CHAPTER 7
Scatter/Gather

So far we've examined systems that replicate for scalability in terms of the number of requests processed per second (the stateless replicated pattern), as well as scalability for the size of the data (the sharded data pattern). In this chapter we introduce the *scatter/gather* pattern, which uses replication for scalability in terms of time. Specifically, the scatter/gather pattern allows you to achieve parallelism in servicing requests, enabling you to service them significantly faster than you could if you had to service them sequentially.

Like replicated and sharded systems, the scatter/gather pattern is a tree pattern with a root that distributes requests and leaves that process those requests. However, in contrast to replicated and sharded systems, with scatter/gather requests are simultaneously farmed out to all of the replicas in the system. Each replica does a small amount of processing and then returns a fraction of the result to the root. The root server then combines the various partial results together to form a single complete response to the request and then sends this request back out to the client. The scatter/gather pattern is illustrated in Figure 7-1.

Scatter/gather is quite useful when you have a large amount of mostly independent processing that is needed to handle a particular request. Scatter/gather can be seen as sharding the computation necessary to service the request, rather than sharding the data (although data sharding may be part of it as well).

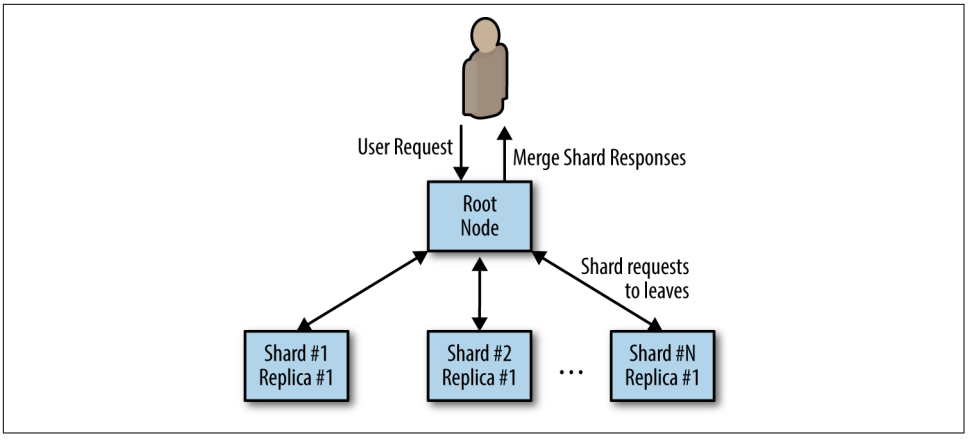

Figure 7-1. A scatter/gather pattern

Scatter/Gather with Root Distribution

The simplest form of scatter/gather is one in which each leaf is entirely homogenous but the work is distributed to a number of different leaves in order to improve the performance of the request. This pattern is equivalent to solving an "embarassingly parallel" problem. The problem can be broken up into many different pieces and each piece can be put back together with all of the other pieces to form a complete answer.

To understand this in more concrete terms, imagine that you need to service a user request *R* and it takes one minute for a single core to produce the answer *A* to this request. If we program a multi-threaded application, we can parallelize this request on a single machine by using multiple cores. Given this approach and a 30 core processor (yes, typically it would be a 32 core processor, but 30 makes the math cleaner), we can reduce the time that it takes to process a single request down to 2 seconds (60 seconds of computation split across 30 threads for computation is equal to 2 seconds). But even two seconds is pretty slow to service a user's web request. Additionally, truly achieving a completely parallel speed up on a single process is going to be tricky as things like memory, network, or disk bandwidth start to become the bottleneck. Instead of parallelizing an application across cores on a single machine, we can use the scatter/gather pattern to parallelize requests across multiple processes on many different machines. In this way, we can improve our overall latency requests, since we are no longer bound by the number of cores we can get on a single machine, as well as ensure that the bottleneck in our process continues to be CPU, since the memory, network, and disk bandwidth are all spread across a number of different machines. Additionally, because every machine in the scatter/gather tree is capable of handling every request, the root of the tree can dynamically dispatch load to different nodes at different times depending on their responsiveness. If, for some reason, a particular leaf node is responding more slowly than other machines (e.g., it has a noisy

neighbor process that is interfering with resources), then the root can dynamically redistribute load to assure a fast response.

