K2 Alpinist

Course 3

Table Population

WELCOME

Agenda

- Introduction to Populations
- Properties of Populations
  - Sync Methods
  - Table Population Mode
  - Truncate before Sync/Delete Mode
- Population Types
- Populations – Behind the Scenes
  - How is the SELECT Statement Constructed on the Source?
  - Opening Connection to Source
  - Parallel Populations
- Sync on Demand
- Data Pulling Best Practice
  - LU Schema structure
  - Populations
  - Source Interface
  - Query Optimization

What is population?

Population defines the source table from which to extract data into the LUI and, if necessary, can incorporate data transformation logic.

Population properties:

- Sync Methos – define **when** the population should run
- Population Mode – define the **logic to apply the data** (records) in the LU table
- Delete Mode – define the **logic to delete** the records from the LU table

Note:

On cloud version the Sync Method moved to the table level and applies for all the populations defined for this table.

Sync Methods

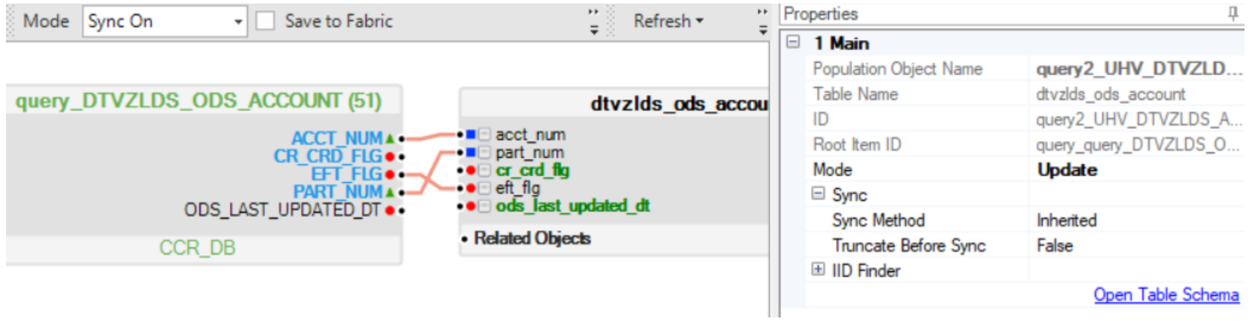
Set WHEN a population should run

- Sync Method
  - **None**: Only runs on first sync or with Sync FORCE
  - **Time Interval**: Runs after a set duration since last run
  - **Decision Function**: Runs if the function returns true
  - **Inherited**: Adopts table or schema sync methods

- Important Notes
  - Methods apply only when Sync is ON
  - Sync OFF means populations will not run
  - Sync FORCE overrides methods, except for decision function
  - First sync acts like Sync FORCE
  - Populations follow their methods, independent of parent table status (if it ran or not)

Population Mode

Logic to apply on the newly fetched data

Population Mode	Description
Insert	Each record extracted from the source is inserted into the LU table using the INSERT operation.
Upsert	Executing **INSERT ON CONFLICT(PK) UPDATE** operation: if a record extracted from the source does not exist in the LU table (based on primary key values), insert is performed. Otherwise, the record is updated. Older versions used **INSERT OR REPLACE** which invoked delete & insert, and therefore sent 2 CDC messages
Update	Update records that are fetched from source. Used to update a specific columns (for example – calculated column). The population is then connected only to the relevant columns. 
Delete	All records will be deleted from the LU table. For example – delete records from Target DB in TDM system

Mode: Sync On — Save to Fabric — Refresh

query_DTVZLDS_ODS_ACCOUNT (51): ACCT_NUM, CR_CRD_FLG, EFT_FLG, PART_NUM, ODS_LAST_UPDATED_DT — CCR_DB

dtvzlds_ods_accou: acct_num, part_num, cr_crd_flg, eft_flg, ods_last_updated_dt — Related Objects

Properties

1 Main	
Population Object Name	query2_UHV_DTVZLD...
Table Name	dtvzlds_ods_account
ID	query2_UHV_DTVZLDS_A...
Root Item ID	query_query_DTVZLDS_O...
Mode	**Update**
Sync	
Sync Method	Inherited
Truncate Before Sync	False
IID Finder	

