15 Distributed Access Structures

In large scale file systems presented in the previous chapter, search operations are based on a *sequential scan* that accesses the whole data set. When it comes to finding a specific object, typically a tiny part of the data volume, direct access is much more efficient than a linear scan. The object is directly obtained using its physical address that may simply be the offset of the object's location with respect to the beginning of the file, or possibly a more sophisticated addressing mechanism.

An *index* on a collection C is a structure that maps the *key* of each object in C to its (physical) address. At an abstract level, it can be viewed as a set of pairs (k, a), called *entries*, where k is a key and a the address of an object. For the purpose of this chapter, an object is seen as raw (unstructured) data, its structure being of concern to the Client application only. You may want to think, for instance, of a relational tuple, an XML document, a picture or a video file. It may be the case that the key uniquely determines the object, as for keys in the relational model.

An index we consider here supports at least the following operations that we thereafter call the *dictionary operations*:

1. insertion *insert(k,a)*,
2. deletion *delete(k)*,
3. key search *search(k): a*.

If the keys can be linearly ordered, an index may also support *range queries* of the form *range(k_1,k_2)* that retrieves all the keys (and their addresses) in that range. Finally, if the key space is associated to a metric (a distance function f), one may consider a nearest neighbor search *kNN(o)* that retrieves the k objects closest (in other words, most similar) to a query object o.

Given a cost unit, the efficiency of an index is expressed as the number of cost units required to execute an operation. In the centralized databases case, the cost unit is usually the disk access. We are more concerned here with communication, so we will assume that the cost unit is the transmission of one message, and this (to simplify), regardless of the message size.

For indexing, two main families of access structures have been considered, namely, *hash tables*, with constant search complexity, and *search trees*, with logarithmic search complexity. In the next two chapters, we consider in turn the distribution of these two kinds of access structures.

15.1 Hash-based structures

Let us first recall the basics of hash-based indexing in centralized databases. The *hash file* structure consists of a memory-resident *directory D* and a set of M disk *buckets* $\{b_0, b_1, \ldots, b_{M-1}\}$. The directory is an array with M cells, each referring to one of the buckets (Figure 15.1).

The placement of objects in the buckets is determined by a *hash function h*. Consider a collection C of objects where each item I in it has a property $I.A$. (A is called the *hash field*.) Each item I in C is stored in the bucket b_j such that $j = h(I.A)$. So, note that the hash function takes as input a value from the hash field domain, and outputs an integer in the range $[0, M-1]$. The hash function should also follow the requirement that it uniformly assigns objects to buckets.

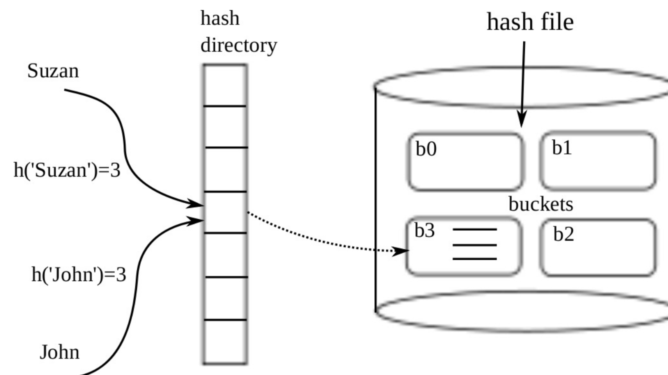


Figure 15.1: The hash file structure

Figure 15.1 shows a simple example. The hash file contains a collection of *Person* objects. The hash field is the name of each person. We find that $h('\text{Suzanne}') = 3$ and $h('\text{John}') = 3$. To insert the Suzanne object in the hash file, one computes its hash value, namely 3, finds in the Directory the address of Bucket 3, and places the object there. To retrieve the Suzanne object, one similarly finds the hash value and retrieve Bucket 3. Observe that both Suzanne and John objects are put in Bucket b_3. Indeed, two objects with totally different hash field values may be mapped to the same bucket. This is called a *collision*. Collisions are quite acceptable in hash files because the purpose of a hash function is indeed to group objects to buckets independently from the hash field distribution. A problem only arises when a bucket is full and there is no place for new objects in this bucket. This is somewhat difficult to control, and may lead to degenerate hash structures. We will discuss further how to handle this issue.

Hash files support dictionary operations in constant time. In non-degenerated cases, one disk access is sufficient.

1. insertion *insert(k, a)*: compute $h(k)$, find the address of $b_{h(k)}$ in $D[h(k)]$ and insert a there;

2. deletion *delete(k)*: find the bucket as in search, remove from it all objects with key k;

3. key search *search(k):* $\{a\}$: compute $h(k)$, find the address of $b_{h(k)}$ in the directory, read $b_{h(k)}$ and take all objects in the bucket with key k (if any).

In the simple variant presented so far, the number M of buckets must be chosen in advance so that the entire collection can be accommodated. If c_B is a bucket capacity, and $|C|$ the expected collection size, then M should be of the order $\lceil \frac{|C|}{c_B} \rceil$. In fact, it has to be somewhat greater because even if in theory, the hash function does distribute uniformly the objects into the buckets, for a particular distribution, there will always be some buckets more used than others.

Observe that the hash file is very efficient for point queries, but does not support range search. If range search is required, search trees will be preferred. An important issue that affects the basic version presented so far is that it does not adapt easily to a dynamic collection that expands or shrinks rapidly. More sophisticated versions exist, but the previous simple presentation is enough to examine the main issues that must be addressed when we want to distribute a hash structure.

Dynamicity

The first issue we consider relates to *dynamicity*. A straightforward distribution strategy consists in assigning each bucket of the hash file to one of the participating servers. For this to work, all the nodes (Clients or Servers) that access the structure have to share the same hash function. In real life, though, data sets evolve, and servers must be added or removed. A naive solution would require the modification of the hash function. For instance, we use a function $\bar{h}$ here and throughout the chapter that maps the domain of keys to integer. Then we do as follows:

- Suppose the servers $S_0, ..., S_{N-1}$ for some N are available.
- We use the hash value $h(key) = modulo(\bar{h}(key), N)$.
- We assign each key of hash value i to server S_i for each $i \in [0, N-1]$.

If a server S_N is added, the hash function is modified to:

$$h(key) = modulo(\bar{h}(key), N+1)$$

Observe that this "naive" strategy typically results in modifying the hash value of most objects, moving them to different buckets, so essentially totally rebuilding the hash file. Also, the new function h has to be transmitted to all the participants, notably all the Clients. While these changes take place, the use of the old hash function is likely to result in an error. Guaranteeing the consistency of such an addressing when the hash function is changing, in a highly dynamic, distributed environment, is a challenging task.

Location of the hash directory

The second issue that needs to be considered when distributing a hash file is the location of the hash directory itself. Recall that this directory establishes a mapping between the hashed values and the physical locations of data repositories. Any operation on the structure requires an access to the directory which therefore constitutes a potential bottleneck.

We present next two hash-based indexing techniques adapted to a distributed setting. The first one called *linear hashing* (an extension of the well-known dynamic hashing method) has been proposed a while ago in the context of centralized systems. We recall it and consider its distribution. The second one, called *consistent hashing*, is a direct attempt at instantiating the hashing paradigm in a distributed context. We will see that it is better adapted when servers enter and leave the system at a rapid pace. Both approaches provide interesting insights on common design patterns for data-centric distributed structures: caching, replication, routing tables, and lazy adjustment. Consistent hashing is further illustrated with the system CHORD, a Distributed hash tables (DHT) designed for P2P environments.

@ Serge Abiteboul, Ioana Manolescu, Philippe Rigaux, Marie-Christine Rousset, Pierre Senellart, 2011; to be published by Cambridge University Press 2011.

15.1.1 Distributed Linear Hashing

The goal of linear hashing (LH) is to maintain an efficient hash structure when the collection is very dynamic, and in particular when it may grow very rapidly. The maintenance involves a dynamic enlargement of the hash directory that entails an extension of the hash function, and the reorganization of the buckets. As we will see, *linear hashing* provides a linear growth of the file one bucket at a time. We first recall the method in a centralized case, then develop the distributed version.

Linear hashing

The simple manner of expending a hash file, when the collection expands, and a bucket b become too small, consists in introducing an *overflow* bucket. More precisely, this consists in (i) adding a new bucket b' to the file, (ii) moving some items from b to b', adding a pointer from b to b'. If we are not careful and the collection keeps expanding, we have to perform more and more linear scans in the buckets corresponding to one hash value. To avoid this issue, in linear hashing, when we introduce an overflow bucket, we simultaneously augment the number of hash values. The LH innovative idea is to decouple the extension of the hash function from the overflowing bucket. More precisely, when a bucket b overflows, this triggers the following modifications:

1. An overflow bucket is linked from b, in order to accommodate the new items.

2. The bucket b_p (or the chain list of buckets) corresponding to the hash value p, *usually distinct from* b, is split, where p is a special index value maintained by the structure and called the *split pointer*.

Observe that a bucket that overflows is not split. It is just linked to an overflow bucket. This bucket together with the overflow bucket will eventually be split when the split pointer will point to it. Surprisingly, this behaves nicely. Hash values that raise problems are eventually dealt with and the number of hash values somewhat gracefully adapt to the size of the collection.

Initially, $p = 0$, so bucket b_0 is the first that must split, *even if it does not overflow*. The value of p is incremented after each split. Look at Figure 15.2. We abstractly represented the presence of an object of key k in a bucket by placing $\bar{h}(k)$ in it. So, for instance, some object k with $h(k) = 17$ is in b_1. Here, the size of the hash directory is 4 and we assume that each bucket holds at most 4 objects (we only show $h(k)$ for each key k). For simplicity, we use the *mod()* function for hashing, so $h(k) = \bar{h}(k)\ mod\ N$, where N is the size of the hash directory.

An object with key 42 must be inserted in bucket b_2, which overflows its capacity. A bucket is linked to b_2, and receives object 42. At the same time, bucket b_0 (recall that $p = 0$) is split. Its content is partially reassigned to a new bucket added to the hash file, b_4.

