Distributed Access Structures

Hash-based techniques

Serge Abiteboul Ioana Manolescu Philippe Rigaux
Marie-Christine Rousset Pierre Senellart

Web Data Management and Distribution
http://webdam.inria.fr/textbook

September 23, 2011

Outline

Outline

Indexing techniques for very large collections

$\Rightarrow$ in the present talk, focus on hash-based approaches.

Indexing techniques in centralized databases

- fixed hashing, dynamic hashing and linear hashing.

Distributed indexing techniques for very large collections

- Distributed linear hashing
- Consistent hashing
- Illustration with the Dynamo system (Amazon) and Chord (P2P systems)

Indexing structures

We assume a (very) large collection C of pairs (k, v), where k is a key and v is the value of an object (seen as row data).

An index on C is a structure that associates the key with the (physical) address of v. It supports *dictionary operations*:

1. insertion *insert(k, v)*,
2. deletion *delete(k)*,
3. key search *search(k): v*.
4. (optional) range search *range(k_1, k_2): v*$\}$.

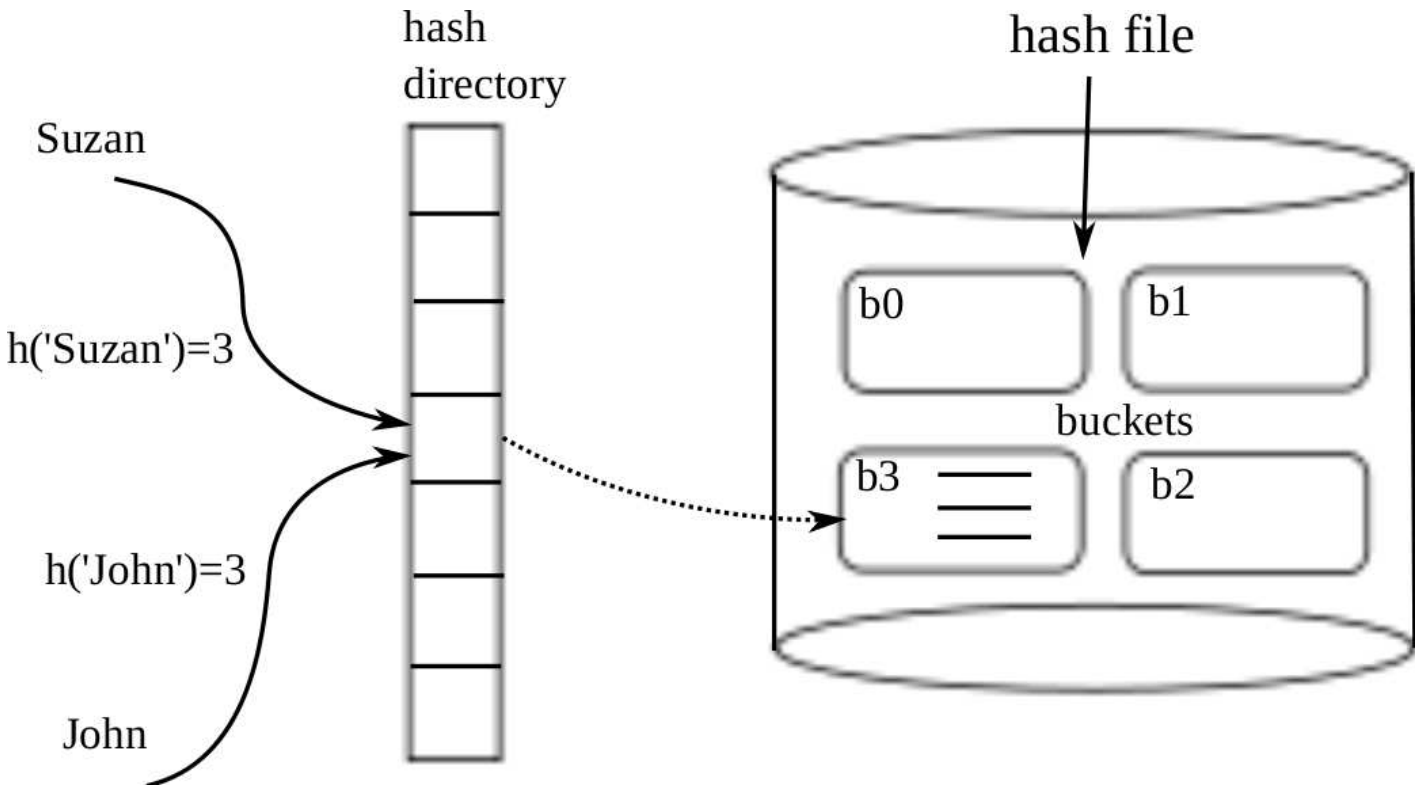
The efficiency of an index is expressed as the number of unit costs required to execute an operation.

NB: in a distributed index, one should also consider (node) *leave* and (node) *join* operations.

Outline

Basics: Centralized Hash files

The collection consists of $(key, value)$ pairs. A hash function evenly distributes the values in buckets w.r.t. the key.



This is the basic, static, scheme: the number of buckets is fixed.
Dynamic hashing extends the number of buckets as the collection grows – the most popular method is linear hashing.

