the partition key
- All the remaining ones (if any) form the clustering columns

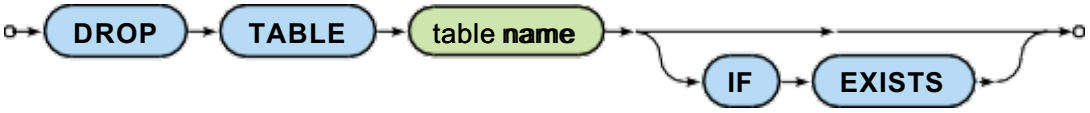



```
PRIMARY KEY(user_id, timestamp, action)
PRIMARY KEY((user_id, year), month, day)
```

Tables

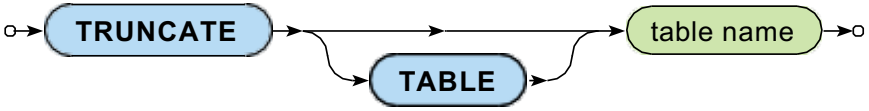
DROP TABLE

- Removes a table together with all data it contains



TRUNCATE TABLE

- Preserves a table but removes all data it contains



ALTER TABLE

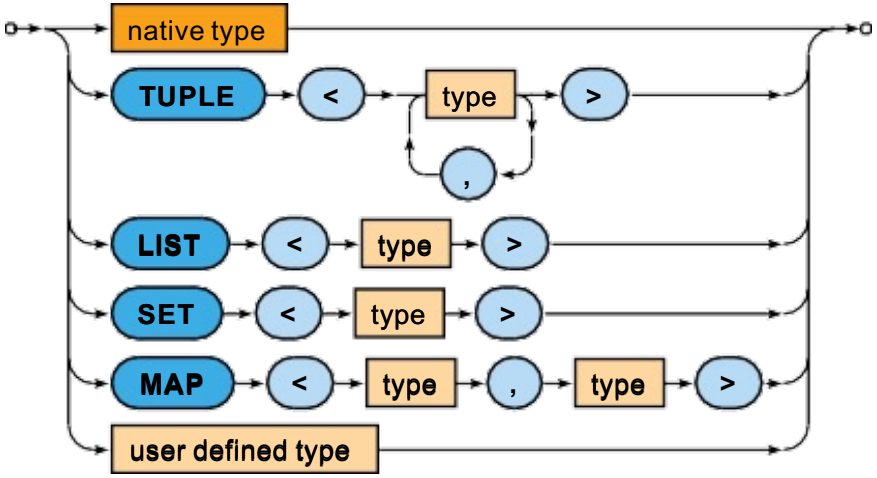
- Allows to alter, add or drop table columns, modify table properties and settings, manage secondary indexes,change caching option

It is recommended to add **IF EXISTS** to all these commands

Data Types

Types of columns

- Native types
- **Tuples**
- Collection types: **lists**, **sets**, and **maps**
- **User-defined types**



Native Data Types

Native types

- tinyint, smallint, **int**, bigint
  - Signed numbers (1B, 2B, 4B, 8B)
- **varint**
  - Arbitrary-precision integer
- **decimal**
  - Variable-precision decimal
- `float`, **double**
  - Floating point numbers (4B, 8B)
- **boolean**
  - Boolean values `true` and `false`

Native Data Types

Native types

- **`text`** (preferable), `varchar`
  - UTF8 encoded string (identical types)
  - Enclosed in **single** quotes (not double quotes)
    - Escaping sequence: ' '
- `ascii`
  - ASCII encoded string
- **date**, **time**, **timestamp**
  - Dates, times and timestamps
  - Supports timezone
  - E.g. '2016-12-05', '2016-12-05 09:15:00', '2016-12-05 09:15:00+0300'

