

# LAB CASSANDRA



**Professeur : BOUBCHIR, Larbi**

**Module : ADIF82**

# Table des matières

CREATE THE KEYSPACE.....	3
CREATION OF THE TABLE .....	3
INSERTION .....	4
QUERYING.....	4
PHYSICAL STORAGE .....	6
PARTITIONED STORAGE .....	6
IS THIS A SOLUTION? TRY IT .....	7
QUERIES.....	7
CASSANDRA-UPSERTS.....	8
CLUSTERING COLUMNS .....	9
CLUSTERING COLUMN WITH ORDER.....	9
QUERYING CLUSTERING COLUMNS.....	9
ALTER TABLE .....	10
MULTI-VALUED COLUMN.....	10
UDT (USER DEFINED TYPE) .....	11
ALTER TABLE VIDEOS (SET) .....	12
ALTER TABLE VIDEOS (LIST).....	14
ALTER TABLE VIDEOS (MAP) .....	15
UDT .....	16
ALTER TABLE AND ADD INFO .....	17
COUNTER.....	18
Display the result.....	19
Temporal Data .....	22

## CREATE THE KEYSPACE

```
cqlsh> CREATE KEYSPACE demoVideo WITH REPLICATION = { 'class' : 'SimpleStrategy', 'replication_factor' : 1 };
cqlsh> USE demoVideo;
cqlsh:demoVideo> _
```

**CREATE KEYSPACE demoVideo WITH REPLICATION = { 'class' : 'SimpleStrategy', 'replication\_factor' : 1};**

Cette commande crée un espace de noms (keyspace) nommé demoVideo. L'espace de noms est configuré pour utiliser la stratégie de réplication SimpleStrategy, ce qui est généralement utilisé pour les environnements de développement ou de test, avec un facteur de réplication de 1, ce qui signifie qu'il n'y aura pas de copies redondantes de données sur d'autres nœuds.

### USE demoVideo;

Après la création de l'espace de noms, cette commande sélectionne demoVideo comme l'espace de noms actif pour les opérations suivantes. Toutes les commandes CQL subséquentes s'exécuteront dans cet espace de noms jusqu'à ce qu'un autre espace de noms soit sélectionné ou que la session se termine.

## CREATION OF THE TABLE

```
cqlsh:demoVideo> CREATE TABLE videos( id int, name text, runtime int, year int, PRIMARY KEY (id) );
cqlsh:demoVideo> _
```

**CREATE TABLE videos (id int, name text, runtime int, year int, PRIMARY KEY (id));**

Cette commande crée une nouvelle table nommée videos dans l'espace de noms demoVideo qui a été sélectionné auparavant avec la commande USE. La table videos est définie avec les colonnes suivantes :

id int : une colonne pour l'identifiant de la vidéo avec un type de données entier.

name text : une colonne pour le nom de la vidéo avec un type de données texte.

runtime int : une colonne pour la durée de la vidéo en minutes avec un type de données entier.

year int : une colonne pour l'année de sortie de la vidéo avec un type de données entier.

La table a une clé primaire id, ce qui signifie que chaque vidéo sera unique identifiée par son id.

```
cqlsh:demoVideo> describe tables
videos
```

### DESCRIBE TABLES

Cette commande liste toutes les tables qui existent dans l'espace de noms (keyspace) demoVideo. Dans la sortie affichée, videos est la table qui existe dans cet espace de noms. Cela signifie que la table videos est la seule table ou l'une des tables présentes dans demoVideo.

## INSERTION

```
cqlsh:demovideo> CREATE TABLE videos( id int, name text, runtime int, year int, PRIMARY KEY (id) );
cqlsh:demovideo> INSERT INTO videos (id, name, runtime, year) VALUES (1, 'Insurgent', 119, 2015);
cqlsh:demovideo> INSERT INTO videos (id, name, runtime, year) VALUES (2, 'Interstellar', 98, 2014);
cqlsh:demovideo> INSERT INTO videos (id, name, runtime, year) VALUES (3, 'Mockingjay', 122, 2014);
cqlsh:demovideo>
```

**CREATE TABLE videos (id int, name text, runtime int, year int, PRIMARY KEY (id));**

Cette commande crée une nouvelle table nommée videos avec les colonnes id, name, runtime et year. id est désigné comme clé primaire, ce qui implique que chaque entrée dans la table doit avoir un identifiant unique.

**INSERT INTO videos (id, name, runtime, year) VALUES (1, 'Insurgent', 119, 2015);**

**INSERT INTO videos (id, name, runtime, year) VALUES (2, 'Interstellar', 98, 2014);**

**INSERT INTO videos (id, name, runtime, year) VALUES (3, 'Mockingjay', 122, 2014);**

Chacune de ces instructions INSERT ajoute une ligne individuelle à la table videos, en utilisant les valeurs spécifiées pour chaque colonne.

## QUERYING

```
cqlsh:demovideo> SELECT COUNT(*) FROM videos;

count
-----
      3

(1 rows)

Warnings :
Aggregation query used without partition key

cqlsh:demovideo>
```

**SELECT COUNT(\*) FROM videos;**

Cette commande est une requête d'agrégation qui calcule le nombre total de lignes présentes dans la table videos. Le résultat affiché est 3, ce qui signifie qu'il y a trois enregistrements dans la table.

**Warnings : Aggregation query used without partition key**

Ce message est un avertissement de Cassandra. Il indique que la requête d'agrégation a été utilisée sans spécifier de clé de partition, ce qui peut ne pas être optimal pour les performances, en particulier dans les clusters de grande taille, car cela pourrait potentiellement requérir de scanner toutes les partitions de la table à travers tous les nœuds du cluster pour compter les lignes.

```
cqlsh:demovideo> SELECT * FROM videos;
```

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

```
(3 rows)
cqlsh:demovideo> _
```

#### **SELECT \* FROM videos;**

Cette requête récupère toutes les colonnes pour toutes les lignes de la table videos. Les résultats montrent trois lignes avec les colonnes id, name, runtime, et year.

```
cqlsh:demovideo> CREATE INDEX ON videos (name);
cqlsh:demovideo> SELECT * FROM videos WHERE name= 'Insurgent';
```

id	name	runtime	year
1	Insurgent	119	2015

```
(1 rows)
cqlsh:demovideo> _
```

#### **CREATE INDEX ON videos (name);**

Cette commande crée un index sur la colonne name de la table videos. Un index permet d'améliorer la performance des requêtes qui filtrent sur cette colonne. Après la création de cet index, il sera plus rapide de rechercher des vidéos par leur nom.

