12 OpenClinica Insight

OpenClinica Insight makes it easy to ask questions of your clinical and operational data and visualize answers via interactive reports and dashboards. Once Insight is activated for your study, contact support@openclinica.com for a demo on how to get the best insights into your data.

Introduction

OpenClinica Insight - What could you do with intelligent reporting from OpenClinica, LLC on Vimeo. Insight is designed to be intuitive and self-documenting. Here are a few pointers to get you started...

Do I Have Insight?

OpenClinica Insight is available for OpenClinica 3 Enterprise and OpenClinica 4. It is an add-on module, so you may have it if your organization has purchased it. It lives at a separate URL and requires a separate login from your main OpenClinica environment. Contact your OpenClinica administrator to get access to your Insight domain or click here.

How do I Get Started?

Insight includes some out-of-the-box reports and visual tools for creating reports and dashboards that allow you to explore your data without a lot of hassle. Here are a few key concepts to help you out:

- **Reports** in Insight are called **Questions**. A Question can be built using the question builder, or written in SQL. The point and click question builder provides tools for filtering, aggregation, grouping, and more. You can also decide if you'd like to see the results as a table, chart, map, etc. When done, save the Question so others can use it and you can come back to it later.
- **Dashboards** are pages that display the results of several questions at once. Dashboards can be set up with filters so you can look at different slices of your data easily. You can drill down into a graph or table for more details on the results.
- **Pulses** and **Alerts** will send you and colleagues the results of a Question, on a schedule or when triggered by a change in the data.
- **Data Reference** makes it easy to explore your data by providing descriptions of tables and columns. Some of these descriptions are part of Insight itself, while others are generated from your study metadata. These descriptions can be customized and added to as needed by your Insight administrator.

The user interface runs on software called Metabase. Access the Metabase User Guide for details on how to create questions, set up alerts, and more.

Where do my Data Come from?

The foundation of OpenClinica Insight is a data warehouse that is generated from your OpenClinica transactional database. Built with PostgreSQL and Python, it provides:

- Each OpenClinica study environment as a schema

- Each CRF/Item Group as a table, with items as columns
- All non-removed data, plus audit logs for all data
- Transactional data re-modelled for optimal reporting via a constellation data model
- Snapshot updates of data (refresh rate is hourly and can be tuned higher / lower if desired)

What Else can Insight Help me do?

You can share and embed your questions and dashboards, or integrate Insight with your existing backend toolchains and reporting platforms. Sharing and embedding options are accessible through the user interface. Contact us if you'd like to access the data warehouse backend directly or use the Metabase API. Take a deeper dive into Insight by reviewing the slides to see how you can track enrollment, monitor adverse events, and identify missing forms. The slides also describe Insight's data model, architecture, and advanced features of the question builder. A one-hour long webinar recording is also available.

Tell Insight What You Want to See, and Take Control of How You See it on your Dashboard:

- Display responses to multiple questions at once
- Arrange your dashboard on a grid
- Resize and place responses where you want
- Filter one, some, or all questions at once

Insight Also Allows You to:

- Reference your data
- Build reports using an easy question-building interface
- Write SQL queries
- Alert sponsors, monitors, and/or sites of important milestones or data concerns
- Assign permissions
- Access via API
- Automate data retrieval from Insight
    - Results for existing questions
    - Results for ad-hoc SQL queries
    - Return as json, csv, or Excel (xlsx)
    - Automate object creation, setup, or any UI action

Embed Insight Objects in Another App

- Embed a dashboard or question in another app
- Either use public link, or use parameter controls
- Parameter controls:
    - **Disabled:** Nobody can edit, default value used
    - **Editable:** Users can adjust them
    - **Locked:** Set and signed by the embedding application

12.1 SQL Guide

Contents

1 Introduction

Many users will want or need to use SQL Questions to fulfill reporting requirements. Challenges for this task include:

1. Understanding SQL syntax in general
2. Understanding the Insight data model
3. Understanding the report requirements
4. Implementing the report correctly

This guide is aimed at helping with the first 2 challenges. The intended audience is data analysts / data scientists, or project managers / team members with an interest in preparing their own reports.

Postgres Documentation

The Insight back-end is a database system called Postgres. When either using the Metabase Question Builder, SQL Questions, or some other query tool with a direct connection, ultimately the request for data is processed by Postgres. Postgres has a vast feature set which is thoroughly documented online. Frequently useful entries include:

- Page on the SELECT command
- Chapter on Queries
- Chapter on Data Types
- Chapter on Functions and Operators

Note that the permissions for non-system Insight users only allow for reading data, so entries about modifying tables or updating data (insert/update/delete) are not relevant.

Other Resources

Among the hundreds of SQL guides and tutorials online, a beginner-friendly, interactive, and step-by-step tutorial is available for free at SQL Zoo.

2 Terminology

The following terminology and abbreviations are used in this guide.

- ES = Non-Repeating Event (S for Singular)
- GS = Non-Repeating Group
- ER = Repeating or Common Event
- GR = Repeating Group

3 SQL Joins in General

Join Types

There are 5 different join types available. For most queries however, only 2 of these will be necessary, with other 3 remaining useful for more advanced queries. The main 2 join types are:

- Inner Join:
  - a row for every match between Table A and B
  - excludes non-matching Table A or B
  - may result in multiple rows from Table A or B if there are multiple matches
- Left Join:
  - for every Table A row, show Table B data if there is a match
  - excludes non-matching Table B
  - may result in multiple rows from Table A or B if there are multiple matches

The other 3 types are Cross Joins (all combinations of rows from both tables), Right Joins (opposite of Left), and Full Joins (all rows, but like doing both Left and Right at once).

