

THE VAMDC Query Store Implementing Note

By Carlo Maria Zwölf



TABLE OF CONTENT

1	Introduction.....	3
2	The Query Store entry point: the <i>NotificationListener</i> service.	3
3	The Query processing pipeline.....	5
3.1	The pipeline pattern.....	5
3.1.1	The StateVAMDCQueryUniqueness class.....	6
3.1.2	The NewVAMDCQueryProcessor class.....	7
3.1.3	The ExistingVAMDCQueryProcessor class.....	8
3.1.4	The VamdcDataFileDownloader class.....	8
3.1.5	The VamdcReferenceGetter class.....	9
3.1.6	The VamdcReferenceGetter class.....	10
3.2	Handling the errors of the processing pipeline: the ErrorHandler class	10
4	The other web services of the Query Store.	11
4.1	The AssociationService.....	11
4.2	The PortalAssociationService.....	12
4.3	The InfoQuery service.....	14
4.4	The FindQueriesByTime service.....	15
4.5	The getUUIDByToken service.....	16
5	The Purger Daemon	16
6	The internal database schema.....	17
7	The source code package structure.....	18

1 Introduction

This document is a technical deepening of the general overview given in <https://github.com/VAMDC/QueryStore/blob/master/Readme.md>. As we explained in this last document (cf. paragraph “Main issues in implementing the Recommendation to the VAMDC case”), the VAMDC Query Store may be seen as a kind of log-service. The easiest way for understanding its functioning is “walk the way” of a query notification.

The VAMDC Query Store is a set of web services developed using the Java Servlet technology. For security reasons the Query Store should be exposed using only the HTTPS protocol. The Query Store uses a my-SQL internal database for storing its data. Other databases (including non SQL ones) may be used without affecting the functional logic and behavior of the Query Store.

2 The Query Store entry point: the *NotificationListener* service.

The servlet *NotificationListener*¹ implements the entry point listening for query notifications. An entity wishing to notify to the Query Store a query will invoke (by post or get) the *NotificationListener* by providing the following fields:

- *accededResource* : this is the name of the resource the query is extracting the information from. In the VAMDC landscape this corresponds to the VAMDC data-node (which is a particular database exposing its data through the VAMDC infrastructure). This field is mandatory
- *resourceVersion*: this is the version of the resource the query is extracting the information from. This field is mandatory
- *userEmail*: this is the email of the users running the query. This field is optional
- *usedClient*: this is the name of the client used by the users for extracting the information from the “acceded resource”. This field is optional
- *accessType*: in VAMDC there is a difference between a “HEAD” query (submitted for discovering if there is data associated to a given query and internal infrastructure monitoring) and a “GET” query (submitted for getting

¹ src/org/rda/QueryStore/NotificationListener.java

the data associated to a given query). This parameter is a flag for specifying the type of the query. This field is mandatory.

- **outputFormatVersion**: This is the version of the standards (<https://github.com/VAMDC/Standards>) used by the “**accededResource**” for formatting the output data obtained with the query. This field is mandatory
- **dataURL**: the data generated by the query may be downloaded from this URL.
- **queryToken**: This is a unique identifier generated by the “**accededResource**” for each served query. This identifier is mandatory and is built as follow: ‘short name of the resource’:UUID:‘accessType’.
- **secret**: this is a password for verifying that the entity notifying the query is allowed to use the Query Store.
- A set of couple (parameterName, parameterValue), which defines the query as it, was received/processed by the “**accededResource**”.

When the **NotificationListener** servlet receives a notification (formatted as explained in the previous items-list) it executes the **getNotificationDetails** function which:

- Get the Unix-timestamp of the moment when the notification arrived to the Query Store, get the IP of the notifying entity and, starting from the information associated with the service invocation, builds the internal object model (instantiation of a **NotificationDetail** object).
- Checks, by comparing with the password stored into the Query Store internal database, if the notifying entity is allowed to use the Query Store.
- It checks if the notification is compliant with the rules and policies defined by the Query-Store owner(s) (by invoking the function **isNotificationCompliant** of the class **NotificationVerifier**).
 - Mandatory fields are checked
 - The Query Store owner(s) may want to introduce restrictions on some fields. For example he/she may decide to accept only queries submitted with a specific client and/or formatting outputs with a given specific version of standards. These restrictions may be easily defined by adding some lines into the table **AuthorizedFields** of the internal database (cf. section 6&): let us consider a field registered into this table; a notification is then compliant if and only if the notified field value is equal to (at least) one of the **authorizedValues** defined for this field. If there is no restriction defined, the notified field may take any value.
 - The set of the couple (parameterName, parameterValue) are checked. A mechanism similar to what described for the restrictions on fields: the Query Store owner(s) may want to accept only some specific parameters. They will define the list of the accepted parameters into

the *AuthorizedParameters* table of the internal database. A given parameter is authorized if and only if it is defined into this table. If this particular table is empty, there is no restriction on the parameters.

- If the notifying entity has a wrong password and/or the notification is not compliant, then the service returns the http 400 code.
- If the notifying entity has the good password and the notification is compliant, then the service
 - Return the http 200 code
 - Persist into the internal database the information associated with the notification (table *Notifications*).
 - Instantiate and run the pipeline for processing the notified information.

3 The Query processing pipeline

One of the goals of the development of the Query Store was to build a software layer that can be easily adapted to other projects and configurations beyond VAMDC. The particular design of the pipeline processing the queries guarantees the generality and flexibility for easily adapting our development to other projects.

3.1 The pipeline pattern

The *PipelineTask*² class is the core element of the processing mechanism. This is an abstract class containing:

- An attribute *followingTask*, which is an object of *PipelineTask* type too.
- An abstract method *runCurrentTask*
- An abstract method *instanciateFollowingTask*.
- A method *executeTask*. This sequentially runs the methods *runCurrentTask* and *instanciateFollowingTask* and, if the attribute *followingTask* is initialized, it executes the *executeTask* method on the *followingTask* object. This method may raise *PipelineNodeException*³ exceptions.

With this particular pattern, even complex pipeline may be divided into atomic simple tasks extending the abstract *PipelineTask* class:

- The task specific processing will be performed into the *runCurrentTask* method.

² src/org/rda/QueryStore/business/pipeline/tasks/PipelineTask.java

³ src/org/rda/QueryStore/business/pipeline/tasks/PipelineNodeException.java

- The business logic for building the further steps of the pipeline will be performed into the *instanciateFollowingTask* method.
- This is an iterative process: the execution of the *runFollowingTask* will trigger the execution of the *currentTask* of the *followingTask* object.

The *Pipeline*⁴ class is just a container for the attribute *rootTask* (of *PipelineTask* type): calling the method *executeTask* of the *rootTask* object will run the entire pipeline.

The *rootTask* of the Pipeline object built by the *NotificationListener* servlet (cf. last item of the paragraph 2) is an instance of the *StateVAMDCQueryUniqueness* class.

3.1.1 The *StateVAMDCQueryUniqueness* class

The class *StateVAMDCQueryUniqueness*⁵ is an extension of *PipelineTask*. The goal of this atomic task is to state if an incoming query notification corresponds to a completely new query or if an identical query has been processed before. Its constructor receives as an argument an object of type *NotificationDetail*, which contains all the information to process.

- In the *runCurrentTask* method
 - Parameters are put into a canonical form, using the software component that has been ad hoc developed for parsing SQL-based VAMDC queries (<https://github.com/VAMDC/VamdcSqlRequestComparator>).
 - We look into the internal database (table *Queries*) searching for an eventual already stored table having
 - The same canonical representation
 - Submitted to the same version of a given VAMDC Node (i.e. hitting the same *accededResource* with the same *resourceVersion*)
 - Formatting the results following the same version of the standards (i.e. same *outputFormatVersion*).
 - If such a query exists, we get its unique identifier (UUID).
- In the *instanciateFollowingTask*
 - If the query is new, we instantiate a new object of type *NewVAMDCQueryProcessor* and associate it with the attribute *followingTask*.

⁴ `src/org/rda/QueryStore/business/pipeline/Pipeline.java`

⁵ `/src/org/rda/QueryStore/business/pipeline/tasks/StateVAMDCQueryUniqueness.java`

- If the query already exists, we instantiate a new object of type *ExistingVAMDCQueryProcessor* and associate it with the attribute *followingTask*.