Native Data Types

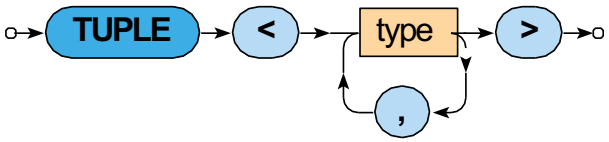
Native types

- **`counter`** – 8B signed integer
  - Only 2 operations supported: incrementing and decrementing
    - I.e. value of a counter cannot be set to a particular number
  - Restrictions in usage
    - Counters cannot be a part of a primary key
    - Either all table columns (outside the primary key) are counters, or none of them
    - TTL is not supported
    - Counters do not support secondary indexes
    - …
- `blob` – arbitrary bytes
- `inet` – IP address (both IPv4 and IPv6)
- …

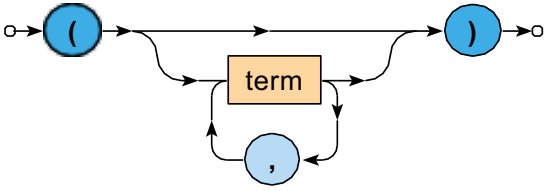
Tuple Data Types

Tuples

- Declaration



- Literals



  - E.g. ('Jiří', 'Macháček')

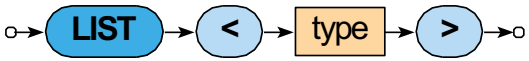
When working with non-ASCII characters in tuples:

- Ensure cluster uses UTF-8 encoding
- Verify client app handles Unicode properly
- Maintain consistent encoding across all system components

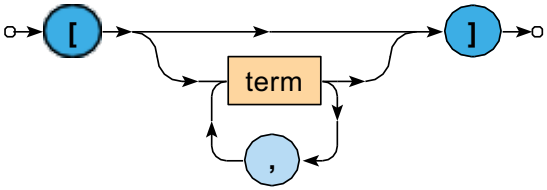
Collection Data Types

Lists

- Declaration



- Literals



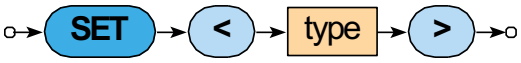
  - E.g. `[ 'comedy', 'drama' ]`

Note: Lists maintain order and allow duplicates.
Consider performance impact for large lists.

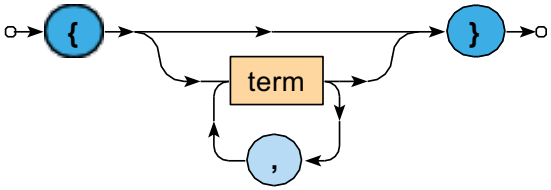
Collection Data Types

Sets

- Declaration



- Literals



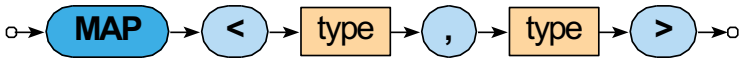
  - E.g. `{ 'medvidek', 'vratnelahve', 'samotari' }`

Note: Sets ensure unique values, unordered collection. Efficient for membership testing.

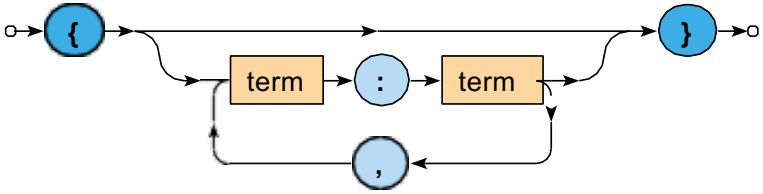
Collection Data Types

Maps

- Declaration



- Literals



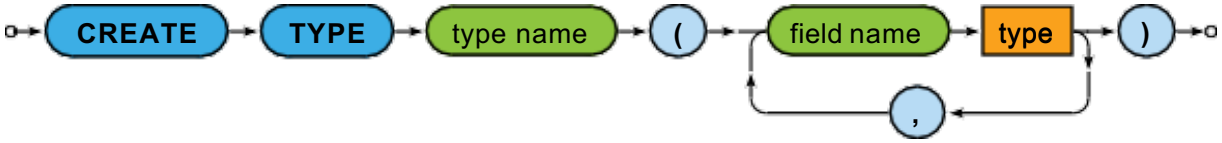
  - E.g. `{ 'machacek': 'Robert Landa' }`

Note: Maps are key-value pairs. Keys must be unique. Efficient for key lookups.

