14 Parallel Information Retrieval

Information retrieval systems often have to deal with very large amounts of data. They must be able to process many gigabytes or even terabytes of text, and to build and maintain an index for millions of documents. To some extent the techniques discussed in Chapters 5–8 can help us satisfy these requirements, but it is clear that, at some point, sophisticated data structures and clever optimizations alone are not sufficient anymore. A single computer simply does not have the computational power or the storage capabilities required for indexing even a small fraction of the World Wide Web.[1]

In this chapter we examine various ways of making information retrieval systems scale to very large text collections such as the Web. The first part (Section 14.1) is concerned with parallel query processing, where the search engine's service rate is increased by having multiple index servers process incoming queries in parallel. It also discusses redundancy and fault tolerance issues in distributed search engines. In the second second part (Section 14.2), we shift our attention to the parallel execution of off-line tasks, such as index construction and statistical analysis of a corpus of text. We explain the basics of MapReduce, a framework designed for massively parallel computations carried out on large amounts of data.

14.1 Parallel Query Processing

There are many ways in which parallelism can help a search engine process queries faster. The two most popular approaches are *index partitioning* and *replication*. Suppose we have a total of n index servers. Following the standard terminology, we refer to these servers as *nodes*. By creating n replicas of the index and assigning each replica to a separate node, we can realize an n-fold increase of the search engine's service rate (its theoretical throughput) without affecting the time required to process a single query. This type of parallelism is referred to as *inter-query parallelism*, because multiple queries can be processed in parallel but each individual query is processed sequentially. Alternatively, we could split the index into n parts and have each node work only on its own small part of the index. This approach is referred to as *intra-query*

[1] While nobody knows the exact size of the indexable part of the Web, it is estimated to be at least 100 billion pages. In August 2005, when Yahoo! last disclosed the size of its index, it had reached a total size of 19.2 billion documents (http://www.ysearchblog.com/archives/000172.html).

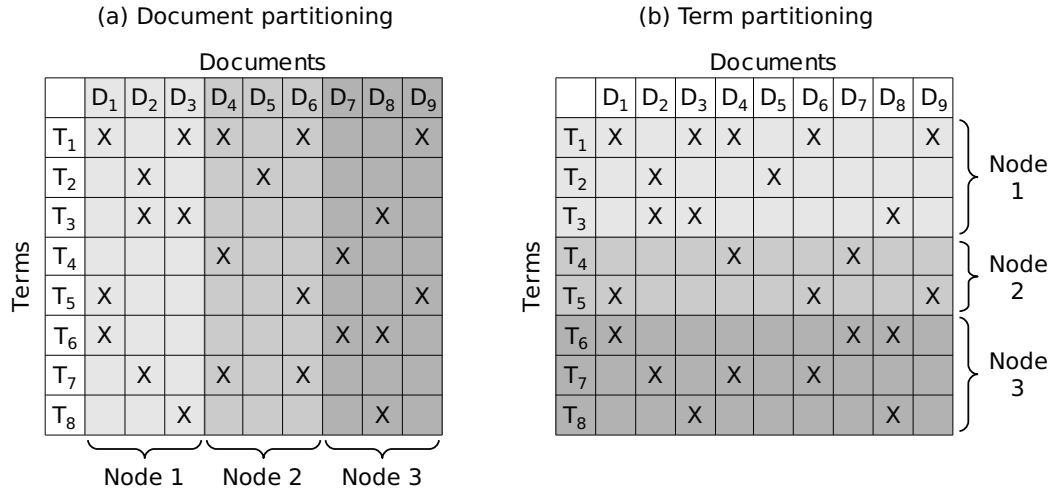

Figure 14.1 The two prevalent index partitioning schemes: document partitioning and term partitioning (shown for a hypothetical index containing 8 terms and 9 documents).

parallelism, because each query is processed by multiple servers in parallel. It improves the engine's service rate as well as the average time per query.

In this section we focus primarily on methods for intra-query parallelism. We study index partitioning schemes that divide the index into independent parts so that each node is responsible for a small piece of the overall index.

The two predominant index partitioning schemes are *document partitioning* and *term partitioning* (visualized in Figure 14.1). In a document-partitioned index, each node holds an index for a subset of the documents in the collection. For instance, the index maintained by node 2 in Figure 14.1(a) contains the following docid lists:

$$L_1 = \langle 4, 6 \rangle, \quad L_2 = \langle 5 \rangle, \quad L_4 = \langle 4 \rangle, \; L_5 = \langle 6 \rangle, \quad L_7 = \langle 4, 6 \rangle.$$

In a term-partitioned index, each node is responsible for a subset of the terms in the collection. The index stored in node 1 in Figure 14.1(b) contains the following lists:

$$L_1 = \langle 1, 3, 4, 6, 9 \rangle, \quad L_2 = \langle 2, 5 \rangle, \quad L_3 = \langle 2, 3, 8 \rangle.$$

The two partitioning strategies differ greatly in the way queries are processed by the system. In a document-partitioned search engine, each of the n nodes is involved in processing all queries received by the engine. In a term-partitioned configuration, a query is seen by a given node only if the node's index contains at least one of the query terms.

14.1.1 Document Partitioning

In a document-partitioned index, each index server is responsible for a subset of the documents in the collection. Each incoming user query is received by a frontend server, the *receptionist*, which forwards it to all n index nodes, waits for them to process the query, merges the search results received from the index nodes, and sends the final list of results to the user. A schematic of a document-partitioned search engine is shown in Figure 14.2.

The main advantage of the document-partitioned approach is its simplicity. Because all index servers operate independently of each other, no additional complexity needs to be introduced into the low-level query processing routines. All that needs to be provided is the receptionist server that forwards the query to the backends and, after receiving the top k search results from each of the n nodes, selects the top m, which are then returned to the user (where the value of m is typically chosen by the user and k is chosen by the operator of the search engine). In addition to forwarding queries and search results, the receptionist may also maintain a cache that contains the results for recently/frequently issued queries.

