Internal Deal Discovery Caching

Date	User	Change
24 May 2016	Miguel Cordones	Created

Introduction

Internal deal discovery presents a use case where we need to aggregate data from sources other than ACE. In particular, LANA. Since LANA calls are too slow for page views, and since the LANA estimates wouldn't change significantly over a short period, a caching solution would be desirable. In addition, since deal discovery involves nearly arbitrary searching (a dozen fields according to one story), ElasticSearch seems like a natural choice. ElasticSearch is essentially a fully searchable json document store. Individual json documents are added to collections, which are called a (named) index. You can write search queries (in the Query DSL) to run on an index to search for matching documents.

CAVEAT: ElasticSearch does not support ACID transactions when more than one table is involved. However, I think we can work around that since, ***from the user perspective, the data in ElasticSearch is read only***, and we can engineer a solution such that all insert, update, and deletes occur serially (message queue).

Objectives

- to describe a back end caching architecture
- to describe the cache initialization and update processes
- to describe the cache query processes

Architecture Overview

Figure 1 shown an ACE Centric view. It uses an ACE Job (ETL) that runs LANA queries and stores the results in a temporary table in ACE. The deal discovery API queries the temporary table and presents the data to the client. This configuration uses familiar vocabulary and components.

`Figure 1`

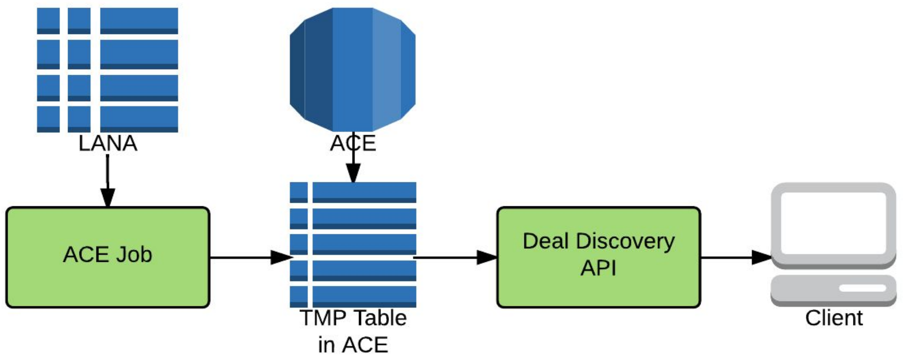
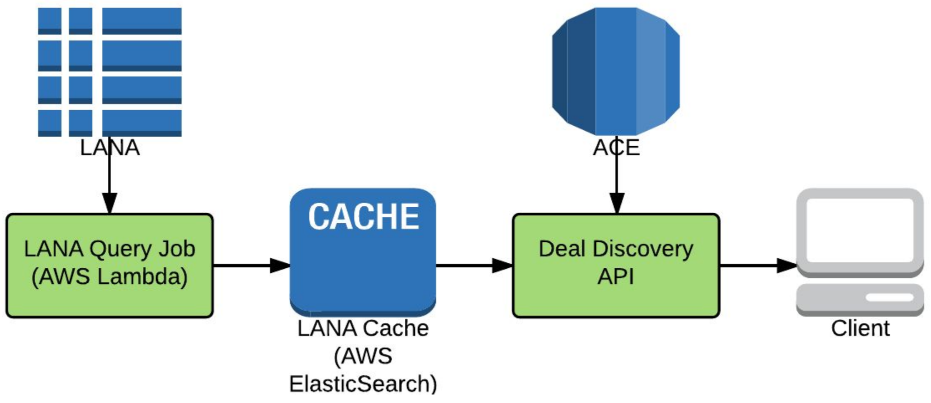


