Cassandra - Advanced Topics

Big Data Systems

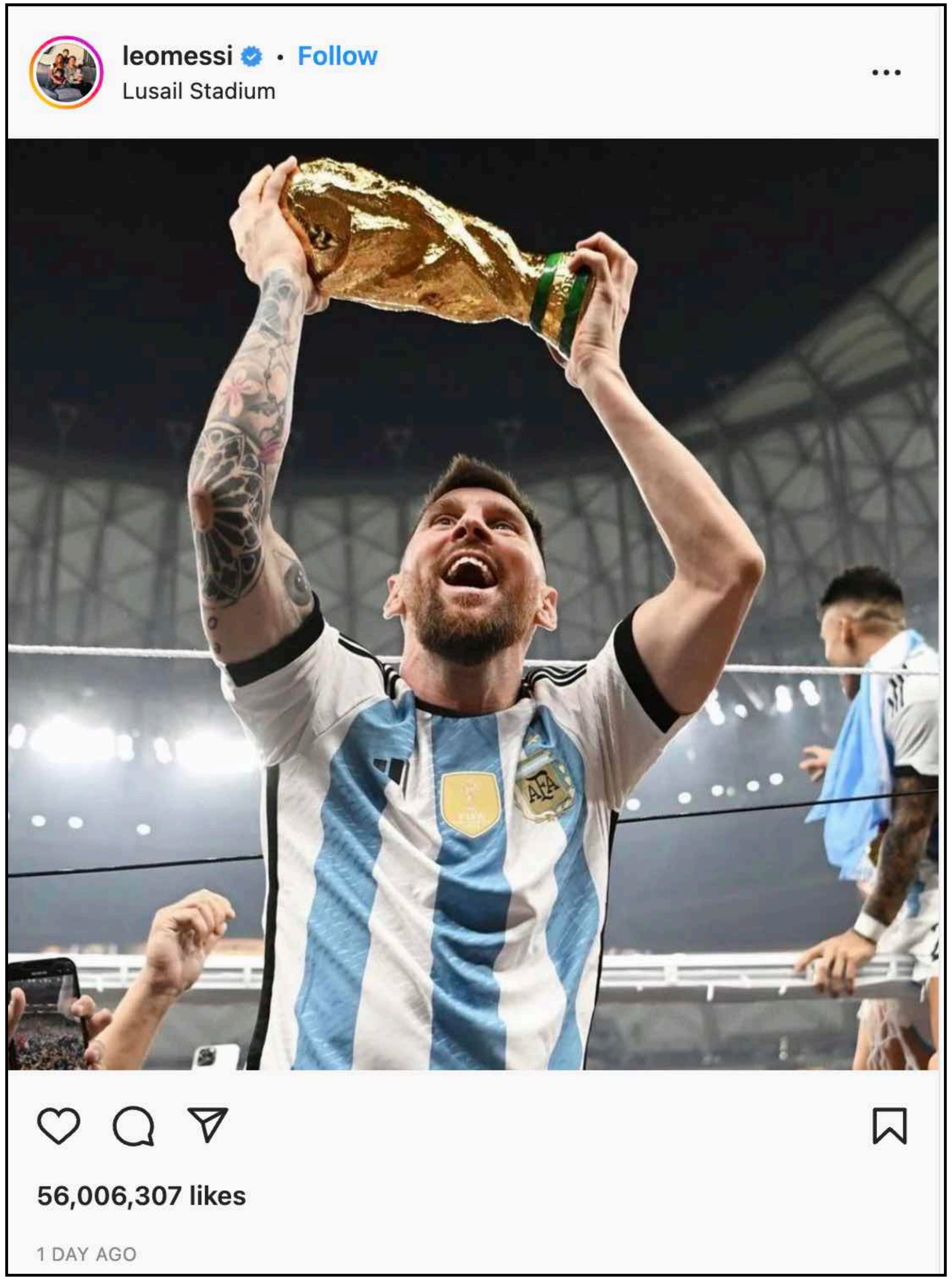
Dr. Rubi Boim

Cassandra advanced topics

- Counters
- Collections
- UDTs
- Batches
- Lightweight transactions
- Tunable consistency
- Deletes & tombstones

Cassandra advanced topics

- **Counters**
- Collections
- UDTs
- Batches
- Lightweight transactions
- Tunable consistency
- Deletes & tombstones

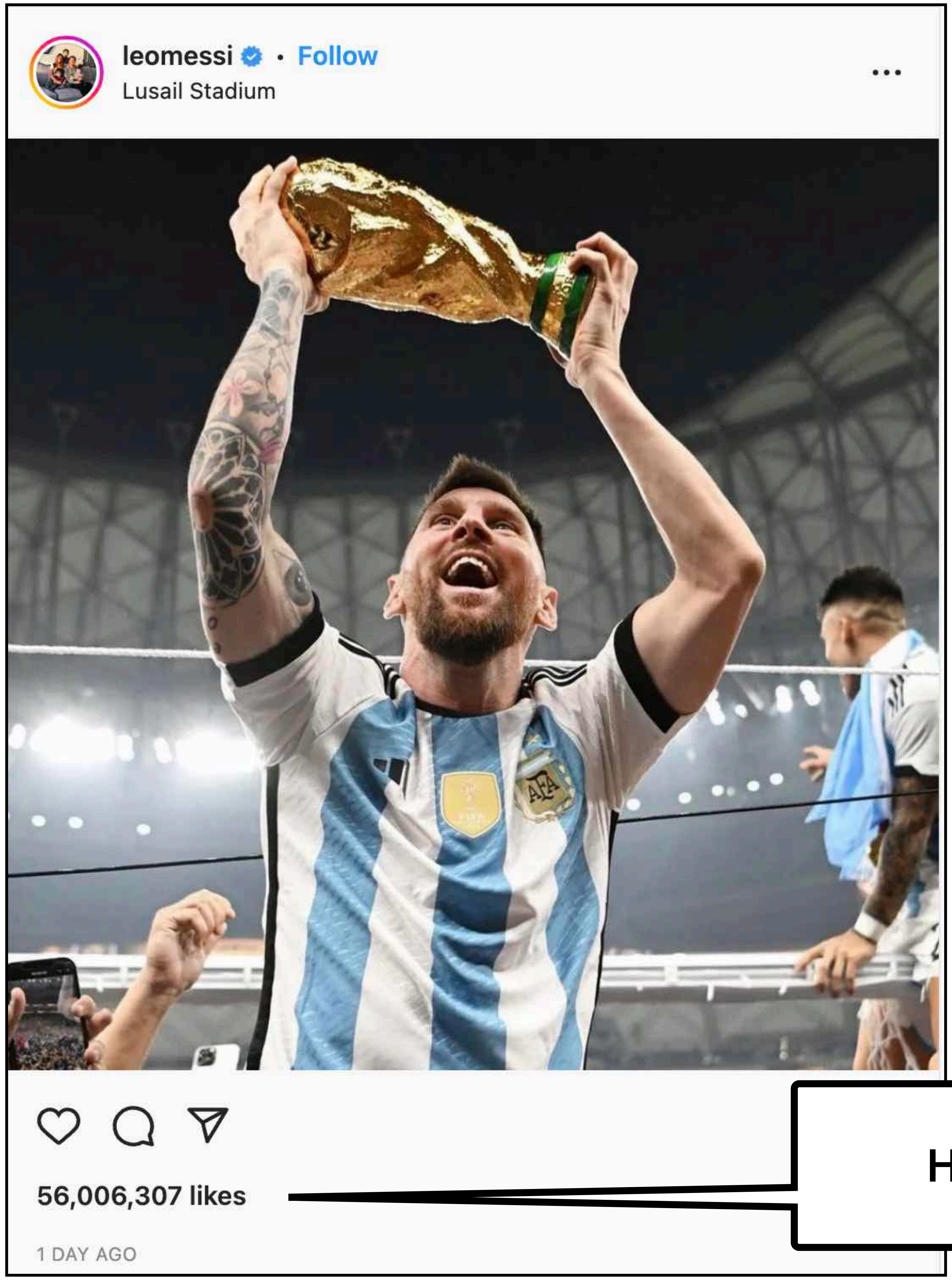


leomessi · Follow

Lusail Stadium

56,006,307 likes

1 DAY AGO



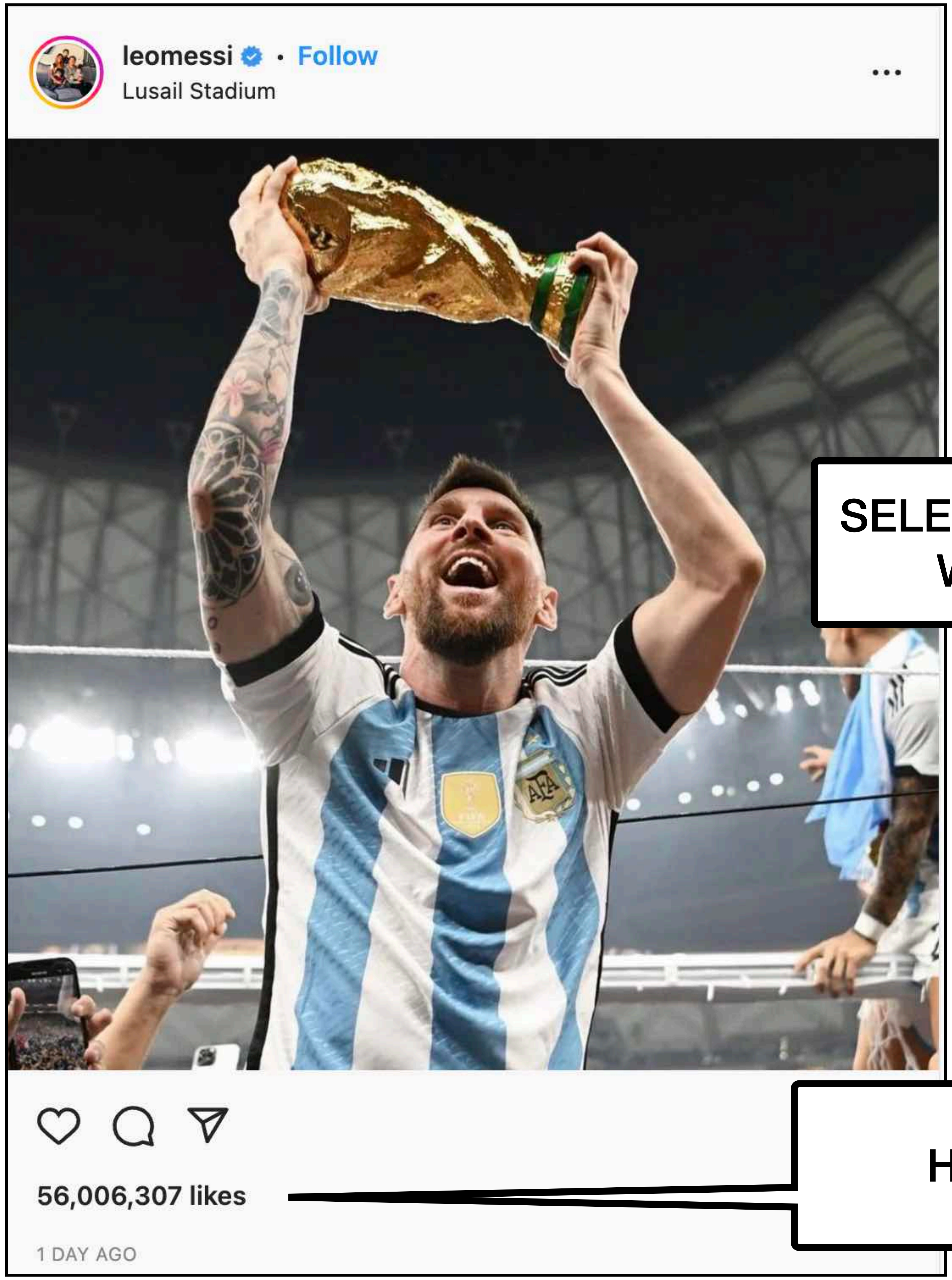
leomessi

Lusail Stadium

56,006,307 likes

1 DAY AGO

How can we calculate this?



