8.5. Denormalizing: using redundant data connections

Denormalizing is about multiplying data in order to avoid expensive joins. Let’s take an example we’ve already discussed: groups and events. It’s a one-to-many relationship because an event can be hosted by only one group, and one group can host many events.

With parent-child or nested structures, groups and events are stored in different Lucene documents, as shown in figure 8.15.

Figure 8.15. Hierarchical relationship (nested or parent-child) between different Lucene documents

This relationship can be denormalized by adding the group info to all the events, as shown in figure 8.16.

Figure 8.16. Hierarchical relationship denormalized by copying group information to each event

Next we’ll look at how and when denormalizing helps and how you’d concretely index and query denormalized data.

8.5.1. Use cases for denormalizing

Let’s start with the disadvantages: denormalized data takes more space and is more difficult to manage than normalized data. In the example from figure 8.16, if you change the group’s details, you have to update three documents because those details appear three times.

On the positive side, you don’t have to join different documents when you query. This is particularly important in distributed systems because having to join documents across the network introduces big latencies, as you can see in figure 8.17.

Figure 8.17. Joining documents across nodes is difficult because of network latency.

Nested and parent-child documents get around this by making sure a parent and all its children are stored in the same node, as shown in figure 8.18:

Nested documents are indexed in Lucene blocks, which are always together in the same segment of the same shard.

Child documents are indexed with the same routing value as their parents, making them belong to the same shard.

Figure 8.18. Nested/parent-child relations make sure all joins are local.

Denormalizing one-to-many relations

Local joins done with nested and parent-child structures are much, much faster than remote joins could be. Still, they're more expensive than having no joins at all. This is where denormalizing can help, but it implies that there's more data. Your indexing operations will cause more load because you'll index more data and queries will run on larger indices, making them slower.

You can see that there's a tradeoff when it comes to choosing among nested, parent-child, and denormalizing. Typically, you'll denormalize for one-to-many relations if your data is fairly small and static and you have lots of queries. This way, disadvantages hurt less—index size is acceptable and there aren't too many indexing operations—and avoiding joins should make queries faster.

Tip

If performance is important to you, take a look at chapter 10, which is all about indexing and searching fast.

Denormalizing many-to-many relationships

Many-to-many relationships are dealt with differently than one-to-many relationships in Elasticsearch.

For example, a group can contain multiple members, and a person could be a member of multiple groups.

Here denormalizing is a much better proposition because unlike one-to-many implementations of nested and parent-child, Elasticsearch can't promise to contain many-to-many relationships in a single node. As shown in figure 8.19, a single relationship may expand to your whole dataset. This would make expensive, cross-network joins inevitable.

Figure 8.19. Many-to-many relationships can contain a huge amount of data, making local joins impossible.

Because of how slow cross-network joins would be, as of version 1.5, denormalizing is the only way to represent many-to-many relationships in Elasticsearch. Figure 8.20 shows how the structure of figure 8.19 looks when members are denormalized as children of each group they belong to. We denormalize one side of the many-to-many relationship into more one-to-many relationships.

Figure 8.20. Many-to-many relation denormalized into multiple one-to-many relations, allowing local joins

Next we'll look at how you can index, update, and query a structure like the one in figure 8.20.

8.5.2. Indexing, updating, and deleting denormalized data

Before you start indexing, you have to decide how you want to denormalize your many-to-many into oneto-many, and there are two big decision points: which side of the relationship you should denormalize and how you want to represent the resulting one-to-many relationship.

Which side will be denormalized?

Will members be multiplied as children of groups or the other way around? To pick one you have to understand how data is indexed, updated, deleted, and queried. The part that's denormalized—the child— will be more difficult to manage in all aspects:

You index those documents multiple times, once for each of its parents.

When you update, you have to update all instances of that document.

When you delete, you have to delete all instances.

When you query for children separately, you'll get more hits with the same content, so you have to remove duplicates on the application side.

