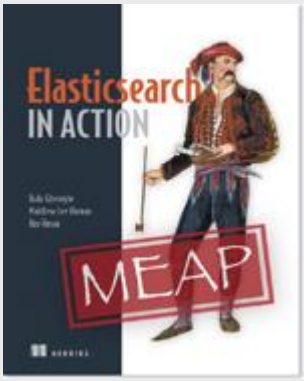


A look at how Elasticsearch is organized

By Radu Gheorghe, Matthew Lee Hinman, and Roy Russo

In this article, excerpted from Elasticsearch in Action, we explain how Elasticsearch is organized.

Imagine you're tasked with creating a way to search through millions of documents, like a website that allows people to build common interest groups and get together. In this case, documents could be the get-together groups, individual events. You need to implement this in a fault-tolerant way, and you need your setup to be able to accommodate more data and more concurrent searches, as your get-together site becomes more successful.

In this article, we'll show you how to deal with such a scenario by explaining how Elasticsearch data is organized.

All operations will be done using *cURL*, a nice little command-line tool for HTTP requests. Later, you can translate what cURL does into your preferred programming language if you need to.

We'll get started with data organization. To understand how data is organized in Elasticsearch, we'll look at it from two angles:

- *Logical layout*—What your search application needs to be aware of

  The unit you'll use for indexing and searching is a document, and you can think of it like a row in a relational database. Documents are grouped into types, which contain documents in a similar way to how tables contain rows. Finally, one or multiple types live in an *index*, the biggest container, similar to a database in the SQL world.

- *Physical layout*—How Elasticsearch handles your data in the background

  Elasticsearch divides each index into *shards*, which can migrate between servers that make up a cluster. Typically, applications don't care about this because they work with Elasticsearch in much the same way, whether it's one or more servers. But when you're administering the cluster, you care because the way you configure the physical layout determines its performance, scalability, and availability.

Figure 1 illustrates the two perspectives:

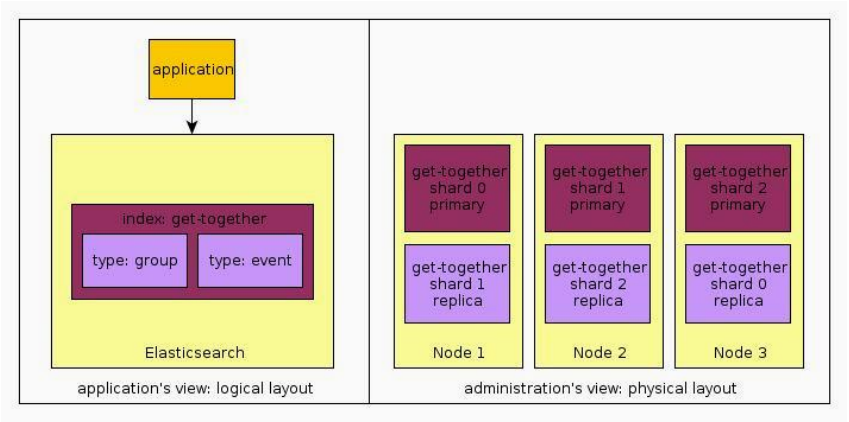


Figure 1 An Elasticsearch cluster from the application's and administrator's points of view

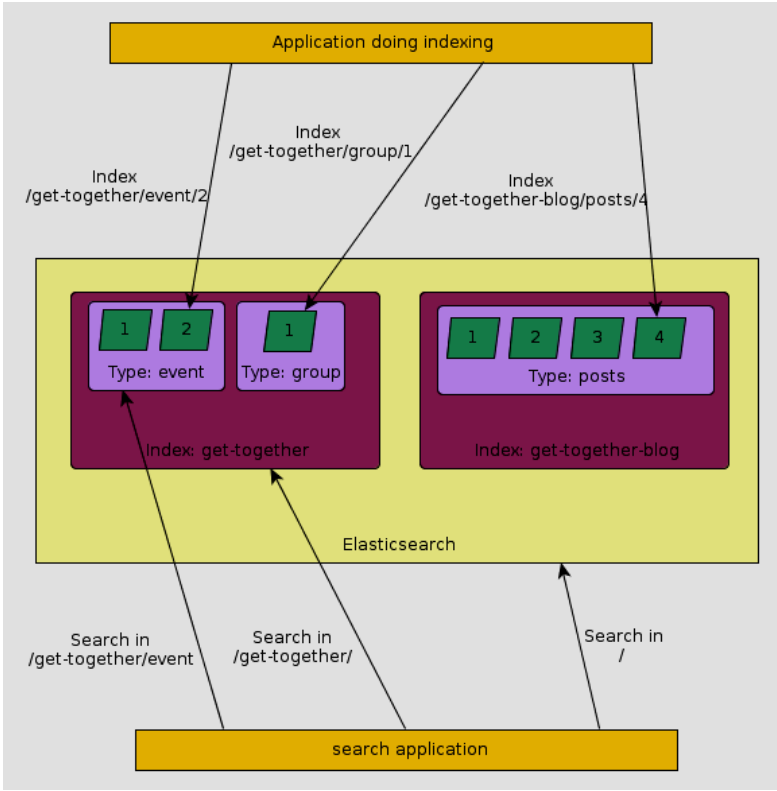
Let's start with the logical layout—or what the application sees.