Dynamic hashing

Now, we aim at reorganizing the hash file when insertions/deletions occur. We adopt the following constraints:

- the directory size (number of entries) is a power of 2.
- the hash function returns a 4-bytes integer (32 bits)

Basic idea: we use the n first bits of the hash result, with $n < 32$.

Example : ash values for the 16 movies

titre	h(title)
Vertigo	01110010
Brazil	10100101
Twin Peaks	11001011
Underground	01001001
Easy Rider	00100110
Psychose	01110011
Greystoke	10111001
Shining	11010011

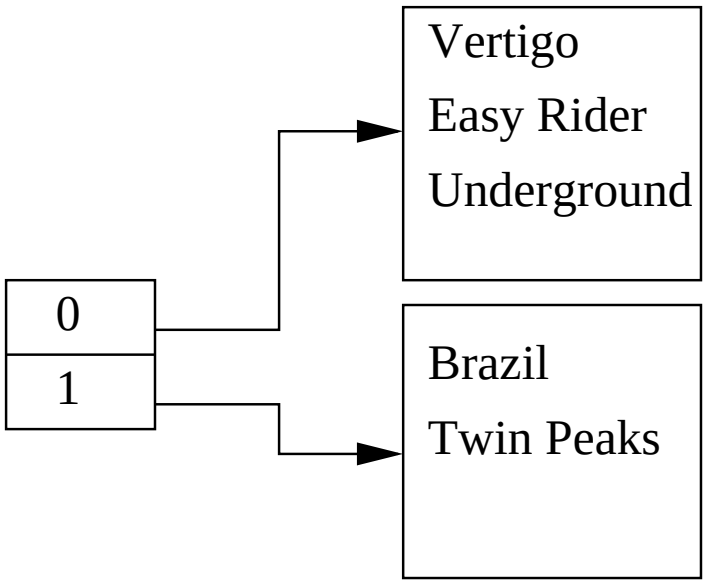
Build the hash file

We only use the first bit.

- Two possible values: 0 et 1
- Two directory entries; 2 buckets.
- The assignement of a record to a buckets depends on its last bit.

=> at this point, still a classical framework.

After 5 movies



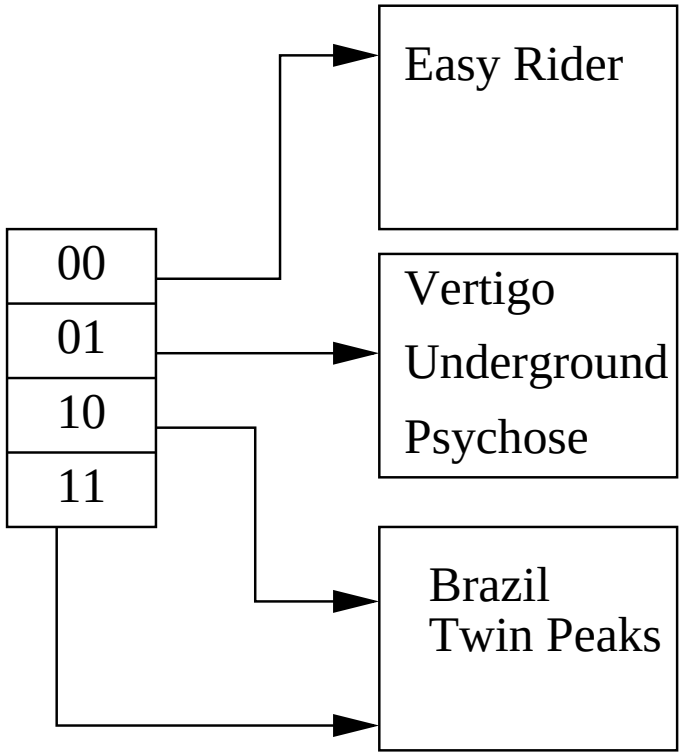
Insertions

Assume 3 films max in each bucket. Inserting Psycho (hash value 01110011) overflows the first block. Then:

- The directory size doubles.
- A new bucket is allocated for entry 01
- Entries 10 and 11 both refer to the *same* bucket.

This, on add only the minimal number of buckets – but the directory grows with a fixed rate.