If the search engine maintains a dynamic index that allows updates (e.g., document insertions/deletions), then it may even be possible to carry out the updates in a distributed fashion, in which each node takes care of the updates that pertain to its part of the overall index. This approach eliminates the need for a complicated centralized index construction/maintenance process that involves the whole index. However, it is applicable only if documents may be assumed to be independent of each other, not if inter-document information, such as hyperlinks and anchor text, is part of the index.

When deciding how to divide the collection across the n index nodes, one might be tempted to bias the document-node assignment in some way — for instance, by storing similar documents in the same node. In Section 6.3.7 we have seen that the index can be compressed better if the documents in the collection are reordered according to their URL, so that documents with similar URLs receive nearby docids. Obviously, this method is most effective if all pages from the same domain are assigned to the same node. The problem with this approach is that it may create an imbalance in the load distribution of the index servers. If a given node primarily contains documents associated with a certain topic, and this topic suddenly becomes very popular among users, then the query processing load for that node may become much higher than the load of the other nodes. In the end this will lead to a suboptimal utilization of the available resources. To avoid this problem, it is usually best to not try anything fancy but to simply split the collection into n completely random subsets.

Once the index has been partitioned and each subindex has been loaded into one of the nodes, we have to make a decision regarding the per-node result set size k. How should k be chosen with respect to m, the number of search results requested by the user? Suppose the user has asked for $m = 100$ results. If we want to be on the safe side, we can have each index node return $k = 100$ results, thus making sure that all of the top 100 results overall are received by the receptionist. However, this would be a poor decision, for two reasons:

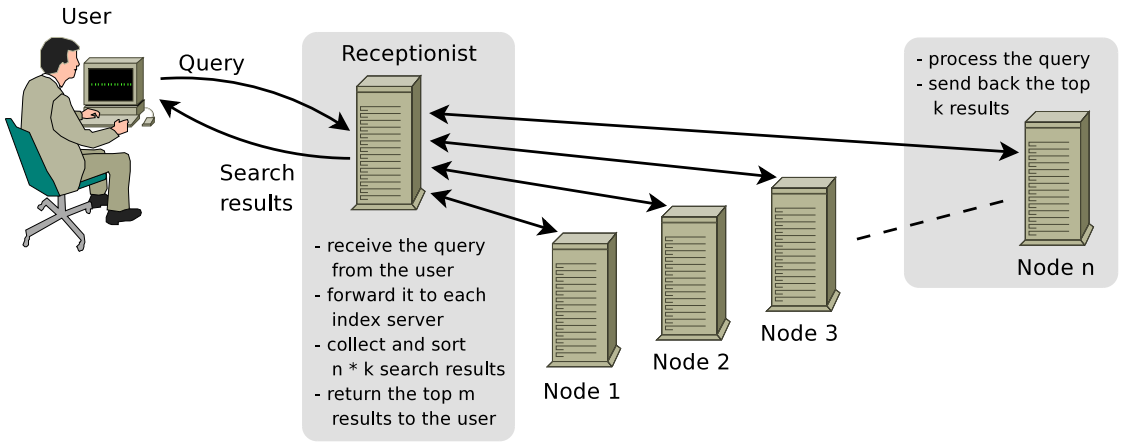

Figure 14.2 Document-partitioned query processing with one receptionist and several index servers. Each incoming query is forwarded to all index servers.

1. It is quite unlikely that all of the top 100 results come from the same node. By having each index server return 100 results to the receptionist, we put more load on the network than necessary.
2. The choice of k has a non-negligible effect on query processing performance, because of performance heuristics such as MaxScore (see Section 5.1.1). Table 5.1 on page 144 shows that decreasing the result set size from $k = 100$ to $k = 10$ can reduce the average CPU time per query by around 15%, when running queries against a frequency index for GOV2.

Clarke and Terra (2004) describe a method to compute the probability that the receptionist sees at least the top m results overall, given the number n of index servers and the per-node result set size k. Their approach is based on the assumption that each document was assigned to a random index node when the collection was split into n subsets, and thus that each node is equally likely to return the best, second-best, third-best, ... result overall.

Consider the set $\mathcal{R}_m = \{r_1, r_2, \ldots, r_m\}$ composed of the top m search results. For each document r_i the probability that it is found by a particular node is $1/n$. Hence, the probability that exactly l of the top m results are found by that node is given by the binomial distribution

$$b(n, m, l) \;=\; \binom{m}{l} \cdot \left(\frac{1}{n}\right)^{l} \cdot \left(1 - \frac{1}{n}\right)^{m-l}. \tag{14.1}$$

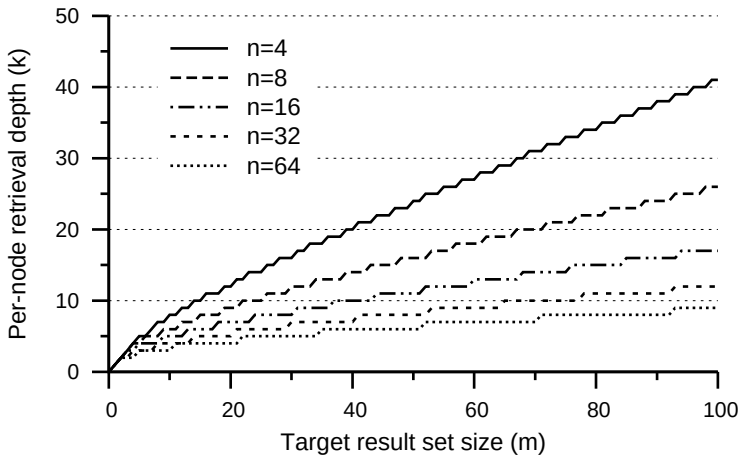

Figure 14.3 Choosing the minimum retrieval depth k that returns the top m results with probability $p(n, m, k) > 99.9\%$, where n is the number of nodes in the document-partitioned index.