Open Table Schema

Population Mode

Logic to apply on the newly fetched data

Note:

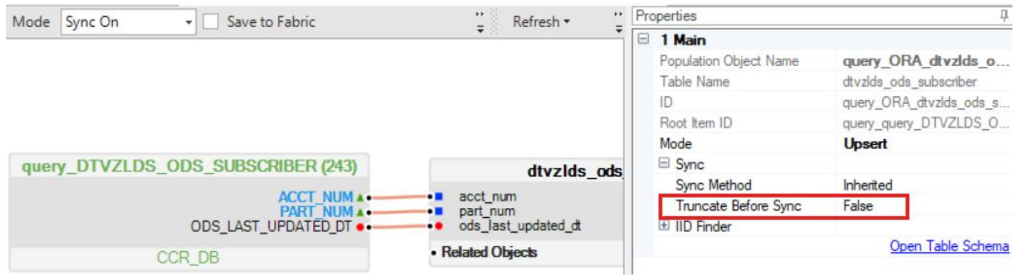
- It is recommended to set up ‘Insert’ mode on the 1st population, followed by Upsert for the next populations
- When using ‘Update’ mode, make sure the tables’ key fields are being marked. Otherwise, all records will be updated on each iteration. As a result, all records will be updated with the last iteration.
  Same for Delete mode

Truncate before sync/Delete Mode

Policy to delete existing records prior to executing the population

Truncate Before Sync (before 6.5.9)

- **True** – truncate the table before the population is executed
- **False** – Records are not deleted from the table



Mode: Sync On | Save to Fabric | Refresh

query_DTVZLDS_ODS_SUBSCRIBER (243): ACCT_NUM, PART_NUM, ODS_LAST_UPDATED_DT — CCR_DB

dtvzlds_ods: acct_num, part_num, ods_last_updated_dt — Related Objects

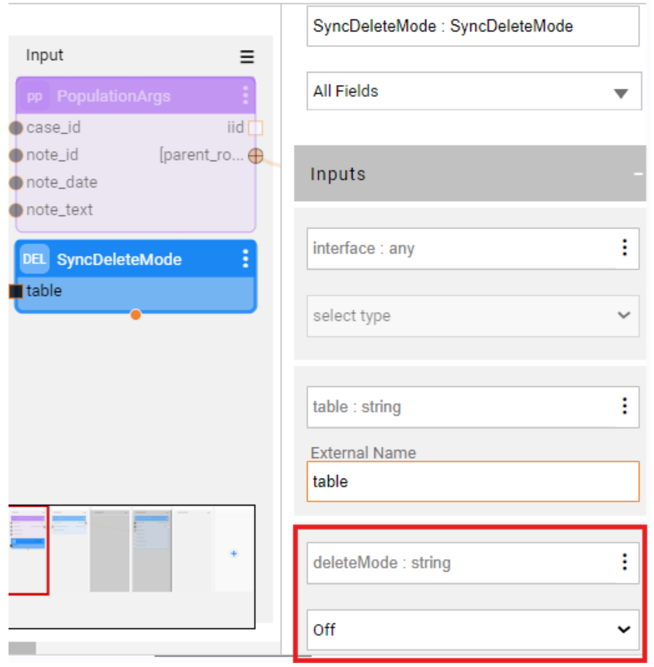
Properties

1 Main	
Population Object Name	query_ORA_dtvzlds_o...
Table Name	dtvzlds_ods_subscriber
ID	query_ORA_dtvzlds_ods_s...
Root Item ID	query_query_DTVZLDS_O...
Mode	Upsert
Sync	
Sync Method	Inherited
Truncate Before Sync	False
IID Finder	

Open Table Schema

Delete Mode (6.5.9 and up)

- **All** (=true) – truncate the table before the population is executed
- **Off** (=false) – Records are not deleted from the table
- **NonUpdated** – delete only records that no longer exist in the source



Input

PopulationArgs: case_id, note_id, note_date, note_text — iid, [parent_ro...