This reassignment raises an issue: if we keep unchanged the hash function, all the objects moved to bucket b_4 cannot be found anymore. This is where the hash function extension occurs. Linear hashing actually relies on a *pair* of hash functions (h_n, h_{n+1}), where for each n:

1. $h_n : k \to \bar{h}(k) \bmod 2^n$, so in particular,

2. $h_{n+1} : k \to \bar{h}(k) \bmod 2^{n+1}$

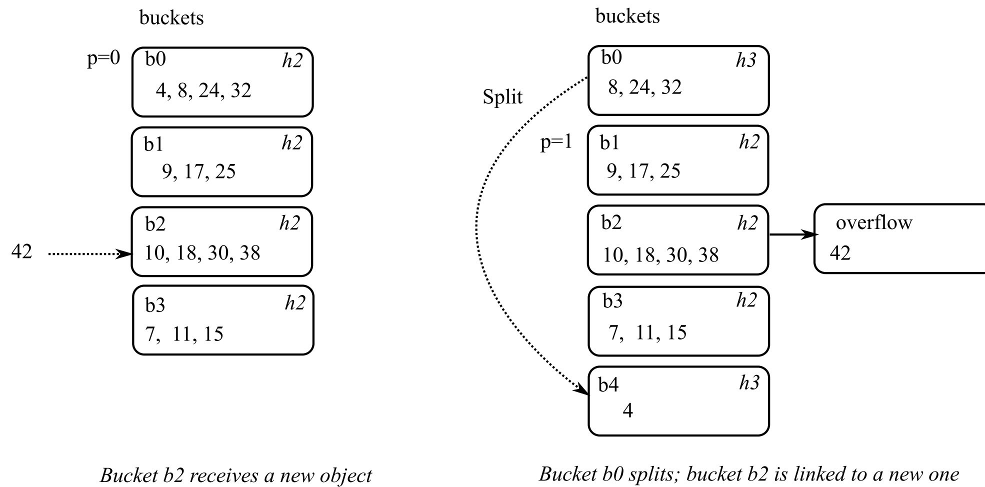


Figure 15.2: Split process in linear hashing

Initially, the pair is (h_0,h_1), $p = 0$ and h_0 applies to all the buckets. As the structure evolves, p is incremented and h_0 applies to the buckets in the range $[p, N-1]$, while h_1 applies to all other buckets. (Recall that N is the size of the hash directory.) For the example of Figure 15.2, after the first split, we have:

1. h_0 applies to buckets b_1,b_2, and b_3,
2. h_1 applies to buckets b_0 and b_4: the ones that just split.

It can be verified in the example that all objects such that $h_1(k) = \bar{h}(k) \bmod 2^3 = 4$ have been moved to bucket b_4, while those for which $h_1(k) = \bar{h}(k) \bmod 2^3 = 0$ stay in bucket b_0. This extension of the hash function is therefore consistent, and allows for a limited reorganization of the hash file.

What happens next? Bucket b_1 is the next one to split, if *any* of the buckets (including b_1 itself) overflows. Then p will be set to 2, and b_2 becomes the split target. When the split of b_2 occurs, its content and that of its associated overflow bucket will be distributed in two first-range buckets, and this will likely eliminate the need for a linked chain. In this perspective, linear hashing can be seen as a delayed management of collision overflows.

Eventually, p will take the value 3. When b_3 splits in turn, the hash function h_1 is applied for all the buckets and h_0 is no longer used. The hash file is "switched" one level up, the pair of hash function becomes (h_1,h_2), p is reset to 0 and the process goes on gracefully. Observe that a large part of the hash directory is left unchanged when we modify the hash function. This is an important advantage of the technique since we avoid having to resend it entirely.

A dictionary operation on some key value k uses the following computation (called the *LH algorithm* in what follows) to obtain the address a of the bucket that contains k:

```
a := h_n(k);
if (a < p) a := h_{n+1}(k)
```

@ Serge Abiteboul, Ioana Manolescu, Philippe Rigaux, Marie-Christine Rousset, Pierre Senellart, 2011; to be published by Cambridge University Press 2011.

In words: one applies first h_n, assuming that no split occurred, and obtain a hash value a in the range $[0, 2^n - 1]$. Next, one checks whether this value corresponds to a bucket that did split, in which case the correct address is obtained with h_{n+1}.

Distributed linear hashing

We now turn to LH*, a distributed version of LH. The mapping of a LH file structure to a cluster of servers is straightforward. Assume for the time being that there exists a global knowledge of the file level, n, with hash functions (h_n, h_{n+1}), and of the split pointer p. Suppose the cluster consists of the servers $\{S_0, S_1, \ldots, S_N\}$, $2^n \leq N < 2^{n+1}$, each holding a bucket. When the bucket of a server S_i overflows, the server referred to by the split pointer p, S_p, splits. This involves the allocation of new server S_{N+1} to the structure, and a transfer from S_p to S_{N+1} of objects, similar to the LH case. This results in a partition of data among servers, often denoted as "sharding", each bucket being a "data shard". To perform this split, we can either wait to have a new server available, or better, have each physical server plays the role of several "virtual servers".

Recall that, as already mentioned, the Linear Hashing technique does not require resending entirely the hash directory each time the hash function is modified. When this happens, we have to let the servers know:

- the level n that determines the pair of hash functions (h_n, h_{n+1}) currently in use,
- the current split pointer p,
- changes of the hash directory.

We meet again a standard trade-off in distributed data structures:

- either all the participants have an accurate and up-to-date view of the whole structure; then searches are fast, but changes to the structure involve a costly propagation of the update to each node (including Client nodes).
- or they only maintain a partial representation, possibly lagged with respect to the actual structure status; in that case, the maintenance cost is possibly much lighter, but searches may have to follow non trivial paths before reaching their targets.

For LH*, for instance, each server and each Client could store a local copy of the localization information: the pair (n, p) as well as the list of all the server nodes addresses. Let us call *Loc* this information. Whenever the LH* evolves by adding or removing a server, an update must be sent to every participant. This yields a *gossiping system*, a perfectly valid choice in a rather controlled environment, assuming the set of participating peers does not evolve at a rapid pace.

Reducing maintenance cost by lazy adjustment

LH* provides a more flexible solution to cope with the maintenance problem. Each Client keeps its local copy Loc' of *Loc*, but this copy may be out-of-date with respect to the "true" *Loc*, e.g., p may have been incremented since Loc' was acquired. This may lead the Client to addressing errors: a dictionary operation with key k may be sent to a wrong server, due to some distributed file evolution ignored by the Client. LH* then applies a *forwarding*

path algorithm that eventually leads to the correct server. This latter server carries out the required operations. Furthermore, with the acknowledgment, it sends back to the Client some information for Client to refresh its copy of *Loc*. The next client request, based on this refreshed information, will be more accurate than the initial one.

We call *client image* the knowledge maintained by a Client on the distributed structure. An image is some partial replication of the global structure in the client cache. It is *partial* because parts of the structure that the Client does not need are not mirrored in the image. Also, it may not record recent evolutions of the structure that followed the image acquisition.

Keeping an outdated replica is imposed for a number of reasons. First, a Client may be temporarily disconnected, and thus incapable of updating its image. Of course, one can imagine that the Client refreshes asynchronously its image when it reconnects. But this is complex and very expensive if Clients connect/reconnect often. Also, the maintenance of all Clients and Servers completely up-to-date is likely to represent an important traffic overhead. A (reasonably) outdated image represents a good trade-off, providing that the Client knows how to cope with referencing errors. We examine next how LH* adjusts to addressing errors.

Details on the LH* algorithms

The adjustment mechanism principles are illustrated in Figure 15.3. Here, we assume that the Client image is $(n_C = 1, p_C = 1)$, whereas several splits led the LH* to the status $(n = 3, p = 2)$.

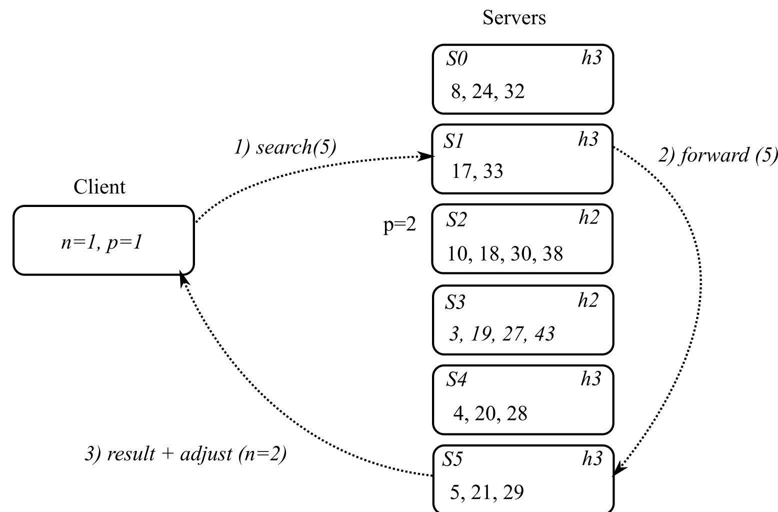


Figure 15.3: The LH* adjustment mechanism.

The Client sends a request *search(5)*. It computes the bucket address with the LH algorithm (see above):

1. $a = h_{n_C}(5) = 5 \bmod 2^1 = 1$
2. since $a \geq p_C$, we keep $a = 1$ and the request is sent to S_1.

When a LH* server receives a request, it first checks whether it is indeed the right recipient by applying the following algorithm (called *the forward algorithm*). The algorithm attempts

@ Serge Abiteboul, Ioana Manolescu, Philippe Rigaux, Marie-Christine Rousset, Pierre Senellart, 2011; to be published by Cambridge University Press 2011.

to find the correct hash value for key k, using the local knowledge of the server on the file structure.

```
// Here, j denotes the server level
a' := h_j(k)
if (a' ≠ a)
  a'' := h_{j-1}(k)
  if (a'' > a and a'' < a') then a' := a''
```

If a', obtained by the above algorithm is not the server address, then the Client made an addressing error. The request is then forwarded to server a'. In our example, S_1 receives the Client request. S_1 is the last server that split, and its level is 3. Therefore, $a' = h_3(5) = 5 \bmod 2^3 = 5$. The request is forwarded to S_5 where the key is found.