The probability that all members of $\mathcal{R}_m$ are discovered by requesting the top k results from each of the n index nodes can be calculated according to the following recursive formula:

$$p(n,m,k) = \begin{cases} 1 & \text{if } m \leq k; \\ 0 & \text{if } m > k \text{ and } n = 1; \\ \sum_{l=0}^{k} b(n,m,l) \cdot p(n-1, m-l, k) & \text{if } m > k \text{ and } n > 1. \end{cases} \tag{14.2}$$

The two base cases are obvious. In the recursive case, we consider the first node in the system and compute the probability that $l = 0, 1, 2, \ldots, k$ of the top m results overall are retrieved by that node (i.e., $b(n, m, l)$). For each possible value for l, this probability is multiplied by the probability $p(n-1, m-l, k)$ that the remaining $m-l$ documents in $\mathcal{R}_m$ are found by the remaining $n-1$ index nodes. Equation 14.2 does not appear to have a closed-form solution but can be solved through the application of dynamic programming in time $\Theta(n \cdot m \cdot k)$.

Figure 14.3 shows the per-node retrieval depth k required to find the top m results with probability at least 99.9%. For $m = 100$ and $n = 4$, a per-node retrieval depth of $k = 41$ achieves the desired probability level. If we decide to relax our correctness requirements from 99.9% to 95%, k can even be decreased a little further, from 41 to 35.

It can sometimes be beneficial to optimize the retrieval depth k for a different value m than the user has asked for. For example, if the receptionist maintains a cache for recently issued queries, it may be worthwhile to obtain the top 20 results for a given query even if the user has asked for only the top 10, so that a click on the "next page" link can be processed from the cache. Allowing the receptionist to inspect a result set $\mathcal{R}_{m'}$ for $m' > m$ is also useful because

it facilitates the application of diversity-seeking reranking techniques, such as Carbonell and Goldstein's (1998) *maximal marginal relevance* or Google's *host crowding* heuristic.[2]

14.1.2 Term Partitioning

Although document partitioning is often the right choice and scales almost linearly with the number of nodes, it can unfold its true potential only if the index data found in the individual nodes are stored in main memory or any other low-latency random-access storage medium, such as flash memory, but not if it is stored on disk.

Consider a document-partitioned search engine in which all postings lists are stored on disk. Suppose each query contains 3 words on average, and we want the search engine to handle a peak query load of 100 queries per second. Recall from Section 13.2.3 that, due to queueing effects, we usually cannot sustain a utilization level above 50%, unless we are willing to accept occasional latency jumps. Thus, a query load of 100 qps translates into a required service rate of at least 200 qps or — equivalently — 600 random access operations per second (one for each query term). Assuming an average disk seek latency of 10 ms, a single hard disk drive cannot perform more than 100 random access operations per second, a factor of 6 less than what is required to achieve the desired throughput. We could try to circumvent this limitation by adding more disks to each index node, but equipping each server with six hard disks may not always be practical. Moreover, it is obvious that we will never be able to handle loads of more than a few hundred queries per second, regardless of how many nodes we add to the system.

Term partitioning addresses the disk seek problem by splitting the collection into sets of terms instead of sets of documents. Each index node v_i is responsible for a certain term set $\mathcal{T}_i$ and is involved in processing a given query only if one or more query terms are members of $\mathcal{T}_i$. Our discussion of term-partitioned query processing is based upon the pipelined architecture proposed by Moffat et al. (2007). In this architecture, queries are processed in a term-at-a-time fashion (see Section 5.1.2 for details on term-at-a-time query processing strategies).

Suppose a query contains q terms $t_1, t_2, \ldots, t_q$. Then the receptionist will forward the query to the node $v(t_1)$ responsible for the term t_1. After creating a set of document score accumulators from t_1's postings list, $v(t_1)$ forwards the query, along with the accumulator set, to the node $v(t_2)$ responsible for t_2, which updates the accumulator set, sends it to $v(t_3)$, and so forth. When the last node in this pipeline, $v(t_q)$, is finished, it sends the final accumulator set to the receptionist. The receptionist then selects the top m search results and returns them to the user. A schematic of this approach is shown in Figure 14.4.

Many of the optimizations used for sequential term-at-a-time query processing also apply to term-partitioned query processing: infrequent query terms should be processed first; accumulator pruning strategies should be applied to keep the size of the accumulator set, and thus the

[2] `www.mattcutts.com/blog/subdomains-and-subdirectories/`

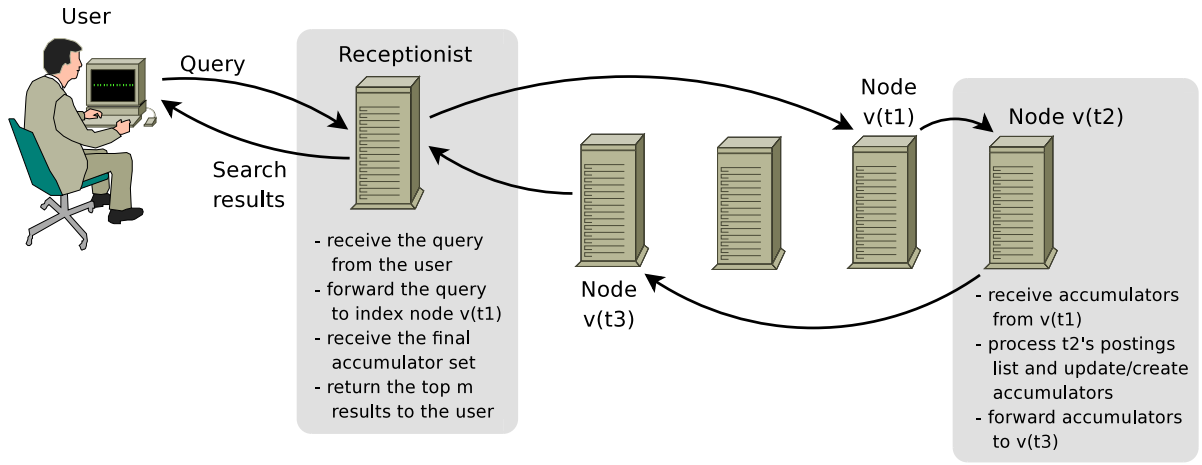

Figure 14.4 Term-partitioned query processing with one receptionist and four index servers. Each incoming query is passed from index server to index server, depending on the terms found in the query (shown for a query containing three terms).

overall network traffic, under control; impact ordering can be used to efficiently identify the most important postings for a given term.

