**JSON QUERIES FOR ANALYSIS SPECIFICATION**

Version 0.1

Kasun Liyanage

Updated: 03/16/2017

**Notice and Disclaimer**

Copyright © 2017 for the person identified as the document author. All rights reserved.

Permission to copy and display the JQA (JSON Queries for Analysis) specification, in any medium without fee or royalty is hereby granted, provided that you include the following on all copies of the JSON for Analysis Specification, or portions thereof, that you make:

* A link or URL to the JSON for Analysis Specification at this location: <https://github.com/Kasun88/JQA>

All published JQA specification content are subject to change without prior notice and it is advised to check the home page of the JQA and get updated when versions are updated.

The specification may contain links to sites owned or operated by independent third parties. These links are provided for your convenience and reference only. We do not control such sites and, therefore, we are not responsible for any content posted on these sites.

This document will be made publicly available from the version 1.0 under <https://github.com/Kasun88/JQA> but title to copyright in the JQA Specification will at all times remain with the Author. Information in this document, including URL and other Internet Web site references, is subject to change without notice. Unless otherwise noted, the example companies, organizations, products, domain names, email addresses, logos, people, places and events depicted herein are fictitious and no association with any real company, organization, product, domain name, email address, logo, person, places or event is intended or should be inferred.

**Table of Contents**

OVERVIEW 1

AUDIENCE 2

DESIGN GOALS 2

DESIGN SUMMARY 2

JSON QUERIES FOR ANALYSIS 4

METHODS 4

META 4

STORE 4

CATALOG 5

DATASOURCE 5

SCHEMA 6

QUERY 7

SEARCH 7

AGGREGATION 9

FILTER 12

OPTIONS 13

CUSTOM 15

DATA TYPES USED IN XML FOR ANALYSIS 15

ERROR HANDLING IN JQA 15

APPENDIX 17

METHODS 17

META 17

STORE 17

CATALOG 17

DATASOURCE 18

SCHEMA 18

QUERY 20

SEARCH 20

AGGREGATION 26

FILTER 31

OPTIONS 33

CUSTOM 34

# OVERVIEW

Traditional data access techniques, such as driver installation was identified earlier as not good design aspects since it coupled on a specific hardware platform, a specific operating system, a specific interface model, a specific programming language, and a specific match between versions of client and server components. Based on the decoupling design goals, specifications such as XMLA for Analysis (XMLA) was introduced for the communication purposes of Online Analytical Processing (OLAP) data cubes. Even though we evade from the tightly coupled driver components still big data analytics suffers from ad-hoc query structures and communication models, coupling query model for a specific data store.

JQA is a simple and light-weight JSON API, designed specifically for standardizing the data analytical interaction between a client application and the data store. It provides universal data access to any data store that supports JQA communication model.

The final goal of the JQA specification is to define standards to provide reliable data access to any client applications regardless of the underlying dependencies such as device, OS or the hardware architecture and without a change to the communication model. This specification primarily defines three methods, Meta, Query, and Custom, which define JSON request and response for data access and analyze. JQA specification is built upon the open Internet standards of HTTP and JSON and is not bound to any specific language or technology.

## AUDIENCE

This specification targets application developers and assumes the followings:

* Knowledge of JavaScript
* Knowledge of JSON
* Understanding of data analytics and query languages
* Related concepts of relational and big data stores
* Basic knowledge of front-end widgets, dashboards and BI suits

## DESIGN GOALS

The primary design goals of this specification include the followings:

* Provide a standard data access and analytical API to data stores so same query can be executed in any data store
* Enforce Reusability in front widgets, dashboards and queries
* Simplify the query model so users can easily develop a query model with minimum effort
* Support technologically independent implementations using any tool, programming language, technology, hardware platform, or device
* Usage of open Internet standards, such as JSON and HTTP
* Use the power of free and open source community for growth and popularity
* Work efficiently with standard data stores supporting full functionality

## DESIGN SUMMARY

The design centers around a JSON-based communication API, called JSON Queries for Analysis, which defines how the request and responses should be handled using JSON structures.