User-Defined Data Types

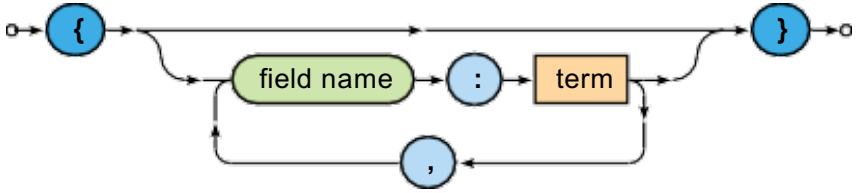
User-defined types (UDT)

- Definition



  - E.g. CREATE TYPE details (length SMALLINT, annotation TEXT)

- Literals



  - E.g. { length: 100 }

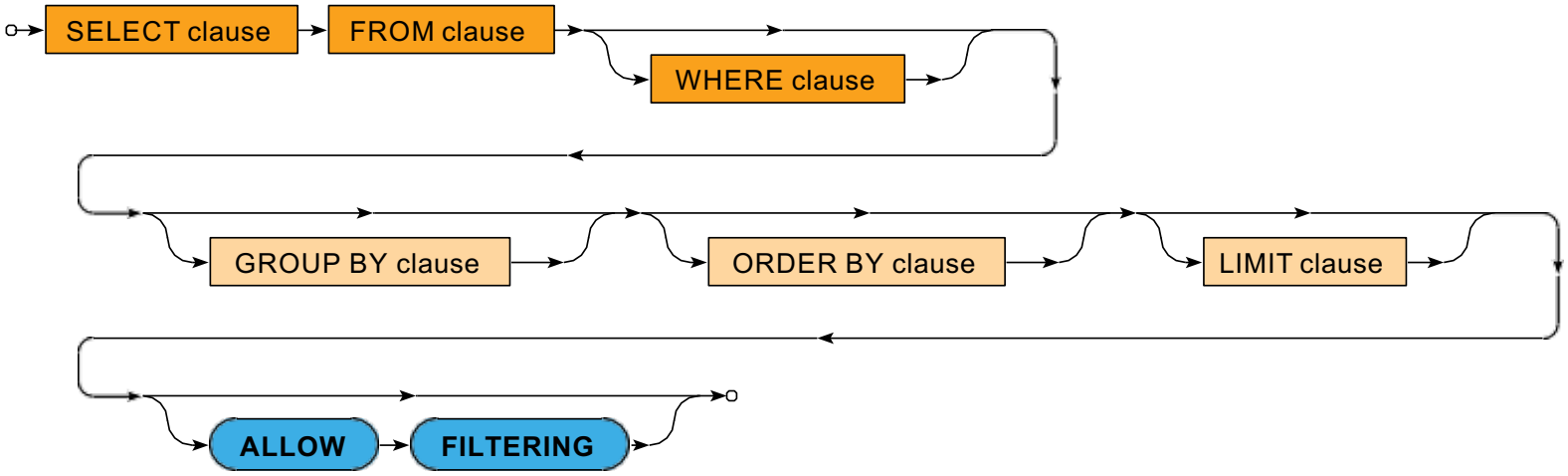
Note: UDTs allow creating complex, reusable data structures. Support nesting.

DML Statements

Selection

SELECT statement

- **Selects matching rows** from a **single** **table**



Notes:

WHERE clause **must** include the complete partition key and can include only clustering columns (in the order they are defined).
Other columns require ALLOW FILTERING (not recommended).
ALLOW FILTERING may significantly impact performance.