It can be shown that the number of messages to reach the correct server is three in the worst case. This makes the structure fully decentralized, with one exception: when a Server overflows, the exact value of p, the (other) Server that splits, must be accurately determined. The LH* recommendation is to assign a special role to one of the servers, called the *Master*, to keep the value of p and inform the other nodes when necessary. Since this only happens during a split, the structure remains scalable. We omit the other technical details. For more, see the bibliographical notes at the end of the section; see also exercises.

The main lesson that can be learned from this design is that a relative inaccuracy of the information maintained by a component is acceptable, if associated to a stabilization protocol that guarantees that the structure eventually converges to a stable and accurate state. Note also that, in order to limit the number of messages, the "metadata" information related to the structure maintenance can be piggybacked with messages that answer Client requests.

We mentioned in the discussion what happens when some Client gets temporarily disconnected. We implicitly assumed that Servers are somewhat stable. We will remove this assumption in the next technique that also addresses settings where there is a high *churn* within the Servers, i.e., Servers come and go at a rapid pace.

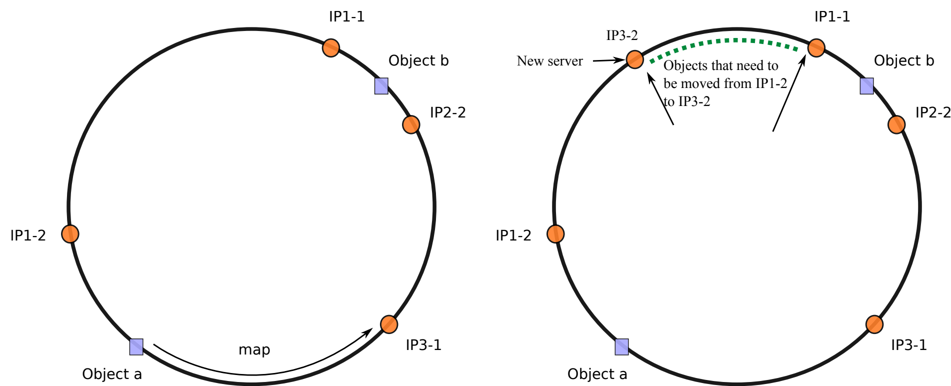
15.1.2 Consistent Hashing

Consistent hashing is a method initially proposed in the context of distributed caching systems. Consider for instance a Web Site that receives an intensive flow of HTTP requests. A useful strategy to distribute the query load is to keep the results of the most common queries in the *caches* of several servers. A dedicated proxy machine records which servers store which query results. When a query is received, the proxy detects whether its result has been cached and when this is the case, it forwards the query to one of the servers that cached this particular result. This is where consistent hashing helps: the assignment of queries to servers is based on the hash value of the query, and the scheme is designed to gracefully adapt itself to a varying number of servers (see the *http://memcached.org* Web site for details).

Distributing data with Consistent Hashing

In the context of data distribution, the same mechanism can be adopted, the only difference lying in the handling of the hash directory, discussed at the end of this section. The first idea is to use a simple, non-mutable hash function h that maps *both* the server address and the

object keys to the same large address space A. Assume for instance that we choose a 64-bits addressing space A. The hash function can be implemented as follows: take the server IP (resp., the object key) and apply the cryptographic MD5 algorithm that yields a 32-bytes string; then interpret the first 8 bytes of the MD5 value as an unsigned integer in the range $[0, 2^{64} - 1]$. And similarly for keys. So, now, both the servers and the keys are mapped to the same very large domain, $[0, 2^{64} - 1]$.



Mapping of objects to servers   Server IP3-2 is added, with local re-hashing

Figure 15.4: The ring of hash values in Consistent Hashing

The second idea is to organize A as a ring, scanned in clockwise order. That is, each element has a successor, the successor of $2^{64} - 1$ being 0. The situation is depicted in the left part of Figure 15.4. The large circle is A; small circles represent servers; and small squares represent objects. Clearly, we do not have 2^{64} available servers, the large size of A being merely intended to avoid collisions. We must therefore define a rule for assigning objects to servers. The Consistent Hashing mapping rule is as follows:

> If S and S' are two adjacent servers on the ring,
> all the keys in range $[h(S), h(S')[$ are mapped to S.

Looking again at Figure 15.4, Object a is hashed to a position of the ring that comes after (the hash value of) IP3-1 and before (the hash value of) IP1-2. Thus, a is mapped to the server IP3-1, and the object is stored there. By a similar mechanism, object b is mapped to IP1-1. One obtains a partition of the whole data sets in “shards” where each server is fully responsible for a subset of the whole collection. The scheme is often completed by a replication of shards on a few nodes for failure management purposes: see further.

What did we gain? The immediate advantage is that when a new server is added, we do not need to re-hash the whole data set. Instead, the new server takes place at a position determined by the hash value on the ring, and part of the objects stored on its successor must be moved. The reorganization is local, as all the other nodes remain unaffected. Figure 15.4, right, shows an example, with a new server IP3-2 inserted on the ring. The set of objects whose hash belongs to the arc between IP3-2 and IP1-1 were initially stored on IP1-2, and must be reallocated on IP3-2. A similar local process holds when a server leaves the ring. Because the hash function remains unaffected, the scheme maintains its consistency over the successive evolutions of the network configuration.

@ Serge Abiteboul, Ioana Manolescu, Philippe Rigaux, Marie-Christine Rousset, Pierre Senellart, 2011; to be published by Cambridge University Press 2011.

Refinements

The global scheme we just presented may be improved. The most important improvement aims at balancing the load between the servers, a concern that is not addressed by the basic approach. The example of Figure 15.4 shows that the size of the arcs allocated to servers may vary. Server IP3-1 for instance receives all the objects hashed to the arc between itself and its successor. If objects are uniformly hashed (as they should be with a correct hash function), the load of IP3-1 is likely to be much more important than that of, say server IP1-1.

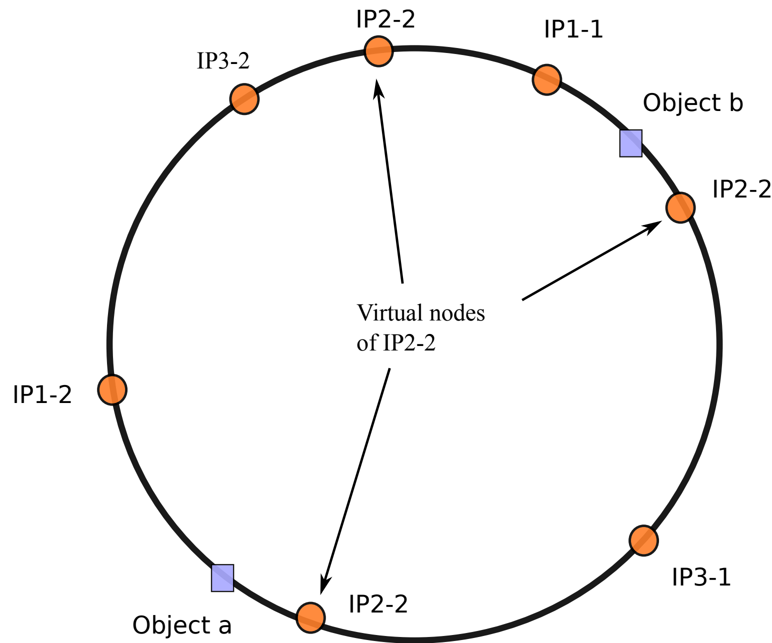


Figure 15.5: Load balancing with Consistent Hashing

To fix this shortcoming, we extend Consistent Hashing with the concept of *virtual nodes*. Basically, the ring consists of a large number of virtual machines, and several virtual machines are hosted on a same physical server. This is illustrated in Figure 15.5. Server IP2-2 has been (virtually) split in three nodes assigned to multiple points of the ring. In practice, a physical server is assigned to hundreds of virtual nodes. This obviously helps balancing the storage and query load: a server may be "lucky" and get assigned a virtual machine with very little data, but is unlikely to be randomly assigned many such virtual machines. So, the workload is more evenly distributed between the physical machines. Moreover, when a physical server is removed from the ring, all its objects are not assigned to a unique unlucky neighbor, but are split between all the successors of its virtual nodes. Similarly, when a server joins the ring, it takes a piece of the workload of many physical servers and not just some of one. Last but not least, observe that virtualization also helps dealing with heterogeneity of servers. A very powerful machine can support many more virtual servers than a very slow one.

A second useful refinement relates to failure management. Data must be replicated to prevent any loss due to server failure. There are many possible replication strategies. One may for instance use several hash functions, say h_1, h_2, h_3 for a *replication factor* of 3. An object of key k is replicated on the servers in charge of $h_i(k)$ for each i. One can fix the replication

factor depending on the needs of the application.

The hash directory

A last question pertains to the location of the hash directory. Recall that this directory maintains the mapping between hash values and the location (i.e., IP address) of servers. As previously mentioned, Consistent Hashing was originally designed for distributing cache hits. In such an environment, a proxy server hosts the directory and routes requests to the appropriate node on the ring. The equivalent architecture in a large-scale data management system would be a single Master – many servers organization, where a dedicated machine, the Master, receives queries and forwards them to data servers. This is a simple choice, that raises concerns regarding its scalability.

Other solutions are possible, the choice depending mostly on the level of dynamicity of the system. We consider two:

Full duplication. The hash directory is duplicated on each node. This enables a quite efficient structure, because queries can be routed in one message to the correct server. It requires a notification of all the participants when a server joins or leaves the network, so this solution is better adapted when the network is stable.

Partial duplication. If the system is highly dynamic, in particular in a P2P context, the amount of "gossiping" required by the full duplication may become an important overhead. In that case, a partial duplication that only stores $O(\log N)$ entries of the hash directory on each server, N being the total number of servers, is a choice of interest. It allows in particular the routing of queries in $O(\log N)$ messages, and thus constitutes a convenient trade-off between the cost of network maintenance and the cost of dictionary operations.

Full duplication is used, for instance, in the DYNAMO system, a distributed hash table implemented by Amazon for its internal infrastructure needs (see references). A typical example of partial duplication is CHORD, a Distributed Hash Table, that we present next.