Note that the pipelined query processing architecture outlined above does not use intra-query parallelism, as each query — at a given point in time — is processed by only a single node. Therefore, although it can help increase the system's theoretical throughput, term partitioning, in the simple form described here, does not necessarily decrease the search engine's response time. To some extent this limitation can be addressed by having the receptionist send prefetch instructions to the nodes $v(t_2), \ldots, v(t_q)$ at the same time it forwards the query to $v(t_1)$. This way, when a node $v(t_i)$ receives the accumulator set, some of t_i's postings may already have been loaded into memory and the query can be processed faster.

Despite its potential performance advantage over the document-partitioned approach, at least for on-disk indices, term partitioning has several shortcomings that make it difficult to use the method in practice:

- **Scalability.** As the collection becomes bigger, so do the individual postings lists. For a corpus composed of a billion documents, the postings list for a frequent term, such as "computer" or "internet", can easily occupy several hundred megabytes. Processing a query that contains one or more of these frequent terms will require at least a few seconds, far more than what most users are accustomed to. In order to solve this problem, large postings lists need to be cut into smaller chunks and divided among several nodes, with each node taking care of a small part of the postings list and all of them working in parallel. Unfortunately, this complicates the query processing logic quite a bit.

- **Load Imbalance.** Term partitioning suffers from an uneven load across the index nodes. The load corresponding to a single term is a function of the term's frequency in the collection as well as its frequency in users' queries. If a term has a long postings list and is also very popular in search queries, then the corresponding index node may experience a load that is much higher than the average load in the system. To address this problem, postings lists that are responsible for a high fraction of the overall load should be replicated and distributed across multiple nodes. This way, the computational load associated with a given term can be shared among several machines, albeit at the cost of increased storage requirements.
- **Term-at-a-Time.** Perhaps the most severe limitation of the term-partitioned approach is its inability to support efficient document-at-a-time query processing. In order to realize document-at-a-time scoring on top of a term-partitioned index, entire postings lists, as opposed to pruned accumulator sets, would need to be sent across the network. This is impractical, due to the size of the postings lists. Therefore, ranking methods that necessitate a document-at-a-time approach, such as the proximity ranking function from Section 2.2.2, are incompatible with a term-partitioned index.

Even with all these shortcomings, term partitioning can sometimes be the right choice. Recall, for instance, the three-level cache hierarchy from Section 13.4.1. The second level in this hierarchy caches list intersections (in search engines with Boolean-AND query semantics; see Section 2.2). Instead of following a document partitioning approach and equipping each index node with its own intersection cache, we may choose a term-partitioned index and treat the cached intersections just like ordinary postings lists. In the example shown in Figure 14.4, if we have already seen the query $\langle t_1, t_2 \rangle$ before, we may have cached the intersected list $(t_1 \wedge t_2)$ in index node $v(t_2)$ and may skip $v(t_1)$ when processing the new query $\langle t_1, t_2, t_3 \rangle$.

More generally, term partitioning suggests itself if postings lists are relatively short — either due to the nature of the information they represent (as in the case of list intersections) or because they are artificially shortened (for instance, by applying index pruning techniques; see Section 5.1.5 for details).

14.1.3 Hybrid Schemes

Consider a distributed index with n nodes, for a collection of size $|\mathcal{C}|$. Document partitioning becomes inefficient if $|\mathcal{C}|/n$ is too small and disk seeks dominate the overall query processing cost. Term partitioning, on the other hand, becomes impractical if $|\mathcal{C}|$ is too large, as the time required to process a single query is likely to exceed the users' patience.

Xi et al. (2002) propose a hybrid architecture in which the collection is divided into p subcollections according to a standard document partitioning scheme. The index for each subcollection is then term-partitioned across n/p nodes, so that each node in the system is responsible for

all occurrences of a set of terms within one of the p subcollections. With the right load balancing policies in place, this can lead roughly to a factor-n increase in throughput and a factor-p latency reduction.

As an alternative to the hybrid term/document partitioning, we may also consider a hybrid of document partitioning and replication. Remember that the primary objective of term partitioning is to increase throughput, not to decrease latency. Thus, instead of term-partitioning each of the p subindices, we may achieve the same performance level by simply replicating each subindex n/p times and load-balancing the queries among n/p identical replicas. At a high level, this is the index layout that was used by Google around 2003 (Barroso et al., 2003). The overall impact on maximum throughput and average latency is approximately the same as in the case of the hybrid document/term partitioning. The storage requirements are likely to be a bit higher, due to the (n/p)-way replication. If the index is stored on disk, this is usually not a problem.

14.1.4 Redundancy and Fault Tolerance

When operating a large-scale search engine with thousands of users, reliability and fault tolerance tend to be of similar importance as response time and result quality. As we increase the number of machines in the search engine, so as to scale to higher query loads, it becomes more and more likely that one of them will fail at some point in time. If the system has been designed with fault tolerance in mind, then the failure of a single machine may cause a small reduction in throughput or search quality. If it has not been designed with fault tolerance in mind, a single failure may bring down the entire search engine.

Let us compare the simple, replication-free document partitioning and term partitioning schemes for a distributed search engine with 32 nodes. If one of the nodes in the term-partitioned index fails, the engine will no longer be able to process queries containing any of the terms managed by that node. Queries that do not contain any of those terms are unaffected. For a random query q containing three words (the average number of query terms for Web queries), the probability that q can be processed by the remaining 31 nodes is

$$\left(\frac{31}{32}\right)^3 \approx 90.9\%. \tag{14.3}$$

If one of the nodes in the document-partitioned index fails, on the other hand, the search engine will still be able to process all incoming queries. However, it will miss some search results. If the partitioning was performed bias-free, then the probability that j out of the top k results for a random query are missing is

$$\binom{k}{j} \cdot \left(\frac{1}{32}\right)^j \cdot \left(\frac{31}{32}\right)^{k-j}. \tag{14.4}$$

Thus, the probability that the top 10 results are unaffected by the failure is

$$\left(\frac{31}{32}\right)^{10} \approx 72.8\%. \tag{14.5}$$

It therefore might seem that the impact of the machine failure is lower for the term-partitioned than for the document-partitioned index. However, the comparison is somewhat unfair, because the inability to process a query is a more serious problem than losing one of the top 10 search results. If we look at the probability that at least 2 of the top 10 results are lost due to the missing index node, we obtain

$$1 - \left(\frac{31}{32}\right)^{10} - \binom{10}{1} \cdot \left(\frac{1}{32}\right) \cdot \left(\frac{31}{32}\right)^{9} \approx 3.7\%. \tag{14.6}$$

Thus, the probability that a query is impacted severely by a single node failure is in fact quite small, and most users are unlikely to notice the difference. Following this line of argument, we may say that a document-partitioned index degrades more gracefully than a term-partitioned one in the case of a single node failure.