leomessi · Follow

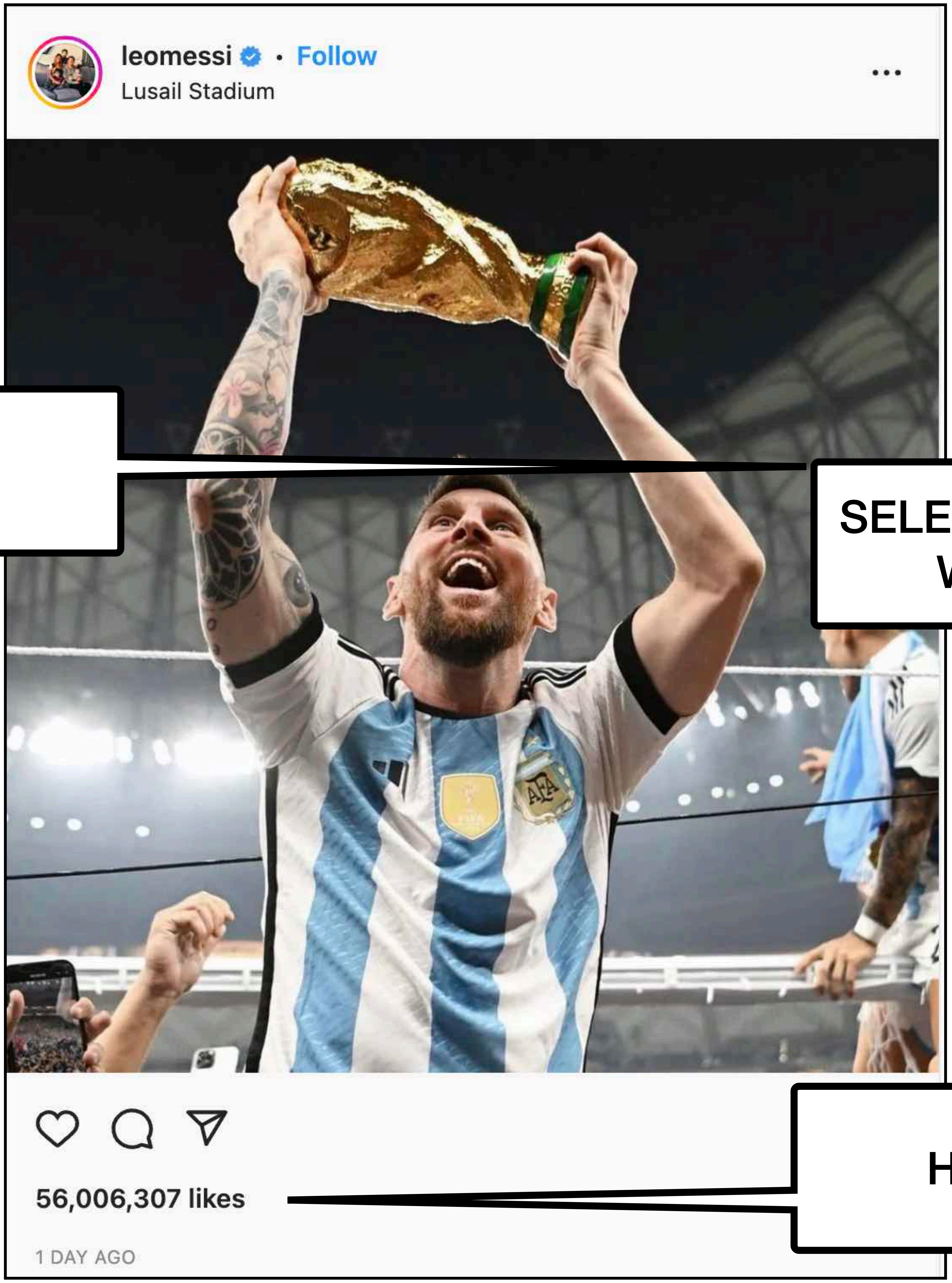
Lusail Stadium

56,006,307 likes

1 DAY AGO

SELECT count(*) FROM likes_by_item
WHERE itemID = 123456789

How can we calculate this?



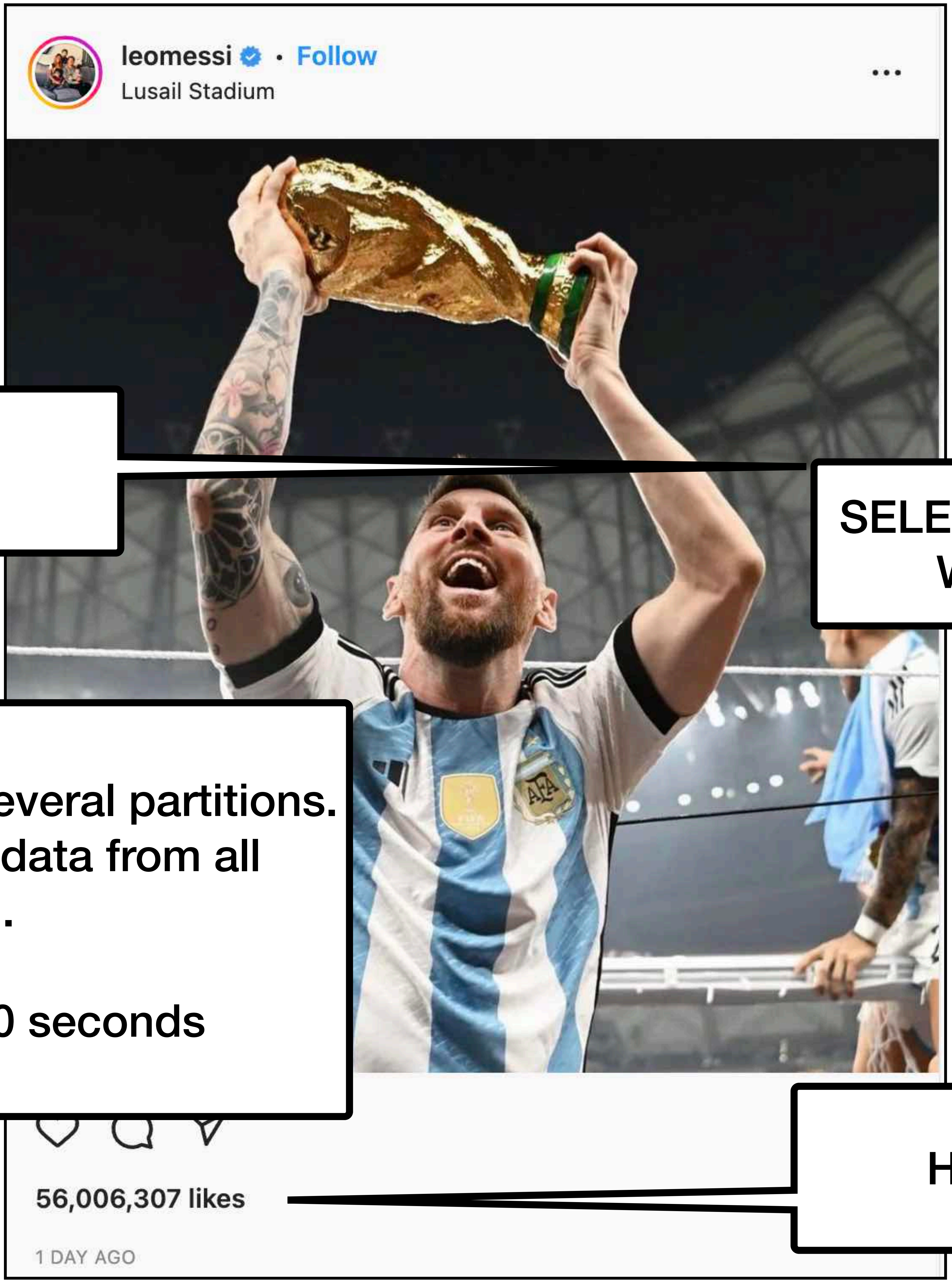
56,006,307 likes

1 DAY AGO

What is the problem?

SELECT count(*) FROM likes_by_item
WHERE itemID = 123456789

How can we calculate this?



56,006,307 likes

How can we calculate this?

```
SELECT count(*) FROM likes_by_item
WHERE itemID = 123456789
```

What is the problem?

The “like” data is probably saved on several partitions. Will take too much time to read all data from all partitions and count all.

We need this data in 1ms, not 10 seconds

Cassandra counters

- A special (powerful) data type
- 64bit signed integer (long)
- Cannot be set - only increment/decrement

  (initial value == 0)
- Used with “UPDATE”

Cassandra counters - example

```
CREATE TABLE movie_view_counts (
   movie_id     BIGINT,
   view_count   COUNTER,
   PRIMARY KEY (movie_id)
);
```

```
UPDATE movie_view_counts
SET view_count = view_count + 1
WHERE movie_id = 123
```

```
SELECT view_count FROM movie_view_counts
WHERE movie_id = 123
```

Cassandra counters - example

```
CREATE TABLE movie_view_counts (
   movie_id      BIGINT,
   view_count    COUNTER,
   PRIMARY KEY (movie_id)
);
```

You can also use “-“ and values different than 1

```
UPDATE movie_view_counts
SET view_count = view_count + 1
WHERE movie_id = 123
```

```
SELECT view_count FROM movie_view_counts
WHERE movie_id = 123
```

Cassandra counters - limitations

- Counter cannot be part of the primary key
- A table that contains a counter can only contain counters
  either all the columns of a table outside the PRIMARY KEY have the counter type, or none of them have it
- Counters does not support expiration (TTL)
- Not idempotent by nature
- Slight consistency issues in distributed scenarios
  due to in-memory and speed optimizations to deal with “read before write”
- Counters can be deleted, but not reused
  can you think of an example this might cause a problem?

Cassandra counters - limitations

- Counter cannot be part of the primary key
- A table that contains a counter can only contain counters
  either all the columns of a table outside the PRIMARY KEY have the counter type, or none of them have it
- Counters does not support expira
- Not idempotent by nature
- Slight consistency issues in distributed scenarios
  due to in-memory and speed optimizations to deal with “read before write”
- Counters can be deleted, but not reused
  can you think of an example this might cause a problem?