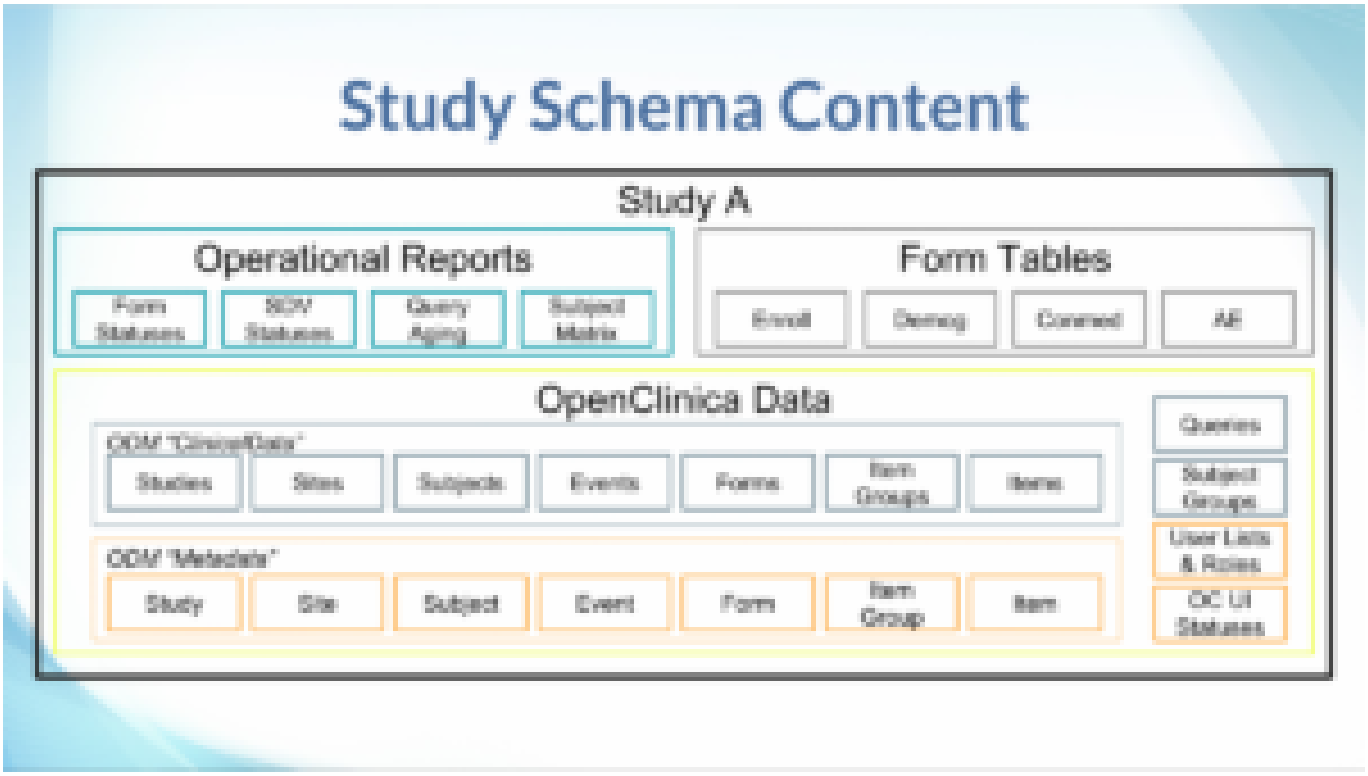
Join Expressions

When defining a join, an expression that provides a boolean (true/false/null) result must be provided. This expression determines which rows will be included in the join result. In Insight, the minimum join criteria will typically be the participant_table_id (OC4) or study_subject_table_id (OC3), which appears in most tables. For more advanced queries, other metadata columns may be included in the expression, such as identifiers for the Event, CRF, Item Group, repeat keys, etc. The below sections discuss the various identifiers available.

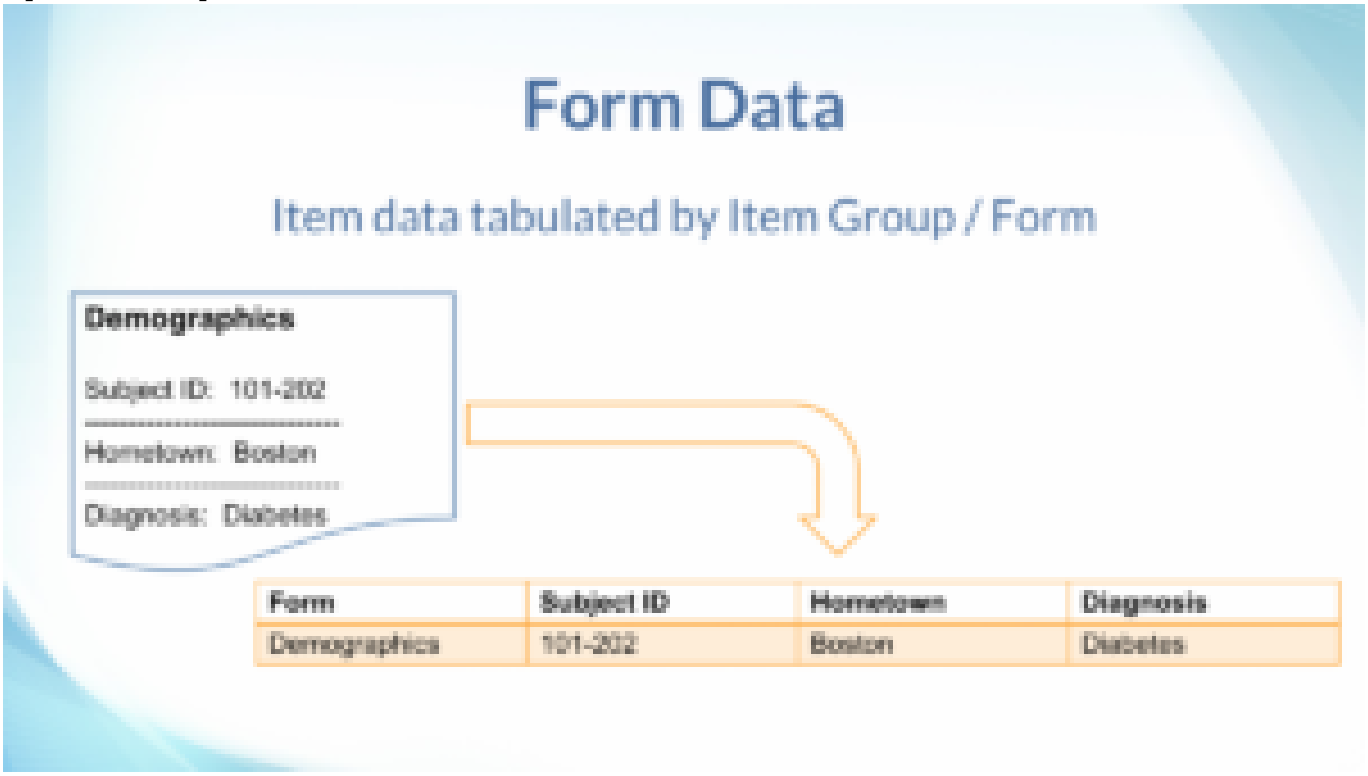
4 Insight Data Model

Overview

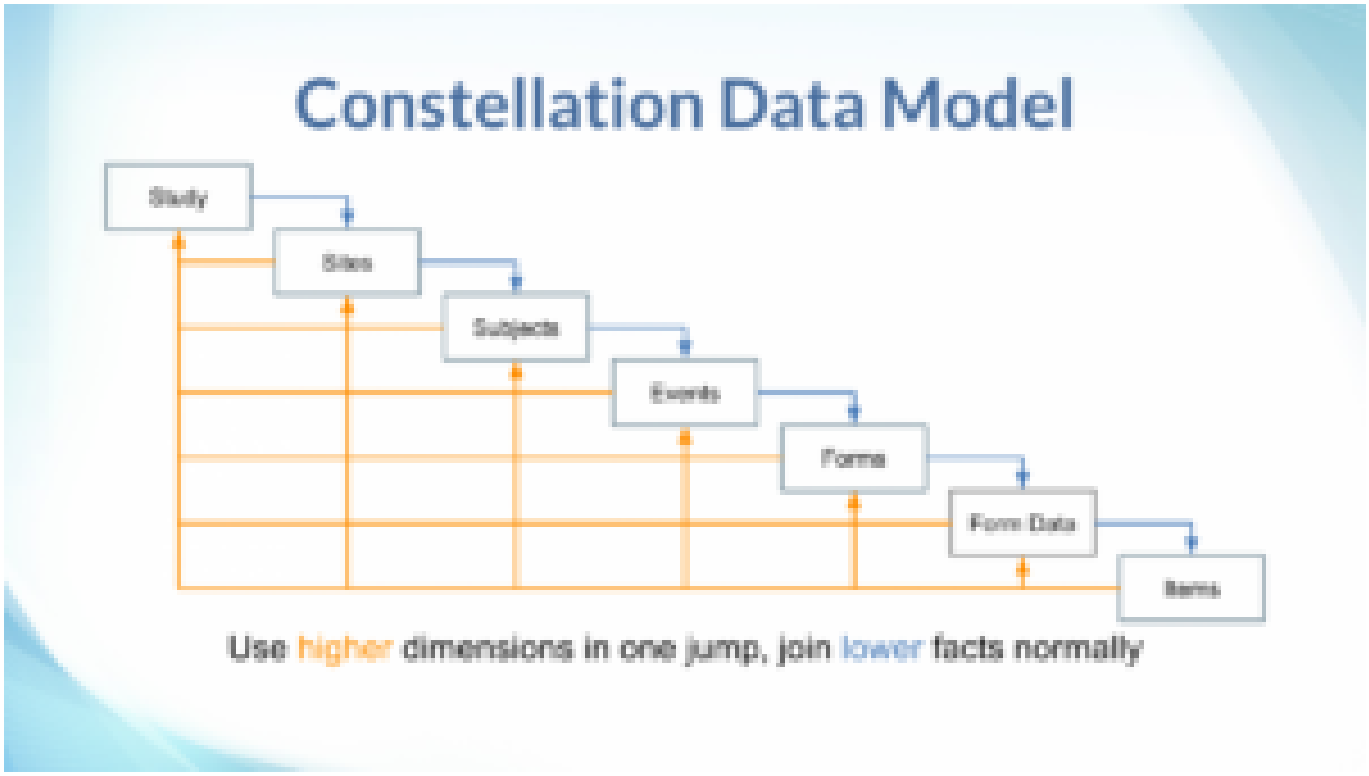
The following slides illustrate the basic structure of the Insight data model. A common request is for a full Entity Relationship Diagram, but it is of limited usefulness since there are so many relationships defined.



