Understanding Sumo Logic Query Language Design Patterns

sumologic.com/blog/query-language-patterns

August 21, 2018

The Sumo query language can be a source of joy and pain at times. Achieving mastery is no easy path and all who set on this path may suffer greatly until they see the light. The Log Operators Cheat Sheet is a valuable resource to learn syntax and semantics of the individual operators, but the bigger questions become “how can we tie them together” and “how can we write query language that matters?”

Fortunately, there are people who have walked this path before — and succeeded. In this post, we consolidate their queries (we’ve analyzed a 615 MB sample of query string for this) into a set of query design patterns, so that everyone can learn from their wisdom and become a Sumo Master.

Anatomy of a Query

A Sumo query decomposes in *SearchExpression* and *TargetExpression*. *SearchExpression* identifies the relevant log lines through search keywords and *TargetExpression* specifies how to slice and dice the data to provide meaning. A *TargetExpression* can have one or more clauses. Clauses combine operators and their arguments.

Typically, a *SearchExpression* contains search keywords like ‘error’ that select a relevant set of log lines from the log stream. The *TargetExpression* is then used to slice and dice the data to extract insights such as an error code. A *TargetExpression* is specified by a sequence for clauses. Clauses are separated through the pipe ‘|’ symbol. Each clause contains one operator that specifies its function and some arguments that are specific to the log lines under consideration. Finally, a *TargetExpression* may or may not end with an aggregation operator such as ‘count’ to produce a condensed view.

There are myriads of sequences of operators to retrieve information from the logs. It can be challenging to find an effective sequence. There are, however, some canonical operator sequences that are used more often than others. To find these, we’ve sampled a representative set of user queries, derived the operator sequences and counted their occurrence. This led us to the five design patterns that we’ll out line below.

Common Operator Sequences

This table shows the probability of different operators to occur. The first table shows shows the occurrence probability of individual operators. The second and the third table show the occurrence of operator tuples and triples, respectively. In the top 10 operator list `parse` takes the lead,

followed by `where` and `count.` These three already form a powerful trifecta. Learn all 10 operators and you are ready for most of the action. Moreover, we use this table to distill some common patterns.

Pattern #1: Parse Where it Counts

The Parse Where it Counts pattern leverages the parse-where-count power triple. Variants of this are the most frequent queries that we process. A search expression might outline some kind of data. Parsing filters and extracts specific values such as numbers, status codes, timestamps from the search result. The `where` clause enforces some condition such as a threshold or matches a specific string value. Finally, counting aggregates the remaining data. It can target a single number or a histogram if used with a `by` parameter.

Example 1
Input:
192.168.2.20 – – [28/Jul/2006:10:27:10 -0300] “GET /cgi-bin/try/ HTTP/1.0” 200 3395
127.0.0.1 – – [28/Jul/2006:10:22:04 -0300] “GET / HTTP/1.0” 200 2216

Query:
```
_sourceCategory=apache.log
| parse “” * “ as status
| where status=”200”
| count by status
```

Result:

status	_count
200	2

Pattern #2: Staccato Parse

The most common operator is the parse operator. This operator is used individually or in succession. We often see a parse operator followed by another parse operator and another one. Each parse clause may define another constraint to reduce the search result and/or parse out variables that are used in a downstream clause.

Example 2
Input:
192.168.2.20 – – [28/Jul/2006:10:27:10 -0300] “GET /cgi-bin/try/ HTTP/1.0” 200 3395
127.0.0.1 – – [28/Jul/2006:10:22:04 -0300] “GET / HTTP/1.0” 200 2216

Query:
```
_sourceCategory=apache.log
| parse “* – -” as ip nodrop
```

| parse “- [*]” as date
| parse “” * “ as status

Result:

date	ip	status
28/Jul/2006:10:27:10 -0300	192.168.2.20	200
28/Jul/2006:10:22:04 -0300	127.0.0.1	200

Pattern #3: Count Over Time

Counting over time never gets old. This is the go-to pattern for creating a time-series from logs. It implements a bucket sort where the buckets are regular intervals and get filled with log lines. On return, the count operator yields the number of logs per bucket and we end up with a list of <interval start, count> tuples. This time-series can easily be visualized to reveal trends or unusual behavior.