#### **SELECT \* FROM videos WHERE name = 'Insurgent';**

Après la création de l'index, cette requête sélectionne toutes les colonnes de toutes les lignes de la table videos où la colonne name est égale à 'Insurgent'.

```
cqlsh:demovideo> SELECT * FROM videos WHERE year > 2014 ALLOW FILTERING;
```

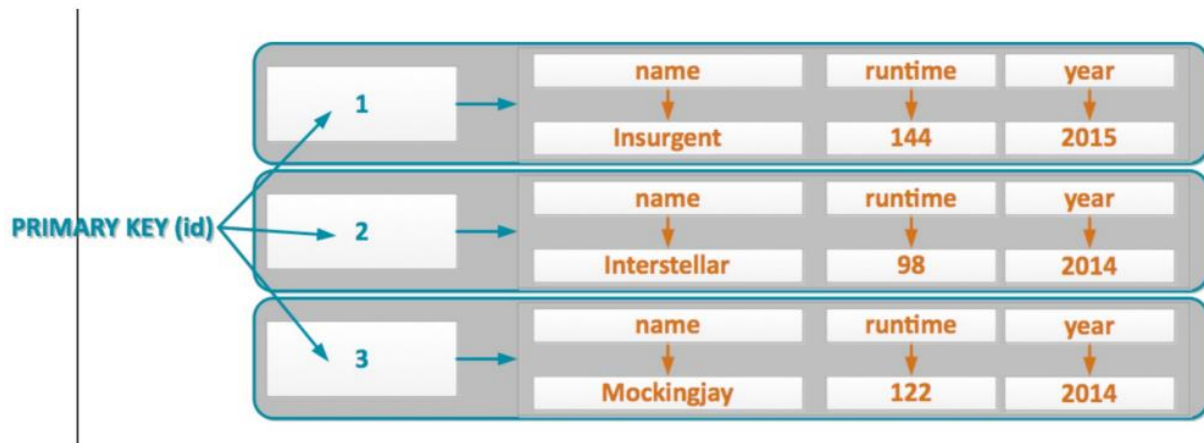
id	name	runtime	year
1	Insurgent	119	2015

```
(1 rows)
cqlsh:demovideo>
```

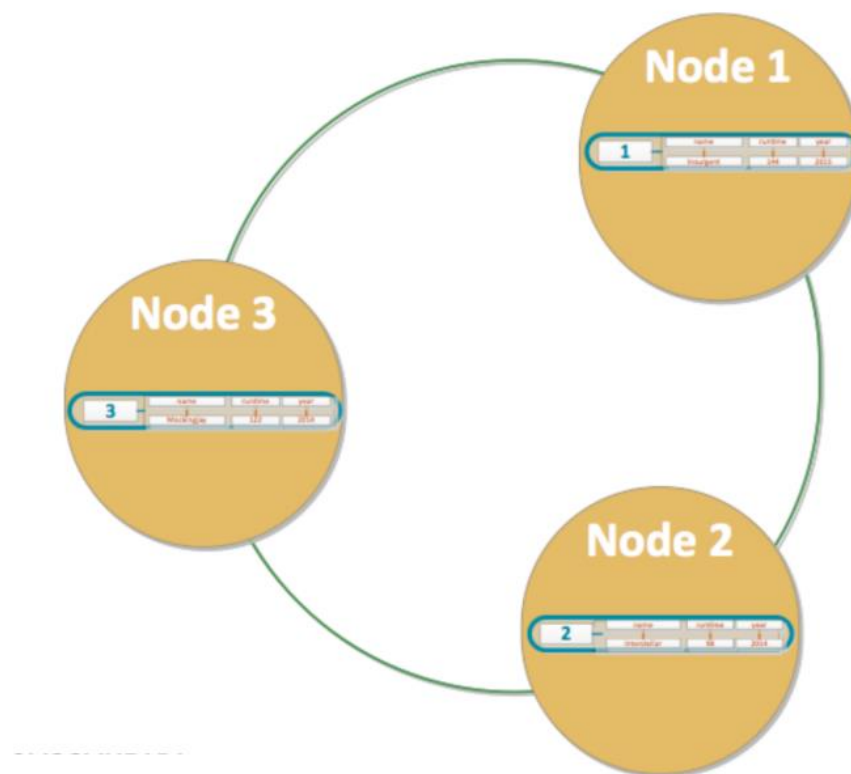
#### **SELECT \* FROM videos WHERE year > 2014 ALLOW FILTERING;**

Cette requête sélectionne toutes les colonnes des enregistrements de la table videos où la colonne year est supérieure à 2014. L'utilisation de ALLOW FILTERING permet d'effectuer des recherches sur des colonnes qui ne sont généralement pas prévues pour le filtrage parce qu'elles ne font pas partie de la clé primaire et qu'il n'y a pas d'index secondaire qui les supporte.

## PHYSICAL STORAGE



## PARTITIONED STORAGE



## IS THIS A SOLUTION? TRY IT

```
cqlsh:demovideo> CREATE TABLE videos_by_name_year(name text, runtime int, year int, PRIMARY KEY ((name, year)));
cqlsh:demovideo>
```

**CREATE TABLE videos\_by\_name\_year(name text, runtime int, year int, PRIMARY KEY ((name, year)));**

Cette commande crée une table nommée videos\_by\_name\_year avec les colonnes name, runtime, et year.

```
cqlsh:demovideo> INSERT INTO videos_by_name_year (name, runtime, year) VALUES ('Insurgent', 119,2015);
cqlsh:demovideo> INSERT INTO videos_by_name_year (name, runtime, year) VALUES ('Interstellar',98,2014);
cqlsh:demovideo> INSERT INTO videos_by_name_year (name, runtime, year) VALUES ('Mockingjay',122,2014);
cqlsh:demovideo>
```

**INSERT INTO videos\_by\_name\_year (name, runtime, year) VALUES ('Insurgent', 119, 2015);**

**INSERT INTO videos\_by\_name\_year (name, runtime, year) VALUES ('Interstellar', 98, 2014);**

**INSERT INTO videos\_by\_name\_year (name, runtime, year) VALUES ('Mockingjay', 122, 2014);**

Chaque commande INSERT ajoute une ligne à la table videos\_by\_name\_year, en utilisant la combinaison de name et year comme clé primaire composite. Cela permettra des requêtes optimisées pour rechercher des vidéos en fonction de leur nom et de leur année de sortie.

```
cqlsh:demovideo> SELECT * FROM videos_by_name_year WHERE year > 2014 ALLOW FILTERING;

name      | year | runtime
-----+-----+-----
Insurgent | 2015 |    119

(1 rows)
cqlsh:demovideo>
```

**SELECT \* FROM videos\_by\_name\_year WHERE year > 2014 ALLOW FILTERING;**

Cette commande sélectionne toutes les colonnes de la table videos\_by\_name\_year où la colonne year est supérieure à 2014.