For those familiar with OpenClinica extracts, the Insight data model contains tables representing entities in the Metadata and ClinicalData elements, pivoted tables for form data, as well as some Operational reports.



The form data tables take the list-like (entity-attribute-value / EAV) item data values, and "pivot" them such that each item is a column with it's proper data type (text, numeric, date, etc.).



In reference to typical data warehouse terminology, the data model is most like a constellation, where there is a hierarchy of tables which can serve either as fact tables, or as dimensions of lower-level (higher detail) facts. Additionally, there many reference / dimension-only tables as well, such as lists of status choices.

Inspecting the Database

The Insight front-end has a built-in Data Reference, which catalogs the tables in each study schema (mart), and each table's columns and column data types. These entries are annotated to help describe the meaning or purpose of the table or column. For common tables, such as Participants, Events, etc., these annotations are provided by OpenClinica as part of Insight. For form data tables, these annotations include information from the form definitions, such as the left-item-text for an Item column. In both cases, the description states the underlying table or column name, for quick reference when writing SQL. To further inspect the structure of the database, users can query the Postgres System Catalog tables (in the `pg_catalog` schema), as documented at: https://www.postgresql.org/docs/current/catalogs.html. Alternatively, for users for whom a direct back-end connection has been set up, many database development tools provide inspection tools. For example, the following query lists all the explicitly-defined foreign key relationships in the database, optionally filtered for a particular schema (uncomment and modify last line). The `pg_catalog` however can't show implicit relationships, such as between two tables that both have an `event_oid` column.

```
SELECT
  nsp.nspname AS from_schema_name,
  cls.relname AS from_table_name,
  cna.attlist AS from_column_names,
  fns.nspname AS to_schema_name,
  fcl.relname AS to_table_name,
  cnf.attlist AS to_column_names,
  con.conname AS constraint_name,
  CASE con.confmatchtype
    WHEN 'f' THEN 'FULL'
```

```
    WHEN 'p' THEN 'PARTIAL'
    WHEN 's' THEN 'NONE'
  END AS match_type,
  CASE con.confupdtype
    WHEN 'c' THEN 'CASCADE'
    WHEN 'n' THEN 'SET NULL'
    WHEN 'd' THEN 'SET DEFAULT'
    WHEN 'r' THEN 'RESTRICT'
    WHEN 'a' THEN 'NO ACTION'
  END AS update_rule,
  CASE con.confdeltype
    WHEN 'c' THEN 'CASCADE'
    WHEN 'n' THEN 'SET NULL'
    WHEN 'd' THEN 'SET DEFAULT'
    WHEN 'r' THEN 'RESTRICT'
    WHEN 'a' THEN 'NO ACTION'
  END AS delete_rule
FROM pg_catalog.pg_constraint AS con
INNER JOIN pg_catalog.pg_namespace AS nsp
  ON nsp.oid = con.connamespace
INNER JOIN pg_catalog.pg_class AS cls
  ON cls.oid = con.conrelid
INNER JOIN LATERAL (
  SELECT
    array_agg(attname::text ORDER BY attname) AS attlist
  FROM pg_catalog.pg_attribute AS att
  WHERE att.attrelid = con.conrelid
    AND att.attnum = ANY(con.conkey)
) AS cna
  ON TRUE
INNER JOIN pg_catalog.pg_class AS fcl
  ON fcl.oid = con.confrelid
INNER JOIN pg_catalog.pg_namespace AS fns
  ON fns.oid = fcl.relnamespace
INNER JOIN LATERAL (
  SELECT
    array_agg(attname::text ORDER BY attname) AS attlist
  FROM pg_catalog.pg_attribute AS att
  WHERE att.attrelid = con.confrelid
    AND att.attnum = ANY(con.confkey)
) AS cnf
  ON TRUE
WHERE con.contype = 'f'
/*  AND nsp.nspname = 's_thejunod' */
```

Row Identifiers

There are 4 ways to identify rows in Insight, as described in detail below. The first two are auto-generated numeric IDs, best suited to join expressions. The second two are user-specified text IDs, best suited to filters but may also be used in join expressions. First, each table has a "table_id" column which is typically named after the table, for example, the "participant" table has a

"participant_table_id". This column is an auto-generated in Insight, and has no particular meaning besides providing a convenient way to join tables together. Specific values should not be referred to in queries (e.g. `WHERE participant_table_id = 3`), since the table_id may change. Table IDs can change for example, if a participant is removed from the study and then later restored, or if the mart needs to be re-built during a version upgrade. To refer to / filter on specific values, use the relevant user-specified name or label, or the OID (see below). Second, many tables include an "original_table_id" column, which is typically named after the table (or it's source table), for example "participant_original_table_id". This column is the auto-generated value from the source / transactional OpenClinica database, and has no particular meaning besides defining how to join tables together, and tracing Insight data back to the source. It can also be useful in tables with adjacency lists, for example the "Queries" / "discrepancy_note" table has a "parent_query_original_table_id" which for child-notes refers to it's thread-title record's "query_original_table_id". A shortcoming (compared to the Insight table_id) is that if an Insight mart includes data from more than one source, the "original_table_id" may not be unique, in which case the study_oid would need to be included in the join expression. Third, many tables have an OpenClinica "OID" column, which is a unique text identifier, typically derived from a name or label provided by a user. For example, the "Form Definitions" / "crf" table has a "crf_oid" column, which takes the first 12 alphanumeric / underscore characters from the "crf_name", converted to uppercase. If two CRFs would have the same crf_oid, the second one would have an underscore then 3 or 4 random numbers appended to the end, to make it unique within the study (OC4) or unique across all studies (OC3). Fourth, as mentioned above, many tables have a user-specified name or label column. Where the OID is a bit esoteric, the name or label can be used instead, for example it may be clearer to write `WHERE crf_name = 'Demographics'` rather than `WHERE crf_oid = 'F_DEMOGRAPHICS'`.

Form Data Tables

To simplify report preparation, a table is generated for each Item Group from all CRFs (Form Definitions) in the study. The Items in each Item Group is shown in table columns. Item Group data from any usage of the relevant CRF, such as in Scheduled, Common, or Repeating Events are saved into the same table. The name of the table is the Item Group's OID in lower case. Each row identifies where the data came from, including the site name, participant ID, event name, event repeat key, form name, form version, and item group repeat key. For example, if you use the same Visit CRF across a collection of Weekly Visits, the data in the Item Groups will already be collated. If you use slightly different forms for different visits, or want to combine totally different forms, SQL can be used to join or append data as needed. Items in the Item Group are shown in columns, with a data type corresponding to the CRF definition: dates are dates, integers are integers, text is text, while files and partial dates are represented as text (where applicable). The name of the column is the Item's OID in lower case. For single-select / radio items, the code value (not visible in the form) is shown in the column named with the Item's OID, while the code label (visible in the form) is shown in a column named almost the same except it uses the prefix "il_" instead of "i_". For multi-choice / checkbox items, the code values are shown as comma-separated values. While it is possible to look up the corresponding code labels in the form metadata, a future Insight version will aim to make working with multi-choice more convenient.