Understanding the logical layout: documents, types, and indices

When you index a document in Elasticsearch, you put it in a type within an index. You can see this idea in figure 2, where the get-together index contains two types: event and group. Those types contain documents, such as the one labeled 1. The label `1` is that document's ID.

> **TIP** The ID doesn't have to be an integer. It's actually a string, and there are no constraints – you can put there whatever makes sense for your application.

The index-type-ID combination uniquely identifies a document in your Elasticsearch setup. When you search, you can look for documents in that specific type, of that specific index, or you can search across multiple types or even multiple indices.



#A (Add arrow to the left of /get-together/event/1) Index name + type name + document ID = uniquely identified document

Figure 2 Logical layout of data in Elasticsearch: how an application sees data

At this point you might be asking: What exactly is a document, type, and an index? That's exactly what we're going to discuss next.

DOCUMENTS

Elasticsearch is *document-oriented*, meaning the smallest unit of data you index or search for is a document. A document has a few important properties in Elasticsearch:

- It's *self-contained*. A document contains both the fields (`name`) and their values (`Elasticsearch Denver`).
- It can be *hierarchical*. Think of this as documents within documents. A value of a field can be simple, like the value of the location field can be a string. It can also contain other fields and values. For example, the `location` field might contain both a city and a street address within it.
- It has a *flexible structure*. Your documents don't depend on a predefined schema. For example, not all events need description values, so that field can be omitted altogether. But it might require new fields, such as the latitude and longitude of the location.

A document is normally a JSON representation of your data. JSON over HTTP is the most widely used way to communicate with Elasticsearch. For example, an event in your get-together site can be represented in the following document:

```
{
  "name": "Elasticsearch Denver",
  "organizer": "Lee",
  "location": "Denver, Colorado, USA"
}
```

NOTE Field names are `darker/blue`, and values are in `lighter/red`.

You can also imagine a table with three columns: name, organizer, and location. The document would be a row containing the values. But there are some differences that make this comparison inexact.One difference is that, unlike rows, documents can be hierarchical. For example, the location can contain a name and a geolocation:

```
{
  "name": "Elasticsearch Denver",
  "organizer": "Lee",
  "location": {
    "name": "Denver, Colorado, USA",
    "geolocation": "39.7392, -104.9847"
  }
}
```

A single document can also contain arrays of values. For example:

```
{
  "name": "Elasticsearch Denver",
  "organizer": "Lee",
```

```
  "members": ["Lee", "Mike"]
```

Documents in Elasticsearch are said to be *schema-free*, in the sense that not all your documents need to have the same fields, so they're not bound to the same schema. For example, you could omit the location altogether, in case the organizer needs to be called before every gathering:

```
{
  "name": "Elasticsearch Denver",
  "organizer": "Lee",
  "members": ["Lee", "Mike"]
}
```

Although you can add or omit fields at will, the type of each field matters: some are strings, some are integers, and so on. Because of that, Elasticsearch keeps a mapping of all your fields and their types, and other settings. This mapping is specific to every type of every index. That's why types are sometime call*ed mappi*ng types in Elasticsearch terminology

TYPES

Types are logical containers for documents, similar to how tables are containers for rows. You'd put documents with different structures (schemas) in different types. For example, you could have a type that defines get-together groups and another type for the events when people gather.

The definition of fields in each type is called a *mappi*ng. For example, `name` would be mapped as a `string`, but the `geolocation` field under `location` would be mapped as a special `geo_point` type. Each kind of field is handled differently. For example, you search for a word in the `name` field, and you search for groups by location to find those that are located near where you live.