## QUERIES

```
cqlsh:demovideo> SELECT * FROM videos_by_name_year WHERE name='Insurgent' AND year=2015;

name      | year | runtime
-----+-----+-----
Insurgent | 2015 |    119

(1 rows)
cqlsh:demovideo>
```

**SELECT \* FROM videos\_by\_name\_year WHERE name='Insurgent' AND year=2015;**

Cette requête sélectionne toutes les colonnes pour les lignes où le name est égal à 'Insurgent' et l'year est égal à 2015.

```
cqlsh:demovideo> SELECT * FROM videos_by_name_year WHERE name='Interstellar' ALLOW FILTERING;
```

name	year	runtime
Interstellar	2014	98

```
(1 rows)
cqlsh:demovideo> _
```

**SELECT \* FROM videos\_by\_name\_year WHERE name='Interstellar' ALLOW FILTERING;**

Cette requête récupère toutes les colonnes des enregistrements de la table videos\_by\_name\_year où la colonne name est égale à 'Interstellar'

```
cqlsh:demovideo> SELECT * FROM videos_by_name_year WHERE year=2014 ALLOW FILTERING;
```

name	year	runtime
Interstellar	2014	98
Mockingjay	2014	122

```
(2 rows)
```

**SELECT \* FROM videos\_by\_name\_year WHERE year=2014 ALLOW FILTERING;**

Cette requête récupère toutes les colonnes des enregistrements dans la table où la colonne year est égale à 2014.

## CASSANDRA-UPSERTS

```
cqlsh:demovideo> INSERT INTO videos_by_name_year(name,year,runtime)VALUES('Insurgent',2015,127);
cqlsh:demovideo> SELECT count(*) from videos_by_name_year;
```

count
3

```
(1 rows)

Warnings :
Aggregation query used without partition key
cqlsh:demovideo> _
```

**INSERT INTO videos\_by\_name\_year(name,year,runtime)VALUES('Insurgent', 2015,127);**

Cette commande insère une nouvelle ligne dans la table videos\_by\_name\_year avec le nom 'Insurgent', l'année 2015, et une durée de 127 minutes. Si une ligne avec le nom 'Insurgent' et l'année 2015 existe déjà, elle sera remplacée par cette nouvelle ligne à cause de la nature de la clé primaire composite (name, year).

**SELECT count(\*) FROM videos\_by\_name\_year;**

Cette requête compte le nombre total de lignes dans la table videos\_by\_name\_year



## CLUSTERING COLUMNS

```
cqlsh:demovideo> CREATE TABLE videos_by_year (id int, name text, runtime int, year int, PRIMARY KEY ((year),name));
cqlsh:demovideo>
```

**CREATE TABLE videos\_by\_year (id int, name text, runtime int, year int, PRIMARY KEY ((year), name));**

Cette commande crée une table nommée videos\_by\_year avec les colonnes suivantes : id, name, runtime, et year.

## CLUSTERING COLUMN WITH ORDER

```
cqlsh:demovideo> DROP TABLE videos_by_year;
cqlsh:demovideo> _
```

**DROP TABLE videos\_by\_year;**

Cette commande supprime la table videos\_by\_year de l'espace de noms actuel (demoVideo), ce qui signifie que la table elle-même, ainsi que toutes ses données, seront supprimées de manière permanente de la base de données.

```
cqlsh:demovideo> CREATE TABLE videos_by_year (id int, name text, runtime int, year int, PRIMARY KEY ((year),name)) WITH CLUSTERING ORDER BY (name DESC);
cqlsh:demovideo> _
```

**CREATE TABLE videos\_by\_year (id int, name text, runtime int, year int, PRIMARY KEY ((year), name)) WITH CLUSTERING ORDER BY (name DESC);**

Cette commande crée une table appelée videos\_by\_year avec les colonnes id, name, runtime, et year. La clé primaire est composée d'une clé de partition year et d'une clé de clustering name. La spécification WITH CLUSTERING ORDER BY (name DESC) indique que les données à l'intérieur de chaque partition de year seront ordonnées par name en ordre décroissant.

## QUERYING CLUSTERING COLUMNS

```
cqlsh:demovideo> SELECT * FROM videos_by_year WHERE year = 2014 AND name = 'Mockingjay';

 year | name       | id | runtime
-----+-----+----+-----
  2014 | Mockingjay |  3 |     113

(1 rows)
cqlsh:demovideo>
```

**SELECT \* FROM videos\_by\_year WHERE year = 2014 AND name = 'Mockingjay';**

Cette requête récupère toutes les colonnes pour l'enregistrement dans la table videos\_by\_year où la colonne year est égale à 2014 et la colonne name est égale à 'Mockingjay'

```
cqlsh:demovideo> SELECT * FROM videos_by_year WHERE year = 2014 AND name >= 'Interstellar';
```

year	name	id	runtime
2014	Mockingjay	3	113
2014	Interstellar	2	98

```
(2 rows)  
cqlsh:demovideo> _
```

**SELECT \* FROM videos\_by\_year WHERE year = 2014 AND name >= 'Interstellar';**

Cette requête sélectionne toutes les colonnes des enregistrements de la table `videos_by_year` pour l'année 2014 où le nom est alphabétiquement supérieur ou égal à 'Interstellar'.

## ALTER TABLE

```
cqlsh:demovideo> ALTER TABLE videos_by_year ADD another_column text;  
cqlsh:demovideo> _
```

**ALTER TABLE videos\_by\_year ADD another\_column text;**

Cette commande modifie la table `videos_by_year` en ajoutant une nouvelle colonne nommée `another_column` de type `text`.

```
cqlsh:demovideo> ALTER TABLE videos_by_year DROP another_column;  
cqlsh:demovideo> _
```

**ALTER TABLE videos\_by\_year DROP another\_column;**

Cette commande modifie la table `videos_by_year` en supprimant la colonne `another_column`.

```
cqlsh:demovideo> TRUNCATE videos_by_year;  
cqlsh:demovideo> _
```

**TRUNCATE videos\_by\_year;**

Cette instruction CQL est utilisée pour supprimer toutes les données de la table `videos_by_year`, mais pas la table elle-même. Après l'exécution de cette commande, la table restera dans le schéma de la base de données, mais elle sera vide. Cette opération est utile pour réinitialiser rapidement une table en supprimant toutes les données sans avoir à supprimer et recréer la table.