5 Joining Two Tables

The following are examples of joining two tables, and what happens in different scenarios depending on whether the data is from an Event which is repeating and/or the Item Group is repeating. A form may appear in many different events, both repeating and non-repeating, however whether the item group is repeating or non repeating does not vary. In these examples then, it is assumed that there

is a filter for the relevant Event e.g. `WHERE event_name = 'Baseline'`, such that there is not a mix of repeating and non-repeating event data.

When Table A is ES+GS

When the Table A data is in a non-repeating event (ES) and non-repeating group (GS), then:

- And Table B is ES+GS, then:
  - inner join result = A(ES:1,GS:1):B(ES:1,GS:1) or NULL:NULL
    - e.g. show data for all participants with both Enrollment data and Study Termination data.
  - left join result = A(ES:1,GS:1):B(ES:1,GS:1) or A(ES:1,GS:1):NULL
    - e.g. show data for participants with Baseline Enrollment data, but who might not have Study Termination data.
- And Table B is ES+GR, then:
  - inner join result = A(ES:1,GS:1):B(ES:1,GS:1+N) or NULL:NULL
    - e.g. show data for participants with both Baseline Demographic data and Baseline Medical History Log
  - left join result = A(ES:1,GS:1):B(ES:1,GS:1+N) or A(ES:1,GS:1):NULL
    - e.g. show data for participants with Baseline Randomisation, but who might not have any Baseline Concomitant Meds List data.
- And Table B is ER+GS, then:
  - inner join result = A(ES:1,GS:1):B(ER:1+N,GS:1) or NULL:NULL
    - e.g. show data for participants with both Enrollment data and Repeating Visits data
  - left join result = A(ES:1,GS:1):B(ER:1+N,GS:1) or A(ES:1,GS:1):NULL
    - e.g. show data for participants with Baseline Demographic data, but who may not have any Ad-Hoc Follow-Up Visit data.
- And Table B is ER+GR, then:
  - inner join result = A(ES:1,GS:1):B(ER:1+N,GS:1+N) or NULL:NULL
    - e.g. show data for participants with both Baseline Demographic data, and Common-Event Concomitant Meds Log data.
  - left join result = A(ES:1,GS:1):B(ER:1+N,GS:1+N) or A(ES:1,GS:1):NULL
    - e.g. show data for participants with Baseline Demographic data, but who may not have any Common-Event Adverse Event Sequelae List data.

When Table A is not ES+GS

When the Table A data is repeating at the event level (ER) or group level (GR) and Table B is ES+GS, then the outcomes are the reverse of the above examples for "Table A is ES+GS". When and Table B is not ES+GS (that is, both tables repeat at either level), then there are many possible approaches, including:

- Ignoring the presence of repeats, which produces a results like a cross-join, where for example event repeats A1,A2 and B1,B2 are shown as A1:B1, A1:B2, A2:B1, A2:B2.
- Joining on the repeat keys, which essentially matches rows based on the order of data entry, for example joining on event repeats A1,A2 and B1,B2 to show A1:B1, A2:B2.
- Joining on some other row content, such as a Item which has the visit date, for example using data A1(01-Feb), A2(06-Oct) and B1(06-Oct), B2(01-Feb) to show A1:B2(01-Feb), A2:B1(06-Oct).
- Joining on a "best match" type of algorithm, where rows with the most Item matches are selected, for example ranking a match on both visit_date and participant_status above a match

on only visit_date.
- Joining on some other criteria, for example correlating repeating follow-up form visit_date with repeating adverse-event form onset_date values that are after the previous follow-up (if any).
- Filtering out all but one repeat in Table A and B based on some criteria such that either (or both) tables are no longer repeating, for example matching only the most recent adverse event and follow-up visit for each participant.

6 Joining Multiple Tables

Example 1: Baseline Data Overview for Participants with Conmeds

- Title: Baseline Data Overview for Participants with Conmeds.
- Goal: show various baseline data together, but only where data exists across all 3 tables.
- Tables:
  - Table A (ES1:GS1) = Baseline Demographics, data of interest: education level.
  - Table B (ES1:GS1) = Baseline Enrollment, data of interest: enrollment date.
  - Table C (ES1:GS1+N) = Baseline Concomitant Medications Log, data of interest: medication names.
- Result pattern: A(ES1:GS1):B(ES1:GS1):C(ES1:GS1+N) or NULL:NULL:NULL

Example SQL:

```
SELECT
  table_a.participant_id AS "Participant ID",
  table_a.il_demog_educ_level AS "Education Level",
  table_b.i_enrol_enrol_date AS "Enrollment Date",
  table_c.item_group_repeat_key AS "ConMed Repeat Key",
  table_c.i_conmed_conmed_name AS "ConMed Name"
FROM table_a
INNER JOIN table_b
  ON table_a.particpant_table_id = table_b.participant_table_id
INNER JOIN table_c
  ON table_c.participant_table_id = table_a.participant_table_id
```

Example Output:

Participant ID	Education Level	Enrollment Date	ConMed Repeat Key	ConMed Name
101-001	High School	2019-01-23	1	Aspirin
101-003	University	2019-02-06	1	Ibuprofen
101-003	University	2019-02-06	2	Salbutamol

Note that in the above output:

- Participant "101-001" has 1 ConMed recorded, and gets 1 row.
- Participant "101-002" had no ConMed data, so gets no rows (if we wanted to see Table A/B anyway, use LEFT JOIN table_c).
- Participant "101-003" has 2 ConMed group records, so it gets 2 rows with the same Table A/B data.

12.2 Front-end API Guide

Front-end API Guide

Contents

1 Introduction

Some users will want to automate interaction with the Insight front-end, which uses a business intelligence application called Metabase. The most common interaction is to retrieve data, for example to export all rows of one or more tables, or the results of a given SQL statement. Other interactions may include tasks like creating similar metrics (aggregations) or segments (filters) across a range of tables.

This guide is aimed at describing how to interact with the Metabase API, highlighting the relevant endpoints, with reference to common programming tools.

The intended audience is statisitical or generalist programmers.

The Metabase API is not currently guaranteed by OpenClinica to be stable, fully validated, or have full backwards compatibility. In the future we plan to incorporate functional testing and change management of the API into our validation & release process but have not done so yet. We've found it to be reliable and useful thus far, but ensure you perform testing appropriate for your intended use case(s).

Metabase Documentation

The Metabase API lists the available endpoints, accepted HTTP verbs, parameters, and purpose of each endpoint: https://github.com/metabase/metabase/blob/master/docs/api-documentation.md

As a technical reference, the API docs do not cover the how-to's of using the API. In some cases it does not fully describe parameters, for example when the required value is a JSON object that may itself contain arrays or other objects.

This guide aims to fill some of these gaps, but in general the recommended strategy is to observe how the browser client app interacts with the API. In other words, since the Metabase UI that we interact with in the browser is essentially a ReactJS client app, we can learn how to perform an interaction by observing the client app uses the API to interact with the Metabase backend app.