Think about an eCommerce store which saves a counter for the number of views for a specific item. The item is removed from the dataset and after a few months it is added again with the same id (key)

Question on the example

- Previous implementation counted the total views
- How can we support the query “views per day”?

```
CREATE TABLE movie_view_counts (
   movie_id      BIGINT,
   view_count    COUNTER,
   PRIMARY KEY (movie_id)
);
```

(previous implementation)

Question on the example - answer

- Add a timestamp column to the key
- Describe a day by rounding (down) to 00:00:00 UTC

```
// a quick version instead of using calendar...
public static long getTSDayRound(long timestamp) {
  long portion = timestamp % MILLISECONDS_IN_DAY;
  return timestamp - portion;
}

// returns the round day for the current time
return getTSDayRound(System.currentTimeMillis());
```

Question on the example - answer

```
CREATE TABLE movie_view_counts_by_day (
   movie_id      BIGINT,
   ts            TIMESTAMP,
   view_count    COUNTER,
   PRIMARY KEY (movie_id, ts)
);
```

```
UPDATE movie_view_counts
SET view_count = view_count + 1
WHERE movie_id = 123 AND
      ts = 1627344000000
```

```
SELECT view_count FROM movie_view_counts
WHERE movie_id = 123 AND
      ts = 1627344000000
```

July 27 2021

Question on the example (2)

- How can we support the query “views per day” and “views per month”?

```
CREATE TABLE movie_view_counts_by_day (
   movie_id      BIGINT,
   ts            TIMESTAMP,
   view_count    COUNTER,
   PRIMARY KEY (movie_id, ts)
);
```

(previous implementation)

Question on the example (2) - answer

- Use the same table
- Use the same “day rounding (down)”
- Use a different query
- Group and sum results on **client side**

Client is the backend which uses Cassandra, not the end user

Question on the example (2) - answer

July 01 2021

July 31 2021

```
SELECT ts, view_count FROM movie_view_counts
WHERE movie_id = 123 AND
      ts >= 1625097600000 AND
      ts <= 1627689600000
```

Query result

ts	view_count
1625097600000	50,023
1625184000000	78,288
…	…
1627689600000	28,052

final result - sum of all values (<u>on client</u>)

Client is the backend which uses Cassandra, not the end user

Question on the example (3)

- How can we support the query “views per day”, “views per month” AND “views per hour”?

```
CREATE TABLE movie_view_counts_by_day (
   movie_id     BIGINT,
   ts           TIMESTAMP,
   view_count   COUNTER,
   PRIMARY KEY (movie_id, ts)
);
```

(previous implementation)

Question on the example (3) - answer

- Use the same table
- Use the a different rounding function:
  “hour rounding (down)”

```
// a quick version instead of using calendar...
public static long getTSHourRound(long timestamp) {
  long portion = timestamp % MILLISECONDS_IN_HOUR;
  return timestamp - portion;
}
```

Discussion (1)

- What is the partition key in the examples?
  why is this super important here?

```
CREATE TABLE movie_view_counts_by_day (
   movie_id     BIGINT,
   ts           TIMESTAMP,
   view_count   COUNTER,
   PRIMARY KEY (movie_id, ts)
);
```

Discussion (1)

- What is the partition key in the examples?
  why is this super important here?

```
CREATE TABLE movie_view_counts_by_day (
   movie_id      BIGINT,
   ts            TIMESTAMP,
   view_count    COUNTER,
   PRIMARY KEY (movie_id, ts)
);
```

We need to read a range of data and we want to do it in a single (read) call

Discussion (2)

- Are there any performance differences between using “round by hour” vs “round by day”?

Discussion (2)

- Are there any performance differences between using “round by hour” vs “round by day”?
  - The number of events should be the same
    (unless you allow a daily event to be saved several times during the day)
  - The number of counters can be X24
    - Query / client runtime
    - storage

Discussion (2)

- Are there any performance differences between using “round by hour” vs “round by day”?
  - The number of events should be the same
    (unless you allow a daily event to be saved several times during the day)
  - The number of counters can be X24
    - Query / client runtime
    - storage

It can be either negligible or crucial - depends on the exact use case

Cassandra advanced topics

- Counters
- **Collections**
- UDTs
- Batches
- Lightweight transactions
- Tunable consistency
- Deletes & tombstones

Cassandra collections

- Multi value columns
  Set / List / Map
- Designed for relatively small amount of data
- Retrieved all together
  no paging / indexes
- Type is fixed for all elements
- Cannot nest (*only FROZEN)
  more on FROZEN later

SET

- Unique, unordered, returned sorted

```
CREATE TABLE movies (
   movie_id      BIGINT,
   title         TEXT,
   genres        SET<text>
   PRIMARY KEY (movie_id)
);
```

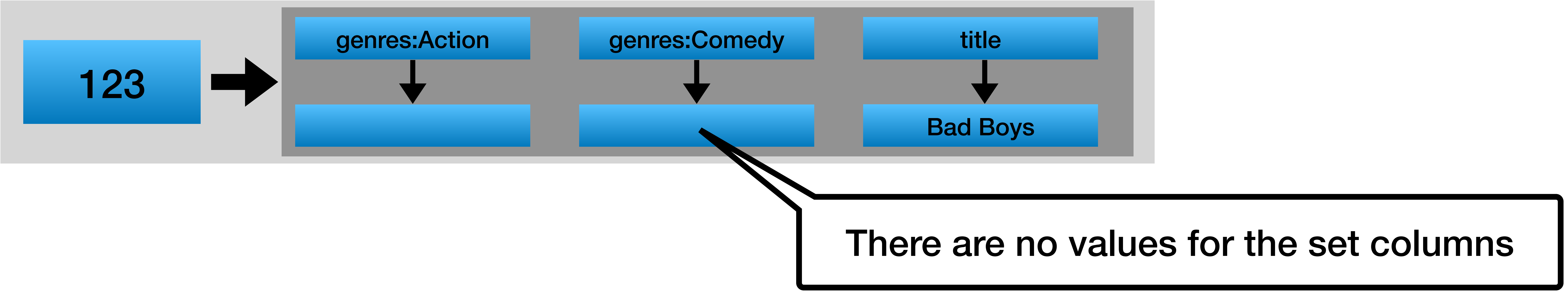
```
INSERT INTO movies
VALUES (123, “Bad Boys”, {“Action”, “Comedy”})

UPDATE movies
SET genres = {“Action”, “Comedy”, “Teen”}
WHERE  id = 123

UPDATE movies
SET genres = genres + {“Teen”}
WHERE  id = 123

UPDATE movies
SET genres = genres - {“Teen”}
WHERE  id = 123
```

SET

- In practice

```
CREATE TABLE movies (
    movie_id      BIGINT,
    title         TEXT,
    genres        SET<text>
    PRIMARY KEY (movie_id)
);
```

```
INSERT INTO movies
VALUES (123, "Bad Boys", {"Action", "Comedy"})
```

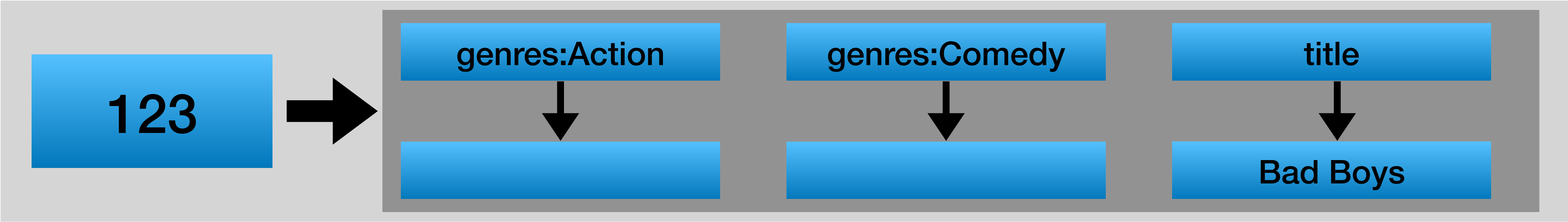


SET

- In practice

```
CREATE TABLE movies (
    movie_id        BIGINT,
    title           TEXT,
    genres          SET<text>
    PRIMARY KEY (movie_id)
);
```

```
INSERT INTO movies
VALUES (123, “Bad Boys”, {“Action”, “Comedy”})
```



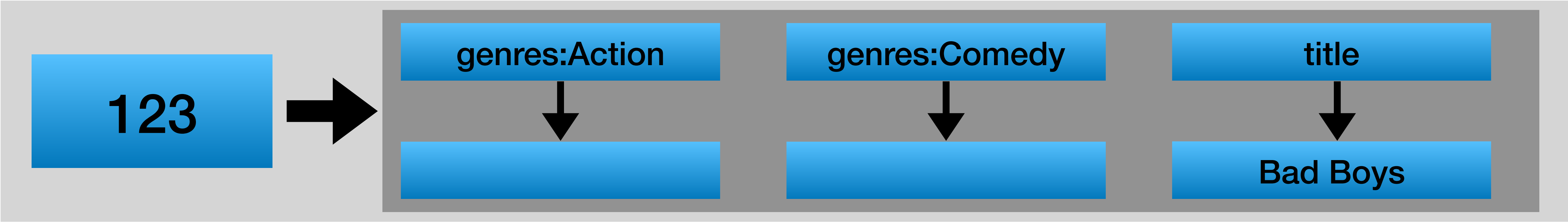
```
UPDATE movies
SET genres = genres + {“Teen”}
WHERE  id = 123
```

SET

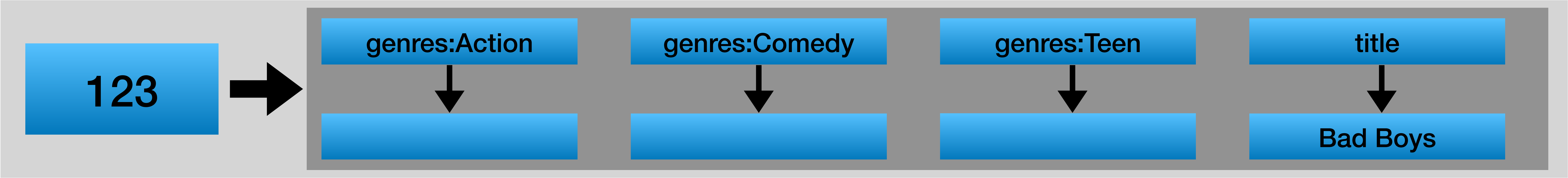
- In practice

```
CREATE TABLE movies (
   movie_id      BIGINT,
   title         TEXT,
   genres        SET<text>
   PRIMARY KEY (movie_id)
);
```

```
INSERT INTO movies
VALUES (123, "Bad Boys", {"Action", "Comedy"})
```



```
UPDATE movies
SET genres = genres + {"Teen"}
WHERE  id = 123
```



LIST

- Duplicated, ordered
- (may) requires read before write

```
CREATE TABLE movies (
   movie_id      BIGINT,
   title         TEXT,
   cast          LIST<text>
   PRIMARY KEY (movie_id)
);
```

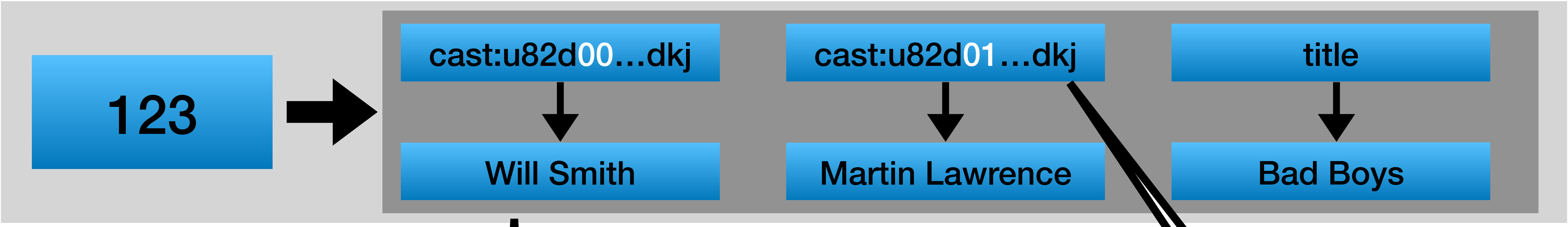
```
INSERT INTO movies
VALUES (123, “Bad Boys”, {“Will Smith”, “Martin Lawrence”})

UPDATE movies
SET cast = cast - {“Martin Lawrence”}  // all matching elements
WHERE  id = 123                           NOT thread-safe


UPDATE movies
SET cast[1] = {“Martin Lawrence”}
WHERE  id = 123

DELETE cast[1] FROM movies WHERE id = 123
```