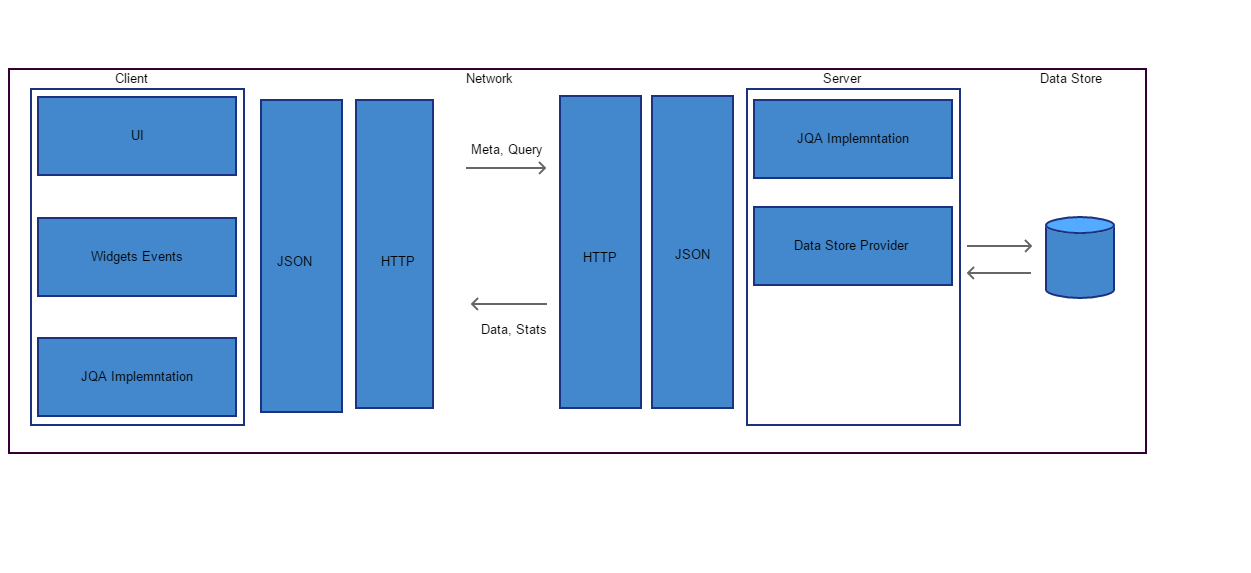


Figure 1: High-level Overview

High-level architecture of JQA would be typical client-server model. The client implements the JQA query generator in on its business layer based on data representation layer widget’s interactions and events (pie chart, bar chart etc...). Then JQA queries will be delivered as JSON HTTP request to the server.

Once the server received the request, it extracts the query and passes it to data store provider. Data store provider parses the query and generates raw commands to execute on the data store to get the result. The result should be transformed back to satisfy the JQA specification and passed back to the client as JSON response. The client then validates and render the data based on the front end visualization widget.

# JSON QUERIES FOR ANALYSIS

## METHODS

JQA specification mainly structured into three methods, Meta, Query, and Custom.

### META

Meta section is used whenever a client needs to get the Meta information about the data store such as data source information, schema details and etc... Meta section acts as a get API in JQA, which means clients only can retrieve the information using this method and cannot pass an update to the data store. The current version of JQA specification supports following methods under Meta section.

* **STORE**
* **DATASOURCE**
* **CATALOG**
* **SCHEMA**

General Syntax for Meta API is

*{*

*"meta": {*

*"<META FUNCTION>": {}*

*}*

*}*

#### STORE

Store method is used to get the information about underlying data store.

*{*

*"meta": {*

*"store": {}*

*}*

*}*

The response returns the data store name and version.

*{*

*"result": {*

*"store": {*

*"store": "<STORE NAME>",*

*"version": "<VERSION INFOTMATION>"*

*}*

*}*

*}*

#### CATALOG

Catalog refers to the logical categorization of stored data. This is somewhat similar to a database in the relational database context.

*{*

*"meta": {*

*"catalog": {}*

*}*

*}*

The response list all available catalog names in the data store.

*{*

*"result": {*

*"catalog": ["<CATALOG NAME>"]*

*}*

*}*

#### DATASOURCE

Data source is the logical grouping of data witlings a catalog. This is similar to a table in relational databases.

Datasource methods take the CATALOG as a parameter and list all available data sources within given catalog.

*{*

*"meta": {*

*"datasource": {*

*"catalog": "<CATALOG NAME>"*

*}*

*}*

*}*

*{*

*"result": {*

*"datasource": ["<DATASOURCE NAME>"]*

*}*

*}*

#### SCHEMA

Schema normally defines as the structure how data stored. Normally it contains the column names and the data types for respective columns. To get the schema information schema method must be requested with a data source parameter.