15.1.3 Case study: CHORD

A *distributed hash table* is a hash structure distributed in a fully decentralized manner, and thus particularly adapted to unstable networks where nodes can leave or join at any moment. "Decentralized" in this context has a strong implication: there cannot be a node, or a group of nodes, responsible for any critical part of the system maintenance. DHTs are mostly used in P2P systems, and the presentation that follows explicitly adopts this context. We now call *peers* the server nodes, each peer being identified by a unique pId (e.g., URI).

Overview

A DHT supports the search, insert and delete dictionary operations. Range queries are not possible, although range structures can be "overlaid" upon a DHT infrastructure (see last section). Since a DHT is itself already overlaid over the Internet, a specialized range structure based on the search tree paradigm is better adapted to range searches. We will consider search trees in the next section.

A DHT must also handle the following operations related to the network topology:

@ Serge Abiteboul, Ioana Manolescu, Philippe Rigaux, Marie-Christine Rousset, Pierre Senellart, 2011; to be published by Cambridge University Press 2011.

- *join(pId)*: let a peer *pId* joins the network, and possibly transfer some objects to this new player;
- *leave(pId)*: peer *pId* leaves the network; its objects are distributed to other peers.

CHORD is one of the first DHTs proposed at the beginning of the millennium as a complete solution to the problem of indexing items stored in a P2P network. CHORD is based on Consistent Hashing, as presented above. Figure 15.6 (left) shows a CHORD ring over an address space $A = 2^3$. Each peer with Id n is located at node n mod 8 on the ring (e.g., peer 22 is assigned to location 6). Each peer has a *successor*, the next peer on the ring in clockwise order, and a *predecessor*.

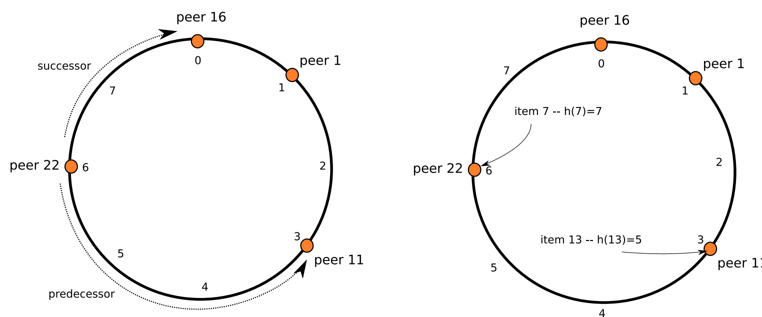


Figure 15.6: The CHORD ring, with $m = 3$ (left part), and key assignment in CHORD (right part)

Each peer p is *responsible* for a range of keys: an object is assigned to the server that *precedes* its hash value on the ring. Object 13 for instance is hashed to $h(13) = 5$ and assigned to the peer p with the largest $h(p) \leq 5$. So far, this is a direct application of Consistent Hashing. A main contribution of CHORD comes from the design of its routing tables.

Routing tables

Let $A = 2^m$ be the address space, i.e., the number of positions of the ring. Each peer maintains a routing table, called $friends_p$, that contains (at most) $\log 2^m = m$ peer addresses. For each i in $[1 \ldots m]$, the ith friend p_i is such that

- $h(p_i) \leq h(p) + 2^{i-1}$
- there is no p' such that $h(p_i) < h(p') \leq h(p) + 2^{i-1}$

In other words, p_i is the peer responsible for key $h(p) + 2^{i-1}$. Figure 15.7 shows the friends of peer 16, with location 0 (note the collisions). Peer 16 does *not* know peer 22.

Example 15.1.1 Let $m = 10$, $2^m = 1024$; consider peer p with $h(p) = 10$. The first friend p_1 is the peer responsible for $10 + 2^0 = 11$; the second friend p_2 is the peer responsible for $10 + 2^1 = 12$; finally the last friend p_{10} is the peer responsible for $10 + 512 = 522$

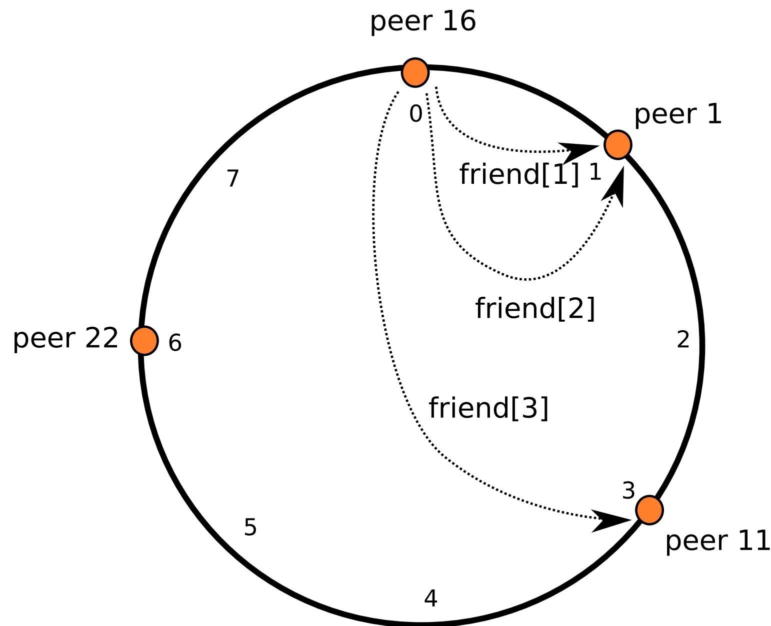


Figure 15.7: The friend nodes in CHORD

The CHORD routing tables imply some important useful properties. First, a peer's routing table contains at most m references. Each peer has no more than 16 friends in a ring with $m = 16$ and $2^{16} = 65,536$ nodes. Second, each peer knows better the peers close on the ring that the peers far away. Finally, a peer p cannot (in general) find directly the peer p' responsible for a key k, *but p can always find a friend that holds a more accurate information about k.*

CHORD operations

We illustrate the operations supported by CHORD with the *search()* algorithm (the other can be easily inferred). Assume that p searches for key k. Two cases occur:

- if p is responsible for k, we are done;
- else let i such that $h(p) + 2^{i-1} \leq h(k) < h(p) + 2^i$: p forwards the search to its friend p_i.

For instance, looking at Figure 15.8, peer 16 receives a request for item k, with $h(k) = 7$. First, peer 16 forwards the request to peer 11, its third friend; then peer 11 forwards to peer 22, its third friend; and, finally, peer 22 finds k locally. As a matter of fact, the search range is (at worse) halved at each step, and thus the search converges in $O(\log 2^m) = O(m)$ messages.

In a P2P network, nodes can join and leave at any time. When a peer p wants to join, it uses a *contact peer* p' which helps p carry out three tasks: (i) determine the friends of p, (ii) inform the peers for which p becomes a friend, and (iii) move some objects to p.

Let N be the current number of nodes. In order to locate the friends of p, p' uses its own routing table. This involves $O(\log N)$ times (the number of friends) a lookup that costs $O(\log N)$ messages, hence a total cost of $O(\log^2 N)$ messages (see exercises).

@ Serge Abiteboul, Ioana Manolescu, Philippe Rigaux, Marie-Christine Rousset, Pierre Senellart, 2011; to be published by Cambridge University Press 2011.

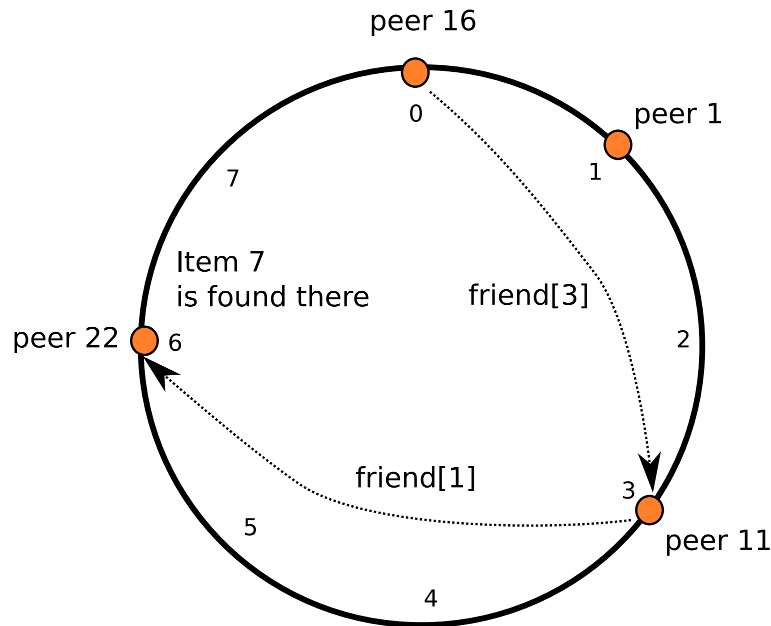


Figure 15.8: The *search()* operation in CHORD

Next, the routing table of the existing nodes must be updated to reflect the addition of p. This is the trickiest part of the process. We note first that p becomes the ith friend of a peer p' if and only if the following conditions hold:

1. $h(p) - 2^{i-1} \leq h(p') < h(p) - 2^{i-2}$
2. The current ith friend of p' is before p on the ring.

And, finally, p takes from its predecessor all the items k such that $h(p) \leq h(k)$. At this point the join procedure is completed and the network is consistent.

Example 15.1.2 Consider the example of Figure 15.9 that shows how peer 13 joins the ring, taking the slot 5. We assume that its contact node is peer 22. First, peer 22 computes the routing table of peer 13. Next, one finds that peer 13 is the third friend of either a peer at slot 2 ($5 - 2^{3-2} - 1$), or 1 ($5 - 2^{3-1}$). Finally, peer 13 receives part of the data stored on peer 11. Details of the process are left to the reader as an exercise.

We do not elaborate the leaving procedure which is essentially similar in its principles with the operations seen so far. As usual for failure handling, one distinguishes two cases:

Cold. Peer p leaves "cold" when it knows it is leaving and has the time to take appropriate measures. This case is sometimes called "fail-stop". The local part of the file stored at p must be transmitted to the predecessor, and the routing tables must be updated.

