Activity Trail Design

Maxim Noah Khailo

December 3, 2015

1 Purpose

We need to track all important activity of the system, both machine and human so that our customers can keep track and organize their business.

2 Forces

Forces are problems we have to balance.

- Activity is both human and machine.
- Machine activity can lead to human activity and vice versa.
- The information for each activity needs to be viewable.
- You can view activities from multiple perspectives.

3 Data Structure

The following data structure design is motivated by the fact that we want to linearize activities along many dimensions. It is composed of two parts, activities and trails along dimensions.

Each activity belongs to many trails. A trail belongs to a dimension. A trail is implemented as a doubly-linked list where each node in the list is called a Connection.

Each Connection, connects the activity to the trail.

3.1 Activity

An activity tracks changes to state made by both the computer and human actors.

We use the term Activity here instead of Event. Though you can think of them as the same thing.

```
case class Activity(
    id: Id,
    type: Type,
    data: Json,
    created: Instant)
```

One requirement is that the activity stores all data needed to visualize and compute on that activity in the data field. The type is used to understand how to decode the json for visualization.

An activity object is also immutable since it tracks changes to state but is NOT state.

3.2 Dimensions and Trails

So what are dimensions? Dimensions are a collection of activity trails.

```
case class Dimension(
    id: Id,
    name: String,
    description: String)
```

For example, we can look at activities for a customer. So we can have a *customer_activity* dimension which will have a head activity that contains the customer.id in the json.

Another example, we may want to look at activities for an order. So we can have an *order_activity* dimension which will have a head activity that contains the order.id in the json.

Another example, we may want to look at all activities by an admin. So we can have an *admin_activity* dimension.

Another example, watch list is just another activity trail. So we would have a *watched_activities* dimension.

3.3 Connection

Each activity exists in many trails. The trails are modeled as doubly-linked lists where a node in the list is called a Connection. It connects an activity to a trail.

```
case class Connection(
    id: Id,
    dimension: Id,
    activity: Id,
    previous: Id,
    next: Id,
    tail: Id,
    trailId: Id,
    data: Json)
```

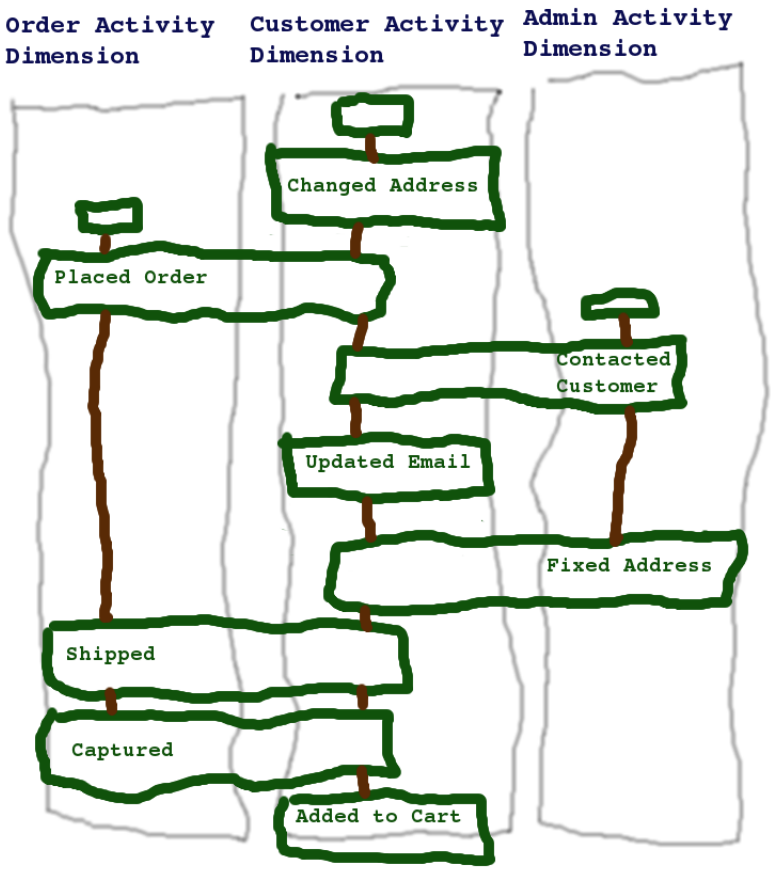
Each activity is connected to a trail by a connection creating a doubly linked list. Each trail belongs to some dimension.

Each list has a tail activity and the tail activity has a connection where the *previous* points to the last activity in the trail and the *next* points to the oldest activity (the head).

Each connection points to the tail for optimization purposes. We can create an index on the tail id to do quick queries based on the dimension.

Each connection also has a trailId which is used to quickly query for all connections grouped by that id. The use case is if a dimension represents activity for a type of object, say customer, where we can put the customer id as the trail id.

Also each connection can have data. For example, if we model a notification list as a trail in the watched_activities dimension, we can model whether a user seen the activity by storing that state in the data section on a connection.



4 Visualization

One requirement for an activity is that all the data necessary for visualization is stored in the JSON. While this will increase the amount of data stored, it will improve performance in the UI by reducing the amount of queries.

5 Navigation

One huge advantage to the data structure is that you can easily navigate it along multiple dimensions efficiently. This provides interesting UI options in the future when you need to jump from one view of activities to another.

6 Implementation

6.1 How are Activities Generated?

Activities themselves, since they are targeted towards end user will mostly be generated in phoenix and inserted into one activity table. Some activities might be able to be created via kafka consumers but phoenix usually has all the metadata on hand to create an activity.

6.2 How are Activities Connected to Trails?

Activities are connected to a trail in a dimension by many Kafka consumers. The consumers will listen to the activity stream and then connect the activity to a trail.

For example, we will have a notification consumer which will take an activity, check who is watching the object that activity tracks and add it to the user's notification activity trail.

Magical.

7 Pros

1. Performance. Cachable completely in Elastic Search
2. Simplified UI. Activity trail can use the search api instead of hitting DB. We can use existing search infrastructure to filter activity trail.
3. Simplified schema. DB schema is super simple and extensible as we add more dimensions. Otherwise we would have to store nulled ids in columns, or have different tables for different trails.
4. Sets us up for implementing workflows.
5. Implementing watch and assignment is easy because of how dimensions work. You can watch an order, or a customer, or an admin, etc.

8 Cons

There are many problems to this design.

1. Adding an activity requires adding it to multiple trails in many dimensions.
2. Size used is greater because each activity stores all data needed for visualization.
3. Tracking activity is now explicit. Either phoenix needs changes or we create Kafka consumers which look at db changes and create the activity objects.