*{*

*"meta": {*

*"schema": {*

*"catalog": "<CATALOG NAME>",*

*"datasource": "<DATASOURCE NAME>"*

*}*

*}*

*}*

*{*

*"result": {*

*"schema": [{*

*"field": "<COLUMN NAME>",*

*"datatype": "<DATATYPE>"*

*}]*

*}*

*}*

### QUERY

Query API provides a simple and powerful way to access and perform the analytics on the data store. Query API mainly divided into four sections, SEARCH, AGGREGATION, FILTER and OPTION.

Query API should always follow with the STORE parameter that specifies Catalog and the Datasource which Query should apply.

*{*

*"query": {*

*"search | aggregation | filter | option": {*

*"<QUERY FUNCTION>": {}*

*}*

*},*

*"store": {*

*"catalog": "<CATALOG NAME>",*

*"datasource": "<DATASOURCE NAME>"*

*}*

*}*

#### SEARCH

Search API is responsible for getting none analytical result such as select all or specific column.

General syntax for search function is

*{*

*"query": {*

*"search": {*

*"<SEARCH FUNCTION>": {}*

*}*

*}*

*}*

The current version of JQA supports following methods under SEARCH API.

##### LIST

List method simply selects all the columns and respective values for the client, if the column values are passed to the method it will only list the given columns otherwise act as select all in normal SQL context.

List all syntax defined as below,

*{*

*"query": {*

*"search": {*

*"list": {}*

*}*

*}*

*}*

Query to List set of fields

*{*

*"query": {*

*"search": {*

*"list": {*

*"fields": ["<FIELD NAME>","<FIELD NAME>”]*

*}*

*}*

*}*

*}*

##### TOP\_LIST

When you want to list only the top most records, TOP\_LIST method is used. Top parameter decides how many records to be returned in the result.

*{*

*"query": {*

*"search": {*

*"top\_list": {*

*"fields": ["<FIELD NAME>", "<FIELD NAME>"],*

*"top": "<NUMBER OF RECORDS>"*

*}*

*}*

*}*

*}*

#### AGGREGATION

Aggregation API supports standards analytical functionalities such as COUNT, SUM, and AVG.

Current version of JQA supports following methods under AGGREGATION API. General syntax for the aggregation is below,

*{*

*"query": {*

*"aggregation": {*

*"function": "<AGGREGATION FUNTION>",*

*"field": "<FIELD NAME>",*

*"grouping": "<FIELD NAME>",*

*"having": {*

*"aggregation": {*

*"function": "<AGGREGATION FUNTION>",*

*"field": "<FIELD NAME>",*

*"filters": "<FIELD NAME>"*

*}*

*}*

*}*

*}*

*}*

* **COUNT**
* **COUNT\_D**
* **SUM**
* **AVG**
* **MIN**
* **MAX**

##### COUNT

The count is the standard mathematical count function it counts the occurrence of given columns and returns the result as the scalar value. Count method takes the column or list of columns as the parameter.

*{*

*"query": {*

*"aggregation": {*

*"function": "count",*

*"field": "<FIELD NAME>"*

*}*

*}*

*}*

Default response for an aggregation functions is a single value as below,

{

"result": {

"aggregation": {

"value": "<VALUE>"

}

}

}

##### COUNT\_D

COUNT\_D act same as COUNT function but it only counts the distinct values.

*{*

*"query": {*

*"aggregation": {*

*"function": "count\_d",*

*"field": "<FIELD NAME>"*

*}*

*}*

*}*

##### SUM

Sum up the given column values and provide a scalar value as the response.

*{*

*"query": {*

*"aggregation": {*

*"function": "sum”,*

*"field": "<FIELD NAME>"*

*}*

*}*

*}*

##### AVG

Average across all the values in given column.

*{*

*"query": {*

*"aggregation": {*

*"function": "avg",*

*"field": "<FIELD NAME>"*

*}*

*}*

*}*

##### MIN

Standard Min function to get the minimum value of the given column.

*{*

*"query": {*

*"aggregation": {*

*"function": "min",*

*"field": "<FIELD NAME>"*

*}*

*}*

*}*

##### MAX

Standard Max function to get the maximum value of the given column.