Hot. This is the more difficult case resulting from a failure, and all kinds of failures typically happen often. We want to reach eventually and as fast as possible a consistent state,

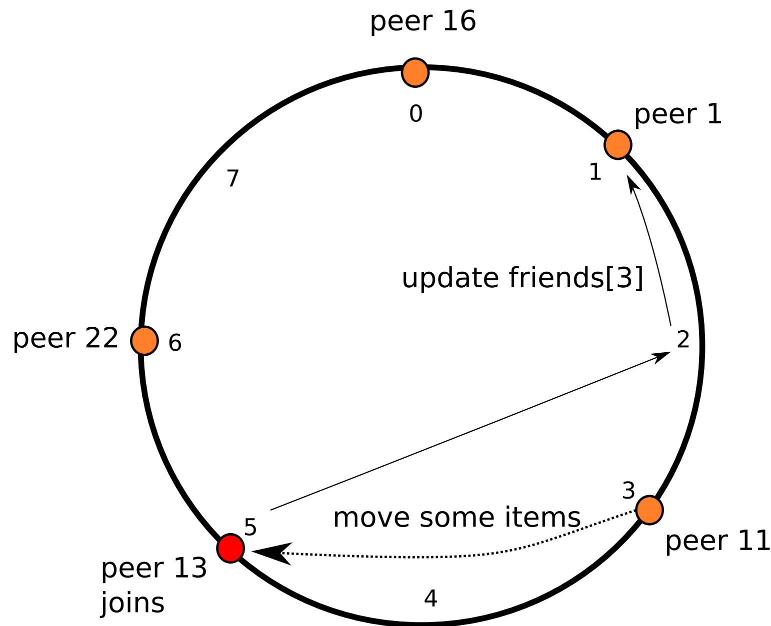


Figure 15.9: Joining a node in CHORD

with as in the cold case, the local part of the file stored at p transmitted to its predecessor, and the routing tables updated. For the local part of the file, we rely on replication. More precisely, the content of a peer is replicated on r predecessors, where the replication factor r depends on the expected network stability. The general rule to fix addressing errors to a peer p that does not answer is to re-route the query to a predecessor of p which chooses another route.

This concludes our presentation of hash-based indexing structures. In short, these structures are very efficient. Both LH* and DHT are quite adapted to rapidly evolving collections. DHT also support very well high churn of the network.

15.2 Distributed indexing: Search Trees

Hash-based data structures do not support range queries or nearest-neighbors searches. This is a well-known limitation that justifies the coexistence, in centralized systems, of hash tables and tree indexes (generally, B+trees). We study in this section the design of distributed search trees.

15.2.1 Design issues

A general problem with tree structures is that operations usually execute a top-down traversal of the tree. A naive distribution approach that would assign each tree node to a server would result in a very poor load balancing. The server hosting the root node, in particular, is likely to become overloaded. Figure 15.10 shows a tree structure, where each black dots represents

@ Serge Abiteboul, Ioana Manolescu, Philippe Rigaux, Marie-Christine Rousset, Pierre Senellart, 2011; to be published by Cambridge University Press 2011.

both a node of the tree and the server where it resides. For the sake of illustration we assume a binary tree but the discussion holds for any node fanout.

Consider a query that searches for objects located on node (i.e., a server) e. The left part of the figure illustrates what happens in naive implementation. A client node always contacts the root node a which forwards the query down the tree, eventually reaching e. Statistically, a receives twice more messages that its children, and generally 2^h more messages than any node located at level h.

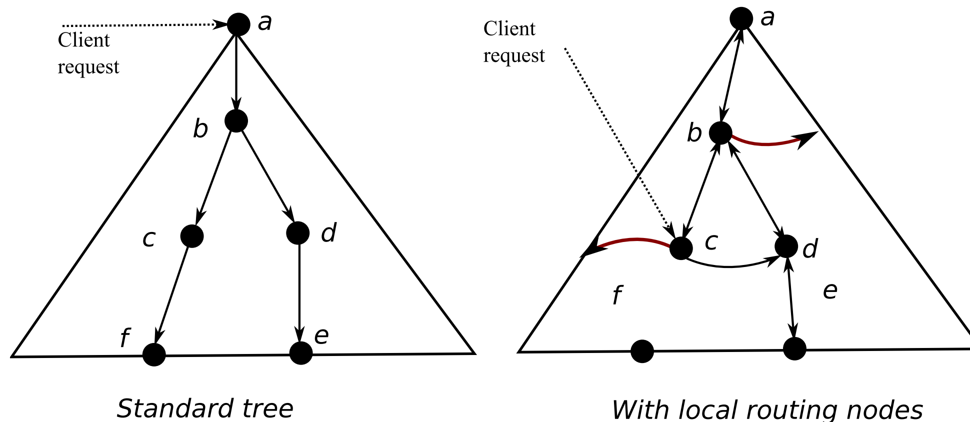


Figure 15.10: Design issues for distributed trees

The design of distributed tree structures attempts to avoid such bottlenecks with a combination of three ideas:

1. *caching* of the tree structure, or part of it, on the Client node, so that the Client node can directly access the server that can serve its request,
2. *replication* of the upper levels of the tree, to avoid overloading servers in charge of nodes in these levels.
3. *routing tables*, stored at each node, enabling horizontal and vertical navigation in the tree.

With respect to caching, a standard means of improving the search for a particular server is to keep in the Client cache an *image* that records the part of the tree already visited by the Client. If, for instance, a previous query led the client to node f through node c, this local part of the structure can be memorized in the client image. Any query in that Client addressing the same part of the key space can then directly access node c, avoiding a full traversal from the root to f. Otherwise, the client must still use the standard algorithm. If the client only knows c, the query must be first forwarded up from c to the nearest ancestor (here, b) that covers the query criteria, then down to the leaf. Thus, the use of such caching does not avoid an imbalance of the servers load.

A more drastic approach is to replicate the whole tree on each node of the cluster. This way, a client node can directly send its query to any server of the cluster and get an exact answer. It suffices to apply a protocol that evenly distributes the keys over all the servers to solve the load balancing issue. As already seen with hash-based approaches, this clearly trades one

problem for another, namely the maintenance of an exact replica of the whole tree at each node. For instance, in some variant, a server sends the whole tree image to the client at each request. In a system with say, hundreds of servers, hundreds of thousands of clients, and millions of queries every day, this represents a huge overhead.

Finally, assume that each node of the distributed tree stores the list of its siblings, along with the part of the key space covered by each (Figure 15.10, right part). The navigation is not anymore limited to the "parent" and "child" axis (borrowing the XPath terminology). A search that targets node e can start from c (assuming the client identifies in its image this node as the closest to its query) which inspects its local routing table and forwards the query to its sibling d. From d, a short top-down path leads to the leaf.

We next present in detail two representative approaches to the distribution of a search tree. The first one, namely BATON, is tailored to P2P networks, whereas the second, namely BIGTABLE, is built for clusters of machines.

15.2.2 Case study: BATON

BATON is a P2P tree structure that aims at efficiently supporting range queries. It is actually representative of several data structures designed to overcome the limitation of hash-based approaches regarding range queries. The goal is to index a collection of objects using keys from a linearly ordered domain. We assume a homogeneous collection of servers, each with maximal capacity of B entries (an entry is a pair $[k,o]$, k being the key and o an object).

Kernel structure

The structure of the distributed tree is conceptually similar to that of a binary tree (e.g., AVL trees). It satisfies the following properties (for some fixed value B):

- Each internal node, or *routing node*, has exactly two children.
- To each node a of the tree is associated a range:
  - The range of the root is $]-\infty,+\infty[$.
  - The range of a nonleaf node is the union of the ranges of its children. The "cut" point is assigned to the right child. (For instance, a node of range $[12,72[$ may have children with ranges $[12,42[$ and $[42,72[$ with 42 belonging to the second.)
- Each leaf node, or *data node*, stores the subset of the indexed objects whose keys belong to its range.
- Each leaf node contains at least $B/2$ and at most B entries.

Observe that the definition is almost that of a standard binary tree, except for the leaves data storage. We note also that a binary tree with n leaves has exactly $n-1$ internal nodes. This permits a simple mapping of the conceptual structure to a set of n servers. Each server S_i (except server S_n) stores exactly a pair (r_i,l_i), r_i being a routing node and l_i a leaf node. As a leaf node, a server acts as an objects repository up to its maximal capacity.

Figure 15.11 shows a first example with three successive evolutions. Leaf nodes are shown with circles, internal (routing) nodes with rectangles, and each node, whether leaf or internal, is associated to the server where it resides. Initially (part 1) there is one node a on server S_0

@ Serge Abiteboul, Ioana Manolescu, Philippe Rigaux, Marie-Christine Rousset, Pierre Senellart, 2011; to be published by Cambridge University Press 2011.

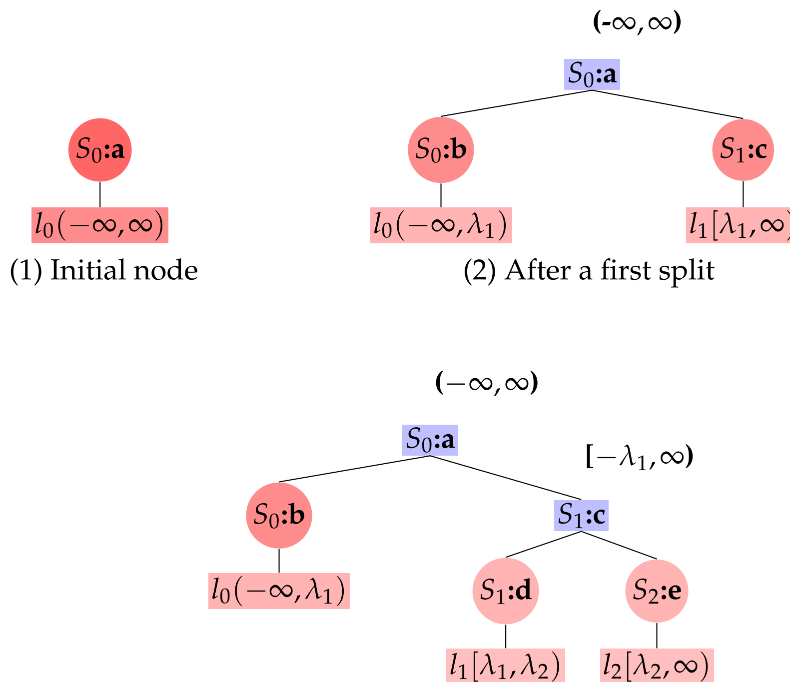


Figure 15.11: Evolution of the binary search tree

storing the leaf l_0, and its range is $]-\infty,\infty[$. Any insert request will put the new entry on this server, whatever its key.

