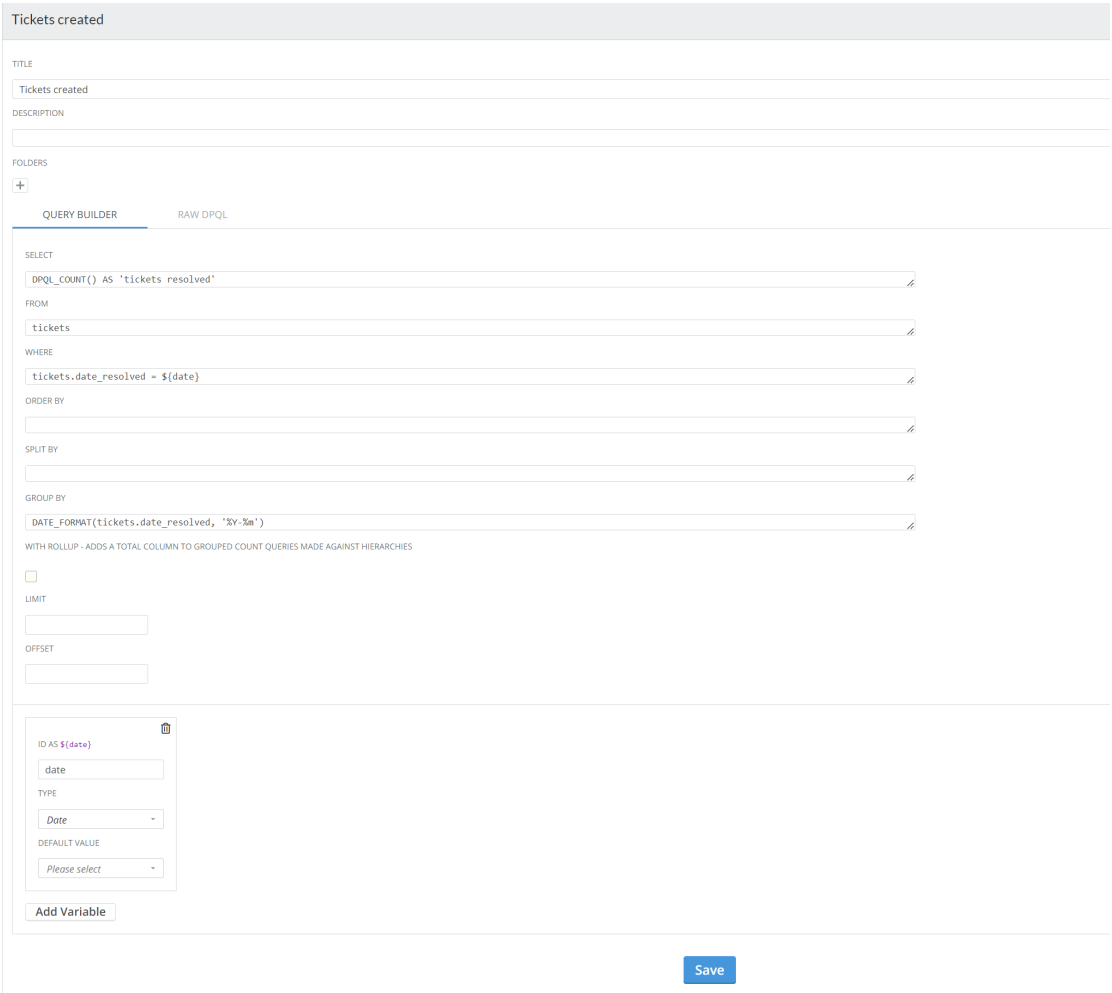
Reports : DPQL Reference

Matthew Wray - 2023-09-27 - Yorumlar (0) - Developer & Reporting

Anatomy of a DPQL query

Reports in the report builder are written using the Deskpro Query Language, known as DPQL. It is similar to SQL, but you don't need to know SQL to use it.

Select any Stat in the **Stat Builder** and click the **Edit Stat button** to see its DPQL query.