Other Resources

On GitHub there are a few open source Metabase API client libraries, as listed below. These have not been vetted /endorsed for use with Insight, and have various levels of endpoint and test coverage, but may serve as either useful tools or examples:

- C# https://github.com/elevate/elevate.metabase.tools
- Clojure https://github.com/OpenCHS/metabase-migrator
- Golang https://github.com/kucuny/go-metabase
- JavaScript https://github.com/callawaywilson/metabasecloner
- PHP https://github.com/Germanaz0/php-metabase-sdk
- Python https://github.com/STUnitas/metabase-py
- Python https://github.com/mertsalik/metabasepy
- Ruby https://github.com/shimoju/metabase-ruby

Statistical programmers could either use a generalist language (+/- the above libraries) to perform the export then load the data into their stats software, or utilise the relevant capabilities in their stats software for interacting with HTTP APIs. For example:

- SAS has a PROC HTTP for JSON data: https://blogs.sas.com/content/sasdummy/2016/12/02/json-libname-engine-sas/
- Stata can call external programs using `shell` for example to invoke http tools like `curl`, `wget`, or `powershell` to actually perform the API calls, and then import the result CSV data with `insheet`, Excel data with `import excel`, or parse JSON data with ssc modules like `jsonio`.
- R has many packages on CRAN for working with HTTP APIs, such as `httr` and `RCurl`.

For integrating Metabase dashboards reports with other applications, there are also example embedding apps for JavaScript (Node), Python (Django), and Ruby (Rails) at: https://github.com/metabase/embedding-reference-apps

2 Authentication

The first interaction will always be authentication. The endpoint is POST `/api/session`, which accepts a username (email) and password. This email must correspond to an active Metabase user. The response will be a JSON dict with an `id` key, e.g. `{"id": "abcd"}` where the value is the session key.

This session key should then be included in the HTTP header of all subsequent requests, using the key `X-Metabase-Session`, e.g. `X-Metabase-Session: abcd`. An alternative approach is to save it as a session cookie value, with the the key `metabase.SESSION_ID`.

In preparing scripts, be sure to log in once then re-use the session key, ideally retaining it until it has expired. Metabase has non-configurable login throttling, such that even valid login requests can be rejected if they are sent too often.

To check if a session key is still valid, you can request any endpoint and check for HTTP 403, but a good generic one to use is GET `/api/user/current` and check for HTTP 200.

3 Retrieving Data

To retrieve data, the easiest endpoint to use is called `dataset`, which allows exporting ad-hoc query results as CSV, JSON, or XLSX. The other option is the `query` sub-path of `card`, which allows exporting results of saved questions.

The first task is to find the relevant Metabase ID for the database being queried, from GET `/api/database`, and filter the response JSON by `name` to find the integer `id`.

To use the `dataset` endpoint, send a POST to a sub-path of `/api/dataset`. The sub-path specifies the result format. For example to get results as CSV, send to `/api/dataset/csv`, for JSON `/api/dataset/json`, for XLSX `/api/dataset/xlsx`. The request parameters differ based on whether you intend to retrieve a table based on the Metabase ID, or if you intend to retrieve results from an SQL statement. In both cases, the request body should be of the type `x-www-form-urlencoded`, and the request body data should include a key `query` with a value as described below.

To query for table data by Metabase ID, first get the list of tables from GET `/api/tables`, then filter the response JSON by `schema` and table `name` to find the table `id`. Next, send a POST to `/api/dataset/<format>`. The request body value for `query` should be of the format `{"database":2,"type":"query","query": {"source_table":1234},"parameters":[]}`, where the `2` is the database ID found earlier, and `1234` is the table ID.

To query for SQL statement results, send a POST to `/api/dataset/<format>`. The request body value for `query` should be of the format `{"database":2,"type":"native","native":{"query":"SELECT * FROM s_mystudy.participant","template_tags":{}},"parameters":[]}`, where `2` is the database ID found earlier, and the inner `"query"` value is the SQL statement to execute; in this case it is `SELECT * FROM s_mystudy.participant`.

The keys for `"parameters"` and `"template_tags"` relate to the use of variables and filters, which are used in reports created in Metabase. However in this context the simplest approach would be to vary the provided SQL statement to include the appropriate parameters.

3.1 Example R Syntax

```
# Set as needed for a particular query.
metabase_url <- "customer-name.insight.openclinica.io"
session_id <- "a120c64b-bf33-41f3-ac9f-6121ac7f8c79"
database_id <- 2
sql <- "SELECT a as col1, a + 1 as col2 FROM generate_series(1, 10) as t(a)"

# Generic steps. Requires "httr" and "jsonlite" packages.
dataset_url <- paste0(metabase_url, "/api/dataset/json")
query <- list(
  native=list(query=sql, "template-tags"=c()),
  type="native",
  database=database_id,
  parameters=list()
)
response <- httr::POST(
  url=dataset_url,
  httr::add_headers(c("X-Metabase-Session"=session_id)),
  body=list(query=jsonlite::toJSON(query, auto_unbox=TRUE)),
  encode="form"
)

# Contains observations and columns as per input SQL query.
query_data <- jsonlite::fromJSON(httr::content(response, as="text"))
```

4 Getting All Data

As mentioned in the SQL Guide, the Postgres `pg_catalog` allows end users to retrieve information about the database structure. This can be useful in scripting data retrieval, for example to get all data from all tables at once, in combination with the above techniques for data retrieval.

The following catalog query returns a list of all non-system tables by their schema name. To get a list of schema names to filter the below query further, use `SELECT nspname FROM pg_catalog.pg_namespace`.

```
SELECT
  nsp.nspname AS schema_name,
  cls.relname AS table_name
FROM pg_catalog.pg_class AS cls
INNER JOIN pg_catalog.pg_namespace AS nsp
  ON nsp.oid = cls.relnamespace
WHERE cls.relkind = 'r'
  AND nsp.nspname NOT IN ('public', 'pg_catalog', 'information_schema')
  AND nsp.nspname NOT LIKE 'pg_temp%'
  AND nsp.nspname NOT LIKE 'pg_toast%'
```

Sending the above query to `/api/dataset/json` then provides the names of all the schemas and tables, which can in turn be used to generate the relevant SQL statements for a series of further calls. For example, we may receive an array of JSON objects like:

```
[
  {"schema_name":"s_study1","table_name":"crf"},
  {"schema_name":"s_study1", table_name":"event_crf"},
  {"schema_name":"s_study1","table_name":"item_data}
]
```

Which can then be processed into a series of request parameters:

```
{"database":2,"type":"native","native":{"query":"SELECT * FROM
s_study1.crf","template_tags":{}},"parameters":[]}
{"database":2,"type":"native","native":{"query":"SELECT * FROM
s_study1.event_crf","template_tags":{}},"parameters":[]}
{"database":2,"type":"native","native":{"query":"SELECT * FROM
s_study1.item_data","template_tags":{}},"parameters":[]}
```

Which in turn can be send to `/api/dataset/<format>` to fetch all table data. The results could then be saved for future reference, loaded into some other system, or analysed.

Bulk Export Consistency

Something to be aware of is how the Insight data refresh process and the above batch export job may affect data consistency among the exported results files.

Each API call would ultimately be executed in it's own "read committed" database transaction, while the Insight data refresh is in one "serializable" transaction so that users see a consistent snapshot. However, if some API calls execute before a refresh starts, and some after it finishes, the retrieved

copy of the latter tables may refer to data not present in the retrieved copy of the former tables.

For example, if the `event_crf` table is exported before a refresh commits and the `item_data` table after, there may be some newly added `item_data` rows referring to `event_crf` rows that weren't present at the time the `event_crf` table was exported.

A strategy to avoid this issue is to be aware of the schema's refresh frequency, either by querying the schema's `refresh_log` table, or asking Customer Support, and scheduling / running batch exports such that they complete before a refresh completes (or begin immediately after).

If this is not practicable, other options include a custom-developed solution, or using a direct back-end database connection. In these scenarios, it's then possible to open a single database transaction to retrieve all data, i.e. using "repeatable read" or "serializable" transaction isolation to ensure that all SELECT statements se the same atomic version of the data.