*{*

*"query": {*

*"aggregation": {*

*"function": "max",*

*"field": "<FIELD NAME>"*

*}*

*}*

*}*

#### FILTER

This section defines the filtering criteria for the query, SEARCH or the AGGREGATIONS should be applied top of these filters.

Current filter method supports basic boolean algebra expressions and comparison operators such as AND, &, OR, ||, =, >, <, >=, <=, !=. Filters can be nested inside filters to make complex filtering criteria.

The basic filter contains the column name, operator and a value to filter.

*{*

*"query": {*

*"filters": {*

*"filter": {*

*"field": "<FIELD NAME>",*

*"operator": "<OPERATOR>",*

*"filter\_value": "<FILTER VALUE>",*

*"condition": "<CONDITION>",*

*"filter": {…}*

*}*

*}*

*}*

*}*

#### OPTIONS

Options section allows users to set additional parameters that manipulate the query response. We have following methods under the soptions.

##### SORT

Typical sorting function to sort the result based on columns. It allows adding one or more sorts of specific fields. Basic sort takes column name and direction, ASC or DSC as a parameter.

*{*

*"query": {*

*"options": {*

*"sort": {*

*"fields": ["<FIELD NAME>"],*

*"order": "<ORDER>"*

*}*

*}*

*}*

*}*

##### PAGE

Pagination of results can be done by using the PAGE\_SIZE and PAGE\_NUM parameters. PAGE\_NUM represent the page number and PAGE\_SIZE is the record size on one page.

*{*

*query": {*

*"options": {*

*"page": {*

*"page size": "<SIZE>",*

*"page\_num": "<PAGE\_NUMBER>"*

*}*

*}*

*}*

*}*

##### STAT

Stat field determines whether stats information re included with each query response, If the client wants to get the stats information about the query been executed, STAT field should be set to TRUE specifically.

*{*

*"query": {*

*"options": {*

*"stat": "<TRUE| FALSE>"*

*}*

*}*

*}*

If stats is set to true, then response include the following, additional to the normal query result,

* Number of records processed
* Time take to execute the query
* Query information

*{*

*"result": {*

*"stat": {*

*"<QUERY TYPE>": {*

*"function": "<FUNCTION NAME>",*

*"field": "<FIELD NAME>",*

*"value": "<VALUE>"*

*},*

*"record\_count": "<NUMBER OF RECORDS>",*

*"time": "<TIME TAKE TO EXECUTE THE QUERY >"*

*}*

*}*

*}*

### CUSTOM

Custom section is provided to support the customization out of the defined JQA structure. This can be used to support any arbitrary options, query or R&D level features. TYPE determine the specific categorization of customization and VALUE can be any string that supports that categorization.

{

"custom": {

"type": "<TYPE>",

"value": "<VALUE>"

}

}

## DATA TYPES USED IN XML FOR ANALYSIS

Bool

The Boolean type uses the standard JSON Boolean data type.

Decimal

The Decimal type noted uses the standard JSON decimal data type.

Integer

The Integer type noted in this document refers to the standard JSON int data type.

Text

The String type corresponds to the standard JSON string data type.

## ERROR HANDLING IN JQA

Errors are handled differently, depending on their type. The following types of errors can occur:

* Failure to execute a method call
* Success in executing a method call, but with errors or warnings
* Success in executing a method call, but errors within the result set
* The method call itself did not fail, but a server failure occurred after the method call succeeded.

The format of the general error message as follows.

*{*

*"result": {*

*"<WARNING | ERROR>": {*

*"code": "<CODE>",*

*"description": "<DESCRIPTION>",*

*"message": "<MESAAGE | STACK TRACE | EXCEPTION>"*

*}*

*}*

*}*

The client has the responsibility to handle the error messages returned by the server.

# APPENDIX

## METHODS

Under appendix, sample queries and expected responses are provided for user convenience. All the examples are based on Druid 0.10.1 wikiticker dataset that contains Wikipedia article related data.