Based on these assumptions, it looks like it makes more sense to make members children of groups. Member documents are smaller in size, change less often, and are queried less often than groups are with their events. As a result, managing cloned member documents should be easier.

How do you want to represent the one-to-many relationship?

Will you have parent-child or nested documents? You'd choose here based on how often groups and members are searched and retrieved together. Nested queries perform better than has_parent or has_child queries.

Another important aspect is how often membership changes. Parent-child structures perform better here because they can be updated separately.

For this example, let's assume that searching and retrieving groups and members together is rare and that members often join and leave groups, so we'll go with parent-child.

Indexing

Groups and their events would be indexed as before, but members have to be indexed once for every group they belong to. The following listing will first define a mapping for the new member type and then index Mr. Hinman as a member of both the Denver Clojure and the Denver Elasticsearch groups from the code samples.

Multiple indexing operations can be done in a single HTTP request by using the bulk API. We'll discuss the bulk API in chapter 10, which is all about performance.

Updating

Once again, groups get lucky and you update them just as you saw in chapter 3, section 3.5. But if a member changes its details because it's denormalized, you'll first have to search for all its duplicates and then update each one. In listing 8.11, you'll search for all the documents that have an _id of "10001" and update his first name to Lee because that's what he likes to be called.

You're searching for IDs instead of names because IDs tend to be more reliable than other fields, such as names. You may recall from the parent-child section that when you're using the _parent field, multiple documents within the same type within the same index can have the same _id value. Only the _id and _parent combination is guaranteed to be unique. When denormalizing, you can use this feature and intentionally use the same _id for the same person, once for each group they belong to. This allows you to quickly and reliably retrieve all the instances of the same person by searching for their ID.

Listing 8.11. Updating denormalized members

Multiple updates can also be done in a single HTTP request over the bulk API. As with bulk indexing, we'll discuss bulk updates in chapter 10.

Deleting

Deleting a denormalized member requires you to identify all the copies again. Recall from the parentchild section that in order to delete a specific document, you have to specify both the _id and the _parent; that's because the combination of the two is unique in the same index and type. You'd have to identify members first through a term filter like the one in listing 8.11. Then you'd delete each member instance:

% curl -XDELETE 'localhost:9200/get-together/member/10001**?parent=1**'

% curl -XDELETE 'localhost:9200/get-together/member/10001**?parent=2**'

Now that you know how to index, update, and delete in denormalized members, let's look at how you can run queries on them.

8.5.3. Querying denormalized data

If you need to query groups, there's nothing denormalizing-specific because groups aren't denormalized. If you need search criteria from their members, use the has_child query as you did in section 8.4.2.

Members got the shortest straw with queries, too, because they're denormalized. You can search for them, even including criteria from the groups they belong to, with the has_parent query. But there's a problem: you'll get back identical members. In the following listing, you'll index another two members, and when you search, you'll get them both back.

Listing 8.12. Querying for denormalized data returns duplicate results

As of version 1.5, you can only remove those duplicate members from your application. Once again, if the same person always has the same ID, you can use that ID to make this task easier: two results with the same ID are identical.

The same problem occurs with aggregations: if you want to count some properties of the members, those counts will be inaccurate because the same member appears in multiple places.

The workaround for most searches and aggregations is to maintain a copy of all members in a separate index. Let's call it “members.” Querying that index will return just that one copy of each member. The problem with this workaround is that it only helps when you query members alone, unless you're doing application-side joins, which we'll discuss next.

Using denormalization to define relationships: pros and cons

As we did with the other methods, we provide a quick overview of the strengths and weaknesses of denormalizing. The plus points:

It allows you to work with many-to-many relationships.

No joins are involved, making querying faster if your cluster can handle the extra data caused by duplication.

The downsides:

Your application has to take care of duplicates when indexing, updating, and deleting. Some searches and aggregations won’t work as expected because data is duplicated.