k2view

K2 Alpinist

Big LUIs & MDB Built-in functions

WELCOME

Agenda

- Big LUIs
- Useful Fabric commands
- Useful built-in Fabric functions
- Design Considerations

Big LUIs

Storing Big LUIs in Cassandra

When using Cassandra for LUI storage, it's important to note that due to Cassandra's 2GB per column limitation, large LUI blobs are stored in chunks.

- **Chunk Size Configuration**:
  The chunk size is configured in config.ini under [system_db_entity_storage].INSTANCE_CHUNK_SIZE.
  Any instance exceeding this size (post-compression) will be stored in chunks.
- **Parallel Chunk Writing**:
  The chunks of the SQLite file are written into the Cassandra entity_chunks table in parallel, utilizing the Cassandra Loader.
  To configure parallel saving, you can adjust the Loader settings in config.ini: add [LUtype]_cassandra_entity_storage section for each LU and increasing the **NUMBER_OF_THREADS** parameter.
- **Data Flow**:
  The LUI data is first written into the entity_chunks table, after which the entity table is populated.

Big LUIs

Storing Big LUIs in Cassandra

Data Structure Overview

Entity Table:

The entity table contains the following key information:

- batch_id: A unique identifier used to link records between the entity and entity_chunks tables.
- chunks_count: The total number of chunks into which the data is split.

Entity_Chunks Table:

- id: The instance ID.
- chunk_index: The specific chunk number.
- sync_version: The version, identical to the one recorded in the entity table.
- batch_id: The same unique identifier as in the entity table.
- data: Contains the portion of the compressed SQLite file corresponding to the chunk index

Big LUIs

Storing Big LUIs in Cassandra

Note:

Adjusting the chunk size in INSTANCE_CHUNK_SIZE may require additional changes in Cassandra.yaml (like mutation and commitlog sizes) and in Fabric's Cassandra Loader configuration within config.ini.
Consult with the Product Solution team before making any changes to this parameter.

Cassandra Loader Configuration:

The Cassandra Loader can be configured in one of the following sections, in order of priority:

- default_loader – for all Cassandra operations
- Cassandra_entity_storage_loader – for big entities configuration
- [lu_type]_cassandra_entity_storage – for big entities of a specific LUT

Entity storage may utilize a different method from other Fabric operations, and specific LU_types can have unique settings that override the default configuration.

Big LUIs

Loading Big LUIs from Cassandra

Parallel Loading of Large LUIs from Cassandra

When loading large LUIs from Cassandra into Fabric as part of a GET command, the chunks are combined and decompressed. There is a balance between load performance and the memory allocated to this process.

To enhance performance, you can configure the number of parallel threads and the maximum memory allocated:

- **Config.ini.[system_db_entity_storage].ASYNC_LOAD_MAX_THREADS**: Set the maximum number of threads (across all Fabric sessions on the same node) that can be allocated. The default is set to 0, meaning parallel load is disabled by default.
- **Config.ini.[system_db_entity_storage].ASYNC_LOAD_MAX_MEMORY_IN_MB**: Defines the maximum memory allocated for the parallel load process. The default is set to 2000 MB.

Big LUIs

Storing Big LUIs in Cassandra

Big LUI Partitioning

To distribute the load across multiple nodes, you can store entity chunks using a new partition key.

Enable this feature using the followin parameters:

