**JSON QUERIES FOR ANALYSIS SPECIFICATION**

Version 0.1

Kasun Liyanage

Updated: 06/16/2018

**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**

LIST OF FIGURES 2

OVERVIEW 4

AUDIENCE 5

DESIGN GOALS 5

DESIGN SUMMARY 5

JSON QUERIES FOR ANALYSIS 7

METHODS 7

META 7

STORE 7

CATALOG 8

DATASOURCE 9

SCHEMA 10

QUERY 11

SEARCH 11

AGGREGATION 13

FILTER 17

OPTIONS 18

CUSTOM 21

DATA TYPES USED IN XML FOR ANALYSIS 22

ERROR HANDLING IN JQA 22

APPENDIX 24

METHODS 24

META 24

STORE 24

CATALOG 25

DATASOURCE 25

SCHEMA 26

QUERY 28

SEARCH 28

AGGREGATION 37

FILTER 42

OPTIONS 45

CUSTOM 47

# LIST OF FIGURES

Figure 1: High-level Overview 6

Figure 2: General Syntax for Meta API 7

Figure 3: General Syntax for Store Request 8

Figure 4: General Response for Store Request 8

Figure 5: General Syntax for Catalog Request 8

Figure 6: General Response for Catalog Request 9

Figure 7: General Syntax for Datasource Request 9

Figure 8: General Response for Datasource Request 10

Figure 9: General Syntax for Schema Request 10

Figure 10: General Response for Schema Request 10

Figure 11: General Syntax for Query API 11

Figure 12: General Syntax for Search API 11

Figure 13: General Syntax for List All 12

Figure 14: General Syntax for List Selected Fields 12

Figure 15: General Syntax for Top List Request 13

Figure 16: General Syntax for Aggregation API 14

Figure 17: General Syntax for Count 15

Figure 18: General Response for Aggregation Functions 15

Figure 19: General Syntax for Distinct Count 15

Figure 20: General Syntax for Sum 16

Figure 21: General Syntax for Average 16

Figure 22: General Syntax for Minimum 16

Figure 23: General Syntax for Maximum 17

Figure 24: General Syntax for Filter API 18

Figure 25: Sorting Syntax for Option 19

Figure 26: Paging Syntax for Option 19

Figure 27: Statistics Syntax for Option 20

Figure 28: General Response for Statistics 21

Figure 29: General Syntax for Custom API 21

Figure 30: General Response Format for Errors 23

Figure 31: Store Request 24

Figure 32: Store Response 24

Figure 33: Catalog Request 25

Figure 34: Catalog Response 25

Figure 35: Datasource Request 25

Figure 36: Datasource Response 26

Figure 37: Schema Request 26

Figure 38: Schema Response 28

Figure 39: List Request 29

Figure 40: List Response 32

Figure 42: Response for List selected 33

Figure 41: List Seleceted Fields 33

Figure 43: Top 1 Request 34

Figure 44: Top 1 Response 35

Figure 45: Top 3 Request 36

Figure 46: Top 3 Response 36

Figure 47: General Aggregation API Syntax 37

Figure 48: Count Request 38

Figure 49: Count Response 38

Figure 50: Cound\_d Request 38

Figure 51: Count\_d Response 39

Figure 52: Sum Request 39

Figure 53: Sum Response 39

Figure 54: Avg Request 40

Figure 55: Avg Response 40

Figure 56: Min Request 41

Figure 57: Min Response 41

Figure 58: Max Request 41

Figure 59: Max Response 42

Figure 60: Filter API Syntax 42

Figure 61: Filter Request with Equal Operator 43

