Building a Data-Driven Search Application Using LucidWorks SILK

Ravi Krishnamurthy
Director of Solutions

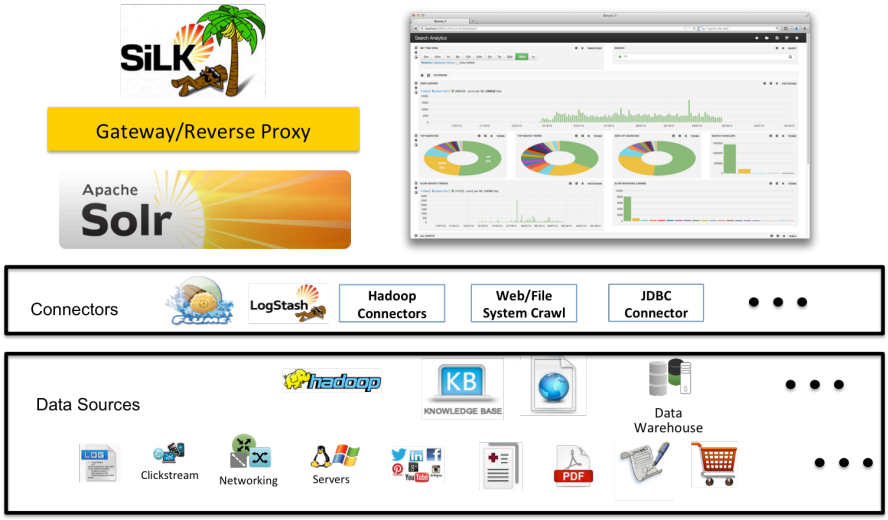
Version 1.01, published 02 May 2014.

Abstract

LucidWorks SILK (http://www.lucidworks.com/lucidworks-silk/) refers to the powerful dashboard and visualization layer that works with Apache Solr, the leading open source search platform that is *the* choice of those building data-driven applications at massive scale—companies such as Amazon, Tumblr, Pinterest, eBay, and Beats Music, among others. By using LucidWorks SILK, developers can rapidly build big data search applications on top of Solr—highlighting key, actionable insights to a broad set of users across the organization—while business users can customize and share user-friendly dashboards with minimal or no IT help.

The name SILK came from an acronym for Solr integrated with LogStash and Kibana[1]. The UI layer began by porting the popular OS visualization tool Kibana to work with Solr and has been extended from there—this OS port is available at https://github.com/LucidWorks/banana. The LogStash backend is unmodified except for a Solr Output Plug-in (https://github.com/LucidWorks/solrlogmanager). We chose to write a LogStash plug-in rather than use the in-built LogStash *solr-http* in order to: provide greater flexibility in creating fields with Solr's managed schema; and handle special characters in LogStash fieldnames that are not fully supported in Solr. This is already in open source; we also expect to make it a part of the LogStash community supported plug-ins.

While LogStash is a powerful and flexible tool for log management, it is not the only means of getting time series data into Solr. LucidWorks provides about 20 other connectors that help load data into Solr collections; any one of these collections can be used to power the SILK Visualizer, as illustrated below.



[1] LogStash and Kibana are trademarks of ElasticSearch BV

To demonstrate how to rapidly develop data-driven search applications, I am going to describe how we ingest user search logs and provide insight into web visitors use of LucidFind—a hub for information related to Solr (http://find.searchhub.org). The process we followed has the following key steps.

1. Understand your Users: What insights will bring business value along key KPI's—such as time-to-resolution, IT productivity, customer satisfaction, clicks, etc.?
2. Know your Data: How can I access the data that can be analyzed to produce these insights? How can I load them into Solr?
3. Ingest Data into Solr: Use one or more connectors to get data into Solr.
4. Build Visualizations: Make events searchable across multiple sources. Highlight key indicators that will help users identify problems & opportunities and guide response.
5. Iterate: During initial setup it is typical to go through two or three iterations with sample data in order to get the visualizations users want. Typical iterations involve making fields facet-able and sortable, and properly parsing time fields; however, we sometimes decide on more index-time pre-processing. On an ongoing basis, as users begin to get value from the application, they tend to ask for new insights that can further improve the business.