- **Config.ini.[system_db_entity_storage].ENABLE_PARTITIONED_MDB**=true

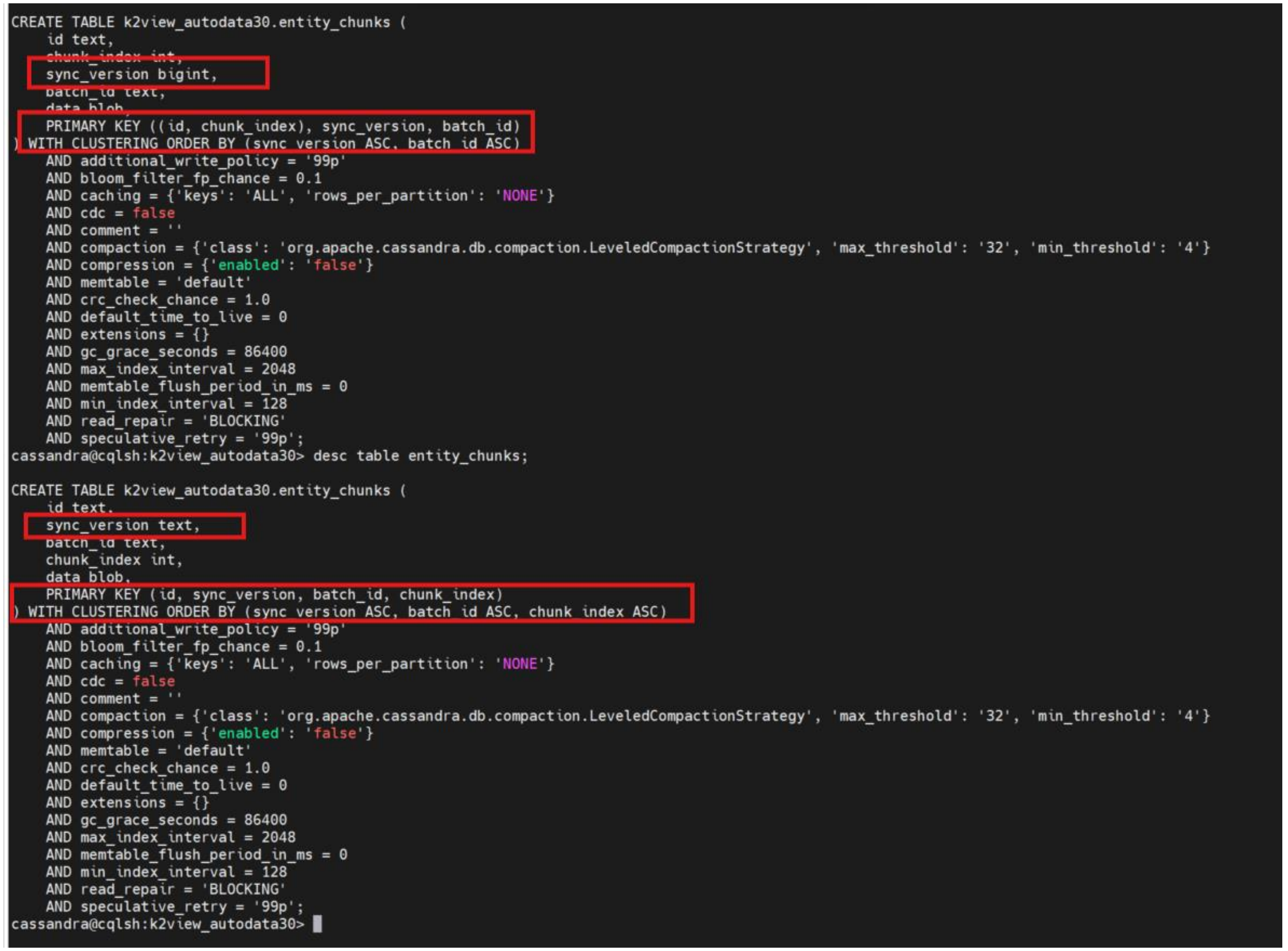
Note: The chunk_index column is added to the partition key, meaning there is no upgrade path for existing projects. You must clean all data in Fabric and reload it.

Big LUIs

Storing Big LUIs in Cassandra

ENABLE_PARTITIONED_MDB=true

ENABLE_PARTITIONED_MDB=false



```
CREATE TABLE k2view_autodata30.entity_chunks (
    id text,
    chunk_index int,
    sync_version bigint,
    batch_id text,
    data blob,
    PRIMARY KEY ((id, chunk_index), sync_version, batch_id)
) WITH CLUSTERING ORDER BY (sync_version ASC, batch_id ASC)
    AND additional_write_policy = '99p'
    AND bloom_filter_fp_chance = 0.1
    AND caching = {'keys': 'ALL', 'rows_per_partition': 'NONE'}
    AND cdc = false
    AND comment = ''
    AND compaction = {'class': 'org.apache.cassandra.db.compaction.LeveledCompactionStrategy', 'max_threshold': '32', 'min_threshold': '4'}
    AND compression = {'enabled': 'false'}
    AND memtable = 'default'
    AND crc_check_chance = 1.0
    AND default_time_to_live = 0
    AND extensions = {}
    AND gc_grace_seconds = 86400
    AND max_index_interval = 2048
    AND memtable_flush_period_in_ms = 0
    AND min_index_interval = 128
    AND read_repair = 'BLOCKING'
    AND speculative_retry = '99p';
cassandra@cqlsh:k2view_autodata30> desc table entity_chunks;

CREATE TABLE k2view_autodata30.entity_chunks (
    id text,
    sync_version text,
    batch_id text,
    chunk_index int,
    data blob,
    PRIMARY KEY (id, sync_version, batch_id, chunk_index)
) WITH CLUSTERING ORDER BY (sync_version ASC, batch_id ASC, chunk_index ASC)
    AND additional_write_policy = '99p'
    AND bloom_filter_fp_chance = 0.1
    AND caching = {'keys': 'ALL', 'rows_per_partition': 'NONE'}
    AND cdc = false
    AND comment = ''
    AND compaction = {'class': 'org.apache.cassandra.db.compaction.LeveledCompactionStrategy', 'max_threshold': '32', 'min_threshold': '4'}
    AND compression = {'enabled': 'false'}
    AND memtable = 'default'
    AND crc_check_chance = 1.0
    AND default_time_to_live = 0
    AND extensions = {}
    AND gc_grace_seconds = 86400
    AND max_index_interval = 2048
    AND memtable_flush_period_in_ms = 0
    AND min_index_interval = 128
    AND read_repair = 'BLOCKING'
    AND speculative_retry = '99p';
cassandra@cqlsh:k2view_autodata30>
```

