Intro to SQL

Learn to write SQL to interact with your data via Query

SQL—short for "Structured Query Language" and pronounced either "sequel" or "ess-queue-el" depending on who you ask— is a programming language for interacting with data. **Query**, the Civis platform's in-browser SQL client, allows you to execute SQL queries on your data. By leveraging various commands, you can flexibly explore, shape, and aggregate your data.

For more in-depth, technical coverage of the topics introduced in this section, please refer to the official Redshift documentation, which provides explanation and examples for all supported commands and functions.

Clusters, Databases, Tables, and Schemas

Before we get started, it is important to understand the difference between these related terms. A **cluster** is a group of nodes, which refers to the physical hardware on which data is stored and processed. A cluster may have one or more **database**, which is made up of tables. A **table** consists of rows and columns, much like an Excel spreadsheet. A **schema** functions like a folder or an entire Excel workbook, holding related tables.

A note on syntax

SQL is *not* case-sensitive. Type functions and keywords in whatever case you are comfortable. Likewise, SQL is *not* line-sensitive. For the sake of readability in this document, keywords are preceded by line breaks, but these may be omitted without changing the nature of the queries.

Basic query structure

Below is the structure of a select query. Keep this order in mind. Queries will fail if clauses aren't defined in this order:

```
select list
from schema.table
where condition
group by list
order by list
limit n;
```

public.people

The examples in this section use a hypothetical table, '`public.people`'. Assume this table contains one row per person and contains the following data about each individual:

Column Name	Column Description	Data Type	Example Value
`id`	Unique person identifier	integer	`12345`
`first_name`	Person's first name	varchar	`FOZZIE`
`last_name`	Person's last name	varchar	`BEAR`
`email`	Person's email address	varchar	`wockawocka@muppets.com`
`state`	Person's state of residence	char(2)	`CA`
`signup_date`	Date the person first signed up for the organization's newsletter	date	`2008-11-04`
`is_active`	Contains 1 if the person is currently subscribed to the organization's newsletter, 0 if not	integer	`1`
`months_subscribed`	How many months the person has been subscribed	integer	`80`

You can view a full layout and sample rows for this or any other table by navigating to it in Query's database browser, located on the left side of the page, and clicking the arrow on the right-hand side of that pane to display the table details:

SELECT...FROM

Let's look at a basic SQL query:

```
select *
from public.people;
```

The above query will return a list of all rows and all columns in the public.people table. You can use the **select** command to read data from tables. In the **from** clause, name the table you want to select from. A semi-colon is placed at the end of each query.

The asterisk operator (*) means "all columns." You can instead select specific columns by listing them, separating each with a comma:

```
select id, is_active
from public.people;
```

LIMIT

We may want to look at a subset of records using **limit**. Here we select only five rows:

```
select id, is_active
from public.people
limit 5;
```

The limit clause should always appear *last*.

WHERE

What if we want to look only at records where the 'is_active' column is set to 1 (which, in this case, means true)? You can filter results using a **where** clause:

```
select id, is_active
from public.people
where is_active = 1
limit 5;
```

The query above will only return records marked as active. You can define conditions for the where clause using the following operators:

=	Equal (for booleans, 'is' or 'is not' may be used, ex. `is_active is true`)
!= *or* <>	Not equal
<, >	Less than, greater than
<=, >=	Less than or equal to, greater than or equal to
`between`	In an inclusive range, typically used for numeric or date ranges ex. `signup_date between '2014-01-01' and '2014-01-31'`
`in`	In a comma-separated list of values ex. `state in ('IL', 'NE', 'CA')`
`ilike`	Matching a pattern (case-insensitive); use one or more '%' to denote wildcards ex. `first_name ilike '%Bob%'` would match 'Bob', 'Bobby', 'Jim Bob', etc.
`like`	Matching a pattern (case-sensitive); use one or more '%' to denote wildcards ex. `first_name like '%BOB%'` would match 'BOBBY', but not 'Bob'

Note that strings and dates must be enclosed in single quotes, while booleans and numeric values can be unquoted. To filter on text that includes a single quote, replace each instance with *two* single quotes to escape the character, ex. 'Bob's' would become 'Bob''s'.