Illustration



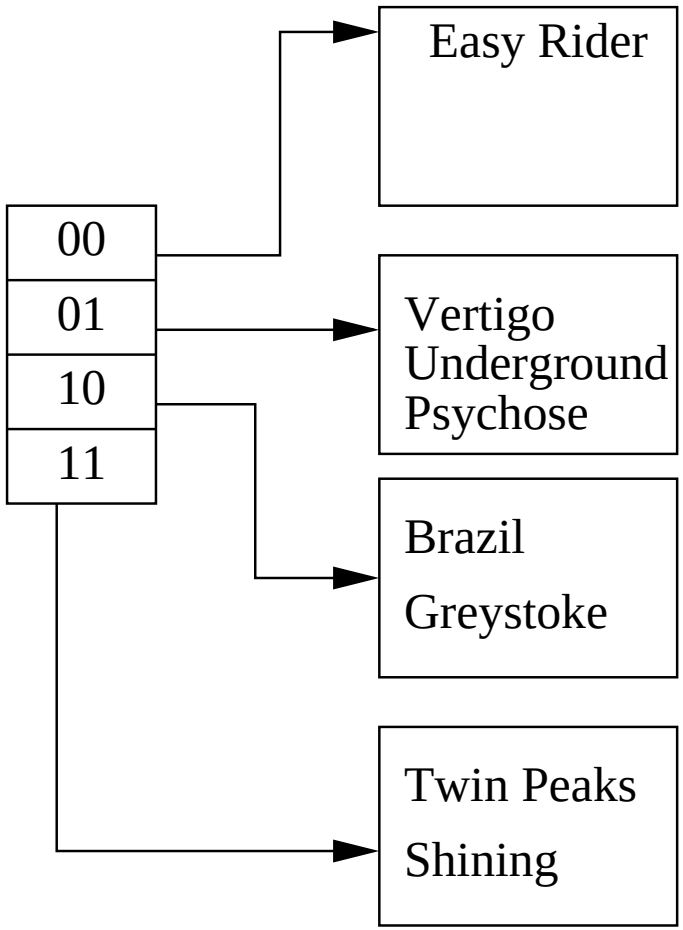
Subsequent insertions

Several cases

- One insert in a full bucket, referred to by several directory entries.
  - a new bucket is allocated; pointers are distributed; no directory enlargment.
- One insert in a full bucket, referred to by only one entry.
  - One doubles again the directory.

  The directory size may become very large.

Greystoke 10111001 et Shining 11010011



Exercises

The following is a list of French *départements*.

3 Allier	36 Indre	18 Cher	75 Paris
39 Jura	9 Ariège	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 Ardèche

We assume that a bucket contains up to 5 records. Build a static hash file.

Exercises (cont')

Same exercise, but now use an extendible hash file based on the following hash values.

Allier	1001	Indre	1000	Cher	1010	Paris	0101
Jura	0101	Ariège	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	Ardèche	1001

Linear hashing, a dynamic hash method

The basic idea is to split a bucket when the index gets full.

- add a new bucket b' to the file;
- move some records from b to b'.

First approach: split a bucket that gets full. Problem: this also changes the hash function which must accurately reflect the distribution of keys.

Intuition of linear hashing: decouple the split of buckets, and the evolution of the hash function, such that they eventually converge.

Linear hashing: how does it work?

When a bucket b overflows:

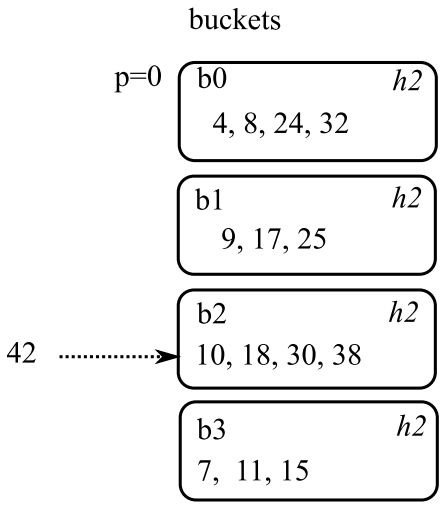
1. a chained bucket is linked to b, in order to accommodate the new records,
2. a pre-determined bucket b_p, *usually distinct from b* is split, p being a special index value maintained by the structure and called the *pointer*.

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.

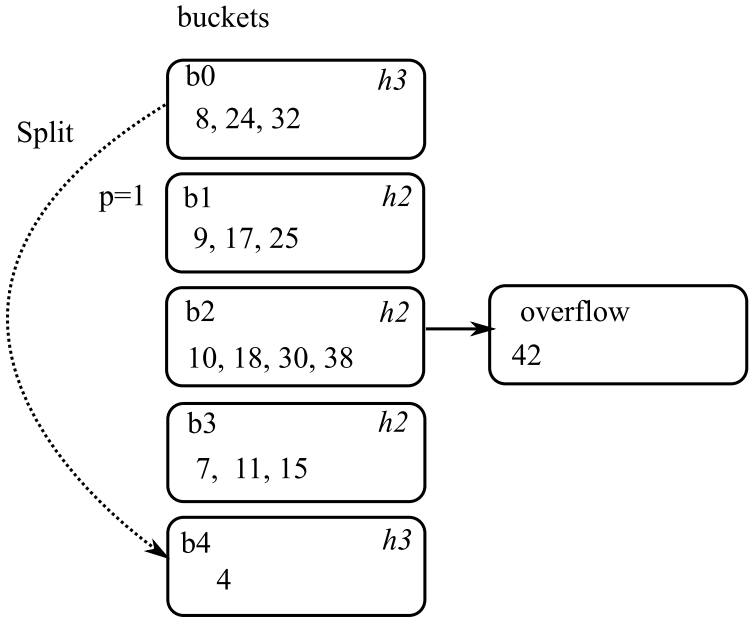
Illustration

Here, $M = 4$ and each bucket holds at most 4 objects (we only show the key). The hash function is $h(k) = k \ mod \ M$.

An object with key 42 must be inserted in bucket b_2. A bucket is linked to b_2, and bucket b_0 (recall that $p = 0$) is split.



Bucket b2 receives a new object



Bucket b0 splits; bucket b2 is linked to a new one

Evolution of the hash function

If we keep unchanged the hash function, all the objects moved to bucket b_4 cannot be found anymore.