Big LUIs

Key Considerations for Working with Big LUIs

LU Design Considerations

- **Split the LU Schema**: When possible, split big LUTs into smaller ones. This optimizes performance by allowing other processes, such as APIs, to skip fetching unnecessary LUTs.
- **Minimize Storage**:
  - Store only relevant columns.
  - Store only relevant data
    - ***Implement Data Purging****: Purge unnecessary data when possible.*
    - ***Historical Data****: If history data is required, consider storing it separately (e.g., in Cassandra or a history LUI).*

Sync Time

- **Longer Sync Duration**: Syncing large LUIs may take extra time due to the volume of records.
  It's not recommended to set the sync timeout based on the largest LUI. Therefore, consider using a separate process for handling big LUIs sync, and adjust the session sync timeout using the set sync timeout command.

Big LUIs

Key Considerations for Working with Big LUIs

Tuning

- **Increase fetching Size**: If the LUI is going to source many times, consider increasing the size in the SourceDbQuery actor.
- **Optimize Network Fetching:** If the network is slow for a specific source, increase the fetch size to transfer more data per roundtrip.

  Fetch size settings determine the number of rows that are retrieved in any subsequent trips to the database for a result set.
  - Tune FETCH_SIZE in config.ini (will affect all DB Interfaces)
  - Using 'Post connection commands' property of the interface
  - Use the below code:

```
Connection con = getConnection("EDF_UAT");
try(Statement stm = con.createStatement()) {
    stm.setFetchSize(10000);
    try (ResultSet rs = stm.executeQuery("select ID FROM GDB.ORGANIZATION")) {
        while (rs.next()) {
            yield(new Object[]{rs.getString(1)});
        }
    }
}
```

- **Cassandra Storage Optimization:**
  - Enable parallel saving to the storage
  - Enable parallel loading from the storage.
  - For very large LUIs, configure Fabric to distribute entity chunks across different Cassandra nodes.

Big LUIs

Key Considerations for Working with Big LUIs

Cache Storage

- Relocate Cache: For very large LUIs, consider moving the cache directory to separate storage to avoid exhausting memory.

Note:

When using cloud storage for the LUIs, the fetch time and cost are not impacted by the LUI size.

Useful Fabric commands

GET Instance

Parallel GETs for Different LUTs

- Use the following command to concurrently retrieve multiple instances of different LUTs, for a better performance:
  get Customer.1, Crm.1 WITH parallel=true [STOP_ON_ERROR=true/false];
  If one of the instances encounters an error, setting stop_on_error=true will halt the fetching process for the remaining instances.
- The config.ini.[fabric].PARALLEL_GET_POOL_SIZE limits the number of parallel GETs when using this command, per node.
  The default is set to 200.
- Use sync WITH PARALLEL when possible, to reduce the time that the API is waiting for the LUI fetch.

Parallel GETs for the Same LUT

- Only a single instance per LUT can be fetched at a time into a Fabric session.
  To work with 2 LUIs of the same LUT, use the **openFabricSession** command as follows:

  fabric().execute("get Customer.?", cust1);
  openFabricSession*("fabric2");*
  db("fabric2").execute("get Customer.?",cust2);

Useful Fabric commands

GET Instance

getf

GET an instance using a function that returns the IID to fetch.

For example:
getf Customer.fnCreateInstId(235);
fnCreateInstId function adds 1000 to the input value and returns the value 1235. Fabric gets Customer # 1235.

Best practice:

- Avoid executing set sync ON before a GET command as this is already the default (can be configured in config.ini file)
- Avoid executing set sync off after a GET command and before executing the queries, as such queries will not trigger a sync (applicable for Fabric Version 5.x and above).
- Use binding in GET commands: fabric().execute("get Customer.?", cust1);

Useful Fabric commands

Release Instance

Release Instance

Detach the LUI from the session for a list of LUs or for all:

release [<LU_NAME>,<LU_NAME]]

Example:

- Release;
- Release Customer,Collection – release only the 2 LUTs. The rest remains attached.