## MULTI-VALUED COLUMN

```
cqlsh:demovideo> ALTER TABLE videos_by_year ADD actors SET <TEXT>;  
cqlsh:demovideo> _
```

**ALTER TABLE videos\_by\_year ADD actors SET<TEXT>;**

Cette commande modifie la structure de la table `videos_by_year` en y ajoutant une nouvelle colonne nommée `actors`. Cette nouvelle colonne est de type `SET<TEXT>`, ce qui signifie qu'elle peut stocker un ensemble de valeurs textuelles uniques. Les ensembles sont des collections non ordonnées de valeurs uniques, ce qui est utile pour stocker des données comme une liste d'acteurs sans doublons pour chaque film.

```
cqlsh:demovideo> ALTER TABLE videos_by_year ADD reviews LIST <TEXT>;  
cqlsh:demovideo> _
```

**ALTER TABLE videos\_by\_year ADD reviews LIST<TEXT>;**

Cette commande modifie la structure de la table videos\_by\_year en y ajoutant une nouvelle colonne nommée reviews. Cette nouvelle colonne est de type LIST<TEXT>, ce qui signifie qu'elle peut stocker une liste ordonnée de valeurs textuelles.

```
cqlsh:demovideo> ALTER TABLE videos_by_year ADD rating MAP <TEXT, INT>;  
cqlsh:demovideo> _
```

**ALTER TABLE videos\_by\_year ADD rating MAP<TEXT, INT>;**

Cette commande modifie la structure de la table videos\_by\_year en ajoutant une nouvelle colonne nommée rating. Le type de cette colonne est défini comme une MAP avec des clés de type TEXT et des valeurs de type INT. En d'autres termes, cela permet de stocker une collection de paires clé/valeur où la clé est une chaîne de caractères (texte) et la valeur est un entier (int).

## UDT (USER DEFINED TYPE)

Les UDT sont utiles pour regrouper des données qui sont souvent utilisées ensemble dans des colonnes de table, permettant ainsi une meilleure organisation et une structure plus claire de la base de données.

```
cqlsh:demovideo> CREATE TYPE address(street text, city text, zip_code int, phones set<text>;  
cqlsh:demovideo> _
```

**CREATE TYPE address(street text, city text, zip\_code int, phones set<text>;**

Cette commande crée un UDT nommé address. Ce type est composé de plusieurs champs: street et city qui sont des chaînes de texte (text), zip\_code qui est un entier (int), et phones qui est un ensemble (set) de chaînes de texte.

```
cqlsh:demovideo> CREATE TYPE full_name(first_name text, last_name text);  
cqlsh:demovideo>
```

**CREATE TYPE full\_name(first\_name text, last\_name text);**

Cette commande définit un UDT nommé full\_name qui contient deux champs : first\_name et last\_name, tous deux de type text. Cet UDT peut être utilisé pour stocker des noms complets de personnes, avec un prénom et un nom de famille, dans les colonnes des tables de la base de données.

```
cqlsh:demovideo> DESCRIBE TYPE address;

CREATE TYPE demovideo.address (
    street text,
    city text,
    zip_code int,
    phones set<text>
);
cqlsh:demovideo> DESCRIBE TYPE full_name;

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

#### **DESCRIBE TYPE address;**

Cette commande affiche la structure du UDT address que nous avons précédemment créé. Il montre que address comprend les champs street, city, zip\_code, et phones, avec leurs types correspondants (text, int, set<text>).

#### **DESCRIBE TYPE full\_name;**

Cette commande affiche la structure du UDT full\_name. Il montre que full\_name est composé des champs first\_name et last\_name, tous deux de type text.

### ALTER TABLE VIDEOS (SET)

```
cqlsh:demovideo> ALTER TABLE videos ADD tags SET<TEXT>;
cqlsh:demovideo>
```

#### **ALTER TABLE videos ADD tags SET<TEXT>;**

Cette commande modifie la table videos en ajoutant une nouvelle colonne nommée tags. Le type de cette colonne est un SET<TEXT>, ce qui signifie qu'elle peut contenir une collection de chaînes de texte uniques.

```
cqlsh:demovideo> DESCRIBE TABLE videos

CREATE TABLE demovideo.videos (
    id int PRIMARY KEY,
    name text,
    runtime int,
    year int,
    tags set<text>
);
```

#### **DESCRIBE TABLE videos;**

Après l'exécution de cette commande, la sortie affiche la définition de la table videos dans le schéma de la base de données.

```
cqlsh:demovideo> INSERT INTO videos (id, name, runtime, tags) VALUES (1, 'Avengers',120, {'tag1','tag2'});
cqlsh:demovideo>
```

#### **INSERT INTO videos (id, name, runtime, tags) VALUES (1, 'Avengers', 120, {'tag1', 'tag2'});**

Cette commande insère un nouvel enregistrement dans la table videos

```
cqlsh:demovideo> UPDATE videos SET tags = tags + {'tag3'} WHERE id=1;
cqlsh:demovideo> _
```

**UPDATE videos SET tags = tags + {'tag3'} WHERE id=1;**

Cette commande met à jour un enregistrement existant dans la table videos. Elle cible la colonne tags et y ajoute un nouvel élément, 'tag3', à l'ensemble de tags existants. L'enregistrement à mettre à jour est identifié par la condition WHERE id=1, ce qui signifie que seule la ligne où id est égal à 1 sera mise à jour. (tags = tags + {'tag3'}) est utilisée pour ajouter un élément à un ensemble (set) sans supprimer les éléments existants.

```
cqlsh:demovideo> UPDATE videos SET tags = tags - {'tag1'} WHERE id=1;
cqlsh:demovideo> _
```

**UPDATE videos SET tags = tags - {'tag1'} WHERE id=1;**

Cette commande est utilisée pour retirer un élément d'une colonne de type ensemble (set) dans la table videos. tags = tags - {'tag1'} indique que l'élément 'tag1' doit être retiré de l'ensemble de tags existant pour la colonne tags. La condition WHERE id=1 spécifie que la modification doit être appliquée à l'enregistrement dont la clé primaire id vaut 1.

```
cqlsh:demovideo> SELECT * FROM videos WHERE id=1;
```

id	name	runtime	tags	year
1	Insurgent	119	{'tag3'}	2015

**SELECT \* FROM videos WHERE id=1;**

Cette commande récupère et affiche toutes les colonnes (\*) pour l'enregistrement où la clé primaire (id) est 1. Dans ce cas, il y a un seul tag ('tag3') associé à la vidéo, qui peut refléter les modifications apportées précédemment par les instructions UPDATE.

