An efficient schema for hierarchical data on Elasticsearch

Description	Supporting arbitrary nesting and fast queries
Period	Fall 2016
Languages & Libs	Python
Tags	Business Intelligence, Databases, Elasticsearch

Many businesses generate rich datasets from which valuable insights can be discovered. A basic starting point is to analyze separate events such as item sales, tourist attraction visits or movies seen. From these a time series (total sales / item / day, total visits / tourist spot / week) or basic metrics (histogram of movie ratings) can be aggregated. Things get a lot more interesting when individual data points can be linked together by a common id, such as items being bought in the same basket or by the same house hold (identified by a loyalty card), the spots visited by a tourist group through out their journey or movie ratings given by a specific user. This richer data can be used to build recommendation engines, identify substitute products or services and do clustering analysis. This article describes a schema for Elasticsearch which supports efficient filtering and aggregations, and is automatically compatible with new data values.

It is true that even moderately large datasets may have a decent performance on a single traditional SQL server with modern technologies such as columnar storage engines. However I find horizontally scalable solutions more interesting as they don't put a hard limit on the amount of data or users you can handle at once, and expensive vertical scaling isn't the only option. Also you can reduce the amount of raw data by a few orders of magnitude by sampling techniques but then your answers won't be exact anymore, and ideally you'd handle and indicate this uncertainty on your reporting tools somehow. Also in SQL you don't need to think about the expected queries so much in advance as you can flexibly `JOIN` data from different tables on a ad-hoc manner. Also the data size on disk could be smaller as you don't need to de-normalize data on documents as you have to do with most NoSQL solutions.

Many questions can be answered by just storing a set of numerical ids into a field, whereas some questions require additional data such as total amount of money or time spent on a specific item. The first dataset can apply filters like "tourists who visited Paris" but the former can filter for "tourists who spent more than two days in Paris". Also the first one can only aggregate "most frequently visited restaurants of tourists who visited a museum in Paris", but the second one can aggregate "restaurants with most money spent by tourists who stayed in Paris for just one day". The first one is a lot simpler to implement and query in Elasticsearch as it directly supports multiple values on a field, the second one requires nested documents such as `{"location_id": 123, "location_type": "city", "money_spent": 123.4, "time_spent": 17.5}`. Note that this data model does not record the order in which the cities were visited, but can easily be handled by having `prev_location_id` and `next_location_id` fields. On some domains a graph database could be more suitable option than others.

Naturally you should store top-level aggregates directly to the root document, such as `total_money_spent` and `total_time_spent`. They have minimal effect on data size but greatly simplify filtering and aggregations. Also at the time of writing Kibana does not seem to support nested aggregations (discussion at Github). I would also store other types of information such as the number of cities and POIs visited, and fields which can assumed to be static such as number of tourists and their nationalities.

An other important topic is how to model hierarchical nature of the data. For example we have the geographical dimension with the location itself (ideally an integer id), its street, region of city, city, region of country, country, region of continent and the continent itself. A straight-forward option is a pure coordinates-based one, which has implicit regional hierarchy as defined by the geohash prefix aggregation. It can create cool heat-maps at desired resolution but loses the explicit information on data's hierarchy. On some use cases a pie diagram with two or more levels of hierarchy might be an useful representation, for example to see the percentage of time spent on most popular cities, and for each city the percentage of time spent on each type of attraction. This is made possible by simply adding `parent_location_id` to each sub-document.