Note:

The release command will fail if there is an open result set or transaction

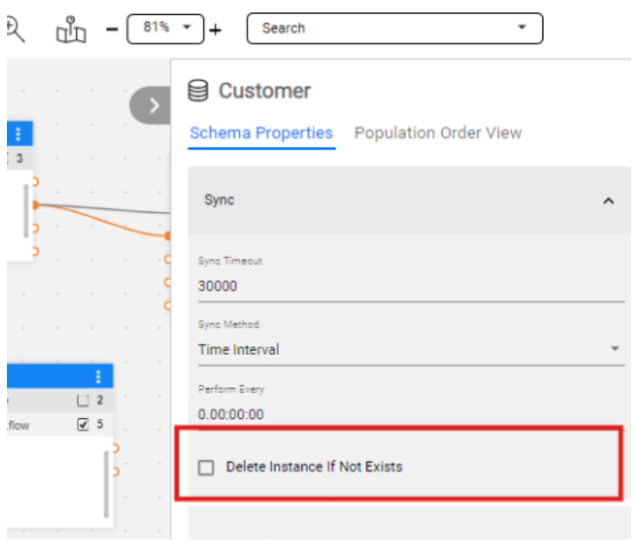
Useful Fabric commands

Delete Instance

Options for deleting an instance:

1. LU Schema property: **DELETE INSTANCE IF NOT EXIST**

    - When set to True, Fabric deletes the LUI if it is not found in the source system (root table does not have data in source).
    - An error is thrown if this is the first GET.
    - When set to False (default), the instance is retained in Fabric even if it is empty.



Sync Timeout: 30000
Sync Method: Time Interval
Perform Every: 0.00:00:00
☐ Delete Instance If Not Exists

2. Fabric command: **delete instances if not exist <LUT_Name>**;
   Delete all LUIs that do not exist in the source system.

To use this command, enable it by setting config.ini.[fabricdb].DELETE_INSTANCES_IF_NOT_EXIST_COMMAND_ENABLED parameter to True.

Useful Fabric commands

Delete Instance

1. Use the command **delete instance <LUT_Name>.'<instance_id>'** [mdbFinder=<false/true>];
   - You can delete multiple instances by adding additional **<LUT_Name>.'<instance_id>'** to the command.
   - mdbFinder = true will also delete from iidf_info table instead of marking it as 'deleted'. When an instance is marked as deleted, iidfinder will discard all its incoming transactions.

2. Use the command **SET INSTANCE_TTL** to set the Time-To-Live (TTL) in seconds for each LUI running in the session;
   The LUI is deleted automatically from Fabric after the TTL ends.

   Note: supported only in case the storage supports TTL functionality.

Note: delete instance do not delete data from IIDFinder cache tables.

k2view

Useful Fabric commands

Select on Source System

Get MDB file size

MDB_SIZE command

Return the actual size (in bytes) of an instance/s blob in storage:

- MDB_SIZE <LUT_NAME>.'<INSTANCE_ID>'
- MDB_SIZE <LUT_NAME>.(<instance 1,instance 2,etc…>)

mdbFileSize function

Returns the length, in bytes, of the file in cache, or 0L if the file does not exist.

fabric>get Customer.1;

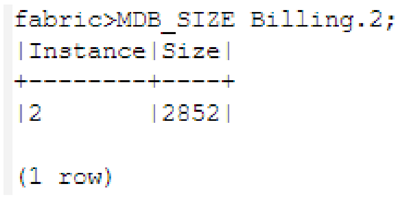
fabric>select mdbFileSize('Customer');

Get MDB file path

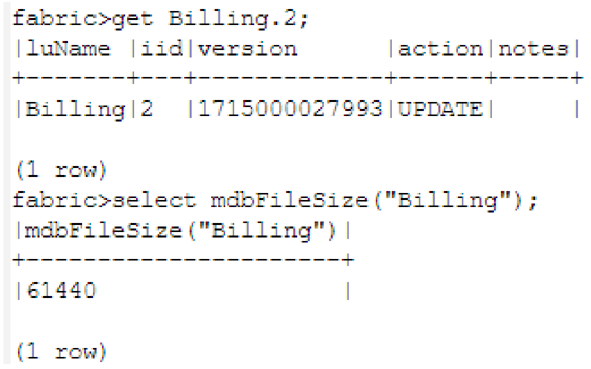
fabric>get Customer.1;

fabric>select mdbFilePath('Customer');

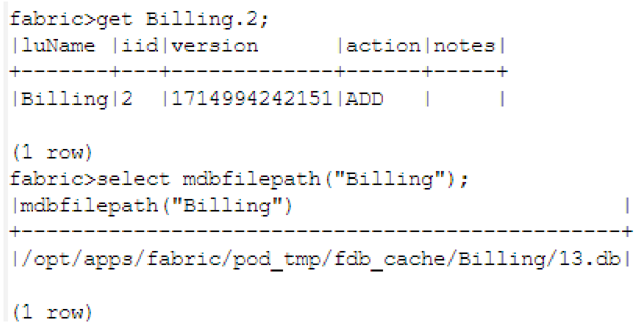
Example



```
fabric>MDB_SIZE Billing.2;
|Instance|Size|
+--------+----+
|2       |2852|

(1 row)
```



```
fabric>get Billing.2;
|luName |iid|version      |action|notes|
+-------+---+-------------+------+-----+
|Billing|2  |1715000027993|UPDATE|     |

(1 row)
fabric>select mdbFileSize("Billing");
|mdbFileSize("Billing")|
+----------------------+
|61440                 |

(1 row)
```



```
fabric>get Billing.2;
|luName |iid|version      |action|notes|
+-------+---+-------------+------+-----+
|Billing|2  |1714994242151|ADD   |     |

(1 row)
fabric>select mdbfilepath("Billing");
|mdbfilepath("Billing")                        |
+----------------------------------------------+
|/opt/apps/fabric/pod_tmp/fdb_cache/Billing/13.db|

(1 row)
```