3.1.2 The NewVAMDCQueryProcessor class

The *NewVAMDCQueryProcessor*⁶ class is an extension of *PipelineTask*. As its name indicates, the goal of this atomic task is to process the new queries (never processed before). Its constructor receives as an argument an object of type *NotificationDetail*, which contains all the information to process.

- In the *runCurrentTask* method
 - An UUID is generated as persistent identifier for the new query, which is being processed.
 - We persist into the internal database the information concerning the new query:
 - We store into the table *Queries* the information about the query: the query-UUID, the *accededResource*, the *resourceVersion*, the *outputFormatVersion* and the set of the parameters into the canonical form.
 - We store into the table *QueryUserLink* the information concerning the query execution: the query-UUID, the Unix-timestamp of the moment when the query notification arrived to the Query Store, the set of original parameters submitted by the users (not in the canonical form), the user email (if provided), the client software used for submitting the query (if provided) and the query token generated by the node software.
- In the *instanciateFollowingTask* method
 - If the query submitted to the Node is of type 'HEAD', there is no following task to set. In this case the *NewVAMDCQueryProcessor* is the last task of the processing pipeline.
 - If the query submitted to the Node is of type 'GET', we instantiate a new object of type *VamdcDataFileDownloader* and associate it with the attribute *followingTask*.

⁶ src/org/rda/QueryStore/business/pipeline/tasks/NewVAMDCQueryProcessor.java

3.1.3 The ExistingVAMDCQueryProcessor class

The [ExistingVAMDCQueryProcessor](#)⁷ class is an extension of [PipelineTask](#). As its name indicates, the goal of this atomic task is to process already existing queries. Its constructor receives as arguments an object of type [NotificationDetail](#) containing all the information to process and the UUID of the already existing query.

- In the [runCurrentTask](#) method
 - We store into the table [QueryUserLink](#) the information concerning the query re-execution: the query-UUID, the Unix-timestamp of the moment when the query notification arrived to the Query Store, the set of original parameters submitted by the users (not in the canonical form), the user email (if provided), the client software used for submitting the query (if provided) and the query token generated by the node software.
- In the [instanciateFollowingTask](#) method
 - If the query submitted to the Node is of type 'HEAD', there is no following task to set. In this case the [ExistingVAMDCQueryProcessor](#) is the last task of the processing pipeline.
 - If the query submitted to the Node is of type 'GET', we instantiate a new object of type [VamdcDataFileDownloader](#) and associate it with the attribute [followingTask](#).

Remark: the query token is an identifier external to the Query Store (generated by the VAMDC nodes software), while the Query UUID is an internal identifier. Two different query tokens may be associated to the same UUID: indeed if one submit twice the same query to a given VAMDC data node, this will notify twice the Query Store with two different tokens. The Query Store processing pipeline will create only one UUID for the queries since it will recognise twice the same query.

3.1.4 The VamdcDataFileDownloader class

The [VamdcDataFileDownloader](#)⁸ class is an extension of [PipelineTask](#). As its name indicates, the goal of this atomic task is to download the data file generated on the VAMDC data node side by the query that is being processed. Its constructor takes two arguments: an object of type [NotificationDetail](#) containing all the information to process and the UUID of the query.

- In the [runCurrentTask](#) method

⁷ /src/org/rda/QueryStore/business/pipeline/tasks/ExistingVAMDCQueryProcessor.java

⁸ /src/org/rda/QueryStore/business/pipeline/tasks/VamdcDataFileDownloader.java

- We check (by testing in the table *Queries* of the internal database if the field *dataURL* corresponding to the processed query is defined or not) if the data-file has already been downloaded.
 - If the data-file has not been already downloaded, we download the file that the VAMDC data Node created for answering the query identified by the current UUID. This file is locally stored on the Query Store and its absolute path is stored in the *dataURL* field of the table *Queries*. This is the reason why this field may be used for testing if a given result file has already been downloaded by the Query Store.
- In the *instanciateFollowingTask* method we instantiate a new object of type *VamdcReferenceGetter* and associate it with the attribute *followingTask*.

