The Google Query Language syntax is designed to be similar to SQL syntax. However, it is a subset of SQL, with a few features of its own that you'll need to learn. If you're familiar with SQL, it shouldn't be too hard to learn.

**Table used in all examples:**

Throughout this section, all examples of queries refer to the following table. The column headers are the column identifiers.

| name<br>string | dept<br>string | lunchTime<br>timeofday | salary | hireDate<br>date | age<br>number | isSenior<br>boolean | seniorityStartTim<br>datetime |
|---|---|---|---|---|---|---|---|
| John | Eng | 12:00:00 | 10000 | 2005-03-19 | 35 | true | 2007-12-02 15:56:00 |
| Dave | Eng | 12:00:00 | 5000 | 2006-04-19 | 27 | false | null |
| Sally | Eng | 13:00:00 | 6000 | 2005-10-10 | 30 | false | null |
| Ben | Sales | 12:00:00 | 4000 | 2002-10-10 | 32 | true | 2005-03-09 12:30:00 |
| Dana | Sales | 12:00:00 | 3500 | 2004-09-08 | 25 | false | null |
| Mike | Marketing | 13:00:00 | 8000 | 2005-01-10 | 24 | true | 2007-12-30 14:40:00 |

# Language Clauses

The syntax of the query language is composed of the following clauses. Each clause starts with one or two keywords. All clauses are optional. Clauses are separated by spaces. The order of the clauses must be as follows:

| Clause | Usage |
| --- | --- |
| `select` (#Select) | Selects which columns to return, and in what order. If omitted, all of the table's columns are returned, in their default order. |
| `where` (#Where) | Returns only rows that match a condition. If omitted, all rows are returned. |
| `group by` (#Group_By) | Aggregates values across rows. |
| `pivot` (#Pivot) | Transforms distinct values in columns into new columns. |
| `order by` (#Order_By) | Sorts rows by values in columns. |
| `limit` (#Limit) | Limits the number of returned rows. |
| `offset` (#Offset) | Skips a given number of first rows. |
| `label` (#Label) | Sets column labels. |
| `format` (#Format) | Formats the values in certain columns using given formatting patterns. |
| `options` (#Options) | Sets additional options. |

## Select

The **`select`** clause is used to specify the columns to return and their order. If this clause is not specified, or if **`select *`** is used, all of the columns of the data source table are returned, in their original order. Columns are referenced by the identifiers (not by labels). For example, in a Google Spreadsheet, column identifiers are the one or two character column letter (A, B, C, ...).

Items in a **`select`** clause can be column identifiers, or the output of aggregation functions (#aggregation_functions), scalar functions (#scalar_functions), or operators (#operators).

**Examples:**

```
select *
```

```
select dept, salary
```

```
select max(salary)
```

In the following example, back-quotes are used to reference column ids that contain spaces (email address) or that are reserved words (date):

```
select `email address`, name, `date`
```

Running the following query on the example table (#Table_used_in_Examples):

```
select lunchTime, name
```

Returns the following response:

| lunchTime | name |
|---|---|
| 12:00:00 | John |
| 12:00:00 | Dave |
| 13:00:00 | Sally |
| 12:00:00 | Ben |
| 12:00:00 | Dana |
| 13:00:00 | Mike |

## Where

The `where` clause is used to return only rows that match a specified condition.

The simple comparison operators are `<=`, `<`, `>`, `>=`, `=`, `!=`, `<>`. Both comparison operators `!=` `<>` mean not-equal. Strings are compared by lexicographic value. *Note that equality is indicated by =, not == as in most computer languages.* Comparing to `null` is done using `is null` or `is not null`.

You can join multiple conditions using the logical operators `and`, `or`, and `not`. Parentheses can be used to define explicit precedence.

The where clause also supports some more complex string comparison operators. These operators take two strings as arguments; any non-string arguments (for example, dates or numbers) will be converted to strings before comparison. String matching is case sensitive (you can use `upper()` or `lower()` scalar functions (#scalar_functions) to work around that).

