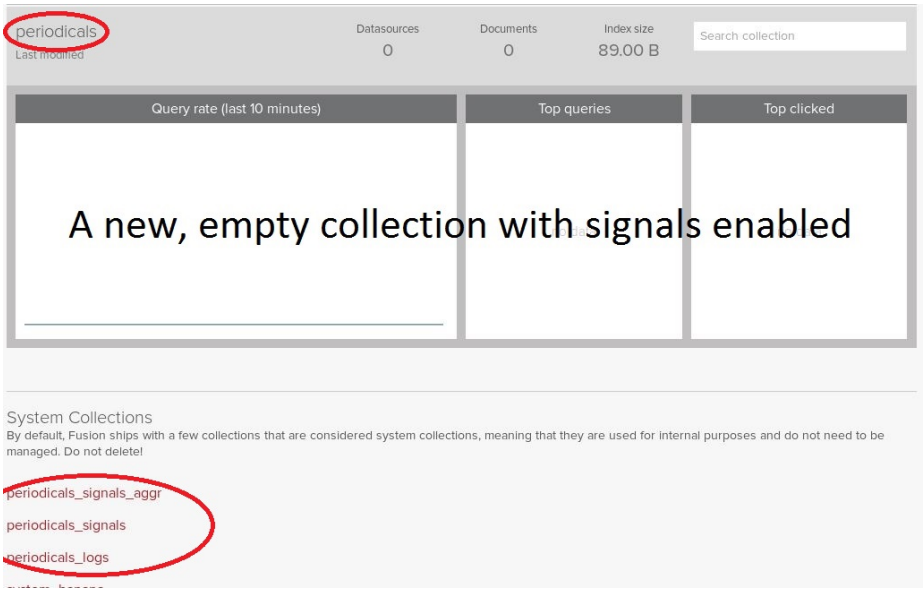
Lucidworks

Mixed Signals – Using Fusion's Signals API

One of my favorite features in Fusion is the Signals API – a RESTful, flexible, easily implemented mechanism for capturing interesting user events, like (but not limited to) queries and result clicks, and their associated metadata. When signals are enabled for a collection (*the default behavior – unless the collection is created using the REST API*), Fusion automatically creates a corresponding "signals" collection in which raw signal data submitted to the Signals API is indexed. In fact, if you've been working with Fusion you may have noticed several collections that appear when a new collection is created. This signals collection is named after the original collection with "_signals" suffixed; for example, given a signals-enabled collection called "periodicals", Fusion will automatically create a collection called "periodicals_signals." All Signals API calls are made referencing the original collection, as the API manages the signals collection. Thus any Signals API calls should be made to the "periodicals" collection, and they will automatically be indexed into "periodicals_signals."

Another collection you may have noticed is the aggregated signals collection, named after the signals collection with "_aggr" suffixed; for the previous example, the aggregation collection would be called "periodicals_signals_aggr." Aggregation is the automated method by which Fusion ingests the raw signal data and processes it into summaries, metrics and other key performance indicators which are indexed and stored in the aforementioned collection. This collection can then be visualized through the Fusion Dashboard or queried for Recommendations.



Those of you who worked with Lucidworks Search, the previous platform release, may remember the "Click Scoring Relevance Framework," usually just referred to as "Click Scoring" – a straightforward API that captured a limited set of data to a log file that LWS could then analyze on a scheduled basis. Fusion's signals are a big step up in many ways:

- As mentioned, raw and aggregated signals are captured in collections, with all the advantages that offers
- In addition to a few predefined parameters (like event type and timestamp), the API can accept any number of user-defined parameters
- Raw signal data can be visualized in near real time in a variety of formats

There are several other advantages in the newest release, but these relate most directly to the first step in analysis, visualization and recommendations: capturing events using the Signals API.