DPQL queries consists of a series of clauses:

SELECT *select expression*
FROM *database table*

[WHERE *conditions*]
[SPLIT BY *split fields*]
[GROUP BY *group fields*]
[ORDER BY *order fields*]
[LIMIT *amount* [OFFSET *offset amount*]]

[LAYER WITH *another query]*

[IN *subquery]*

- Clauses in [square brackets] are optional.
- *Italic text* is a placeholder for a value that you enter.

SELECT

The SELECT clause works with the WHERE clause to define what information you want your report to include. It often contains **field references** which specify the data you want to retrieve from the table you choose in the FROM clause.

Your SELECT clause would be a comma-separated list of column references:

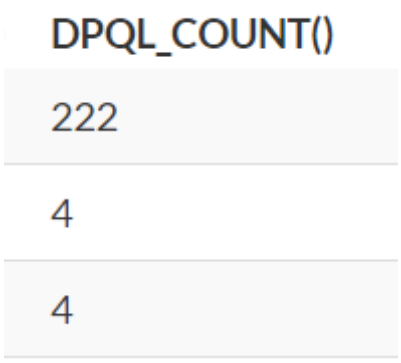
```
SELECT tickets.id, tickets.subject, tickets.person, tickets.department,
tickets.date_created, tickets.agent
```

The SELECT clause can also use DPQL functions.

For example, suppose you didn't want a detailed table of all matching tickets, just the total number. You would use the DPQL_COUNT() function to count the total number of matching tickets.

```
SELECT DPQL_COUNT()
```

In table output, the header text for a column is automatically derived from the SELECT used to produce the column. This may not always result in a good column name; for example, if you just used , you'll produce a table like this:

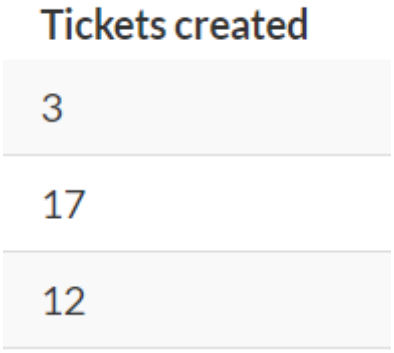


DPQL_COUNT()
222
4
4

You can use an **alias** to specify a better name for a column.

```
DPQL_COUNT() AS 'Total Created'
```

would make the result display like this:



Tickets created
3
17
12

FROM

The MySQL database which stores all your helpdesk data is organised into a series of tables. For example, the table stores information about tickets.

The FROM clause specifies which database table your DPQL query is asking about. You can only pick one table.

All column references in the query must start with the table name specified in the FROM clause.

The tables available are listed here.

WHERE

The conditions in the optional WHERE clause are used to limit what data is displayed or used in a calculation.

For example, if your query was

```
SELECT tickets.id
FROM tickets
```

you'd get a table with all ticket's ID.

But with:

```
SELECT tickets.id
FROM tickets
WHERE tickets.status = 'awaiting_agent'
```

you'd get a table with the IDs of just the tickets with a status of Awaiting Agent.

Conditions can be joined by operators such as AND, OR, NOT, IN and parentheses (brackets) to make complex expressions.

IN is useful if you want to match any one of a number of different values. For example:

```
WHERE tickets.status IN ('awaiting_agent','awaiting_user')
```

will return all tickets with either Awaiting Agent or Awaiting User status.

```
tickets.labels.label != NULL
```