Selection

Clauses of SELECT statements

- `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 to be 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

Examples

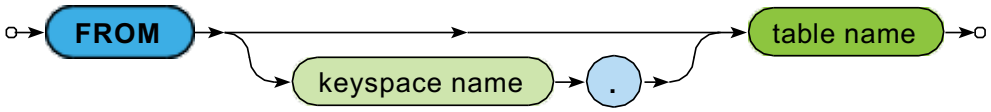
```
SELECT id, title, actors
FROM movies
WHERE id = 'medvidek'
```

```
SELECT id, title, actors
FROM movies
WHERE year = 2000 AND genres CONTAINS  'comedy'
```

Selection

FROM clause

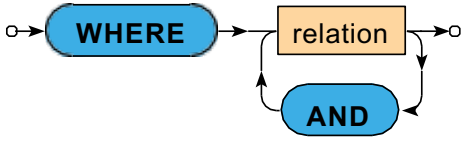
- Defines the **table to be queried**
  - From the current / selected keyspace
- Limited JOIN operations are supported:
  - Only within the same partition
  - Both tables must share the same partition key
  - Best practice is still data denormalization



Selection

WHERE clause

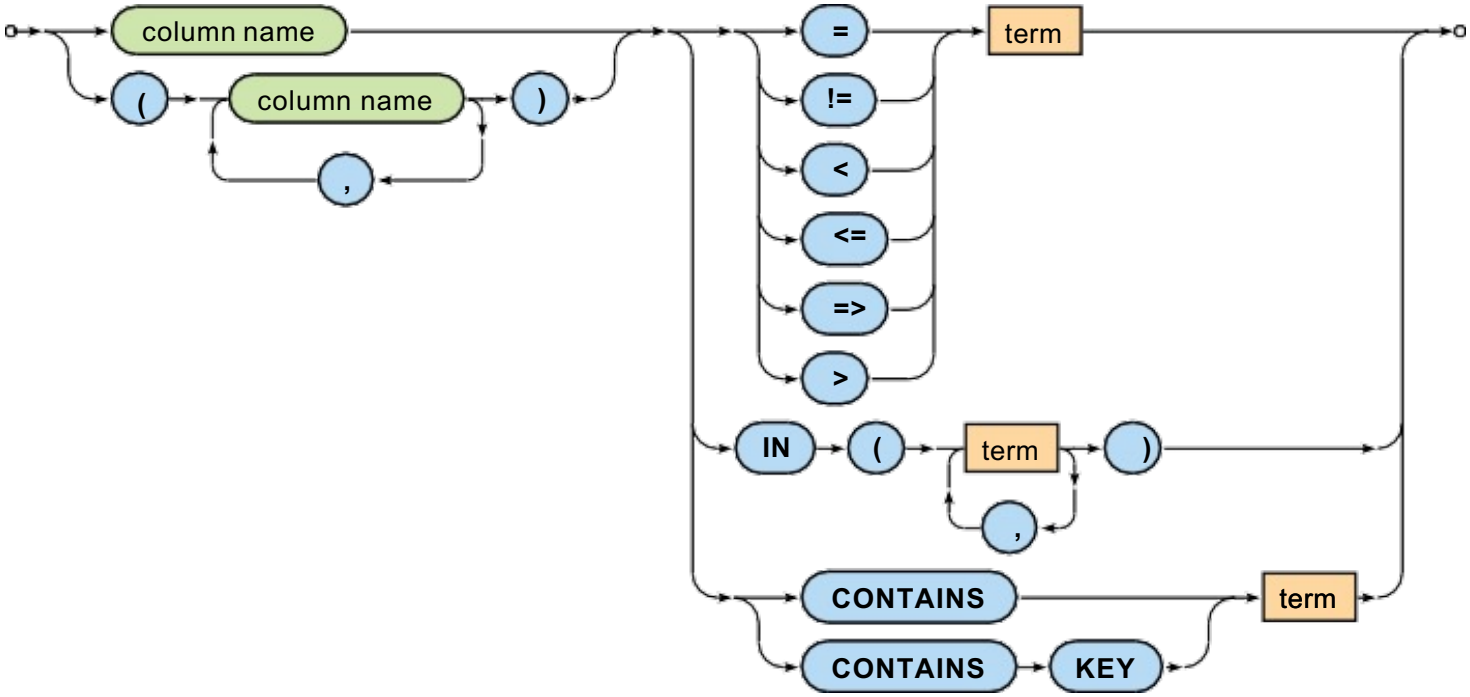
- **One or more relations a row must satisfy** in order to be included in the query result