For informational queries, where there are often multiple relevant results, this behavior might be good enough. For navigational queries, however, this is clearly not the case (see Section 15.2 for the difference between informational and navigational queries). As an example, consider the navigational query ⟨"white", "house", "website"⟩. This query has a single vital result (http://www.whitehouse.gov/) that *must* be present in the top search results. If 1 of the 32 nodes in the document-partitioned index fails, then for each navigational query there is a 3.2% chance that the query's vital result is lost (if there is a vital result for the query). There are many ways to address this problem. Three popular ones are the following:

- **Replication.** We can maintain multiple copies of the same index node, as described in the previous section on hybrid partitioning schemes, and have them process queries in parallel. If one of the r replicas for a given index node fails, the remaining $r-1$ will take over the load of the missing replica. The advantage of this approach is its simplicity (and its ability to improve throughput and fault tolerance at the same time). Its disadvantage is that, if the system is operating near its theoretical throughput and r is small, the remaining $r-1$ replicas may become overloaded.
- **Partial Replication.** Instead of replicating the entire index r times, we may choose to replicate index information only for important documents. The rationale behind this strategy is that most search results aren't vital and can easily be replaced by an equally relevant document. The downside of this approach is that it can be difficult to predict which documents may be targeted by navigational queries. Query-independent signals such as PageRank (Section 15.3.1) can provide some guidance.

- **Dormant Replication.** Suppose the search engine comprises a total of n nodes. We can divide the index found on each node v_i into $n-1$ fragments and distribute them evenly among the $n-1$ remaining nodes, but leave them dormant (on disk) and not use them for query processing. Only when v_i fails will the corresponding $n-1$ fragments be activated inside the remaining nodes and loaded into memory for query processing. It is important that the fragments are loaded into memory, for otherwise we will double the overall number of disk seeks per query. Dormant replication roughly causes a factor-2 storage overhead, because each of the n nodes has to store $n-1$ additional index fragments.

It is possible to combine the above strategies — for example, by employing dormant replication of partial indices. Instead of replicating the whole index found in a given node, we replicate only the part that corresponds to important documents. This reduces the storage overhead and limits the impact on the search engine's throughput in case of a node failure.

14.2 MapReduce

Apart from processing search queries, there are many other data-intensive tasks that need to be carried out by a large-scale search engine. Such tasks include building and updating the index; identifying duplicate documents in the corpus; and analyzing the link structure of the document collection (e.g., PageRank; see Section 15.3.1).

MapReduce is a framework developed at Google that is designed for massively parallel computations (thousands of machines) on very large amounts of data (many terabytes), and that can accomplish all of the tasks listed above. MapReduce was first presented by Dean and Ghemawat (2004). In addition to a high-level overview of the framework, their paper includes information about many interesting implementation details and performance optimizations.

14.2.1 The Basic Framework

MapReduce was inspired by the *map* and *reduce* functions found in functional programming languages, such as Lisp. The map function takes as its arguments a function f and a list of elements $l = \langle l_1, l_2, \ldots, l_n \rangle$. It returns a new list

$$map(f, l) \;=\; \langle\, f(l_1),\; f(l_2),\; \ldots,\; f(l_n)\,\rangle. \tag{14.7}$$

The reduce function (also known as *fold* or *accumulate*) takes a function g and a list of elements $l = \langle l_1, l_2, \ldots, l_n \rangle$. It returns a new element l', such that

$$l' \;=\; reduce(g, l) \;=\; g(l_1,\; g(l_2,\; g(l_3,\; \ldots))). \tag{14.8}$$

When people refer to the map function in the context of MapReduce, they usually mean the function f that gets passed to *map* (where *map* itself is provided by the framework). Similarly,

```
    map (k, v) ≡
1       split v into tokens
2       for each token t do
3           output(t, 1)
4       return

    reduce (k, ⟨v_1, v_2, ..., v_n⟩) ≡
5       count ← 0
6       for i ← 1 to n do
7           count ← count + v_i
8       output(count)
9       return
```

Figure 14.5 A MapReduce that counts the number of occurrences of each term in a given corpus of text. The input values processed by the map function are documents or other pieces of text. The input keys are ignored. The outcome of the MapReduce is a sequence of (t, f_t) tuples, where t is a term, and f_t is the number of times t appeared in the input.

when they refer to the reduce function, they mean the function g that gets passed to *reduce*. We will follow this convention.

From a high-level point of view, a MapReduce program (often simply called "a MapReduce") reads a sequence of key/value pairs, performs some computations on them, and outputs another sequence of key/value pairs. Keys and values are often strings, but may in fact be any data type. A MapReduce consists of three distinct phases:

- In the *map phase*, key/value pairs are read from the input and the map function is applied to each of them individually. The function is of the general form

$$map: \ (k, v) \ \mapsto \ \langle \, (k_1, v_1), \ (k_2, v_2), \ \ldots \rangle. \tag{14.9}$$

  That is, for each key/value pair, map outputs a sequence of key/value pairs. This sequence may or may not be empty, and the output keys may or may not be identical to the input key (they usually aren't).

- In the *shuffle phase*, the pairs produced during the map phase are sorted by their key, and all values for the same key are grouped together.