k2view

Useful Fabric commands

MDB Import/Export

Export/Import the MicroDB

- Back up Fabric data
- Share data with other system
- Import data from external data sources into Fabric.

MDB_EXPORT <LU>[.<IID>] WITH INTERFACE_NAME=<name> [EXCLUDED_TABLES=<TBL1>,<TBL2>] [FK=<false>] [REMOTE_IID=<OTHER-IID>]

- MDB_IMPORT with IID
- MDB_EXPORT with or without IID
  Without IID - create the LUI schema in the target DB, including PK, FK..
  With IID - export the LUI data (schema should already exist on the target DB).

Both import and export commands have an optional parameter, EXCLUDED_TABLES, to specify a list of tables to be excluded from the import/export process.

Useful Fabric commands

MDB Import/Export

SET OUTPUT FILE

Direct SELECT statements' outputs to a file.

SET OUTPUT FILE=<file_name> with arg1=val1 and arg2=val2 |

where args could be:

- DELIMITER
- LINE_TERMINATOR
- HEADER
- QUOTE
- FORCE_QUOTE
- ENCODING
- APPEND

The directory used for the file is set in config.ini.[fabric].EXPORT_DIR

To cancel file redirection and output to the standard output (stdout), use the command:
SET OUTPUT=stdout.

Useful Fabric commands

Select on Source System

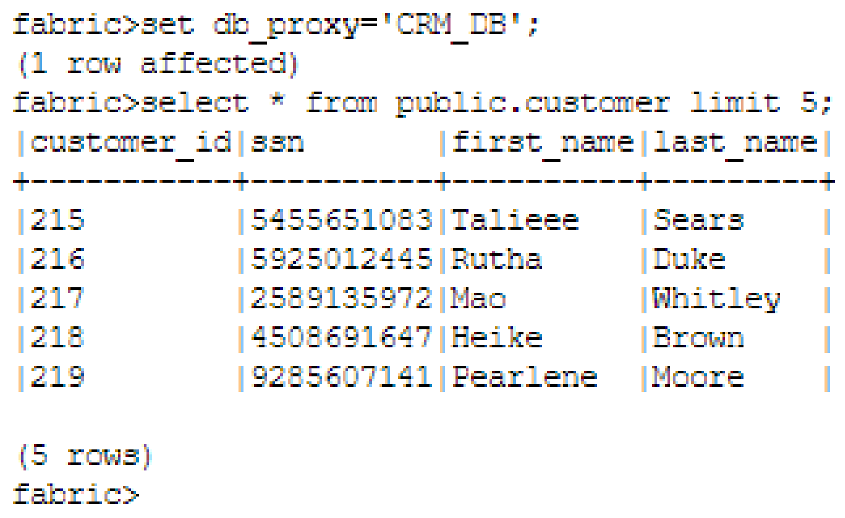
SET DB_PROXY

Activates an operations' scope toward the specified DB interface, so that until it is turned off, all operations are done against this interface.

SET DB_PROXY [= <interface name>]

- To turn it off use: set db_proxy=off.
- To enable this command, set config.ini.[fabric].ENABLE_DB_INTERFACE_PROXY to TRUE.

Example



```
fabric>set db_proxy='CRM_DB';
(1 row affected)
fabric>select * from public.customer limit 5;
|customer_id|ssn       |first_name|last_name|
+-----------+----------+----------+---------+
|215        |5455651083|Talieee   |Sears    |
|216        |5925012445|Rutha     |Duke     |
|217        |2589135972|Mao       |Whitley  |
|218        |4508691647|Heike     |Brown    |
|219        |9285607141|Pearlene  |Moore    |

(5 rows)
fabric>
```

Useful Fabric commands

List

List

List LU_TYPES/LUT [COUNT=<'Y'/'N'> [LU_NAME=<'NAME'>]] [STORAGE=<'Y'/'N'> [LU_NAME=<'NAME'>]]

