CrateDB

REPORT

CORSO DI:

Complementi di Basi di Dati
CdL Informatica @UniMiB
A.A. 2023-2024

PROGETTO DI:

Silvia Cambiago
Matricola 879382

1. INTRODUZIONE

CrateDB è un DBMS open-source SQL distribuito, scritto in Java ed ottimizzato per analizzare in tempo reale grandi quantità di dati con struttura relazionale o semi-strutturata. Dispone sia di una ricerca full-text (il motore di ricerca esamina ogni parola in ciascun documento archiviato, con lo scopo di trovare un riscontro secondo determinati criteri) che di una vettoriale (supporta l'indicizzazione e l'esecuzione di query su rappresentazioni numeriche del contenuto), entrambe integrate.

CrateDB mira ad essere un ibrido tra i modelli relazionale e documentale, unendo l'efficienza dei primi e la flessibilità dei secondi. Per questo, risulta particolarmente adatto per applicazioni che richiedono l'analisi di dati provenienti da fonti eterogenee ed in continua evoluzione, come Internet of Things (IoT), intelligenza artificiale, machine learning, log di sistema o dati di monitoraggio.
Per fornire un esempio pratico, si supponga di avere una tabella CrateDB che memorizzi i dati provenienti da sensori IoT come la seguente:

```
CREATE TABLE sensor_data (
    id INTEGER PRIMARY KEY,
    sensor_name STRING,
    timestamp TIMESTAMP,
    data OBJECT(DYNAMIC)
);
```

per la colonna `"data"` è necessaria una struttura flessibile, in quanto i diversi sensori possono inviare informazioni aventi attributi e strutture diverse. Ad esempio, un sensore di temperatura rileverà solo quest'ultima, mentre uno di qualità dell'aria potrà fornire dati su vari fattori inquinanti. CrateDB supporta JSON come tipo di dato nativo, e consente l'uso di `OBJECT(DYNAMIC)` per contenerlo, permettendo di aggiungere nuovi campi senza la necessità di modificare lo schema della tabella.
Un esempio di inserimento che metta in luce ciò che è stato appena spiegato è:

```
INSERT INTO sensor_data (id, sensor_id, timestamp, data) VALUES
(1, 'sensor_1', '2024-06-01T12:00:00Z', '{"temperature": 22.5,
"humidity": 60}'),
(2, 'sensor_2', '2024-06-01T12:05:00Z', '{"light": 350, "status":
"active"}'),
(3, 'sensor_3', '2024-06-01T12:10:00Z', '{"temperature": 23.0,
"humidity": 55, "battery": "full"}');
```

Questo tipo di integrazione è estremamente difficile da implementare in database relazionali come MySQL, ma rispetto a quelli documentali, come MongoDB, CrateDB consente l'uso di SQL per l'interrogazione e la manipolazione dei dati, offrendo una curva di apprendimento più bassa per chi ha familiarità con i database relazionali e beneficiando dell'efficienza di uno schema tabellare.

Un’altra delle principali caratteristiche di questo DBMS è la scalabilità, soprattutto orizzontale, in quanto CrateDB può essere distribuito su un cluster di nodi, attraverso un’architettura “shared-nothing” e masterless, dove ogni nodo è indipendente e non ne esiste uno centrale di coordinamento. Ciò elimina i bottleneck ed i singoli punti di fallimento, garantendo che l'aggiunta di nuovi nodi aumenti proporzionalmente la capacità di gestione dei dati e delle query. Inoltre, i dati vengono partizionati automaticamente e distribuiti tra i nodi del cluster. Questo significa che, al momento dell’aggiunta di nuovi nodi, i dati vengono ridistribuiti per bilanciare il carico.

Inoltre, è anche disponibile CrateDB Cloud, piattaforma di database come servizio (DBaaS) che permette di sfruttare le potenzialità di CrateDB in un ambiente gestito, facilitando l'implementazione, la gestione e la scalabilità delle applicazioni. CrateDB Cloud può essere distribuito su AWS, Microsoft Azure e Google Cloud, sfruttando le loro infrastrutture globali ed i rispettivi servizi complementari per l’analisi dati, il supporto all’intelligenza artificiale ed al machine learning o la gestione delle API.

CrateDB dispone di un’Admin UI estremamente intuitiva e user-friendly, che offre agli amministratori un controllo completo e una visione dettagliata delle operazioni del database.
Di seguito alcune immagini:

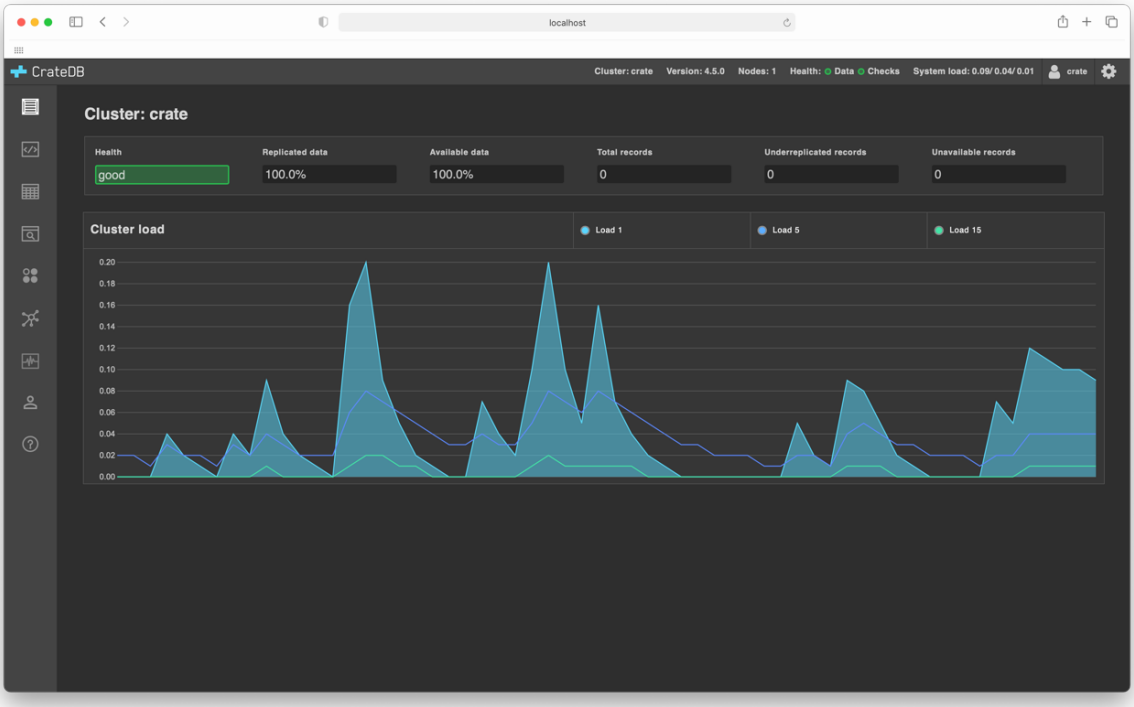


Fig. 1: overview generale

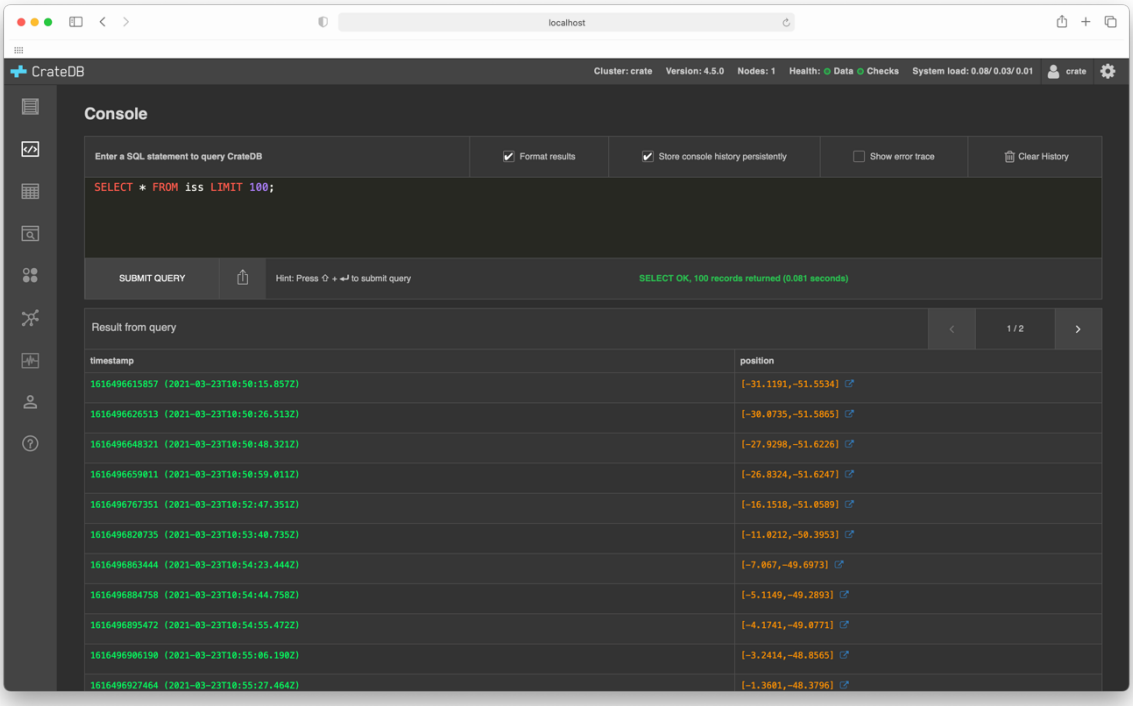


Fig. 2: SQL console

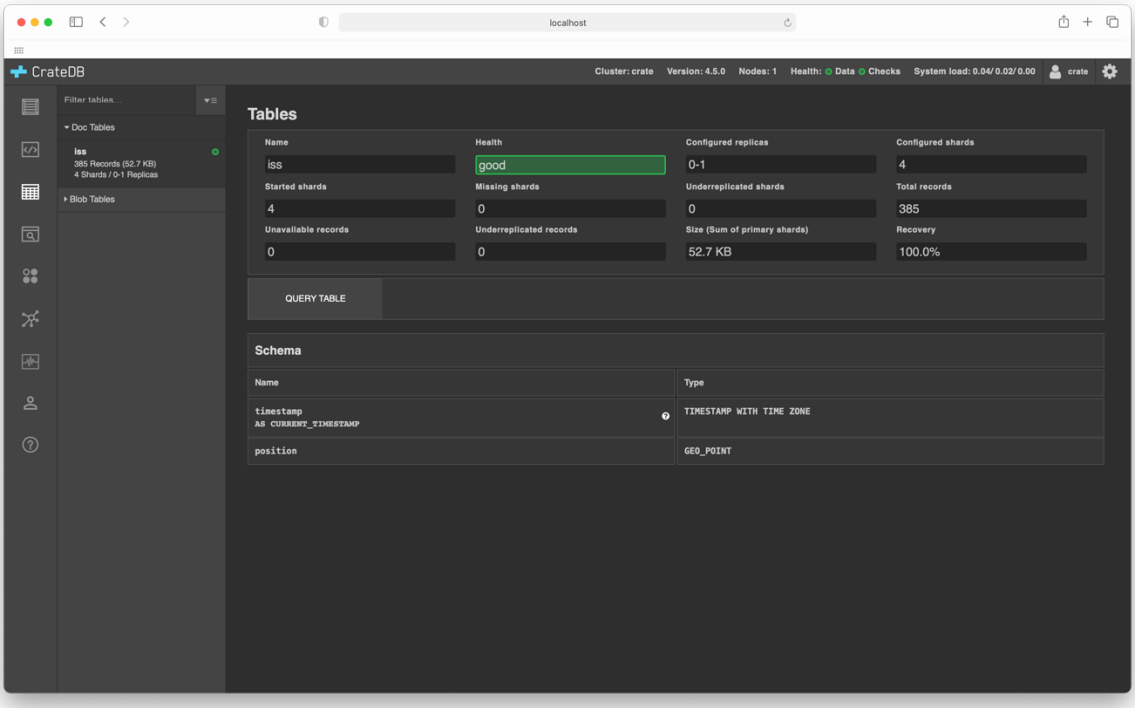


Fig. 3: schema delle tabelle

2. INSTALLAZIONE SULLE VARIE PIATTAFORME

CrateDB è open source, il che vuol dire che dal sito web https://cdn.crate.io/downloads/releases/cratedb/ è possibile scaricare gratuitamente il package per i sistemi operativi Linux, Windows e MacOS.