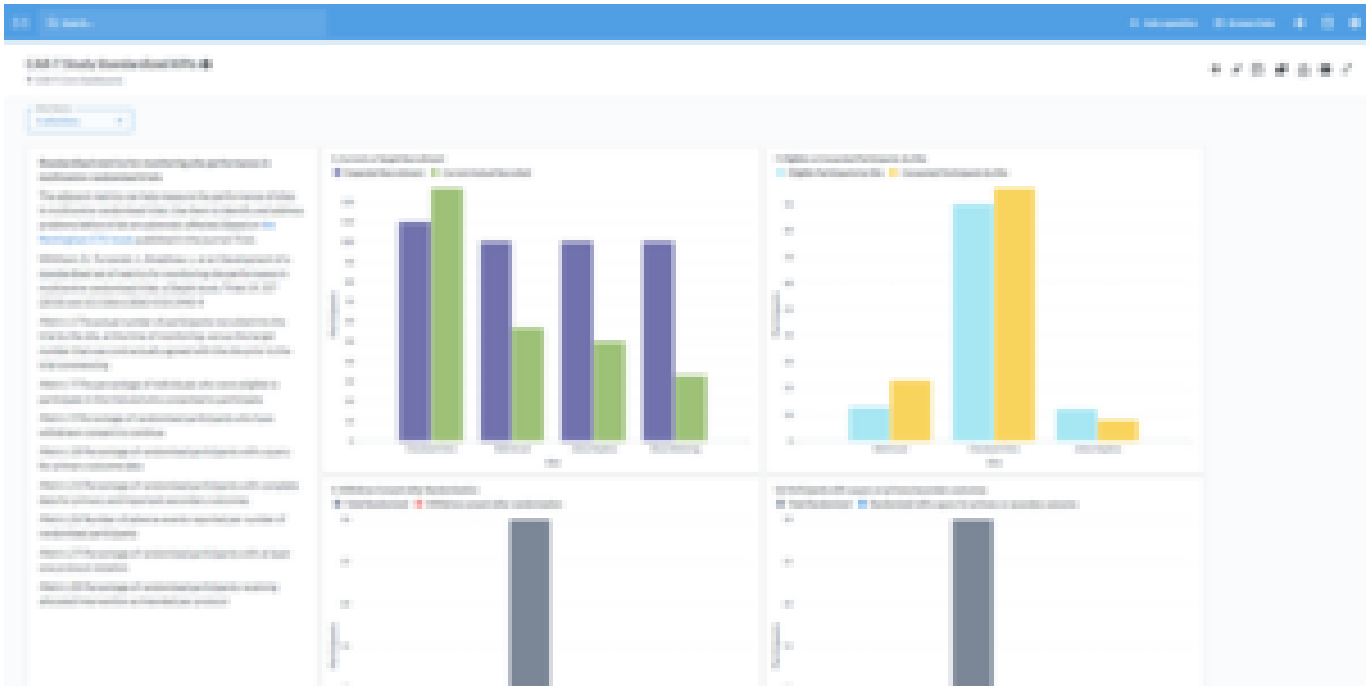
12.3 Insight Reporting Guide

Introduction

While Insight incorporates documentation for reference within the reporting interface, it can be hard to know where to start. This guide explains how to find and work with commonly used data to answer frequently asked questions, using practical examples along the way.

It is possible to get started and answer questions without knowledge of SQL, but usually for more complicated questions, it is required. This guide does not assume SQL knowledge. If you need help, the professional services team can assist you with preparing SQL questions.

Insight allows you to view data in different ways. Below is an example dashboard showing some of the ways you can view your data:



You can use the filters above the table to configure your screen to display different data. Below is an example of filters on the **Event CRF** table:

Questions About CRFs

Where are we with data entry? How is monitoring going?

For answering questions about CRFs, there are two main tables to consider.

The first table is the **Event CRF** table, which contains a row for each copy of a CRF that has been started anywhere in the study. It includes details such as, the site, Participant, Event, status, workflow SDV status, and links to many other tables to make it possible to include (join) extra details in a question (such as the study Start Date, Event Start Date, etc.)

The second table is the **Participant Matrix**, which contains a row for each CRF, which can be started by each Participant in the study at any time during the study. The concept is similar to the **Participant Matrix** in OpenClinica, in which users see started and possible Events in the study for each Participant. This table contains fewer details about Event CRFs than the first table but does have a link back to the Event CRF table so that it is possible to include (join) any extra details in a question.

CRF Status Examples

The **Event CRF** and **Participant Matrix** tables can be used to answer the following questions about CRF statuses:

How Many Started CRFs Are There?

1. Select the **Event CRF** table.

2. Select **Count of rows**.

How Many Completed CRFs Are There?

1. Select the **Event CRF** table.
2. Add a filter for **Event CRF Status Label** is **Completed**.
3. Select **Count of rows**.

Can the Above be Separated per Event and per Participant and a Total of All Completed CRFs per Event/Participant?

1. Select the **Event CRF** table.
2. Add a filter for **Event CRF Status Label** is **Completed**.
3. Select **Count of rows**.
4. Group by **Participant Id** and **Event Name**.

What is the Percentage of Completed CRFs Compared to Expected CRFs? What is the Definition of Expected CRFs?

A standard question called **CRF Completion By Event** shows percentages completed, not completed, and total (by site and Event). This question uses the **Participant Matrix** and considers an Event CRF to be complete if the **Event CRF Status Label** is **Completed**. In this question, the definition of expected for any Participant is any CRF in the study definition, plus CRFs in started Events (such as repeats).

Can the Definition of an Expected CRF be Customized?

For example, is it possible to incorporate an expected time window, or a rule that the End of Study Form is not expected to be completed until the End of Treatment Form has been completed?

Often the definition of an expected CRF is study-specific, intricate, and considers both study metadata and a Participant's Form data. Typically, such definitions are implemented in a SQL question, in which it is possible to use conditional logic expressions, multiple data processing steps, date calculations, combinations of data from multiple tables, and much more.

SDV Status Examples

The **Event CRF** and **Participant Matrix** tables can be used to answer the following questions about SDV statuses:

How Many SDV'd CRFs Are There?

1. Select the **Event CRF** table.
2. Add a filter for **Event CRF SDV Status Label** is **Verified**.
3. Select **Count of rows**.

Can the Above be Separated per Event and per Participant and a Total of All Completed CRF's per Event/Participant?