> **TIP** Whenever you search in a field that isn't at the root of your JSON document, you must specify its path. For example, the `geolocation` field under location is referred to as `location.geolocation`.

You may ask yourself: If Elasticsearch is schema-free, why does each document belong to a type, and each type contains a mapping, which is like a schema?

We say *schema-free* because documents are not bound to the schema. They aren't required to contain all the fields defined in your mapping and may come up with new fields. How does it work? First, the mapping contains all the fields of all the documents indexed so far in that type. But not all documents have to have all fields. Also, if a new document gets indexed with a field that's not already in the mapping, Elasticsearch automatically adds that new field to your mapping. To add that field, it has to decide what type it is, so it guesses it. For example, if the value is `7`, it assumes it's a `long` type.

This autodetection of new fields has its downside because Elasticsearch might not guess right. For example, after indexing `7`, you might want to index `hello world`, which will fail because it's a `string` and not a `long`. In production, the safe way to go is to define your mapping before indexing data.

Mapping types only divide documents logically. Physically, documents from the same index are written to disk regardless of the mapping type they belong to.

INDICES

Indices are containers for mapping types. An Elasticsearch *index* is an independent chunk of documents, much like a database is in the relational world: each index is stored on the disk in the same set of files; it stores all the fields from all the mapping types in there, and it has its own settings. For example, each index has a setting called `refresh_interval`, which defines the interval at which newly indexed documents are made available for searches. This *refresh* operation is quite expensive in terms of performance, and this is why it's done occasionally—by default, every second—instead of doing it after each indexed document. If you've read that Elasticsearch is *near-real-time*, this refresh process is what it refers to.

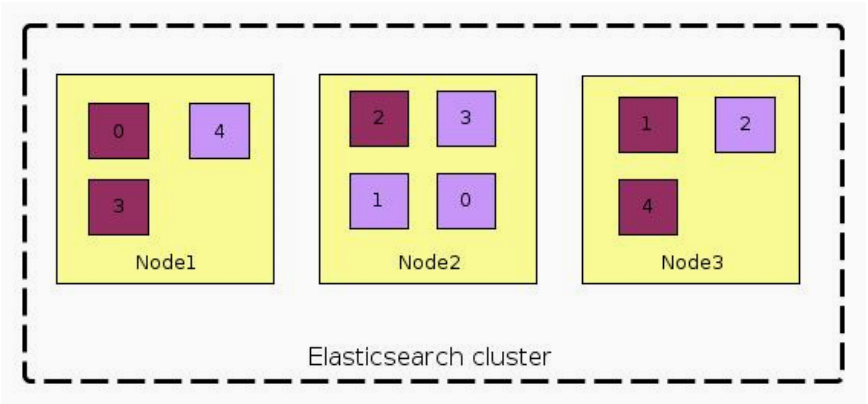
> **TIP** Just like you can search across types, you can search across indices. This gives you flexibility in the way you can organize documents. For example, you can put your get-together events and the blog posts about them in different indices or in different types of the same index. Some ways are more efficient than others, depending on your use case.

One example of index-specific settings is the number of shards. An index can be made up of one or more chunks called shards. This is good for scalability: you can run Elasticsearch on multiple servers and have shards of the same index live on all of them. Next, we'll have a closer look at how sharding works in Elasticsearch.

Understanding the physical layout: nodes and shards

Understanding how data is physically laid out boils down to understanding how Elasticsearch scales. We'll introduce you to how scaling works by looking at how multiple nodes work together in a cluster, how data is divided in shards and replicated, and how indexing and searching works with multiple shards and replicas.

To understand the big picture, let's review what happens when an Elasticsearch index is created. By default, each index is made up of five primary shards, each with one replica, for a total of ten shards, as illustrated in figure 3.



#A [Arrow from the left to Node1] A node is a process running Elasticsearch
#B [Arrow from the top to primary shard 2 of Node 2] A primary shard is a chunk of your index
#C [Arrow from the right to replica 2 of Node3] A replica is a copy of a primary shard

Figure 3 A three-node cluster with an index divided into five shards with one replica per shard

As you'll explore next, replicas are good for reliability and search performance. Technically, a shard is a directory of files where Lucene stores the data for your index. A shard is also the smallest unit that Elasticsearch moves from node to node.

CREATING A CLUSTER OF ONE OR MORE NODES

A *node* is an instance of Elasticsearch. When you start Elasticsearch on your server, you have a node. If you start Elasticsearch on another server, it's another node. You can even have more nodes on the same server, by starting multiple Elasticsearch processes.