SISTEMI UNIX-LIKE (LINUX E MACOS)

Per questi sistemi, una volta scaricata la release adeguata, con estensione `.tar`, si può procedere ad estrarre la cartella manualmente oppure da terminale, assicurandosi di aver aperto il percorso della stessa e digitando l'istruzione:

```
tar -xzf crate-*.tar.gz
```

dove l'asterisco indica la versione di Crate scaricata.
A questo punto, sempre da terminale, si passa ad aprire la directory estratta:

```
cd crate-*
```

ed è ora possibile avviare un'istanza (singolo nodo) in locale, con l'istruzione:

```
./bin/crate
```

Per MacOS, potrebbe essere necessario un ulteriore passaggio. Se compare l'avviso "Java cannot be opened because the developer cannot be verified", basta aprire System Settings > Privacy & Security, andare alla sezione "Security" dove verrà riportato l'avviso stesso, e cliccare "Allow Anyway". A questo punto, eseguendo nuovamente la stessa istruzione da linea di comando, sarà possibile utilizzare l'Admin UI di CrateDB all'indirizzo:

```
http://localhost:4200/
```

Per interrompere l'esecuzione si può usare `ctrl-c`.

WINDOWS

Anche in questo caso è necessario scaricare la release corrispondente. I passaggi da eseguire sono identici a quelli descritti per i sistemi Unix-like. È consigliato utilizzare PowerShell. Quando si esegue `./bin/crate`, si riceverà una notifica del tipo:

```
[2022-07-04T19:41:12,340][INFO ][o.e.n.Node] [Aiguille Verte] started
```

Analogamente al caso precedente, l'Admin UI web-based sarà disponibile all'indirizzo:

```
http://localhost:4200/
```

Ed anche su Windows, per interrompere Crate si usa `ctrl-c`. Apparirà il prompt:

```
Terminate batch job (Y/N)?
```

ed a questo punto sarà sufficiente digitare Y.

DOCKER CONTAINER

CrateDB è disponibile anche nei container environment, tra cui Docker. Per eseguire rapidamente il database, basta tenere aperto il programma Docker Desktop sul proprio dispositivo (indipendentemente dal SO) ed eseguire:

```
docker run --publish=4200:4200 --publish=5432:5432 --env CRATE_HEAP_SIZE=1g --pull=always crate
```

Va però tenuto presente che, al momento della chiusura del terminale, l'Admin UI all'indirizzo `http://localhost:4200/` cesserà di essere disponibile. In alternativa, è possibile creare un container mediante un file `docker-compose.yml` con la seguente sintassi:

```
version: "3"

services:

  crate:
    image: crate:latest
    container_name: cratedb_container
    ports:
      - "4200:4200"
    environment:
      - CRATE_HEAP_SIZE=1g
    volumes:
      - ./my_datasets:/docker_datasets
```

E, da linea di comando, posizionarsi nella cartella contenente il file `.yml` e digitare:

```
docker compose up
```

3. LIBRERIE PYTHON

Le librerie Python utilizzate in questo progetto per manipolare i dati sono `crate`, `time` e `traceback`. La prima è specifica di CrateDB e consente di interagire con il database da un'applicazione Python.
In particolare, è stata impiegata per gli scopi che verranno elencati di seguito.

CONNESSIONE AL DATABASE

```
from crate import client

with client.connect("http://cratedb_container:4200", username="crate") as connection:
    print("connected")
```

ESECUZIONE DI QUERY SQL

```
cursor = connection.cursor()

cursor.execute(query)
```

Ovviamente comprese tutte le operazioni CRUD che verranno descritte nei paragrafi seguenti.

GESTIONE DELLE TRANSAZIONI

```
connection.commit()
```

La libreria `time` è invece stata impiegata per calcolare i tempi di esecuzione delle query:

```
import time

start_time = time.time()

cursor.execute(query)

connection.commit()

end_time = time.time()

execution_time = end_time - start_time

print(f"Time taken to insert: {execution_time} seconds")
```

L'ultima libreria, `traceback`, è stata impiegata per analizzare gli errori o le eccezioni che si sono verificati nell'esecuzione, in modo tale da comprendere meglio gli statement da correggere:

```
import traceback

try:
      cursor.execute(query)
      connection.commit()
except Exception as e:
      print(f"Error inserting data: {e}")
      print(traceback.format_exc())
```

4. IMPORTAZIONE DEI DATI

CrateDB supporta sia CSV che JSON come formati di file per l'inserimento di dati. I file JSON devono essere formattati in modo tale che compaia un solo oggetto per riga, come nell'esempio di seguito:

```
{"id": 1, "quote": "I got voices in my head again, tread carefully"}
{"id": 2, "quote": "My mind's been where no mind should go"}
```

Per quanto riguarda i CSV, questi possono contenere un header o meno:

```
id,quote
1,"I got voices in my head again, tread carefully"
2,"My mind's been where no mind should go"

1,"I got voices in my head again, tread carefully"
2,"My mind's been where no mind should go"
```

Indipendentemente dal formato, l'istruzione che consente il caricamento è la `COPY FROM`, che connette CrateDB con il file system e permette di importare dati memorizzati in locale o remoto e inserirli in una tabella. La sintassi è decisamente semplice:

```
COPY table_name FROM 'path_to_file';
```

In caso si utilizzi l'immagine Docker, sarà necessario importare il volume, quindi creare una cartella nella quale verranno depositati i file contenenti i dati e specificare il percorso della stessa.
Se il container è stato creato con il file `docker-compose.yml`, allora la directory andrà specificata alla voce `volumes`. Qualora invece si voglia eseguire tutto da linea di comando allora sarà necessario scrivere:

```
docker run --publish=4200:4200 --publish=5433:5432 --volume=local_path:/docker_path --env CRATE_HEAP_SIZE=1g crate:latest
```

sostituendo ovviamente `local_path` e `docker_path` in modo opportuno. A questo punto, sarà possibile aggiungere i dataset alla cartella `local_path`, che sarà accessibile da CrateDB.