Linear hashing actually relies on a *pair* of hash functions (h_n, h_{n+1})

1. $h_n : k \rightarrow k \mod 2^n$
2. $h_{n+1} : k \rightarrow k \mod 2^{n+1}$

h_n applies to the buckets in the range $[p, M-1]$, while h_{n+1} applies to all other buckets.

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 b_3 splits in turn, the hash file is “switched” one level up, the pair of hash function becomes (h_1, h_2), and p is reset to 0.

Dictionary operations

The following computation returns the address a of the bucket that contains a key k:

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

Can be used for insertions, searches, and deletions.

Note: the structure still requires a directory which grows linearly.

Exercises

1. Build a LH file on the list of *départements*.
2. Consider the LH file of the above figure. What happens if we insert an object with key 47 (still assuming that the maximal number of objects in a bucket is 4).

Outline

Issues with hash structures distribution

Straighforward idea: everybody uses the same hash function, and buckets are replaced by servers. Problems:

- Dynamicity. At Web scale, we must be able to add or remove servers at any moment.
- Inconsistencies. It is very hard to ensure that all participants share an accurate view of the system (e.g., the hash function).

Some solutions:

- Distributed linear hashing: sophisticated scheme that allows Client nodes to use an outdated image of the has file; guarantees eventual convergence.
- Consistent hashing: to be presented next.
  NB: consistent hashing is used in several systems, including Dynamo (Amazon)/Voldemort (Open Source), and P2P structures, e.g. Chord.

Distribution strategy: mostly trivial

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 pointer p.

The cluster is a set of servers $\{S_0, S_1, \ldots, S_M\}$, $2^n \leq M < 2^{n+1}$, each holding a bucket.

The bucket of a server S_i overflows? S_p, splits; a new server S_{M+1} is allocated to the structure, and objects are transferred from S_p to S_{M+1} (same as LH).

Fine. So the problem is essentially that of maintaining the global parameters n and p.

How to maintain n and p. Typical scenarios

Fully accurate:

1. each change is broadcasted to all participants.
2. insert and searches are fast; any update must be propagated to servers and clients

Allow some latency in the propagation of updates

1. More flexible: the structure accepts that some participants are informed lazily.
2. More reasonable in a large distributed and/or unstable system.
3. Requests may have to follow a path through several nodes.

The LH* approach

Each server and each Client stores a local copy of the pair (n, p).

This copy may be out-of-date with respect to the "true" parameters n and p used by the file.

A dictionary operation with key k may be sent to a wrong server, due to some distributed file evolution ignored by the Client.

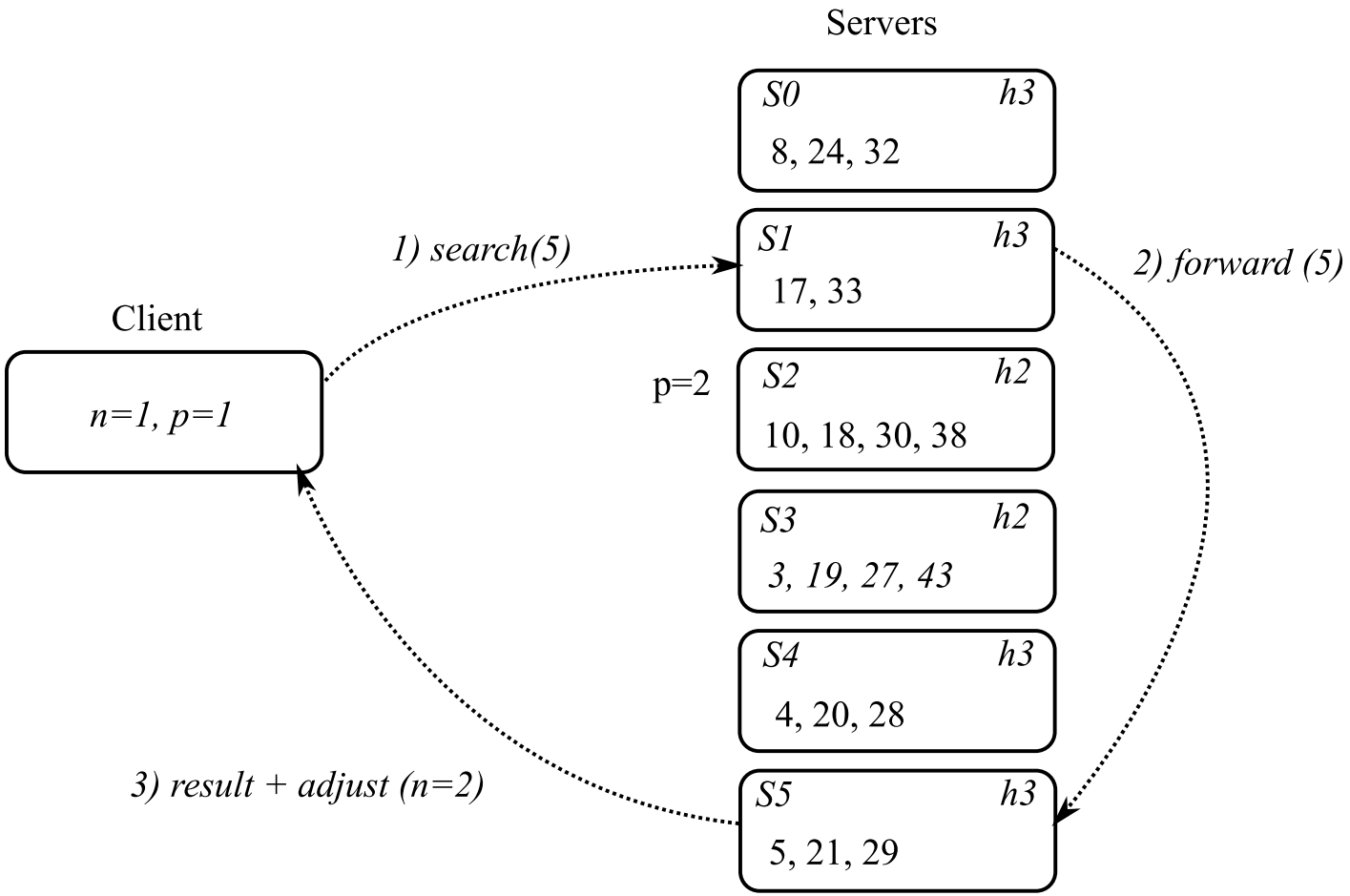
The LH* then applies a *forwarding path* algorithm that eventually leads to the correct server.