My colleagues and I used a similar basic approach when building dashboards for syslogs, weblogs, firewall logs, social media, financial data, etc. By bringing together these logs, we provide a 360-degree view of your customers and the systems that you are using to serve them.

LucidWorks

Table of Contents

What Insights Are Users Looking For?

The search analytics application is designed to automate and enhance the health checks we do as part of our customer success program, *LucidWorks 4 You* (http://www.lucidworks.com/support-services/lucenesolr-support/). In this process, we primarily look for the following metrics.

1. What are people searching for on our e-commerce site, knowledge portal or other search application?
2. Which queries are returning zero hits?
3. Which searches are providing slow response times?
4. Is there a trend in my slow searches that might indicate the need to add hardware or tune my application?
5. What is my system performance—memory/cpu/disk usage, jvm metrics, etc.?
6. Is the cache warm-up time very large? Am I committing too often?

The first 3 metrics are primarily of interest to the business users and content creators, while search administrators and IT are likely interested in all 6 parameters.

Where can I find the Underlying Data?

While using our commercial LucidWorks Search product, the data is available from the following sources:

1. Core Logs
2. Core Request Logs
3. Connector Logs
4. Mbeans API
5. Jmx and Log4j

Solr Schema

While we can use Solr's managed schema in order to automatically detect field types, I generally find it simpler to pre-configure fields that I know I am extracting using my grok patterns. This will ensure that fields that we facet on are stored as "string" rather than "text_en" and that single-valued fields are stored as such for sorting. I can also setup *copyfield's*, such as *queryterms* and *userquery* below, so that I can search on individual tokens within a "text_en" field while also have a "string" version available for display, faceting and sorting. The managed schema (or setting up a dynamic field for the pattern "*") remains useful for detecting new fields that we did not anticipate and configure during application setup.

Below, I have provided some of the key fields from the Solr schema.xml, corresponding to fields created/modified through LogStash's grok and date filters (See Section 4).

```
<field name="userquery"  indexed="true" multiValued="false" omitNorms="false"
omitPositions="true" omitTermFreqAndPositions="true" stored="true"
termVectors="false" type="string"/>

<field name="queryterms"  indexed="true" multiValued="true" stored="true"
type="text_en" termVectors="false" omitNorms="false" omitPositions="false"
omitTermFreqAndPositions="false" termVectors="false" />

<copyField dest="userquery" source="queryterms"/>

  <field name="hits" indexed="true" multiValued="false" omitNorms="true"
omitPositions="true" omitTermFreqAndPositions="true" stored="true"
termVectors="false" type="int"/>

 <field name="qtime"  indexed="true" multiValued="false" omitNorms="true"
omitPositions="true" omitTermFreqAndPositions="true" stored="true"
termVectors="false" type="int"/>

<field name="searchhandler"  indexed="true" multiValued="false" omitNorms="false"
omitPositions="true" omitTermFreqAndPositions="true" stored="true"
termVectors="false" type="text_en"/>

<field name="event_timestamp"  indexed="true" multiValued="false" omitNorms="false"
omitPositions="true" omitTermFreqAndPositions="true" stored="true"
termVectors="false" type="tdate"/>

 <field name="status"  indexed="true" multiValued="false" omitNorms="true"
omitPositions="true" omitTermFreqAndPositions="true" stored="true"
termVectors="false" type="int"/>
```

The full configuration file (along with the other configuration files and dashboards referred to in this document) is provided at http://github.com/LucidWorks/silkusecases.

Data Ingestion

Since I was interested in analyzing data over the past 6 months, I focused on the data in the log files. Also, in order to keep the size of this article manageable, I am only going to describe how I extract and visualize the first 4 metrics listed in Section 2—metrics that are related to user queries, quality of response and responsiveness of the website.

I chose LogStash for data transformation and import for two reasons:

1. It provides a powerful framework for extracting, grokking and transforming log data into a structured format that Solr can consume and that SILK can use for dashboards.
2. LucidWorks' Hadoop Connectors have a GrokIngestMapper that allows me to reuse the same LogStash Filters to work with larger volumes of files on HDFS (more details on this in a future article).

