dtwise
turning noise into insights

DTWISE Search

The intuitive interface for analytics with Elasticsearch

Contents

Acronyms

SE: Search Engine

DSL: Domain Specific Language

SPL: Search Processing Language (Splunk's DSL)

Introduction

In 2012 Splunk seemed the perfect tool for diving into log data. It was impressive how intuitive its domain specific language (DSL) was. Using it for energy data was just another use case and the result was really good. However, its price tag and performance, with the specific data, didn't fit our needs. When Elasticsearch came out it seemed like a good alternative. In fact, it was faster due to the more structured energy data that we used (compared to log data). Back then, Elasticsearch's aggregation features were limited to the so called "facets". After version 1.0 though, it delivered mature aggregation features and nowadays it provides a full suit of capabilities, making it a well-rounded tool for analytics.

Elasticsearch is popular, it's a great search engine with extensive well-written documentation and it's open source and free. We found it was missing an interface for users running ad-hoc queries though. So, around the time Elasticsearch v1.0 was out, we created a first version of DTWISE Search and in this article, we'll go through its details. Later, Kibana came out and since then it has evolved further but it seems it's still missing the intuitiveness of Splunk's DSL. The Apache Lucene syntax used in Kibana's search prompts don't have the intuitiveness that we need, and although Elasticsearch DSL is very good, it's in JSON format and hard for users to use for ad-hoc queries.

To bridge the gap between the fast and free Elasticsearch and the intuitiveness of Splunk we designed and implemented DTWISE Search. The motive for doing so wasn't that it was a fun task for our developers to implement, but rather a necessity to provide our energy engineers with an interface to perform ad-hoc queries and explore data to the limit.

DTWISE Search

DTWISE Search is a combination of a query DSL, parser & executor (Search Engine) and a UI for displaying data (Search Sight). The Search Engine (SE) currently sits on top of Elasticsearch. Its DSL resembles that of Search Processing Language (SPL). So, SPL users would require little effort transitioning to DTWISE Search and Elasticsearch. Using the SE does not require Search Sight, which is just our default UI. In fact, we have other applications using our SE to take advantage of its enhanced capabilities and to help achieve agile delivery of new functionality and prototyping. In this article Search Sight is referenced but not explained in detail as we do for the SE. In the end of this article, we provide a code snippet to demonstrate how small a code base can become using our SE.

DTWISE uses the SE to analyze energy data mostly. However, the abilities of the SE are not limited to our domain. It can be used for analyzing logs, weather, other IoT generated data, etc.

Introduction to DTWISE Search functionality

In this section we'll go through the power of DTWISE search. We'll see how a user can do simple searches to view raw events (as stored in Elasticsearch), and we'll go into depth doing more advanced operations such as aggregations, running downstream commands and nested queries.

The SE DSL is a pipeline that reaches a desired target. A pipeline is a sequence of commands chained together, so that the output of each command feeds directly as input to the next one (downstream

command). This is much like how pipelines in Unix work and hence the SE DSL looks and feels familiar to Unix users from the first moment they start using it.

In this text we use energy data but as mentioned earlier all functionality may be applied to other domains as well.

Let's provide you with some information regarding energy data and how we treat them in DTWISE. Data is produced by energy analyzers (meters) and logged by a separate software we have developed. Meters have a counter for measuring accumulated energy and this is always incremental until it reaches the maximum number it can hold in which case it rolls-over to zero and starts counting again.

For the context of DTWISE data EActivImpTot is the energy counter's name. At its source (meter) EActivImpTot is always incrementing; it's a watt-hours counter. We continually read this counter and for each event, which may be spanning from 30 seconds to 30 minutes (we configure it by case), we enter two fields in Elasticsearch based on this:

- EActivImpTot_max: the max value read from EActivImpTot in the configured time span
- EActivImpTot_dif: the difference of EActivImpTot_max between this event's max and that of the previous (the derivative)

Of course, we log other data as well, but for this text our examples will be based around this KPI mostly.

The base query

Here is a basic query which retrieves the **10 most recent events from index called idx_pm**

```
search index=idx_pm
```

This simple query corresponds to the following Elasticsearch equivalent:

```
{
	"query": {
		"match_all": {}
	},
	"size": 10,
	"sort": {
		"timestamp": {
			"order": "desc"
		}
	}
}
```

As we'll see later our SE will save us a lot of writing as we create more complex queries.