You can filter by multiple conditions by adding 'and' before successive conditions:

```
select id, is_active
from public.people
where is_active = 1
and state = 'IL'
limit 5;
```

To filter by one condition *or* another, use 'or':

```
select id, is_active
from public.people
where is_active = 1
or state = 'IL'
limit 5;
```

For complex conditions, use parentheses to clarify order of operations. To find records corresponding to active subscribers from Illinois *or* all individuals who signed up after November 1, 2014, for example:

```
select id, is_active
from public.people
where (is_active = 1 and state = 'IL')
or signup_date > '2014-11-01'
limit 5;
```

ORDER BY

Results will be returned in a non-deterministic order (i.e. in no particular order) unless you include an **order by** clause. You may order by any column in the table you select from, even if the column is not selected. By default, rows are sorted in ascending order. You can override this by adding '`desc`' after the order by clause.

```
select id, is_active, signup_date
from public.people
order by signup_date desc
limit 5;
```

This query will return the id, active status, and signup date of the five most recent subscribers, with the most recent first.

GROUP BY / Aggregates

Selecting individual records is, of course, of limited utility. Aggregates unleash the analytic power of SQL. Aggregate functions include **avg**, **count**, **min**, **max**, and **sum**. A full list of supported aggregate functions is available in the Redshift documentation.

One of the most common use cases for an aggregate function is a simple row count of a table:

```
select count(*)
from public.people;
```

To get a count of non-null values in a given column, specify the column. This query will tell you for how many people the '`state`' field is populated:

```
select count(state)
from public.people;
```

To count only unique values, use **distinct**:

```
select count(distinct state)
from public.people;
```

Since there are 51 possible values for this column (50 states plus the District of Columbia), the above query will return the number 51.

You can get aggregates grouped by another column by adding a **group by** clause. This query will show how many subscribers are from each state:

```
select state, count(*)
from public.people
group by state
order by state;
```

You can also group by multiple columns. When using an aggregate function, you must group by all other columns listed in your select. As a shortcut, in both group by and order by clauses you may refer to columns by their numeric position in the select list:

```
select is_active, state, count(*)
from public.people
group by 1,2
order by 1,2;
```

When using aggregates like **avg**, specify what column you want to aggregate by enclosing it in parentheses after the function name. Since aggregate columns will by default be titled with the function name, you may wish to give them aliases using **as**.

This query will return the average subscriber tenure in months by state in descending order by tenure:

```
select state, avg(months_subscribed) as avg_tenure
from public.people
group by 1
order by 2 desc;
```

You can use multiple aggregate functions in the same query, as long as you group by the same column(s):

```
select state, count(*), min(months_subscribed) as min_tenure,
    max(months_subscribed) as max_tenure, avg(months_subscribed) as avg_tenure
from public.people
group by 1
order by 1;
```

CREATE TABLE AS

You can save query results as a new table by preceding your select command with **create table schema.table as**, specifying a new and unique table name:

```
create table scratch.active_ids as
    select id
    from public.people
    where is_active = 1;
```

GRANT

By default, only you will be able to access tables you create. To allow other users to view your table in Civis and select from it in Query, use **grant select**. You can specify usernames or user groups (by prefacing the group name with 'group'):

```
grant select on scratch.active_ids to username, group civis;
```

DROP TABLE

You can get rid of any table you create—but not tables created by other users—with **drop**. Be careful, as this action is irreversible!

```
drop table scratch.active_ids;
```

CONCLUSION

Once you have grasped these concepts, you may want to explore more advanced SQL techniques. Our "Advanced SQL" help document covers topics including: how to get data from multiple tables (**joins**), date functions, case statements (which allow you to apply if-then-else logic), and subqueries.

ADDITIONAL READING

For more information on SQL best practices, read this helpful blog post about 10 Rules for a Better SQL Schema.