Sending Signals

As mentioned earlier, the Signals API is a RESTful interface. It accepts POSTs where the request body is JSON consisting of a few predefined parameters and any number of user-defined parameters, intended to capture interesting events occurring from a user's interaction with search. The first step to constructing the request is to determine which

of the full list of input parameters should be explicitly assigned, and what, if any, user-defined parameters should be captured. Of the predefined parameters, there is only one required parameter, "type" (other essential parameters like "id" and "timestamp" are optional; if not present, the API will create and assign default values to them); "type" can be any user-defined string, but should be applied consistently to ensure accurate aggregation. Two of the more interesting interactions a user has with search are the queries they submit (type="query") and the results they select (type="click"), so we'll look at API request examples of each. In all examples, the API will default timestamp and id, and there are a couple custom parameters defined (depending on the event type.) The Signals API accepts request parameters in JSON format, so we'll first define the structure and contents of our JSON events, then demonstrate how to post them to the REST API.

Query Events

This is an event of type "query" that captures a user's query string to help us understand how users are searching. It also capture the user's login ID and the total results for the query string.

Here is an example JSON string representing a query event, capturing those fields:

```
[{params: {query: 'mobile phone reviews', user: 'smare', numresults:
'18729'}, type: 'query'}]
```

This string can be constructed through any mechanism at your disposal – here's an example of constructing this event in Ruby (from the catalog_controller component of a Blacklight-enabled Ruby on Rails application):

```
req.body = [{params: {query: params[:q], user: current_user.login,
numresults: @response.total}, type: 'query'}].to_json
```

Note the use of the "params" property to define custom fields (*user*, *numresults*); any number of additional fields can be added here to capture useful information for later analysis.

Click Events

Here we'll define an event of type "click," which will be triggered whenever a user clicks on a search result. Our intention is to determine which results are actually examined by users for a given query, so in addition to capturing the user and their query, we'll also capture the ID of the result document the user clicked and that document's score. Here's an example of a JSON string capturing this information:

```
[{params: {query: 'galaxy s4 review', user: 'smare', score:
'34.2675357', docId: '8967562'}, type: 'click'}]
```

Here's an example of constructing that JSON in Ruby on Rails:

```
req.body = [{params: {query: params[:query], user: current_user.login,
score: params[:score], docId: params[:docId]}, type: 'click'}].to_json
```

POSTing the Request

Now that the event data is captured and formatted as JSON, it can be posted to the Signals REST API. The path for the request is `<serverName>:8764/api/apollo/signals/<collectionName>`, where <serverName> is the IP address or machine name of the Fusion API server, and `<collectionName>` is the name of the collection for which you want to capture signals. Using the "periodicals" collection, the URL would be `http://fusionserver.local:8764/api/apollo/signals/periodicals`.

The Signals API accepts a few request parameters:

- `commit`, a flag that when set to `true`, instructs Fusion to issue a commit at the end of indexing
- `async`, a flag that when set to `true`, instructs Fusion to index signals asynchronously (issuing autoCommit and suppressing failure reporting.) When omitted (as in this example) this flag defaults to `false`.
- `pipeline`, which can be used to define a specific index pipeline to use when indexing the signals. If not provided (as in this case, since we have no advanced requirements for ingesting signals) Fusion will use the pre-configured "signals_ingest" pipeline by default

Beside the request parameters, the request header "`Content-Type`" should be set to "`application/json`" as JSON is the format of the request body. Furthermore, Fusion requires API clients to authenticate – the request header must be configured for basic authentication with a valid username and password.
Therefore, to send the request with the appropriate headers, parameters, and bodies defined to the example "periodicals" collection, we could execute the following curl command:

```
curl -u admin:xxxxxxxxxx -X POST -H 'Content-type:application/json' -d
'[{params: {query: 'mobile phone reviews', user: 'smare', numresults:
'18729'}, type: 'query'},{params: {query: 'galaxy s4 review', user:
'smare', score: '34.2675357', docId: '8967562'}, type: 'click'}]'
http://fusionserver.local:8764/api/apollo/signals/periodicals?commit=tru
e
```