Eventualy, the Cliet gets a "fresh" pair (n, p).

The LH* approach – details

Assume that the Client image is $(n_C = 1, p_C = 1)$, whereas several splits led the LH* to the status $(n = 3, p = 2)$.

The Client sends *search(5)*. It computes $a = h_{n_C}(5) = 5 \mod 2^1 = 1$ and the request is sent to S_1.



Consistent hashing

Let N be the number of servers. The following functions

$$hash(key) \rightarrow modulo(key, N) = i$$

maps a pair $(key, value)$ to server S_i.

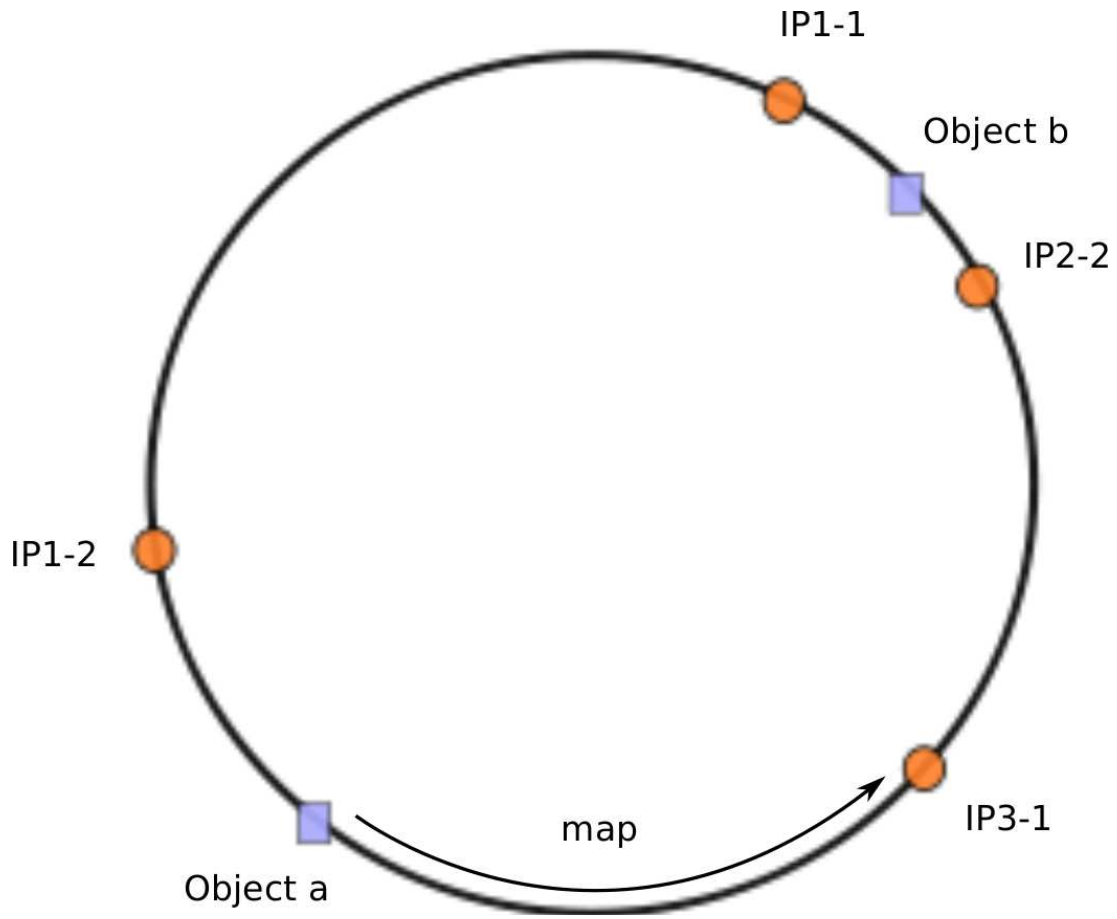
Fact: if N changes, or if a client uses an invalid value for N, the mapping becomes inconsistent.

With Consistent hashing, addition or removal of an instance does not significantly change the mapping of keys to servers.