LIST

- In practice

```
CREATE TABLE movies (
   movie_id      BIGINT,
   title         TEXT,
   cast          LIST<text>
   PRIMARY KEY (movie_id)
);
```

```
INSERT INTO movies
VALUES (123, "Bad Boys", {"Will Smith", "Martin Lawrence"})
```



List values are column values

Column name is added with unique String based on the list order

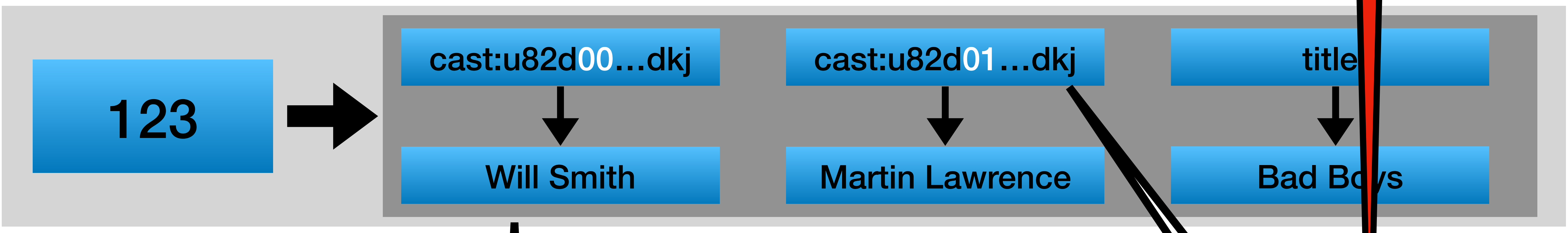
LIST

- In practice

```
CREATE TABLE movies (
    movie_id        BIGINT,
    title           TEXT,
    cast            LIST<text>
    PRIMARY KEY (movie_id)
);
```

```
INSERT INTO m
VALUES (123,                                              )
```

Again - list may required read before write.
Do NOT use unless you know what you are doing



List values are column values

Column name is added with unique String based on the list order

MAP

- Key-Value pair, ordered by keys

```
CREATE TABLE movies (
   movie_id      BIGINT,
   title         TEXT,
   cast          MAP<BIGINT, text>
   PRIMARY KEY (movie_id)
);
```

```
INSERT INTO movies
VALUES (123, “Bad Boys”, {44: “Will Smith”, 45: “Martin Lawrence”})
```

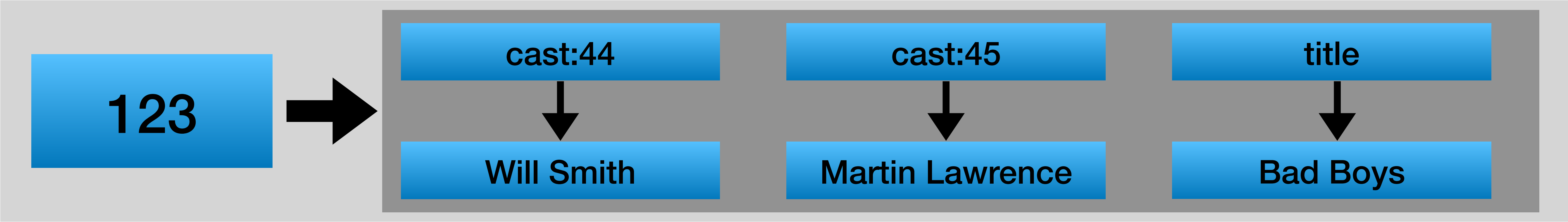
```
UPDATE movies
SET cast = cast - {44}
WHERE  id = 123
```

MAP

- In practice

```
CREATE TABLE movies (
    movie_id      BIGINT,
    title         TEXT,
    cast          MAP<BIGINT, text>
    PRIMARY KEY (movie_id)
);
```

```
INSERT INTO movies
VALUES (123, “Bad Boys”, {44: “Will Smith”, 45: “Martin Lawrence”})
```



Cassandra advanced topics

- Counters
- Collections
- **UDTs**
- Batches
- Lightweight transactions
- Tunable consistency
- Deletes & tombstones

User defined types

frozen == blob for Cassandra
—> all data needs to be set at once

- Attach multiple data fields to a single column
- Any type of field is valid (UDT, Collections)
- Use with FROZEN
  new versions can support non frozen UDT without collections

```
CREATE TYPE full_name (
   first_name   TEXT,
   last_name    TEXT
);
```

```
CREATE TYPE address (
   country      TEXT,
   city         TEXT,
   street       TEXT,
   phones       SET<TEXT>
);
```

User defined types - example

```
CREATE TYPE full_name (
   first_name   TEXT,
   last_name    TEXT
);
```

```
CREATE TABLE users (
   user_id      BIGINT,
   name         FROZEN <full_name>,
   age          INT
);
```

```
INSERT INTO user
VALUES (123, {first_name: “Lebron”, last_name: “James”}, 36)
```

User defined types - notes

- You can love them or hate them
- Useful with collections

```
CREATE TYPE address (
   country       TEXT,
   city          TEXT,
   street        TEXT,
   phones        SET<TEXT>
);
```

```
CREATE TABLE users (
   user_id       BIGINT,
   addresses     SET<FROZEN <ADDRESS>>
);
```

Cassandra advanced topics

- Counters
- Collections
- UDTs
- **Batches**
- Lightweight transactions
- Tunable consistency
- Deletes & tombstones

Important note

- Batches in Cassandra are different from relational databases

- TLDR; they are “half” relational transactions
  batch is isolate and atomic in a single partition

In relational databases

- Batches & Transactions are collections of commands (Insert/Update/Delete) sent together to the server

- Batch
  - NO rollback / ACID
  - used to increase performance by reducing server calls

- Transaction
  - full ACID

In relational databases

Atomicity
Consistency
Isolation
Durability

No ACID

```
“DRIVER START BATCH”
   INSERT INTO users VALUES(“Rubi”);
   ...
   INSERT INTO users VALUES(“Tova”);
“DRIVER END BATCH”
```

If “Tova” fails, “Rubi” is still added

ACID

```
START TRANSACTION
   INSERT INTO flights VALUES(“Rubi”, “TLV-NY”);
   ...
   INSERT INTO hotels VALUES(“Rubi”, “Hilton-NY”);
COMMIT
```

If “Hilton-NY” fails, the flight is NOT added

Cassandra Batch

- Executes several commands
- If statement apply to the same partition: atomic & isolated

Same partition

```
BEGIN BATCH
    INSERT INTO users_by_country VALUES(“Israel”,123, “Rubi”);
    ...
    INSERT INTO users_by_country VALUES(“Israel”,123, “Tova”);
APPLY BATCH
```

If “Tova” Fails, “Rubi” will not be added

Isolation - can NOT read “Rubi” until “Tova” is added

Cassandra Batch - performance?

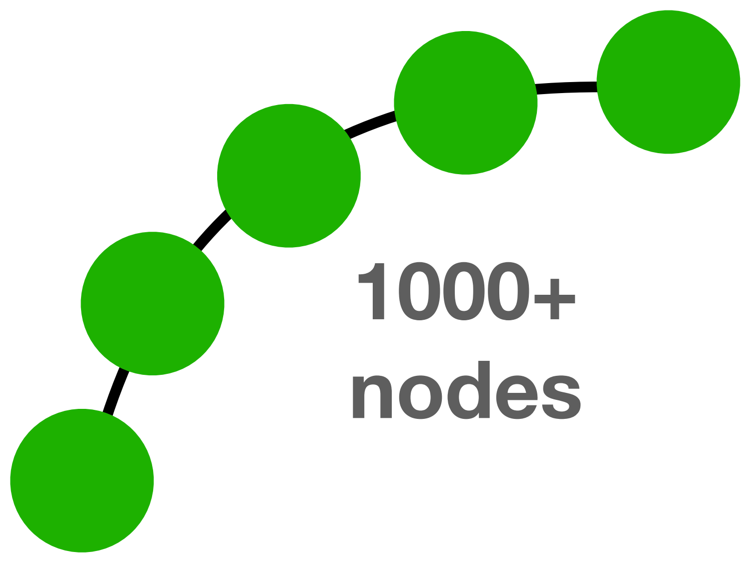
users	
user_id	K
name	

- Each batch is sent to a single coordinator (node), logged and then executed

- What happens in each scenario?

  -
    ```
    BEGIN BATCH
        INSERT INTO users VALUES(123, “Rubi”);
        ...
        INSERT INTO users VALUES(456, “Tova”);
    APPLY BATCH
    ```

  -
    ```
    INSERT INTO users VALUES(123, “Rubi”);
    ...
    INSERT INTO users VALUES(456, “Tova”);
    ```



Cassandra Batch - performance?

users	
user_id	K
name	

- Each batch is send to a single coordinator (node), logged and then executed

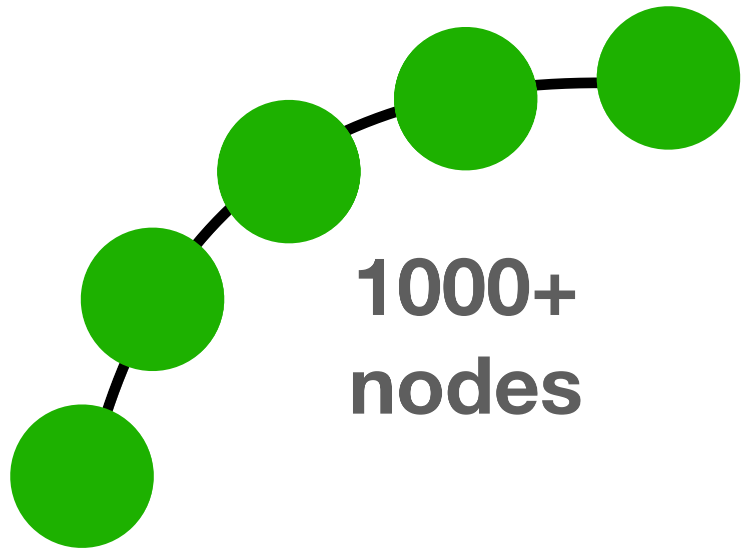
- What happens in each scenario?

  -
    ```
    BEGIN BATCH
        INSERT INTO users VALUES(123, “Rubi”);
        ...
        INSERT INTO users VALUES(456, “Tova”);
    APPLY BATCH
    ```

Contact the node of the first partition key (123) and log the batch.

Then that node contacts the node of partition 456

  -
    ```
    INSERT INTO users VALUES(123, “Rubi”);
    ...
    INSERT INTO users VALUES(456, “Tova”);
    ```



Cassandra Batch - performance?