Example 3
Input:
192.168.2.20 – – [28/Jul/2006:10:27:10 -0300] “GET /cgi-bin/try/ HTTP/1.0” 200 3395
127.0.0.1 – – [28/Jul/2006:10:22:04 -0300] “GET / HTTP/1.0” 200 2216

Query:
_sourceCategory=apache.log
| timeslice 5m
| count by _timeslice

Result:

_timeslice	_count
28/Jul/2006:10:20:00	1
28/Jul/2006:10:25:00	1

Pattern #4: JSON

When handling JSON logs, the json parse command is a valuable ally. It makes the fields of the json structure accessible for downstream operators. If the log message is a valid json, the json operator can be used directly. If the json is contained in a substring of the message, it can be parsed out using a `parse` operator to extract the json string into a field and then point the json operator through its `field` parameter to this field.

Example 4

Input:

{
“timeMillis” : 1513290111664,
“thread” : “main”,
“level” : “DEBUG”,
“loggerName” : “CONSOLE_JSON_APPENDER”,
“message” : “My debug message”,
“endOfBatch” : false,
“loggerFqcn” : “org.apache.logging.log4j.spi.AbstractLogger”,
“threadId” : 1,
“threadPriority” : 5,
}

Query:

```
_sourceCategory=log4j2
| json auto keys “timeMillis”, “level”, “message”
```

Result:

timestamp	level	message
1513290111664	DEBUG	My debug message

Pattern #5: Pivot

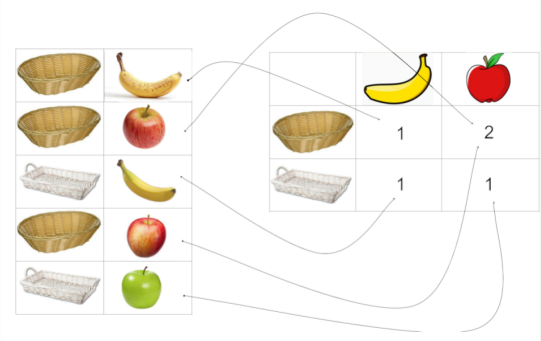
Pivot — or transpose as it is called in Sumo QL — is the “King of the Mountain.” Mastering that operator is the sign of a true Sumo Master. What it does is that it allows users to recast a list as a table. This is a powerful form of grouping and aggregating.

Usually, a group by operation transforms a list to another list with the grouping targets as indices. If there is only one index, a regular list would be the best representation.

Infobox: Pivoting and Transposing
Let's look at an example of how pivoting, or transposing in Sumo Logic, actually works. The list on the left has two columns: baskets and fruit. Now we want to find out the number of fruit in each basket. Since this list can be very long, this can become very cumbersome.

So we pivot/transpose to build a table. This table groups by baskets, which is an aggregation operation. Baskets become rows. Then we count the number of different fruit items. We have apples and bananas. Those become columns.

The counts are new information that we gained through this operation.



Note that we could have reversed rows and columns or grouped by apples with the same result. Pivoting/transposing reveals the relation between two concepts, i.e. baskets and fruit in this example.

The lists resulting from a group by typically have two or more index columns. The values in the data column come from an aggregation such as a 'count.' The index columns repeat their values so that it can become very hard to read them. The `transpose` operator allows you split that one-dimensional list and create a two-dimensional table.

The pivot/transpose operation actually splits the single large list into several smaller lists that can be put side-by-side as columns in a table. All you have to do is to let `transpose` know the split criterion, namely which indices to use as rows and columns.

Example 5
Input:
{ "method" : "GET", "status" : "200" }
{ "method" : "GET", "status" : "200" }
{ "method" : "POST", "status" : "200" }
{ "method" : "POST", "status" : "500" }
{ "method" : "POST", "status" : "500" }
{ "method" : "POST", "status" : "500" }

Query:
_sourceCategory=table
| json auto
| count by method, status
| transpose row method column status