- Conditions are limited:
  - Primary key columns can be queried directly
  - Non-key columns require secondary indexes
  - Partition key supports:
    - Equality (=)
    - IN operator
  - Clustering columns support:
    - Equality (=)
    - Inequality (<, >, <=, >=)
    - IN operator
  - ALLOW FILTERING enables non-indexed queries (not recommended)

Selection

WHERE clause: **relations**



Selection

WHERE clause: **relations**

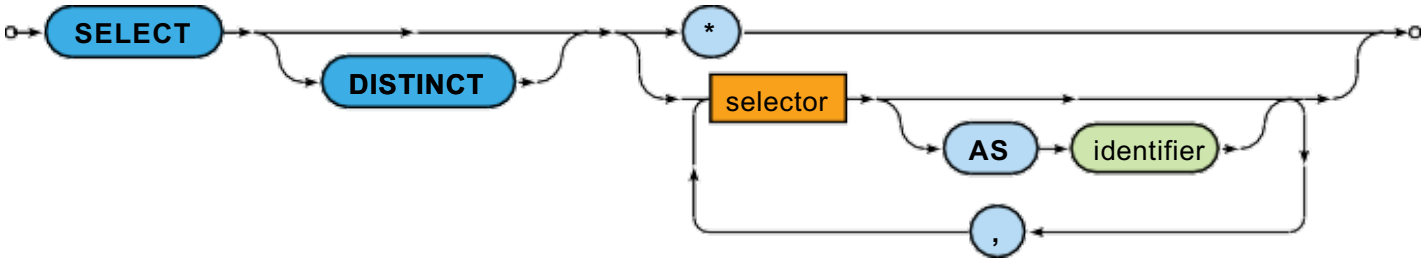
- **Comparisons**
  - =, !=, <, <=, =>, >
- **IN**
  - Returns true when the actual value is one of the enumerated
- **CONTAINS**
  - May only be used on collections (lists, sets, and maps)
  - Returns true when a collection contains a given element
- **CONTAINS KEY**
  - May only be used on maps
  - Returns true when a map contains a given key

Require secondary index for efficient queries

Selection

SELECT clause

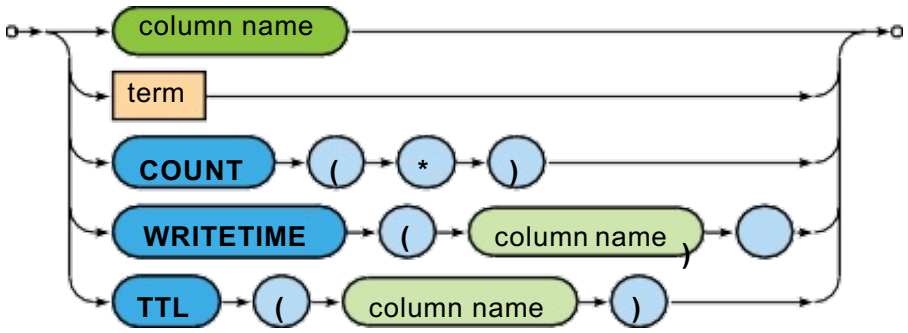
- Defines **columns or values to be included in the result**
  - * = all the table columns
  - Aliases can be defined using AS
  - Supports aggregate functions (COUNT, MIN, MAX, AVG, SUM)
  - Collection functions supported (LIST, SET, MAP operations)
  - Supports user-defined functions (UDFs)



- **DISTINCT** – duplicate rows are removed
  - DISTINCT operates only on partition key columns
  - Performance consideration: DISTINCT operations are memory-intensive

