Trademark Policy

From now on ...

Cassandra == Apache Cassandra™
Materialized Views (MV)
Why Materialized Views?

- Relieve the pain of manual denormalization

```sql
CREATE TABLE user(
    id int PRIMARY KEY,
    country text,
    ...
);
CREATE TABLE user_by_country(
    country text,
    id int,
    ...
,
    PRIMARY KEY(country, id)
);
```
CREATE TABLE user_by_country (
    country text,
    id int,
    firstname text,
    lastname text,
    PRIMARY KEY(country, id));

CREATE MATERIALIZED VIEW user_by_country
AS SELECT country, id, firstname, lastname
FROM user
WHERE country IS NOT NULL AND id IS NOT NULL
PRIMARY KEY(country, id)
Materialized View Syntax

CREATE MATERIALIZED VIEW [IF NOT EXISTS]

keyspace_name.view_name

AS SELECT column₁, column₂, ...

FROM keyspace_name.table_name

WHERE column₁ IS NOT NULL AND column₂ IS NOT NULL ...

PRIMARY KEY(column₁, column₂, ...)

- Must select all primary key columns of base table
- IS NOT NULL condition for now
- more complex conditions in future
- at least all primary key columns of base table (ordering can be different)
- maximum 1 column NOT pk from base table
Materialized Views Demo
Materialized View Impl

UPDATE user
SET country='FR'
WHERE id=1

①
- send mutation to all replicas
- waiting for ack(s) with CL
Materialized View Impl

UPDATE user
SET country='FR'
WHERE id=1

Acquire local lock on base table partition
UPDATE user
SET country='FR'
WHERE id=1

Local read to fetch current values
SELECT * FROM user WHERE id=1
Materialized View Impl

UPDATE user
SET country='FR'
WHERE id=1

Create BatchLog with

- DELETE FROM user_by_country
  WHERE country = 'old_value'
- INSERT INTO
  user_by_country (country, id, ...)
  VALUES ('FR', 1, ...)
Materialized View Impl

UPDATE user
SET country='FR'
WHERE id=1

⑤ Execute async BatchLog to paired view replica with CL = ONE
Materialized View Impl

UPDATE user
SET country='FR'
WHERE id=1

Apply base table update locally
SET COUNTRY='FR'
Materialized View Impl

UPDATE user
SET country='FR'
WHERE id=1

Release local lock
Materialized View Impl

UPDATE user
SET country='FR'
WHERE id=1

Return ack to coordinator
UPDATE user
SET country='FR'
WHERE id=1
Materialized Views impl explained

What is paired replica?

• Base primary replica for \texttt{id=1} is paired with MV primary replica for \texttt{country='FR'}
• Base secondary replica for \texttt{id=1} is paired with MV secondary replica for \texttt{country='FR'}
• etc …
Materialized Views impl explained

Why **local lock** on base replica?
- to provide **serializability** on concurrent updates

Why **BatchLog** on base replica?
- necessary for **eventual durability**
- replay the MV delete + insert until successful

Why **BatchLog** on base replica uses **CL = ONE**?
- each base replica is responsible for update of its **paired** MV replica
- CL > ONE will create un-necessary **duplicated mutations**
MV Failure Cases: concurrent updates

1) UPDATE ... SET country='US'
   - Read base row (country='UK')
   - DELETE FROM mv WHERE country='UK'
   - INSERT INTO mv ...(country) VALUES('US')
   - Send async BatchLog
   - Apply update country='US'

2) UPDATE ... SET country='FR'
   - Read base row (country='UK')
   - DELETE FROM mv WHERE country='UK'
   - INSERT INTO mv ...(country) VALUES('FR')
   - Send async BatchLog
   - Apply update country='FR'
MV Failure Cases: concurrent updates

1) UPDATE ... SET country='US'
   - Read base row (country='UK')
   - DELETE FROM mv WHERE country='UK'
   - INSERT INTO mv ...(country) VALUES('US')
   - Send async BatchLog
   - Apply update country='US'