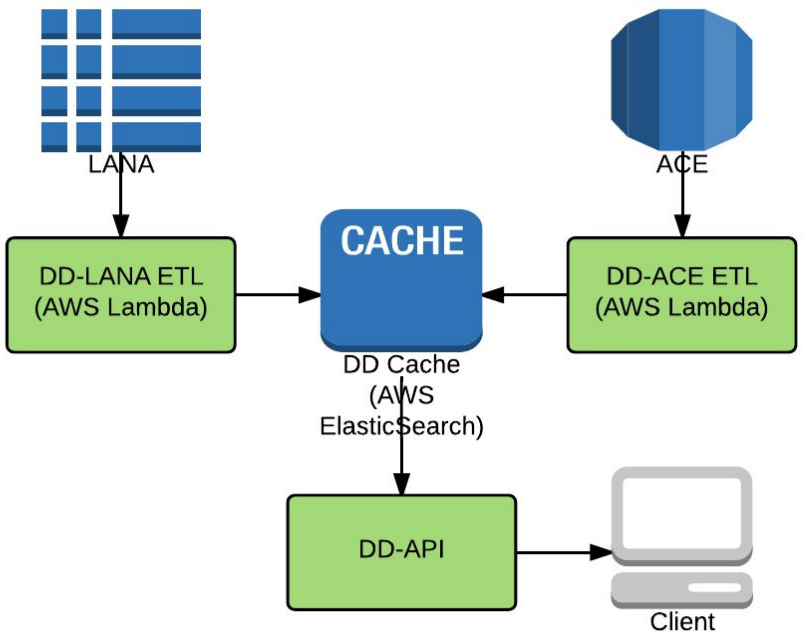
Figure 2 evolves the architecture by replacing the ACE temporary table with ElasticSearch, and the ACE Job ETL with an AWS Lambda function to cache the LANA data.

`Figure 2`



This looks ***slightly*** better to me. My primary problem is the DD-API now needs to access two data sources: ACE and AWS-ES. We have added a bit of complexity. Furthermore, the search traffic is mainly going to fall on ACE, since that's where most of the searchable data resides. This is no different than in figure 1. The final evolution, shown in figure 3, simply adds a new Lambda job to cache the ACE data. The DD-API then accesses a single, uniform, search optimized data source to present data to the client.

```
Figure 3
```



Cache Initialization and Updates

There is a progression of refinements we can apply to the cache initialization and update process.

Batch ETL, Copy & Swap

The simplest solution is a batch ETL that loads the cache 'in the background'. By that, I mean that when the ETL runs, it is loading data into a shadow copy of the cache. The primary copy can continue to service requests without interruption. When the ETL has finished loading the shadow copy, we can rename the primary copy, then name the shadow copy so that it is now primary. Except renaming an index isn't a supported operation. There are index aliases, which look promising. Need to expand this.

Batch ETL feeding a Message Queue

A next step would be for the ETL to not interact directly with the ES domain, but rather feed a message queue. Another process monitors the message queue and performs the inserts into the ES index. There isn't much direct benefit to this evolution, but it will allow for 'concurrent' updates to the ES index by any process which also uses the message queue. In particular, our Deal Management API could post individual transactions to the MQ.

Individual Transactions Feeding a Message Queue

The final evolution is so update the DM-API functions to post insert, update, and delete messages to the message queue. This would be a performance benefit in that the DD Cache would 'see' individual changes in near real time, without a full ETL run. For example, let's say a new deal is added. If the ETL runs every hour, then the new deal wouldn't be available to the DD Cache for an hour (worst case). If the DM-API would post the new deal to the message queue, the DD Cache would see the new deal immediately.

Cache Query

In order to query the DD Cache, code would need to integrate an ElasticSearch client library. My aws-es sample repo, uses the Jest client library for Java. Once we define the json documents representing the deals and LANA estimates, we can craft the ES query templates that we will use to build the query statements.

I'm leaning toward separate indices (data stores) for the deal data and the LANA data. The benefit is that each can be updated independently. The drawback is a potential performance hit because a deal lookup would require an additional lookup for the LANA index.

I envision wrapping all this up behind a service, so from the client it will look like an aggregated data store. My motivation for keeping separate forecast and deal indices are:

1. I expect forecast data to change more frequently than the deal data. We will continually get new deal data, but once we have a deal record, it will be relatively static.
2. The forecast data contains a different set of deals from ACE (or any single provider). We may have forecasts for which we have no other details of the deal, or we may have deals that have no forecast (we will assume 0).

If necessary, we could mitigate the performance penalty by adopting a 'lazy' lookup strategy. By that I mean we could define the deal document to have an optional LANA estimate section, which is initially empty. When we retrieve a deal document from the cache, an empty LANA section would signal the system to fetch the LANA data from the cache and merge it into the deal document. The system would then update the cache (see Individual Transactions Feeding a Message Queue above) with the augmented deal document and return the document to the client. The next time that document is retrieved, we already have the LANA estimates in place.

CAVEAT: we need to make sure that if we get fresh LANA estimates, we refresh or reload the deal data so that we aren't returning stale LANA data.

TODO

1. Align all of the terminology.