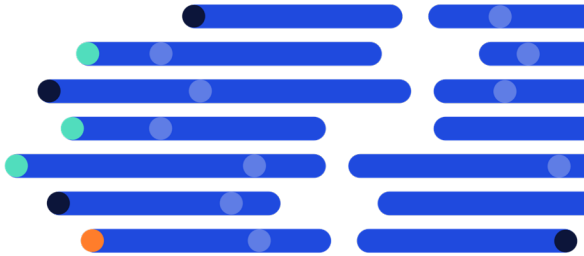
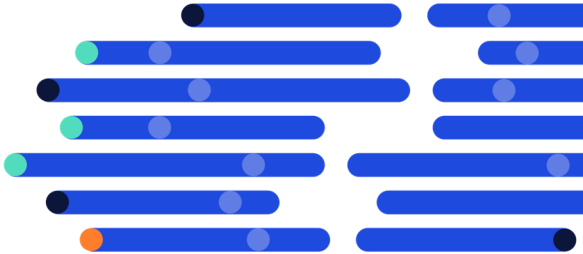
DATASTAX

DSE Fasttrack



DATASTAX

Denormalization



Typical Relational Structure

- This is a relational example—not Apache Cassandra!

videos

id	title	runtime	year
1	Insurgent	119	2015
2	Interstellar	98	2014
3	Mockingjay	122	2014
…	…	…	…

users

id	login	name
a	emotions	Mr. Emotional
b	clueless	Mr. Naïve
c	noshow	Mr. Inactive
…	…	…

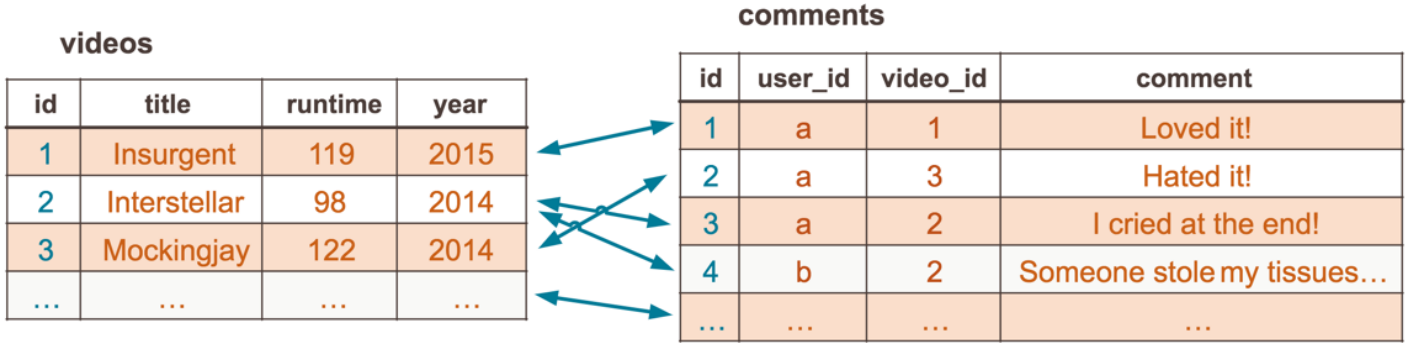
comments

id	user_id	video_id	comment
1	a	1	Loved it!
2	a	3	Hated it!
3	a	2	I cried at the end!
4	b	2	Someone stole my tissues…
…	…	…	…

Typical Relational Structure

- This is a relational example—not Apache Cassandra!

```
SELECT comment
FROM videos
JOIN comments
ON videos.id = comments.video_id
WHERE title = 'Interstellar'
```



videos

id	title	runtime	year
1	Insurgent	119	2015
2	Interstellar	98	2014
3	Mockingjay	122	2014
…	…	…	…

comments

id	user_id	video_id	comment
1	a	1	Loved it!
2	a	3	Hated it!
3	a	2	I cried at the end!
4	b	2	Someone stole my tissues…
…	…	…	…

Typical Relational Structure

- This is a relational example—not Apache Cassandra!

```
SELECT comment
FROM videos
JOIN comments
ON videos.id = comments.video_id
WHERE title = 'Interstellar'
```

videos JOIN comments

id	title	runtime	year	id	user_id	video_id	comment
1	Insurgent	119	2015	1	a	1	Loved it!
3	Mockingjay	122	2014	2	a	3	Hated it!
2	Interstellar	98	2014	3	a	2	I cried at the end!
2	Interstellar	98	2014	4	b	2	Someone stole my tissues…
…	…	…	…	…	…	…	…