- `contains` - A substring match. *whole* `contains` *part* is true if *part* is anywhere within *whole*. **Example:** `where name contains 'John'` matches 'John', 'John Adams', 'Long John Silver' but not 'john adams'.
- `starts with` - A prefix match. *value* `starts with` *prefix* is true if *prefix* is at the beginning of *value*. **Examples:** `where dept starts with 'engineering'` matches 'engineering' and 'engineering managers'. `where dept starts with 'e'` matches 'engineering', 'eng', and 'e'.
- `ends with` - A suffix match. *value* `ends with` *suffix* is true if *suffix* is at the end of *value*. **Example:** `where role ends with 'y'` matches 'cowboy', 'boy', and 'y'.

- `matches` - A (preg) regular expression match. *haystack* `matches` *needle* is true if the regular expression in *needle* matches *haystack*. **Examples:** `where country matches '.*ia'` matches India and Nigeria, but not Indiana. Note that this is not a global search, so `where country matches 'an'` will not match 'Canada'.
- `like` - A text search that supports two wildcards: %, which matches zero or more characters of any kind, and _ (underscore), which matches any one character. This is similar to the SQL LIKE operator. **Example:** `where name like fre%` matches 'fre', 'fred', and 'freddy'.

**Examples:**

```
where salary >= 600
where dept != 'Eng' and date '2005-01-21' < hireDate
where (dept<>'Eng' and isSenior=true) or (dept='Sales') or se
```

Running the following query on the example table (#Table_used_in_Examples):

```
select name where salary > 700
```

Returns the following response:

| name |
| --- |

| |
|---|
| John |
| Mike |

## Group By

The **`group by`** clause is used to aggregate values across rows. A single row is created for each distinct combination of values in the group-by clause. The data is automatically sorted by the grouping columns, unless otherwise specified by an **`order by`** clause.

**Note:** If you use a **`group by`** clause, then *every column* listed in the **`select`** clause must either be listed in the **`group by`** clause, or be wrapped by an aggregation function (#aggregation_functions).

**Examples:**

```
select dept, max(salary) group by dept
```

Running the following query on the example table (#Table_used_in_Examples):

```
select lunchTime, avg(salary), count(age) group by isSenior,l
```

Returns the following response:

| lunchTime | avg-salary | count-age |
|---|---|---|
| 12:00:00 | 425 | 2 |
| 13:00:00 | 600 | 1 |
| 12:00:00 | 700 | 2 |
| 13:00:00 | 800 | 1 |

## Pivot

The `pivot` clause is used to transform distinct values in columns into new columns. For example, a pivot by a column 'year' would produce a table with a column for each distinct year that appears in the original table. This could be useful if, for instance, a line chart visualization draws each column as a separate line. If you want to draw a separate line for each year, and 'year' is one of the columns of the original table, then a good option would be to use a pivot operation to do the necessary data transformation.

**Note:** If you use a `pivot` clause, then *every column* listed in the `select` clause must either be listed in the `group by` clause, or be wrapped by an aggregation function (#aggregation_functions)

Since multiple rows may contain the same values for the pivot columns, pivot implies aggregation. Note that when using `pivot` without using `group by`, the result table will contain exactly one row. For instance, running the following query on the example table (#Table_used_in_Examples):

```
select sum(salary) pivot dept
```

Returns the following response:

| Eng sum-salary | Marketing sum-salary | Sales sum-salary |
|---|---|---|
| 2100 | 800 | 750 |

This is because 2100 is the sum of the salaries for the Eng department, 800 for the Marketing department, etc.

Using `pivot` together with `group by` can be even more useful, since it creates a table where each cell contains the result of the aggregation for the relevant row and the relevant column. For example, running the following query on the example table (#Table_used_in_Examples):

```
select dept, sum(salary)
  group by dept
  pivot lunchTime
```

Returns the following response:

| dept | 12:00:00 sum-salary | 13:00:00 sum-salary |
|---|---|---|
| Eng | 1500 | 600 |
| Marketing | null | 800 |

| | | |
|---|---|---|
| Sales | 750 | null |

You can also "invert" this table, switching columns and rows, by switching between the `pivot` columns and the `group by` columns. Running the following query on the example table (#Table_used_in_Examples):

```
select lunchTime, sum(salary)
  group by lunchTime
  pivot dept
```