Figure 62: Not Equal Filter Request 43

Figure 63: Filter Request with Combined Filters 44

Figure 64: Sort Request 45

Figure 65: Paged Request 45

Figure 66: Request with Stat Option 46

Figure 67: Stat Request for Max Query 46

Figure 68: Response for Max Stat 47

Figure 69: Sample Custom Query SQL 47

# 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.

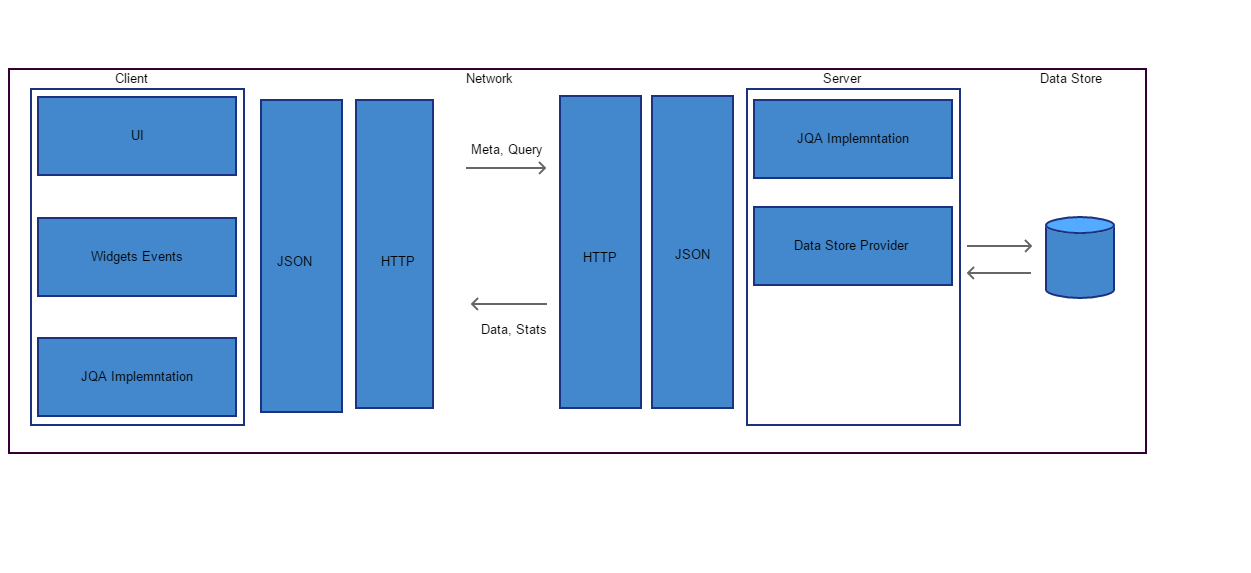


Figure 1: High-level Overview

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**

Figure 2: General Syntax for Meta API

*{*

*"meta": {*

*"<META FUNCTION>": {}*

*}*

*}*

#### STORE

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

Figure 3: General Syntax for Store Request

*{*

*"meta": {*

*"store": {}*

*}*

*}*

The response returns the data store name and version.

Figure 4: General Response for Store Request

*{*

*"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": {}*

*}*

*}*

Figure 5: General Syntax for Catalog Request

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

Figure 6: General Response for Catalog Request

*{*

*"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.

Figure 7: General Syntax for Datasource Request

*{*

*"meta": {*

*"datasource": {*

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

*}*

*}*

*}*

Figure 8: General Response for Datasource Request

*{*

*"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.

Figure 9: General Syntax for Schema Request

*{*

*"meta": {*

*"schema": {*

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

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

*}*

*}*

*}*

Figure 10: General Response for Schema Request

*{*

*"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.

Figure 11: General Syntax for Query API

*{*

*"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 columns.

Figure 12: General Syntax for Search API

*{*

*"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,

Figure : General Syntax for List All

*{*

*"query": {*

*"search": {*

*"list": {}*

*}*

*}*

*}*

Query to List set of fields

Figure : General Syntax for List Selected 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.

Figure : General Syntax for Top List Request

*{*

*"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,

Figure : General Syntax for Aggregation API

*{*

*"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.

Figure : General Syntax for Count

*{*

*"query": {*

*"aggregation": {*

*"function": "count",*

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

*}*

*}*

*}*

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

Figure : General Response for Aggregation Functions

{

"result": {

"aggregation": {

"value": "<VALUE>"

}

}

}

##### COUNT\_D

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

Figure : General Syntax for Distinct Count

*{*

*"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>"*

*}*

*}*

*}*

Figure : General Syntax for Sum

##### AVG

Average across all the values in given column.