Going back to our query, running it would return a JSON with the following output:


```
{
  "error": false,
  "result": {
    "data": [
      {
        "_index": "idx_pm",
        "_type": "doc",
        "_id": "MoJnIJCH-1GujTOqzq-3Soc5j",
        "_source": {
          "srcId": "srcdu3kIyQuy28",
          "SmplNo": "43",
          "TSpan": 450,
          "EActivImpTot_dif": 1234,
          ...
          "type": "pm",
          "timestamp": 1541071350000
      }
      ...
    ],
    "meta": {
      "depth": 0,
      "totalHits": 1212429,
      "took": 358
    }
  }
}
```


Nothing much different so far, we just got a result close to that of Elasticsearch with an indication of depth=0 which means our result corresponds to raw events.

We can browse through raw events in Search Sight as shown in figure 1 below.

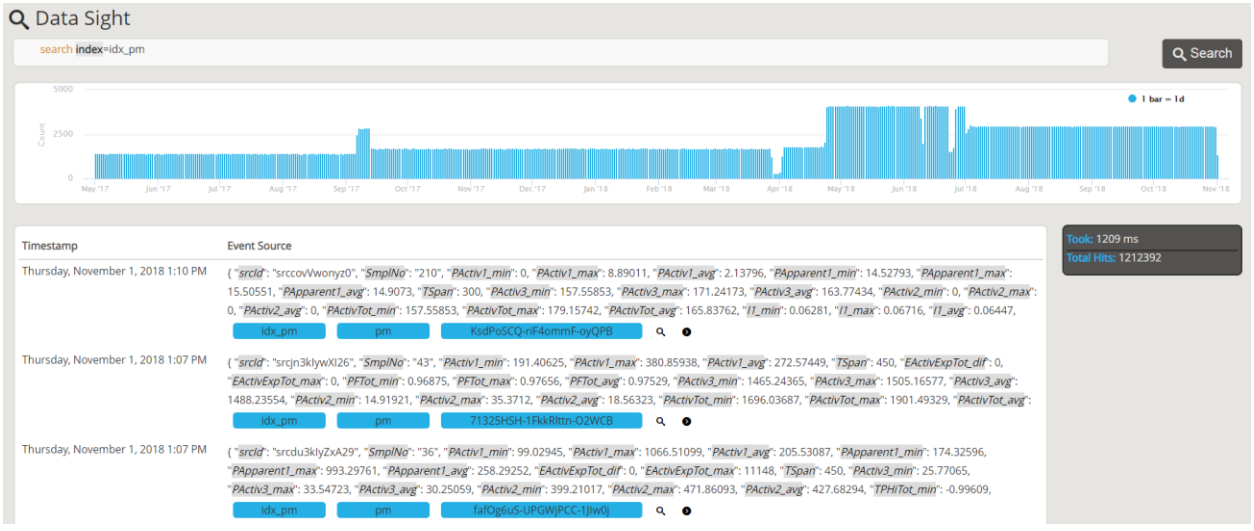

Figure 1 Raw Results

Queries with more filters

The following query retrieves the 10 most recent events from index called idx_pm **where type is pm** and where **timestamp is more recent than 01/10/2018 00:00:00.000** (inclusive)**.**

```
search index=idx_pm type=pm earliest=01/10/2018||/d
```

Next, we add to the query another parameter to **exclude events with srcId srcby3kIz7ys30**.

```
search index=idx_pm type=pm earliest=01/10/2018||/d
srcId!=srcby3kIz7ys30
```

In the following query we want to get the events where EActivImpTot_dif is between 50 and 100.

```
search index=idx_pm type=pm EActivImpTot_dif>=50
EActivImpTot_dif<100
```

A slightly less performant equivalent would be:

```
search index=idx_pm type=pm | filter EActivImpTot_dif>=50 AND
EActivImpTot_dif<100
```

You may wonder why the filter command exists then? The answer is that search works differently and if you wanted to have 2 separate ranges then you would need to have control of the expression. Therefore, if you wanted to get the events where **EActivImpTot_dif is between 50 and 100 or between 150 and 200** you would need to run the following query:

```
search index=idx_pm type=pm | filter EActivImpTot_dif>=50 AND
EActivImpTot_dif<100 OR EActivImpTot_dif>=150 AND
EActivImpTot_dif<200
```