```
cqlsh:demovideo> DELETE tags FROM videos WHERE id=1;
cqlsh:demovideo> _
```

**DELETE tags FROM videos WHERE id=1;**

Cette commande supprime spécifiquement la colonne tags pour l'enregistrement dans la table videos où la clé primaire id est égale à 1.

```
cqlsh:demovideo> SELECT * FROM videos WHERE id=1;
```

id	name	runtime	tags	year
1	Insurgent	119	null	2015

**SELECT \* FROM videos WHERE id=1;**

La valeur null dans la colonne tags indique que la colonne a été vidée suite à la commande DELETE tags FROM videos WHERE id=1; précédente, conformément à ce qu'on s'attendrait à voir après une telle opération.

## ALTER TABLE VIDEOS (LIST)

```
cqlsh:demovideo> ALTER TABLE videos ADD artists LIST<TEXT>;  
cqlsh:demovideo>
```

**ALTER TABLE videos ADD artists LIST<TEXT>;**

Cette commande modifie la table videos en ajoutant une nouvelle colonne nommée artists. Le type de cette colonne est LIST<TEXT>, ce qui indique qu'elle peut contenir une liste ordonnée de chaînes de texte. Contrairement à un SET, une LIST peut contenir des doublons et l'ordre des éléments est préservé.

```
cqlsh:demovideo> DESCRIBE TABLE videos  
  
CREATE TABLE demovideo.videos (  
    id int PRIMARY KEY,  
    name text,  
    runtime int,  
    year int,  
    artists list<text>,  
    tags set<text>
```

**DESCRIBE TABLE videos;**

Cette commande est utilisée pour afficher le schéma actuel de la table, montrant toutes les colonnes et leurs types de données respectifs.

```
cqlsh:demovideo> INSERT INTO videos (id, name, runtime, artists) VALUES (123, 'Iron Man', 120, ['A1', 'A2']);  
cqlsh:demovideo> _
```

**INSERT INTO videos (id, name, runtime, artists) VALUES (123, 'Iron Man', 120, ['A1', 'A2']);**

Cette commande insère un nouvel enregistrement dans la table videos.

```
cqlsh:demovideo> UPDATE videos SET artists = ['A3'] WHERE id=1;  
cqlsh:demovideo> _
```

**UPDATE videos SET artists = ['A3'] WHERE id=1;**

Cette commande met à jour l'enregistrement dans la table videos où la clé primaire id est égale à 1. Elle remplace la liste actuelle des artists par une nouvelle liste contenant un seul élément, 'A3'.

```
cqlsh:demovideo> SELECT * FROM videos WHERE id=1;
```

id	artists	name	runtime	tags	year
1	['A3']	Insurgent	119	null	2015

```
SELECT * FROM videos WHERE id=1;
```

La colonne artists montre que la liste a été mise à jour et contient maintenant seulement l'élément 'A3', ce qui est conforme à la commande UPDATE exécutée précédemment. La colonne tags est toujours null, comme cela avait été défini par une commande DELETE antérieure.

```
cqlsh:demovideo> DELETE artists[0] FROM videos WHERE id=1;
cqlsh:demovideo> _
```

```
DELETE artists[0] FROM videos WHERE id=1;
```

Cette commande supprime l'élément à l'index 0 de la liste artists pour l'enregistrement dans la table videos où l'id est égal à 1. Après l'exécution de cette commande, le premier élément de la liste artists, qui était 'A3', sera supprimé. S'il y avait d'autres éléments dans la liste, ils seraient décalés vers le bas (l'élément à l'index 1 deviendrait l'élément à l'index 0, et ainsi de suite). Si 'A3' était le seul élément de la liste, la liste deviendrait vide.

```
cqlsh:demovideo> SELECT * FROM videos WHERE id=1;
```

id	artists	name	runtime	tags	year
1	null	Insurgent	119	null	2015

```
SELECT * FROM videos WHERE id=1;
```

La sortie indique que la colonne artists est désormais null pour l'enregistrement avec l'id 1 dans la table videos. Cela confirme que l'opération de suppression de l'élément à l'index 0 de la liste artists, exécutée précédemment, a réussi et qu'il n'y a plus d'éléments restants dans la liste, ce qui a conduit à ce que la liste soit considérée comme null. Les autres colonnes (name, runtime, tags, year) restent inchangées avec leurs valeurs précédentes.

## ALTER TABLE VIDEOS (MAP)

```
cqlsh:demovideo> ALTER TABLE videos ADD realisateurs MAP<TEXT, TEXT>;
cqlsh:demovideo>
```

```
ALTER TABLE videos ADD realisateurs MAP<TEXT, TEXT>;
```

Cette commande modifie la table videos en y ajoutant une nouvelle colonne nommée realisateurs. Le type de cette colonne est défini comme une MAP avec des clés de type TEXT et des valeurs de type TEXT également.

```
cqlsh:demovideo> UPDATE videos SET realiseurs = {'nom':'Dupont'} WHERE id=1;
cqlsh:demovideo>
```

**UPDATE videos SET realiseurs = {'nom': 'Dupont'} WHERE id=1;**

Cette commande met à jour l'enregistrement dans la table videos pour l'id 1 en modifiant la colonne realiseurs. La colonne est définie avec une map ayant une paire clé-valeur, où la clé est 'nom' et la valeur associée est 'Dupont'.

```
cqlsh:demovideo> UPDATE videos SET realiseurs = realiseurs + {'prenom' : 'Jean'} WHERE id =1;
cqlsh:demovideo>
```

**UPDATE videos SET realiseurs = realiseurs + {'prenom' : 'Jean'} WHERE id=1;**

Cette commande met à jour l'enregistrement avec l'id 1 dans la table videos. Elle ajoute à la colonne realiseurs une nouvelle paire clé-valeur sans supprimer les existantes. La clé est 'prenom' et la valeur associée est 'Jean'.

```
cqlsh:demovideo> UPDATE videos SET realiseurs['Jean']='Luc' WHERE id=1;
cqlsh:demovideo>
```

**UPDATE videos SET realiseurs['Jean'] = 'Luc' WHERE id=1;**

Cette commande met à jour la valeur associée à la clé 'Jean' dans la colonne realiseurs de la table videos pour l'enregistrement dont l'id est 1. Elle change cette valeur en 'Luc'.