- a simple, non-mutable hash function h maps both the keys ad the servers IPs to a large address space A (e.g., $[0, 2^{64} - 1]$;
- A is organized as a ring, scanned in clockwise order;
- if S and S' are two adjacent servers on the ring: all the keys in range $]h(S), h(S')]$ are mapped to S'.

Illustration

Example: item *A* is mapped to server IP1-2; item B to server . . .



A server is added or removed? A local re-hashing is sufficient.

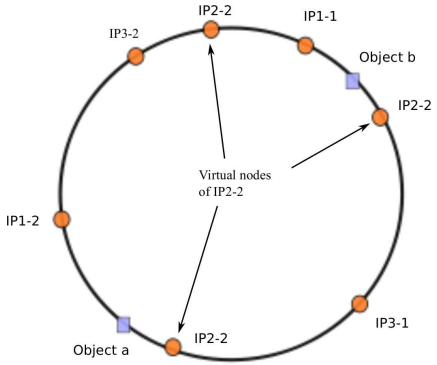
Some (really useful) refinements

What if a server fails? How can we balance the load?

Failure $\Rightarrow$ use replication; put a copy on the next machine (on the ring), then on the next after the next, and so on.

Load balancing $\Rightarrow$ map a server to several points on the ring (virtual nodes)

- the more points, the more load received by a server;
- also useful if the server fails: data rellocation is more evenly distributed.
- also useful in case of heterogeneity (the rule in large-scale systems).



Distributed indexing based on consistent hashing

Main question: where is the hash directory (servers locations)? Several possible answers:

- On a specific (“Master") node, acting as a load balancer. Example: caching systems.
  $\Rightarrow$ raises scalability issues.
- Each node records its successor on the ring.
  $\Rightarrow$ may require $O(N)$ messages for routing queries – not resilient to failures.
- Each node records $\log N$ carefully chosen other nodes.
  $\Rightarrow$ ensures $O(\log N)$ messages for routing queries – convenient trade-off for highly dynamic networks (e.g., P2P)
- Full duplication of the hash directory at each node.
  $\Rightarrow$ ensures 1 message for routing – heavy maintenance protocol which can be achieved through gossiping (broadcast of any event affecting the network topology).

Case study: Dynamo (Amazon)

A distributed system that targets high availability (your shopping cart is stored there!).

- Duplicates and maintains the hash directory at each node via gossiping – queries can be routed to the correct server with 1 message.
- The hosting server replicates N (application parameter) copies of its objects on the N distinct nodes that follow S on the ring.
- Propagates updates asynchronously $\rightarrow$ may result in update conflicts, solved by the application at read-time.
- Use a fully distributed failure detection mechanism (failure are detected by individual nodes when then fail to communicate with others)

An Open-source version is available at *http://project-voldemort.com/*

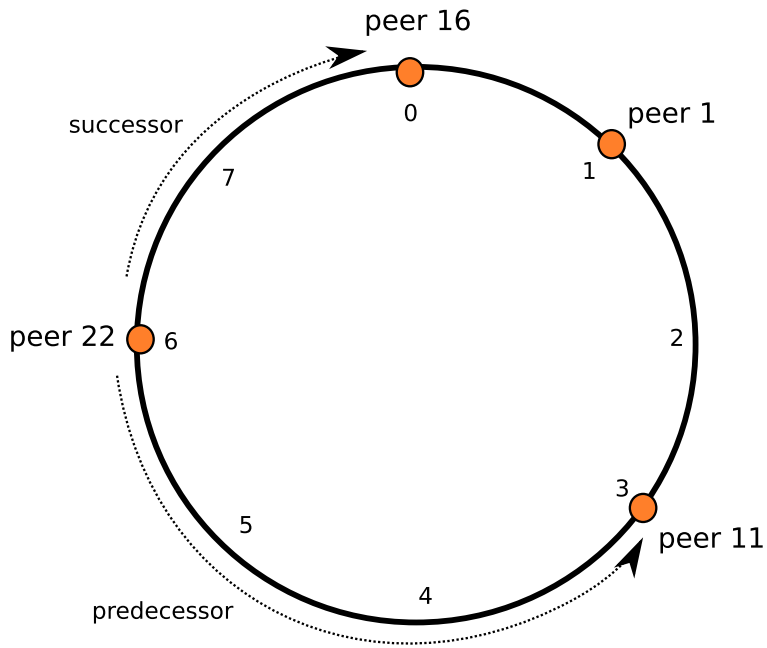
Implementation: Chord Ring

The hash function h takes the URL of a peer and maps it to the address space $[0..2^m - 1]$. Hashing is based on arithmetic modulo 2^m.

$$\text{h} : \text{pId} \rightarrow \textit{pId mod } 2^m$$

h distributes the peers around a *ring*. We assume no two peers have the same $h(pId)$ (m large enough).

The Chord Ring with $m = 3, 2^m = 8$. Each peer with id n is located at node n *mod* 8 on the ring.
Ex: peer 22 is assigned to node 6.