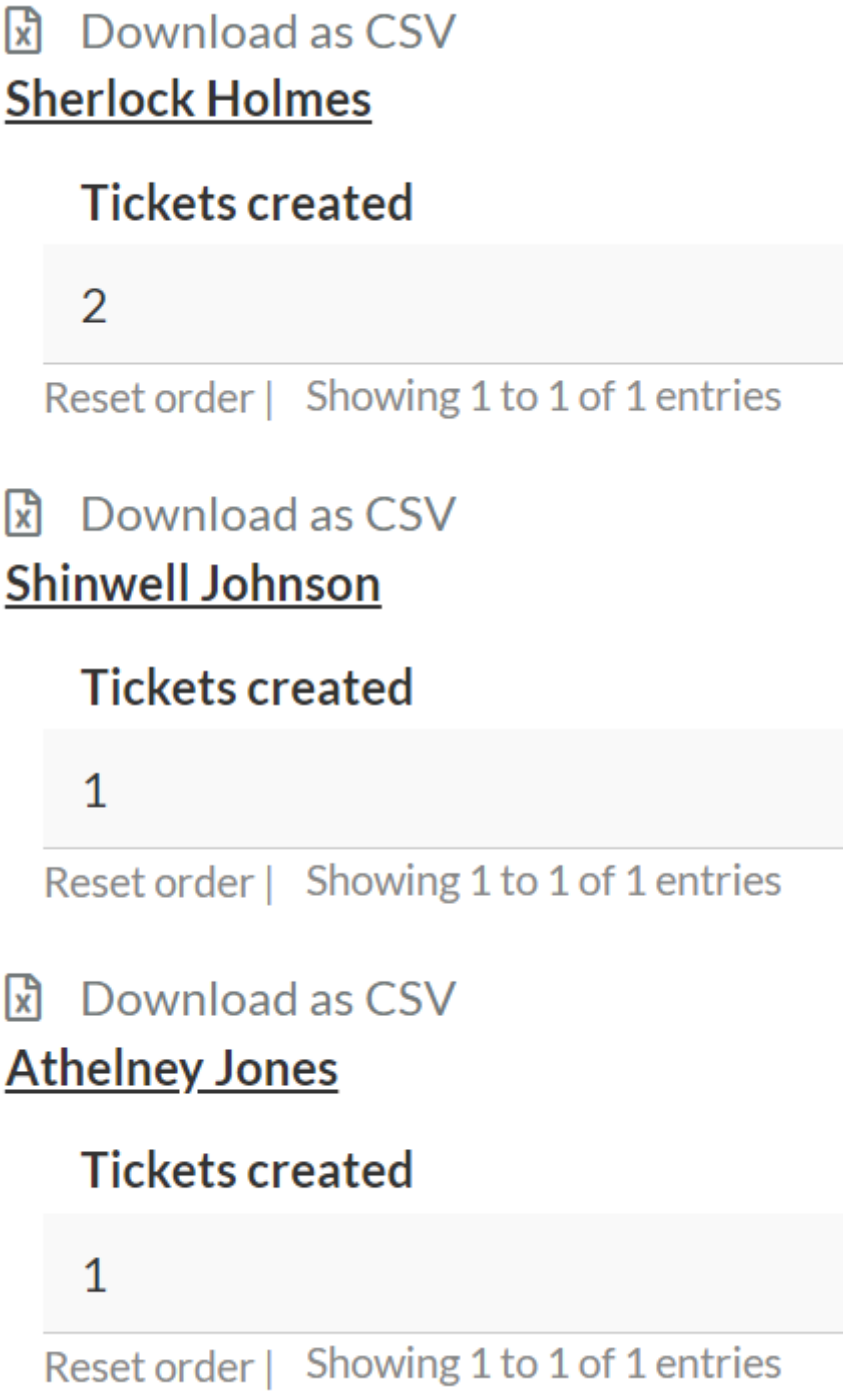
SPLIT BY

The optional SPLIT BY clause enables you split the results into separate tables/graphs by providing split field values.

For example, if you wanted to display each agent's matching tickets for a query in a separate table, you'd use:

```
SPLIT BY tickets.agent
```

and the result would look like this:



with a separate table for each agent.

You can use multiple split fields by providing a comma-separated list of expressions. This will result in separate tables for each combination of of the fields you provide, with separate tables for Agent A/Department A, Agent A/Department B, Agent B/Department A, etc.

GROUP BY

The GROUP BY clause is used to group records that have the same values for the specified fields.