Una volta definiti datasets e directories, il primo passaggio da effettuare è la creazione di una tabella avente tante colonne quante i campi del file contenente il dataset in questione. Per questo progetto, sono stati utilizzati files in formato JSON provenienti da DrugBank, un database online contenente informazioni su farmaci, sostanze, bersagli dei farmaci e proteine, ottenuti mediante licenza accademica. La prima tabella creata (presente anche nel file Jupyter) è `drugs_external`, dal dataset `externaldrugjson.json`, riportata di seguito:

```
CREATE TABLE IF NOT EXISTS drugs_external (
  "DrugBank ID" TEXT PRIMARY KEY,
  "Name" TEXT,
  "CAS Number" TEXT,
  "Drug Type" TEXT,
  "UniProt ID" TEXT,
  "UniProt Title" TEXT,
  "Drugs com Link" TEXT,
  "BindingDB ID" INT
);
```

A questo punto si esegue l'istruzione `COPY FROM`, che importa `externaldrugjson.json` nella tabella `drugs_external`. Per controllare la presenza di eventuali errori si può aggiungere `RETURN SUMMARY` alla fine della query:

```
COPY drugs_external
FROM '/docker_datasets/externaldrugjson.json'
RETURN SUMMARY;
```

Per i CSV funziona esattamente allo stesso modo. Anche la più canonica query SQL `INSERT INTO` è supportata, ed è stata utilizzata nel corso di tutto il progetto. Verrà poi analizzata meglio nel paragrafo successivo, che tratta le query CRUD.

Per quanto riguarda le modalità utilizzate nella pratica in questo progetto, è stato sfruttato Jupyter per definire una funzione Python `load_json(json_file)` per caricare i dati all'interno della tabella:

```
def load_json(file_path):
    try:
        with open(file_path, 'r') as file:
            return json.load(file)
    except json.JSONDecodeError as e:
        print(f"Error decoding JSON: {e}")
        print(traceback.format_exc())
        return None
    except Exception as e:
        print(f"Error reading file: {e}")
        print(traceback.format_exc())
        return None
```

5. ESEMPI DI QUERY CRUD

CrateDB utilizza la sintassi SQL per eseguire query e manipolazione di dati.

CREATE

Per inserire un nuovo record all'interno della tabella si utilizza l'istruzione SQL `INSERT INTO`, specificando poi i `VALUES`. A titolo di esempio si riporta la riga inserita nella tabella `drugs_external` disponibile sul file Jupyter:

```
INSERT INTO drugs_external ("DrugBank ID", "Name", "CAS Number", "Drug Type", "UniProt ID", "UniProt Title", "Drugs com Link", "BindingDB ID")
VALUES ("DB123456", "New Drug", "123-45-6", "SmallMoleculeDrug", "U12345", "New Protein", "https://example.com", 789);
```

Per inserire molteplici record in un passaggio solo non vi è un'istruzione specifica, come nel caso della `insertMany()` di MongoDB, ma è sufficiente specificare in sequenza tutti i record da aggiungere alla tabella.
Di seguito, sempre un esempio tratto dal progetto:

```
INSERT INTO drugs_external ("DrugBank ID", "Name", "CAS Number", "Drug Type", "UniProt ID", "UniProt Title", "Drugs com Link", "BindingDB ID")
VALUES ("DB123457", "Drug1", "123-45-7", "SmallMoleculeDrug", "U123456", "Protein1", "https://example.com", 789),
    ("DB123458", "Drug2", "123-45-8", "BiotechDrug", "U123457", "Protein2", "https://example.com", 790),
```

```
    ("DB123459", "Drug3", "123-45-9", "BiotechDrug", "U123458",
"Protein3", "https://example.com", 791),
    ("DB123460", "Drug4", "123-45-10", "BiotechDrug", "U123459",
"Protein4", "https://example.com", 792),
    ("DB123461", "Drug5", "123-45-11", "SmallMoleculeDrug",
"U123460", "Protein5", "https://example.com", 793);
```

Nel Jupyter è stata usata la funzione:

```
cursor.executemany(insert_query, records_to_insert)
```

per inserire molteplici record nella tabella.

READ

Le operazioni di lettura si eseguono in modo esattamente analogo a qualsiasi database SQL-based.
Di seguito la query che restituisce tutti i record:

```
SELECT * FROM drugs_external;
```

Per ottenere in output informazioni sulle colonne della tabella, inclusi nomi, tipi di dati, se permettono valori `NULL`, se sono chiavi primarie, valori predefiniti ed altre proprietà, è possibile utilizzare `DESCRIBE drugs_external`.

Di seguito, invece, la query che restituisce i campi `Drugbank ID` e `Name` dei farmaci avente `BindingDB ID` diverso da `null`:

```
SELECT "DrugBank ID", "Name" FROM drugs_external WHERE "BindingDB ID"
IS NOT NULL;
```

UPDATE

Per modificare uno o più valori dei campi di un record, l'istruzione da utilizzare è la `UPDATE`, selezionando mediante chiave primaria (`DrugBank ID` in questo caso) l'elemento da modificare:

```
UPDATE drugs_external
SET "Name" = "Drug15", "CAS Number" = "420-45-7", "Drug Type" =
"SmallMoleculeDrug", "UniProt ID" = "U654321", "UniProt Title" =
"Protein15", "Drugs com Link" = "https://example.com/protein",
"BindingDB ID" = 789
WHERE "DrugBank ID" = "DB123457";
```

Nel progetto è stato sfruttato Python in modo tale da creare una tupla `new_values` contenente i valori da inserire nel record:

```
new_values = (
    "Drug15",
    "420-45-7",
    "SmallMoleculeDrug",
```

```
    "U654321",
    "Protein15",
    "https://example.com/protein",
    789,
    "DB123457"
)
```

La query `UPDATE` risultante è quindi:

```
UPDATE drugs_external
SET "Name" = ?, "CAS Number" = ?, "Drug Type" = ?, "UniProt ID" = ?,
"UniProt Title" = ?, "Drugs com Link" = ?, "BindingDB ID" = ?
WHERE "DrugBank ID" = ?;
```

e mediante il `cursor` sono stati aggiornati i valori.