1. Select the **Event CRF** table.
2. Add a filter for **Event CRF SDV Status Label** is **Verified**.

3. Select **Count of rows**.
4. Group by **Participant Id** and **Event Name**.

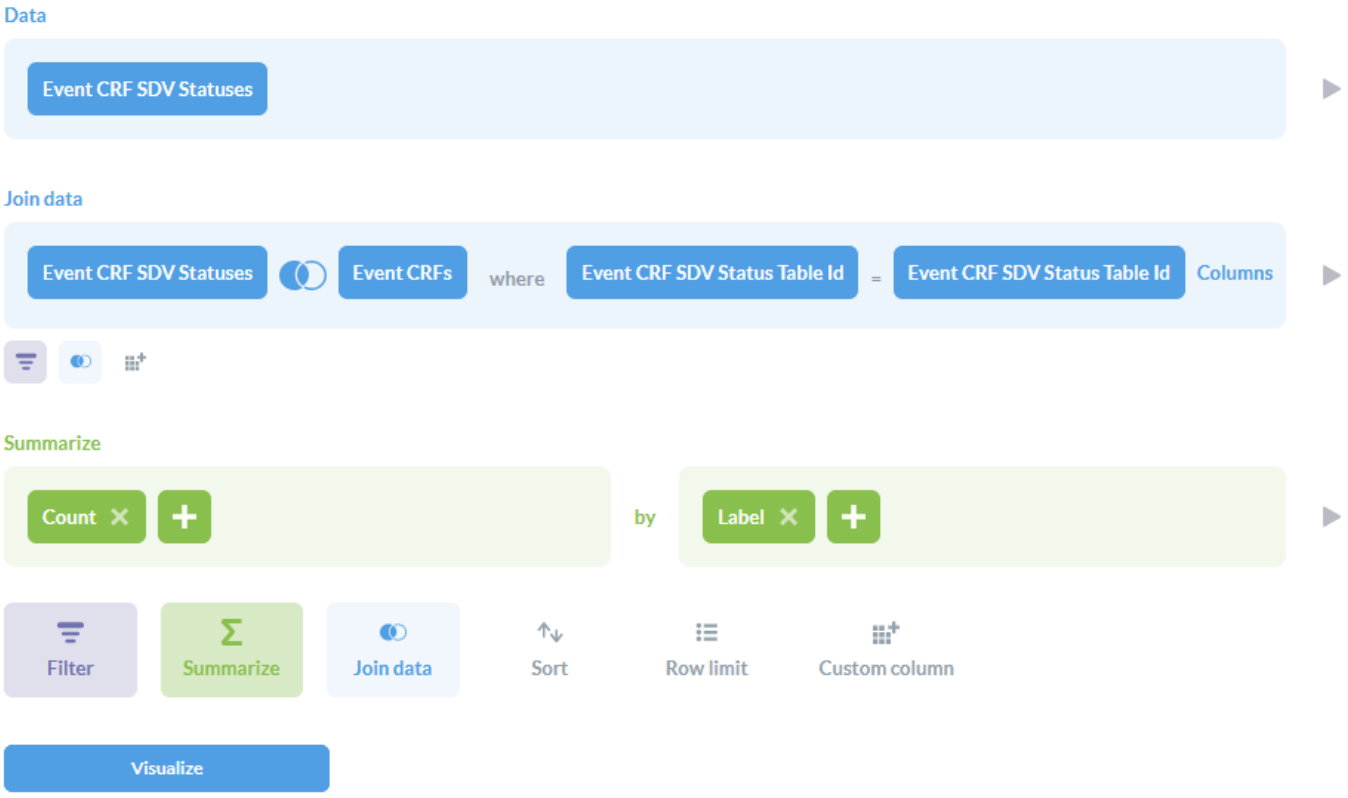
How can Forms that are not SDV'd Yet be Identified? What About Forms that Were Previously SDV'd but Invalidated Due to Data Changes?

As shown in the table **Event CRF SDV Statuses**, there is a status for each SDV state:

- **Verified:** Indicates that SDV is complete.
- **Not verified:** Indicates that a manual change to the Event CRF (such as an item value change) caused the previously complete SDV to be considered incomplete.
- **Changed since verified:** Indicates that an automatic change to the Event CRF (such as a triggered rule or workflow event) caused the previously complete SDV to be considered incomplete.
- **Never verified:** Indicates that SDV is not complete, and SDV has not been complete previously.

If you do not yet have data with these statuses and want to prepare a report counting each one:

1. Open the **Notebook Editor** by selecting **Ask a question** then **Custom question**.
2. For **Data**, select the **Event CRF SDV Status** table.
3. Click **Join data**:
    1. Select the **Event CRFs** table.
    2. The **Join type** (icon with 2 circles) should be **Left outer join**.
    3. For the **Join condition**, choose **Event CRF SDV Status Table Id** on both sides.
4. Click **Summarise**:
    1. For the metric, select **Count of rows**.
    2. For the **Group by**, select the **Event CRF SDV Status** column **Label**.
5. The result should look like the below screenshot. Click **Visualize**.



What is the Percentage of SDV'd CRFs Compared to Expected CRFs? What is the Definition of Expected CRFs?

A standard question called **SDV By Event** shows percentages SDV'd, not SDV'd, and total (by site and Event). This question uses the **Participant Matrix** and considers an Event CRF to be expected if either **100% Required** or **Partial Required** is selected for the CRF's SDV metadata in its Event definition.

Questions about Queries

Where are We with Data Cleaning? Have the Sites Been Responding to Queries?

For answering questions about queries, view the **Queries** table.

The **Queries** table contains details of queries, such as:

- Query type (query, reason for change, etc.)
- Resolution status (new, updated, closed, etc.)
- When the query was created
- Who the query was created by
- Who the query is currently assigned to

Since each query is attached to something (such as an item data value or study Event field), some identifying information about that value or field is shown in the **Queries** table and links back to the relevant table for full details.

Each row in this table corresponds with an action on the query. For example, there would be a row for when the **Monitor** created the query, another row for when the site responded, and another row for when the **Data Manager** closed the query.

The first query action row is considered the "parent" query, and the subsequent rows for that query are connected by the column named **Parent Query Original Table Id** to the parent's **Query Original Table Id**.

Note: *If you create a query in SQL and download a spreadsheet it should be limited to* ***300,000*** *rows or less to prevent the download from crashing.*

How Many Queries are There?

1. Select the **Queries** table.
2. Add a filter for **Parent Query Original Table Id** is ***(empty)***.
3. Select **Count of rows**.

Can the Above be Separated by Resolution Status, Participant, and Site?

1. Select the **Queries** table.
2. Add a filter for **Parent Query Original Table Id** is ***(empty)***.
3. Select **Count of rows**.
4. Group by ***Resolution Status***, ***Participant Id***, and ***Site Name***.

Can the above be separated by the issuing user, and assigned user?

1. Select the **Queries** table.
2. Add a filter for **Parent Query Original Table Id** is ***(empty)***.
3. Select **Count of rows**.
4. Group by **Created By** and **Assigned To**.

Can I report on these user's more generally, for example by the user role type (monitors vs. CRC vs. Data Manager)?

It is possible with a SQL question. The role types that a user has assigned to them are listed in the **User Account Role** table, which is linked to the **User Account** table by user account Id, and the **Queries** table can be matched with the **User Account** table by user name. Since a user can have multiple roles in a study, care should be taken to not double count queries assigned to a user with two roles.

Can I report on the time between status changes? For example, how long did it take for the site to respond to the initial query?

It is possible with a custom question or an SQL question. If we consider a set of query action rows associated with their parent query, ordered by their *Created timestamp*, a SQL feature called "Window functions" can be used in the query to fetch the previous ("lag" function) or next ("lead" function) row within that set of rows. By fetching the adjacent rows it's then possible to calculate the time between query actions, for example:

[previous row created timestamp] minus [current row created timestamp] equals [the time between updates]

Once the time differences (in hours, days, etc.) are calculated for each row, these results can be grouped by assignee, role type, site, etc.

Questions about Participants

How is Enrollment Going? Where are Most of the Participants in the Study?

For answering questions about Participants, start with the **Participants** table, and combine with Form data, if needed.

The **Participants** table contains details for each Participant, such as their Id, when they were entered in the system, and to which site they are assigned.

How Many Participants are There?

1. Select the **Participants** table.

2. Select **Count of rows**.

Can the Above be Separated per Site?

1. Select the **Participants** table.

2. Select **Count of rows**.

3. Group by **Site Name**.

To simplify question building and table relationships, Insight considers the top-level study, a site (i.e., answering this question does not require grouping by **study** then **site**).

Can I Report on Other Participant Statuses that are Collected on my CRFs, such as Screening Failure, End of Treatment, End of study, etc.?

It is possible with a SQL question. Usually the CRFs that collect Participant milestones are completed once (Non-Repeating Events), so the approach could be to:

1. Identify which **Item Group** table that each milestone data is in (the table name is the Item Group OID, and the column name is the Item OID).
2. Start with the **Participant** table so that there is a row for every Participant (no matter their progress) and join each **Item Group** table in the Custom Question Builder or using SQL with the **participant_table_id**
3. Apply the necessary filters/logic/calculations for the question, such as, including or excluding screening failures, calculating the difference between screening and baseline date, etc.