LogStash's pipeline consists of three stages: inputs, filters and outputs. Inputs generate events, filters modify them, and outputs ship them elsewhere. More information about the lifetime of an event and documentation on a number of built-in plug-ins is available at http://logstash.net/docs/1.4.0.

Below, I have provided the LogStash configuration—with inline comments explaining some key elements—for all three stages.

Input

```
input {
        file {
                #The type field is used to differentiate different types of logs that are
stored in a collection.
                type => "lwscorelog"
                #Provide a full path. For this example, I have set it up to parse core logs
        and exclude core request logs, which have different information and formats.
                path => [
"/Users/ravikrishnamurthy/Documents/src/demo_log_generator/samplelogs/lucidfind/cor
e-logs/core*" ]
                exclude => ["*.gz","*.zip","*.tgz", "*request*"]
                sincedb_path => "LWSquerylog_lucidfindlogs.sincedb"
                start_position => "beginning"
        }
}
```

Filter[2]

```
filter {
        if [type] == "lwscorelog" {
                grep {
                        ignore_case => true
                        negate => true
                        # LucidWorksLogs is a system collection and we want to exclude
                it from our metrics
                        match => ["message", "\[LucidWorksLogs\]"]
                }
                grok {
                        # Groks the Queries logged by LucidWorks Search
                        match => ["message", "%{TIMESTAMP_ISO8601:received_at}
INFO %{DATA} \[%{DATA:collection}\] webapp=%{DATA:webapp}
path=%{DATA:searchhandler} params={%{DATA}q=%{DATA:queryterms}[&}]%{DATA}
```

[2] The TIMESTAMP_ISO8601 pattern is available in LogStash (https://github.com/elasticsearch/logstash/blob/master/patterns/grok-patterns). It is easy to add additional patterns to grok other application-specific log formats. For a full explanation of how LogStash grok works, please refer to http://logstash.net/docs/1.4.0/filters/grok.

```
hits=%{BASE10NUM:hits} status=%{BASE10NUM:status}
QTime=%{BASE10NUM:qtime}"]
                }
                if ("_grokparsefailure" in [tags]) {
                        #for this exercise we are only interested in user queries. To detect
cache warm up times we will have to build other match filters. So we drop all events that
do not match above
                        drop{}
                }

                date {
                # Pull the time stamp from the 'received_at' field (parsed above with grok)
                        match => [ "received_at", "yyyy-MM-dd HH:mm:ss,SSS" ]
                }

        }
}
```

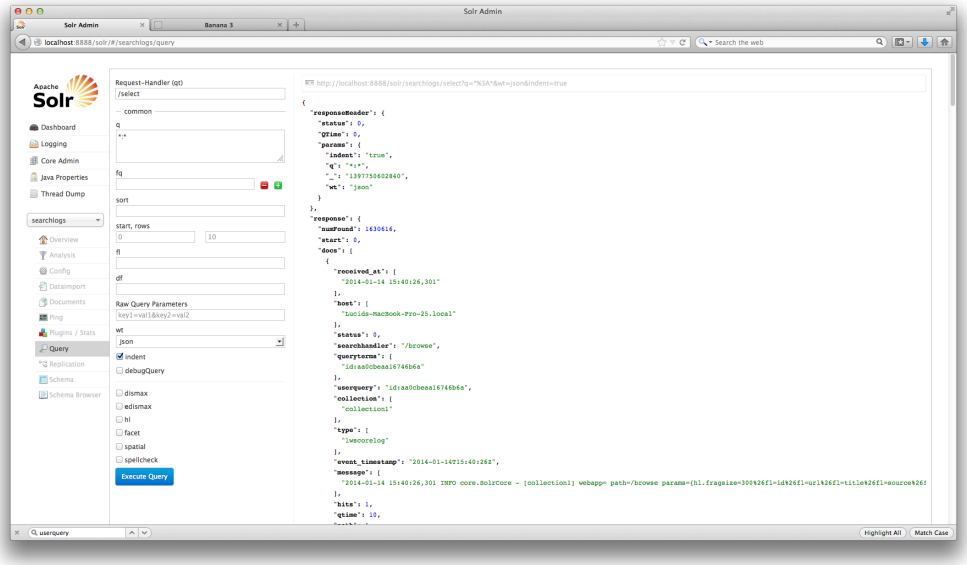
Output

```
 output {
        #stdout { debug => true codec => "rubydebug"}
        lucidworks_solr_lsv133 {
                collection_host => "localhost"
                collection_port => "8888"
                collection_name => "searchlogs"
                # @timestamp and @version replaced by event_timestamp and
                event_version before sending to Solr
                field_prefix => "event_"
                #the output plug-in does no commits, we rely on Solr's auto-commit policy
                force_commit => false
                flush_size => 1000
                idle_flush_time => 1
        }
}
```