SyncDeleteMode: table

SyncDeleteMode : SyncDeleteMode

All Fields

Inputs

interface : any

select type

table : string

External Name: table

deleteMode : string

Off

Truncate before sync/Delete Mode

NonUpdated Delete Mode

NonUpdated Delete Mode was added to support CDC functionality:

1. *Before the population is executed, Fabric creates a temporary table with the PK values.*
2. *Population is executed – data is fetched from the source and applied to the table.*
3. *Delete records – only records whose PK is not in the temporary table are deleted.*
4. *The temporary table is deleted.*

Note:

To use NonUpdated option:

- PK fields must be defined on the table
- The population mode must be either Upsert or Update

Truncate before sync/Delete Mode

LU Table and Table population

The Delete Mode property exists in both population and table level.

- When set on the table level, the table will be truncated prior to executing any of the table's populations.
- When set on the population level (to a value other than OFF), the delete functionality will be triggered **only if and when the population is executed**, and it is operating on the entire table's records.

  NOTE: The 'Truncate' setting on the population level truncates the **entire table** (for example, even if it was set on the 3rd population).

Best practice:

Truncate a table in case the population extracts all data for a given instance. Otherwise (population extracts delta from the source), do not truncate the table.

k2view

Population types

Differences between DB query, Root query and Broadway population

Category	DB Query	Root function	Broadway flow
Content	SQL statement	Java function	
Data filter	WHERE statement is added automatically	No automatic filtering	WHERE statement is added automatically
Population Execution times	One time	Executed for every distinct parent link value	One time
Times fetching data from source	Population is getting distinct values. Each SQL query statement is using MAX_SOURCE_QUERIES_GROUPING number of distinct values (Fabric keeps prepared statement), until all distinct values are used. The remaining distinct values will be used in one query (also as prepared statement)	Custom implementation	Group values to a single query according to the sourceDbQuery.size parameter (same concept as in DB query).

Population types

Root population

Root function population is executed for each distinct value of the parent table linking values.

If the root function is not using the INPUT fields (the distinct parent values) as part of its logic and therefore should not be executed more than one time, consider the below:

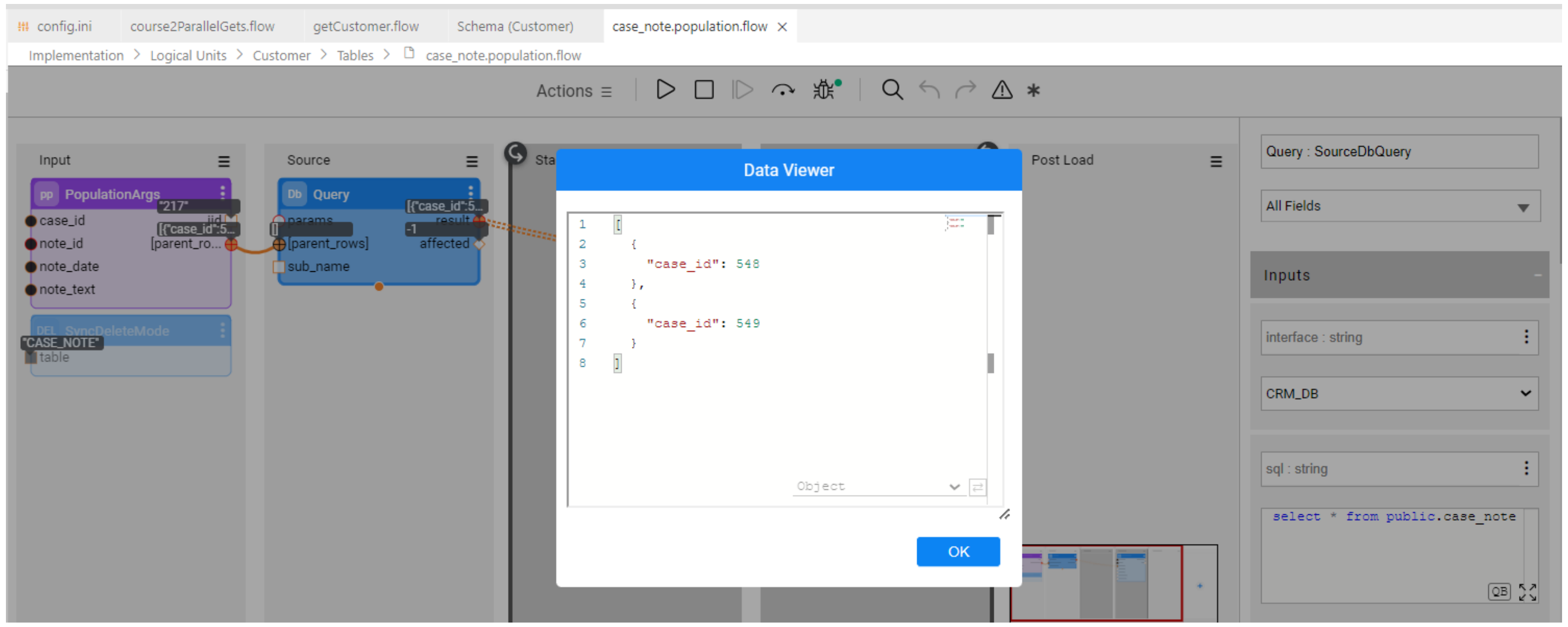
- Connect the table to the LU root table to have it running one time only.
- If connecting the population to the root table cannot be done due to other logical constraints (for example a delete orphans functionality), a thread global should be used to make sure this population is executed only once.