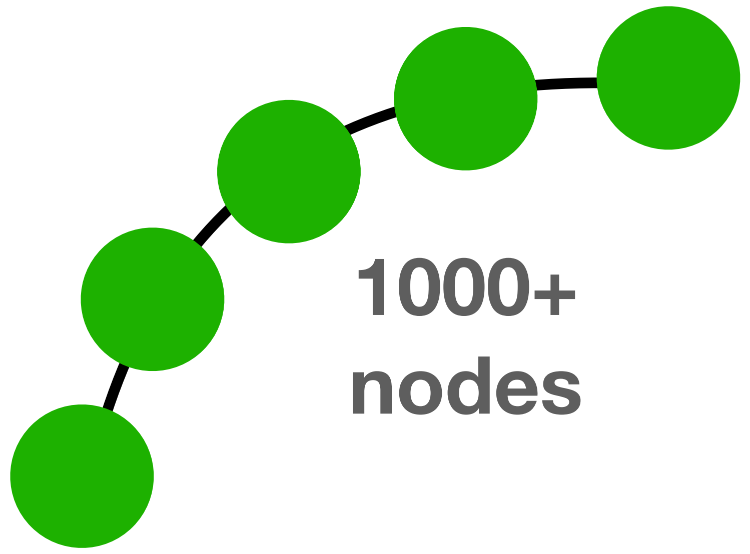
users	
user_id	K
name	

- Each batch is send to a single coordinator (node), logged and then executed
- What happens in each scenario?

  -
    ```
    BEGIN BATCH
        INSERT INTO users VALUES(123, "Rubi");
        ...
        INSERT INTO users VALUES(456, "Tova");
    APPLY BATCH
    ```

  -
    ```
    INSERT INTO users VALUES(123, "Rubi");
    ...
    INSERT INTO users VALUES(456, "Tova");
    ```

Each insert calls directly the relevant node



Cassandra Batch - performance?

users	
user_id	K
name	

- Each ba ode), logged

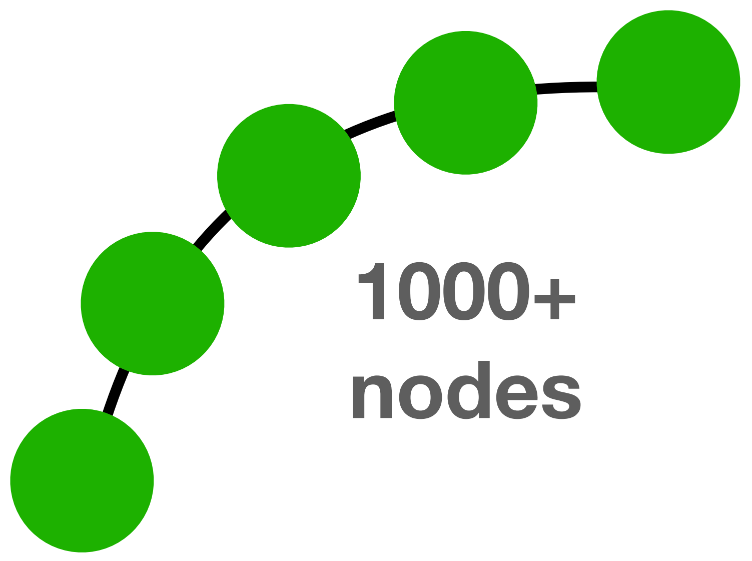
- What ha

Batches in Cassandra almost always do not help with performance

Use it only if you need single partition isolation

-
  ```
  BEGIN B
      INS
      ...
      INSERT INTO users VALUES(456, “Tova”);
  APPLY BATCH
  ```

-
  ```
  INSERT INTO users VALUES(123, “Rubi”);
  ...
  INSERT INTO users VALUES(456, “Tova”);
  ```



Cassandra advanced topics

- Counters
- Collections
- UDTs
- Batches
- **Lightweight transactions**
- Tunable consistency
- Deletes & tombstones

Lightweight Transactions

- Checks a condition prior to Insert/Update/Delete
- Expensive (more than a read and write)
- An “ACID Transaction” at the partition level

Lightweight Transactions - examples

```
INSERT INTO movies
VALUES(3,“American Pie”,1999,96)
IF NOT EXISTS
```

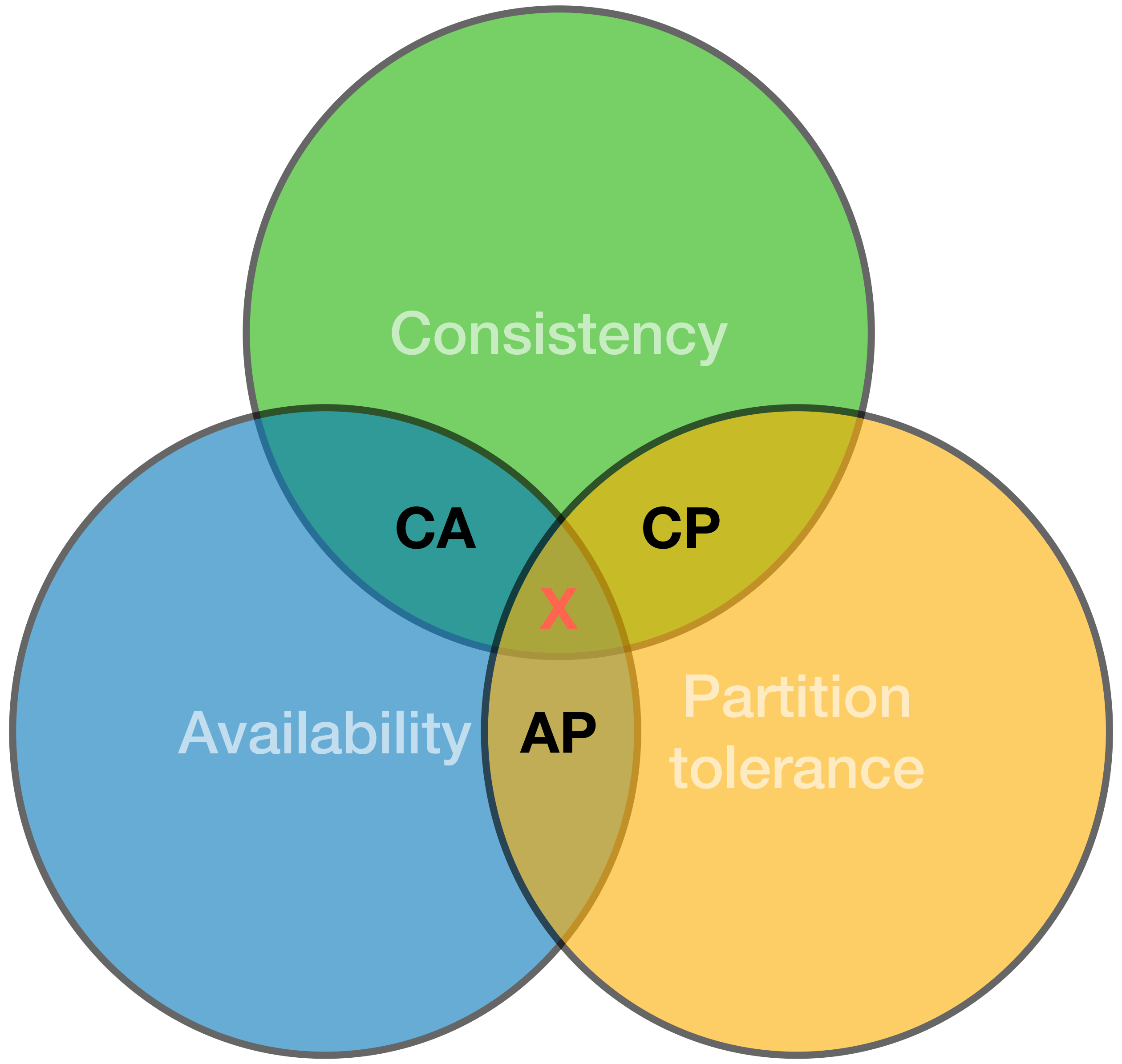
```
UPDATE movies
SET duration = 96
WHERE id = 3
IF year = 1999
```

movies	
id	K
title	
year	
duration	

Cassandra advanced topics

- Counters
- Collections
- UDTs
- Batches
- Lightweight transactions
- **Tunable consistency**
- Deletes & tombstones

Recap - CAP

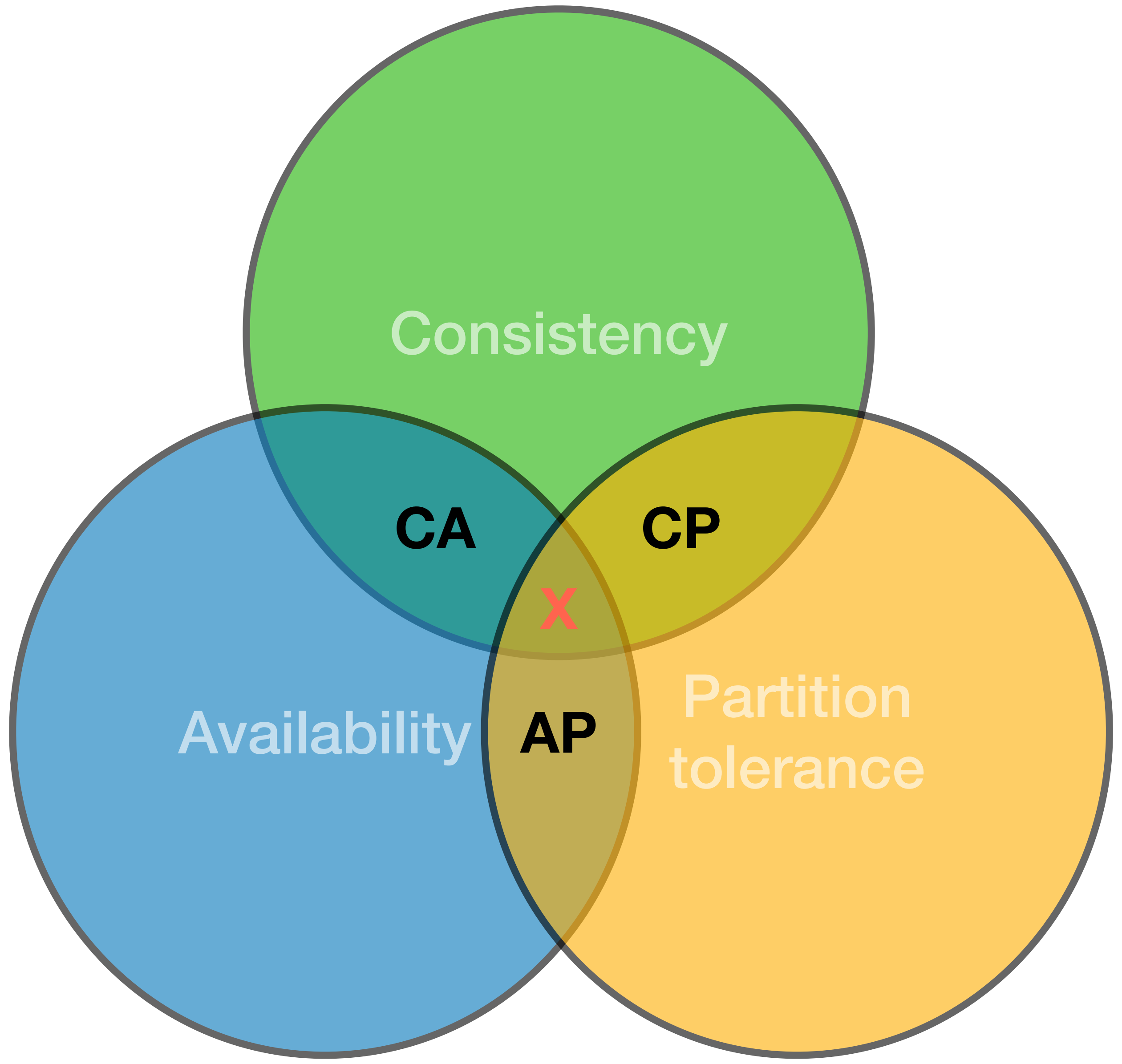


- Consistency
  Every read receives the most recent write or an error

- Availability
  Every request receives a (non-error) response,
  without the guarantee that it contains the most recent write