Selection

SELECT clause: **selectors**

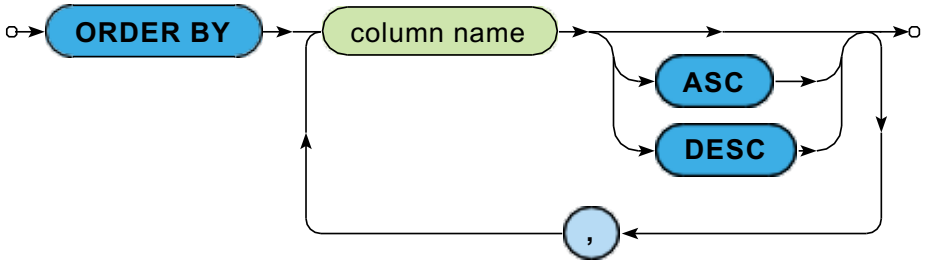


- **COUNT(*)**
  - Number of all the rows in a group (see aggregation)
  - Functions SUM, AVG, MIN, MAX are also available
- **WRITETIME** and **TTL**
  - Selects modification timestamp / remaining time-to-live of a given column
  - Cannot be used on collections and their elements
  - Cannot be used in other clauses (e.g. WHERE)
  - WRITETIME returns timestamp in microseconds since epoch
  - Can only be used on one column per query

Selection

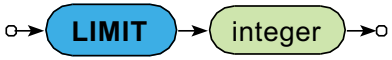
ORDER BY clause

- Defines the **order of rows returned in the query result**
- ORDER BY can only be used on clustering columns defined in table schema
- The order must match the clustering order defined in CREATE TABLE



LIMIT clause

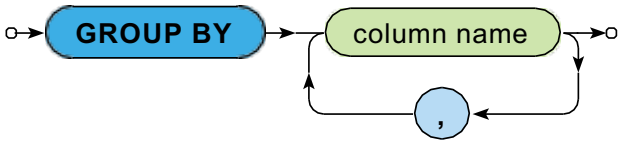
- **Limits the number of rows** returned in the query result



Selection

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 would be accessed directly** in the SELECT clause (i.e. without being wrapped by an aggregate function), the first value encountered will always be returned

Selection

GROUP BY clause: **aggregates**

- Native aggregates
  - **COUNT**(column)
    - Number of all the values in a given column
    - `null` values are ignored
  - **MIN**(column), **MAX**(column)
    - Minimal / maximal value in a given column
  - **SUM**(column)
    - Sum of all the values in a given column
  - **AVG**(column)
    - Average of all the values in a given column
- User-defined aggregates