2) UPDATE ... SET country='FR'
   - Read base row (country='UK')
   - DELETE FROM mv WHERE country='UK'
   - INSERT INTO mv ...(country) VALUES('FR')
   - Send async BatchLog
   - Apply update country='FR'

Without local lock
MV Failure Cases: concurrent updates

1) UPDATE ... SET country='US'

- Read base row (country='UK')
  - DELETE FROM mv WHERE country='UK'
  - INSERT INTO mv ...(country) VALUES('US')
  - Send async BatchLog
  - Apply update country='US'

2) UPDATE ... SET country='FR'

- Read base row (country='US')
  - DELETE FROM mv WHERE country='US'
  - INSERT INTO mv ...(country) VALUES('FR')
  - Send async BatchLog
  - Apply update country='FR'

With local lock

🔒
🔓
🔒
🔓
MV Failure Cases: failed updates to MV

UPDATE user
SET country='FR'
WHERE id=1

⑤ Execute async BatchLog to paired view replica with CL = ONE

MV replica down
MV Failure Cases: failed updates to MV

UPDATE user
SET country='FR'
WHERE id=1
Materialized View Performance

• Write performance
  • local lock
  • local read-before-write for MV → update contention on partition (*most of perf hits*)
  • local batchlog for MV
  • ☞ you only pay this price *once* whatever number of MV
  • for each base table update: `mv_count x 2 (DELETE + INSERT)` extra mutations
Materialized View Performance

- Write performance vs manual denormalization
  - MV better because no client-server network traffic for read-before-write
  - MV better because less network traffic for multiple views (client-side BATCH)

- Makes developer life easier → priceless
Materialized View Performance

- Read performance vs secondary index
  - MV better because **single node read** (secondary index can hit many nodes)
  - MV better because **single read path** (secondary index = read index + read data)
Materialized Views Consistency

- Consistency level
  - CL honoured for base table, **ONE** for MV + local batchlog

- **Weaker consistency guarantees** for MV than for base table.

- Exemple, write at **QUORUM**
  - guarantee that **QUORUM** replicas of base tables have received write
  - guarantee that **QUORUM** of MV replicas will **eventually** receive **DELETE + INSERT**
Materialized Views Gotchas

• Beware of hot spots !!! MV `user_by_gender`

• Only 1 non-pk column for MV

• No **static column** for MV view
  • *1:1 relationship* between static column & partition key
  • if MV supports static column → MV has same partition key as base table → uselessss ...
Materialized Views Operations

- Repair, read-repair, index rebuild, decommission ...
  - repair on base replica (*mutation-based* repair) $\rightarrow$ update on MV paired replica
  - possible to repair a MV *independently from base table*
  - read-repair on MV behaves as normal read-repair
  - read-repair on base table updates MV
  - hints replay on base table updates MV
Materialized Views Operations

- MV build status?
  - system.views_builds_in_progress
  - system.built_views
  - data are node-local

```
cqlsh:system> select * from system.views_builds_in_progress;

keyspace_name | view_name | generation_number | last_token
--------------|-----------|-------------------|------------

(0 rows)

cqlsh:system> select * from system.built_views;

keyspace_name | view_name
--------------|-----------
music         | albums_by_year
music         | artists_by_country
```

@doanduyhai
Materialized Views Schema Ops

- Schema
  - MV can be tuned as normal table (ALTER MATERIALIZED VIEW ...)
  - cannot drop column from base table used by MV
  - can add column to base table, initial value = null from MV
  - cannot drop base table, drop all MVs first
Single non-pk column limitation

- Because of Cassandra consistency model
- Because null value forbidden for primary key column

```
CREATE MATERIALIZED VIEW user_by_gender_and_age
AS SELECT country, id, firstname, lastname
FROM user
WHERE gender IS NOT NULL AND age IS NOT NULL AND id IS NOT NULL
PRIMARY KEY((gender, age) id)
```
Single non-pk column limitation

- Possible RULE: `UPDATE MV ONLY IF ALL COLUMNS IN PK NOT NULL`
Multiple non-PK columns in MV PK

\[ \text{gender=MALE, age=null} \]

\[ \text{gender=null, age=null} \]

\[ \text{gender=MALE, age=null} \]