While this is useful for illustrating all the requirements for a well-formatted API request, the likelihood of capturing signals using `curl` in a real-world scenario is lower than through other mechanisms. Most likely, the events will be indexed in real-time rather than by a batch process, which introduces some complexities depending on the type of event captured and how the event is submitted to the Signals API. The examples thus far illustrate signal data capture from a live application implemented in Ruby on Rails and using the Blacklight gem for Solr integration, and we'll continue to explore signals in that context – but these examples could easily be ported to another implementation framework.
The "query" signal is posted from a server-side component extended from Blacklight that acts as the interface to Solr; the Signals API request is made after receiving the result list of documents from Solr (but before rendering the results.) This component is a Ruby class, so plain old Ruby code is used to create and execute the request.

```
# Note that the following modules are required
# require "uri"
# require "net/http"

api_uri =
'http://fusionserver.local:8764/api/apollo/signals/periodicals?commit=tr
ue'
uri = URI(api_uri)
```

```
        req = Net::HTTP::Post.new(api_uri, initheader = {'Content-Type'
        =>'application/json'})
        req.body = [{params: {query: params[:q], user: current_user.login,
numresults: @response.total}, type: 'query'}].to_json

        req.basic_auth('admin', 'xxxxxxxxxxxx')

        res = Net::HTTP.start(uri.hostname, uri.port) do |http|
              http.request(req)
        end
```

The "click" signal is captured differently. Since the click signal is defined as the event occurring when a user selects a document from a list of search results, it would make sense to embed this API request in the UI rather than on the server. The most obvious approach is calling a JavaScript method on the result (perhaps implemented as the `onclick()` method of a document title in the results list) that posts the Signals API request. AngularJS is a popular web application framework providing extensive client-side functionality that lends itself nicely to this kind of implementation.

```
        angular.module('signalsProcessor', [])
            .controller('SignalsController', ['$scope','$http', function($scope,
        $http) {
                $scope.sendClickSignal = function(docId,query,score,userId) {
                    var data = [];
                    var signal = {"params": {"query":query, "user":userId,
        "docId":docId, "score":score}, "type":"click"};
                    console.log(signal);
                    data.push(signal);
                    var url =
        'http://fusionserver.local:8764/api/apollo/signals/periodicals?commit=tr
        ue';
                    return $http.post(url,data)
                        .success(function(response) {
                            var msg = 'Successfully indexed click data';
                            console.log(msg);
                            $scope.notification = true;
                            $scope.notificationMsg = msg;
                        });
                };
            }])
            .config(['$httpProvider', function($httpProvider) {
                $httpProvider.defaults.useXDomain = true;
                $httpProvider.defaults.withCredentials = true;
```

```
            $httpProvider.defaults.headers.common["Content-type"] =
        "application/json";
            $httpProvider.defaults.headers.common["Authentication"] = "Basic
        admin:xxxxxxxx";
        }]);
```

A link to this method can then be embedded in each result.

```
<div ng-app="signalsProcessor" ng-controller="SignalsController">
      <button ng-click="sendClickSignal('<%= document[:id] %>','<%= params[:q]
%>','<%= document[:score] %>','<%= @current_user.login %>')" class="search-
btn" id="sgnl_<%= document[:id] %>">
            <span class="glyphicon glyphicon-search"></span>
      </button>
</div>
```

This approach works as long as the client is on the same machine as the Fusion API (meaning for the above example, the application submitting the signals requests is deployed on the same machine as Fusion.) However, in a typical production (and test, and most likely development) environment, these applications would be deployed on different machines. Clicking a search result to trigger `sendClickSignal` in such an environment results in an error:

```
XMLHttpRequest cannot load
http://fusionserver.local:8764/api/apollo/signals/periodicals?commit=tru
e. No 'Access-Control-Allow-Origin' header is present on the requested
resource. Origin 'http://192.168.1.175:3000' is therefore not allowed
access. The response had HTTP status code 401.
```