For example, you can group by ticket labels:

```
SELECT DPQL_COUNT() AS 'Total Tickets'
FROM tickets
WHERE tickets.status = 'awaiting_agent'
GROUP BY tickets.labels
```

In the output, tickets with the same label are grouped together:

Labels	Tickets created
+£5k	1
APP Issue	2
AS400	1
Billing	9
bug	1
Complaint	11
Demo Queue	1
Director Issue	3
Forward	3
IT	1

You can also use groups to do calculations such as determining totals and averages.

The group by fields will automatically be displayed in the resulting table, so there is no need to add them to the SELECT clause.

As in the SELECT clause, group fields can be aliased with to change the name of the table header row.

To create a matrix table, use the DPQL_MATRIX() function in the GROUP BY clause to specify two groups. The first specifies the values going across the top of the table, while the second controls the values going down the left side.

ORDER BY

The ORDER BY clause determines how the rows will be ordered when they are returned. If no order is given, the results will be displayed in an undefined order.

The order fields are comma-separated expressions. Ordering will happen across the fields from left to right (ordering by the first expression, then using the second to resolve ties, and so on). Each expression may optionally have or appended to it to control whether ordering is in ascending or descending order. (Ascending is the default if no direction is specified).

For example:

```
ORDER BY SUM(tickets.total_user_waiting) DESC
```

would order a list of users or organizations by their total waiting time, with the highest waiting time at the top.

The ORDER BY clause can access aliases that were specified in the SELECT or GROUP BY clauses using the syntax (for example:). Referencing an alias causes the results to be ordered as if you had written the aliased expression in the ORDER BY clause.

For example, suppose you want to make a table showing how many tickets each agent has, sorted in descending order:

```
SELECT DPQL_COUNT() AS 'Tickets'
FROM tickets
GROUP BY tickets.agent
ORDER BY @'Tickets' DESC
```

Agent	Tickets
Sherlock Holmes	2
Christine	2
Shinwell Johnson	1
Athelney Jones	1
Greg Lestrade	1

LIMIT / OFFSET

The LIMIT clause enables you to limit the number of rows returned. If you don't specify a limit amount, a default limit of 2500 will be used to ensure correct operation.

You can use the OFFSET clause to skip over a certain number of rows before returning the LIMIT *amount* of rows. The OFFSET defaults to 0.

LAYER WITH

LAYER WITH allows you to combine two different queries into one. For example:

```
SELECT DPQL_COUNT(*), tickets.department
FROM tickets
WHERE tickets.date_created = %TODAY%
GROUP BY tickets.department

LAYER WITH

SELECT DPQL_COUNT(*), tickets.department
FROM tickets
WHERE tickets.date_created = %YESTERDAY%
GROUP BY tickets.department
```

IN

You can use IN to create subqueries. For example:

```
SELECT tickets.id
FROM tickets
WHERE tickets.id IN (
    SELECT tickets.id
    FROM tickets
```

```
    WHERE tickets.ref LIKE 'AAAA-%'
)
```

General expression format

Most clauses of a DPQL statement accept a general expression format. You can use expressions to carry out more complicated queries.

Complex expressions are made up of smaller, simpler expressions. Expressions are made up of the following components:

Type	Example	Details
Numbers	5, 37.4	Simple, literal references to integer or decimal values
Strings	‘string’ or “string”	Strings are literal references to text. These will commonly be used in tests/comparisons and aliases.
Null	tickets.labels.label != NULL	You can use this in tests to check whether a certain value exists. If a certain value has been set, it will be not equal to NULL.
Parentheses	(tickets + chat_conversations)/agents	Parentheses are used to enforce the order in which the expression is evaluated.
Column references	table.col[.col2…]	Column references directly retrieve data from your helpdesk. tickets.subject and tickets.agent.name are both valid column references. You can use multiple column parts to combine data from different tables based on a shared field between them (the equivalent of an SQL join). For example, tickets.person.organization.name which would retrieve the name of the organization the person that started the ticket belongs to.
Function calls	COUNT(), CURDATE(), MATRIX(group X, group y)	These are useful functions, for example doing calculations on the inputs (arguments) you give them, changing formatting, or retrieving information. COUNT() is a function. Some functions don’t need any input, e.g. CURDATE() just gets the current date.