Result:

method	500	200
GET		2
POST	3	1

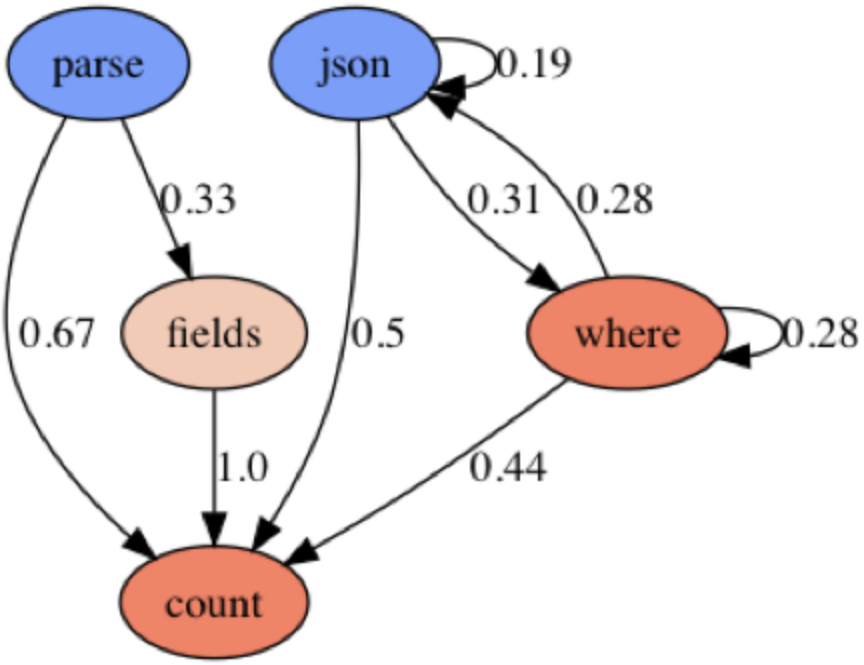
Design Pattern Case Studies

To identify common design patterns, we built a set of graphs representing the relation between operators in the Sumo query language. A graph's nodes represent operators and the edges pairwise transition probabilities between operators. Thus, an edge stores the probability that is the likelihood of an operator to be the successor of the current operator. Therefore, a design pattern is a likely sequence of operators for specific sources.

We use these graphs in the following to illustrate the application of the five design patterns on some common log types.

Kubernetes
Kubernetes log queries seem pretty straight forward in applying the *Parse Where it Counts* pattern. Sources are being parsed either with `json` or `parse` operator. In about half of the cases, the results end up directly in a count aggregator.

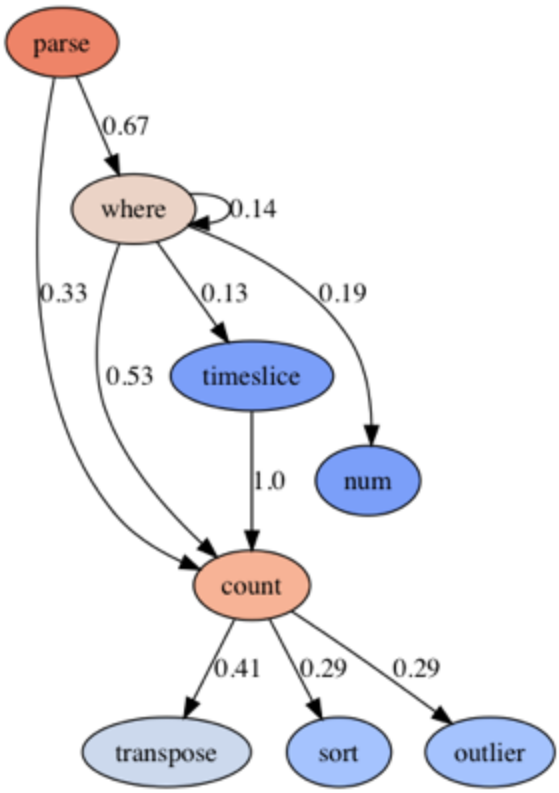
In the other half of the cases the result is reduced using the `where` or `fields` operator.



Nginx

Nginx logs are mostly processed using the *Parse Where it Counts* and the *Count over Time* patterns. Another interesting option is using `num` to cast a value to a number and then most likely using these numbers in a visualization.