Hands On: Distributed Document Search

To see an example of scatter/gather in action, consider the task of searching across a large database of documents for all documents that contain the words "cat" and "dog." One way to perform this search would be to open up all of the documents, read through the entire set, searching for the words in each document, and then return to the user the set of documents that contain both words.

As you might imagine, this is quite a slow process because it requires opening and reading through a large number of files for each request. To make request processing faster, you can build an *index*. The index is effectively a hashtable, where the keys are individual words (e.g., "cat") and the values are a list of documents containing that word.

Now, instead of searching through every document, finding the documents that match any one word is as easy as doing a lookup in this hashtable. However, we have lost one important ability. Remember that we were looking for all documents that contained "cat" *and* "dog." Since the index only has single words, not conjunctions of words, we still need to find the documents that contain both words. Luckily, this is just an intersection of the sets of documents returned for each word.

Given this approach, we can implement this document search as an example of the scatter/gather pattern. When a request comes in to the document search root, it parses the request and farms out two leaf machines (one for the word "cat" and one for the word "dog"). Each of these machines returns a list of documents that match one of the words, and the root node returns the list of documents containing both "cat" and "dog."

A diagram of this process is shown in Figure 7-2: the leaf returns `{doc1, doc2, doc4}` for "cat" and `{doc1, doc3, doc4}` for "dog," so the root finds the intersection and returns `{doc1, doc4}`.

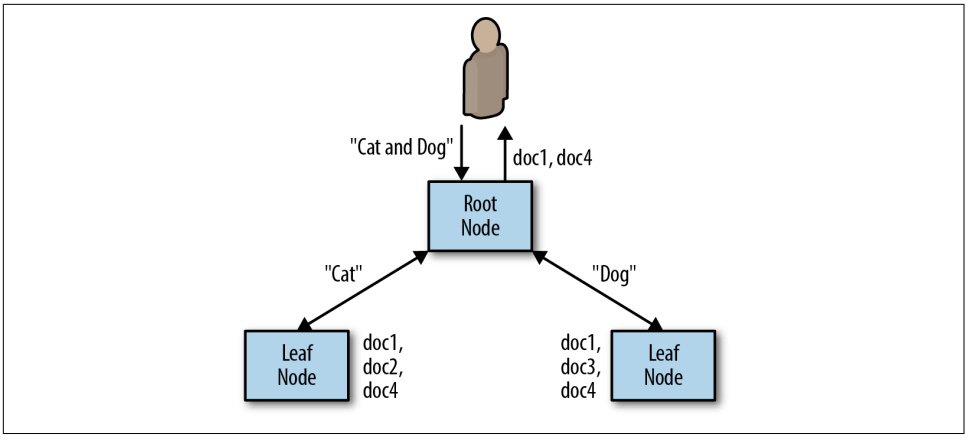

Figure 7-2. Example of a term-sharded scatter/gather system

Scatter/Gather with Leaf Sharding

While applying the replicated data scatter/gather pattern allows you to reduce the processing time required for handling user requests, it doesn't allow you to scale beyond an amount of data that can be held in the memory or disk of a single machine. Much like the replicated serving pattern that was previously described, it is simple to build a replicated scatter/gather system. But at a certain data size, it is necessary to introduce sharding in order to build a system that can hold more data than can be stored on a single machine.

Previously, when sharding was introduced to scale replicated systems, the sharding was done at a per-request level. Some part of the request was used to determine where the request was sent. That replica then handled all of the processing for the request and the response was handed back to the user. Instead, with scatter/gather sharding, the request is sent to all of the leaf nodes (or shards) in the system. Each leaf node processes the request using the data that it has loaded in its shard. This partial response is then returned to the root node that requested data, and that root node merges all of the responses together to form a comprehensive response for the user.

As a concrete example of this sort of architecture, consider implementing search across a very large document set (all patents in the world, for example); in such a case, the data is too large to fit in the memory of a single machine, so instead the data is sharded across multiple replicas. For example, patents 0-100,000 might be on the first machine, 100,001-200,000 on the next machine, and so forth. (Note that this is not actually a good sharding scheme since it will continually force us to add new shards as new patents are registered. In practice, we'd likely use the patent number modulo the total number of shards.) When a user submits a request to find a particular word (e.g., "rockets") in all of the patents in the index, that request is sent to each

shard, which searches through it's patent shard for patents which match the word in the query. Any matches that are found are returned to the root node in response to the shard request. The root node then collates all of these responses together into a single response that contains all the patents that match the particular word. The operation of this search index is illustrated in Figure 7-3.