*{*

*"query": {*

*"aggregation": {*

*"function": "avg",*

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

*}*

*}}*

Figure : General Syntax for Average

##### MIN

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

*{*

*"query": {*

*"aggregation": {*

*"function": "min",*

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

*}*

*}*

*}*

Figure : General Syntax for Minimum

##### MAX

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

*{*

*"query": {*

*"aggregation": {*

*"function": "max",*

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

*}*

*}*

*}*

Figure : General Syntax for Maximum

#### 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": {…}*

*}*

*}*

*}*

*}*

Figure : General Syntax for Filter API

#### OPTIONS

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

##### 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>"*

*}*

*}*

*}*

*}*

Figure : Sorting Syntax for Option

##### 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>"*

*}*

*}*

*}*

*}*

Figure : Paging Syntax for Option

##### 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>"*

*}*

*}*

*}*

Figure : Statistics Syntax for Option

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 >"*

*}*

*}*

*}*

Figure : General Response for Statistics

### 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>"

}

}

Figure : General Syntax for Custom API

## 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.

Figure : General Response Format for Errors

*{*

*"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

Figure : Store Request

*{*

*"meta": {*

*"store": {}*

*}*

*}*

Response

Figure : Store Response

*{*

*"result": {*

*"store": {*

*"store": "Druid","version": "0.10.1"*

*}*

*}}*

#### CATALOG

Request

Figure : Catalog Request

*{*

*"meta": {*

*"catalog": {}*

*}*

*}*

Response

Figure : Catalog Response

*{*

*"result": {*

*"catalog": ["wikiticker"]*

*}*

*}*

#### DATASOURCE

Request

Figure : Datasource Request

*{*

*"meta": {*

*"datasource": {*

*"catalog": "wikiticker"*

*}*

*}*

*}*

Response

Figure : Datasource Response

*{*

*"result": {*

*"datasource": ["wikipedia"]*

*}*

*}*

#### SCHEMA

Request

Figure : 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"*

*},*

Figure : Schema Response

*{*

*"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

Figure : 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….",*

*"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 ….",*

*"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 },*

Figure : List Response

*{*

*"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

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}} …"*

*}, {*

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

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

*"comment": "/\* Status of peremptory …"*

*}, {*

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

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

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

*}, …]}}*

Figure : Response for List selected

Figure : List Seleceted Fields

*{*

*"query": {*

*"search": {*

*"list": {*

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

*}*

*}*

*}* *,*

*"store": {*

*"catalog": "wikiticker",*

*"datasource": "wikipedia"*

*}*

*}*

##### TOP

Request to retrieve the top 1 record

Figure : Top 1 Request

*{*

*"query": {*

*"search": {*

*"list": {*

*"top": 1*

*}*

*}*

*}* *,*

*"store": {*

*"catalog": "wikiticker",*

*"datasource": "wikipedia"*

*}*

*}*

Response

Figure : Top 1 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 comments

*{*

*"query": {*

*"search": {*

*"list": {*

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

*"top": 3*

*}*

*}*

*}* *,*

*"store": {*

*"catalog": "wikiticker",*

*"datasource": "wikipedia"*

*}*

*}*

Figure : Top 3 Request

Response

Figure : Top 3 Response

*{*

*"result": {*

*"search": [{*

*"comment": "added project"*

*}, {*

*"comment": "Robot inserta {{Commonscat}} que enllaça amb ..”*

*}, {*

*"comment": "/\* Status of peremptory norms under …."*

*}]*

*}*

*}*

#### AGGREGATION

General syntax for the aggregation is below.

Figure : General Aggregation API Syntax

*{*

*"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.

Figure : Count Request

*{*

*"query": {*

*"aggregation": {*

*"function": "count",*

*"field": "time"*

*}*

*}*

*}*

Response for the COUNT

Figure : Count Response

*{*

*"result": {*

*"aggregation": {*

*"value": "39244"*

*}*

*}*

*}*

##### COUNT\_D

Request to get distinct Channels

*{*

*"query": {*

*"aggregation": {*

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

*"field": "channel"*

*}*

*}}*

Figure : Cound\_d Request

Response for distinct Channels

Figure : Count\_d Response

*{*

*"result": {*

*"aggregation": {*

*"value": "28"*

*}*

*}*

*}*

##### SUM

Request to get sum of added wikis.

Figure : Sum Request

*{*

*"query": {*

*"aggregation": {*

*"function": "sum",*

*"field": "added"*

*}*

*}*

*}*

Response for wiki addeds

*{*

*"result": {*

*"aggregation": {*

*"value": "1892"*

*}*

*}*

*}*

Figure : Sum Response

##### AVG

Request to get average wiki deletes

Figure : Avg Request

*{*

*"query": {*

*"aggregation": {*

*"function": "avg",*

*"field": "deleted"*

*}*

*}*

*}*

Response for average wiki deletes

*{*

*"result": {*

*"aggregation": {*

*"value": "512"*

*}*

*}*

*}*

Figure : Avg Response

##### MIN

Request to get minimum wiki added count

Figure : Min Request

*{*

*"query": {*

*"aggregation": {*

*"function": "min",*

*"field": "added"*

*}*

*}*

*}*

Response for minimum wiki added

*{*

*"result": {*

*"aggregation": {*

*"value": "0"*

*}*

*}*

*}*

Figure : Min Response

##### MAX

Request to get maximum wiki deletes

Figure : Max Request

*{*

*"query": {*

*"aggregation": {*

*"function": "max",*

*"field": "* *deleted"*

*}*

*}*

*}*

Response for maximum wiki deletes

Figure : Max Response

*{*

*"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.

Figure : Filter API Syntax

*{*

*"query": {*

*"filters": {*

*"filter": {*

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

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

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

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

*"filter": {…}*

*}*

*}*

*}*

*}*

Basic syntax to get anonymous edits from Wikipedia.

Figure : Filter Request with Equal Operator

{

"query": {

"filters": {

"filter": {

"field": "isAnonymous",

"operator": "=",

"filter\_value": "true"

}

}

}

}

Basic syntax to filter country name null records*.*

Figure : Not Equal Filter Request

{

"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"

}

}

}

}

}

Figure : Filter Request with Combined Filters

#### OPTIONS

##### SORT

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

Figure : Sort Request

*{*

*"query": {*

*"options": {*

*"sort": {*

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

*"order": "ASC"*

*}*

*}*

*}*

*}*

##### PAGE

Figure : Paged Request

*{*

*query": {*

*"options": {*

*"page": {*

*"page size": "10",*

*"page\_num": "1"*

*}*

*}*

*}*

*}*

##### STAT

Figure : Request with Stat Option

*{*

*"query": {*

*"options": {*

*"stat": "TRUE"*

*}*

*}*

*}*

Simple STAT request for MAX query.

Figure : 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"*

*}*

*}*

*}*

Figure : Response for Max Stat

### CUSTOM

*{*

*"custom": {*

*"type": "mssql",*

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

*}*

*}*

Figure : Sample Custom Query SQL