Running the DQD in SqlOnly mode

Maxim Moinat

2024-12-22

Contents

1 Description

This article describes how to use DQD to generate only the SQL that executes all DataQualityDashboard checks, without actually executing them. There are a few main advantages of running DQD in Sql-only mode:

- Create queries locally, before sending to server. This allows for generation of the SQL on one machine and execution on another (e.g. when R cannot connect directly to the database server, or you want to run the DQD SQL as part of your ETL).
- Since these are fully functional queries, this can help with debugging.
- [**NEW in v2.3.0!**] Performance. If you use `sqlOnlyIncrementalInsert = TRUE` and `sqlOnlyUnionCount > 1`, multiple checks are unioned within a cte in the output SQL query to speed performance. When testing on Spark, this resulted in a 10x or higher performance gain.
  - Performance for these queries has NOT been benchmarked on all database systems. In order to obtain optimal results in your database you may need to adjust the `sqlOnlyUnionCount` and/or tune database parameters such as indexing and parallelism

The new `sqlOnlyIncrementalInsert` mode generates SQL queries that will actually populate a DQD results table in your database with the results of the checks. There are currently some differences in the result when running these queries, compared to a normal DQD run:

- If you set `sqlOnlyUnionCount` > 1, if one check results in an error, multiple checks might fail (since the queries are unioned in ctes).
- The status `not_applicable` is not evaluated. A check fails or passes.
- The query text is not shown in the results table.
- Notes from threshold file are not included in results.
- Execution metadata is not automatically added (total and query execution time; CDM_SOURCE metadata).

Running DQD with `sqlOnly = TRUE` and `sqlOnlyIncrementalInsert = FALSE` will generate SQL queries that can be run to generate the result of each DQ check, but which will not write the results back to the database.

2 Generating the "Incremental Insert" DQD SQL

A few things to note:

- A dummy `connectionDetails` object is needed where only the `dbms` is used during SQL-only execution.
  - By setting the dbms to 'sql server' the output SQL can still be rendered to any other dialect using `SqlRender` (see example below).
- `sqlOnlyUnionCount` determines the number of check sqls to union in a single query. A smaller number gives more control and progress information, a higher number typically gives a higher performance. Here, 100 is used.

```r
library(DataQualityDashboard)

# ConnectionDetails object needed for sql dialect
dbmsConnectionDetails <- DatabaseConnector::createConnectionDetails(
  dbms = "sql server",  # can be rendered to any dbms upon execution
  pathToDriver = "/"
)

# Database parameters that are pre-filled in the written queries
# Use @-syntax if creating a template-sql at execution-time (e.g. "@cdmDatabaseSchema")
cdmDatabaseSchema <- "@cdmDatabaseSchema"   # the fully qualified database schema name of the CDM
resultsDatabaseSchema <- "@resultsDatabaseSchema"   # the fully qualified database schema name of the r
writeTableName <- "@writeTableName"

sqlFolder <- "./results_sql_only"
cdmSourceName <- "Synthea"

sqlOnly <- TRUE
sqlOnlyIncrementalInsert <- TRUE    # this will generate an insert SQL query for each check type that w
sqlOnlyUnionCount <- 100            # this unions up to 100 queries in each insert query

verboseMode <- TRUE

cdmVersion <- "5.4"
checkLevels <- c("TABLE", "FIELD", "CONCEPT")
tablesToExclude <- c()
checkNames <- c()

# Run DQD with sqlOnly=TRUE and sqlOnlyIncrementalInsert=TRUE. This will create a sql file for each che
DataQualityDashboard::executeDqChecks(
  connectionDetails = dbmsConnectionDetails,
  cdmDatabaseSchema = cdmDatabaseSchema,
  resultsDatabaseSchema = resultsDatabaseSchema,
  writeTableName = writeTableName,
  cdmSourceName = cdmSourceName,
  sqlOnly = sqlOnly,
  sqlOnlyUnionCount = sqlOnlyUnionCount,
  sqlOnlyIncrementalInsert = sqlOnlyIncrementalInsert,
  outputFolder = sqlFolder,
  checkLevels = checkLevels,
  verboseMode = verboseMode,
  cdmVersion = cdmVersion,
  tablesToExclude = tablesToExclude,
  checkNames = checkNames
)
```

After running above code, you will end up with a number of sql files in the specified output directory:

- One sql file per check type: `TABLE|FIELD|CONCEPT_<check_name>.sql`.
- `ddlDqdResults.sql` with the result table creation query.