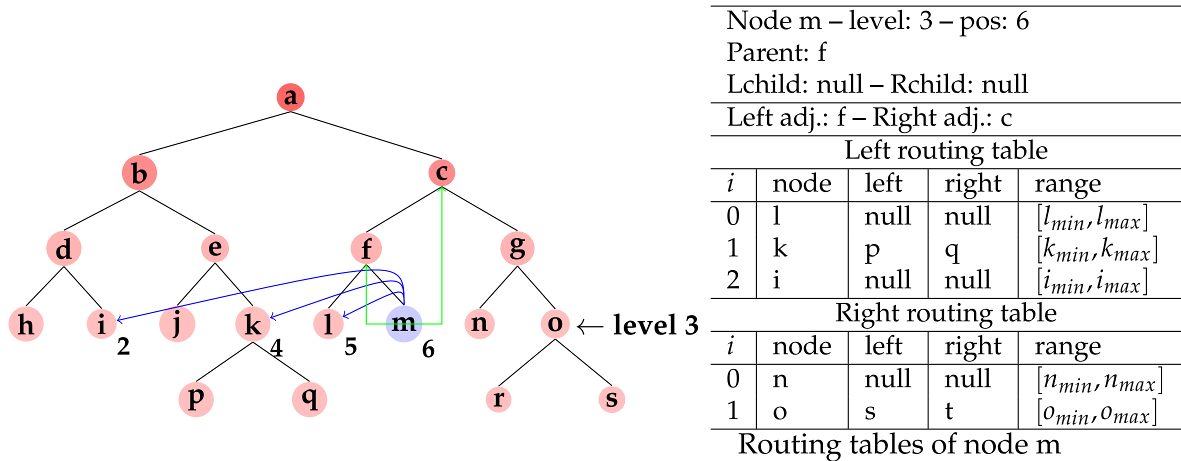
When S_0 gets full because it contains B entries, a split occurs (part 2). A new server S_1 stores the new leaf l_1. The entries initially stored in l_0 have been distributed among the two servers by considering the median value λ_1, or *pivot key*, just like the standard B-tree split.

A routing node a is stored on S_0. The range of this routing node is the union of the ranges of its children, e.g., $]-\infty,\infty[$. A routing node maintains links to its left and right children, and to its parent node, which is a difference with standard binary trees in the centralized context where upward pointers are rarely kept. A *link* is simply a pair (`Id`, `range`), where `Id` is the Id of the server that stores the referenced node, and `range` is its range. For our example, the left link of a if $\langle S_0 : b,]-\infty,\lambda_1[\rangle$ and its right link is $\langle S_1 : c, [\lambda_1,\infty[\rangle$. Links are used during top-down traversals of the distributed tree.

When it is the turn of server S_1 to split, its collection of entries is further divided and the objects distributed among S_1 and a new server S_2 that will store a new leaf node. A routing node c is created with range $[\lambda_1,\infty[$, and the ranges of its left and right children $S_1 : d$ and $S_2 : e$ are respectively $[\lambda_1,\lambda_2[$ and $[\lambda_2,\infty[$.

Performance

This basic distribution schema yields reasonable storage balancing. After a certain number of key insertions and assuming keys are not removed from the access structure, one can see that each server is at least half-full. However, as previously discussed, load balancing is not achieved: servers corresponding to upper-levels of the tree get more work. To perform a



Node m – level: 3 – pos: 6
Parent: f
Lchild: null – Rchild: null
Left adj.: f – Right adj.: c

Left routing table				
i	node	left	right	range
0	l	null	null	$[l_{min}, l_{max}]$
1	k	p	q	$[k_{min}, k_{max}]$
2	i	null	null	$[i_{min}, i_{max}]$

Right routing table				
i	node	left	right	range
0	n	null	null	$[n_{min}, n_{max}]$
1	o	s	t	$[o_{min}, o_{max}]$

Routing tables of node m

Figure 15.12: Routing information in BATON

dictionary operation, a client sends the query to the root node, that forwards it the query to the appropriate nodes down the tree. Since all searches start from the root node, the server that hosts it, is both a bottleneck and a single point of failure. As the number of clients increases, so does the number of incoming requests that this root server must handle. As a result the structure (as just described) does not scale.

What about efficiency? Any dictionary operation takes as input a key value, starts from the root node and follows a *unique* path down to a leaf. If the *insert()* requests are independently and uniformly distributed, the complexity is logarithmic in the number of servers *assuming the tree is balanced*. However, such a binary tree may degenerate to a worst case linear behavior for all dictionary operations. (This is left as an exercise). To fix this issue, some "rotation" operations are applied to maintain the balance of the tree. Rotations techniques for trees are standard and details may be found in textbooks. We rather focus now on the management of routing tables, which is more specific to the distributed context.

Routing tables

In addition to the structural information presented above, nodes maintain *routing tables* that guide tree traversals to find the appropriate nodes. More precisely, each node stores the following information:

1. its level l in the tree (starting from the root, at level 0);
2. its position pos in this level, $0 \leq pos < 2^l$);
3. the addresses of its parent, left and right children;
4. the address of the previous and next adjacent nodes in in-order traversal;
5. left and right routing tables, that reference nodes at the same level and at position $pos + / - 2^i$ for $i = 0, 1, 2, \ldots$.

For instance, for the binary tree of Figure 15.12, node m is at level 3, and its position in this level, for the whole tree, is 6. The left routing table refers to nodes at respective positions

@ Serge Abiteboul, Ioana Manolescu, Philippe Rigaux, Marie-Christine Rousset, Pierre Senellart, 2011; to be published by Cambridge University Press 2011.

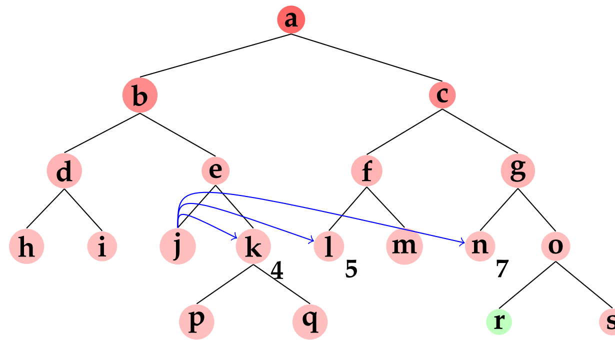


Figure 15.13: Searching with routing tables

$6 - 2^0 = 5$, $6 - 2^1 = 4$, and $6 - 2^4 = 2$. This is strongly reminiscent of the friends in CHORD. A node knows better the nodes that are close, and the number of friends is clearly logarithmic in the total number of nodes. BATON records the range and children of its friends. Note that, since the tree is not complete at each level, a friend may be set to `null`.

BATON operations

First consider search. A client may send a *search(k)* request to any peer p in the tree; so, it suffices to know a single server to access information. A left or right traversal is first initiated, to find the node p' *at the same level as* p whose range covers k. The process is somewhat similar to CHORD search: if p' is not part of the friends of p, p finds, among its friends, the one that is "closest" to p'. Specifically, assuming that k is larger than the upper bound of its range, p chooses the furthest friends p'' whose lower bound is smaller than k. The search continues until p' is found. Then the search continues downwards to the leaf in charge of k.

Looking at figure 15.13, assume a request sent to node j for a key that belongs to node r. From the friends of j, the one that satisfies the horizontal search condition is n, the third friend in the right routing table. Then, the horizontal traversal continues with n forwarding the search to o, its first friend. From o, the top-down search phase leads to node r.

In summary, BATON proceeds with a CHORD-like search in a given tree level, and with a top-down search when the node at the current level whose subtree contains the searched key has been found. The first phase, enabled by the routing tables, enables a horizontal navigation in the tree that avoids to always have to access the root.

Recall that a P2P structure, in addition to dictionary operations (that include range search in the present case), must also provide *join()* and *leave()* operations that affect the network topology.

Just like in CHORD, a peer p that wants to join the network uses any *contact peer* p' to initiate the join in the structure. In BATON, a peer that joins always becomes a leaf, and receives half of the objects of an existing leaf (that becomes its sibling). The join process first searches for the appropriate insertion location for p, then proceeds with the update of routing tables to reflect the role of the new participant.

Intuitively, we choose for insert location, an existing leaf with the smallest possible level (that is, one of the leaves which are nearest to the root). Possible candidates in Figure 15.12 are h, i, j, l, m, and n. The main difficulty is to update the data structure and notably the routing tables.

The last operation is the departure of a node from the network. When a leaf peer p declares that it leaves ("cold" departure), two cases occur:

1. p is one of the deepest leaves in the tree and the balance of the tree is in principle not affected by the departure of p. The main ask is then to distribute keys of p to neighbors.

2. p is one of the deepest leaves. We then need to find a replacement for p and for that we choose one of the deepest leaves in the tree that is moved from its previous location in the tree.

Again the difficulty is the maintenance of the routing tables, in particular in the second case.

If the BATON structure is difficult to maintain when new information is entered (possibly requiring tree balancing) and servers joining/leaving (possibly leading to moving servers in the tree), it is very efficient for search. Also, it clusters objects based on the values of their keys, and supports range queries. The two-phase mechanism based on horizontal navigation in the tree followed by some constant number of vertical navigation, leads to an $O(\log n)$ cost. A design principle that we can see at work in BATON is: find an appropriate trade-off between the amount of replication (here by the number of "friends" at each node) and the efficiency of the structure.

To conclude with BATON , observe that "hot" departures in such a complex structure become very difficult to handle. Although the redundancy of routing tables provides the means of rebuilding the information of a lost peer, this comes at the cost of lots of work and in particular, lots of communications. As a consequence, it becomes difficult to guarantee robustness in a highly changing environment.