Populations – behind the scenes

How is the SELECT Statement Constructed on the Source?

Each population is using the **SourceDbQuery** Actor that builds the SELECT statement to run on source, ensuring that only records belong to this instance are retrieved:

- Receives 'parent_rows' input parameter, holding distinct values of the parent linking fields
- Add WHERE statement with the distinct values



```
[
  {
    "case_id": 548
  },
  {
    "case_id": 549
  }
]
```

```
select * from public.case_note
```

Select * from case_note where case_id in (548,549)

Populations – behind the scenes

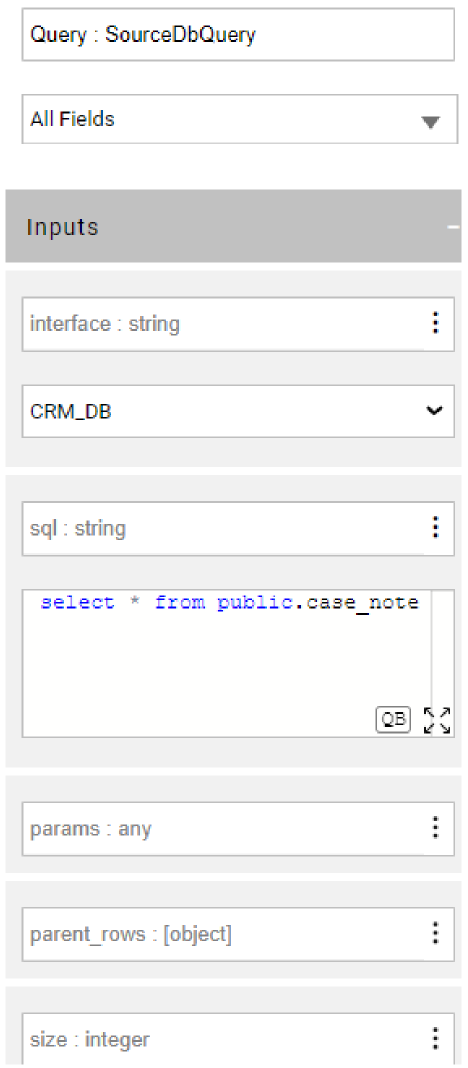
How is the SELECT Statement Constructed on the Source?

The number of values included in the IN statement depends on the **Size** parameter of the SourceDbQuery actor. By default, the Size is set to 100.

Where linkField1 in (?, ?, ?, ?,.... ?)

If more than a single field is used in the link to the parent, logical OR used for multiple record linkage:

Where (linkField1=? And linkField2=?) or (linkField1=? And linkField2=?) or (linkField1=? And linkField2=?)...