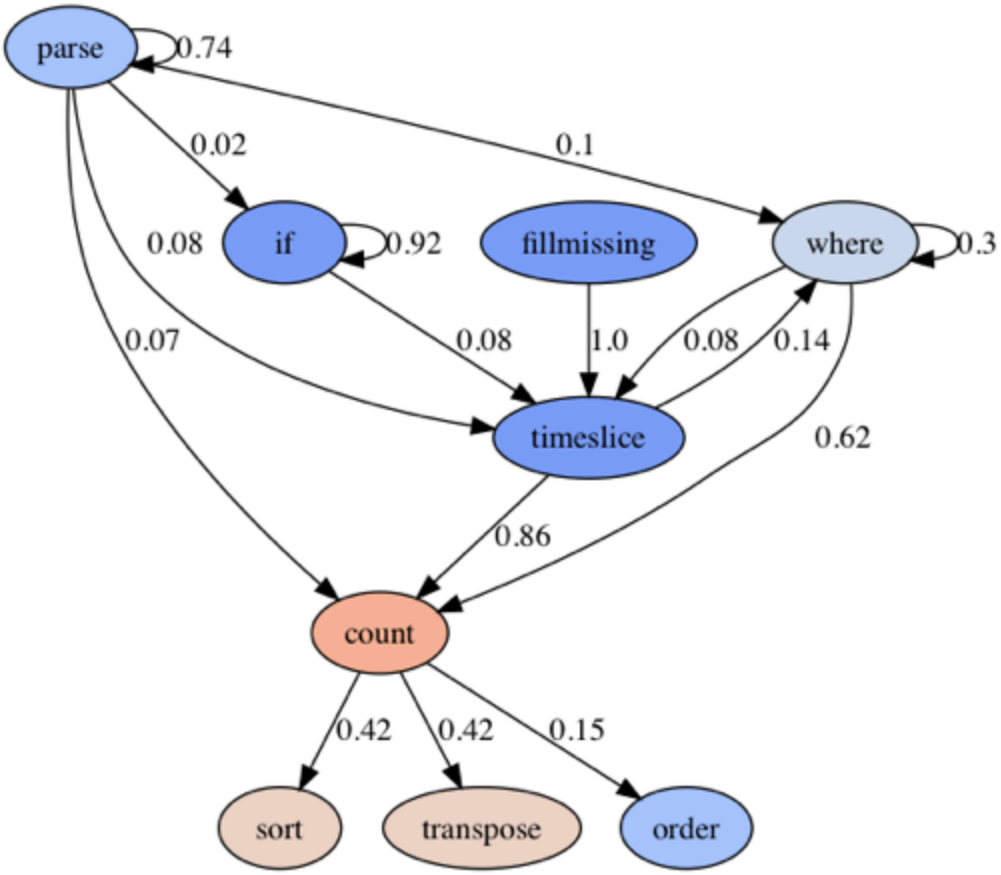
This graph suggests some interesting post-processing of counts using `sort,` `outlier` or `transpose` operators.



Apache

Apache log queries seem to implement the *Staccato Parse* pattern a lot in the first stage. The big picture in the graph is the application of the *Parse Where it Counts*and the *Count Over Time* patterns.

In the final stage, `transpose` and `sort` operators are mostly being used.



Syslog

Syslog is different from the previous logs as it seems to work a lot in the numeric domain. It starts out with a `keyvalue` parse operator. This operator applies structure on the logs and yields lots of fields. After that we observe some *Staccato Parse* patterns using `parse` and `where` to zoom in on specific KPIs. Most interesting is the use of `num` followed by `if.`

This seems to condition the query on some numeric values. Another interesting way of working with numbers is to inject the `pct` (percentiles) operator in a *Count Over Time* pattern. Having an `outlier` operator signifies that this log type is in fact mostly interpreted as a collection of numbers.

Bottom Line

Sumo query language is a rich and expressive language, but can be confusing for the Sumo Novice. This post highlights some best practice use of Sumo Logic operators. We armed the aspiring student with five design patterns based on the frequency of operators, operator pairs and triples. Looking at a set of graphs displaying the co-occurrence probabilities of operators for certain log types, we show how Sumo Masters vary and tweak them.

Armin Wasicek

Armin Wasicek is a senior software engineer at Sumo Logic working on advanced analytics. Previously, he spent many happy years as a researcher in academia and industry. His interests are machine learning, security and the internet of things. Armin holds PhD and MSc degrees from Technical University Vienna, Austria and he was a Marie Curie Fellow at University of California, Berkeley.