We store the configuration in silk_lws.conf, and then run LogStash from within the SILK package—available on the LucidWorks Website—(http://www.lucidworks.com/lucidworks-silk/), using the specific steps described in the included documentation. If you download it separately from github, the process will be similar.

```
cd $SILK_HOME/solrWriterForLogStash/logstash_deploy
java -jar logstash-1.3.3-flatjar.jar agent -p . -f silk_lws.conf
```

By going to the Solr Admin, we can now see records being indexed into the specified collection.

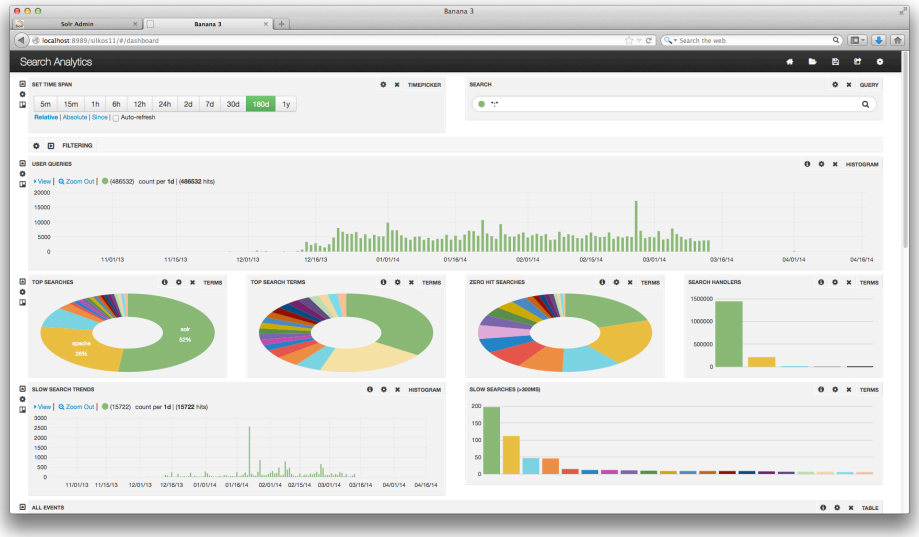


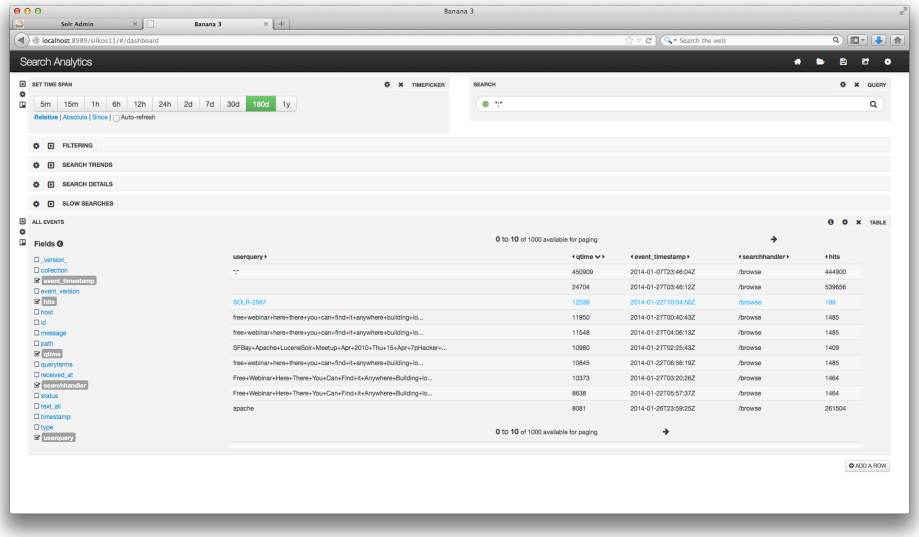
NOTE: The above LogStash filter configuration works with the logs generated by LucidWorks' commercial product. For, Solr logs, I used a slightly modified grok filter.

```
filter {
        if [type] == "solrlog" {

                grok {
                        match => ["message", "INFO %{DATA}
%{TIMESTAMP_ISO8601:received_at}; %{DATA}; \[%{DATA:collection}\]
webapp=%{DATA:webapp} path=%{DATA:searchhandler}
params={%{DATA}q=%{DATA:queryterms}[&}]%{DATA} hits=%{BASE10NUM:hits}
status=%{BASE10NUM:status} QTime=%{BASE10NUM:qtime}"]
                }

                if ("_grokparsefailure" in [tags]) {
                drop{}
        }

                date {
                # Try to pull the time stamp from the 'received_at' field (parsed above with
grok)
                        match => [ "received_at", "yyyy-MM-dd HH:mm:ss.SSS" ]
                }
        }
}
```