Distribute the keys to the peers

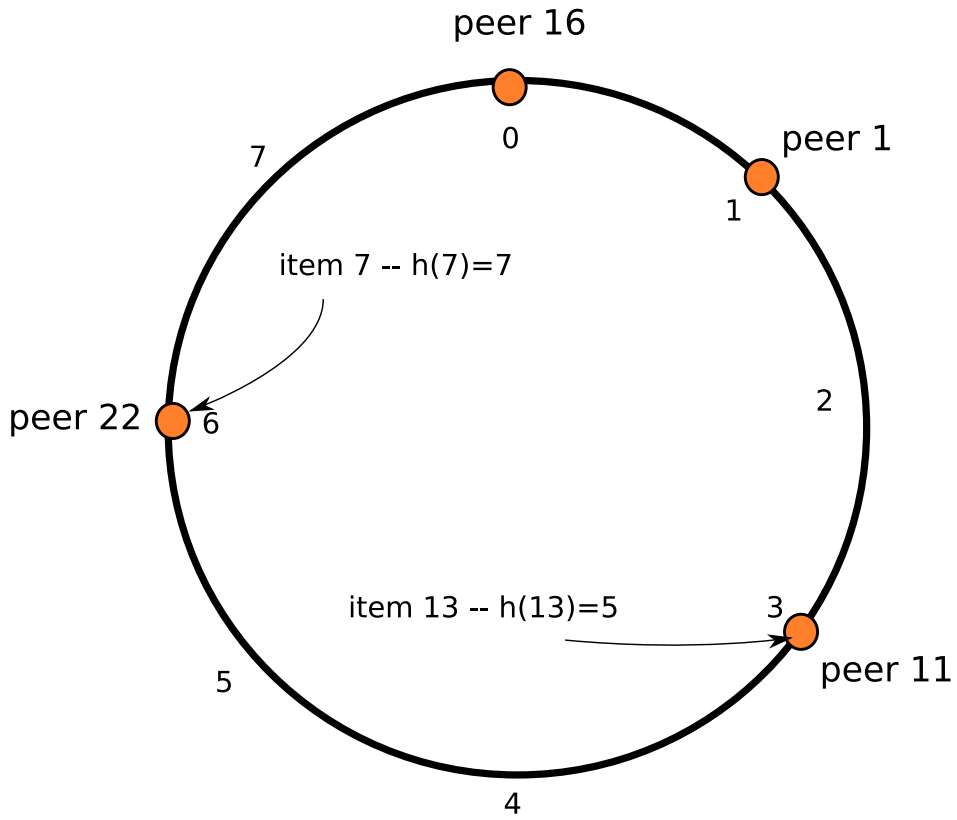
Keys are hashed with function *h* with range $[0..2^m - 1]$

Rule: a key k is assigned to the unique peer p such that

- h(p) $\leq$ h(k)
- and there is no p' such that h(p) $<$ h(p') $\leq$ h(k)

We say that p is *responsible* for key k.

Assignment of item to peer: item 13 is hashed to $h(13) = 5$ and assigned to the peer p with the largest $h(p) \leq 5$.



Routing tables

For each peer p, $friends_p$ contains (at most) $\log 2^m = m$ peer addresses.
For each i in [1..m], the i^{th} 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 $h(p) + 2^{i-1}$.

Examples (Let $m = 10$, $2^m = 1024$; consider peer p with $h(p) = 10$.)

- The first friend p_1 is the peer resp. for $10 + 2^0 = 11$
- The second friend p_2 is the peer resp. for $10 + 2^1 = 12$
- The third friend p_3 is the peer resp. for $10 + 2^2 = 14$
- friend p_7 is the peer resp. for $10 + 2^6 = 74$
- The last friend p_{10} is the peer resp. for $(10 + 512) = 522$

Exercise: express the gap between friends[i] and friends[i+1]

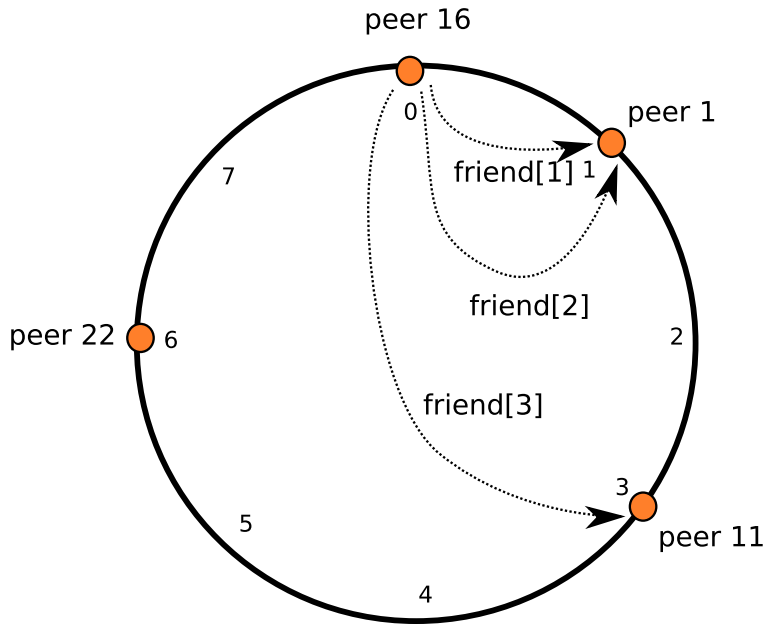
Understanding routing tables

Important properties:

1. a peer maintains a small routing table, e.g., 16 friends for each peer, in a ring with $2^{16} = 65,536$ nodes;
2. each peer knows better the peers close on the ring that the peers far away;
3. a peer p cannot (in general) find directly the peer p' responsible for a key k; *but p can find a friend which holds a more accurate information about k.*

The figure shows the friends of peer p16, located at node 0 (note the collisions). p16 does *not* know p22.