3.1.5 The *VamdcReferenceGetter* class

The *VamdcReferenceGetter*⁹ class is an extension of *PipelineTask*. As its name indicates, the goal of this atomic task is to extract the bibliographic references contained into the data-file that has been downloaded during the execution of the previous *VamdcDataFileDownloader* task. Its constructor takes three arguments: an object of type *NotificationDetail* containing all the information to process, the UUID of the query and the absolute path of the data-file corresponding to the query.

- In the *runCurrentTask* method
 - We check (by testing in the table *Queries* of the internal database if the field *biblioGraphicReferences* corresponding to the processed query is defined or not) if the bibliographic references have already been extracted for the current query.
 - If there are no bibliographic references, these are extracted from the data-file (we recall that the absolute path is passed to this task through its constructor). The extraction is made by applying the XSL transformation defined by the file <https://github.com/VAMDC/QueryStore/blob/master/config/XsamsToBibtex.xsl>. Then the freshly extracted bibliographic information is stored into the *Queries* table (into the *biblioGraphicReferences* field).
- In the *instanciateFollowingTask* method
 - If the bibliographic reference were already available for the current query, there is no following task to set. In this case the *VamdcReferenceGetter* is the last task of the processing pipeline.

⁹ `src/org/rda/QueryStore/business/pipeline/tasks/VamdcReferenceGetter.java`

- If the bibliographic references were not present and are freshly extracted during the execution of the `runCurrentTask` method, we instantiate a new object of type `DataFileZipper` and associate it with the `followingTask` attribute.

3.1.6 The `VamdcReferenceGetter` class

The `DataFileZipper`¹⁰ class is an extension of `PipelineTask`. As its name indicates, the goal of this atomic task is to compress the data-file that has been downloaded during the execution of the previous `VamdcDataFileDownloader` task. Its constructor takes three arguments: an object of type `NotificationDetail` containing all the information to process, the UUID of the query and the absolute path of the data-file corresponding to the query.

- In the `runCurrentTask` method we zip the data-file and delete, after the compression phase, the original uncompressed data-file.
- In the `instanciateFollowingTask` method there is no following task to set. This task is the last of the processing pipeline.

3.2 Handling the errors of the processing pipeline: the `ErrorHandler` class

As we seen in paragraph 3.1, the `runTask` method of the class `PipelineTask` may raise a `PipelineNodeException`.

Each class implementing `PipelineTask`, when an error/exception is met (and just before raising the `PipelineTask` exception) instantiates an `ErrorHandler`¹¹ object for handling the error.

The constructor of `ErrorHandler` will call the method `logErrors`. This last will:

- Log the error detail into the table `Errors` of the internal database. The logged information include
 - The Pipeline phase (i.e. the task of the pipeline) when the error occurs.
 - The UUID of the query having the error
 - The query-token
 - The java error stack-trace
 - The timestamp when the query in error has been processed by the Query Store.
 - The set of parameters constituting the query.
- Get the number of queries associated with the UUID in error
 - If there is only one query-token associated with the UUID, then the entries for the query in errors are removed both from the tables `Query` and `QueryUserLink` of the internal database.

¹⁰ `src/org/rda/QueryStore/business/pipeline/tasks/DataFileZipper.java`

¹¹ `src/org/rda/QueryStore/business/pipeline/tasks/ErrorHandler.java`

- If there are multiple queries associated with the UUID, this means that only the processing of the most recent query generated errors (because of the mechanisms described in the previous item). Then the program simply deletes the link between the current query-token and the UUID in the *QueryUserLink* table.

4 The other web services of the Query Store.

The *NotificationListener* service (cf. paragraph 2) is only one of the web services making up the Query Store. These other web services are:

- The *AssociationService*,
- The *PortalAssociationService*
- The *FindQueriesByTime*
- The *InfoQuery*
- The *getUUIDByToken*

4.1 The AssociationService.

The VAMDC infrastructure is completely distributed with no central management system (cf. <https://github.com/VAMDC/QueryStore/blob/master/Readme.md>). Client software dispatches the user's queries to each data-node and each data node receiving a query notifies this to the Query Store. There is no direct communication between the users and the Query Store during the notification phase.