Building Dashboards

With this data in place, we can now turn to building dashboards. I iteratively added panels, eliciting feedback from users until I settled on this final dashboard. I started from the default syslog dashboard and rapidly transformed it into the dashboard shown below. The first screenshot shows the first 5 rows, while the second one shows the table panel in the 6th row—easily visible after hiding other rows.





Since dashboards are interactive and visual, I have demonstrate various aspects of their configuration on an accompanying LucidWorks webinar (http://programs.lucidworks.com/SiLK-introduction_Register.html). In this article, I will summarize and highlight some key aspects of building this application.

I began dashboard configuration by setting the Solr endpoint and collection and then decided on how many rows I want and how to size them. I decided on six rows that are described below.

1. **Options:** I used the first row for user input, providing a time-picker and a query panel. The first 2 rows are similar to the default syslog dashboard.[3]
2. **Filtering:** Row 2 contains the panel for Global Filters—Solr fq's that apply to the entire dashboard. I left it hidden by default. The initial time window populates it by default.
3. **Search Trends:** In the next row, I had a *histogram* panel spanning the entire row. This shows how many user searches are made in selected time range and time windows. I chose to use bars for display—lines and points are also available
4. **Search Details:** Here, I showed various facets using *terms* panels. Reviewing my user requirements, I showed top queries and top query terms (one faceting on the entire queries while the other faceting on individual tokens in a query). I also used the panel-specific advanced query parameters to create a panel that only shows zero-hit queries; the panel specific query parameter I added was *&fq=hits:0*. I chose either a pie chart or bar chart depending on which provided a better visual insight into the data.
5. **Slow Searches:** Shows trends and specifics about which searches are slower than 300ms, both using a histogram and a terms panel. Here the panel-specific advanced query parameter took the form *&fq:qtime:[300 TO *]*.
6. **Details:** A table view which shows individual events and which can sorted on any single-valued field. Very useful when I want to filter individual events.

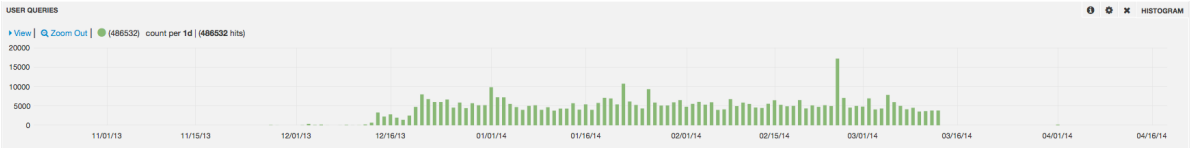
I also added some tweaks that were specific to this particular web application. The web application was making a number of system queries using the /select search handler (queries took the form *q=datasource:<guid>*); I filtered this out using panel-specific filter queries so as to focus on user queries.

Actionable Insights

In this section, I present some insights that became readily apparent from the visualizations.