- Partition tolerance
  The system continues to operate despite an arbitrary number of messages being dropped (or delayed) by the network

Recap - CAP



- TLDR; If a node is down/unreachable
  - Cancel the operation (CP)
  - Return result with (maybe) inconsistency (AP)

Tunable consistency in Cassandra

- When performing read/write, consistency level can be specified
  Consistency level = # of nodes (replicas) needs to response

- `ONE/TWO/QUORUM/LOCAL_QUORUM/ALL/…`

Tunable consistency in Cassandra

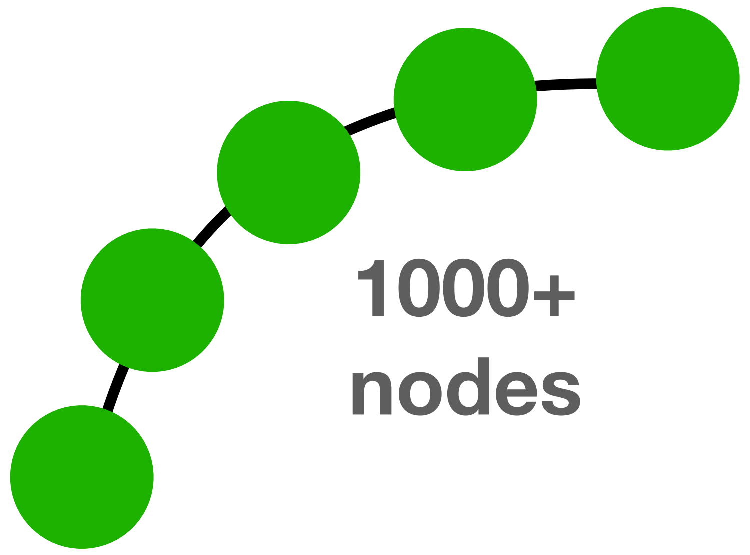
```
// within cqlsh session
CONSISTENCY QUORUM
INSERT INTO movies VALUES(3,“American Pie”,1999,96)
```

When to use which level?

- A function of application logic & resources (money)

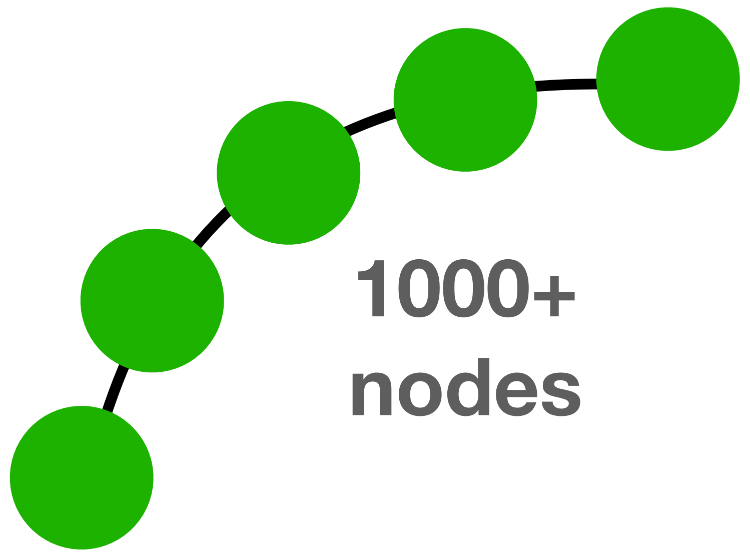
When to use which level?

- A function of application logic & resources (money)
- For example:
  - A “like” event should get ONE or QUORUM?



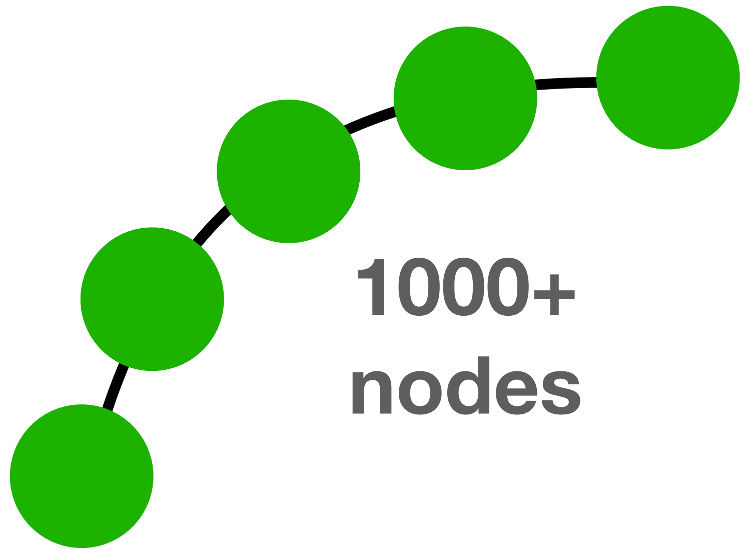
When to use which level?

- A function of application logic & resources (money)
- For example:
  - A “like” event should get ONE or QUORUM?
  - A “buy” event should get ONE or QUORUM?



When to use which level?

- A function of application logic & resources (money)
- For example:
  - A “like” event should get ONE or QUORUM?
  - A “buy” event should get ONE or QUORUM?
  - # of available rooms in a hotel should get ONE or QURUM?

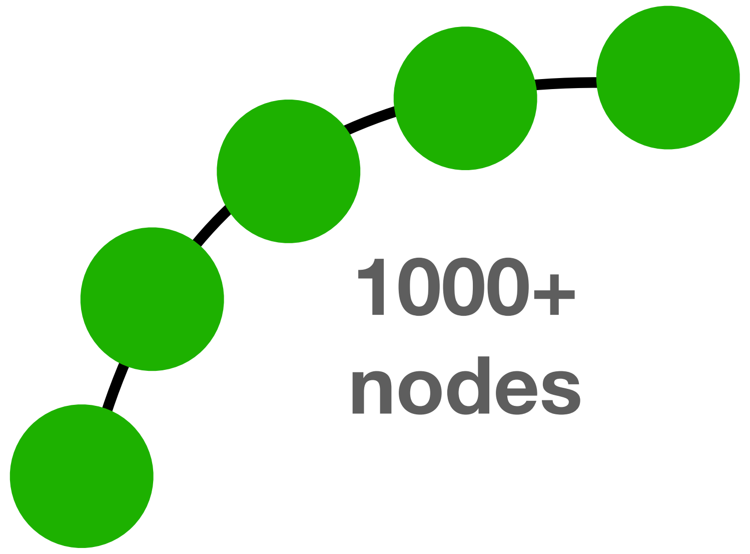


1000+ nodes

When to use which level?

- A function of application logic & resources (money)
- For example:
  - A “like” event should get ONE or QUORUM?
  - A “buy” event should get ONE or QUORUM?
  - # of available rooms in a hotel should get ONE or QURUM?

- <u>Critical for performance on large scale</u>



Cassandra advanced topics

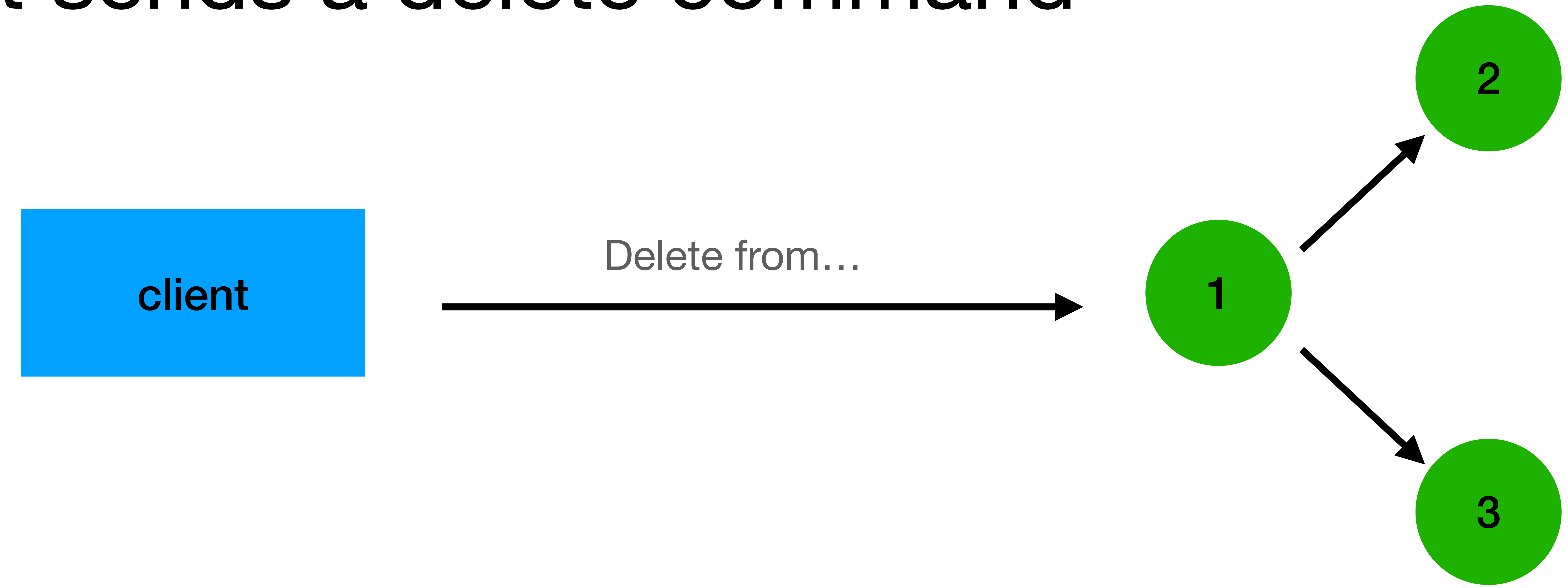
- Counters
- Collections
- UDTs
- Batches
- Lightweight transactions
- Tunable consistency
- **Deletes & tombstones**

Deletes in a distributed system

- A hard problem.
  Why?