The queries can then be run in any SQL client, making sure to run `ddlDqdResults.sql` first. The order of the check queries is not important, and can even be run in parallel. This will run the check, and store the result in the specified `writeTableName`. In order to show this result in the DQD Dashboard Shiny app, this table has to be exported and converted to the .json format. See below for example code of how this can be achieved.

3 (OPTIONAL) Execute queries

Below code snippet shows how you can run the generated queries on an OMOP CDM database using OHDSI R packages, and display the results in the DQD Dashboard. Note that this approach uses two non-exported DQD functions (`.summarizeResults`, `.writeResultsToJson`) that are not tested for this purpose. In the future we plan to expand support for incremental-insert mode with a more robust set of public functions. Please reach out with feedback on our GitHub page if you'd like to have input on the development of this new feature!

```r
library(DatabaseConnector)
cdmSourceName <- "<YourSourceName>"
sqlFolder <- "./results_sql_only"
jsonOutputFolder <- sqlFolder
jsonOutputFile <- "sql_only_results.json"

dbms <- Sys.getenv("DBMS")
server <- Sys.getenv("DB_SERVER")
port <- Sys.getenv("DB_PORT")
user <- Sys.getenv("DB_USER")
password <- Sys.getenv("DB_PASSWORD")
pathToDriver <- Sys.getenv("PATH_TO_DRIVER")
connectionDetails <- DatabaseConnector::createConnectionDetails(
  dbms = dbms,
  server = server,
  port = port,
  user = user,
  password = password,
  pathToDriver = pathToDriver
)
cdmDatabaseSchema <- '<YourCdmSchemaName>'
resultsDatabaseSchema <- '<YourResultsSchemaName>'
writeTableName <- 'dqd_results' # or whatever you want to name your results table

c <- DatabaseConnector::connect(connectionDetails)

# Create results table
ddlFile <- file.path(sqlFolder, "ddlDqdResults.sql")
DatabaseConnector::renderTranslateExecuteSql(
  connection = c,
  sql = readChar(ddlFile, file.info(ddlFile)$size),
  resultsDatabaseSchema = resultsDatabaseSchema,
  writeTableName = writeTableName
)

# Run checks
```

```
dqdSqlFiles <- Sys.glob(file.path(sqlFolder, "*.sql"))
for (dqdSqlFile in dqdSqlFiles) {
  if (dqdSqlFile == ddlFile) {
    next
  }
  print(dqdSqlFile)
  tryCatch(
    expr = {
      DatabaseConnector::renderTranslateExecuteSql(
        connection = c,
        sql = readChar(dqdSqlFile, file.info(dqdSqlFile)$size),
        cdmDatabaseSchema = cdmDatabaseSchema,
        resultsDatabaseSchema = resultsDatabaseSchema,
        writeTableName = writeTableName
      )
    },
    error = function(e) {
     print(sprintf("Writing table failed for check %s with error %s", dqdSqlFile, e$message))
    }
  )
}

# Extract results table to JSON file for viewing or secondary use

DataQualityDashboard::writeDBResultsToJson(
    c,
    connectionDetails,
    resultsDatabaseSchema,
    cdmDatabaseSchema,
    writeTableName,
    jsonOutputFolder,
    jsonOutputFile
  )


jsonFilePath <- R.utils::getAbsolutePath(file.path(jsonOutputFolder, jsonOutputFile))
DataQualityDashboard::viewDqDashboard(jsonFilePath)
```