Moreover, for not slowing down the VAMDC infrastructure, the Query Store is an asynchronous service (a synchronous one would be a bottleneck for the infrastructure). The final UUID computed for a given query may be available only after the user receives the data results from the data-node.

The link between the user and the Query Store is done by the query token, which is generated by the data-node. Indeed this query token is sent both to the user (it is encapsulated into the data-result) and to the Query Store (cf. paragraph 2). At any further moment, the user may exchange the query Token he/she received from the data-Node with the final UUID assigned by the Query Store: by sending to the *AssociationService* the query token the user will receive the unique query identifier associated with his/her query.

The *AssociationService*¹² may be invoked (POST or GET) by providing to this service:

- The queryToken generated by the data-node,
- The user email
- The identifier of the usedClient.

¹² `src/org/rda/QueryStore/AssociationService.java`

The [AssociationService](#) then will execute the method [writeServerResponse](#) which, only if the token corresponds to a GET-type query, will:

- Get the user IP from the request
- Try to get the UUID that the Query Store computed for the Query with the given queryToken.
 - If the query-processing pipeline ended with no errors (cf. section 3), the correspondence between the query UUID and the queryToken is done by checking the table [QueryUserLink](#) (cf. section 6) of the internal database. If a correspondence exists, we will say that the query has been correctly associated and will update the database record with the information provided by the user (email, IP and usedClient).
 - If the query-processing pipeline ended with errors (cf. paragraph 3.2), then the correspondence between the query [UUID](#) and the [queryToken](#) is done by checking the table [Errors](#).
 - If a correspondence is found (as described in the two previous items), we return the UUID to the users, by formatting the information into a JSON text.
 - If no UUID/query-token correspondence is found or if one try to associate an HEAD type query, we return a 204 HTTP code.

4.2 The PortalAssociationService.

As we seen in the previous paragraph, the [AssociationService](#) may associate only token related to GET-type queries.

The HEAD-type queries in VAMDC have some peculiarities: the HEAD queries are used by the VAMDC portal (<http://portal.vamdc.eu>, <https://github.com/VAMDC/VAMDC-Portal>) for discovering if there is data associated to a given query and internal infrastructure monitoring (verify if nodes are correctly running).

Since the HEAD-type queries do not produces any data, it makes little sense to store them for a long time (cf. Section XXX). It has sense only if they are followed by the corresponding GET-query submitted for downloading the data. This twice-query mechanism is a key element in the functioning of the VAMDC portal (<https://github.com/VAMDC/VAMDC-Portal>). The service PortalAssociationService¹³ is designed for dealing with this portal specificity.

The [PortalAssociationService](#)¹⁴ may be invoked (POST or GET) by providing to this service:

¹³ `src/org/rda/QueryStore/PortalAssociationService.java`

¹⁴ `src/org/rda/QueryStore/AssociationService.java`

- The queryToken generated by the data-node,
- The user email
- The identifier of the usedClient (since the portal may have different versions).
- The userIP (since it is the portal is a web service contacting the Query Store, we cannot get the user IP directly from the request submitted by the portal. The portal has to explicitly transfer the user email as a parameter).

The [PortalAssociationService](#) then will execute the method [writeServerResponse](#), which if and only with the provided token corresponds to a HEAD-type query will:

- Try to get the UUID that the Query Store computed for the Query with the given queryToken.
 - As we explained at the beginning of this paragraph, only GET-type queries are stored for long time. As a consequence only GET-type queries may be associated. The program will search into the internal database (table [QueryUserLink](#)) which is temporal nearest GET-type query corresponding to the HEAD-type whose token is transmitted (method [tryAssociationAndGetUUID](#)). If the correspondence is found
 - the token of the GET-type query is associated with its final UUID in the sense that the correspondent database record is updated with the information about the user (email, user IP and usedClient).
 - The record corresponding to the HEAD query is deleted from the database.
 - The provided HEAD token may also corresponds to a GET-token whose processing-pipeline generated errors. In this case the program will search into the internal database (table [QueryUserLink](#)) which is temporal nearest GET-type query corresponding to the HEAD-type whose token is transmitted. Then it will check if this GET-token has an entry into the table [Errors](#). If it is the case, it gets the UUID in error.
 - In very rare cases, the processing of the HEAD-type query may generate some errors. In this case, the UUID associated with the HEAD query token is found by checking the entries of the table [Errors](#) of the internal database.
 - If a correspondence is found (as described in the three previous items) between the HEAD query Token and the final UUID associated with the query, we return the UUID to the users, by formatting the information into a JSON text.
 - If no UUID/query-token correspondence is found or if one try to associate an HEAD type query, we return a 204 HTTP code.