- In the *reduce phase*, the reduce function is applied to each key and its values. The function is of the form

$$reduce: \ (k, \ \langle v_1, v_2, \ldots \rangle) \ \mapsto \ (k, \ \langle v'_1, v'_2, \ldots \rangle). \tag{14.10}$$

  That is, for each key the reduce function processes the list of associated values and outputs another list of values. The output values may or may not be the same as the input values. The output key usually has to be the same as the input key, although this depends on the implementation.

Figure 14.5 shows the map and reduce functions of a MapReduce that counts the number of occurrences of all terms in a given corpus of text. In the reduce function, the output key is omitted, as it is implicit from the input key.

MapReduces are highly parallelizable, because both map and reduce can be executed in parallel on many different machines. Suppose we have a total of $n = m + r$ machines, where m is the number of *map workers* and r is the number of *reduce workers*. The input of the MapReduce is broken into small pieces called *map shards*. Each shard typically holds between 16 and 64 MB of data. The shards are treated independently, and each shard is assigned to one of the m map workers. In a large MapReduce, it is common to have dozens or hundreds of map shards assigned to each map worker. A worker usually works on only 1 shard at a time, so all its shards have to be processed sequentially. However, if a worker has more than 1 CPU, it may improve performance to have it work on multiple shards in parallel.

In a similar fashion, the output is broken into separate *reduce shards*, where the number of reduce shards is often the same as r, the number of reduce workers. Each key/value pair generated by the map function is sent to one of the r reduce shards. Typically, the shard that a given key/value pair is sent to depends only on the key. For instance, if we have r reduce shards, the target shard for each pair could be chosen according to

$$shard(key, value) = hash(key) \bmod r, \tag{14.11}$$

where *hash* is an arbitrary hash function. Assigning the map output to different reduce shards in this manner guarantees that all values for the same key end up in the same reduce shard. Within each reduce shard, incoming key/value pairs are sorted by their key (this is the shuffle phase), and are eventually fed into the reduce function to produce the final output of the MapReduce.

Figure 14.6 shows the data flow for the MapReduce from Figure 14.5, for three small text fragments from the Shakespeare corpus. Each fragment represents a separate map shard. The key/value pairs emitted by the map workers are partitioned onto the three reduce shards based on the hash of the respective key. For the purpose of the example, we assume $hash(\text{“heart”}) \bmod 3 = 0$, $hash(\text{“soul”}) \bmod 3 = 1$, and so forth.

The map phase may overlap with the shuffle phase, and the shuffle phase may overlap with the reduce phase. However, the map phase may never overlap with the reduce phase. The reason for this is that the reduce function can only be called after all values for a given key are available. Since, in general, it is impossible to predict what keys the map workers will emit, the reduce phase cannot commence before the map phase is finished.

14.2.2 Combiners

In many MapReduce jobs, a single map shard may produce a large number of key/value pairs for the same key. For instance, if we apply the counting MapReduce from Figure 14.5 to a typical corpus of English text, 6–7% of the map outputs will be for the key “the”. Forwarding all these tuples to the reduce worker responsible for the term “the” wastes network and storage resources. More important, however, it creates an unhealthy imbalance in the overall load distribution. Regardless of how many reduce workers we assign to the job, one of them will end up doing at least 7% of the overall reduce work.

