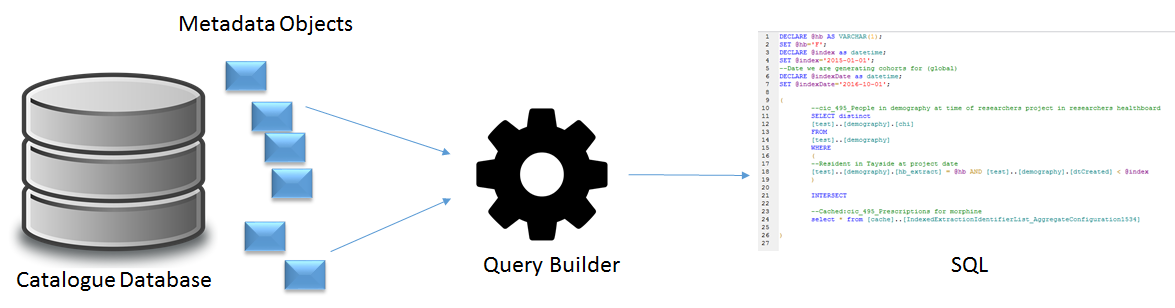
Query Building in RDMP

# Background

The RDMP software is designed to run on top of your existing database architecture. It does not require any schema changes of your data repository. It is intended to support a data analyst familiar with SQL who has been working on tasks such as ‘cohort identification’, ‘dataset linkage’ and ‘project specific data extraction’ (usually as an anonymous subset of your repository) for some time and is familiar with the concepts they entail.

Given the above system requirements the RDMP includes a versatile and powerful ‘query building’ engine. It is able to generate complex queries across multiple tables of arbitrary schema in a way that supports the data analyst and allows him to use his existing knowledge of the datasets he is used to manipulating.

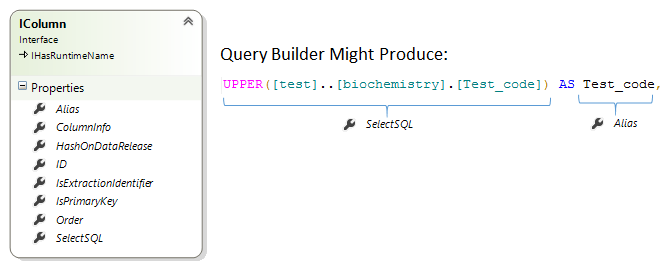
# What is a Query Builder and what does it do?



There are multiple Query Builders (see below), each responsible for generating queries under specific contexts (data extraction, patient cohort identification etc.) but they all behave in a similar manner. They take metadata objects out of the catalogue database and combine them to produce an SQL Select query. Metadata objects are mono-atomic SQL query elements for example ‘*There is a column called TestCode, it is extracted as UPPER([biochemistry]..[TestCode])*’ would be a valid IColumn metadata object. The object would have a reference to another metadata object that describes the table it comes from (biochemistry in this case). This object would be combined with other columns, filters etc to produce a SELECT statement for fetching data from the dataset biochemistry.

By storing mono-atomic metadata objects we allow the user (data analyst) to focus on only a single element at a time (‘do I want to extract this field? Should it be transformed? Is it a patient identifier?’) and provide documentation for future data analysts (‘what a given transform does, any problems with the content of the column etc.’). It also ensures that there is only one definition of each object (e.g. Test\_code is always extracted in the same way wherever it is used).

Below is the class definition of a metadata object designed to be used within the SELECT section of an SQL query and the SQL code that might get produced by a Query Builder using the object.



Note that depending on the specific Query Builder, it might decide to wrap the SelectSQL in a hashing algorithm (if HashOnDataRelease is true) or throw a QueryBuildingException because the column comes from a different table from the rest of the columns in the query (and it was unable to find JOIN logic).

# Existing Query Builders (and what they do)

## Query Builder

The basic QueryBuilder class is primarily used to fetch data from a single dataset (Catalogue). It includes support for ‘limitation SQL’ (distinct, top X etc), hashing, filters, multi table joins, SQL Parameters etc.