Hands On: Sharded Document Search

The previous example scattered the different term requests across the cluster, but this only works if all of the documents are present on all of the machines in the scatter/gather tree. If there is not enough room for all of the documents in all of the leaves in the tree, then sharding must be used to put different sets of documents onto different leaves.

This means that when a user makes a request for all documents that match the words "cat" and "dog," the request is actually sent out to every leaf in the scatter/gather system. Each leaf node returns the set of documents that it knows about that matches "cat" and "dog." Previously, the root node was responsible for performing the intersection of the two sets of documents returned for two different words. In the sharded case, the root node is responsible for generating the union of all of the documents returned by all of the different shards and returning this complete set of documents back up to the user.

In Figure 7-3, the first leaf serves documents 1 through 10 and returns `{doc1, doc5}`. The second leaf serves documents 11 through 20 and returns `{doc15}`. The third leaf serves documents 21 through 30 and returns `{doc22, doc28}`. The root combines all of these responses together into a single response and returns `{doc1, doc5, doc15, doc22, doc28}`.

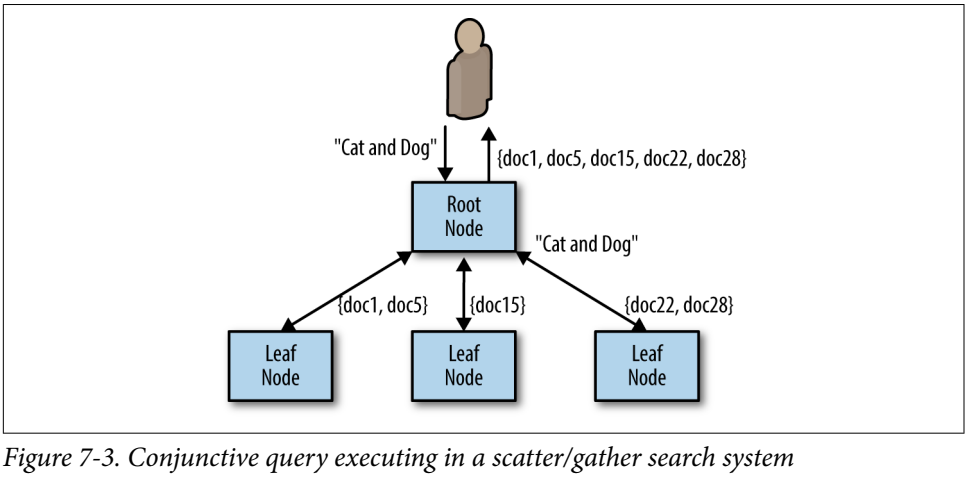

Figure 7-3. Conjunctive query executing in a scatter/gather search system

Choosing the Right Number of Leaves

It might seem that in the scatter/gather pattern, replicating out to a very large number of leaves would always be a good idea. You parallelize your computation and consequently reduce the clock time required to process any particular request. However, increased parallelization comes at a cost, and thus choosing the right number of leaf nodes in the scatter/gather pattern is critical to designing a performant distributed system.

To understand how this can happen, it's worth considering two things. The first is that processing any particular request has a certain amount of overhead. This is the time spent parsing a request, sending HTTP across the wire, and so forth. In general, the overhead due to system request handling is constant and significantly less than the time spent in user code processing the request. Consequently, this overhead can generally be ignored when assessing the performance of the scatter/gather pattern. However, it is important to understand that the cost of this overhead scales with the number of leaf nodes in the scatter/gather pattern. Thus, even though it is low cost, as parallelization continues, this overhead eventually dominates the compute cost of your business logic. This means that the gains of parallelization are asymptotic.