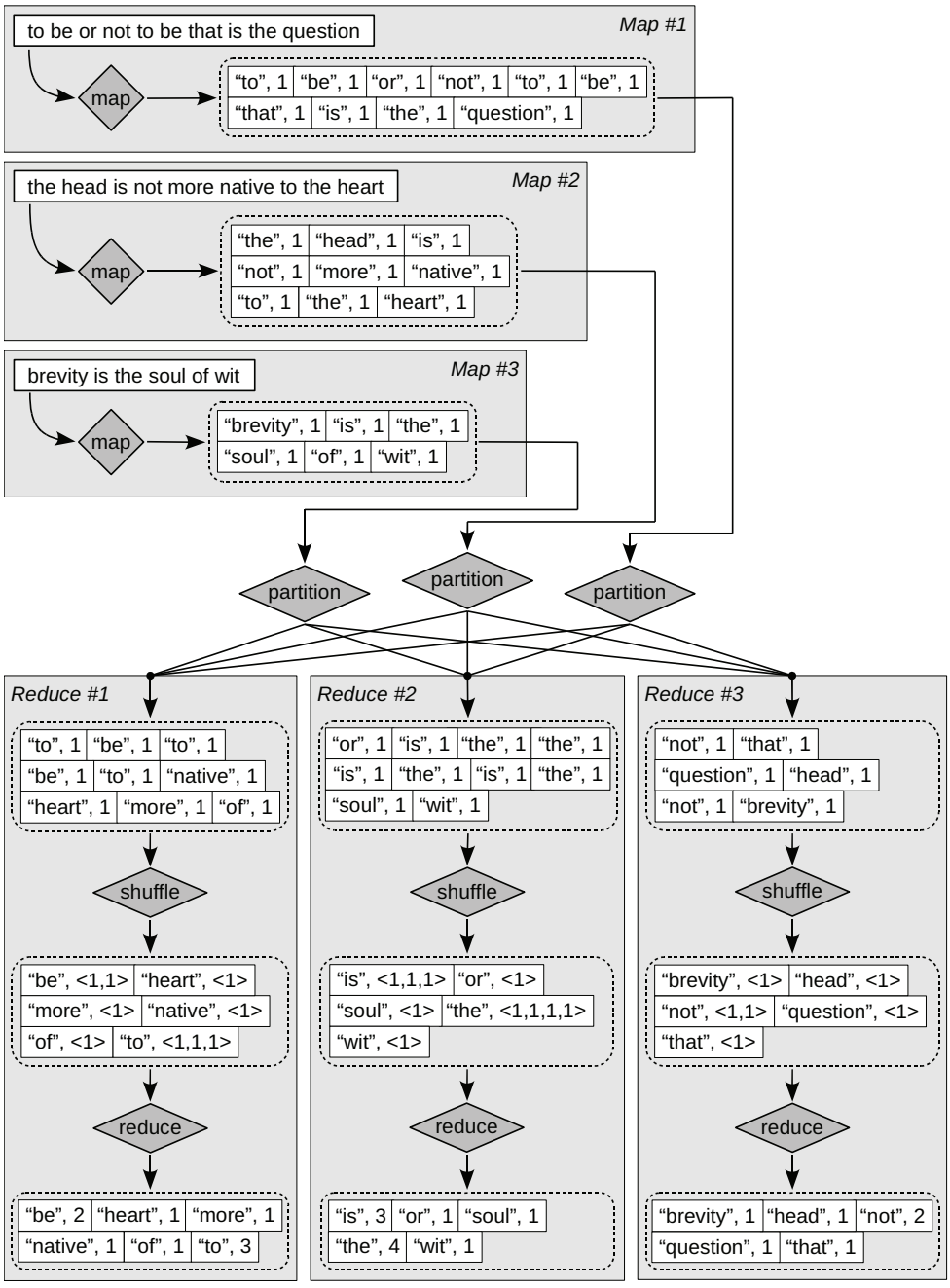

Figure 14.6 Data flow for the MapReduce definition shown in Figure 14.5, using 3 map shards and 3 reduce shards.

To overcome this problem, we could modify the map workers so that they accumulate per-shard term counts in a local hash table and output one pair of the form (t, f_t) instead of f_t pairs of the form $(t, 1)$ when they are finished with the current shard. This approach has the disadvantage that it requires extra implementation effort by the programmer.

As an alternative to the hash table method, it is possible to perform a local shuffle/reduce phase for each map shard before forwarding any data to the reduce workers. This approach has about the same performance effect as accumulating local counts in a hash table, but is often preferred by developers because it does not require any changes to their implementation. A reduce function that is applied to a map shard instead of a reduce shard is called a *combiner*. Every reduce function can serve as a combiner as long as its input values are of the same type as its output values, so that it can be applied to its own output. In the case of the counting MapReduce, this requirement is met, as all input and output values in the reduce phase are integers.

14.2.3 Secondary Keys

The basic MapReduce framework, as described so far, does not make any guarantees regarding the relative order in which values for the same key are fed into the reduce function. Often this is not a problem, because the reduce function can inspect and reorder the values in any way it wishes. However, for certain tasks the number of values for a given key may be so large that they cannot all be loaded into memory at the same time, thus making reordering inside the reduce function difficult. In that situation it helps to have the MapReduce framework sort each key's values in a certain way before they are passed to the reduce function.

An example of a task in which it is imperative that values arrive at the reduce function in a certain, predefined order is index construction. To create a docid index for a given text collection, we might define a map function that, for each term t encountered in a document d, outputs the key/value pair $(t, \mathit{docid}(d))$. The reduce function then builds t's postings list by concatenating all its postings (and potentially compressing them). Obviously, for this to be possible, the reduce input has to arrive in increasing order of $\mathit{docid}(d)$.

MapReduce supports the concept of *secondary keys* that can be used to define the order in which values for the same key arrive at the reduce function. In the shuffle phase, key/value pairs are sorted by their key, as usual. However, if there is more than one value for a given key, the key's values are sorted according to their secondary key. In the index construction MapReduce, the secondary key of a key/value pair would be the same as the pair's value (i.e., the docid of the document that contains the given term).

14.2.4 Machine Failures

When running a MapReduce that spans across hundreds or thousands of computers, some of the machines occasionally experience problems and have to be shut down. The MapReduce framework assumes that the map function behaves strictly deterministically, and that the map

output for a given map shard depends only on that one shard (i.e., no information may be exchanged between two shards processed by the same map worker). If this assumption holds, then a map worker failure can be dealt with by assigning its shards to a different machine and reprocessing them.

Dealing with a reduce worker failure is slightly more complicated. Because the data in each reduce shard may depend on data in every map shard, assigning the reduce shard to a different worker may necessitate the re-execution of all map shards. In order to avoid this, the output of the map phase is usually not sent directly to the reduce workers but is temporarily stored in a reliable storage layer, such as a dedicated storage server or the Google file system (Ghemawat et al., 2003), from where it can be read by the new reduce worker in case of a worker failure. However, even if the map output is stored in a reliable fashion, a reduce worker failure will still require the re-execution of the shuffle phase for the failed shard. If we want to avoid this, too, then the reduce worker needs to send the output of the shuffle phase back to the storage server before it enters the reduce phase. Because, for a given shard, the shuffle phase and the reduce phase take place on the same machine, the additional network traffic is usually not worth the time savings unless machine failures occur frequently.

14.3 Further Reading

Compared with other topics covered by this book, the literature on parallel information retrieval is quite sparse. Existing publications are mostly limited to small or mid-size compute clusters comprising not more than a few dozen machines (the exception being occasional publications by some of the major search engine companies). In some cases it may therefore be difficult to assess the scalability of a proposed architecture. For instance, although the basic version of term partitioning discussed in Section 14.1.2 might work well on an 8-node cluster, it is quite obvious that it does not scale to a cluster containing hundreds or thousands of nodes. Despite this caveat, however, some of the results obtained in small-scale experiments may still be applicable to large-scale parallel search engines.

Load balancing issues for term-partitioned query evaluation are investigated by Moffat et al. (2006). Their study shows that, with the right load balancing policies in place, term partitioning can lead to almost the same query performance as document partitioning. Marín and Gil-Costa (2007) conduct a similar study and come to the conclusion that a term-partitioned index can sometimes outperform a document-partitioned one. Abusukhon et al. (2008) examine a variant of term partitioning in which terms with long postings lists are distributed across multiple index nodes.

Puppin et al. (2006) discuss a document partitioning scheme in which documents are not assigned to nodes at random but based on the queries for which they are ranked highly (according to an existing query log). Documents that rank highly for the same set of queries tend to be assigned to the same node. Each incoming query is forwarded only to those index nodes that

are likely to return good results for the query. Xi et al. (2002) and Marín and Gil-Costa (2007) report on experiments conducted with hybrid partitioning schemes, combining term partitioning and document partitioning. A slightly different view of parallel query processing is presented by Marín and Navarro (2003), who discuss distributed query processing based on suffix arrays instead of inverted files.