Questions about Form Data

How Many Adverse Events Have Been Recorded? How Often are Some Key Outcome Data Items left Unanswered?

For this we must get into the data captured on your CRFs.

The data entered into CRFs are shown in Insight in two main ways:

The first way is in the **Item Data** table, in which there is a row for every (non-removed) item data value in the entire study. This table contains the data value, when and who it was created by, when and who it was last updated by, as well as details of the related Event CRF, study Event, Participant, and associated metadata such as the data type (**text**, **integer**, etc.) and response set choices (e.g. for **single-select**, **multi-select**, etc.).

The item data types map into 4 main types, each with their own column: **text**, **date**, **numeric (float/real)**, or **integer**.

Since the values are actually stored as text, the **Data Text** column is always populated, and if there was a problem in converting the text value to its proper type, the **Cast Failure** column is **True**.

If the item's response type uses single-select choices, then the **Option Label** column shows the relevant label for the selected value. Items using multi-select choices are currently displayed as a text list of comma-separated values (however it is possible to process these values and look up the labels in a SQL question if needed).

The second way item data is shown is through the pivoted Form data tables, with display names that are formatted as, "Form Name - Item Group Name" (e.g. "Demographics - DEMOG") and back-end (SQL) names that have the Item Group OID (e.g. "IG_DEMOG"). These pivoted tables take the item data values for an Item Group, and group them by Participant, study Event Repeat, and Item Group Repeat, so data from all Event usages and versions of the Item Group's Form are shown together with the values for each item in their own column.

The item column display names are the item name (e.g. "EDUCATION"), and the back-end (SQL) name is the Item OID (e.g. "I_DEMOG_EDUCATION"). If the item response type uses single-select

choices, then an additional column is created to show the relevant label for the selected value, with a display name that uses the format "Item Name - Label" (e.g. "EDUCATION - Label") and the back-end (SQL) name that uses the Item OID, except with the letter "L" after the "I" prefix (e.g. "IL_DEMOG_EDUCATION").

How many doses of treatment has each Participant received?

Assuming there is one treatment Form (e.g. *Treatment - TRT*) that is completed multiple times (either in different Events, e.g. Week 1, Week 2, etc. or a Repeating Event), which has a "Was treatment given?" yes/no status item (e.g. *TRTSTATUS*):

1. Select the **Item Group** table, **Treatment - TRT**
2. Add a filter for **Yes** in the **TRTSTATUS - Label** column.
3. Select a **Count of rows**.
4. Group by **Participant Id**.

Approximately how many item data values were entered by each user?

1. Select the **Item Data** table.
2. Select **Count of rows**, and group by **Saved By**. (This is the user that either last updated the value, or if it was never changed, it is the user who initially created the value.)

Can the above be separated by week?

Follow the above steps, then add a grouping on Saved, and change its grouping time scale from the default (Day) to Week.

I have a Form where the items are in 2 or more groups. How do I combine them into one table?

It is possible with a SQL question, or using the Custom Question builder to join the two Item Group tables together. If all Form usages are in Non-Repeating Events, the **Item Group** tables can be combined with just the Participant Id, keeping in mind that if one or more of the Item Groups are Repeating, then the other table's rows are repeated for each repeat.

If some Form usages are in Repeating Events, the **Item Group** tables can be combined with the Event CRF Table Id, which uniquely identifies a copy of a CRF within a specific Participant's Event. If there is a relationship between the rows of a Repeating Group (e.g., row 3 of the Adverse Events Item Group corresponds to row 3 of the SAE Reporting Log Item Group) then the join criteria must include the **Item Group Repeat Key**.

Using the Custom Question Builder

You can use Insight's Simple Question Builder to answer many questions that use a single table, or filter on related items in other tables, but to join multiple tables together you will need to use Insights Custom Question Builder or write a SQL Question.

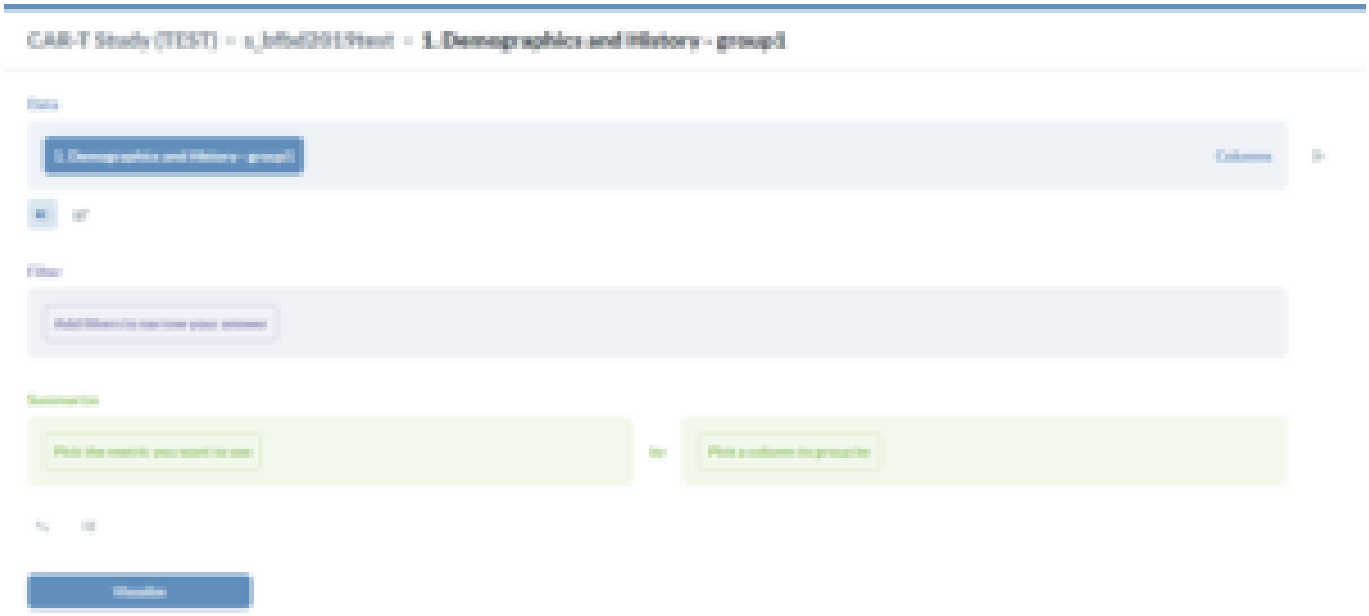
The Custom Question Builder can also be used to add custom columns or perform calculations across columns as part of a question.

To get started building a more complex Question using this advanced notebook editor, select a table. From there you can join additional tables using the **Join** button that looks like a Venn Diagram, or

define a custom column to add to the question, derived from existing columns in your current answer. As an example, here's how you would work through the case above and join together two Item Group tables to get a single result set that includes all the items from a Form that's been split into multiple Item Groups.

1. In the Custom Question Builder, select the first Item Group table in the CRF as your starting point.
2. Click **Join Data.**
3. Select the second Item Group table in the CRF, and select **Event CRF Table Id** from both tables when prompted.
4. Click **Visualize** to see both tables joined into one view.

You can also use the **Columns** menus on the side to hide or show columns, creating a cleaner view of your question if needed.



Filters and Summaries can also be applied as part of the Custom Question Builder, just like in the Simple Question Builder.

Functional approval by Lindsay Stevens. Signed on 2021-09-10 3:17AM

Approved for publication by Ben Baumann. Signed on 2021-09-10 10:19AM