For the unfamiliar, this message is describing a CORS error. Designed to protect against malicious code execution, the Signals API restricts HTTP requests from another domain outside the domain from which the resource originated (hence the server-side example's success.) There are a number of ways to deal with this, and nearly all ultimately implement a sort of proxy that resides in the same originating domain, accepting requests from clients and forwarding them to the server (one alternative that isn't a proxy would be JSONP, which unfortunately does not support basic authentication.) For many enterprise deployments a dedicated, running CORS proxy service is not a feasible solution; but a simple server-side component in your application can act like a proxy with a minimum of effort. Post to the component from a JavaScript

function that passes all the necessary fields, and from there the component creates and posts the Signals API request.
The following JavaScript function uses the jQuery.ajax() API to asynchronously post to a Ruby on Rails controller (but could easily post to any server-side component listening for requests.)

```
function postClickEvent(docId, query, score) {
	$.ajax({
		url: "/signals?docId=" + docId + "&query=" + query +
"&score=" + score,
		type: "post",
		success: function () {
			console.log("Posted click tracking data to signals
controller");
		},
		error: function (msg) {
			alert('Error Marking Relevance');
			console.log("Unable to post to signals controller: "
+ msg);
		}
	});
}
```

The controller then uses the same mechanism as the "query" signal request to post to the Signals API.

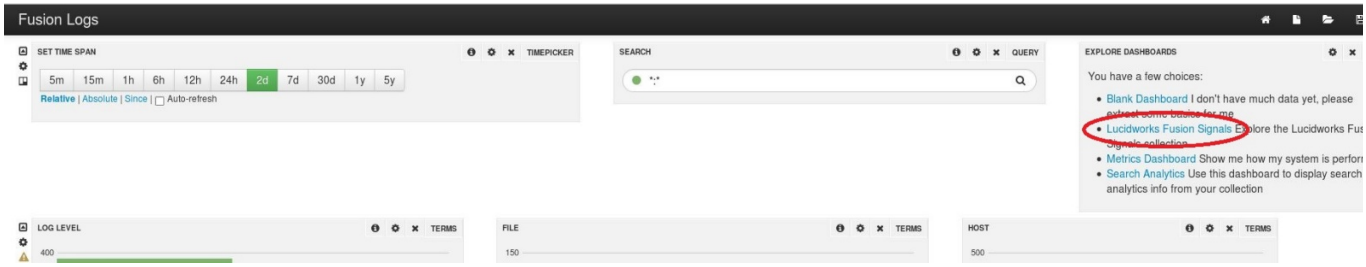
```
require "uri"
require "net/http"
class SignalController < ApplicationController

	def new
		render :layout => false
	end

	def create
		api_uri =
'http://fusionserver.local:8764/api/apollo/signals/periodicals?commit=tr
ue'
		uri = URI(api_uri)
		req = Net::HTTP::Post.new(api_uri, initheader = {'Content-
Type' =>'application/json'})
		req.body = [{params: {query: params[:query], user:
current_user.login, score: params[:score], docId: params[:docId]}, type:
'click'}].to_json
```

```
                req.basic_auth('admin', 'xxxxxxxxxxxx')

                res = Net::HTTP.start(uri.hostname, uri.port) do |http|
                      http.request(req)
                end

                head :ok, content_type: "text/html"
            end
        end
```

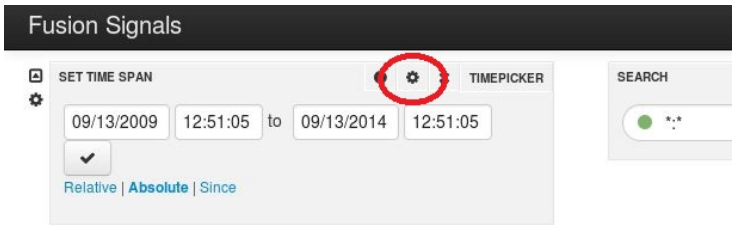
Since the request parameter "commit" is explicitly set to "true" in the Signals API requests, the raw events and their data are available to view immediately.

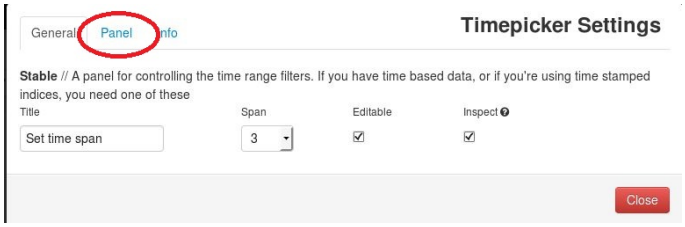
Visualizing Signals

Now that there are some signals indexed, it's time to become familiar with Dashboards and how to visualize the data. The Fusion Dashboards can be found at http://<fusion_home>:8764/banana/index.html#/dashboard. The default dashboard is "Fusion Logs" so navigate to the signals dashboard by clicking "Lucidworks Fusion Signals" in the "Explore Dashboards" panel:

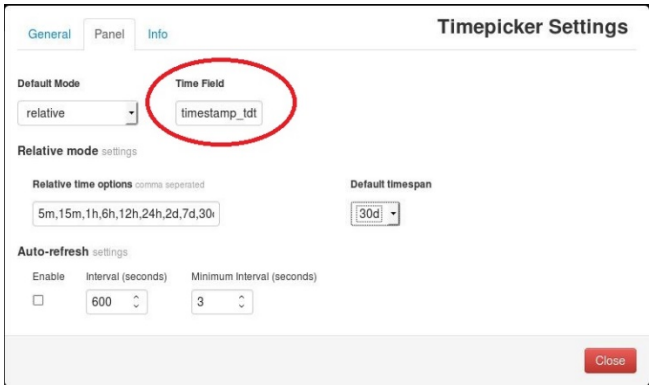


Once in the Fusion Signals dashboard, we'll need to point to our signals collection – but before we do that, we need to change the default datetime field used by the dashboard. In the "Set Time Span" panel, click the gear icon in the upper right corner to open the "Timepicker Settings" window and select the "Panel" tab:

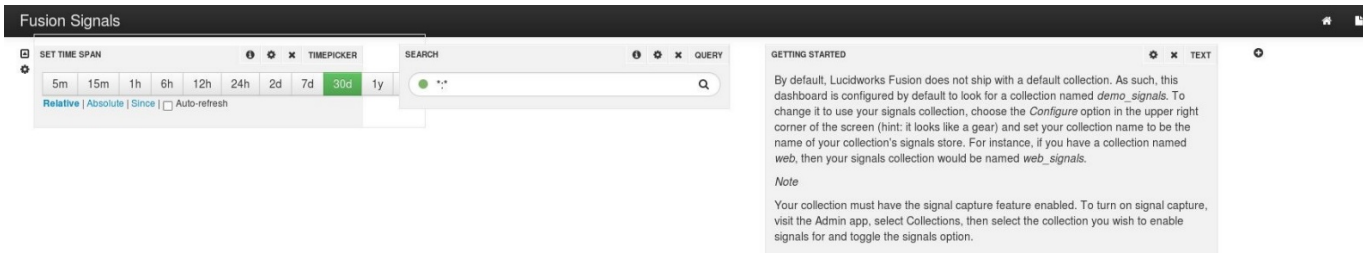




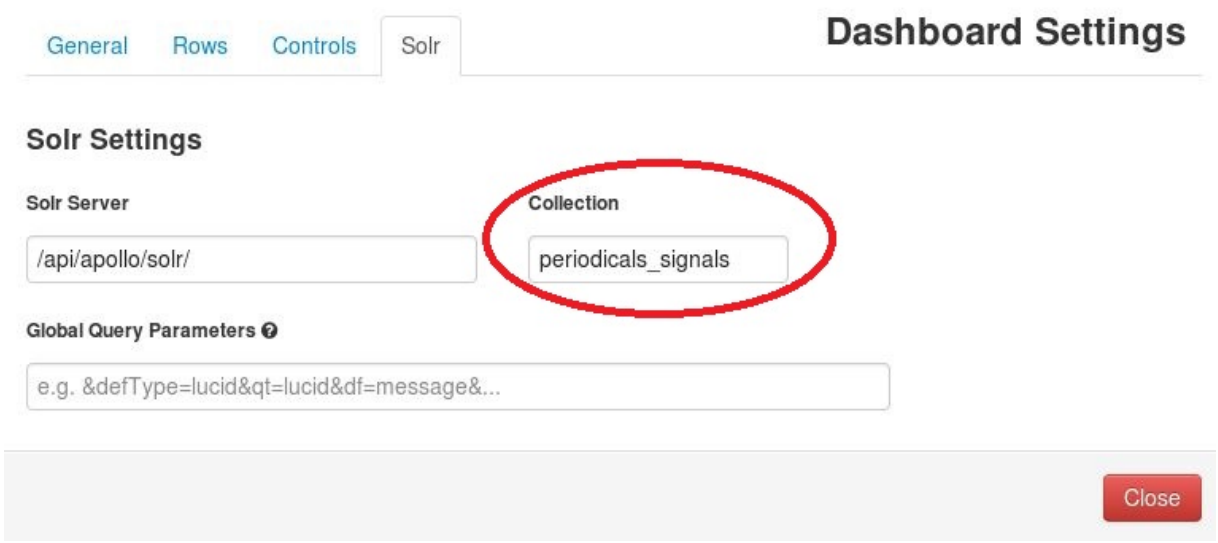
Change the "Default Mode" to "relative", "Default timespan" to "30d" and most importantly, "Time Field" to "timestamp_tdt":