Query : SourceDbQuery

All Fields

Inputs

interface : string

CRM_DB

sql : string

```
select * from public.case_note
```

params : any

parent_rows : [object]

size : integer

Populations – behind the scenes

Opening Connection to Source

DB Interface Pool

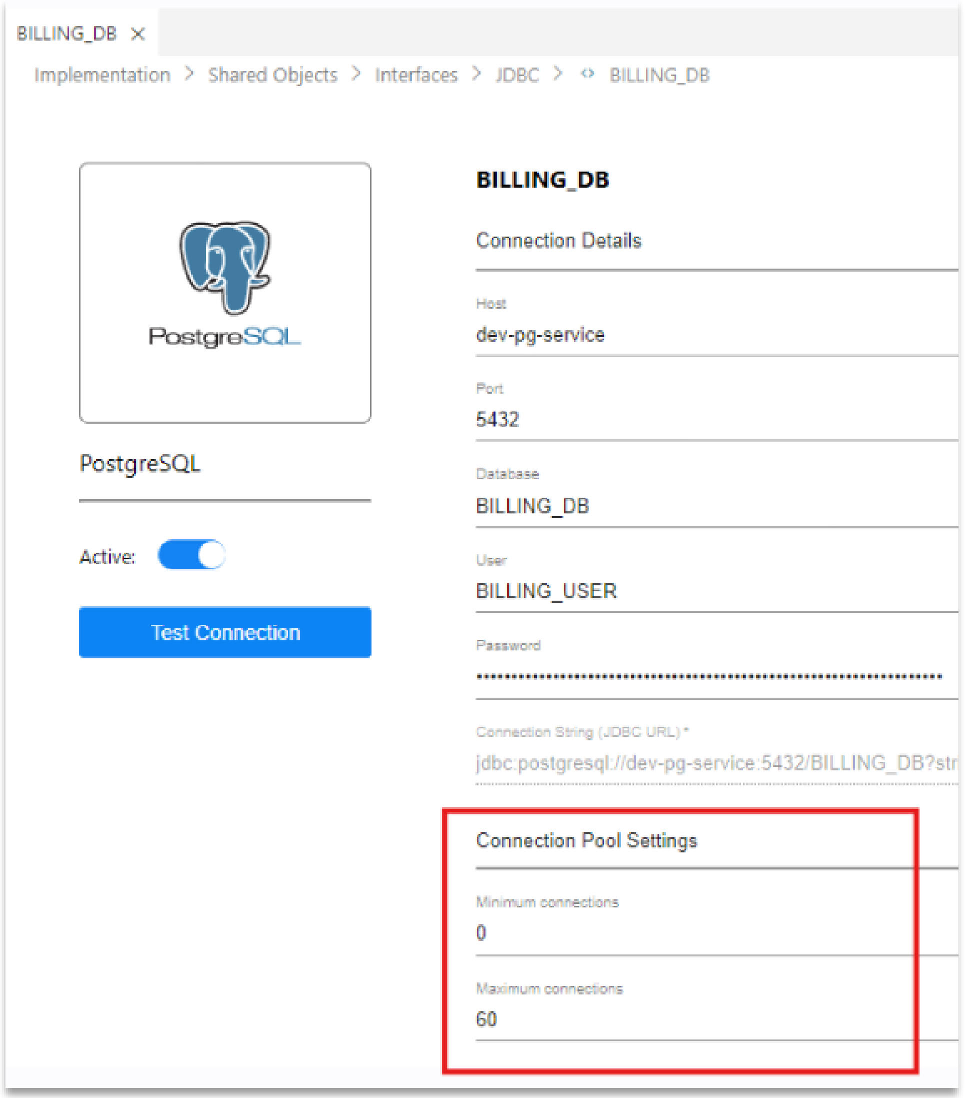
Every population establishes a connection to the source system via the DB Interface.

To optimize performance, each node maintains a pool of open connections for each DB interface.

- The default pool size is defined in config.ini.[fabric].MAX_CONNECTIONS_TO_SOURCE. The definition is per node.