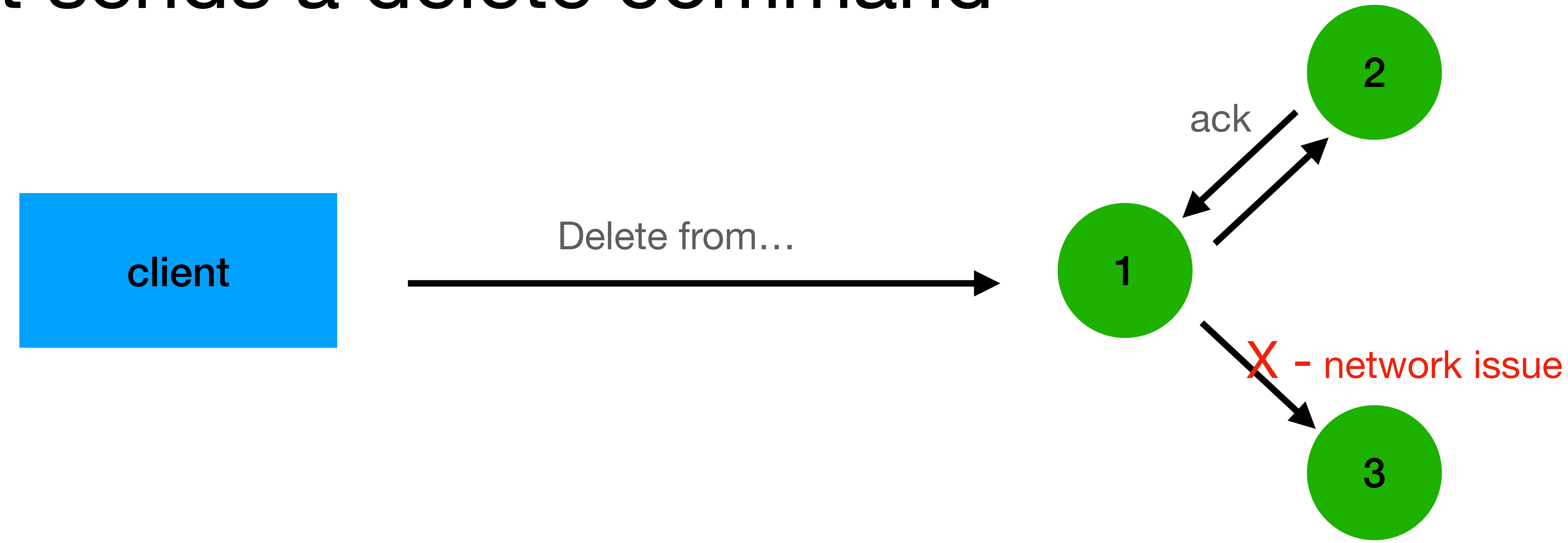
Delete example - distributed system

- 3 nodes, replication factor = 3, consistency=quorum
- A client sends a delete command



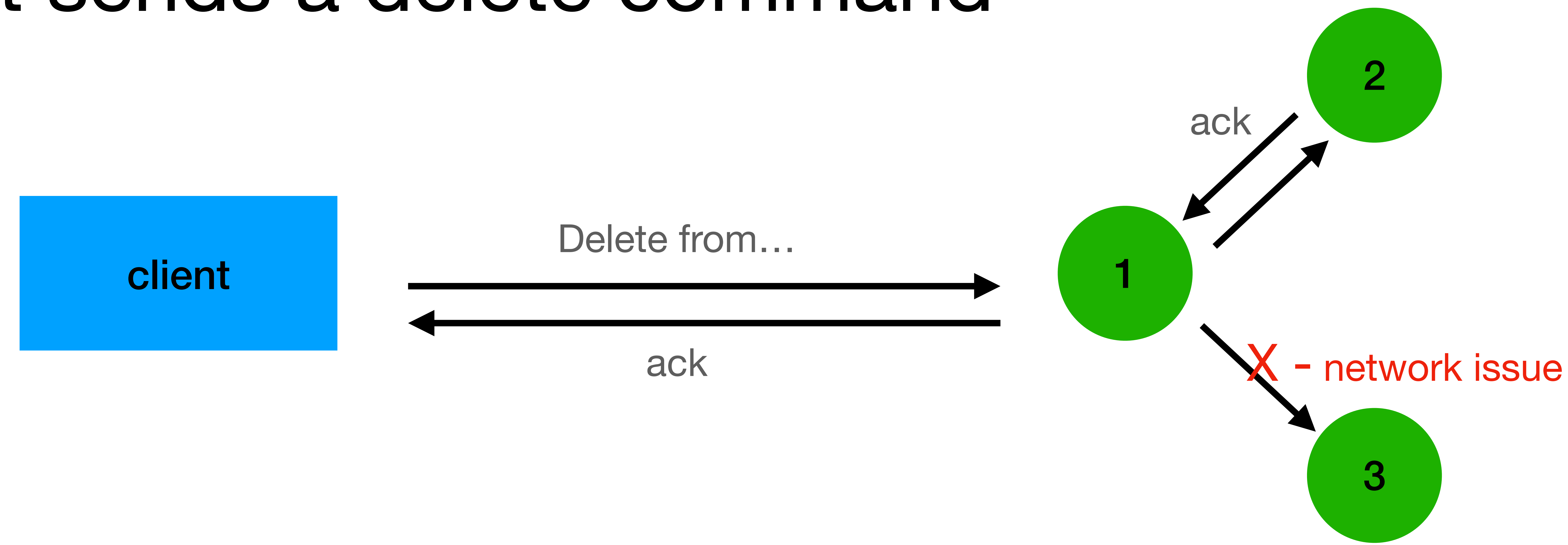
Delete example - distributed system

- 3 nodes, replication factor = 3, consistency=quorum
- A client sends a delete command



Delete example - distributed system

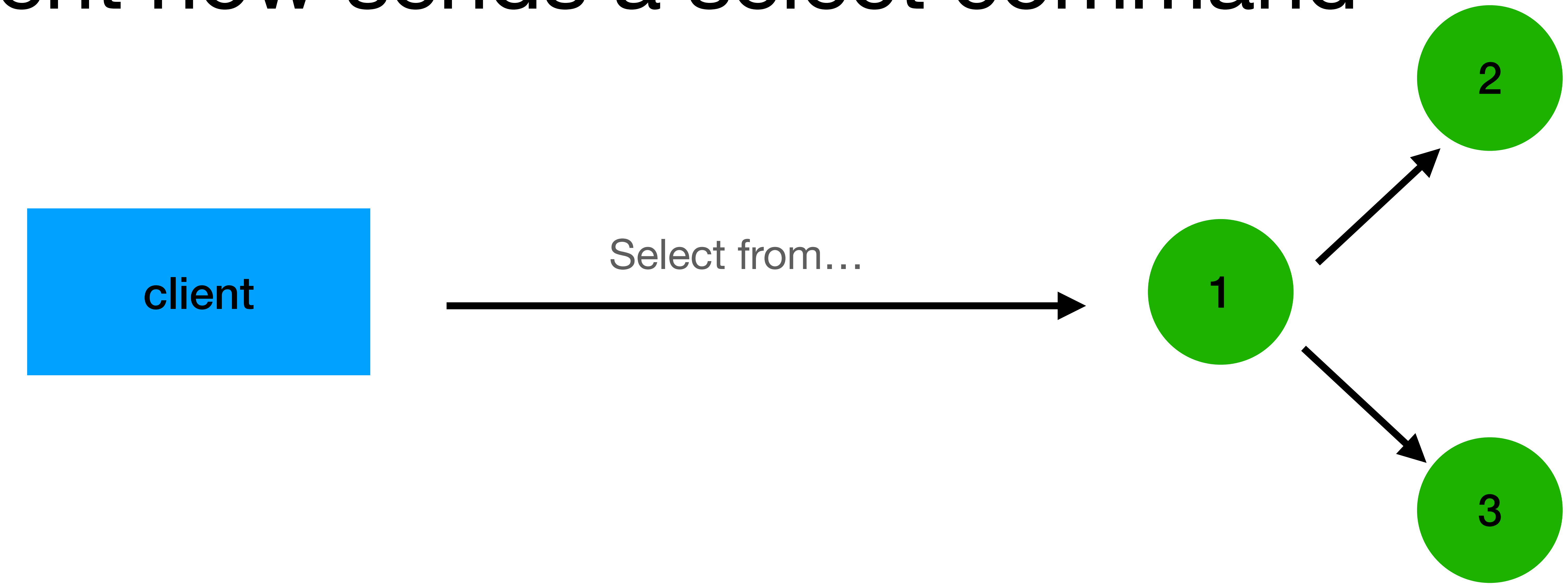
- 3 nodes, replication factor = 3, consistency=quorum
- A client sends a delete command



- Client receives success (2 out of 3)

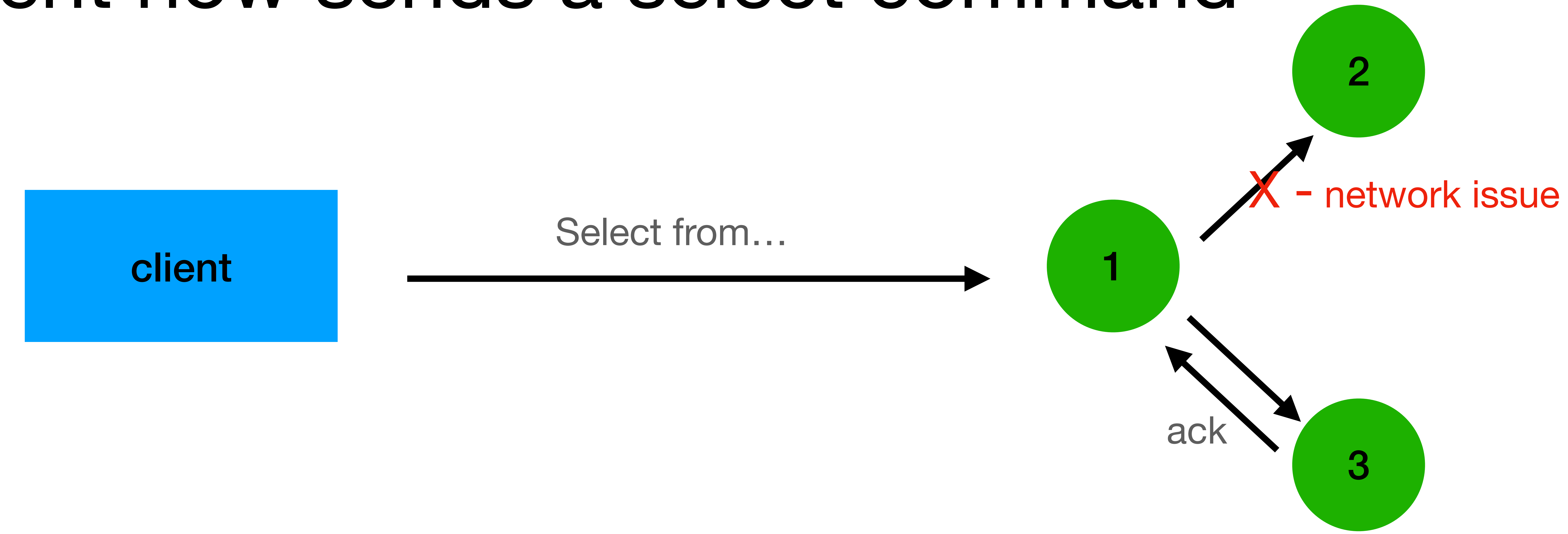
Delete example - distributed system

- 3 nodes, replication factor = 3, consistency=quorum
- The client now sends a select command



