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 events along many dimensions. It is composed of two parts, activities and doubly linked lists along dimensions, where each dimension is a set of these linked lists where the head is used to represent the start of the activity along that dimension.

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 because events are a type of activity, and activity implies a grouping.

```
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. 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 Connection

Each activity exists in doubly-linked lists in multiple dimensions.

We connect activities together instead of linking them. Linking is traditionally used as a term to mean pointer. Instead, we will store a two way *Connection*

```
case class Connection(
    id: Id,
    dimension: Id,
    activity: Id,
    previous: Id,
    next: Id,
    tail: Id,
    groupId: Id)
```

Each activity is connected by a two way connection creating a doubly linked list. Each connection belongs to some dimension and in each dimension an activity belongs to only one list.

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 groupId 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 group id.

3.3 Dimensions

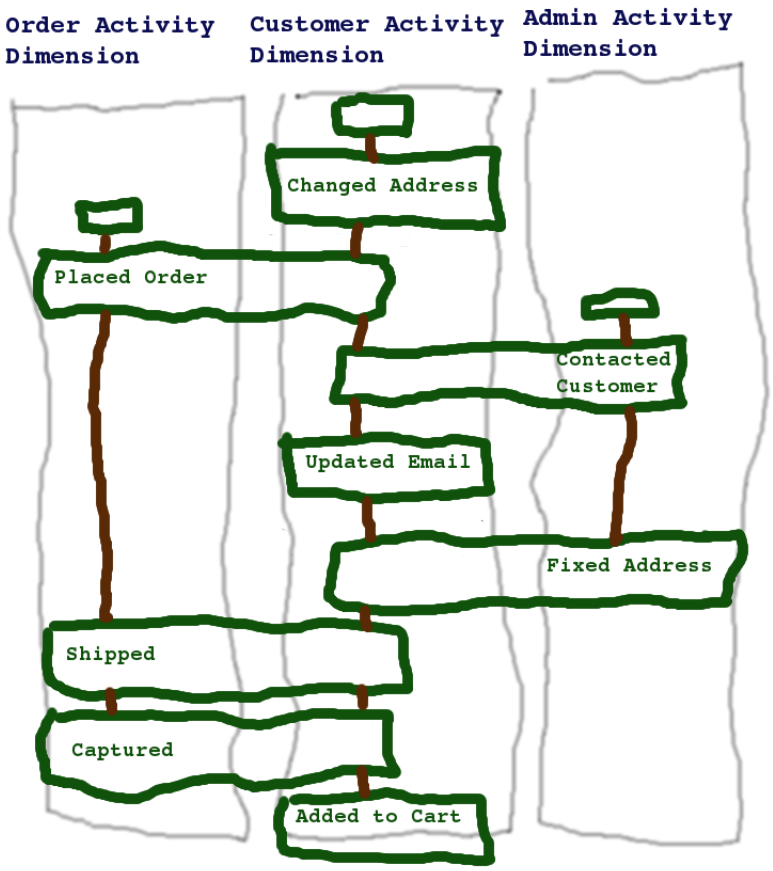
So what are dimensions and how do they help model activities trails? Dimensions are essentially a precomputed linearization of events based on a logical grouping.

```
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.



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 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.

7 Cons

There are many problems to this design.

1. Adding an activity requires adding it to multiple lists 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.