It is used by the RDMP any time it needs to fetch data from a dataset. For example, the Data Quality Engine uses a QueryBuilder to generate SQL so it can fetch data from your repository and validate it against the validation rules you have configured.

It also has methods for programmatically injecting constant strings into the output which is useful if you are writing your own query builder and want to adjust the query generated in a specific way without having an appropriate metadata object.

## Aggregate Builder

The AggregateBuilder is responsible for generating SELECT statements that include a GROUP BY or are patient identifier lists. It includes support for overriding the SELECT logic of columns in a dataset (e.g. Select Year portion of field only). It can also generate a dynamic SQL query that results in a data table with a continuous axis (e.g. ‘count records by month without skipping months with a zero count’). Finally, it can generate a dynamic PIVOT SQL query in which cell values become columns in the result set.

## Extraction Request / Query Builder Host

This factory class takes an Extraction Configuration which is a collection of datasets the user wants to extract for a research project by linking private patient identifiers with anonymous release identifiers. It creates a modified Query Builder in which private identifiers are substituted for release identifiers by doing an SQL JOIN against an identified cohort of patients.

It is unique in that it requires to mine Metadata objects from **both** the ‘Catalogue’ **and** the ‘Data Export Manager’ databases (other Query Builders only read from the Catalogue database). This is because the ‘Data Export Manager’ database acts as an override location for objects in the ‘Catalogue’ on a per project basis. For example, ‘For research project X we only have approval for 3 of the 10 columns in biochemistry and we can only supply records between 2001 and 2008’).

It includes support for substituting cohort identifiers (private for release), joining against custom data (e.g. project specific patient questionnaire results), dataset hashing etc.

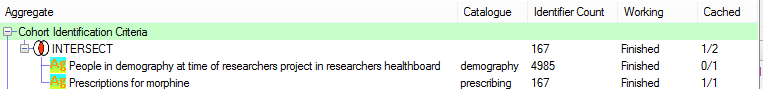
## Cohort Query Builder

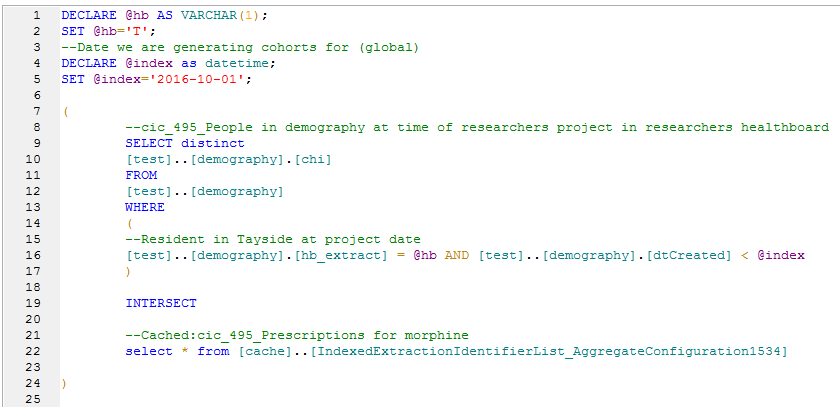
The most complicated query builder. This class generates mammoth SQL queries in which patient identifier lists are combined in a hierarchy of SQL set operations (INTERSECT, UNION, EXCEPT). The SQL for each identifier list is built by hosting Aggregate Builders (1 per dataset being queried) and selecting only the patient identifier column. These are then combined into a single mega query. Because querying many large datasets at once with complex filters on each subquery, the CohortQueryBuilder supports running and caching each individual subquery into an indexed table in a cache (which can be far more rapidly queried).

It is the only Query Builder that explicitly combines multiple datasets (Catalogue) into a composite query. Even still, it is only a single SQL command to execute the SELECT query and only a single result set is returned.