```
cqlsh:demovideo> SELECT * FROM videos WHERE id=1;

id | artists | encoding | name      | realiseurs                                     | runtime | tags | year
---+-----+-----+-----+-----+-----+-----+-----
1  | null    | null     | Avengers | {'Jean': 'Luc', 'nom': 'Dupont', 'prenom': 'Jean'} | 120     | null | 2015

(1 rows)
cqlsh:demovideo>
```

**SELECT \* FROM videos WHERE id=1;**

La colonne realiseurs montre que les mises à jour précédentes de la map ont été appliquées avec succès, montrant à la fois les modifications et les ajouts des paires clé-valeur.

## UDT

```
cqlsh:demovideo> CREATE TYPE video_encoding (encoding text, height int, width int, bit_rates set<text>);
```

**CREATE TYPE video\_encoding (encoding text, height int, width int, bit\_rates set<text>);**

Cette commande crée un nouveau type de données utilisateur (UDT) nommé video\_encoding. Ce type est composé de plusieurs champs : encoding qui est une chaîne de texte (text), height et width qui sont des entiers (int), et bit\_rates qui est un ensemble (set) de chaînes de texte.




```
cqlsh:demovideo> DESCRIBE TYPE video_encoding;

CREATE TYPE demovideo.video_encoding (
    encoding text,
    height int,
    width int,
    bit_rates set<text>
);
```

**DESCRIBE TYPE video\_encoding;**

La sortie affiche la définition du type de données utilisateur (UDT) video\_encoding dans la base de données

 video\_encoding.csv

---video\_encoding.csv---

```
1,"{encoding: '1080p', height: 1080, width: 1920, bit_rates: {'3000 Kbps', '4500 Kbps', '6000 Kbps'}}"
2,"{encoding: '720p', height: 720, width: 1280, bit_rates: {'2000 Kbps', '3500 Kbps', '5000 Kbps'}}"
3,"{encoding: '480p', height: 480, width: 854, bit_rates: {'1000 Kbps', '2000 Kbps', '3000 Kbps'}}"
4,"{encoding: '360p', height: 360, width: 640, bit_rates: {'500 Kbps', '1000 Kbps', '1500 Kbps'}}"
5,"{encoding: '240p', height: 240, width: 426, bit_rates: {'250 Kbps', '500 Kbps', '750 Kbps'}}"
```

## ALTER TABLE AND ADD INFO

```
cqlsh:demovideo> ALTER TABLE videos ADD encoding frozen<video_encoding>;
cqlsh:demovideo> _
```

**ALTER TABLE videos ADD encoding frozen<video\_encoding>;**

Cette commande modifie la table videos en ajoutant une nouvelle colonne nommée encoding. Le type de cette colonne est un type de données utilisateur (UDT) video\_encoding qui a été précédemment défini. L'utilisation du mot-clé frozen indique que les valeurs de cette colonne seront stockées comme une seule valeur immuable.

```
cqlsh:demovideo> DESCRIBE TABLE videos

CREATE TABLE demovideo.videos (
    id int PRIMARY KEY,
    name text,
    runtime int,
    year int,
    artists list<text>,
    encoding video_encoding,
    realisateurs map<text, text>,
    tags set<text>
```

**DESCRIBE TABLE videos**

Cette commande est utilisée pour afficher la définition de la table videos.

```
cqlsh:demovideo> COPY videos (id,encoding) FROM 'video_encoding.csv' WITH HEADER = false
... ;
Using 7 child processes

Starting copy of demovideo.videos with columns [id, encoding].
Processed: 5 rows; Rate:      9 rows/s; Avg. rate:      13 rows/s
5 rows imported from 1 files in 0.391 seconds (0 skipped).
```

### **COPY videos (id, encoding) FROM 'video\_encoding.csv' WITH HEADER = false**

Cette commande est utilisée dans cqlsh pour importer des données dans la table videos du keyspace demovideo à partir d'un fichier CSV nommé video\_encoding.csv.

```
cqlsh:demovideo> select * from videos ;
```

id	artists	encoding	runtime	tags	year	name	realisateurs
5	null	{encoding: '240p', height: 240, width: null, bit_rates: {'250 Kbps', '500 Kbps', '750 Kbps'}}	null	null	null	null	null
1	null	{encoding: '1080p', height: 1080, width: null, bit_rates: {'3000 Kbps', '4500 Kbps', '6000 Kbps'}}	119	null	2015	Insurgent	{'nom': 'machin', 'prenom': 'Jean'}
2	null	{encoding: '720p', height: 720, width: null, bit_rates: {'2000 Kbps', '3500 Kbps', '5000 Kbps'}}	98	null	2014	Interstellar	null
4	null	{encoding: '360p', height: 360, width: null, bit_rates: {'1000 Kbps', '1500 Kbps', '500 Kbps'}}	null	null	null	null	null
3	null	{encoding: '480p', height: 480, width: null, bit_rates: {'1000 Kbps', '2000 Kbps', '3000 Kbps'}}	122	null	2014	Mockingjay	null

### **SELECT \* FROM videos**

Cette commande sélectionne toutes les colonnes de toutes les lignes de la table videos

## COUNTER

```
cqlsh:demovideo> CREATE TABLE videos_count_by_tag (tag TEXT, added_year INT, video_count counter, PRIMARY KEY (tag, added_year));
cqlsh:demovideo> _
```

### **CREATE TABLE videos\_count\_by\_tag (tag TEXT, added\_year INT, video\_count counter, PRIMARY KEY (tag, added\_year));**

Cette commande est utilisée pour créer une nouvelle table videos\_count\_by\_tag.

```
cqlsh:demovideo> DESCRIBE TABLE videos_count_by_tag

CREATE TABLE demovideo.videos_count_by_tag (
  tag text,
  added_year int,
  video_count counter,
  PRIMARY KEY (tag, added_year)
) WITH CLUSTERING ORDER BY (added_year ASC)
```

### **DESCRIBE TABLE videos\_count\_by\_tag**

Cette commande est utilisée pour afficher la définition de la table videos\_count\_by\_tag

```
cqlsh:demovideo> UPDATE videos_count_by_tag SET video_count = video_count + 1 WHERE tag = 'MyTag' AND added_year=2015;
cqlsh:demovideo>
cqlsh:demovideo> UPDATE videos_count_by_tag SET video_count = video_count + 1 WHERE tag='AnotherTag' AND added_year=2020;
cqlsh:demovideo>
cqlsh:demovideo> UPDATE videos_count_by_tag SET video_count = video_count +1 WHERE tag = 'MyTag' AND added_year=2015;
cqlsh:demovideo>
cqlsh:demovideo> UPDATE videos_count_by_tag SET video_count = video_count + 1 WHERE tag = 'NewTag' AND added_year = 2021;
cqlsh:demovideo> _
```

### **UPDATE videos\_count\_by\_tag SET video\_count = video\_count + 1 WHERE tag = 'MyTag' AND added\_year=2015;**

Cette commande augmente de 1 le compteur pour le tag 'MyTag' pour l'année 2015.

### **UPDATE videos\_count\_by\_tag SET video\_count = video\_count + 1 WHERE tag='AnotherTag' AND added\_year=2020;**

Similaire à la première, mais pour le tag 'AnotherTag' et l'année 2020.

**UPDATE videos\_count\_by\_tag SET video\_count = video\_count + 1 WHERE tag = 'MyTag' AND added\_year=2015;**

Répète l'augmentation du compteur pour le même tag 'MyTag' et la même année 2015, ce qui suggère que c'est peut-être une répétition ou une autre vidéo ajoutée.

**UPDATE videos\_count\_by\_tag SET video\_count = video\_count + 1 WHERE tag = 'NewTag' AND added\_year = 2021;**

Incrémente le compteur pour un nouveau tag 'NewTag' pour l'année 2021.