15.2.3 Case Study: BIGTABLE

In 2006, the Google Labs team published a paper on "BIGTABLE: A Distributed Storage System for Structured Data". It describes a distributed index designed to manage very large data sets ("petabytes of data") in a cluster of data servers. BIGTABLE supports key search, range search and high-throughput file scans. BIGTABLE also provides a flexible storage for structured data. As such, it can also be seen as a large distributed database system with a B-tree-like file organization.

The presentation that follows highlights the main architectural and technical aspects of BIGTABLE. Many details are omitted. Please refer to the Further Reading section at the end of the chapter.

Structure overview

Figure 15.14 gives an overview of the structure. The data representation is roughly similar to the relational model. A *table* contains a list of *rows*. Each row consists of a key and a list of columns. The rows are sorted in lexicographic order by the key values. A large table is partitioned horizontally in "tablets" which constitute the leaves of the distributed tree. The size of a tablet is typically a few hundreds of megabytes.

The content of a row differs from those of standard relational tables. First, columns can be grouped in "families" which form the basic data management unit in BIGTABLE. The columns of a same family are stored independently from those of the other families. Hence, BIGTABLE captures both aspects of a row store (several columns in one family) and that of

@ Serge Abiteboul, Ioana Manolescu, Philippe Rigaux, Marie-Christine Rousset, Pierre Senellart, 2011; to be published by Cambridge University Press 2011.

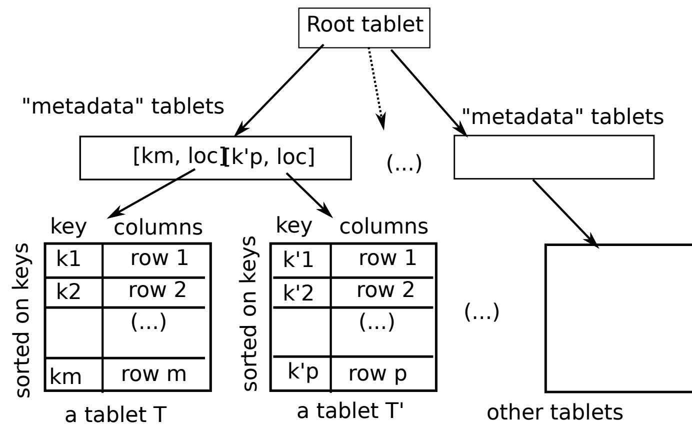


Figure 15.14: Overview of BIGTABLE structure

a column-oriented store, with the typical advantage that compression can benefit from the homogeneity of the values stored in a column. Moreover, a "cell" (i.e., the intersection of a row and a column) can store many versions of a same content (e.g., a Web page collected from the Web), each identified by a timestamp. This completes the multi-map organization of BIGTABLE which can therefore be summarized as a four-level access pattern:

$$\text{key} \rightarrow \text{family} \rightarrow \text{column} \rightarrow \text{timestamp}$$

It is possible to access each content independently from the others by combining those four criteria. Note finally that the data model is flexible enough to allow for the addition of new columns on a per-row basis.

The tablets of a table are distributed in a cluster. In order to perform efficient lookups during exact and range search, tablets are indexed on the range of the keys. At this point, the BIGTABLE organization differs from our generic search tree. Instead of a binary tree reflecting the split history of the structures, BIGTABLEs collects the tablet ranges and store them in another table, called the "metadata" table. One obtains what would be called, in a centralized context, a non-dense index on a sorted file. Since the metadata table is managed just as any other table, it is itself partitioned into tablets. The indexing process can be applied recursively by creating upper levels until the collection of ranges occupies a single tablet, the root of a distributed tree, quite similar to centralized B-trees.

Figure 15.14 shows the first level of metadata. Its rows consist of pairs (key, loc), where loc is a tablet location, and key is the key of the last row in the tablet. Note that, because the table is sorted, this is sufficient to determine the range of a table. The first pair in the metadata table represented on Figure 15.14 for instance refers to a tablet covering the range $]-\infty, k_m]$.

Conceptually, the number of levels in the distributed tree is unbounded. In practice, BIGTABLE limits the number of levels to 3: the root node is a tablet above the metadata table which is never split. This suffices to potentially index very large data sets. Indeed, assume a (realistic) tablet size of $2^{28} = 268$ MBs, and 1 KB entries in the metadata tablet. Then:

1. a metadata tablet can index $268\,000 \approx 2^{18}$ data tablets,
2. the root tablet can index in turn up to 2^{18} metadata tablets, hence 2^{36} data tablets.

Since the capacity of a tablet is 2^{28} bytes, this represents a maximal storage of 2^{64} bytes for a single table (16,384 Petabytes!).

Note that a single metadata tablet is already sufficient to index a 2^{46} bytes data set. If this metadata tablet were stored without replication, and without using Client caches, it would likely become a bottleneck of the system. BIGTABLE uses caching and replication to distribute the load evenly over many servers.

Distribution strategy

A BIGTABLE instance consists of a Master server, many (tablet) servers, and Client nodes. The Master acts as a coordinator for a few tasks that would be complicated to handle in a fully decentralized setting. This includes:

1. maintenance of the table schemas (columns names, types, families, etc.);
2. monitoring of servers and management of failures;
3. assignment of tablets to server.

Each tablet server handles the storage and access to a set of tablets (100-1000) assigned by the Master. This involves persistence management, error recovery, concurrency, and split request when the server capacity is exhausted. The tablet server also provides search and update services on all the rows that belong to its range.

The evolution of the tree is rather different from that of a P2P structure like BATON. Three cases occur: (i) a tablet server decides to split and to distribute a part (about half) of its tablets to another server, (ii) a tablet server merges its content with another server and (iii) a failure (of any kind) occurs and keeps the server from participating to the cluster.

The failure is handled by the Master which sends heartbeat messages and initiates replacement, in a way similar to that already seen for GFS (see Section 14.5.2, page 299). The other two cases modify the range of the tablet server. At this point we have two choices: either the modification is reported to all the participants, including the Client nodes, or a lazy strategy is adopted. This is the same kind of trade-off already encountered with the LH* structure, CHORD and BATON.

BIGTABLE relies on lazy updates. This may affect the result of operations required by other components of the system because their requests may fall "out of range" due to some discrepancy between their local information and the actual status of the structure. The system is always ready to handle such errors. An "internal" out-of-range can be met when the Master requires from a server a tablet which does not correspond to its range. In that case, there is a discrepancy between the actual range covered by the server, and the range stored at the metadata level, which can be viewed as an replicated "image" of the tablet range. The tablet server then initiates an adjustment message that informs the Master of the past split. Another case of out-of-range affects Client nodes: the stabilization protocol in case of out-of-range is explained next.

Adjustment of the Client image

A Client is a Google application that uses the BIGTABLE client library. A Client initially connects to the cluster through the Master, which sends back information regarding the tablet

@ Serge Abiteboul, Ioana Manolescu, Philippe Rigaux, Marie-Christine Rousset, Pierre Senellart, 2011; to be published by Cambridge University Press 2011.

servers of interest to the Client initial requests. This information is stored in the Client local cache (its image in our terminology) and used to directly communicate with tablet servers later on. The Master is therefore rarely involved in the message exchanges, and this keeps it from being overloaded. So far, this is akin to GFS.

The Client maintains its image regardless of the changes that affect the tree. If a request is sent to a server that does no longer hold the required key, an out-of-range answer is sent back. The Client requires then the correct address to the metadata level. In the worse case, this induces a new out of range request, and another round-trip with the root is necessary. Assume that, initially, the situation is that shown on Figure 15.15. The Client stores an outdated image of the tree with only five tablets (including one metadata tablet, and the root). Since the last refresh of the Client image, the tree has actually grown to the state shown on the right, with ten tablets. The figure gives the range owned by each leaf.

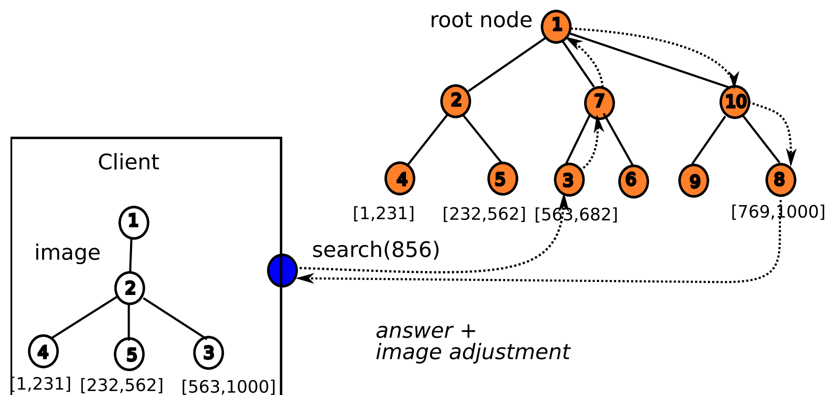


Figure 15.15: Example of an out-of-range request followed by an adjustment

The Client sends a *search*(856) request. From its image, it determines that node **3** should own the key. When node **3** processes the message, it observes an out of range. The search is forwarded to node **7**, which forwards it to the root node. From there, a standard top-down traversal finally leads to the true owner of key 856. The Client image is then refreshed with the information collected during this process, so that the addressing error does not repeat (at least for this part of the tree). In case of an out of range request, 6 networks round trips may be necessary in the worse case (three for an upward path, three for a downward one).

Persistence

Each tablet server manages locally the persistence of its tablets. Figure 15.16 shows the components involved in *write()* and *read()* operations (the latter being a full scan) of a single tablet. Recall first that a tablet is sorted by its key. It would be very inefficient to attempt an insertion of each new row in a sorted file, because of the time necessary to find its location, and to the complicated space management incurred by the order maintenance. BIGTABLE uses a more sophisticated, yet rather classical strategy, which can be summarized as an incremental sort-merge with REDO logging. When a row must be inserted in the tablet, the *write()* operates in two steps:

1. the row is appended to a log file;
2. after the log has acknowledged the append operation, the row is put in an in-memory sorted table.