As mentioned earlier EActivImpTot_dif displays watt-hours of energy. Assume that you wanted to see the number as kilowatt-hours (kWh) on your raw results. In this case you would evaluate a new field i.e. the division of EActivImpTot_dif by 1000, i.e. the following query:

```
search index=idx_pm | filter type=pm | eval kWh=
EActivImpTot_dif/1000
```

Your raw results would then look like so:

```
{
  "error": false,
  "result": {
    "data": [
      {
        "_index": "idx_pm",
        "_type": "doc",
        "_id": "MoJnIJCH-1GujTOqzq-3Soc5j",
        "_source": {
          ...
          "EActivImpTot_dif": 1234,
          "kWh": 1.234,
          ...
          "type": "pm",
          "timestamp": 1541071350000
      }
      ...
    ],
    "meta": {...}
  }
}
```

Aggregation Queries

Now the more fun stuff. Suppose you wanted to calculate the sum of kWh per day. A native Elasticsearch query could look like:

```
{
  "query": {
    "bool": {
      "must": [
        { "exists": { "field": "type" } },
        { "term": { "type": "pm" } }
      ]
    }
  },
  "size": 0,
  "aggregations": {
    "timestamp": {
      "date_histogram": {
        "field": "timestamp",
        "interval": "1d",
        "min_doc_count": 1,
        "time_zone": "Europe/Athens"
      },
      "aggregations": {
        "kWh_sum": {
          "sum": {
            "script": {
              "source": "doc['EActivImpTot_dif'].value/1000",
              "lang": "expression"
            }
          }
        }
      }
    }
  }
}
```

The equivalent query in our SE can be much more elegantly and intuitively expressed as follows:

```
search index=idx_pm | filter type=pm | eval
kWh=EActivImpTot_dif/1000 | calc sum(kWh) by day
```

(Ref: q1)

which is equivalent to

```
search index=idx_pm | filter type=pm | eval
kWh=EActivImpTot_dif/1000 | calc sum(kWh) by time(timestamp, 1d)
```

as you can imagine, “day” in query q1 above is an alias to “time(timestamp, 1d)”. The latter as we’ll see below allows us on the one hand to select a different date field from the configured default (in this case timestamp), have custom intervals (e.g. 2 days) and add formatting options.

Another equivalent for the previous 2 queries is the following:

```
search index=idx_pm | filter type=pm | calc
sum(EActivImpTot_dif) by day | eval
kWh=EActivImpTot_dif_sum/1000 | calc sum(kWh) by day
```

(Ref: q2)

Note how the EActivImpTot_dif changes to EActivImpTot_dif_sum on the downstream command. This happens after each calc command. It helps the user to have multiple aggregations of the same field and get an expected downstream name of that aggregated field. So, for example the following query would yield 2 fields called EActivImpTot_dif_sum and EActivImpTot_dif_avg respectively.

```
search index=idx_pm | filter type=pm | calc
sum(EActivImpTot_dif) avg(EActivImpTot_dif) by day
```

Back to query q2, you may wonder why we do the extra calc instead of ending it with the upstream eval command. The reason is that the eval command creates new fields and keeps all existing ones. Therefore, by ending the query with eval it would result in having 2 fields EActivImpTot_dif_sum and kWh.

In addition, query q2 would be equivalent to:

```
search index=idx_pm | filter type=pm | calc
sum(EActivImpTot_dif) by day | eval
kWh=EActivImpTot_dif_sum/1000 | calc max(kWh) by day
```

So, we changed the sum aggregation on the final calc with max, so why it should be the same? The reason is that the upstream calc and the downstream calc both have the same grouping, i.e. day. If the upstream calc was grouped by hour this change would obviously make a significant difference.

As all these equivalent queries would result to the same data returned. One example of displaying this would be a bar chart like the one in Figure 2 below.

Figure 2 Data Sight (Bar Chart)

We won't go into the details of Search Sight in this article, but the reader should know that we support several different ways of presenting data (pie, donut, bars, lines, scatter, heatmap, etc) and continually add more.

Suppose we didn't store EActivImpTot_dif in Elasticsearch. Then we could find the energy consumed each day using the delta function on EActivImpTot_max, for example:

```
search index=idx_pm | filter type=pm | eval
kWh=EActivImpTot_max/1000 | calc delta(kWh) by day, srcId
```

The delta function does what Elasticsearch derivative does only it handles the first day differently. The derivative would return the energy from the second day onwards. To overcome this, behind the scenes we calculate the max and min value of EActivImpTot_max of the first day and place its difference to that day. Obviously, the difference would be zero if the bucket had only one event inside. There are other ways to go around doing this, but we'll stick to this for now.