Di seguito la seconda query di `UPDATE` eseguita:

```
update_query = """
UPDATE drugs_external
SET "Name" = ?, "CAS Number" = ?, "Drug Type" = ?, "UniProt ID" = ?,
"UniProt Title" = ?, "Drugs com Link" = ?, "BindingDB ID" = ?
WHERE "DrugBank ID" = ?;
"""

new_values = (
    "Drug45",
    "321-45-10",
    "ANewKindOfDrug",
    "U658428",
    "Protein45",
    "https://example.com/newdrug",
    792,
    "DB123460"
)

cursor.execute(update_query, new_values)
```

DELETE

Per l'operazione di eliminazione, il discorso è assolutamente analogo a quello dell'inserzione: non esiste, a differenza di database quali MongoDB, un'operazione specifica che permetta di eliminare molteplici record in un solo passaggio (`deleteMany()`). Ciò si può però eseguire aggiungendo un vincolo alla `DELETE` tradizionale di SQL, ovvero definire una query che elimini i record corrispondenti a determinati valori di chiave primaria, selezionati con `WHERE (key) IN (values)`. Si riporta innanzitutto un esempio di eliminazione di un singolo record:

```
DELETE FROM drugs_external WHERE "DrugBank ID" = "DB123456";
```

Mentre l'eliminazione multipla rimane:

```
DELETE FROM drugs_external WHERE "DrugBank ID" IN ("DB123457", "DB123458", "DB123459", "DB123460", "DB123461")
```

Per il progetto sono state definite una tupla `drugbank_ids_to_delete` con le chiavi dei record da cancellare ed una query `delete_query` con tanti "`?`" quanti sono i valori effettivi, per poi eseguire il tutto con il `cursor`:

```
delete_query = """
DELETE FROM drugs_external WHERE "DrugBank ID" IN (?, ?, ?, ?, ?);
"""

drugbank_ids_to_delete = ("DB123457", "DB123458", "DB123459", "DB123460", "DB123461")

cursor.execute(delete_query, drugbank_ids_to_delete)
```

6. ESEMPI DI QUERY COMPLESSE

In questa sezione verranno trattate tre tipologie di query più complesse delle precedenti, ovvero quelle che coinvolgono join, nested queries ed interrogazioni che eseguono molteplici operazioni. CrateDB supporta CROSS JOIN, INNER JOIN, EQUI JOIN, LEFT JOIN, RIGHT JOIN e FULL JOIN, tutti eseguiti con l'algoritmo del nested loop, tranne l'EQUI JOIN che implementa l'hash join. Tutti gli output sono riportati nel Jupyter, ma non trascritti per ragioni di spazio, trattandosi di migliaia di record.

Per cominciare, sono stati importati altri due datasets e create le rispettive tabelle. Il primo di essi è una raccolta di enzimi, il secondo di carriers, ovvero sostanze che servono come meccanismo in grado di migliorare la somministrazione e l'efficienza dei farmaci. Entrambe le tabelle prevedono la presenza dei campi `DrugBank ID` e `UniProt ID` analoghi al dataset già in uso. Per gli enzimi, avendo riscontrato problemi nella creazione della tabella e nell'importazione dei dati dal file JSON, è stato creato manualmente un campo `ActualKey` di tipo intero, che costituisse la nuova chiave primaria.
Di seguito l'importazione degli enzimi:

```
CREATE TABLE IF NOT EXISTS enzymes (
  "ActualKey" INT PRIMARY KEY,
  "DrugBank ID" TEXT,
  "Name" TEXT,
  "Type" TEXT,
  "UniProt ID" TEXT,
  "UniProt Name" TEXT
);

INSERT INTO enzymes ("ActualKey", "DrugBank ID", "Name", "Type", "UniProt ID", "UniProt Name")
VALUES (?, ?, ?, ?, ?, ?);
```

```
data = load_json('enzymejson.json')

for record in data:
        cursor.execute(insert_query, (
            record['ActualKey'],
            record['DrugBank ID'],
            record['Name'],
            record['Type'],
            record['UniProt ID'],
            record['UniProt Name'],
        ))
```

Per i carriers:

```
CREATE TABLE IF NOT EXISTS carriers (
  "DrugBank ID" TEXT,
  "Name" TEXT,
  "Type" TEXT,
  "UniProt ID" TEXT PRIMARY KEY,
  "UniProt Name" TEXT
);

INSERT INTO carriers ("DrugBank ID", "Name", "Type", "UniProt ID",
"UniProt Name")
VALUES (?, ?, ?, ?, ?);

data = load_json('carrierjson.json')

for record in data:
        cursor.execute(insert_query, (
            record['DrugBank ID'],
            record['Name'],
            record['Type'],
            record['UniProt ID'],
            record['UniProt Name'],
        ))
```

JOIN

Ovviamente, anche per i join vale la sintassi SQL. La query effettuata a titolo di esempio prevede di trovare ID e nomi dei farmaci che sono sia enzimi che carriers. Per eseguirla è stato necessario effettuare il join tra la tabella `drugs_external` e le altre due, sulla base del campo `DrugBank ID`:

```
SELECT ed."DrugBank ID", ed."Name"
FROM drugs_external ed
JOIN enzymes e ON ed."DrugBank ID" = e."DrugBank ID"
JOIN carriers c ON ed."DrugBank ID" = c."DrugBank ID"
GROUP BY ed."DrugBank ID", ed."Name";
```

NESTED QUERIES

Come esempio di query annidata, si vogliono trovare i nomi dei farmaci che hanno più di un carrier associato:

```
SELECT ed."Name" AS drug_name
    FROM drugs_external ed
    WHERE ed."DrugBank ID" IN (
        SELECT c."DrugBank ID"
        FROM carriers c
        GROUP BY c."DrugBank ID"
        HAVING COUNT(c."UniProt ID") > 1
    );
```

La sottoquery:

```
SELECT c."DrugBank ID"
FROM carriers c
GROUP BY c."DrugBank ID"
HAVING COUNT(c."UniProt ID") > 1
```

seleziona i farmaci che sono associati a più di un drug carrier, in quanto la clausola `HAVING COUNT(c."UniProt ID") > 1` assicura che vengano selezionati solo i `DrugBank ID` dei carrier che hanno un numero di `UniProt ID` associati maggiore di 1. La query principale utilizza poi questi `DrugBank ID` per filtrare la tabella `drugs_external`.