It includes support for parameter resolution (e.g. multiple datasets all share a parameter @healthboard), caching (see above), Table-Valued Functions, patient index tables etc.

See below for an example of a simple 2 dataset query generated by CohortQueryBuilder





# Metadata Objects

## Sql Parameters

### Class

|  |  |
| --- | --- |
| Class/Interface | Example Query Builder Output |
|  | --Date we are generating cohorts for (global)  DECLARE @index as datetime;  SET @index='2016-10-01'; |

### Background

Sql Parameters are important for a number of use cases. If a complex query has many filters that all reference the same value, then promoting the value to a parameter (rather than a constant string) means that it can be changed in one place only by the user. Another use case is when using Table-Valued Functions which require a populated parameter for each argument.

Sql Parameters are always output as either 2 or 3 lines of SQL code (depending on whether they have a Comment) and always appear at the top of the query.

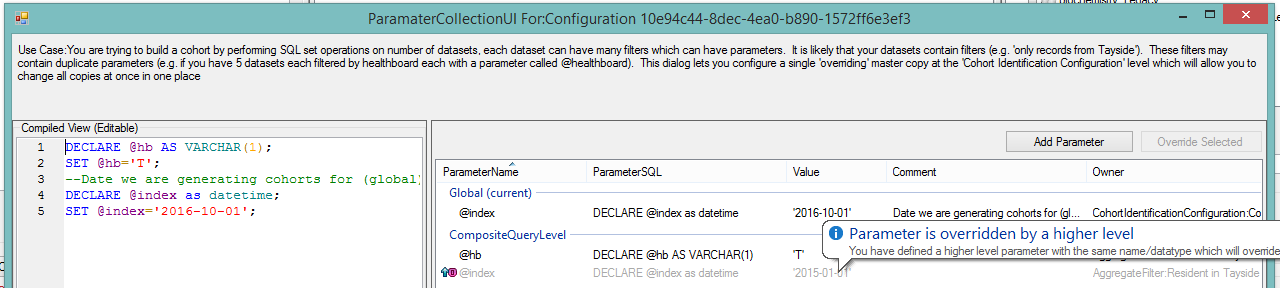
### Parameter Manager

Query Builders identify relevant ISqlParameter as they go along, whenever they find an ICollectSqlParameters class they take note of the parameters (if any) that are needed. Then after identifying all parameters they are resolved into a single list of unique parameters and output at the top of the query.

This allows you to (for example) include multiple filters in a query that all require a parameter called @hb without having to declare it every time.

Within Parameter Manager each ISqlParameter encountered during query generation has a recorded level at which it was found. This allows overriding/renaming. For example in a CohortIdentificationConfiguration you might have 2 datasets each with variables called @conditionCode but with different values. Both of these would be found by CohortQueryBuilder at ‘QueryLevel' and then ­be resolved by renaming to @conditionCode and @conditionCode \_2 (because the values were different). If the values were the same they would be merged into a single variable without renaming.

It is also possible to create Sql Parameters at Global level, these override values found at lower levels by replacing them in Parameter Manager. The figure below shows the user interface for modifying Global ISqlParameter of a CohortIdentificationConfiguration. It shows a single global variable called @index which can be used by all sub queries and then edited by the user in a single place.



## Columns

### Class

|  |  |
| --- | --- |
| Class/Interface | Example Query Builder Output |
|  | UPPER([test]..[biochemistry].[Test\_code]) as Test\_code |

### Background

Every SELECT statement in SQL is done on 1 or more columns. These are represented in the Query Builder by classes derived from IColumn. This includes anything that appears between the SELECT and the FROM sections of a query. This includes columns such as ‘count (\*) as MyCount’ (AggregateCountColumn) in the case of a GROUP BY statement.

Each IColumn only results in a single line of SQL output. The user is not permitted to add newlines into SelectSQL. Every Column has a ‘RuntimeName’ this will be the name of the column as it will appear in the result set (e.g. in a DbDataReader, DataTable etc.). You should always be able to successfully index a reader with the columns Runtime Name. Any column which contains a transform in its SelectSQL (e.g. ‘UPPER’) are required to have an Alias since SQL would otherwise call such a column ‘Column1’ in a result set.