- You can override this value using the DB Interface properties in the studio:

  - **Min Connection Pool**: number of connections kept open, even if not in use.
    Note: the minimum connections are opened only upon first request, and not automatically on Fabric start.
  - **Max Connection Pool** - max number of connections allowed to source system.


BILLING_DB ×
Implementation > Shared Objects > Interfaces > JDBC > BILLING_DB
PostgreSQL
Active:
Test Connection

BILLING_DB

Connection Details

Host: dev-pg-service

Port: 5432

Database: BILLING_DB

User: BILLING_USER

Password: ••

Connection String (JDBC URL) *: jdbc:postgresql://dev-pg-service:5432/BILLING_DB?str

Connection Pool Settings

Minimum connections: 0

Maximum connections: 60

Populations – behind the scenes

Opening Connection to Source

Population Use of the DB Interface Pool

- When the GET command is executed, populations within the LUIs are executed sequentially according to a predefined order.
- Each population is associated with a particular DB Interface.
- The population requests a connection from the connection pool:
  - If an available connection exists, it will be utilized.
  - If there are no available connections but the pool isn't at capacity, a new connection is opened.
  - If the pool is at capacity, the process waits until a connection becomes available.
    Timeout to wait on connection is hard coded 30 seconds.

- Idle connection will be removed from the pool after 60 seconds.
  The time can be configured though the driver parameters.

Populations – behind the scenes

Opening Connection to Source

- Every connection is returned to the pool after each use (either db(…) or any BW DB actor).

  If two populations are using the same DB interface, each one will request its own connection from the pool and return it once done (most likely the second population will get the same connection instance as the first one).
- This can be controlled by a parameter in the config 'ENABLE_SELF_CLOSE_CONNECTION'.

JMX Stats:

1. jdbcActiveSessions
2. dbcIdleSessions - idle connections in the pool
3. JdbcMaxSessions – max connections as configured for the pool

Note: No available way to check waiting on the pool

Populations – behind the scenes

Opening Connection to Source

Parallel Populations

By default, the populations with the same execution order are running sequentially.

To improve the LUI sync time, Fabric supports parallel syncs of multiple populations within the same execution order.

Use config.ini. [fabric] .MAX_PARALLEL_SYNC_SAME_ORDER (default value = 1) to set the desired number of parallel populations execution.

Note:

- Although the populations are fetching the data in parallel, the data write to the LUI remains sequential.
- When two populations are running in parallel, they will use the connection in turns, giving each other time to read on write.
- The capability to share the same connection relies on the JDBC driver's support for multithreading & multiplex.

Parallel population best practice:

Allowing more than a single population to run in parallel means consuming more resources by a single LUI sync, which may therefore block other LUIs that are syncing in parallel.

Sync on Demand

The purpose of Sync On Demand mode is to reduce Fabric's LUI sync time by only synchronizing relevant data.

Sync On Demand logic:

- Executing the GET command does not trigger instance synchronization (like SYNC OFF mode).
    - If the instance doesn't exist in Fabric, a full sync is performed.
- When executing SELECT statements on LU tables, an evaluation determines if a sync is needed.
    - This evaluation applies only to the LU tables in the SELECT statement and their parent tables up to the Root table.
    - Synchronization follows the standard Sync mechanism rules based on LU's predefined sync method and mode.

Sync On Demand can run in two modes:

- **True**: Each table can be synchronized only once per GET, even if multiple SELECT statements are executed and the source table changes.
- **Always**: Each table can be synchronized on each SELECT, assuming the sync conditions are met..

Sync on Demand

To set Sync On Demand mode:

- Set the **`SYNC_ON_DEMAND`** parameter in the `config.ini` file to `True` (default is `False`).
- Run the **`SET SYNC_ON_DEMAND = [TRUE/FALSE/ALWAYS]`** command to set it at the session level.

NOTE:
The implementor should manage transactions efficiently. When a Web Service or GraphIt invokes multiple SELECT statements on the same LU, it is the implementor's responsibility to minimize writes to the MDB Storage.