Comparisons	(expression) =, !=, >=, >, <, <=	Enable you to compare two expressions e.g. tickets.count_agent_replies >= 10 checks if the number of agent replies on a ticket is greater than or equal to 10. If the comparison is true, the value of the comparison = 1. If it's false, it = 0.
Placeholders	%TODAY%, %LAST_YEAR%	Placeholders are dynamic elements of a DPQL query that are automatically updated to the appropriate value when the query is run. They are useful in comparisons e.g. you can use tickets.created_date = %PAST_7_DAYS% would match tickets created over the last 7 days. Note that these are relative to the local time, e.g. %TODAY% means the current day in the timezone set in your agent preferences.
Logic	AND, OR	Used to logically combine two expressions.

Date and time references

There are two ways to reference dates and times in your queries:

- **absolute** dates/times: e.g. 31st October 2013, or 11am on 1 May 2014
- **relative** time periods using placeholders: e.g. %LAST_WEEK%

To avoid confusion, it's best to avoid combining absolute and relative dates/times in one query.

Absolute dates and times

Absolute dates/times can be referenced in two formats:

YYYY-MM-DD - e.g. ; refers to a date only. This implicitly has a time of 00:00:00 of the specified day.

YYYY-MM-DD HH:MM:SS - e.g. ; refers to a date and a specific time in 24-hour format.

Note that you must specify these dates in the UTC timezone - however, the results returned will be shown adjusted to your timezone (as set in

the **Preferences** section of the agent interface).

This is an example DPQL query to list the tickets created from October 1st to 15th, 2012 in UTC:

```
SELECT tickets.id
 FROM tickets
 WHERE tickets.date_created >= '2012-10-01'
 AND tickets.date_created < '2012-10-16'
```

In date/time comparisons, > (greater than) matches dates/times that are *later*, and < (less than) matches dates/times *earlier*.

You can add or subtract periods of time using the MySQL INTERVAL argument. For example:

```
'2012-10-01' + INTERVAL 2 WEEK
```

means a date/time two weeks after October 1st 2012.

You can use , , , etc.

This is useful when adjusting absolute dates to match your timezone. If you wanted to adjust the above example query to find tickets created from October 1st to 15th Eastern Standard Time, you could change it to:

```
WHERE tickets.date_created >= '2012-10-01' + INTERVAL 5 HOUR
AND tickets.date_created < '2012-10-16' + INTERVAL 5 HOUR
```

This adjusts the times from UTC to 5 hours later, ie EST.

A timezone that is *behind* UTC needs the time difference added to the comparison date/time; a timezone that is *ahead* needs the time difference subtracted.

Relative dates with placeholders

It’s often more useful to have a report that matches helpdesk data for a relative time period, e.g. the current week or the last month, rather than specific dates.

You can write a report like this using **date placeholders** such as .

For example, if your WHERE clause is:

```
WHERE tickets.date_created = %PAST_24_HOURS%
```

it will match all the tickets created within the 24 hours before you run the query.

Note that you use = (equals sign) with placeholders, not < or > as you do with absolute dates/times.

Placeholders use your timezone, as set in your agent account **Preferences**.

For example, if you run a query to find all the tickets created %TODAY%, it will match all the tickets since the current day began *in your timezone*.

If there isn't a placeholder for the interval you need, you can use the SQL NOW function to get the current date, then subtract an INTERVAL.