```
cqlsh:demovideo> SELECT * FROM videos_count_by_tag;
```

tag	added_year	video_count
MyTag	2015	2
AnotherTag	2020	1
NewTag	2021	1

```
(3 rows)
cqlsh:demovideo>
```

**SELECT \* FROM videos\_count\_by\_tag;**

Cette commande est utilisée pour interroger toutes les colonnes pour tous les enregistrements de la table videos\_count\_by\_tag

```
cqlsh:demovideo> UPDATE videos_count_by_tag SET video_count = video_count + 1 WHERE tag = 'MyTag' AND added_year=2015;
cqlsh:demovideo>
```

**UPDATE videos\_count\_by\_tag SET video\_count = video\_count + 1 WHERE tag = 'MyTag' AND added\_year=2015;**

Cette commande augmente de 1 le compteur video\_count pour les enregistrements où tag est égal à 'MyTag' et added\_year est égal à 2015

## Display the result

```
cqlsh:demovideo> UPDATE videos_count_by_tag SET video_count = video_count + 1 WHERE tag='MyTag' AND added_year=2015;
cqlsh:demovideo> SELECT * FROM videos_count_by_tag ;
```

tag	added_year	video_count
MyTag	2015	3
AnotherTag	2020	1
NewTag	2021	1

**SELECT \* FROM videos\_count\_by\_tag;**

Le résultat montre que le tag 'MyTag' pour l'année 2015 a maintenant un compteur de 3, ce qui signifie que cette dernière commande UPDATE a été exécutée avec succès, augmentant le compteur de 2 à 3.

```
cqlsh:demovideo> UPDATE videos_count_by_tag SET video_count = video_count + 1 WHERE tag = 'MyTagNotExist' AND added_year=2015;  
cqlsh:demovideo> _
```

**UPDATE videos\_count\_by\_tag SET video\_count = video\_count + 1 WHERE tag = 'MyTagNotExist' AND added\_year=2015;**

Cette commande tente d'incrémenter de 1 le compteur video\_count pour le tag 'MyTagNotExist' pour l'année 2015. Si le tag 'MyTagNotExist' n'existe pas déjà dans la table avec l'année 2015, Cassandra créera un nouvel enregistrement avec le compteur video\_count initialisé à 1 pour cette combinaison de tag et d'année, car les compteurs dans Cassandra sont par défaut initialisés à 0 lors de la première mise à jour.

```
cqlsh:demovideo> SELECT * FROM videos_count_by_tag;
```

tag	added_year	video_count
MyTagNotExist	2015	1
MyTag	2015	3
AnotherTag	2020	1
NewTag	2021	1

(4 rows)

**SELECT \* FROM videos\_count\_by\_tag;**

Le résultat montre quatre lignes, indiquant que la table contient maintenant les compteurs de vidéos pour quatre tags différents pour des années spécifiques :

'MyTagNotExist' pour l'année 2015 avec un compteur de vidéo de 1. Cela correspond à la dernière mise à jour que nous avons vue, où un compteur pour un tag qui n'existait pas auparavant a été incrémenté.

'MyTag' pour 2015 avec un compteur de 3.

'AnotherTag' pour 2020 avec un compteur de 1.

'NewTag' pour 2021 avec un compteur de 1.

```
cqlsh:demovideo> UPDATE videos_count_by_tag SET video_count = video_count + 1 WHERE tag='MyTag' AND added_year=2026;  
cqlsh:demovideo> _
```

**UPDATE videos\_count\_by\_tag SET video\_count = video\_count + 1 WHERE tag='MyTag' AND added\_year=2026;**

Cette commande est conçue pour incrémenter de 1 le compteur video\_count pour les enregistrements où le tag est égal à 'MyTag' et l'added\_year est égal à 2026. Si un enregistrement avec ce tag et cette année n'existe pas déjà, il sera créé avec video\_count initialisé à 1 en raison de cette mise à jour.

```
cqlsh:demovideo> SELECT * FROM videos_count_by_tag;
```

tag	added_year	video_count
MyTagNotExist	2015	1
MyTag	2015	3
MyTag	2026	1
AnotherTag	2020	1
NewTag	2021	1

```
(5 rows)
```

**SELECT \* FROM videos\_count\_by\_tag**

Le résultat montre cinq lignes :

'MyTagNotExist' pour l'année 2015 a un compteur de vidéo de 1.

'MyTag' pour 2015 a un compteur de 3.

'MyTag' pour 2026 a maintenant un compteur de 1, ce qui indique que la commande UPDATE pour ajouter un compteur à 'MyTag' pour 2026 a été exécutée avec succès.

'AnotherTag' pour 2020 a un compteur de 1.

'NewTag' pour 2021 a également un compteur de 1.

```
cqlsh:demovideo> UPDATE videos_count_by_tag SET video_count = video_count + 1 WHERE tag = 'NOTEXISTING' AND added_year=2050;
cqlsh:demovideo> _
```

**UPDATE videos\_count\_by\_tag SET video\_count = video\_count + 1 WHERE tag = 'NOTEXISTING' AND added\_year=2050;**

Cette commande tente d'incrémenter de 1 le compteur video\_count pour le tag 'NOTEXISTING' pour l'année 2050. Si aucun enregistrement correspondant à cette combinaison de tag et d'année n'existe, Cassandra créera un nouvel enregistrement avec video\_count initialisé à 1 suite à cette mise à jour.