Returns the following response:

| lunchTime | Eng sum-salary | Marketing sum-salary | Sales sum-salary |
|---|---|---|---|
| 12:00:00 | 1500 | null | 750 |
| 13:00:00 | 600 | 800 | null |

You can also use more than one column in the `pivot` clause. In such a case the columns of the response table are composed of all the unique combinations of values in the columns that exist in the original table. For instance, running the following query on the example table (#Table_used_in_Examples):

```
select sum(salary)
  pivot dept, lunchTime
```

Returns the following response:

| Eng,12:00:00 sum-salary | Eng,13:00:00 sum-salary | Marketing,13:00:00 sum-salary | Sales,12:00:00 sum-salary |
|---|---|---|---|
| 1500 | 600 | 800 | 750 |

Note that only the combinations that appear in the original table are given columns in the response table. This is why there is no column for Marketing,12:00:00 or for Sales,13:00:00.

Using more than one aggregation is also possible. For instance, running the following query on the example table (#Table_used_in_Examples):

```
select sum(salary), max(lunchTime)
  pivot dept
```

Returns the following response:

| Eng sum-salary | Marketing sum-salary | Sales sum-salary | Eng max-lunchTime | Marketing max-lunchTime | Sales max-lunchTime |
|---|---|---|---|---|---|
| 2100 | 800 | 750 | 13:00:00 | 13:00:00 | 12:00:00 |

You can combine multiple aggregations in the `select` clause, multiple columns in the `group by` clause and multiple columns in the `pivot` clause. Internally, aggregation is performed by the concatenation of the columns in the group by and pivot clauses.

Columns specified in the `pivot` clause may not appear in the `select`, `group by` or `order by` clauses. When `pivot` is used, the `order by` clause cannot contain any aggregation columns. The reason for that is that for each aggregation specified in the `select` clause, many columns are generated in the result table. However, you can format aggregation columns when `pivot` is used. The result of such a format is that all of the new columns relevant to the specific aggregation, that are generated by the pivot operation, are formatted by the specified pattern. In the example above, adding `format sum(salary) "some_format_string"` will affect the following columns: Eng sum-salary, Marketing sum-salary and Sales sum-salary.

You can label aggregation columns. If no label is specified in the `label` clause, the label of a column that is produced as a result of pivoting is composed of the list of values in the pivot columns, the aggregation type (min, max, sum, ...) and the aggregated column's label. For example "Eng,12:00:00 sum Salary". If only one aggregation was specified in the `select` clause then the aggregation part is removed from the label, and only the list of values in the pivot columns is kept. For example "Eng,12:00:00". When a `label` clause specifies a label for an aggregation column, then the label requested is appended to the list of values, both when there is only one aggregation in the `select` clause, and when there is more than one. For example, `label sum(salary) "sumsal"` will result in the column labels "Eng,12:00:00 sumsal", "Eng,13:00:00 sumsal", etc.

## Order By

The `order by` clause is used to sort the rows by the values in specified columns.

Items in an `order by` clause can be column identifiers, or the output of aggregation functions (#aggregation_functions), scalar functions (#scalar_functions), or operators (#operators).

**Examples:**

```
order by dept, salary desc
```

```
select dept, max(salary) group by dept order by max(salary)
```

## Limit

The `limit` clause is used to limit the number of returned rows.

**Example:**

```
limit 100
```

## Offset

The `offset` clause is used to skip a given number of first rows. If a `limit` (#Limit) clause is used, `offset` is applied first: for example, `limit 15 offset 30` returns rows 31 through 45.

**Examples:**

```
offset 10
limit 30 offset 210
```

## Label

The **`label`** clause is used to set the label for one or more columns. Note that you cannot use a label value in place of an ID in a query.

Items in a **`label`** clause can be column identifiers, or the output of aggregation functions (#aggregation_functions), scalar functions (#scalar_functions), or operators (#operators).

**Syntax:**

```
label column_id label_string [,column_id label_string]
```

*`column_id`*

The identifier of the column (#Identifiers) being assigned the label.

*`label_string`*

The label to assign to that column. Many visualizations use the column label as text to display to the end-user, such as a legend label in a pie chart. Labels are string literals (#stringliteral), and follow those syntax rules.