4.3 The InfoQuery service.

The goal of the [InfoQuery](#) service is to obtain detailed (and anonym) information about a given query.

The [InfoQuery](#)¹⁵ may be invoked by providing (POST or GET) to this service the UUID of the query.

This service

- Will check if the UUID provided exists into the internal database and if it corresponds to a query that has successfully (i.e. with no errors) processed or if it corresponds to a query with errors.
- If the UUID corresponds to query in error, the details of the error are extracted from the database:
 - The UUID
 - The query-token,
 - The error message,
 - The set of parameters constituting the query
 - The timestamp when the query was notified to the Query Store
 - The phase of the processing pipeline where the processing error occurs
- If the UUID corresponds to a query processed with no errors, the following information are extracted from the tables Queries and QueryUserLink of the internal database:
 - The UUID
 - The accededResource (the name of the data-node that answered the query)
 - The resourceVersion (the version of the data-node when it answered the query)
 - The outputFormatVersion (the version of the standard that the node used for formatting the output)
 - The dataURL (the link where data can be downloaded)
 - The queryRexecutionLink (the link for re-executing the query)
 - The biblioGraphicReferences (the set of the bibliographic references used for compiling the output data produced by the query).
 - A set of couple (parameterName, parameterValue) which represent the query parameters.
 - A set of couple (UNIX-timestamp, queryToken), which represent the history of the query submission (since the same query may be

¹⁵ `src/org/rda/QueryStore/InfoQuery.java`

submitted several times and has a different query-token at each execution).

- All the information collected for the query are returned to the user, formatted into a JSON text format.

4.4 The FindQueriesByTime service.

The goal of this service is to return summarized and anonym information about the queries that have been registered during a temporal range.

The [FindQueryByTime](#)¹⁶ may be invoked by providing to this service (POST or GET) the upper and lower bounds (expressed as UNIX-timestamps) of the temporal interval. These two parameters (whose technical names are *from* and *to*) are optional and the service may also be invoked with no argument.

The program will get, from the internal database, the information about all the queries submitted during the interval:

- [lower bound, upper bound] if both parameters are provided
-]-∞, upper bound] if lower bound is not provided
- [lower bound, +∞[if the upper bound is not provided
-]-∞, +∞[if no bound is provided.

The information gathered for each query is

- The UUID
- The accededResource (the name of the data-node that answered the query)
- The resourceVersion (the version of the data-node when it answered the query)
- The outputFormatVersion (the version of the standard that the node used for formatting the output)
- A set of couple (parameterName, parameterValue), which represent the query parameters.
- A set of couple (UNIX-timestamp, queryToken), which represent the history of the query submission (since the same query may be submitted several times and has a different query-token at each execution).

The program will then return a JSON list, each element of the list containing the information detailed into the previous item. The list is empty if no result is found.

¹⁶ `src/org/rda/QueryStore/FindQueriesByTime.java`

4.5 The `getUUIDByToken` service.

The goal of this service is simply informative: it returns the UUID that has already been associated (cf. paragraphs 4.1 and 4.2) with a given query-token.

The service `getUUIDByToken`¹⁷ may be invoked by providing to this service (POST or GET) a valid query-token.

The program will check into the internal database (table `QueryUserLink`) if an UUID has been associated with the received query-token. If an UUID is found, this is returned to user, formatted into a JSON text. If no correspondence is found, a 204 HTTP code is returned.

5 The Purger Daemon

As we explained in paragraph 4.2 HEAD-type queries are not followed by a corresponding GET-type query. This queries are generating no data and is has no sense to store them for a long term on the Query Store. The role of the `Purger`¹⁸ class is to clean the system by eliminating the HEAD-queries that, after a given long time, has no GET-query associated.