In the previous query you might have noticed that we group by day and by srcId. This is because in our domain each srcId corresponds to one meter, and therefore in order to get valid numbers we need to calculate the delta for each srcId separately.

After calculating the energy, i.e. EActivImpTot_max_delta, by day and srcId we can then calculate the total energy by day using a downstream calculation. For example, we could run the following query:

```
search index=idx_pm | filter type=pm | eval
kWh=EActivImpTot_max/1000 | calc delta(kWh) by day, srcId | calc
sum(kWh_delta) by day
```

(Ref: q3)

If data is complete one should expect queries q1 and q3 to yield the same results. This is because delta is something like the derivative of EActivImpTot_max, i.e. has a similar calculation process as the EActivImpTot_dif.

Now assume that:

- **srcby3kIz7ys30** is the srcId of the **main meter**
- **srcji3iJLqM217** is the srcId of the meter in **Basement 1**
- **srcby3iJLz8s18** is the srcId of the meter in **Basement 2**

If we wanted to **calculate the energy by month and meter** and wanted to add **human readable labels** to our data, then we could use the **rename command** to do this. For example:

```
search … srcId=srcby3kIz7ys30,srcji3iJLqM217,srcby3iJLz8s18 |
calc delta(EActivImpTot_max) by month, srcId | rename Main
Meter=srcby3kIz7ys30 of srcId, Basement 1=srcji3iJLqM217 of
srcId, Basement 2=srcby3iJLz8s18 of srcId
```

Assume now, that we wanted to calculate the total basements' energy and the remaining energy. We would have to add Basement 1 and Basement 2 to calculate Basements and subtract Basements from Main to calculate Remaining. For this we would need to do numeric calculation in a per column fashion rather the normal per row. To do this we would need to use the colcalc command so, the query would look like the following:

```
search … | colcalc Basements=srcId("Basement 1"+"Basement 2") of
EActivImpTot_max_delta Remaining=srcId("Main Meter"-Basements)
of EActivImpTot_max_delta
```

Nested Queries

Our search engine also supports nested queries. Assume that you wanted to calculate the energy consumed when the Active Power of each event is greater than its average of all events. In that case you could use nested queries to do this. For example:

```
search index=idx_pm srcId=srcji3iJLqM217 PActivTot_avg>[search index=idx_pm srcId=srcji3iJLqM217 | calc avg(PActivTot_avg)] | calc sum(EActivImpTot_dif)
```

These were the most important commands of SE, there are other commands like flatten, transpose, correlate, join, etc that are also supported, and we continue adding more.

Code snippet

Now that we've seen how our DSL simplifies a lot of tasks, we can see how we can use it for rapid development. Here is a very neat JS code snippet of an express.js callback. This code returns the total energy per specified interval for a specified year. As we saw earlier using native Elasticsearch would require writing a long json for doing this, and the specific query in the code below would be even longer than the examples above as it would require pipeline aggregations. Then we would need to format the result, implement downstream processing and so on. All this is taken care off by our SE behind the scenes using Elasticsearch as much as possible. SE formats data to Highcharts and Flot data structures for charting purposes.

```
exports.energy = async (req, res) => {

    const {interval, srcIds, date} = req.body;

    const roundings = {
        'month': 'y',
        'day': 'M',
        'hour': 'd'
    };

    if (!interval) res.json({error: true, message: 'Invalid interval.'});

    const range = date ? `${date}||/y` : '';

    const query = `search index=idx_pm type=pm srcId=${srcIds.join(',')}
                          earliest=${range}
                          latest=${range} |
                   calc delta(EActivImpTot_max) by ${interval}, srcId |
                   calc sum(EActivImpTot_max_delta) by ${interval}`;

    searchRequest(query, res);
};
```

In case you wonder what the searchRequest function does here is its code. This snippet would be commonly used by all calls to our SE (think of it as an API).

```
const searchRequest = async (query, res) => {

    try {
        const response = await axios.post(config.searchApiUrl, { query });

        if (response.data.error) res.json({ error: true, message: response.data.description });

        else {

            res.json(result);
        }
    }

    catch (error) {

        res.json({ error: true, message: error });
    }
};
```

End Note

This is the second version of our search solution which is currently in beta.

This document does not cover the full functionality of DTWISE Search. It is a quick intro to its capabilities.

If you want to make an enquiry send us an email at: dtwise.search@dtwise.com