### Derived Classes

Classes that implement IColumn can be divided into those that derive from ConcreteColumn and those that don’t. Those that derive from ConcreteColumn are stored in the ‘Catalogue Database’ (or Data Export Manager Database) and are created by the user. They reflect how the user wants a given dataset column to be extracted, the governance rules and contain links to descriptive metadata (CatalogueItem) which describe peculiarities of the column/data, the nature of any transforms applied etc.

The classes which do not derive from ConcreteColumn are those that are created programmatically by Query Builders and their hosts at runtime. For example, QueryBuilderHost uses instances of the class ReleaseIdentifierSubstitution to substitute private patient identifiers with project release identifiers when performing a data extraction. You should never attempt to programmatically modify the SQL returned by a Query Builder to add your own custom columns, instead you should inherit from IColumn and pass it into the Select list of the Query Builder.

### Underlying Tables

Every IColumn may be linked to a specific table (via ColumnInfo which has a property TableInfo). This lets the Query Builder know that it should include the table in the FROM section of the query. If the column collection includes multiple different tables, then the Query Builder will attempt to locate JoinInfo to connect up the tables. If it cannot find the required joins, then it will throw a QueryBuildingException (see method FindRequiredJoins).

## Joins

### Class

|  |  |
| --- | --- |
| Class/Interface | Example Query Builder Output |
|  | [test]..[prescribing] Left JOIN [test]..[DrugCosts] ON [test]..[DrugCosts].[drugname] = [test]..[prescribing].[name] collate Latin1\_General\_BIN |

### Background

Query Builders always require at least 1 target table for the FROM section of the query. These are instances of TableInfo and are usually automatically determined by the Query Builder while processing the IColumn collection. It is possible to force joins on additional tables that are not referenced by any IColumn for example you might have a table Tests and a table Results with a 1-M relationship, an aggregate count of records on TestDate (in Tests) could benefit from a force join on Results to produce the total number of results received each day.

When generating the FROM section of the query, the Query Builder must be able to combine all these tables using IJoin instances (found in the Catalogue Database).

Query building supports multi-column joins between tables (e.g. Table1.Forename = Table2.Forename AND Table1.Surname = Table2.Surname). This is done by an IJoin having any number of ISupplementalJoin objects which each reflect an additional column pairing required when assembling the ON section of the JOIN Sql.

### Join Order

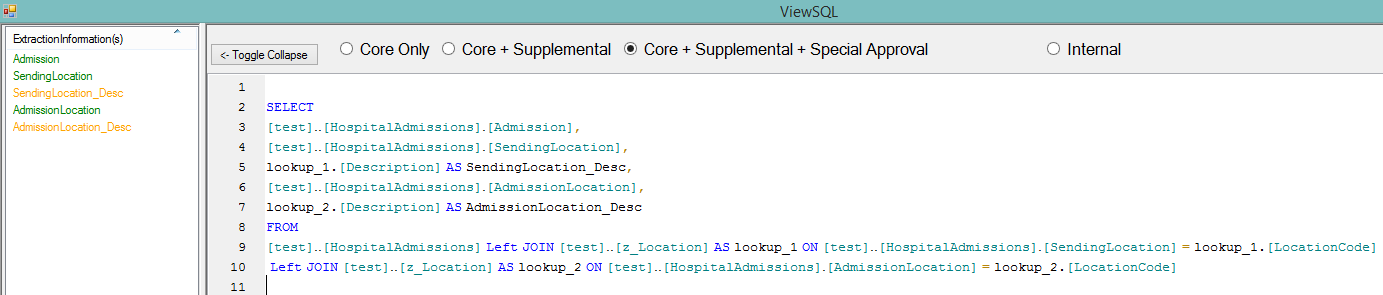
The Query Builder will always try to start joins from the table on the PrimaryKey side but sometimes a query will have so many tables involved that identifying the correct join order becomes problematic. In these cases, the Query Builder requires one of the TableInfo to be marked as IsPrimaryExtractionTable. It will then start the FROM statement with this table and include each IJoin down into other required tables. Lookup joins are always included last.