List of date placeholders

%EVER%	A range covering any date.
%LAST_WEEK%	A range covering the entirety of the previous week, based on a new week starting on a Monday. For example, if today is Wednesday the 12th, this will match Monday the 3rd to Sunday the 9th.
%LAST_MONTH%	A range covering the entirety of the previous month. For example, if today is any day in September 2012, this placeholder will match all of August 2012.
%LAST_YEAR%	A range covering the entirety of the previous year. For example, if today is any day in 2012, this placeholder will match all of 2011.
%PAST_HOUR%	A range covering the hour prior to the time when the report is run.
%PAST_12_HOURS%	A range covering the 12 hours prior to the time when the report is run.
%PAST_24_HOURS%	A range covering from exactly 24 hours ago until when the report is run.
%PAST_7_DAYS%	A range covering the 7 full days prior to the day when the report is run.
%PAST_30_DAYS%	A range covering the 30 full days prior to the day when the report is run.
%PAST_6_MONTHS%	A range covering the 6 months prior to the day when the report is run.
%PAST_12_MONTHS%	A range covering the 12 months prior to the day when the report is run.
%THIS_WEEK%	A range covering the entirety of the current week, from 00:00 on the first day until 23:59 on the last day. Weeks are assumed to start on a Monday and end on a Sunday
%THIS_MONTH%	A range covering the entirety of the current month, from 00:00 on the first day until 23:59 on the last day.
%THIS_YEAR%	A range covering the entirety of the current calendar year, from 00:00 on the first day until 23:59 on the last day.
%TODAY%	A range covering the entirety of the current day.
%TOMORROW%	A range covering the entirety of the next day.

%YESTERDAY%	A range covering the entirety of the previous day.

Remember, placeholders use your timezone, as set in your agent account **Preferences**.

If you have agents in different timezones, they can get different results from queries that use date placeholders.

Dates from custom fields

If you're retrieving a date stored in a **custom field**, it is stored as an integer timestamp rather than real date types, so you need to pass it through the **FROM_UNIXTIME** function first, e.g.:

```
SELECT
    DATE_FORMAT(FROM_UNIXTIME(tickets.custom_data[1]), '%Y-%m-%d') AS 'Date'
```

List of functions

DPQL_COUNT([condition])	Counts the total number of rows matched by the query or current group. If a condition is provided, counts the number of rows in the query or current group that match the condition.
DPQL_COUNT_DISTINCT(expression)	Counts the total number distinct values for the given expression within the rows matched by the query or current group.
DPQL_CURDATE()	Gets the current date in the time zone of the person running the query.
DPQL_CURTIME()	Gets the current time in the time zone of the person running the query.
DPQL_DATE_OFFSET_GROUP(seconds)	Groups a date offset/difference in seconds into human-readable ranges (0–15 minutes, 15–30 minutes, etc).
DPQL_DATE_OFFSET_GROUP(to date, from date)	Displays the difference between the two provided dates as human-readable ranges (0–15 minutes, 15–30 minutes, etc).

DPQL_FORMAT(value, format)	formats the value into the specified format. Possible formats include boolean, number, numberraw, datetime, date, time, year, percent, string.
DPQL_JSON_EXTRACT	This function operates in a similar way to MySQL's JSON_EXTRACT. It lets you SELECT a field in the database that is stored as JSON, and extract a specific value for it for display.
DPQL_LINK(value, link[, params..])	Links to value using the URL provided in link. Placeholders in link are replaced by the additional params. Placeholder values should be represented in sprintf format.
DPQL_MATRIX(group X, group Y)	If both provided groups are non-null creates a matrix table from them. Otherwise, it creates a standard grouped table.
DPQL_NOW()	Gets the current date and time in the time zone of the person running the query.
DPQL_PERCENT(condition[, decimals])	determines the percentage of total rows or rows within the current group that match the condition The results are displayed as a percentage with decimals controlling the precision. 2 decimals are shown by default.
DPQL_PERCENT(sql, printed)	gets the sql but ensures that printed is used if the value is ever going to be displayed. This is mostly helpful in the GROUP BY clause where you need to group on one expression but display the results of another.
DPQL_TO_UTC(date)	converts date to UTC from the current person’s time zone.

DPQL_UTC(expression)	ensures that all dates and times within this function are calculated using UTC. This can increase performance.