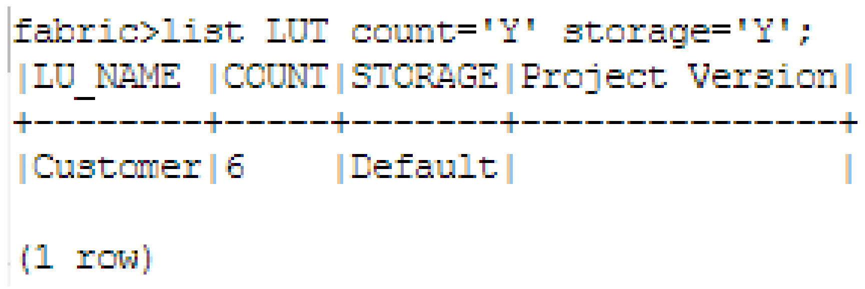
- COUNT - counts the number of instances
- If LU_NAME is added, count only for this LUT.
- Storage - storage property value

List INSTANCES LU_NAME={NAME} - list of Instances per LU name

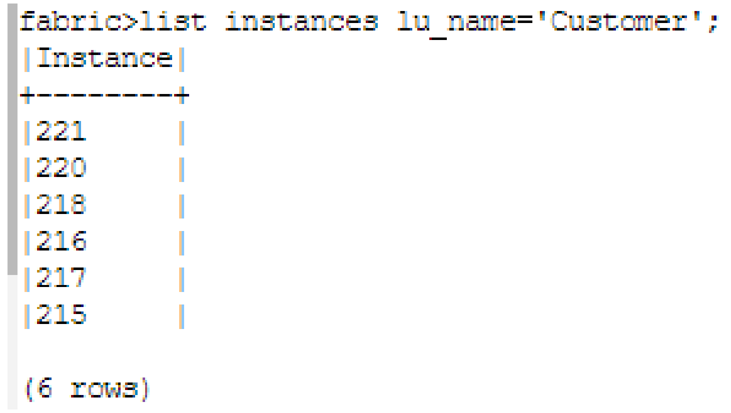
Best practice:

- Count the number of instances:
  If the LUI storage is Cassandra - use the Cassandra COPY command.
  Else, use the LIST command.
- Get the full list of IIDs:
  If the LUI storage is Cassandra – use 'reverse migration' approach.
  Else, Use the LIST command.

Example



```
fabric>list LUT count='Y' storage='Y';
|LU_NAME |COUNT|STORAGE|Project Version|
+--------+-----+-------+---------------+
|Customer|6    |Default|               |

(1 row)
```



```
fabric>list instances lu_name='Customer';
|Instance|
+--------+
|221     |
|220     |
|218     |
|216     |
|217     |
|215     |

(6 rows)
```

Useful Fabric commands

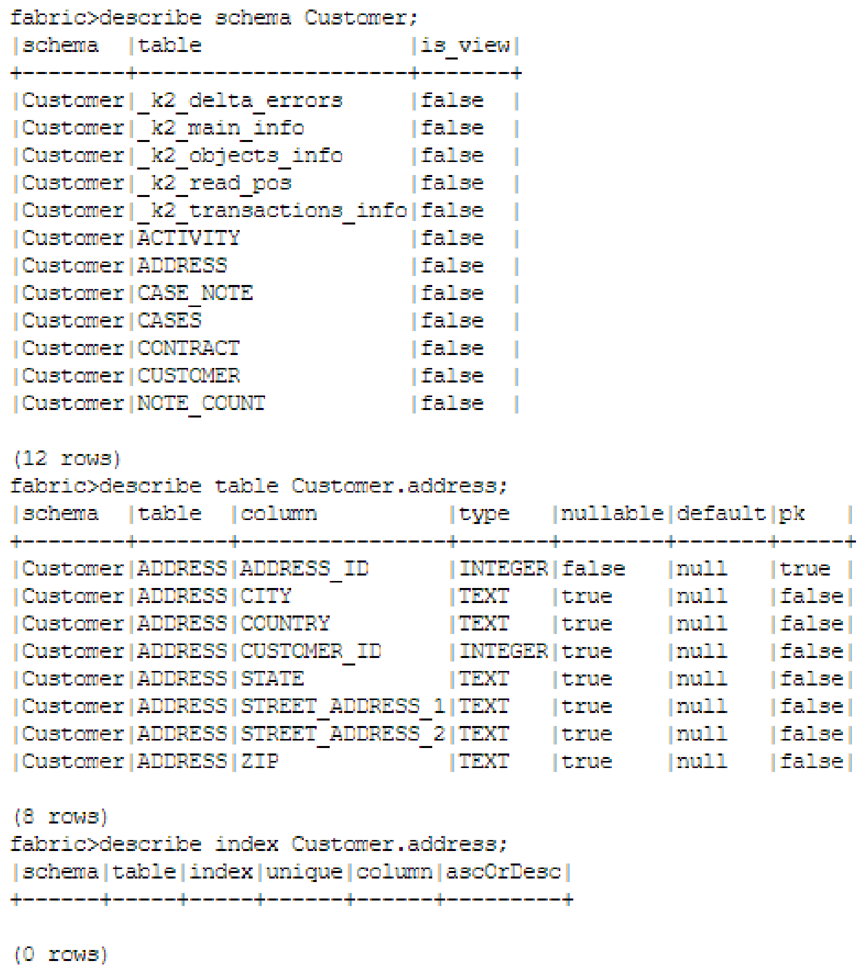
LU Schema Metadata

Describe

The describe command provides LU schema metadata information:

1. list of tables per LUT schema
2. list of columns per table
3. list of indexes per table

Example



```
fabric>describe schema Customer;
|schema  |table                |is_view|
+--------+---------------------+-------+
|Customer|_k2_delta_errors     |false  |
|Customer|_k2_main_info        |false  |
|Customer|_k2_objects_info     |false  |
|Customer|_k2_read_pos         |false  |
|Customer|_k2_transactions_info|false  |
|Customer|ACTIVITY             |false  |
|Customer|ADDRESS              |false  |
|Customer|CASE_NOTE            |false  |
|Customer|CASES                |false  |
|Customer|CONTRACT             |false  |
|Customer|CUSTOMER             |false  |
|Customer|NOTE_COUNT           |false  |

(12 rows)
fabric>describe table Customer.address;
|schema  |table  |column          |type   |nullable|default|pk   |
+--------+-------+----------------+-------+--------+-------+-----+
|Customer|ADDRESS|ADDRESS_ID      |INTEGER|false   |null   |true |
|Customer|ADDRESS|CITY            |TEXT   |true    |null   |false|
|Customer|ADDRESS|COUNTRY         |TEXT   |true    |null   |false|
|Customer|ADDRESS|CUSTOMER_ID     |INTEGER|true    |null   |false|
|Customer|ADDRESS|STATE           |TEXT   |true    |null   |false|
|Customer|ADDRESS|STREET_ADDRESS_1|TEXT   |true    |null   |false|
|Customer|ADDRESS|STREET_ADDRESS_2|TEXT   |true    |null   |false|
|Customer|ADDRESS|ZIP             |TEXT   |true    |null   |false|

(8 rows)
fabric>describe index Customer.address;
|schema|table|index|unique|column|ascOrDesc|
+------+-----+-----+------+------+---------+

(0 rows)
```

Useful Built-in Functions

Package Com.k2view.cdbms.shared.user.UserCode:

- **isFirstSync**() - In the context of sync, indicate if this is the first sync for this instanc
- **isStructureChanged**() - Return true is current LU table had structure change
- **getSyncMode**() - Get current session sync mode
- **skipSync**() - skip current sync, rollback changes done and mark sync as SKIP
- **rejectInstance**(String message) - Reject the current instance in the context of a sync and delete it from Fabric if already exists.
- **setInstanceFound**(boolean isFound) - During sync, use this method to explicitly affect the if the sync should return instance not found
- **getInstanceID**() - Get the instance id of the LU currently attached.
- **getLastSyncTime**() - Get the last sync time of the LU currently attached This works in the context of sync

Useful Built-in Functions

Example:
Decision function on the LU schema level to check the source environment and enable the sync of the LUI in the following scenarios:

- First sync of the LUI.
- LU schema has been changed.
- The source environment is Production.

```
Boolean syncInd = true;


String env = getActiveEnvironmentName(); // Check the source environment


// If the active environment is not production, this sync is not the first sync of the instance, and the schema is not changed => do not sync the instance
if(!env.toLowerCase().equals("prod") && !isFirstSync() && !isStructureChanged())
    syncInd = false;

// getTableName() was added to the log, since when setting a decision function on the LU schema, and all table's sync mode is inherited - it runs on each population of every LU table of the schema
log.info("fnCheckSourceEnv- table name: " + getTableName() + ", active env: " + env.toLowerCase() + ", isFirstSync: " + isFirstSync() +
```

Useful Built-in Functions

Function	Description
getLuType()	Get the LU Type associated with the current context This method gives access to an internal class and is subject to change.
getLuKeyspaceName()	Get the Cassandra Keyspace associated with the LU of the active context.
getPopulationName()	Get the population name of the current map execution.
getTableName()	Get the active table name of the current map execution.
getActiveEnvironmentName()	Get the active environment name for the current session this can be "_dev" for the default environemnt or any other defined in Studio and activated on this server/session
inDebugMode()	Use this method to find out if the current contest is running in the Studio debug interface
openFabricSession(String interfaceName)	Open a new fabric session based on the "fabric" interface credentials.
serializeLU(String startingLudbObjectName)	Serialize the current LU instance into JSON