Come secondo esempio di nested query, si vogliono restituire i farmaci associati sia con enzimi che con carriers aventi numero di entrambi maggiore o uguale alla media, con i risultati ordinati per numero di enzimi e carriers decrescente:

```
SELECT
        outer_query."DrugBank ID",
        outer_query."Name",
        outer_query.enzyme_count,
        outer_query.carrier_count
    FROM (
        SELECT
            ed."DrugBank ID",
            ed."Name",
            COUNT(DISTINCT e."UniProt ID") AS enzyme_count,
            COUNT(DISTINCT c."UniProt ID") AS carrier_count
        FROM drugs_external ed
        JOIN enzymes e ON ed."DrugBank ID" = e."DrugBank ID"
        JOIN carriers c ON ed."DrugBank ID" = c."DrugBank ID"
        GROUP BY ed."DrugBank ID", ed."Name"
        HAVING COUNT(DISTINCT e."UniProt ID") > 0
           AND COUNT(DISTINCT c."UniProt ID") > 0
    ) AS outer_query
    WHERE outer_query.enzyme_count >= (
```

```
        SELECT AVG(inner_query.enzyme_count)
        FROM (
            SELECT
                COUNT(DISTINCT e."UniProt ID") AS enzyme_count
            FROM drugs_external ed
            JOIN enzymes e ON ed."DrugBank ID" = e."DrugBank ID"
            GROUP BY ed."DrugBank ID"
        ) AS inner_query
    )
    AND outer_query.carrier_count >= (
        SELECT AVG(inner_query.carrier_count)
        FROM (
            SELECT
                COUNT(DISTINCT c."UniProt ID") AS carrier_count
            FROM drugs_external ed
            JOIN carriers c ON ed."DrugBank ID" = c."DrugBank ID"
            GROUP BY ed."DrugBank ID"
        ) AS inner_query
    )
    ORDER BY outer_query.enzyme_count DESC, outer_query.carrier_count
DESC;
```

La prima sottoquery chiamata `inner_query` calcola il numero medio di enzimi associati ai farmaci, mentre la seconda esegue la stessa operazione con i carriers:

```
SELECT AVG(inner_query.enzyme_count)
        FROM (
            SELECT
                COUNT(DISTINCT e."UniProt ID") AS enzyme_count
            FROM drugs_external ed
            JOIN enzymes e ON ed."DrugBank ID" = e."DrugBank ID"
            GROUP BY ed."DrugBank ID"
        ) AS inner_query
```

```
SELECT AVG(inner_query.carrier_count)
        FROM (
            SELECT
                COUNT(DISTINCT c."UniProt ID") AS carrier_count
            FROM drugs_external ed
            JOIN carriers c ON ed."DrugBank ID" = c."DrugBank ID"
            GROUP BY ed."DrugBank ID"
        ) AS inner_query
```

Mentre la sottoquery `outer_query` esegue il join della tabella `drugs_external` con `enzymes` e `carriers` per ottenere il numero di enzimi e carriers univoci per farmaco. Filtra poi il tutto considerando solo i farmaci aventi almeno un enzima ed un carrier associato:

```
SELECT
            ed."DrugBank ID",
            ed."Name",
            COUNT(DISTINCT e."UniProt ID") AS enzyme_count,
            COUNT(DISTINCT c."UniProt ID") AS carrier_count
        FROM drugs_external ed
        JOIN enzymes e ON ed."DrugBank ID" = e."DrugBank ID"
        JOIN carriers c ON ed."DrugBank ID" = c."DrugBank ID"
        GROUP BY ed."DrugBank ID", ed."Name"
        HAVING COUNT(DISTINCT e."UniProt ID") > 0
           AND COUNT(DISTINCT c."UniProt ID") > 0
    ) AS outer_query
```

La query principale filtra infine i risultati di `outer_query` per fornire in output solo quelli aventi un numero di enzimi e carriers maggiore o uguale alla media, ordinando i dati in modalità decrescente.

QUERY CHE ESEGUONO MOLTEPLICI OPERAZIONI

Come esempio di query che al suo interno esegue più operazioni, si vogliono trovare i 5 farmaci associati con il maggior numero di carrier, disposti come una classifica, quindi in ordine decrescente:

```
SELECT "DrugBank ID", "Name", carrier_count
    FROM (
        SELECT ed."DrugBank ID", ed."Name", COUNT(c."UniProt ID") AS
        carrier_count
        FROM drugs_external ed
        JOIN carriers c ON ed."DrugBank ID" = c."DrugBank ID"
        GROUP BY ed."DrugBank ID", ed."Name"
    ) sub
    ORDER BY carrier_count DESC
    LIMIT 5;
```

La query interna:

```
SELECT ed."DrugBank ID", ed."Name", COUNT(c."UniProt ID") AS
carrier_count
FROM drugs_external ed
JOIN carriers c ON ed."DrugBank ID" = c."DrugBank ID"
GROUP BY ed."DrugBank ID", ed."Name"
```

combina le tabelle `drugs_external` e `carriers` e conta il numero di `UniProt ID` per ogni `DrugBank ID`, assegnando questo conteggio all'alias `carrier_count`. I risultati vengono poi raggruppati per ogni combinazione unica di `DrugBank ID` e `Name`, assicurandosi che il `COUNT` venga eseguito per ciascun gruppo.

La query esterna:

```
SELECT "DrugBank ID", "Name", carrier_count
FROM (
    ...
) sub
ORDER BY carrier_count DESC
LIMIT 5;
```

viene trattata come una tabella temporanea chiamata `sub`. Dispone i risultati della query interna in ordine decrescente di `carrier_count`, quindi i farmaci con il maggior numero di drug carrier appariranno per primi. Il risultato è limitato ai primi 5 record.
Come secondo esempio di query che esegue molteplici operazioni, si vogliono ottenere `Name` e `DrugBank ID` dei farmaci associati sia con enzimi che con carrier, insieme al numero di questi ultimi, con i dati ordinati in modalità decrescente per numero di enzimi e carriers associati:

```
SELECT ed."DrugBank ID", ed."Name",
           COUNT(DISTINCT e."UniProt ID") AS enzyme_count,
           COUNT(DISTINCT c."UniProt ID") AS carrier_count
    FROM drugs_external ed
    LEFT JOIN enzymes e ON ed."DrugBank ID" = e."DrugBank ID"
    LEFT JOIN carriers c ON ed."DrugBank ID" = c."DrugBank ID"
    GROUP BY ed."DrugBank ID", ed."Name"
    HAVING COUNT(DISTINCT e."UniProt ID") > 0
       AND COUNT(DISTINCT c."UniProt ID") > 0
    ORDER BY enzyme_count DESC, carrier_count DESC;
```

L'utilizzo del `LEFT JOIN` assicura che tutti i farmaci siano inclusi, anche quelli che non hanno enzimi e carriers associati (almeno inizialmente). I risultati vengono raggruppati per i campi `Name` e `DrugBank ID`, per assicurare che ogni farmaco compaia nell'output una sola volta.
Viene dunque applicata una selezione per mezzo della clausola `HAVING`, per filtrare i soli farmaci che abbiano almeno un enzima ed un carrier associato, per poi ordinare il tutto in forma decrescente.