```
cqlsh:demovideo> SELECT * FROM videos_count_by_tag;
```

tag	added_year	video_count
MyTagNotExist	2015	1
NOTEXISTING	2050	1
MyTag	2015	3
MyTag	2026	1
AnotherTag	2020	1
NewTag	2021	1

```
(6 rows)
cqlsh:demovideo> _
```

**SELECT \* FROM videos\_count\_by\_tag**

Le résultat montre six lignes, indiquant que la table contient maintenant les compteurs de vidéos pour les tags suivants :

'MyTagNotExist' pour l'année 2015 a un compteur de vidéo de 1.

'NOTEXISTING' pour l'année 2050 a été ajouté avec un compteur de vidéo de 1, ce qui correspond à la commande UPDATE précédente pour un tag non existant pour l'année 2050.

'MyTag' pour 2015 avec un compteur de 3.

'MyTag' pour 2026 avec un compteur de 1.

'AnotherTag' pour 2020 avec un compteur de 1.

'NewTag' pour 2021 avec un compteur de 1.

## Temporal Data

```
cqlsh:demovideo> CREATE TABLE user (id int primary key, name text);
cqlsh:demovideo> INSERT INTO user (id,name) VALUES (1, 'user 1');
cqlsh:demovideo> INSERT INTO user (id,name) VALUES (2, 'user 2') using TIMESTAMP 10;
cqlsh:demovideo> INSERT INTO user (id, name) VALUES (3, 'user 3');
cqlsh:demovideo> SELECT * FROM user;
```

id	name
1	user 1
2	user 2
3	user 3

```
(3 rows)
cqlsh:demovideo> SELECT id,name, writetime(name) FROM user;
```

id	name	writetime(name)
1	user 1	1706258219812529
2	user 2	10
3	user 3	1706258269584759

```
(3 rows)
cqlsh:demovideo>
```

**CREATE TABLE user (id int primary key, name text);**

Cette commande crée une nouvelle table nommée user avec deux colonnes : id qui est une clé primaire de type entier, et name de type texte.

**INSERT INTO user (id, name) VALUES (1, 'user 1');**

Insère un enregistrement avec id 1 et name 'user 1'.

**INSERT INTO user (id, name) VALUES (2, 'user 2') USING TIMESTAMP 10;**

Insère un enregistrement avec id 2 et name 'user 2' et utilise un timestamp personnalisé pour l'opération d'insertion (équivalent à 10 en unités de temps Unix).

**INSERT INTO user (id, name) VALUES (3, 'user 3');**

Insère un enregistrement avec id 3 et name 'user 3'.

**SELECT \* FROM user;**

Sélectionne et affiche toutes les colonnes pour tous les enregistrements de la table user. Le résultat montre trois utilisateurs avec leurs identifiants respectifs.

```
SELECT id, name, writetime(name) FROM user;
```

Sélectionne l'id, le name et le timestamp de la dernière mise à jour de la colonne name pour tous les enregistrements de la table user. Les résultats montrent les timestamps pour chaque name. Pour l'id 2, le timestamp est étonnamment bas (10), ce qui indique que le timestamp a été manuellement défini lors de l'insertion

```
cqlsh:demovideo> UPDATE user USING TIMESTAMP 11 set name = 'user 4' WHERE id=2;
cqlsh:demovideo> SELECT id,name, writetime(name) FROM user;
```

id	name	writetime(name)
1	user 1	1706258219812529
2	user 4	11
3	user 3	1706258269584759

(3 rows)

```
cqlsh:demovideo> UPDATE user USING TIMESTAMP 12 SET name = 'user 5' WHERE id = 2;
cqlsh:demovideo> SELECT id, name, writetime(name) FROM user;
```

id	name	writetime(name)
1	user 1	1706258219812529
2	user 5	12
3	user 3	1706258269584759

(3 rows)

```
cqlsh:demovideo> _
```

```
SELECT id, name, writetime(name) FROM user;
```

Cette requête affiche l'identifiant, le nom, et le timestamp de la dernière modification pour la colonne name. Le résultat montre que l'utilisateur avec id 2 a été mis à jour avec un timestamp manuel de 11, ce qui a modifié le nom de 'user 2' à 'user 4'.

```
UPDATE user USING TIMESTAMP 12 SET name = 'user 5' WHERE id = 2;
```

Cette commande met à jour le nom de l'utilisateur avec id 2 en 'user 5' et utilise un timestamp manuel de 12. Ceci est une valeur de timestamp spécifique pour l'opération d'update.

```
SELECT id, name, writetime(name) FROM user;
```

La requête est répétée pour afficher le nouveau timestamp de la colonne name. Elle montre que le nom associé à id 2 a changé pour 'user 5' avec un timestamp mis à jour à 12.

```
cqlsh:demovideo> DELETE name FROM user USING TIMESTAMP 13 WHERE id=2;
cqlsh:demovideo> SELECT id,name, writetime(name) FROM user;
```

id	name	writetime(name)
1	user 1	1706258219812529
2	null	null
3	user 3	1706258269584759

(3 rows)

```
cqlsh:demovideo>
```

**DELETE name FROM user USING TIMESTAMP 13 WHERE id=2;**

Cette commande supprime la valeur de la colonne name pour l'enregistrement où id est égal à 2. La clause USING TIMESTAMP 13 attribue un timestamp spécifique à l'opération de suppression.

**SELECT id, name, writetime(name) FROM user;**

Après l'opération de suppression, cette requête sélectionne l'id, le name, et le timestamp de la dernière modification pour la colonne name. Le résultat montre que la colonne name pour id 2 est maintenant null, et la fonction writetime(name) retourne également null, ce qui indique que la colonne a été supprimée. Les autres enregistrements (pour id 1 et 3) restent inchangés.

```
cqlsh:demovideo> UPDATE user USING TTL 60 SET name = 'user 10' WHERE id = 2;
cqlsh:demovideo> SELECT id, name, ttl(name) FROM user;
```

id	name	ttl(name)
1	user 1	null
2	user 10	47
3	user 3	null

(3 rows)

```
cqlsh:demovideo>
```

**UPDATE user USING TTL 60 SET name = 'user 10' WHERE id = 2;**

Cette commande met à jour le nom de l'utilisateur avec id 2 en 'user 10' et définit un TTL (Time To Live) de 60 secondes. Le TTL est une propriété qui détermine la durée de vie d'une donnée dans la base; après l'expiration du TTL, la donnée est supprimée automatiquement.

**SELECT id, name, ttl(name) FROM user;**

Après l'opération de mise à jour, cette requête sélectionne l'id, le name, et le TTL restant pour la colonne name. Le résultat montre que le nom de l'utilisateur id 2 est 'user 10' et que le TTL restant pour cette valeur de colonne est de 47 secondes, ce qui signifie que 13 secondes se sont écoulées depuis la mise à jour.