Selection

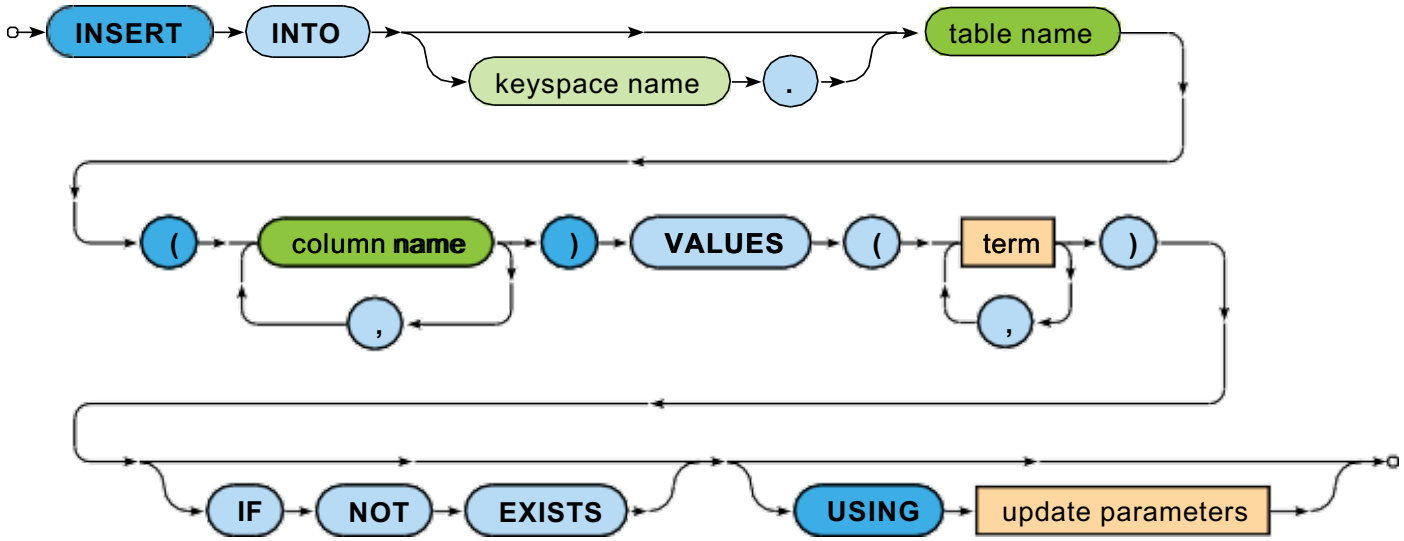
ALLOW FILTERING modifier

- 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
    - They will execute in a time that is proportional to the amount of data returned
- `ALLOW FILTERING` **enables (some) filtering queries**
  - May cause full table scan
  - Not recommended for production use
  - Better alternative: proper data modeling
- Best Practice
  - Use only on small tables or testing environments

Insertions

INSERT statement

- **Inserts a new row** into a given table
  - When a row with a given primary key already exists, it is updated
- Values of at least primary key columns must be set
- Names of columns must always be explicitly enumerated



Insertions

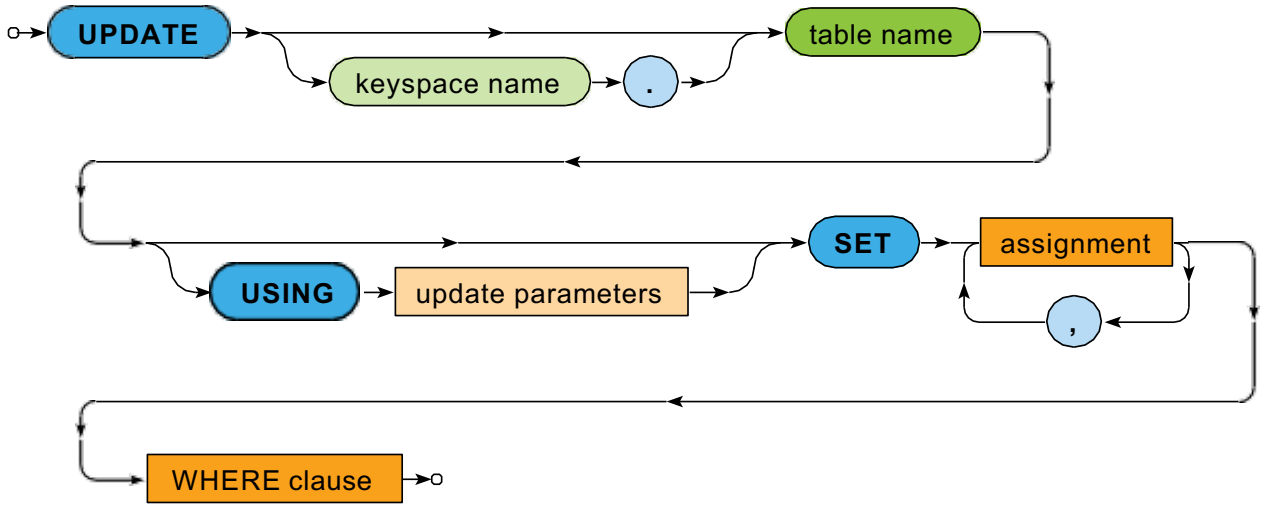
Example

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

Updates

UPDATE statement

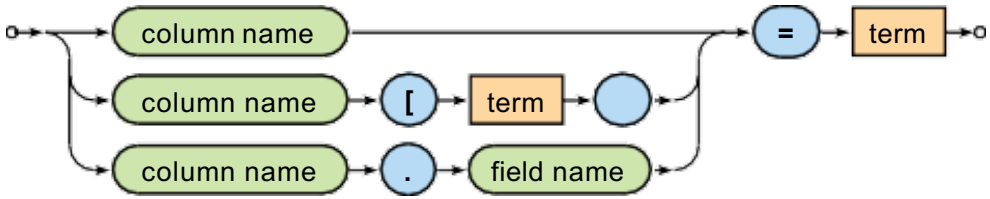
- **Updates existing rows** within a given table
  - When a row with a given primary key does not yet exist, it is inserted
- At least all primary key columns must be specified in the WHERE clause