Design Considerations

1. LU Schema:
   1. How to choose the right IID (lookup if needed)
   2. Movement of data between LUIs handling.
   3. Multi parents handling - data that belongs to cross LUIs.
2. Table Population:
   1. What is the source of the table (DB, File, Kafka)?
   2. How will it be synced for the initial load?
   3. How will it be synced as part of BAU? Which interval is needed?
   4. Are we getting ongoing updates or a full refresh from source?
   5. If on-going updates - what is the delay from source?
   6. What is the expected volume of GG transactions per each table? Are there any peak times?
   7. What types of DML? Only insert / insert+delete / insert+update+delete.
   8. Are there any batch processes running on the source side?
   9. Is there a retention of the data? Is purging needed?
   10. Is any transformation logic required?
   11. Is this table a real table or a joint of a few tables?
   12. How many records are there on the source system? Partitions?
   13. Does a PK exist? Indexes? FK?
   14. Any special column types (BLOB/CLOB)?
   15. Are there PII fields?
   16. Which columns are needed?
   17. Is it a reference table or an application table?
   18. Is it a SOR table?
   19. How is this table being used?

Design Considerations

3. LU Table vs Common Table Decision:
    1. Is the table data associated with many instances, and can it be considered reference data?
    2. How many records does the table contain?
    3. How often is this table expected to be changed?
    4. Full refresh from source or ongoing transaction?
    5. NRT needed?
4. Source APIs:
    1. How to call the API? Authentication?
    2. What is the API output format?
    3. Does it include cross instances data?
    4. How many parallel calls can be executed?
    5. What is the trigger to call the API?
    6. Is there a dependency between API calls or any sequence of the API calls?
    7. Error handling? Failover?
    8. Expected response time? After how long should we raise timeout?
    9. Expected response size?

Design Considerations

5. Files as Source:
    1. How many files per day? Peak time?
    2. Pull or push?
    3. What is the scheduling of the files?
    4. What is the size of the files?
    5. What is the type of files/file format?
    6. Must the files be processed in the order in which they were received?
    7. Must the records in the files be processed by order?
    8. File structure, Header/footer, encryption?
    9. What are the required File/data validations?
    10. Is the File full dump or delta?
    11. Transformation logic?
    12. Files purging logic?
    13. Is the file data related to multiple instances?
    14. Error handling?
    15. How can we identify when the file is ready to commence processing, rather than still undergoing writing operations?

Design Considerations

6. Source DBs
    1. What is the DB type?
    2. Can we connect directly using JDBC?
    3. Can we connect 24/7 or are there any limitations?
    4. How many parallel connections are allowed?
    5. Are there any PK or Unique indexes available on the tables?
    6. Are indexes defined on the tables for the linked fields?
    7. Is it the source DB or replica? If replica, what is the delay?
    8. Is it a one DB instance or multiple? If multiple - what is the logic?
    9. Do we have different schema names in different environments (prod/test)?
7. General
    1. How many instances per LUT do we have?
    2. LUI purging logic?
    3. Is there encrypted data (file or field)? Masking?
    4. How many trail files are expected to be sent per day?

Design Considerations

8. External Interface Assumptions
   1. Fabric will have access to the source DB to extract data. The number of connections required to extract data should be defined as soon as possible, as it may impact the design.
   2. Fabric's ability to access source DBs for a reading activity will not be limited by time.
   3. The schema names for source DB won't change based on the environment.
   4. Source DB will define indexes on the fields used to fetch data into the LU.
   5. Any other restrictions on external interfaces, such as DB, Kafka queue, APIs, etc., should be documented as soon as possible and considered while building the design. Example:
      1. Number of allowed parallel APIs call.
      2. Limited access hours.

Course assignment

Business table

1. Use the below query to populate a new business table, to populate a report:

   select *activity.activity_id, cases.case_id, last_case_note.note_id, activity.activity_date, activity.activity_note, cases.case_date, cases.case_type, cases.status, last_case_note.note_text from activity inner join cases on activity.activity_id = cases.activity_id inner join (select * from case_note inner join (select case_id, max(note_date) max_note_date from case_note group by case_id) max_case_note on case_note.case_id = max_case_note.case_id and case_note.note_date = max_case_note.max_note_date) last_case_note on cases.case_id = last_case_note.case_id where cases.status != 'Closed'*

2. Add logic to run this population only if one of the tables used in the select fetches data from source.
   - Change activity table population to run every 30 sec
   - Change cases to run every 45 seconds
   - Change case_note to run every 1 minute

3. Make sure the business table population is running only if one (or more) of the tables above got refreshed

Course assignment - hints

Business table

1. Actor to use:
   - FabricSet: to set a session global by Actor
2. Functions to use:
   - fabric().fetch("set SESSION_GLOBAL_NAME").firstValue(); to get a session global value
   - fabric().execute("set SESSION_GLOBAL_NAME=?");   to set a session global value, by code