Often a table will resolve to many joins (e.g. if it is a lookup/reusable table), such joins are only included if they are between 2 tables that are required by the query.

### Lookups

Traditionally lookup tables are supplied to researchers independently of the data in order to save on space and simplify analysis. Some researchers however have expressed a desire for lookup values to be embedded in the main data file in line with the coded column. This requires joining each coded column (foreign key column) with the relevant lookup table column (primary key column) and including the relevant descriptive column (description column) in the SELECT list.

The Query Builder treats Lookup relationships almost exactly the same way as regular joins with one exception. When there are multiple columns coded with the same lookup table (e.g. if there are 2 columns Sending Location and Admission Location which both link to the lookup table z\_Location) then it becomes necessary to join against the Lookup table twice (see figure below for example).



To support multiple joins to the same Lookup table the Query Builder automatically aliases each lookup table (lookup\_1, lookup\_2 etc) and uses the column order to determine which foreign key column matches which description. This is all handled automatically by the Query Builder and will work as long as you do not reorder the columns (e.g. move all the lookup description columns to the bottom of the select order).

## Filter & Containers

### Classes

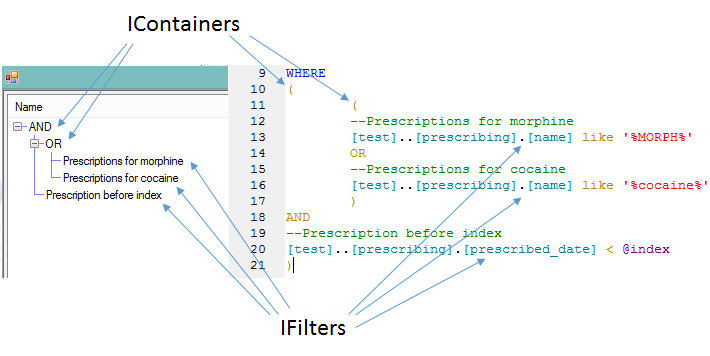
|  |  |
| --- | --- |
| Class/Interface | Example Query Builder Output |
|  | WHERE  (  --Prescriptions for morphine  [test]..[prescribing].[name] like '%MORPH%'  AND  --Prescription before index  [test]..[prescribing].[prescribed\_date] < @index  ) |

### Background

Most of the challenge in research data analysis is in identifying meaningful subsets of data. For example, ‘I want to identify records of patients with elevated blood pressure’. The implementation of this in SQL might involve a WHERE block to identify ‘blood pressure measurements’ and another for ‘value > X’ where X is a clinical threshold. Alternatively, X might vary based on patient BMI, age etc. Once an appropriate bit of SQL has been created and tested it is important that it be documented and preserved for reuse by other system users who might lack the clinical background required to understand the exact implementation. This is achieved­ through the creation of filters (IFilter).

### Container Resolution

In order to be used by a Query Builder, all IFilter in a given query must be held in an IContainer. Generating WHERE SQL begins by enumerating all the filters and sub containers of the root container. If there are 2+ objects (filters and/or sub containers) then they are each written out separated by the IContainer Operation (AND/OR). Sub containers are then processed recursively with an incrementally increasing tab depth. If there is no root IContainer configured, then there will be no WHERE SQL produced by the Query Builder.



In the above example we can see the root container AND has 1 sub container and 1 filter (‘prescriptions before index’). Therefore, in the SQL generated the sub container is written out then the AND Operation is written then the filter. The sub container however is an OR so when it is resolved it’s two objects (both filters) are separated by OR. The WHERE SQL for this configuration translates into ‘Any prescriptions for morphine or cocaine before the project start date @index’.