Typical Relational Structure

- This is a relational example—not Apache Cassandra!

```
SELECT comment
FROM videos
JOIN comments
ON videos.id = comments.video_id
WHERE title = 'Interstellar'
```

WHERE title = 'Interstellar'

id	title	runtime	year	id	user_id	video_id	comment
1	Insurgent	119	2015	1	a	1	Loved it!
3	Mockingjay	122	2014	2	a	3	Hated it!
2	**Interstellar**	**98**	**2014**	**3**	**a**	**2**	**I cried at the end!**
2	**Interstellar**	**98**	**2014**	**4**	**b**	**2**	**Someone stole my tissues…**
…	…	…	…	…	…	…	…

Typical Relational Structure

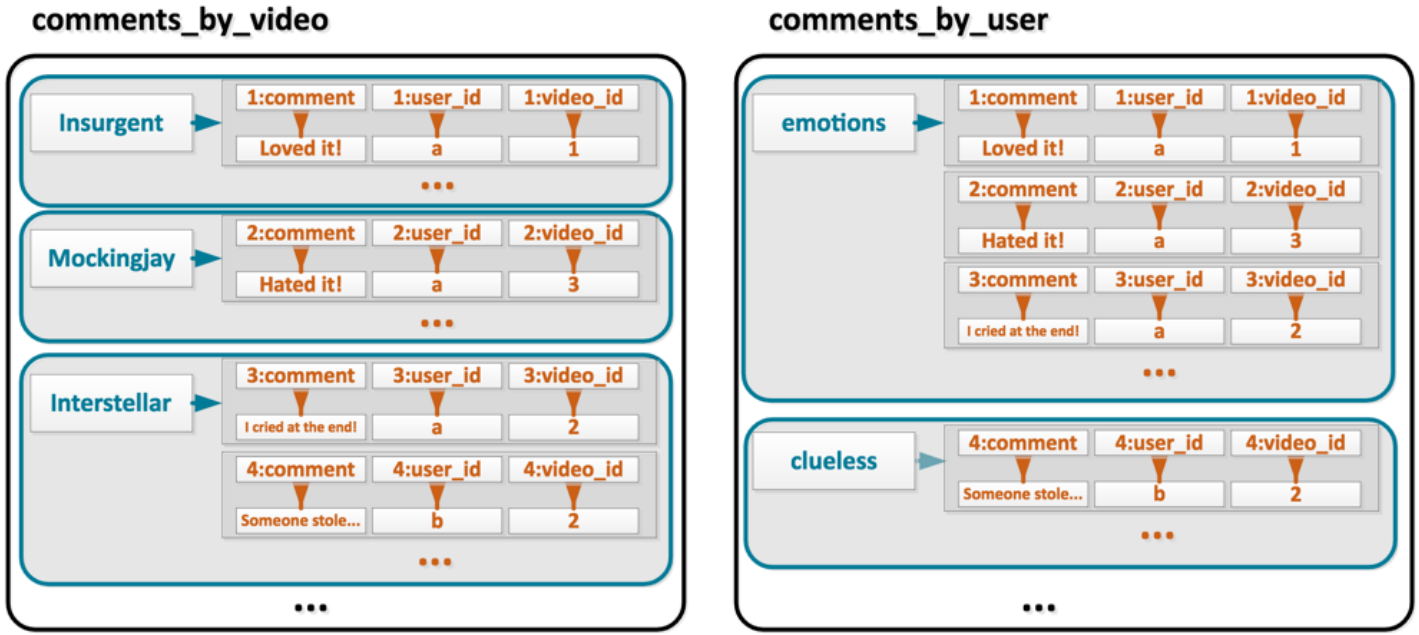
- This is a relational example—not Apache Cassandra!

```
SELECT comment
FROM videos
JOIN comments
ON videos.id = comments.video_id
WHERE title = 'Interstellar'
```

id	title	runtime	year	id	user_id	video_id	comment
2	Interstellar	98	2014	3	a	2	I cried at the end!
2	Interstellar	98	2014	4	b	2	Someone stole my tissues…