A large number of functions are also available that have the exact same behavior as their :

'ABS'
'ACOS'
'ADDDATE'
'ADDTIME'
'ASCII'
'ASIN'
'ATAN'
'ATAN2'
'AVG'
'BIN'
'BIT_AND'
'BIT_COUNT'
'BIT_LENGTH'
'BIT_OR'
'BIT_XOR'
'CEIL'
'CEILING'
'CHAR'
'CHAR_LENGTH'
'CHARACTER_LENGTH'
'COALESCE'
'CONCAT'
'CONCAT_WS'
'CONV'
'COS'
'COT'
'CRC32'
'DATE_FORMAT'
'DATEDIFF'
'DAYOFYEAR'
'DAY'
'DEGREES'
'ELT'
'EXP'
'EXPORT_SET'
'FIELD'
'FIND_IN_SET'

'FLOOR'
'FORMAT'
'FROM_DAYS'
'FROM_UNIXTIME'
'GREATEST'
'GROUP_CONCAT'
'HEX'
'IF'
'IFNULL'
'INET_ATON'
'INET_NTOA'
'INSERT'
'INSTR'
'INTERVAL'
'ISNULL'
'LAST_DAY'
'LCASE'
'LEAST'
'LEFT'
'LENGTH'
'LN'
'LOCATE'
'LOG10'
'LOG2'
'LOG'
'LOWER'
'LPAD'
'LTRIM'
'MAKE_SET'
'MAKEDATE'
'MAKETIME'
'MAX'
'MICROSECOND'
'MID'
'MIN'
'MOD'
'NOW'
'NULLIF'
'OCT'
'OCTET_LENGTH'
'ORD'
'PERIOD_ADD'
'PERIOD_DIFF'
'POW'

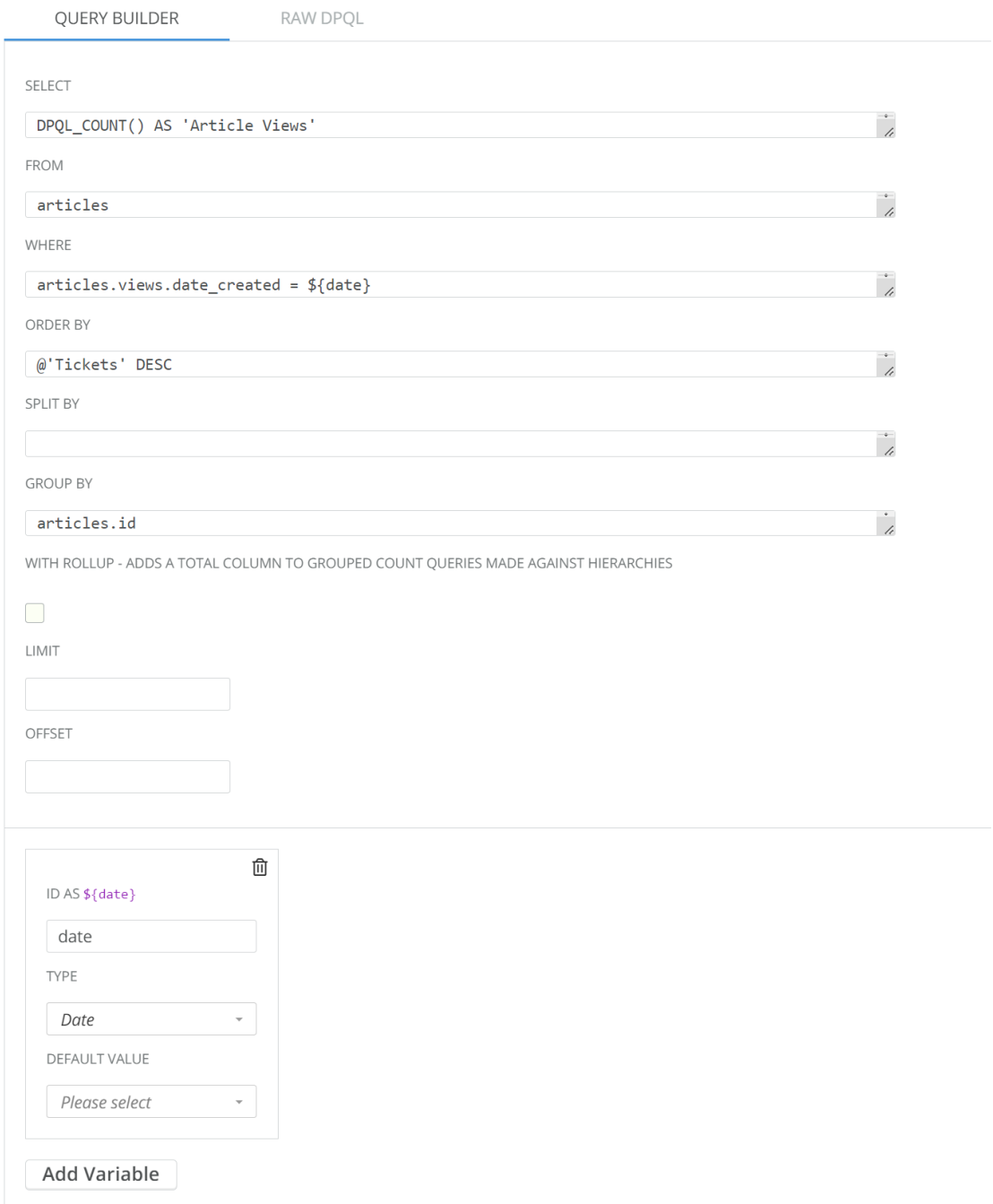
'POWER'
'QUARTER'
'RADIANS'
'RAND'
'REPEAT'
'REPLACE'
'REVERSE'
'RIGHT'
'ROUND'
'RPAD'
'RTRIM'
'SEC_TO_TIME'
'SECOND'
'SIGN'
'SIN'
'SOUNDEX'
'SPACE'
'SQRT'
'STDDEV_POP'
'STDDEV_SAMP'
'STR_TO_DATE'
'STRCMP'
'SUBDATE'
'SUBSTR'
'SUBSTRING'
'SUBSTRING_INDEX'
'SUBTIME'
'SUM'
'TAN'
'TIME'
'TIME_FORMAT'
'TIME_TO_SEC'
'TIMEDIFF'
'TIMESTAMP'
'TO_DAYS'
'TO_SECONDS'
'TRIM'
'TRUNCATE'
'UCASE'
'UNHEX'
'UNIX_TIMESTAMP'
'UPPER'
'UTC_DATE'
'UTC_TIME'