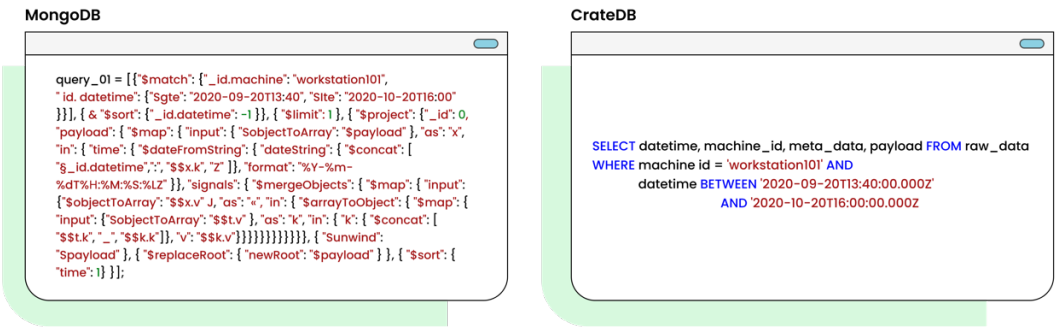
7. CONFRONTO CON MYSQL

MySQL è un database relazionale decisamente noto ed ampiamente utilizzato in molte applicazioni. Essendo uno dei RDBMS più popolari e diffusi, sono presenti molteplici tutorial online, oltre che una community estremamente vasta.
CrateDB, al contrario, è un database distribuito che sta guadagnando popolarità negli ultimi anni, soprattutto tra coloro che lavorano con grandi volumi di dati e richiedono scalabilità orizzontale. Anche se vi è meno documentazione disponibile rispetto a MySQL, il sito ufficiale fornisce tutorial sia testuali che multimediali per le più comuni operazioni. Inoltre, essendo CrateDB SQL-based per query e manipolazione di dati, risulta estremamente semplice passare dall'uno all'altro. La

struttura di CrateDB, a differenza di quella di MySQL, non è prettamente relazionale, ma più un ibrido tra quest'ultima ed un modello orientato ai documenti. A differenza di MySQL, infatti, può memorizzare e gestire dati in formato semi-strutturato come JSON, gestendo dati nidificati (come oggetti JSON contenenti altri oggetti o array). Ciò garantisce maggiore flessibilità rispetto a MySQL, che richiede invece uno schema rigido, in quanto CrateDB consente di aggiungere nuovi campi ai dati JSON senza dover modificare la struttura della tabella, favorendo l'applicazione pratica nell'ambito IoT.
In termini di scalabilità, MySQL non supporta nativamente lo sharding (distribuzione automatica dei dati su più nodi), nonostante possa essere implementato manualmente, richiedendo però una logica applicativa complessa per distribuire ed accedere ai dati su diversi server MySQL. Da questo punto di vista CrateDB è notevolmente più efficiente, in quanto partiziona automaticamente i dati in shard, che possono essere distribuiti su più nodi nel cluster. Ogni shard è una suddivisione della tabella che contiene un sottoinsieme dei dati, ed è possibile aggiungere facilmente nuovi nodi al cluster, perchè CrateDB ridistribuirà automaticamente i dati per sfruttare le nuove risorse.
Per quanto riguarda le prestazioni, entrambi i database sono decisamente performanti nelle query eseguite su tabelle relazionali strutturate (come sarà poi evidenziato nella trattazione dei tempi di esecuzione).

8. CONFRONTO CON MONGODB

MongoDB è un database NoSQL orientato ai documenti divenuto decisamente popolare negli ultimi anni, che utilizza un formato JSON-like (BSON) per la memorizzazione dei dati. Come nel caso di MySQL, la documentazione disponibile è vastissima e supportata da una community molto attiva, anche se il passaggio da un modello relazionale ad uno di questo tipo potrebbe essere complicato per gli utenti abituati all'utilizzo di SQL, problema che CrateDB non presenta. Il linguaggio di query di MongoDB, proprietario, risulta infatti più complesso rispetto a SQL, soprattutto in certi casi, come le query eseguite su date:



MongoDB ha una natura schemaless, anche se è consentito definire regole di validazione degli schemi a livello di collezione per garantire la coerenza dei dati. Ciò fornisce vantaggi soprattutto in termini di flessibilità, ma la definizione di uno schema iniziale, come avviene in CrateDB, favorisce l'efficienza e le prestazioni, che rendono quest'ultimo preferibile al concorrente documentale per applicazioni

che richiedono di lavorare su grandi quantità di dati (come sarà poi trattato anche nell'analisi dei tempi di esecuzione). Inoltre, come già riportato, CrateDB adotta un approccio flessibile agli schemi, combinando caratteristiche dei database relazionali con quelli orientati ai documenti, implementando quindi tutti i vantaggi di MongoDB, ma con una maggior efficienza.
Entrambi i database sono orientati alla scalabilità orizzontale attraverso lo sharding. Il funzionamento di questa procedura in CrateDB è riportato al punto precedente, nella trattazione del confronto con MySQL. In MongoDB l'approccio è fondamentalmente lo stesso, in quanto una collezione viene divisa in shards, ciascuno dei quali è un sottoinsieme dei dati, ed ognuno di questi può essere distribuito su nodi diversi.