**Example:**

The following example sets the label for the dept column to "Department", the label for the name column to "Employee Name", and the label for the location column to "Employee Location":

```
label dept 'Department', name "Employee Name", location 'Empl
```

## Format

The `format` clause is used to specify a formatted value for cells in one or more columns. The returned data should include both an actual value and a formatted value for each cell in a formatted column. Many visualizations use the unformatted value for calculations, but the formatted value for display. The patterns that you specify in this clause are usually returned in the pattern (/chart/interactive/docs/reference#dataparam) property of the corresponding columns.

**Pattern Syntax:**

**`number`, `date`, `timeofday`, `datetime`**

> The date (https://unicode-org.github.io/icu-docs/apidoc/released/icu4j/com/ibm/icu/text/SimpleDateFormat.html) and number (https://unicode-org.github.io/icu-docs/apidoc/released/icu4j/com/ibm/icu/text/DecimalFormat.html) patterns defined by the ICU (http://site.icu-project.org/).

`boolean`

Pattern is a `string` in the format '*value-if-true*:*value-if-false*'.

**Example:**

```
format salary '#,##0.00', hireDate 'dd-MMM-yyyy', isSenior 'Y
```

## Options

The `options` clause is used to control additional options for query execution. Possible keywords that can follow the `options` clause are:

- `no_format` Removes formatted values from the result, and leaves only the underlying values. Can be used when the specific visualization does not use the formatted values to reduce the size of the response.
- `no_values` Removes underlying values from the result, and leaves only the formatted values. Can be used when the specific visualization uses only the formatted values to reduce the size of the response.

# Data Manipulation Functions

There are several kinds of operators and functions that let you manipulate or aggregate data in a single column, or compare or combine data across columns. Examples include sum() (to add all values in a column), max (to find the largest value in a column), and + (to add the values of two columns together in the same row).

Some functions can appear in any clause; some can appear in a subset of clauses. This is documented below.

**Example:**

**Given this table...**

| Name | Salary | Tax | StartDate |
|---|---|---|---|
| Sharon | 1000 | 100 | 1/1/2009 |
| Avital | 2000 | 200 | 21/21/2008 |
| Moran | 3000 | 300 | 2/12/2008 |

**If we apply this query...**

```
select upper(name), year(startDate)
```

**We get**

| Name |
|---|
| AVITAL |
| MORAN |
| SHARO |

The following data manipulation functions are defined by the Google Visualization API query language:

- **Aggregation Functions** (#aggregation_functions)
- **Scalar Functions** (#scalar_functions)
- **Arithmetic Operators** (#operators)

## Aggregation Functions

Aggregation functions are passed a single column identifier (#Identifiers), and perform an action across all values in each group (groups are specified by **`group by`** (#Group_By) or **`pivot`** (#Pivot) clauses, or all rows if those clauses are not used).

**Examples:**

```
select max(salary)                // Returns a table with one
select max(salary) group by dept // Returns a table with the
select max(salary) pivot dept    // Returns a one-row table w
                                  //   and the max salary for
```

Aggregation functions can be used in **`select`**, **`order by`**, **`label`**, **`format`** clauses. They *cannot* appear in **`where`**, **`group by`**, **`pivot`**, **`limit`**, **`offset`**, or **`options`** clauses.

Here are the supported aggregation functions:

| Name | Description | Supported Column Types | Return Type |
|---|---|---|---|
| **`avg()`** | Returns the average value of all values in the column for a group. | **`number`** | **`number`** |
| **`count()`** | Returns the count of elements in the specified column for a group. Null cells are not counted. | Any type | **`number`** |

| | | | |
|---|---|---|---|
| `max()` | Returns the maximum value in the column for a group. Dates are compared with earlier being smaller, **`string`**s are compared alphabetically, with case-sensitivity. | Any type | Same type as column |
| `min()` | Returns the minimum value in the column for a group. Dates are compared with earlier being smaller, **`string`**s are compared alphabetically, with case-sensitivity | Any type | Same type as column |
| `sum()` | Returns the sum of all values in the column for a group. | **`number`** | **`number`** |

**Note:** Aggregation functions can only take a column identifier as an argument:

```
max(startDate)                      // OK
min(firstScore) + min(secondScore)  // OK
max(year(startDate))                // INVALID. max requires
sum(salary + perks)                 // INVALID. sum requires
```

## Scalar Functions

Scalar functions operate over zero or more parameters to produce another value. Scalar functions can be passed any expression that evaluates to the parameter of the appropriate type. Note that these types are the types defined in the Literals (#Literals) section of this document, which might be slightly different than the similarly named JavaScript objects.