This class contain a `main` method. When this is executed, this removes from the system all the HEAD-type queries whose age is greater than two hours and smaller then thirty hours.

This method has been designed for being executed at least once per day (typically at midnight).

¹⁷ `src/org/rda/QueryStore/getUUIDByToken.java`

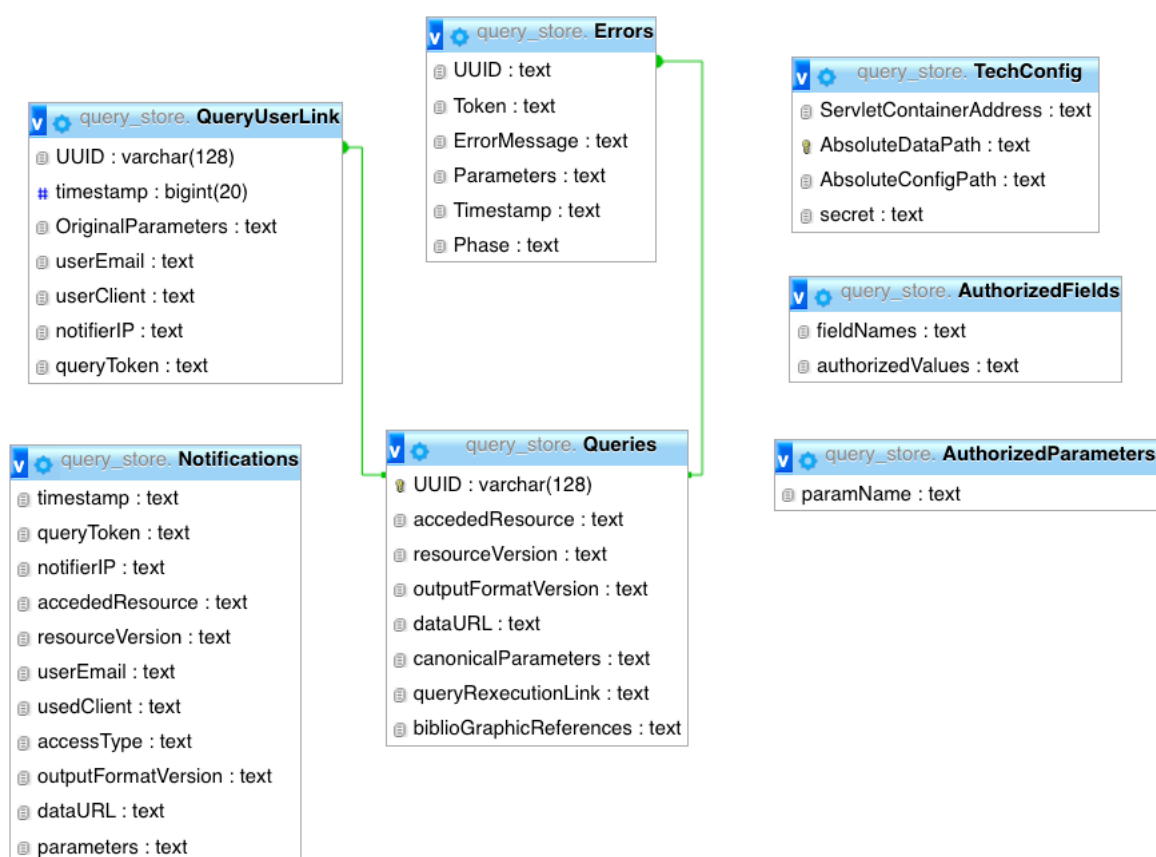
¹⁸ `src/org/rda/QueryStore/daemons/Purger.java`