Multiple nodes can join the same *cluste*r. Starting nodes with the same cluster name, and otherwise defaut settings, is enough to make a cluster.With a cluster of multiple nodes, the same data can be spread across multiple servers. This helps performance because Elasticsearch has more resources to work with. It also helps reliability: if you have at least one replica per shard, any node can disappear, and Elasticsearch will still serve you all the data. For an application that's using Elasticsearch, having one or more nodes in a cluster is transparent. By default, you can connect to any node from the cluster and work with the whole data just as if you had a single node.

> While clustering is good for performance and availability, it has its disadvantages: you have to make sure nodes can communicate with each other quickly enough and that you won't have a split brain (two parts of the cluster which can't communicate and think the other part dropped out).

WHAT HAPPENS WHEN YOU INDEX A DOCUMENT?

By default, when you index a document, it's first sent to one of the primary shards, which is chosen based on a hash of the document's ID. That primary shard may be located on a different node, like it is on Node 2 in figure 4, but this is transparent to the application.

Then, the document is sent to be indexed in all of that primary shard's replicas (see left side of figure 4). This keeps replicas in sync with data from the primary shards. Being in sync allows replicas to serve searches and to be automatically promoted to primary shards in case the original primary becomes unavailable.

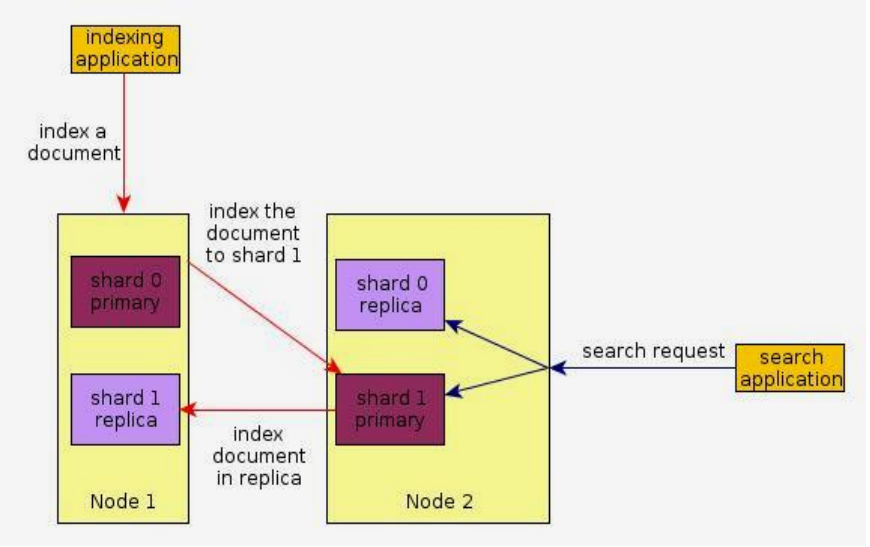


Figure 4 Documents are indexed to random primary shards and their replicas. Searches run on complete sets of shards, regardless of their status as primaries or replicas.

WHAT HAPPENS WHEN YOU SEARCH AN INDEX?

When you search an index, Elasticsearch has to look in a complete set of shards for that index (see right side of figure 4). Those shards can be either primary or replicas because primary and replica shards typically contain the same documents. Elasticsearch distributes the search load between the primary and replica shards of the index you're searching, making replicas useful for both search performance and fault tolerance.

Next, we'll look at the details of what primary and replica shards are and how they're allocated in an Elasticsearch cluster.

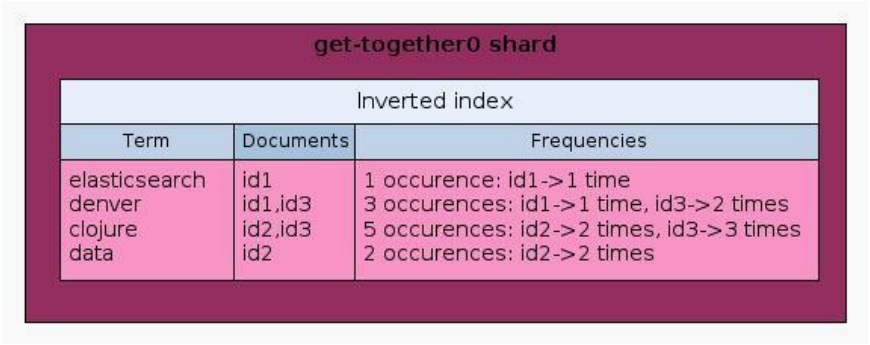
Understanding primary and replica shards

Let's start with the smallest unit Elasticsearch deals with: a shard. A *shard* is a Lucene index: a directory of files containing an inverted index. An *inverted index* is a structure that enables Elasticsearch to tell you which document contains a term (a word) without having to look at all the documents.