CL = QUORUM

UPDATE user
SET gender='MALE'
WHERE id=1
Multiple non-PK columns in MV PK

CL = QUORUM

UPDATE user
SET age=34
WHERE id=1
Multiple non-PK columns in MV PK

CL = QUORUM

SELECT age,gender
FROM user
WHERE id=1

| 1 | 34 | MALE |

- gender=MALE, age=34
- gender=MALE, age=null
- gender=MALE, age=34
- gender=null, age=34
- gender=MALE, age=null

@doanduyhai
Multiple non-PK columns in MV PK

`SELECT * FROM user_by_gender_and_age
WHERE gender='MALE'
AND age=34`
Single non-pk column limitation

• Possible RULE 2: **ALLOW NULL VALUE FOR COLUMN IN PK**

```sql
INSERT INTO user(id, name) VALUES(1, 'John DOE');
...
...
INSERT INTO user(id, name) VALUES(1000_000, 'Helen SUE');
```

*No age, No gender*
Single non-pk column limitation

• Possible RULE 2: \texttt{ALLOW NULL VALUE FOR COLUMN IN PK}

\texttt{(null, null)} single partition with $10^6$ users

Living in Danger
Q & A
User Define Functions (UDF)
Rationale

• Push computation server-side
  • save network bandwidth (1000 nodes!)
  • simplify client-side code
  • provide standard & useful function (sum, avg ...)
  • accelerate analytics use-case (**pre-aggregation** for Spark)
CREATE [OR REPLACE] FUNCTION [IF NOT EXISTS]
[keystepse.]functionName (param₁ type₁, param₂ type₂, …)
CALL ON NULL INPUT | RETURNS NULL ON NULL INPUT
RETURN returnType
LANGUAGE language
AS $$
    // source code here
$$;
CREATE [OR REPLACE] FUNCTION [IF NOT EXISTS] [keyspace.]functionName (param\_1 type\_1, param\_2 type\_2, ...) CALLED ON NULL INPUT | RETURNS NULL ON NULL INPUT RETURN returnType LANGUAGE language AS $$
   // source code here $$;
How to create an UDF?

CREATE [OR REPLACE] FUNCTION [IF NOT EXISTS]
[keystpace.]functionName (param1 type1, param2 type2, ...)
CALLED ON NULL INPUT | RETURNS NULL ON NULL INPUT
RETURN returnType
LANGUAGE language
AS $$
  // source code here
$$;
How to create an UDF?

CREATE [OR REPLACE] FUNCTION [IF NOT EXISTS]
[keystpace.]functionName (param\textsubscript{1} type\textsubscript{1}, param\textsubscript{2} type\textsubscript{2}, ...)
CALLED ON NULL INPUT | RETURNS NULL ON NULL INPUT
RETURN returnType
LANGUAGE language
AS $$
   // source code here
$$;

Always called
Null-check mandatory in code
How to create an UDF?

CREATE [OR REPLACE] FUNCTION [IF NOT EXISTS] 
[keystage.]functionName (param\textsubscript{1} type\textsubscript{1}, param\textsubscript{2} type\textsubscript{2}, ...)
CALLED ON NULL INPUT | RETURNS NULL ON NULL INPUT
RETURN returnType
LANGUAGE language // jav
AS $$
// source code here
$$;
How to create an UDF?

CREATE [OR REPLACE] FUNCTION [IF NOT EXISTS]
[keyspace.]functionName (param1 type1, param2 type2, ...)
CALLED ON NULL INPUT | RETURNS NULL ON NULL INPUT
RETURN returnType
LANGUAGE language
AS $$
// source code
$$;
CREATE [OR REPLACE] FUNCTION [IF NOT EXISTS] [keyspace.]functionName (param_1 type_1, param_2 type_2, ...) CALLED ON NULL INPUT | RETURNS NULL ON NULL INPUT RETURN returnType LANGUAGE language AS $$
// source code here$$;
How to create an UDF?