### META

* **Datasource**
* **Catalog**
* **Schema**

#### STORE

Request

*{*

*"meta": {*

*"store": {}*

*}*

*}*

Response

*{*

*"result": {*

*"store": {*

*"store": "Druid",*

*"version": "0.10.1"*

*}*

*}*

*}*

#### CATALOG

Request

*{*

*"meta": {*

*"catalog": {}*

*}*

*}*

Response

*{*

*"result": {*

*"catalog": ["wikiticker"]*

*}*

*}*

#### DATASOURCE

Request

*{*

*"meta": {*

*"datasource": {*

*"catalog": "wikiticker"*

*}*

*}*

*}*

Response

*{*

*"result": {*

*"datasource": ["wikipedia"]*

*}*

*}*

#### SCHEMA

Request

*{*

*"meta": {*

*"schema": {*

*"catalog": "wikiticker",*

*"datasource": "wikipedia"*

*}*

*}*

*}*

Response

*{*

*"result": {*

*"schema": [{*

*"field": "time",*

*"datatype": "datetime"*

*}, {*

*"field": "channel",*

*"datatype": "text"*

*}, {*

*"field": "cityname",*

*"datatype": "text"*

*}, {*

*"field": "comment",*

*"datatype": "text"*

*}, {*

*"field": "countryisocode",*

*"datatype": "text"*

*}, {*

*"field": "countryname",*

*"datatype": "text"*

*}, {*

*"field": "isanonymous",*

*"datatype": "bool"*

*}, {*

*"field": "isminor",*

*"datatype": "bool"*

*}, {*

*"field": "isnew",*

*"datatype": "bool"*

*}, {*

*"field": "isrobot",*

*"datatype": "bool"*

*}, {*

*"field": "isunpatrolled",*

*"datatype": "bool"*

*}, {*

*"field": "metrocode",*

*"datatype": "int"*

*}, {*

*"field": "namespace",*

*"datatype": "text"*

*}, {*

*"field": "page",*

*"datatype": "text"*

*}, {*

*"field": "regionisocode",*

*"datatype": "int"*

*}, {*

*"field": "regionname",*

*"datatype": "text"*

*}, {*

*"field": "user",*

*"datatype": "text"*

*}, {*

*"field": "delta",*

*"datatype": "int"*

*}, {*

*"field": "added",*

*"datatype": "int"*

*}, {*

*"field": "deleted",*

*"datatype": "int"*

*}]*

*}*

*}*

### QUERY

#### SEARCH

##### LIST

Request

*{*

*"query": {*

*"search": {*

*"list": {}*

*}*

*}* *,*

*"store": {*

*"catalog": "wikiticker",*

*"datasource": "wikipedia"*

*}*

*}*

Response (Truncated for brevity)

*{*

*"result": {*

*"search": [{*

*"time": "2015-09-12T00:46:58.771Z",*

*"channel": "#en.wikipedia",*

*"cityName": null,*

*"comment": "added project",*

*"countryIsoCode": null,*

*"countryName": null,*

*"isAnonymous": false,*

*"isMinor": false,*

*"isNew": false,*

*"isRobot": false,*

*"isUnpatrolled": false,*

*"metroCode": null,*

*"namespace": "Talk",*

*"page": "Talk:Oswald Tilghman",*

*"regionIsoCode": null,*

*"regionName": null,*

*"user": "GELongstreet",*

*"delta": 36,*

*"added": 36,*

*"deleted": 0*

*}, {*

*"time": "2015-09-12T00:47:00.496Z",*

*"channel": "#ca.wikipedia",*

*"cityName": null,*

*"comment": "Robot inserta {{Commonscat}} que enllaça amb [[commons:category:Rallicula]]",*

*"countryIsoCode": null,*

*"countryName": null,*

*"isAnonymous": false,*

*"isMinor": true,*

*"isNew": false,*

*"isRobot": true,*

*"isUnpatrolled": false,*

*"metroCode": null,*

*"namespace": "Main",*

*"page": "Rallicula",*

*"regionIsoCode": null,*

*"regionName": null,*

*"user": "PereBot",*

*"delta": 17,*

*"added": 17,*

*"deleted": 0*

*}, {*

*"time": "2015-09-12T00:47:05.474Z",*

*"channel": "#en.wikipedia",*

*"cityName": "Auburn",*

*"comment": "/\* Status of peremptory norms under international law \*/ fixed spelling of 'Wimbledon'",*