Barroso et al. (2003) provide an overview of distributed query processing at Google. Other instruments for large-scale data processing at Google are described by Ghemawat et al. (2003), Dean and Ghemawat (2004, 2008), and Chang et al. (2008).

Hadoop[3] is an open-source framework for parallel computations that was inspired by Google's MapReduce and GFS technologies. Among other components, Hadoop includes HDFS (a distributed file system) and a MapReduce implementation. The Hadoop project was started by Doug Cutting, who also created the Lucene search engine. Yahoo is one of the main contributors to the project and is believed to be running the world's largest Hadoop installation, comprising several thousand machines.

Recently the use of graphics processing units (GPUs) for general-purpose, non-graphics-related computations has received some attention. Due to their highly parallel nature, GPUs can easily beat ordinary CPUs in applications in which long sequences of data have to be processed sequentially or cosequentially, such as sorting (Govindaraju et al., 2006; Sintorn and Assarsson, 2008) and disjunctive (i.e., Boolean-OR) query processing (Ding et al., 2009).

14.4 Exercises

Exercise 14.1 When replicating a distributed search engine, the replication may take place either at the node level (i.e., a single cluster with $2n$ index nodes, where two nodes share the query load for a given index shard), or at the cluster level (i.e., two identical clusters, but no replication within each cluster). Discuss the advantages and disadvantages of each approach.

Exercise 14.2 Describe possible scalability issues that may arise in the context of document-partitioned indices even if the index is held in main memory. (Hint: Consider query processing operations whose complexity is sublinear in the size of the index.)

Exercise 14.3 Given a document-partitioned index with $n = 200$ nodes and a target result set size of $m = 50$, what is the minimum per-node result set size k required to obtain the correct top m results with probability 99%?

Exercise 14.4 Generalize the dormant replication strategy from Section 14.1.4 so that it can deal with k simultaneous machine failures. How does this affect the overall storage requirements?

[3] `hadoop.apache.org`

Exercise 14.5 Describe how dormant replication for a term-partitioned index differs from dormant replication for a document-partitioned index.

Exercise 14.6 (a) Design a MapReduce (i.e., a map function and a reduce function) that computes the average document length (number of tokens per document) in a given corpus of text. (b) Revise your reduce function so that it can be used as a combiner.

Exercise 14.7 Design a MapReduce that computes the conditional probability $\Pr[t_1|t_2]$ of seeing the term t_1 in a document that contains the term t_2. You may find it useful to have your map function emit secondary keys to enforce a certain ordering among all values for a given key.

Exercise 14.8 (project exercise) Simulate a document-partitioned search engine using the BM25 ranking function you implemented for Exercise 5.9. Build an index for 100%, 50%, 25%, and 12.5% of the GOV2 collection. For each index size, measure the average time per query (for some standard query set). What do you observe? How does this affect the scalability of document partitioning?

14.5 Bibliography

Abusukhon, A., Talib, M., and Oakes, M. P. (2008). An investigation into improving the load balance for term-based partitioning. In *Proceedings of the 2nd International United Information Systems Conference*, pages 380–392. Klagenfurt, Austria.

Barroso, L. A., Dean, J., and Hölzle, U. (2003). Web search for a planet: The Google cluster architecture. *IEEE Micro*, 23(2):22–28.

Carbonell, J. G., and Goldstein, J. (1998). The use of MMR, diversity-based reranking for reordering documents and producing summaries. In *Proceedings of the 21st Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 335–336. Melbourne, Australia.

Chang, F., Dean, J., Ghemawat, S., Hsieh, W. C., Wallach, D. A., Burrows, M., Chandra, T., Fikes, A., and Gruber, R. E. (2008). Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems*, 26(2):1–26.

Clarke, C. L. A., and Terra, E. L. (2004). Approximating the top-m passages in a parallel question answering system. In *Proceedings of the 13th ACM International Conference on Information and Knowledge Management*, pages 454–462. Washington, D.C.

Dean, J., and Ghemawat, S. (2004). MapReduce: Simplified data processing on large clusters. In *Proceedings of the 6th Symposium on Operating System Design and Implementation*, pages 137–150. San Francisco, California.

Dean, J., and Ghemawat, S. (2008). MapReduce: Simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113.

Ding, S., He, J., Yan, H., and Suel, T. (2009). Using graphics processors for high performance IR query processing. In *Proceedings of the 18th International Conference on World Wide Web*, pages 421–430. Madrid, Spain.

Ghemawat, S., Gobioff, H., and Leung, S. T. (2003). The Google file system. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, pages 29–43. Bolton Landing, New York.

Govindaraju, N., Gray, J., Kumar, R., and Manocha, D. (2006). GPUTeraSort: High performance graphics co-processor sorting for large database management. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, pages 325–336. Chicago, Illinois.

Marín, M., and Gil-Costa, V. (2007). High-performance distributed inverted files. In *Proceedings of the 16th ACM Conference on Information and Knowledge Management*, pages 935–938. Lisbon, Portugal.

Marín, M., and Navarro, G. (2003). Distributed query processing using suffix arrays. In *Proceedings of the 10th International Symposium on String Processing and Information Retrieval*, pages 311–325. Manaus, Brazil.

Moffat, A., Webber, W., and Zobel, J. (2006). Load balancing for term-distributed parallel retrieval. In *Proceedings of the 29th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 348–355. Seattle, Washington.

Moffat, A., Webber, W., Zobel, J., and Baeza-Yates, R. (2007). A pipelined architecture for distributed text query evaluation. *Information Retrieval*, 10(3):205–231.

Puppin, D., Silvestri, F., and Laforenza, D. (2006). Query-driven document partitioning and collection selection. In *Proceedings of the 1st International Conference on Scalable Information Systems*. Hong Kong, China.

Sintorn, E., and Assarsson, U. (2008). Fast parallel GPU-sorting using a hybrid algorithm. *Journal of Parallel and Distributed Computing*, 68(10):1381–1388.

Xi, W., Sornil, O., Luo, M., and Fox, E. A. (2002). Hybrid partition inverted files: Experimental validation. In *Proceedings of the 6th European Conference on Research and Advanced Technology for Digital Libraries*, pages 422–431. Rome, Italy.