CREATE [OR REPLACE] FUNCTION [IF NOT EXISTS]
[keystore.]functionName (param₁ type₁, param₂ type₂, ...) CALLED ON NULL INPUT | RETURNS NULL ON NULL INPUT
RETURN returnType
LANGUAGE language
AS $$
  // source code here
$$;
UDF Demo
UDA

- Real use-case for UDF
- Aggregation server-side → huge network bandwidth saving
- Provide similar behavior for Group By, Sum, Avg etc ...
How to create an UDA?

CREATE [OR REPLACE] AGGREGATE [IF NOT EXISTS] [keyspace.]aggregateName(type₁, type₂, ...) SFUNC accumulatorFunction STYPE stateType [FINALFUNC finalFunction] INITCOND initCond;

- Only type, no param name
- State type
- Initial state type
How to create an UDA?

CREATE [OR REPLACE] AGGREGATE [IF NOT EXISTS] [keyspace.]aggregateName(type$_1$, type$_2$, …) SFUNC accumulatorFunction STYPE stateType [FINALFUNC finalFunction] INITCOND initCond;

Accumulator function. Signature: accumulatorFunction(stateType, type$_1$, type$_2$, …) RETURNS stateType
How to create an UDA?

CREATE [OR REPLACE] AGGREGATE [IF NOT EXISTS]
[keystone.]aggregateName(type₁, type₂, ...)
SFUNC accumulatorFunction
STYPE stateType
[FINALFUNC finalFunction]
INITCOND initCond;

Optional final function. Signature: finalFunction(stateType)
How to create an UDA?

CREATE [OR REPLACE] AGGREGATE [IF NOT EXISTS]
[keyspace.]aggregateName(type₁, type₂, ...)
SFUNC accumulatorFunction
STYPE stateType
[FINALFUNC finalFunction]
INITCOND initCond;

UDA return type?
If finalFunction
• return type of finalFunction
Else
• return stateType
Gotchas
Why do not apply UDF/UDA on replica node?
Gotchas

1. Because of eventual consistency
2. UDF/UDA applied AFTER last-write-win logic

• apply accumulatorFunction
• apply finalFunction

UDA
Gotchas

• UDA in Cassandra is **not distributed**!

• Execute UDA on a large number of rows ($10^6$ for ex.)
  • single fat partition
  • multiple partitions
  • full table scan

• → Increase client-side timeout
  • default Java driver timeout = 12 secs
  • JAVA-1033 JIRA for *per-request timeout setting*
## Cassandra UDA or Apache Spark?

<table>
<thead>
<tr>
<th>Consistency Level</th>
<th>Single/Multiple Partition(s)</th>
<th>Recommended Approach</th>
</tr>
</thead>
<tbody>
<tr>
<td>ONE</td>
<td>Single partition</td>
<td>UDA with token-aware driver because node local</td>
</tr>
<tr>
<td>ONE</td>
<td>Multiple partitions</td>
<td>Apache Spark because distributed reads</td>
</tr>
<tr>
<td>&gt; ONE</td>
<td>Single partition</td>
<td>UDA because data-locality lost with Spark</td>
</tr>
<tr>
<td>&gt; ONE</td>
<td>Multiple partitions</td>
<td>Apache Spark definitely</td>
</tr>
</tbody>
</table>
Cassandra UDA or Apache Spark?

<table>
<thead>
<tr>
<th>Consistency Level</th>
<th>Single/Multiple Partition(s)</th>
<th>Recommended Approach</th>
</tr>
</thead>
<tbody>
<tr>
<td>ONE</td>
<td>Single partition</td>
<td>UDA with token-aware driver because node local</td>
</tr>
<tr>
<td>ONE</td>
<td>Multiple partitions</td>
<td>Apache Spark because distributed reads</td>
</tr>
<tr>
<td>&gt; ONE</td>
<td>Single partition</td>
<td>UDA because data-locality lost with Spark</td>
</tr>
<tr>
<td>&gt; ONE</td>
<td>Multiple partitions</td>
<td>Apache Spark definitely</td>
</tr>
</tbody>
</table>
Q & A
Thank You

@doanduyhai

duy_hai.doan@datastax.com