*"countryIsoCode": "AU",*

*"countryName": "Australia",*

*"isAnonymous": true,*

*"isMinor": false,*

*"isNew": false,*

*"isRobot": false,*

*"isUnpatrolled": false,*

*"metroCode": null,*

*"namespace": "Main",*

*"page": "Peremptory norm",*

*"regionIsoCode": "NSW",*

*"regionName": "New South Wales",*

*"user": "60.225.66.142",*

*"delta": 0,*

*"added": 0,*

*"deleted": 0*

*}, {*

*"time": "2015-09-12T00:47:08.770Z",*

*"channel": "#vi.wikipedia",*

*"cityName": null,*

*"comment": "fix Lỗi CS1: ngày tháng",*

*"countryIsoCode": null,*

*"countryName": null,*

*"isAnonymous": false,*

*"isMinor": true,*

*"isNew": false,*

*"isRobot": true,*

*"isUnpatrolled": false,*

*"metroCode": null,*

*"namespace": "Main",*

*"page": "Apamea abruzzorum",*

*"regionIsoCode": null,*

*"regionName": null,*

*"user": "Cheers!-bot",*

*"delta": 18,*

*"added": 18,*

*"deleted": 0*

*}, …]*

*}*

*}*

Query to List set of fields

*{*

*"query": {*

*"search": {*

*"list": {*

*"fields": ["time", "channel", "comment"]*

*}*

*}*

*}* *,*

*"store": {*

*"catalog": "wikiticker",*

*"datasource": "wikipedia"*

*}*

*}*

Response (Truncated for brevity)

*{*

*"result": {*

*"search": [{*

*"time": "2015-09-12T00:46:58.771Z",*

*"channel": "#en.wikipedia",*

*"comment": "added project"*

*}, {*

*"time": "2015-09-12T00:47:00.496Z",*

*"channel": "#ca.wikipedia",*

*"comment": "Robot inserta {{Commonscat}} que enllaça amb [[commons:category:Rallicula]]"*

*}, {*

*"time": "2015-09-12T00:47:05.474Z",*

*"channel": "#en.wikipedia",*

*"comment": "/\* Status of peremptory norms under international law \*/ fixed spelling of 'Wimbledon'"*

*}, {*

*"time": "2015-09-12T00:47:08.770Z",*

*"channel": "#vi.wikipedia",*

*"comment": "fix Lỗi CS1: ngày tháng"*

*}, …]*

*}*

*}*

##### TOP

Request to retrieve the top 1 record

*{*

*"query": {*

*"search": {*

*"list": {*

*"top": 1*

*}*

*}*

*}* *,*

*"store": {*

*"catalog": "wikiticker",*

*"datasource": "wikipedia"*

*}*

*}*

Response

*{*

*"result": {*

*"search": [{*

*"time": "2015-09-12T00:46:58.771Z",*

*"channel": "#en.wikipedia",*

*"cityName": null,*

*"comment": "added project",*

*"countryIsoCode": null,*

*"countryName": null,*

*"isAnonymous": false,*

*"isMinor": false,*

*"isNew": false,*

*"isRobot": false,*

*"isUnpatrolled": false,*

*"metroCode": null,*

*"namespace": "Talk",*

*"page": "Talk:Oswald Tilghman",*

*"regionIsoCode": null,*

*"regionName": null,*

*"user": "GELongstreet",*

*"delta": 36,*

*"added": 36,*

*"deleted": 0*

*}]*

*}*

*}*

Request to terive top 3 commnets

*{*

*"query": {*

*"search": {*

*"list": {*

*"fields": ["comment"],*

*"top": 3*

*}*

*}*

*}* *,*

*"store": {*

*"catalog": "wikiticker",*

*"datasource": "wikipedia"*

*}*

*}*

Response

*{*

*"result": {*

*"search": [{*

*"comment": "added project"*

*}, {*

*"comment": "Robot inserta {{Commonscat}} que enllaça amb [[commons:category:Rallicula]]"*

*}, {*

*"comment": "/\* Status of peremptory norms under international law \*/ fixed spelling of 'Wimbledon'"*

*}]*

*}*

*}*

#### AGGREGATION

General syntax for the aggregation is below,

*{*

*"query": {*

*"aggregation": {*

*"function": "<AGGREGATION FUNTION>",*

*"field": "<FIELD NAME>",*

*"grouping": "<FIELD NAME>",*

*"having": {*

*"aggregation": {*

*"function": "<AGGREGATION FUNTION>",*

*"field": "<FIELD NAME>",*

*"filters": "<FIELD NAME>"*

*}*

*}*

*}*

*}*

*}*

* **COUNT**
* **COUNT\_D**
* **SUM**
* **AVG**
* **MIN**
* **MAX**

##### COUNT

Request to get the record count, note that is doesn’t matter which column if column contains in all the records.