Updates

UPDATE statement: **assignments**

- Describe modifications to be applied
- Allowed assignments:
  - Value of a whole column is replaced
  - Value of a list or map element is replaced
    - Items of lists are numbered starting with 0
  - Value of a user-defined type field is replaced



Updates

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['vilhelmova'] = 'Helenka',
    genres[1] = 'comedy',
    properties.length = 99
WHERE  id = 'vratnelahve'
```

Note: Due to consistency issues, working with list indices (genres[1] = 'comedy') in Cassandra is not recommended. Use add/remove operations instead

Updates

Examples: modification of collection elements

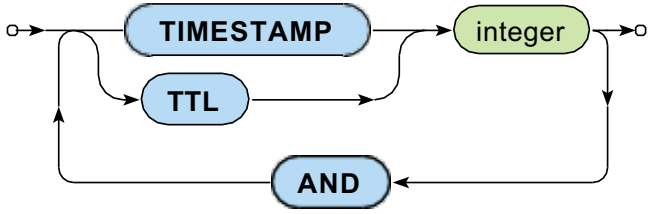
```
UPDATE movies
SET
   actors = actors + { 'vilhelmova': 'Helenka' },
   genres = [ 'drama' ] + genres,
   countries = countries + { 'SK' }
WHERE id = 'vratnelahve'
```

```
UPDATE movies
SET
   actors = actors - { 'vilhelmova', 'landovsky' },
   genres = genres - [ 'drama', 'sci-fi' ],
   countries = countries - { 'SK' }
WHERE id = 'vratnelahve'
```

Insertions and Updates

Update parameters

- **TTL**: time-to-live
  - 0, null or simply missing for persistent values
- **TIMESTAMP**: writetime

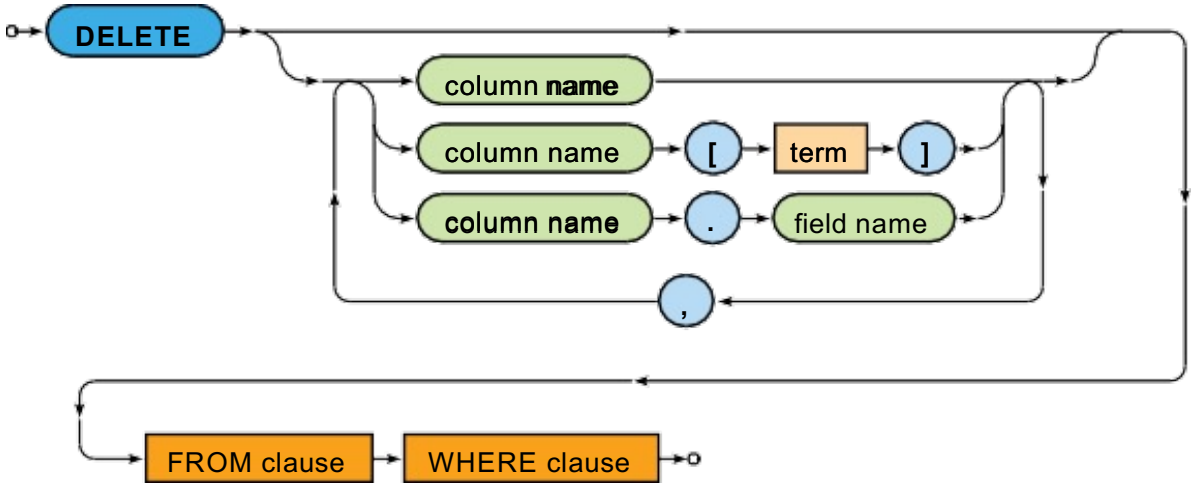


- Only newly inserted / updated values are really affected

Deletions

DELETE statement

- **Removes the matching rows** /
  Preserves these rows but **removes the selected columns** /
  Preserves these columns but **removes elements of collections** or **fields of UDT values**



Lecture Conclusion

Cassandra

- **Wide column store**

Cassandra query language

- DDL statements
- DML statements
  - **SELECT**, **INSERT**, **UPDATE**, **DELETE**