Denormalizing For Query Performance

- How Apache Cassandra™ works around no joins



Denormalizing For Query Performance

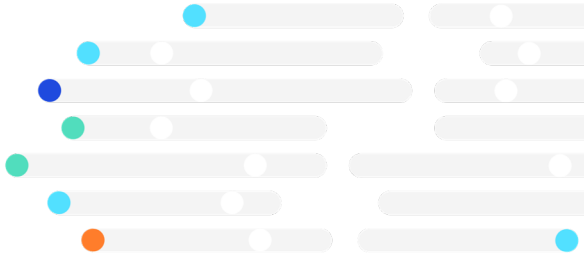
- How Apache Cassandra™ works around no joins

```
CREATE TABLE comments_by_video (
    video_title text,
    comment_id timeuuid,
    user_id text,
    video_id timeuuid,
    comment text,
    PRIMARY KEY ((video_title), comment_id)
);

CREATE TABLE comments_by_user (
    user_login text,
    comment_id timeuuid,
    user_id text,
    video_id timeuuid,
    comment text,
    PRIMARY KEY ((user_login), comment_id)
);
```

Exercise 02.01

DATASTAX

Denormalization

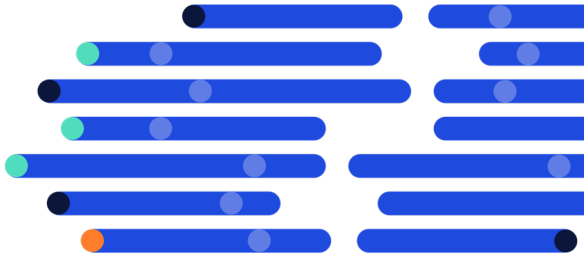


Exercise 02.01 – Denormalization

- In this exercise, you will accomplish the following:
  - Create two tables by denormalizing the data to support your two separate queries
  - Get you comfortable with duplicate data

DATASTAX

Collections, Counters & User Defined Types



Objective for this Module

- Why you should know these other CQL features before you create your data model
- How and why to use collection data types
- How UDTs can be implemented for a more flexible model
- Counter columns and their implications

Some Other Cool CQL Features

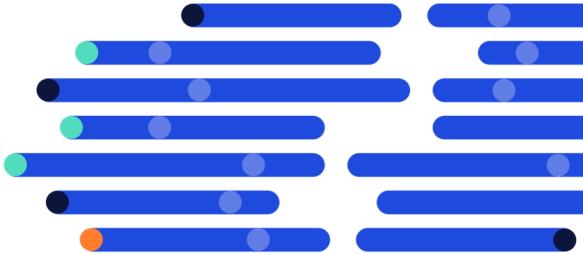
- Knowing what other functionality there is can factor into your design
- Apache Cassandra™ has additional data types for flexibility
- These include:
  - Collections
  - Counters
  - UDTs

Why Do I Care?

- Simplifying table design
- Optimize/streamline table functionality
- Store data more efficiently
- Might change table designs completely

DATASTAX

Collections



Collection Columns

- Group and store data together in a single column
- Collection columns are multi-valued columns
- Designed to store a small amount of data
- Retrieved in its entirety
- Cannot nest a collection inside another collection
  - Unless you use FROZEN
  - More on FROZEN to come!

SET Collection Type

- Typed collection of unique values
- Stored unordered, but retrieved in sorted ordered
- SET example:

```
CREATE TABLE users (
   id text PRIMARY KEY,
   fname text,
   lname text,
   emails set<text>
);

INSERT INTO users (id, fname, lname, emails)
   VALUES ('cass123', 'Cassandra', 'Dev', {'cass@dev.com', 'cassd@gmail.net'});
```

LIST Collection Type

- Like SET—collection of values in same cell
- Do not need to be unique and can be duplicated
- Stored in a particular order
- LIST example:

```
ALTER TABLE users ADD freq_dest list<text>;

UPDATE users SET freq_dest = ['Berlin', 'London', 'Paris']
    WHERE id = 'cass123';
```

MAP Collection Type

- Typed collection of key-value pairs—name and pair of typed values
- Ordered by unique keys
- MAP example:

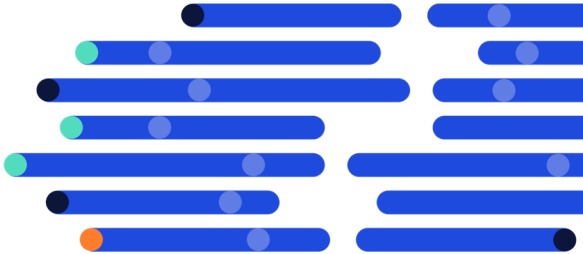
```
ALTER TABLE users ADD todo map<timestamp, text>;

UPDATE users SET todo = { '2018-1-1' : 'create database', '2018-1-2' :
'load data and test' : '2018-2-1’ : 'move to production '}
   WHERE id = 'cass123';
```

Using FROZEN

- If you want to nest datatypes, you must use FROZEN
- Using FROZEN in a collection will serialize multiple components into a single value
- Values in a FROZEN collection are treated like blobs
- Non-frozen types allow updates to individual fields

DATASTAX

User Defined Types



User Defined Types

- User-defined types group related fields of information
- User-defined types (UDTs) can attach multiple data fields, each named and typed, to a single column
- Can be any datatype including collections and other UDTs
- Allows embedding more complex data within a single column

UDT Example

```
CREATE TYPE address (
    street text,
    city text,
    zip_code int,
    phones set<text>
);

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

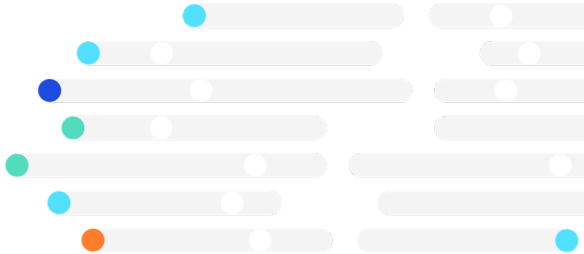
Using a Newly Created UDT

```
CREATE TABLE users (
    id uuid,
    name frozen <full_name>,
    direct_reports set<frozen <full_name>>,
    addresses map<text, frozen <address>>,
    PRIMARY KEY ((id))
);
```

Exercise 02.02

DATASTAX

UDTs

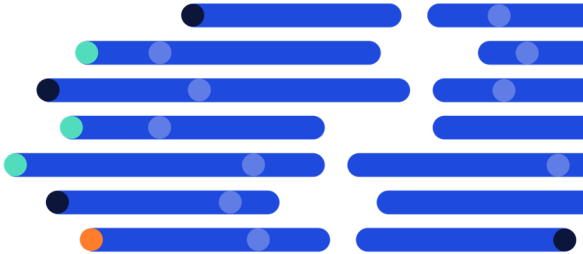


Exercise 02.02 – UDTs

- In this exercise, you will accomplish the following:
  - Create a UDT
  - Alter an existing table to add columns

DATASTAX

Counters



Counters

- Column used to store a 64-bit signed integer
- Changed incrementally—incremented or decremented
- Values are changed using UPDATE
- Need specially dedicated tables—can only have primary key and counter columns
  - Can have more than one counter column

Counter Example

```
CREATE TABLE moo_counts (
    cow_name text,
    moo_count counter,
    PRIMARY KEY((cow_name))
);

UPDATE moo_counts
SET moo_count = moo_count + 8
WHERE cow_name = 'Betsy'
```

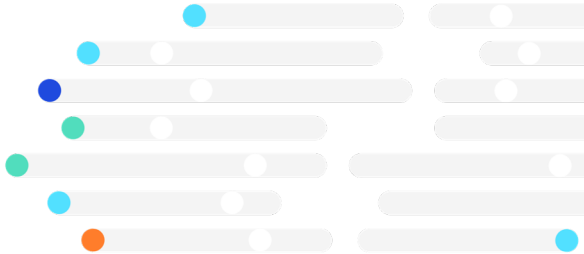
Counter Considerations

- Distributed system can cause consistency issues with counters in some cases
- Cannot INSERT or assign values—default value is “0”
- Must be only non-primary key column(s)
- Not idempotent
- Must use UPDATE command—DataStax Enterprise rejects USING TIMESTAMP or USING TTL to update counter columns
- Counter columns cannot be indexed or deleted

Exercise 02.03

DATASTAX

Counters



Exercise 02.03 – Counters

- Create a new table that makes use of the counter type
- Load the newly created table with data
- Run queries against the table to test counter functionality