TESTING APPROFONDITO

Per avere un confronto migliore con MongoDB, oltre che per testare adeguatamente il tipo di dato `OBJECT(DYNAMIC)` disponibile in CrateDB, è stato creato un secondo dataset, per mezzo di uno script Python che ha generato 10.000 records, simulando informazioni raccolte da sensori, con la seguente configurazione:

```
CREATE TABLE IF NOT EXISTS sensor_data (
    "sensor_id" STRING PRIMARY KEY,
    "timestamp" TIMESTAMP,
    "data" OBJECT(DYNAMIC)
)
```

ed il campo `"data"` è un file JSON che contiene temperatura, umidità e stato.
Di seguito un esempio di record:

```
{
    "sensor_id": "sensor_1",
    "timestamp": 1718987007000,
    "data": {
      "temperature": 20.9,
      "humidity": 56.37,
      "status": "error"
    }
  }
```

Su questo dataset sono state eseguite due query. La prima calcola la temperatura media dall'oggetto `"data"` per ogni sensore.
Viene riportata la sintassi per Crate:

```
SELECT
        sensor_id,
        AVG(CAST(data['temperature'] AS DOUBLE)) AS avg_temperature
    FROM
        sensor_data
    WHERE
```

```
        data['temperature'] IS NOT NULL
    GROUP BY
        sensor_id;
```

La seconda seleziona invece tutti i sensori con stato “normal”:

```
SELECT
        sensor_id,
        timestamp,
        data['status'] AS status
    FROM
        sensor_data
    WHERE
        data['status'] = 'normal';
```

Le stesse query sono state eseguite anche in MongoDB, insieme alla misurazione dei tempi di esecuzione. I risultati sono stati raccolti nella seguente tabella, con la misura espressa in secondi (*):

Query	CrateDB	MongoDB
Temperatura media	0.1008765763728	0.0709107635712
Stato sensore “normal”	0.0280430356389	0.0244283568761

Questi dati evidenziano come i tempi di esecuzione siano simili tra loro, soprattutto nella seconda query, ma con MongoDB leggermente avvantaggiato. Ciò è probabilmente dovuto al fatto che la gestione di dati dinamici è il principale motivo di utilizzo di MongoDB, che risulta quindi essere particolarmente ottimizzato in questo ambito. Tuttavia, anche CrateDB fornisce l’output quasi in tempo reale.

9. ANALISI DEI TEMPI DI ESECUZIONE

Per fornire maggiore chiarezza e completezza, per tutte le query eseguite, incluse le CRUD, è stato fornito in output, oltre al risultato delle stesse, anche il tempo di esecuzione, ottenuto mediante l’implementazione della libreria `time` di Python. L’analisi più significativa è però data dal confronto dei tempi di esecuzione delle query più complesse, raccolti per CrateDB, MongoDB e MySQL. Di seguito viene riportata una tabella nella quale sono riportati i tempi di esecuzione delle prime tre query complesse, una per tipo, misurati in secondi (*):

Query	CrateDB	MySQL	MongoDB
Join	0.0301299095153	0.0070637722015	10.934037685394
Nested	0.0397119522094	0.0104673519320	1.2760083675384
Multi-operation	0.0243415832519	0.0115916728973	4.6762816905975

Dai dati emerge come MySQL vanti le prestazioni migliori, con CrateDB che segue e MongoDB significativamente più lento. I tempi di risposta di MongoDB sono, con tutta probabilità, dovuti alla natura non relazionale del database ed alle operazioni di aggregazione più complesse da gestire. CrateDB e MySQL hanno invece risultati paragonabili, in quanto entrambi funzionano bene con uno schema preciso come quello delle tabelle del dataset utilizzato. In particolare, viene evidenziato come MySQL sia il più ottimizzato, soprattutto per operazioni gravose quali i join, risultato sensato visto che è un database unicamente relazionale.

Ciò che risulta particolarmente interessante è vedere come CrateDB, pur essendo multipotenziale ed adattabile a più schemi (caratteristica che spesso va a penalizzare l'efficienza), garantisca tempi di esecuzione quasi real-time sia per query complesse che coinvolgono unione di più tabelle, sia per operazioni su tipi di dati dinamici, non raggiungendo quasi mai i risultati dei DB più specializzati ed ottimizzati ma avvicinandosi sempre e comunque di molto al rispettivo competitore principale.

10. PRODOTTI E SERVIZI CHE IMPLEMENTANO CRATEDB

CrateDB è un database distribuito progettato per gestire grandi volumi di dati in tempo reale, specialmente in contesti quali IoT, analisi di dati ed applicazioni di machine learning. Essendo un progetto open-source, necessita di investitori che ne garantiscano il mantenimento.
Negli ultimi anni ha ricevuto fondi da importanti società finanziarie quali:

- Deutsche Invest Venture Capital: azienda specializzata nell'investimento nel campo della tecnologia e dell'innovazione, in particolare nel settore IoT, analisi dei dati, finanza e salute;
- Molten: membro di Draper Venture Network;
- Vsquared Ventures: investe in aziende innovative rivoluzionarie nei rispettivi settori. Fornisce finanziamenti, risorse e supporto strategico;
- Zetta Venture Partners: investe in aziende che impiegano il machine learning e l'analisi dei dati per fornire previsioni e suggerimenti che ottimizzino le operazioni aziendali e migliorino le decisioni strategiche;
- Breeze Invest: azienda che si concentra principalmente sulle opportunità di investimento nelle regioni DACH e CEE, offrendo un valore aggiunto attraverso il suo know-how nel business development;
- SpeedInvest: combina risorse finanziarie con esperienza imprenditoriale, per aiutare le startup a crescere rapidamente e con successo.

Per quanto riguarda invece le aziende che hanno scelto CrateDB come database, la maggior parte di esse si occupa di IoT industriale, AI, machine learning ed analisi di dati, ma non mancano eccezioni importanti.

Di seguito un elenco:

- Adobe: leader globale nel software creativo, che offre una vasta gamma di strumenti e soluzioni per la progettazione, l'elaborazione delle immagini, la creazione di contenuti digitali ed altro ancora;
- McAfee: fornisce soluzioni di sicurezza informatica avanzate per proteggere utenti e aziende da minacce online come virus, malware o phishing;
- Bitmovin: azienda specializzata nella fornitura di soluzioni per la trasmissione video online, inclusi codec video ad alte prestazioni, infrastrutture di distribuzione dei contenuti e strumenti per la gestione e l'ottimizzazione dello streaming video;
- Rauch Group: azienda attiva nel settore alimentare, che produce una vasta gamma di succhi di frutta e bevande analcoliche;
- ABB: fornisce soluzioni avanzate per l'energia, l'automazione industriale, i trasporti e le infrastrutture, con un focus particolare sull'efficienza energetica e sull'innovazione sostenibile;
- Gantner Instruments: azienda specializzata nella fornitura di strumenti ad alta precisione per una vasta gamma di settori, come quello energetico, dell'automazione industriale e della ricerca scientifica.

Le fonti da cui provengono i dati sopra elencati sono:

- https://www.discovery.hgdata.com/product/cratedb
- https://cratedb.com/

(*): i risultati possono leggermente variare in seguito a più esecuzioni delle stesse query nel file Jupyter.