6 The internal database schema

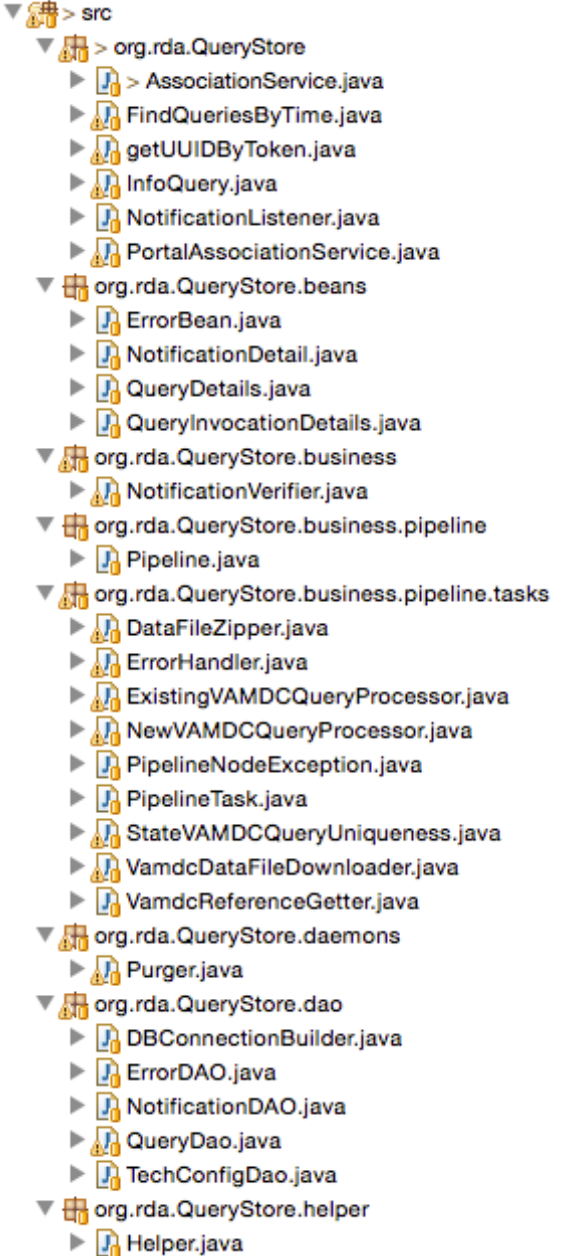
The script for creating from scratch the internal Query Store database may be downloaded at the following link:

<https://github.com/VAMDC/QueryStore/blob/master/DBScripts/QueryStore.sql>

The database structure is represented in the following image



7 The source code package structure

 <pre> src ├── org.rda.QueryStore │ ├── AssociationService.java │ ├── FindQueriesByTime.java │ ├── getUIDByToken.java │ ├── InfoQuery.java │ ├── NotificationListener.java │ └── PortalAssociationService.java ├── org.rda.QueryStore.beans │ ├── ErrorBean.java │ ├── NotificationDetail.java │ ├── QueryDetails.java │ └── QueryInvocationDetails.java ├── org.rda.QueryStore.business │ └── NotificationVerifier.java ├── org.rda.QueryStore.business.pipeline │ └── Pipeline.java ├── org.rda.QueryStore.business.pipeline.tasks │ ├── DataFileZipper.java │ ├── ErrorHandler.java │ ├── ExistingVAMDCQueryProcessor.java │ ├── NewVAMDCQueryProcessor.java │ ├── PipelineNodeException.java │ ├── PipelineTask.java │ ├── StateVAMDCQueryUniqueness.java │ ├── VamdcDataFileDownloader.java │ └── VamdcReferenceGetter.java ├── org.rda.QueryStore.daemons │ └── Purger.java ├── org.rda.QueryStore.dao │ ├── DBConnectionBuilder.java │ ├── ErrorDAO.java │ ├── NotificationDAO.java │ ├── QueryDao.java │ └── TechConfigDao.java └── org.rda.QueryStore.helper └── Helper.java </pre>	<ul style="list-style-type: none"> • The package <code>org.rda.QueryStore</code> contains all the web services composing the Query Store. • The package <code>org.rda.QueryStore.beans</code> contains all the Java Beans used for building the object model for the data handled by the Query Store. • The package <code>org.rda.QueryStore.business</code> contains all the business logic and the definition of the processing pipeline. • The package <code>org.rda.QueryStore.dao</code> contains the persistence layer and the methods for interacting with the internal database • The package <code>org.rda.QueryStore.daemons</code> contains the purger daemon. • The package <code>org.rda.QueryStore.helper</code> contains a set of useful methods used through all the code.
--	--