For example - several SELECTs in a Web Service or GraphIt:

1. Set Sync On Demand to either true or always, whatever is required.
2. Perform GET LUI.
3. Begin the transaction.
4. Perform all the required SELECT statements:
5. On each SELECT, Fabric checks whether sync should be performed (based on sync mode and sync method). If it should, the relevant tables are synchronized - as per the above logic.
6. Commit the transaction.

Data Pulling Best Practice

LU Schema Structure

1. Minimize LU Table Columns – include only columns that are needed
   - Reduces fetch and data transmission over the network
   - Decreases LUI size for faster db operations
2. Connect parent-child tables using the minimum required fields to have a better performant query statement.

For example:
Activity table contains customer_id, activity_id.
Activity_history table contains customer_id, activity_id, history_id.

We can link these tables using both customer_id and activity_id.
However, if we require all customer's records in the activity_history table, it is sufficient to connect the 2 tables using only the customer_id field.

3. A table should always be connected to the parent that will minimize the number of times that the population will go to source.

For example:
connect a table to Account (using account_id) instead on connecting it to Subscriber table (using subscriber_id), as anyway all the subscribers belong to the same account.

4. Avoid creating indexes on tables before conducting performance testing. Indexes are typically necessary for very large datasets and can potentially slow down Data Manipulation Language (DML) commands.

Data Pulling Best Practice

LU Schema Structure

5. Always consider the order in which the table is populated.
6. Mark the Key fields of each table and set them as a unique index.

Data Pulling Best Practice

Populations

1. Ensure that there is an index on the source side that corresponds to the WHERE statement executed by the population.

2. Ensure that linking fields are of the same type in order to prevent the source from converting them during fetch operations. For instance, if a parent field of type String is linked to a child table field of type int, certain database types may result in failure, while others may convert the String to Int, potentially slowing down the fetch performance.

3. Fetch from source only the records that are needed.
   For example, if accounts that are closed are not required, do not fetch ‘closed’ accounts.

4. If a population is accessing the source system many times (i.e., the distinct values of the parent tables exceed the Size parameter), consider changing the Size parameter to reduce the number or roundtrips. However, increasing it excessively may exhaust the network bandwidth and potentially cause blockages for other processes requiring it.

Data Pulling Best Practice

Source Interface

1. Make sure the DB Interface connection pool is sized as required, in order not to wait for connection to source when running multiple sync processes at the same time.

Table Population Best Practice

Query Optimization

1. **Creating Relevant Indexes**
   - Create indexes based on your query requirements to improve SELECT statement performance.
   - Note: While indexes enhance SELECT performance, they can slow down INSERT, UPDATE, and DELETE operations. Use EXPLAIN QUERY PLAN to validate the utilization of the correct indexes.
2. **Avoiding Index Manipulation in Queries**
   - Avoid applying additional manipulation or transformation on index fields within queries, as this can prevent the index from being used. Example: Concatenating two fields in a WHERE clause will not utilize the index on those fields.
3. **Enforcing Index Utilization**
   - If necessary, enforce index utilization in the query using INDEXED BY.
   - If parsing fails, use /* sqlite */ or /* k2_no_parse */ before the SELECT statement, depending on the Fabric version.
4. **Simplifying Queries**
   - Simplify queries to achieve better performance and readability.
   - Avoid excessive use of the JOIN operator in a single query; consider splitting complex queries into simpler ones.
5. **Optimizing UNION Operator Usage**
   - Limit the use of the UNION operator and prefer UNION ALL when the data retrieved by sub-queries is unique. This eliminates the need for an additional DISTINCT step, enhancing performance.
6. **Validating Record Existence**
   - To validate that a record exists, select the first row with the required WHERE condition using LIMIT 1 or ROWNUM < 2, depending on the database.
   - Avoid using COUNT(*) in queries, as it can be time-consuming and impact performance.

Course assignment

Sync a specific table(s) from source

Design a solution to sync a specific table(s) from source, while the system is up & running in production

1. The solution should apply to a specific LU table(s)
2. Each LUI should sync those tables only one time (next GET shouldn't sync them again)
3. Avoid deployment before every fix
4. Minimal to no client impact
5. Minimize the performance impact
6. Data returned to client should be fixed as fast as possible
7. During fix of one table, we may find another table to be fixed