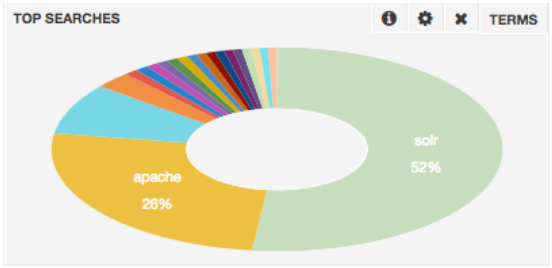
[3] All dashboards require a time-picker and a filtering panel. Almost every application will have a query panel. In newer versions of SILK, the query panel can be configured with different search default fields, query parsers and search handlers. I modified the time picker configuration, adding 180 days and 1 year to the choices available in relative time mode, in order to make it convenient for users to explore the data.

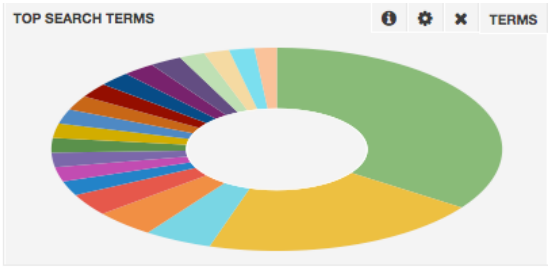
Search Trends



By looking at count of user queries (above), we can see that there was a distinct increase in mid-December 2013 that has been sustained over the following 3 months. Understanding the causes—such as marketing and application changes—enables us to tune our efforts and better serve LucidFind users.

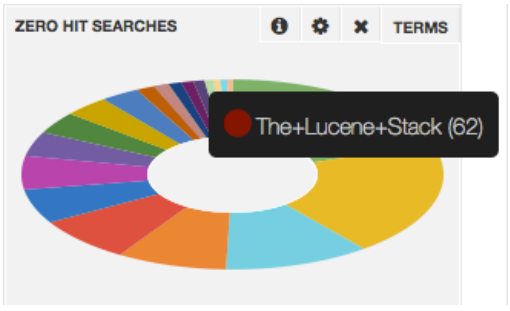
Search Terms

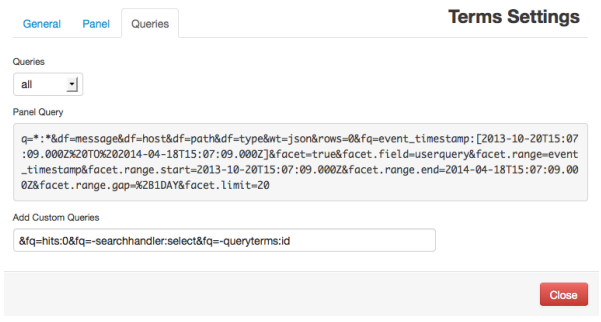




The two panels are similar: One shows complete search phrases while the other shows individual tokens. The terms “apache” and “solr” come through checkboxes and are not typed by the user. If we move towards the longer tail, we see that “join,” “import” and “faceting” are very popular queries, indicating what Solr users are most concerned about. This insight helps us add relevant content and also provides input to the product roadmap.

Zero Hit Searches

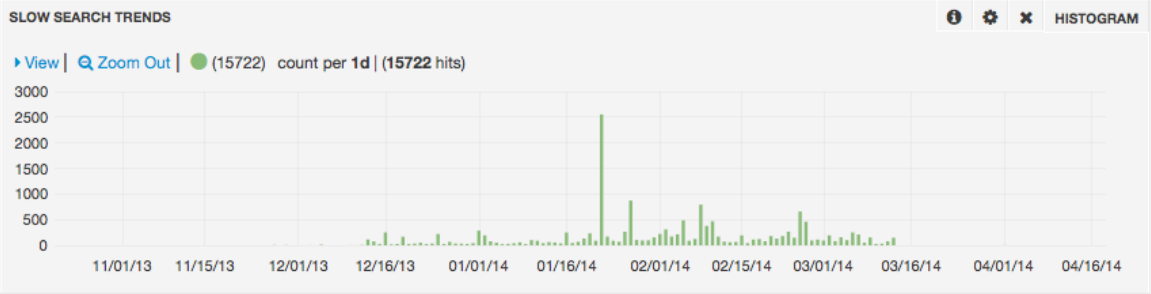




This panel shows us the most popular search phrases that return zero results. It indicates where we are failing to meet user needs and how we can address them (by adding content and/or modifying the search handler/query parser).