Now we can select our signals collection – click the gear icon in the upper right corner of the dashboard (note the "Getting Started" panel that includes these directions):

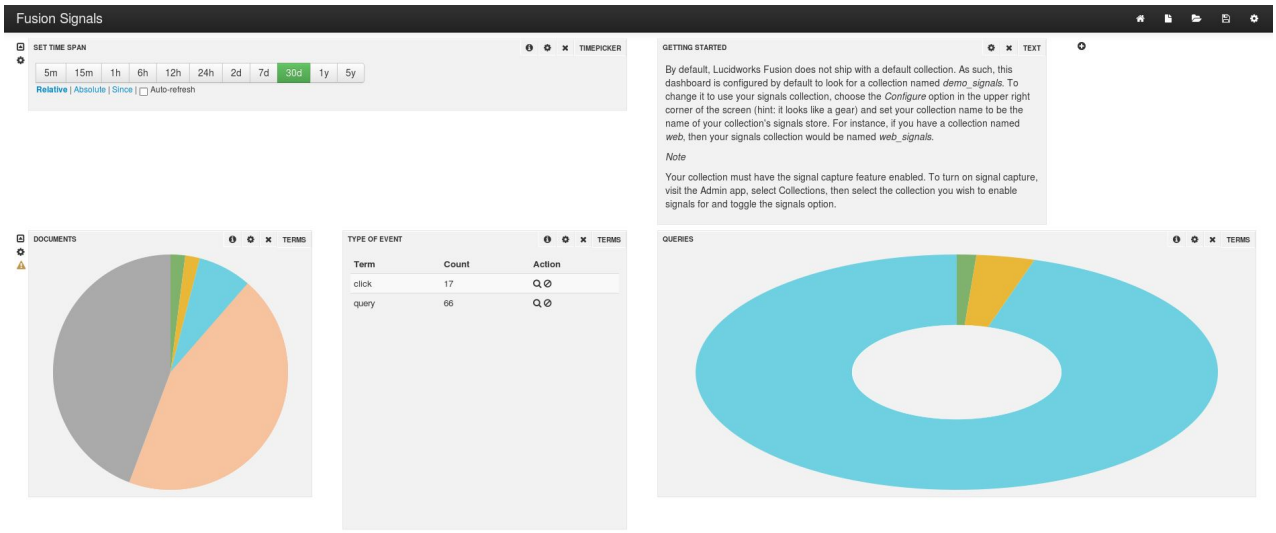


The "Dashboard Settings" window will open; click the "Solr" tab and change "Collection" to the name of your signals collection:



The dashboard will automatically refresh and the default panels will display metrics for the signals indexed in the collection:

Lucidworks



Next - Signal Aggregation

While the raw signals show some interesting information, for the data to be most useful you will need to aggregate it - read in raw signals and return interesting summaries, ranging from simple sums to sophisticated statistical functions. We'll explore signal aggregation in a future blog post, using the signals we are now capturing using the Signals API.

About the Author

Sean Mare is a technologist with almost 20 years of experience in enterprise application design and development. As Solution Architect with Knowledgent, a leading Big Data and Analytics consulting organization and partner with Lucidworks, he leverages the power of enterprise search to enable people and organizations to explore their data in exciting and interesting ways. He resides in the greater New York City area.