*{*

*"query": {*

*"aggregation": {*

*"function": "count",*

*"field": "time"*

*}*

*}*

*}*

Response for the COUNT

*{*

*"result": {*

*"aggregation": {*

*"value": "39244"*

*}*

*}*

*}*

##### COUNT\_D

Request to get distinct Channels

*{*

*"query": {*

*"aggregation": {*

*"function": "count\_d",*

*"field": "channel"*

*}*

*}*

*}*

Response for distinct Channels

*{*

*"result": {*

*"aggregation": {*

*"value": "28"*

*}*

*}*

*}*

##### SUM

Request to get sum of added wikis.

*{*

*"query": {*

*"aggregation": {*

*"function": "sum",*

*"field": "added"*

*}*

*}*

*}*

Response for wiki addeds

*{*

*"result": {*

*"aggregation": {*

*"value": "1892"*

*}*

*}*

*}*

##### AVG

Request to get average wiki deletes

*{*

*"query": {*

*"aggregation": {*

*"function": "avg",*

*"field": "deleted"*

*}*

*}*

*}*

Response for average wiki deletes

*{*

*"result": {*

*"aggregation": {*

*"value": "512"*

*}*

*}*

*}*

##### MIN

Request to get minimum wiki added count

*{*

*"query": {*

*"aggregation": {*

*"function": "min",*

*"field": "added"*

*}*

*}*

*}*

Response for minimum wiki added

*{*

*"result": {*

*"aggregation": {*

*"value": "0"*

*}*

*}*

*}*

##### MAX

Request to get maximum wiki deletes

*{*

*"query": {*

*"aggregation": {*

*"function": "max",*

*"field": "* *deleted"*

*}*

*}*

*}*

Response for maximum wiki deletes

*{*

*"result": {*

*"aggregation": {*

*"value": "278"*

*}*

*}*

*}*

#### FILTER

Current filter method supports basic boolean algebra expressions and comparison operators such as AND, &, OR, ||, =, >, <, >=, <=, !=. Filters can be nested inside filters to make complex filtering criteria.

The basic filter syntax.

*{*

*"query": {*

*"filters": {*

*"filter": {*

*"field": "<FIELD NAME>",*

*"operator": "<OPERATOR>",*

*"filter\_value": "<FILTER VALUE>",*

*"condition": "<CONDITION>",*

*"filter": {…}*

*}*

*}*

*}*

*}*

*Basic syntax to get anonymous edits from Wikipedia.*

{

"query": {

"filters": {

"filter": {

"field": "isAnonymous",

"operator": "=",

"filter\_value": "true"

}

}

}

}

*Basic syntax to filter country name null records.*

{

"query": {

"filters": {

"filter": {

"field": " countryName",

"operator": "!=",

"filter\_value": "null"

}

}

}

}

*Combined filters to get anonymous edits done by adults.*

{

"query": {

"filters": {

"filter": {

"field": "isAnonymous",

"operator": "=",

"filter\_value": "true",

"condition": "and",

"filter": {

"field": "isMinor",

"operator": "=",

"filter\_value": "false"

}

}

}

}

}

#### OPTIONS

##### SORT

*Basic syntax to get Wikipedia results sorted by time in ascending order.*

*{*

*"query": {*

*"options": {*

*"sort": {*

*"fields": ["time"],*

*"order": "ASC"*

*}*

*}*

*}*

*}*

##### PAGE

*{*

*query": {*

*"options": {*

*"page": {*

*"page size": "10",*

*"page\_num": "1"*

*}*

*}*

*}*

*}*

##### STAT

*{*

*"query": {*

*"options": {*

*"stat": "TRUE"*

*}*

*}*

*}*

Simple STAT request for MAX query.

*{*

*"query": {*

*"aggregation": {*

*"function": "max",*

*"field": “deleted"*

*},*

*"options": {*

*"stat": "TRUE"*

*}*

*}*

*}*

Sample STAT response for MAX query.

*{*

*"result": {*

*"stat": {*

*“aggregation": {*

*“FUNCTION ": "MAX",*

*"Field": “deleted ",*

*"value": "278"*

*},*

*"record\_count": "39244",*

*"time": "10"*

*}*

*}*

*}*

### CUSTOM

*{*

*"custom": {*

*"type": "mssql",*

*"value": "select top 10 countryName from wikiticker;"*

*}*

*}*