In addition to the fact that adding more leaf nodes may not actually speed up processing, scatter/gather systems also suffer from the "straggler" problem. To understand how this works, it is important to remember that in a scatter/gather system, the root node waits for requests from *all* of the leaf nodes to return before sending a response back to the end user. Since data from every leaf node is required, the overall time it takes to process a user request is defined by the slowest leaf node that sends a response. To understand the impact of this, imagine that we have a service that has a 99th percentile latency of 2 seconds. This means that on average one request out of every 100 has a latency of 2 seconds, or put another way, there is a 1% chance that a request will take 2 seconds. This may be totally acceptable at first glance: a single user out of 100 has a slow request. However, consider how this actually works in a scatter/gather system. Since the time of the user request is defined by the slowest response, we need to consider not a single request but all requests scattered out to the various leaf nodes.

Let's see what happens when we scatter out to five leaf nodes. In this situation, there is a 5% chance that one of these five scatter requests has a latency of 2 seconds ($0.99 \times 0.99 \times 0.99 \times 0.99 \times 0.99 == 0.95$). This means that our 99th percentile latency for individual requests becomes a 95th percentile latency for our complete scatter/gather system. And it only gets worse from there: if we scatter out to 100 leaves, then we are more or less guaranteeing that our overall latency for *all* requests will be 2 seconds.

Together, these complications of scatter/gather systems lead us to some conclusions:

- Increased parallelism doesn't always speed things up because of overhead on each node.
- Increased parallelism doesn't always speed things up because of the straggler problem.
- The performance of the 99th percentile is more important than in other systems because each user request actually becomes numerous requests to the service.

The same straggler problem applies to availability. If you issue a request to 100 leaf nodes, and the probability that any leaf node failing is 1 in 100, you are again practically guaranteed to fail every single user request.

Scaling Scatter/Gather for Reliability and Scale

Of course, just as with a sharded system, having a single replica of a sharded scatter/gather system is likely not the desirable design choice. A single replica means that if it fails, all scatter/gather requests will fail for the duration that the shard is unavailable because all requests are required to be processed by all leaf nodes in the scatter/gather pattern. Likewise, upgrades will take out a percentage of your shards, so an upgrade while under user-facing load is no longer possible. Finally, the computational scale of your system will be limited by the load that any single node is capable of achieving. Ultimately, this limits your scale, and as we have seen in previous sections, you cannot simply increase the number of shards in order to improve the computational power of a scatter/gather pattern.

Given these challenges of reliability and scale, the correct approach is to replicate each of the individual shards so that instead of a single instance at each leaf node, there is a replicated service that implements each leaf shard. This replicated, sharded scatter/gather pattern is shown in Figure 7-4.

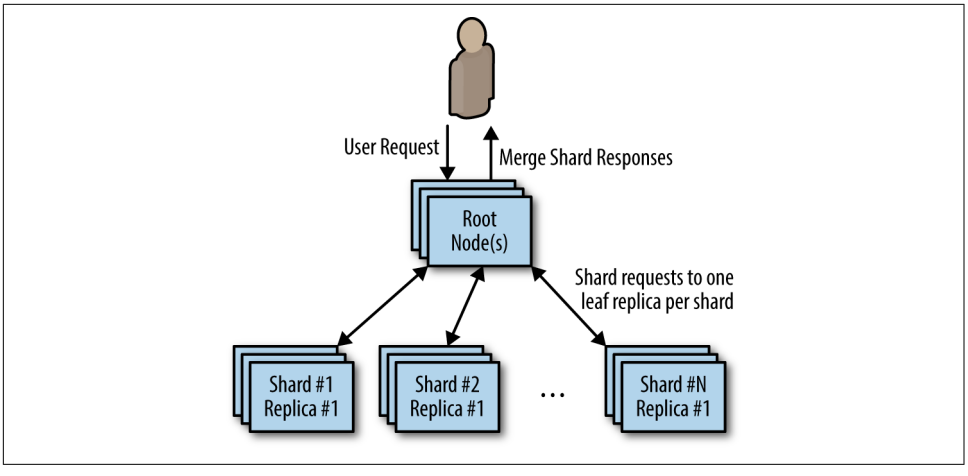

Figure 7-4. A sharded, replicated scatter/gatther system

Built this way, each leaf request from the root is actually load balanced across all healthy replicas of the shard. This means that if there are any failures, they won't result in a user-visible outage for your system. Likewise, you can safely perform an upgrade under load, since each replicated shard can be upgraded one replica at a time. Indeed, you can perform the upgrade across multiple shards simultaneously, depending on how quickly you want to perform the upgrade.