Note that the column name will be changed by wrapping it with a scalar function.

Scalar functions can take as a parameter anything that evaluates to a single value:

```
year(max(startDate))
datediff(now(), todate(1234567890000))
```

Scalar functions can be used in any of the following clauses: **`select`** (#Select), **`where`** (#Where), **`group by`** (#Group_By), **`pivot`** (#Pivot), **`order by`** (#Order_By), **`label`** (#Label), and **`format`** (#Format).

| Name | |
|---|---|
| `year()` | Returns the year value from a date or datetime value. For example: `year(date "2009-02-05")` returns 2009.<br><br>**Parameters:** One parameter of type `date` or `datetime`<br>**Return Type:** `number` |

| | |
|---|---|
| `month()` | Returns the zero-based month value from a date or datetime value. For example: `month(date "2009-02-05")` returns 1. Note: the months are 0-based, so the function returns 0 for January, 1 for February, etc.<br><br>**Parameters:** One parameter of type `date` or `datetime`<br>**Return Type:** `number` |
| `day()` | Returns the day of the month from a `date` or datetime value. For example: `day(date "2009-02-05")` returns 5.<br><br>**Parameters:** One parameter of type `date` or `datetime`<br>**Return Type:** `number` |
| `hour()` | Returns the hour value from a datetime or `timeofday` value. For example: `hour(timeofday "12:03:17")` returns 12.<br><br>**Parameters:** One parameter of type `datetime` or `timeofday`<br>**Return Type:** `number` |
| `minute()` | Returns the minute value from a `datetime` or `timeofday` value. For example: `minute(timeofday "12:03:17")` returns 3.<br><br>**Parameters:** One parameter of type `datetime` or `timeofday`<br>**Return Type:** `number` |
| `second()` | Returns the second value from a `datetime` or `timeofday` value. For example: |

| | |
|---|---|
| | `second(timeofday "12:03:17")` returns 17.<br><br>**Parameters:** One parameter of type `datetime` or `timeofday`<br>**Return Type:** `number` |
| `millisecond()` | Returns the millisecond part of a `datetime` or `timeofday` value. For example: `millisecond(timeofday "12:03:17.123")` returns 123.<br><br>**Parameters:** One parameter of type `datetime` or `timeofday`<br>**Return Type:** `number` |
| `quarter()` | Returns the quarter from a `date` or `datetime` value. For example: `quarter(date "2009-02-05")` returns 1. Note that quarters are 1-based, so the function returns 1 for the first quarter, 2 for the second, etc.<br><br>**Parameters:** One parameter of type `date` or `datetime`<br>**Return Type:** `number` |
| `dayOfWeek()` | Returns the day of week from a `date` or `datetime` value. For example: `dayOfWeek(date "2009-02-26")` returns 5. Note that days are 1-based, so the function returns 1 for Sunday, 2 for Monday, etc.<br><br>**Parameters:** One parameter of type `date` or `datetime`<br>**Return Type:** `number` |

| | |
|---|---|
| `now()` | Returns a datetime value representing the current **`datetime`** in the GMT timezone.<br><br>**Parameters:** None<br>**Return Type:** `datetime` |
| `dateDiff()` | Returns the difference in days between two **`date`** or **`datetime`** values. Note: Only the **`date`** parts of the values are used in the calculation and thus the function always returns an integer value. For example: `dateDiff(date "2008-03-13", date "2008-02-12")` returns 29; `dateDiff(date "2009-02-13", date "2009-03-13")` returns -29. Time values are truncated before comparison.<br><br>**Parameters:** Two parameters of type **`date`** or **`datetime`** (can be one of each)<br>**Return Type:** `number` |
| `toDate()` | Transforms the given value to a **`date`** value.<br><br>• Given a **`date`**, it returns the same value.<br>• Given a **`datetime`**, it returns the **`date`** part. For example: `toDate(dateTime "2009-01-01 12:00:00")` returns "2009-01-01".<br>• Given a **`number`** N, it returns a **`date`** N milliseconds after the Epoch. The Epoch is defined as January 1,1970, 00:00:00 GMT. For example: `toDate(1234567890000)` returns "2009-02-13".<br><br>**Parameters:** One parameter of type **`date`**, **`datetime`**, or **`number`**<br>**Return Type:** `date` |