The log file belongs to the persistent storage, resilient to failures (and managed by GFS). As soon as a row has been written to the log, GFS guarantees that it can be recovered, even if the tablet server crashes. Moreover, the log is an append-only file. A row can be added with minimal overhead because it is always inserted right away at the end of the current log file.

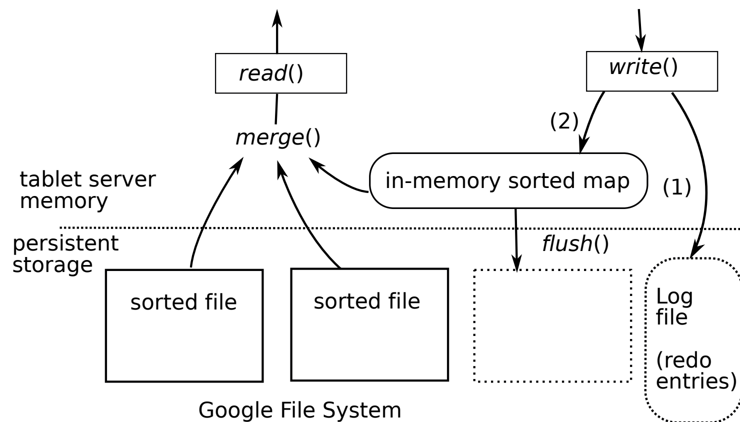


Figure 15.16: Persistence management in a tablet server

The second step puts the row in a sorted table in memory. This does not require a disk access, and can therefore be performed efficiently. Moreover, rows in this table can be merged very quickly with the rest of the tablet content, to satisfy read requests. Thus, the row becomes immediately accessible even though it is still not in the sorted file.

The memory table grows and eventually reaches a predefined threshold. At this point the sorted table is reconstructed and stored. The space of the memory table becomes available for subsequent *write()*. So, eventually, a tablet therefore consists in a set of files (called SSTables in Google terminology), each sorted by the key, and an in-memory collection also sorted by the key. Now, if a *read()* requests the content of the tablet, this content must be supplied in key order. This is achieved with a *merge()* of the files that constitutes the tablet, plus the current content of the in-memory table. Note that the hybrid process that involves persistent file and in-memory structures is reminiscent of the merge-based construction of an inverted index.

As the number of SSTables increases, so does the cost of their merging. A practical obstacle is that a merge, to be efficient, requires a sequential scan of the SSTables, which can only be achieved if each is stored on a distinct disk entirely devoted to the scan operation. Therefore, in order to limit the fan-in, merges are carried out periodically between the sorted files to limit their number.

Example 15.2.1 Take the example of a server with eight disks and a 2 GB volatile memory allocated to the sorted map. Each time the map gets full, a flush creates a new 2GB SSTable, and we take care of assigning each of them to a distinct disk. When seven such “runs” have been created, the periodic merge should be triggered: each of the seven used disk carries

@ Serge Abiteboul, Ioana Manolescu, Philippe Rigaux, Marie-Christine Rousset, Pierre Senellart, 2011; to be published by Cambridge University Press 2011.

out a *sequential* scan of its SSTable, the merge occurs in memory, and the result is *sequentially* written on the eigth disk.

15.3 Further reading

Centralized indexing can be found in any textbook devoted to database management [61]. Linear hashing [114, 112] is implemented in most off-the-shelf relational databases. LH* is presented in [116]. Consistent Hashing is proposed and analyzed in [107]. The technique is used in the Open Source *memcached* library (`http://memcached.org`), in the Amazon's DYNAMO system [52] and in the CHORD DHT [150]. The authors of [52] describe in detail the design of DYNAMO, whose data structure is a large distributed hash table intended to provide fast and reliable answer to point queries. Beyond the indexing aspect, DYNAMO features many interesting aspects representative of the main issues that must be addressed during the design and implementation of a large-scale data-centric system. The data model for instance is limited to a "keystore", i.e., pairs $(key, value)$, and supports only a few basic operations: *put()* and *get()*. Several open-source implementation of Amazon's project now exist, including VOLDEMORT `http://project-voldemort.com/`, and CASSANDRA `http://cassandra.apache.org/`. DYNAMO has inspired many of the recent distributed key-value stores collectively known as "NoSQL" (Not Only SQL) databases (see `http://nosql-databases.org/`).

Distributed hash tables is a term coined by [53] who first proposed the extension of hash techniques to distributed systems. Several DHT structures have been proposed, notably, CHORD [150], Pastry [141], Tapestry [188], and CAN [140]. Distributed hash tables have been developed as a means of overcoming shortcomings of the flooding strategy used in early P2P systems such as Napster. The $O(\log N)$ number of messages required to route a query incurs a latency which may be deemed unsuitable for highly demanding application. Through extended replication, a $O(1)$ message cost can be reached: see [139], and the already mentioned DYNAMO paper. DHTs are widely used in P2P systems: BitTorrent, the Kad network, the Storm botnet, YaCy, and the Coral Content Distribution Network.

Distributed strategies for tree-based structures has been first tackled in [115, 111] which propose several important principles later adopted for Web-scale indexing. In particular, details on the maintenance of a tree image in the Client cache can be found in [111]. BATON is a P2P tree structure presented in [102]. See also [101] for a multiway tree structure based on the same principles and [103] for a multidimensional structure, the VBI-tree. Another proposal for offering range search is the P-tree [50] which is essentially a B+tree distributed over a CHORD network. Each node stores a leaf of the B+tree, as well as the path from the root to the leaf. Point and range queries can be answered in $O(\log N)$, but the maintenance of the structure as peers join or leave the network is costly: in addition to the $\log N + \log^2 N$ costs of adding the new peer to CHORD, information must be obtained on the tree structure to build the tree branch.

BIGTABLE has inspired several other projects outside Google, including the HYPERTABLE Open source project (*http://www.hypertable.org/*), the HBASE data structure of HADOOP, and CASSANDRA, which combines features from both DYNAMO and BIGTABLE. BIGTABLE is described in [44]. Its canonical usage is the storage of documents extracted from the Web and indexed by their URL. BIGTABLE (and its Open-source variants) is more than just an

indexing mechanism, as it features a data model that can be roughly seen as an extension of the relational one, with flexible schema and versioning of cell values.

15.4 Exercises

Exercise 15.4.1 (Static and extendible hashing) *The following is a list of French* "départements"*:*

```
3  Allier       36 Indre       18 Cher       75 Paris
39 Jura         9  Ariege      81 Tarn       11 Aude
12 Aveyron      25 Doubs       73 Savoie     55 Meuse
15 Cantal       51 Marne       42 Loire      40 Landes
14 Calvados     30 Gard        84 Vaucluse    7 Ardeche
```

The first value is the key. We assume that a bucket contains up to 5 records.

1. *Propose a hash function and build a static hash file, taking the records in the proposed order (left-right, then top-bottom).*
2. *Same exercise, but now use an linear hash file based on the following hash values:*

```
Allier  1001    Indre   1000    Cher    1010    Paris   0101
Jura    0101    Ariege  1011    Tarn    0100    Aude    1101
Aveyron 1011    Doubs   0110    Savoie  1101    Meuse   1111
Cantal  1100    Marne   1100    Loire   0110    Landes  0100
Calvados 1100   Gard    1100    Vaucluse 0111   Ardeche 1001
```

Exercise 15.4.2 (LH*) *Consider Figure 15.3, page 309. What happens if we insert an object with key 47, still assuming that the maximal number of objects in a bucket is 4.*

Exercise 15.4.3 (LH*) *Prove that the number of messages for LH* insertion is three in the worst case.*

Exercise 15.4.4 (Consistent Hashing) *Assume that someone proposes the following solution to the problem of distributing the hash directory: each node maintains the hash value and location of its successor. Discuss the advantage and disadvantages of this solution, and examine in particular the cost of the dictionary operation (insert, delete, search) and network maintenance operations (join and leave).*

Exercise 15.4.5 (CHORD friends) *Express the gap between* $\text{friends}[i]$ *and* $\text{friends}[i+1]$ *in the routing table of a* CHORD *peer, and and use the result to show formally that a search operations converges in logarithmic time.*

Exercise 15.4.6 *Consider the* CHORD *ring of Figure 15.17. What are the friends of* $p11$*, located at 3?*

Exercise 15.4.7 *Develop an example of the worst case for the search() operation in* CHORD*, with* $m = 10$*.*

Exercise 15.4.8 (BATON range search) *Explain the range search algorithm of* BATON*.*

@ Serge Abiteboul, Ioana Manolescu, Philippe Rigaux, Marie-Christine Rousset, Pierre Senellart, 2011; to be published by Cambridge University Press 2011.

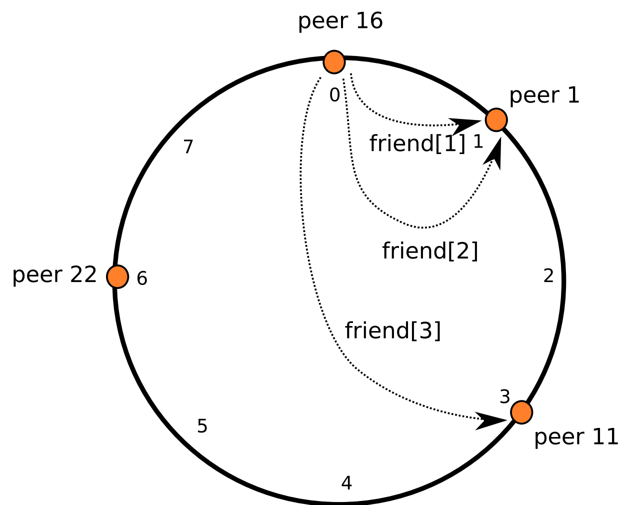


Figure 15.17: A CHORD ring

Exercise 15.4.9 (BATON range search) *Complete the description of the Join algorithm in* BATON *regarding the modification of routing tables in order to take into account the new node.*

Exercise 15.4.10 (BIGTABLE) *Describe the split algorithm of* BIGTABLE. *Compare with standard B-tree algorithms.*

Exercise 15.4.11 *Compare two of the index structures presented in the Chapter and identify, beyond their differences, some of the common design principles adopted to cope with the distribution problem. Take for instance distributed linear hashing and* BIGTABLE *and consider how a Client communicates with the structure.*