In Figure 5, you can see what sort of information the first primary shard of your get-together index may contain. The shard get-together0, as we'll call it from now on, is a Lucene index—an inverted index. By default, it stores the original document's content plus additional information, such as *term dictionary* and *term frequencies*, which helps searching.

The *term dictionary* maps each term to identifiers of documents containing that term (see figure 5). When searching, Elasticsearch doesn't have to look through all the documents for that term—it uses this dictionary to quickly identify all the documents that match.

Term frequencies give Elasticsearch quick access to the number of appearances of a term in a document. This is important for calculating the relevancy score of results. For example, if you search for "Denver," documents that contain "denver" many times are typically more relevant. Elasticsearch gives them a higher score, and they appear higher in the list of results. By default, the ranking algorithm is tf-idf, but you have a lot more options.



 # Arrow from the top to the black title: A shard is a Lucene index

Figure 5 Term dictionary and frequencies in a Lucene index

A shard can be either a primary or a replica shard, with *replicas* being exactly that–copies of the primary shard. A replica is used for searching, or it becomes a new primary shard if the original primary shard is lost.

An Elasticsearch index is made up of one or more primary shards and zero or more replica shards. In Figure 6, you can see that the Elasticsearch get-together index is made up of six

total shards: two primary shards (darker boxes), and two replicas for each shard (lighter boxes) for a total of four replicas.

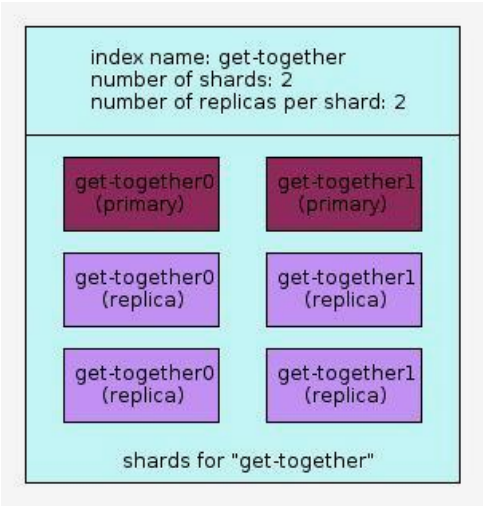


Figure 6 Multiple primary and replica shards make up the get-together index.

Replicas can be added or removed at runtime—primaries can't

You can change the number of replicas per shard at any time because replicas can always be created or removed. This doesn't apply to the number of primary shards an index is divided into; you have to decide on the number of shards before creating the index.

Keep in mind that too few shards limit how much you can scale, but too many shards impact performance. The default setting of five is typically a good start.

All the shards and replicas you've seen so far are distributed to nodes within an Elasticsearch cluster. Next, we'll look at some details about how Elasticsearch distributes shards and replicas in a cluster having one or more nodes.

Distributing shards in a cluster

The simplest Elasticsearch cluster has one node: one machine running one Elasticsearch process. When you installed Elasticsearch and started it, you created a one-node cluster.

As you add more nodes to the same cluster, existing shards get balanced between all nodes. As a result, both indexing and search requests that work with those shards benefit from the extra power of your added nodes. Scaling this way (by adding nodes to a cluster) is

called *horizontal scaling*; you add more nodes, and requests are then distributed so they all share the work. The alternative to horizontal scaling is to scale vertically; you add more resources to your Elasticsearch node, perhaps by dedicating more processors to it if it's a virtual machine, or adding RAM to a physical machine. Although vertical scaling helps performance almost every time, it's not always possible or cost-effective. Using shards enables you to scale horizontally.

Suppose you want to scale your get-together index, which currently has two primary shards and no replicas. As shown in figure 7, the first option is to scale vertically by upgrading the node: for example, adding more RAM, more CPUs, faster disks and so on. The second option is to scale horizontally by adding another node and having your data distributed between the two nodes.