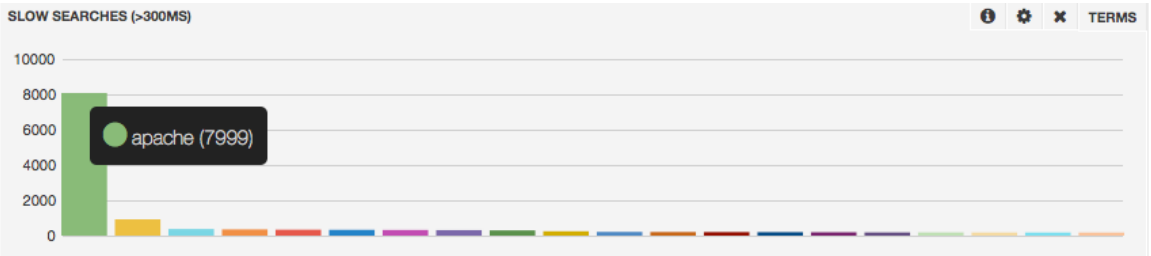
I have also shown the query parameters used to generate this panel. The *&fq=hits:0* provides the searches which returned no results while the *&fq=-searchhandler:select&fq=-queryterms:id* is used to filter out system-generated queries that I described earlier.

Slow Searches

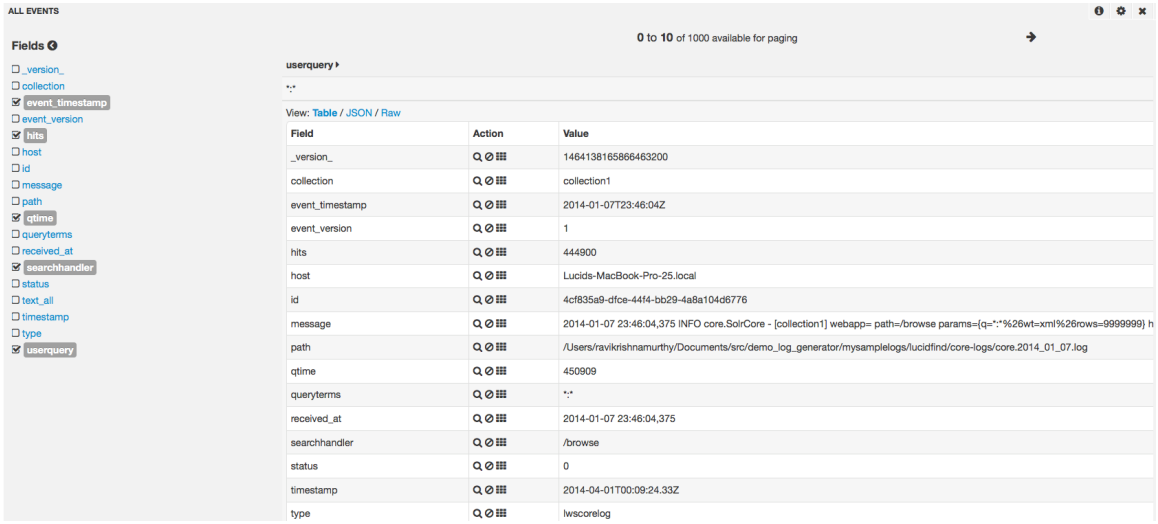


Trends of slow searches tell us about how user experience may be changing over time, and whether we may need to add new hardware or modify the application (by blocking certain kinds of queries, for example). In this case, we see a big spike on 21 January 2014. We can then correlate it to number of user searches to see if this spike was a result of increased volume.

Interestingly, in our case, while 21 January 2014 showed a high query volume, there were much busier days that did not show a similar degradation in performance. This indicates that the issue is likely related to the *types of queries* in addition to the *volume of queries.*



The next panel shows (ordered by count) the specific query terms that are resulting in slow performance, providing an important indicator of user experience.