'UTC_TIMESTAMP'
'VAR_POP'
'VAR_SAMP'
'WEEK'
'WEEKDAY'
'WEEKOFYEAR'
'YEARWEEK'
'CURRENT_DATE'
'CURRENT_TIME'
'CURRENT_TIMESTAMP'
'CONVERT_TZ'

Variables

When creating custom reports, you can set up specific values to be dynamically replaced by the user's selection from a pull-down menu.

The same mechanism is used in the built-in reports which have pulldowns to choose a date range, ticket property, grouping field, order, etc.



Each variable has two components:

The title - this is how you define the title of the variable. For example, adding articles.views.date_created = ${date} to a query, would make ${date} the title of the variable for dynamic replacement.

The query - this is where the actual variable is defined to run the correct query. You can build these queries using the variable builder by clicking the 'add variable button'.

The available variables are as follows:

Date Ranges

Possible defaults:

today, yesterday, this_week, this-month, this_year, last_week, last_month, last_year, past_24_hours, past_7_days, past_30_days, ever

Ticket Statuses

Possible defaults:

awaiting_user, awaiting_agent, unresolved, resolved, hidden, any

Field Groups

Possible defaults:

department, agent, agent_team, person, organization, language, usergency, category, priority, workflow, sla, sla_status, hour_created, day_week_created, day_month_created, month_created, year_created, *ticketfield#* (for custom ticket fields), *personfield#* (for custom person fields), *orgfield#* (for custom organization fields), none

Ordering

Possible defaults:

date_created_asc, date_created_desc, last_agent_reply_asc, last_agent_reply_desc, last_user_reply_asc, last_user_reply_desc, total_waiting_asc, total_waiting_desc