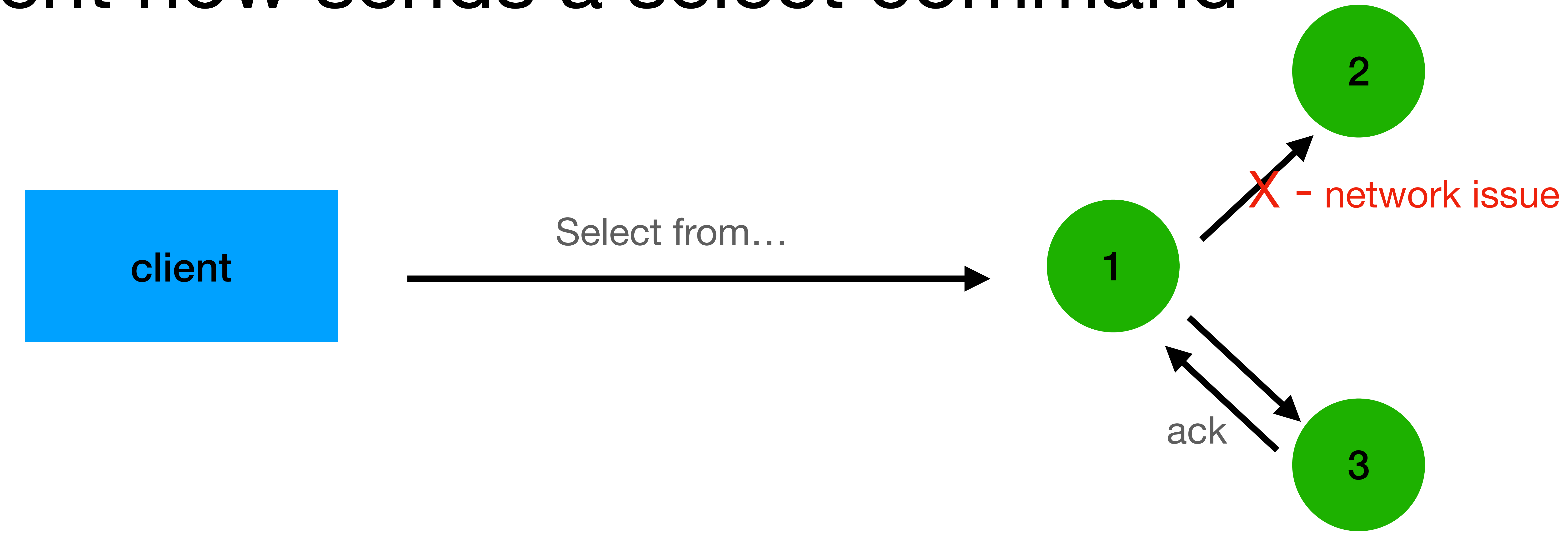
Delete example - distributed system

- 3 nodes, replication factor = 3, consistency=quorum
- The client now sends a select command



Delete example - distributed system

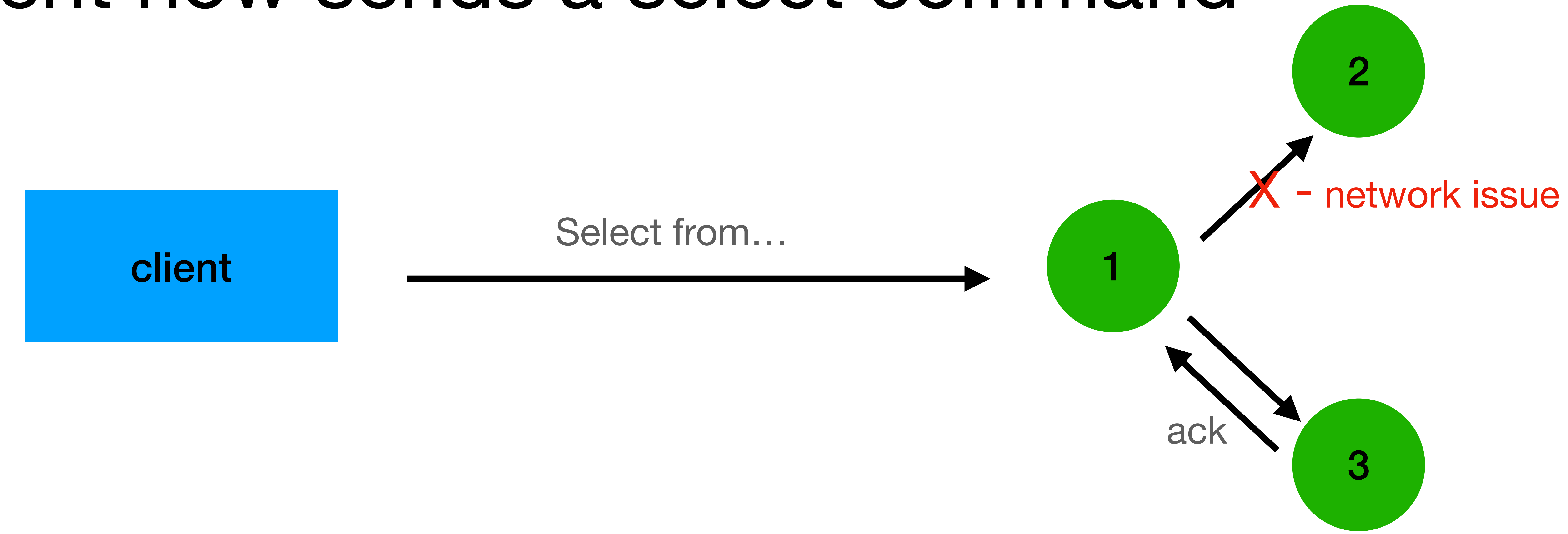
- 3 nodes, replication factor = 3, consistency=quorum
- The client now sends a select command



- Conflict! Node 3 contains data, Node 1 does not

Delete example - distributed system

- 3 nodes, replication factor = 3, consistency=quorum
- The client now sends a select command



- Conflict! Node 3 contains data, Node 1 does not

—> Cassandra will return “zombie” / “ghost” data

Cassandra solution (simplified)

- When deleting, create a “delete entry" - **tombstone**
- Solves 2 problems:
  - the “ambiguous read”
  - immutable storage (SSTables)

- Before reads - Cassandra checks for relevant tombstones

Tomestones

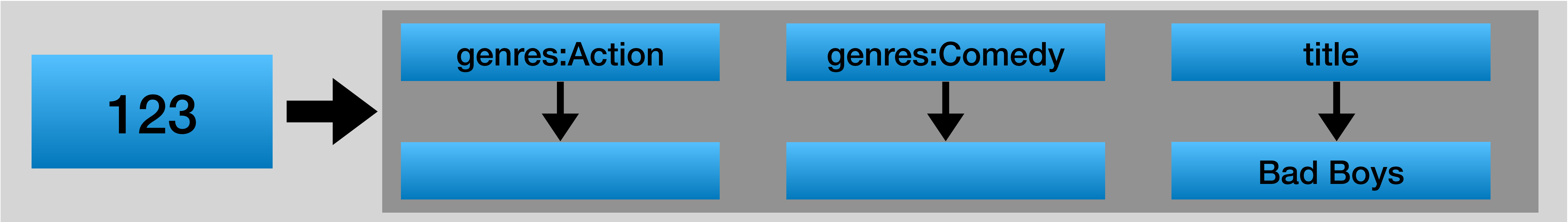
- Created when
  - DELETE
  - Setting TTLs
  - Inserting NULLs (avoid!)
  - Inserting data into a collection
    when inserting the entire collection

Why?

Tombstone & SET

```
CREATE TABLE movies (
   movie_id       BIGINT,
   title          TEXT,
   genres         SET<text>
   PRIMARY KEY (movie_id)
);
```

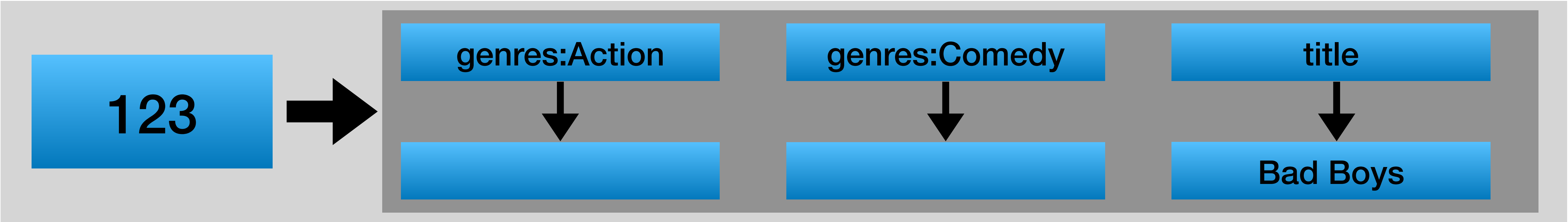
```
INSERT INTO movies
VALUES (123, “Bad Boys”, {“Action”, “Comedy”})
```



Tombstone & SET

```
CREATE TABLE movies (
   movie_id       BIGINT,
   title          TEXT,
   genres         SET<text>
   PRIMARY KEY (movie_id)
);
```

```
INSERT INTO movies
VALUES (123, “Bad Boys”, {“Action”, “Comedy”})
```

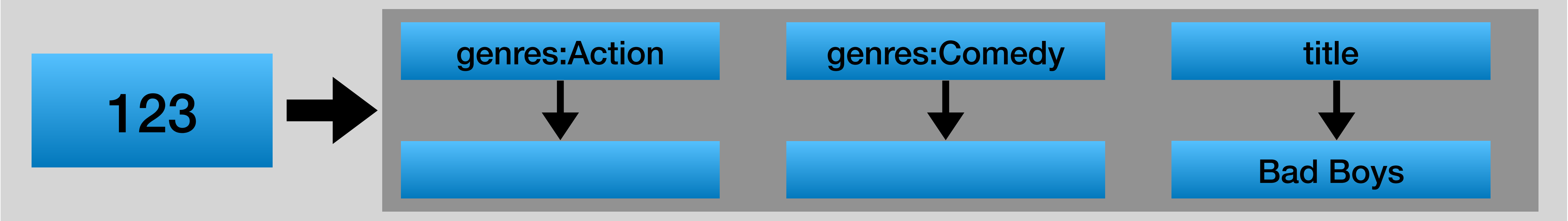


```
INSERT INTO movies
VALUES (123, “Bad Boys”, {“Teen”, “Drama”})
```

Tombstone & SET

```
CREATE TABLE movies (
   movie_id     BIGINT,
   title        TEXT,
   genres       SET<text>
   PRIMARY KEY (movie_id)
);
```

```
INSERT INTO movies
VALUES (123, “Bad Boys”, {“Action”, “Comedy”})
```



```
INSERT INTO movies
VALUES (123, “Bad Boys”, {“Teen”, “Drama”})
```

As data is stored in separate columns, we need to delete all previous existing columns

Tomestones - how long do we keep them?

Any ideas?

Tomestones - how long do we keep them?

Tombstones can be removed once:

- Creation time is longer than `gc_grace_seconds`
  default is 10 days
  - A **repair** should run at least once every `gc_grace_seconds`
    repairs assures consistency among all nodes

- All sstables that could contain the relevant data are involved in the compaction

Tomestones - problem

- Tombstones had performance hit for queries
- Warning in 1k tombstones per partition query
- Error in 100k tombstones per partition query

Tomestones - problem - SOLUTION

- It all comes down to the data model
  - Adjusting and `gc_grace_seconds` and Repairs
    if you are doing this —> probably problems in production :(

- More on this later…
  modeling multi tenants for example