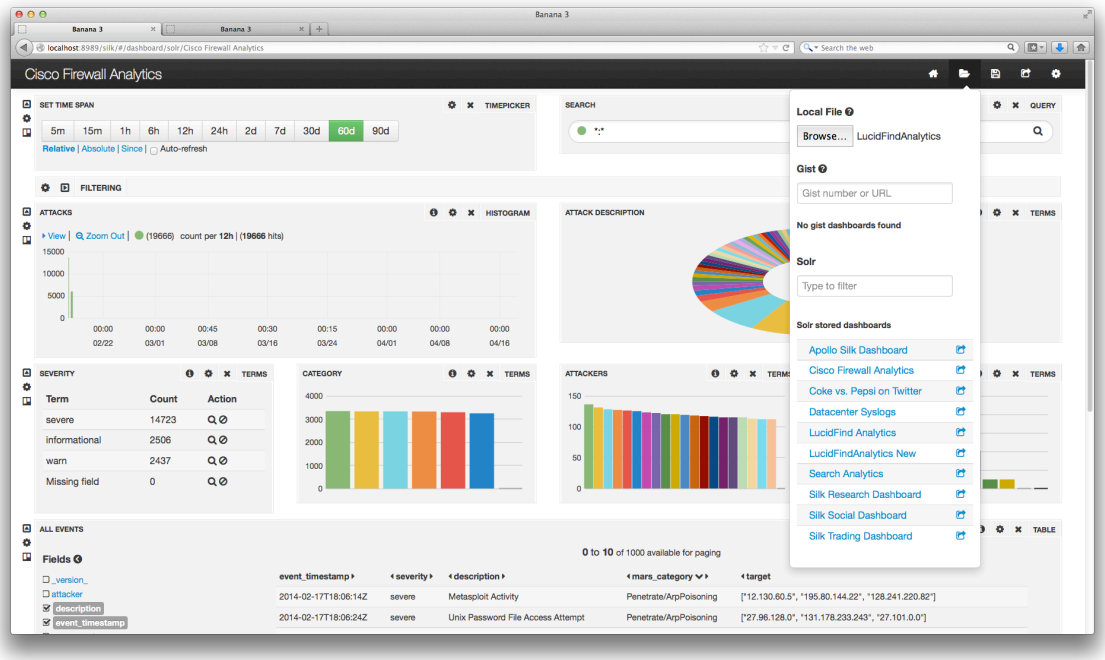


Finally, we drill into specific slow queries (by fetching the results by descending order of *qtime* and zooming into specific time periods of interest). This particular query took 45 seconds—the reason is likely because it requests 9999999 rows. By understanding why such a query is issued by the application—and blocking/modifying it—we can significantly improve user experience.

Summary

I have presented how Apache Solr and LucidWorks SILK can be used to derive value from one type of application logs. Admittedly, these insights can be derived (with significantly more effort) using a combination of *grep, sed and awk* (or other scripting tools), but that approach is not viable if we need results in near real-time and on an ongoing basis. Further, insights of interest to business users are not readily available to them, and much is lost in translation.

The situation become more complex when we want to get a more holistic view of our customers and our systems. LucidWorks SILK can be used to ingest and visualize a variety of time series data—such as syslogs, weblogs, clicks and conversion data, firewall logs, activity records from CRM systems, call center transactions, social media streams, database logs, application logs, etc.—and create multiple dashboards, thus providing a near real-time, 360-degree view of your customers and operations.



Resources

1. LucidWorks SILK: http://www.lucidworks.com/lucidworks-silk/. Individual components are available at:
    a. Apache Solr: http://lucene.apache.org/solr/.
    b. Visualizer: https://github.com/LucidWorks/banana
    c. Solr Output Plug-in for LogStash: https://github.com/LucidWorks/solrlogmanager
2. LucidWorks Search: http://www.lucidworks.com/lucidworks-search/
3. Webinar of LucidWorks SILK: http://programs.lucidworks.com/SiLK-introduction_Register.html.
4. LogStash: http://logstash.net/
5. SILK Use Cases: https://github.com/LucidWorks/silkusecases. Provides all the configuration files, schemas and dashboards required to build the application.