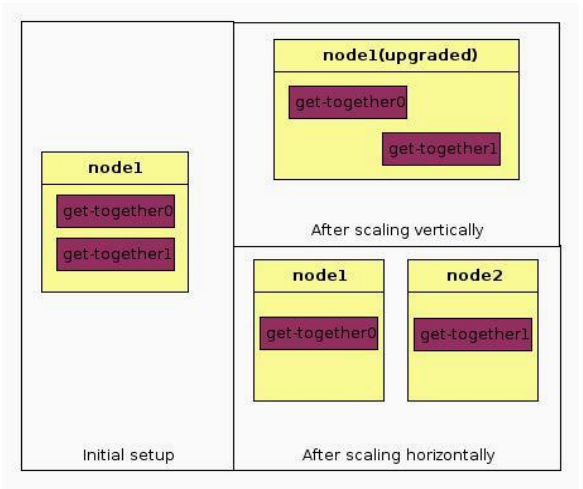


Figure 7 To improve performance, scale vertically (upper-right) or scale horizontally (lower-right).

Now let's see how indexing and searching work across multiple shards and replicas.

Distributed indexing and searching

At this point you might wonder how indexing and searching works with multiple shards spread across multiple nodes.

Let's take indexing, as shown in figure 8. The Elasticsearch node that receives your indexing request first selects the shard to index the document to. By default, documents are distributed evenly between shards: for each document, the shard is determined by hashing its ID string. Each shard has an equal hash range, with equal chances of receiving the new document. Once the target shard is determined, the current node forwards the document to

the node holding that shard. Subsequently, that indexing operation is replayed by all the replicas of that shard. The indexing command successfully returns after all the available replicas finish indexing the document.

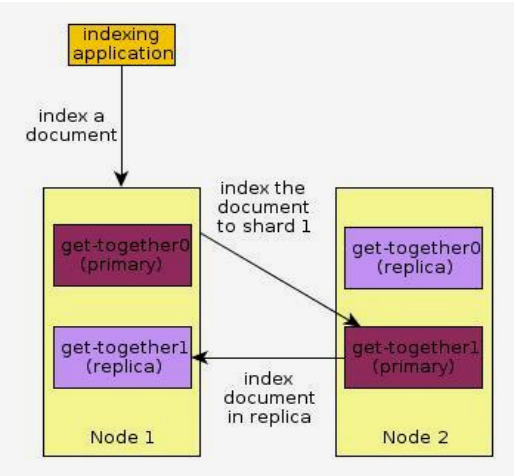


Figure 8 Indexing operation is forwarded to the responsible shard, and then to its replicas

With searching, the node that receives the request forwards it to a set of shards containing all your data. Using a round-robin, Elasticsearch selects an available shard (which can be primary or replica) and forwards the search request to it. As shown in figure 9, Elasticsearch then gathers results from those shards, aggregates them into a single reply, and forwards the reply back to the client application.

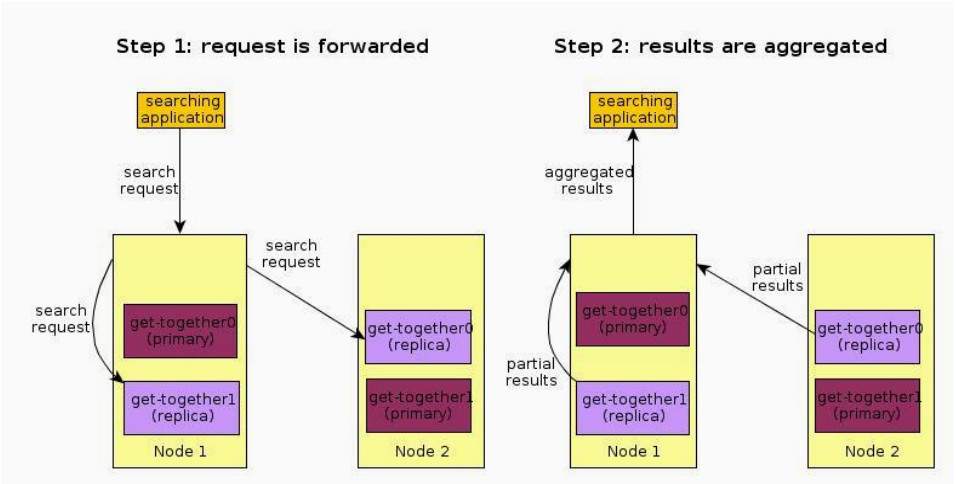


Figure 9 Search request is forwarded to primary/replica shards containing a complete set of data. Then, results are aggregated and sent back to the client.

By default, primary and replica shards get hit by searches in round-robin, assuming all nodes in your cluster are equally fast(identical hardware and software configurations). If that's not the case, you can organize your data or configure your shards to prevent the slower nodes from becoming a bottleneck.