| | |
|---|---|
| `upper()` | Returns the given `string` in upper case letters. For example: `upper("foo")` returns "FOO".<br><br>**Parameters:** One parameter of type `string`<br>**Return Type:** `string` |
| `lower()` | Returns the given `string` in lower case letters. For example: `lower("Bar")` returns "bar".<br><br>**Parameters:** One parameter of type `string`<br>**Return Type:** `string` |

## Arithmetic Operators

You can use arithmetic operators to perform mathematic operations upon anything that evaluates to single number (that is, the output of appropriate aggregate functions, operators, or constants).

**Examples:**

```
select empSalary - empTax
select 2 * (max(empSalary) / max(empTax))
```

The following operators are defined:

| Name | Description | Parameters | Return Type |
|---|---|---|---|
| `+` | Returns the sum of two `number` values. | Two `number`s | `number` |

| | | | |
|---|---|---|---|
| - | Returns the difference between two **number** values. | Two **numbers** | **number** |
| * | Returns the product of two **numbers**. | Two **numbers** | **number** |
| / | Returns the quotient of two **numbers**. Division by zero returns null. | Two **numbers** | **number** |

# Language Elements

## Literals

Literals are values used for comparisons or assignments. Literals can be **strings**, numbers, boolean values, or various date/time types. Here are some examples of literals used in query syntax:

```
where startDate < date "2008-03-18"  // date "2008-03-18" is
limit 30                             // 30 is a numeric liter
format salary '#,##0.00', isSenior 'not yet:of course!'  // '
```

Here are the formats for each type of literal:

**string**

> A **string** literal should be enclosed in either single or double quotes. **Examples:** **"fourteen" 'hello world' "It's raining"**.

`number`

Numeric literals are specified in decimal notation. **Examples:** `3  3.0  3.14  -71  -7.2  .6`

`boolean`

Boolean literals are either `true` or `false`.

`date`

Use the keyword `date` followed by a `string` literal in the format `yyyy-MM-dd`. **Example:** `date "2008-03-18"`.

`timeofday`

Use the keyword `timeofday` followed by a `string` literal in the format `HH:mm:ss[.SSS]` **Example:** `timeofday "12:30:45"`.

`datetime`

A date and a time, using either the keyword `datetime` or the keyword `timestamp` followed by a `string` literal in the format `yyyy-MM-dd HH:mm:ss[.sss]`. **Example:** `datetime '2008-03-18 12:30:34.123'`

## Identifiers

Identifiers (or IDs) are text `string`s that identify columns.

**Important:** If your identifier

- **Has spaces,**

- **Is a reserved word** (#Reserved_Words),
- **Contains anything but alphanumeric characters or underscores** ([a-zA-Z0-9_]), or
- **Starts with a digit**

it *must* be surrounded by back-quotes (not single quotes).

**Otherwise,** your identifier does not need to be quoted. (Note that not all *keywords* defined by the syntax are *reserved* words; so, for example, you can use "max" as an identifier, without having to back-quote it.)

**Examples:** `col1` `employee_table` `` `start date` `` `` `7 days traffic` `` `` `select` ``

We recommend against choosing an identifier that requires back-quotes, because it can be easy to forget to use the back-quotes, or to accidentally use 'single quotes' instead of `back-quotes`. These are common mistakes, and often hard to debug.

## Case Sensitivity

Identifiers and string literals are case-sensitive. All other language elements are case-insensitive.

## Reserved Words

The following reserved words must be back-quoted if used as an identifier (#Identifiers):

```
and
asc
by
date
datetime
desc
false
format
group
label
limit
not
offset
options
or
order
pivot
select
timeofday
timestamp
true
where
```