Exercise: what are the friends of p11, located at node 3?



Searching Chord: The *get* algorithm

Peer p looks for index(k):

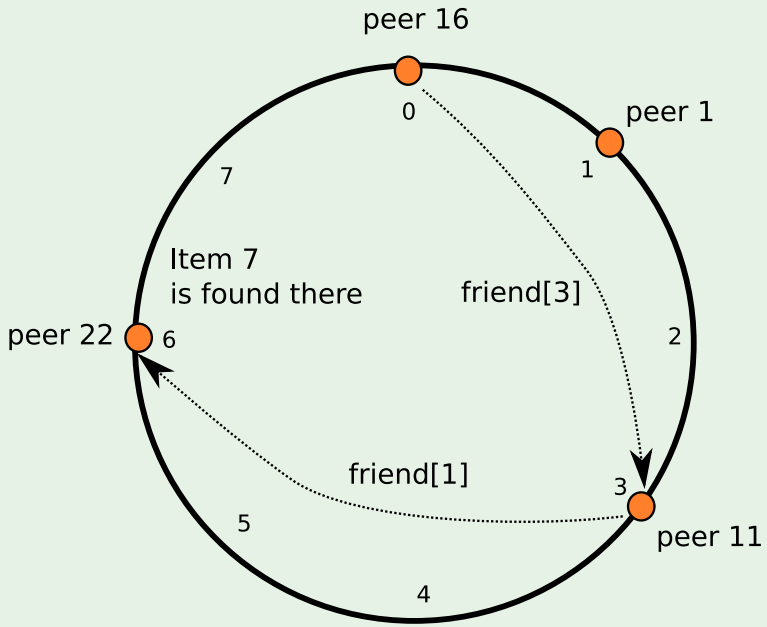
- if p is responsible for k, we are done;
- else let i such that h(p)+$2^{i-1} \leq$ h(k) < h(p)+2^i : forward the search to the friend p_i.

Fact: the search range is (at worse) halved at each step $\rightarrow$ the search converges in $O(\log 2^m) = O(m)$ hops.

Examples (p16 receives a request for item k, with h(k)=7.)

1. p16 forwards the request to p11, its third friend (why?).
2. then p11 forwards to p22, its third friend (why?).
3. p22 find item k locally.

Exercise: develop an example of the worst case with $m = 10$.



Joining Chord

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 carries out three tasks:

- p must initialize its own routing table
  $\Rightarrow$ p' uses its routing table to locate p's friends. Cost: $O\left(\log^2 N\right)$, where N is the current number of nodes.
- the routing table of the existing nodes must be updated to reflect the addition of p;
  $\Rightarrow$ more tricky: see details on next slide.
- finally p takes from its predecessor all the items k such that $h(p) \leq h(k)$.

For highly dynamic networks, Chord relies on a stabilization protocol (not presented).

Details: updating the routing tables of existing nodes

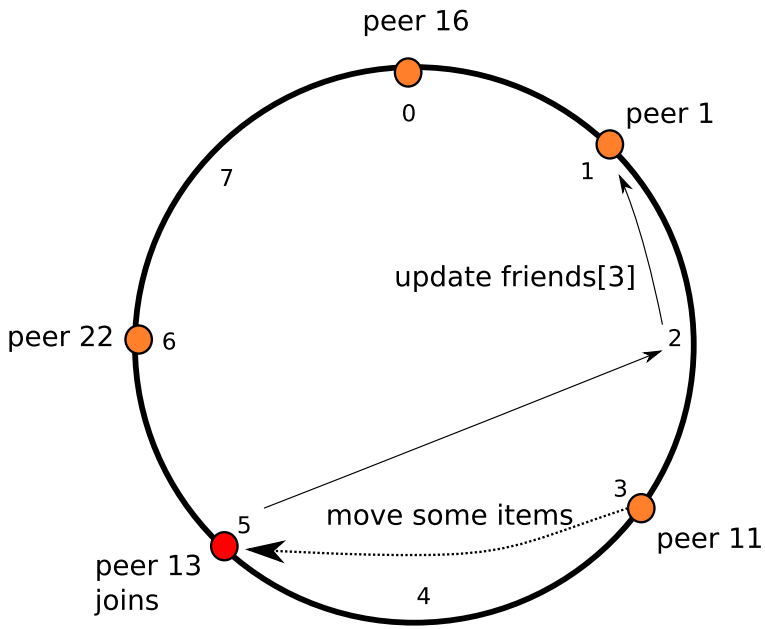
Peer p joins the network. Existing routing tables must be updated

Fact: p becomes the i^{th} friend of a peer p' iff the following conditions hold:

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

Example: Peer 13 joins. It takes the slot 5. Then

1. p13 computes its own routing table (explain how, assuming its contact node is p22).
2. p13 is the third friend of a peer at slot 2 ($5 - 2^{3-2} - 1$) or 1 ($5 - 2^{3-1}$) (check that it's true).
3. p13 moves part of the data stored on peer 11.



Leaving Chord

A peer p may leave “cold” (you know you are leaving $\Rightarrow$ take appropriate measures) or leave “hot” (failure). In both cases

- local index at p must be transmitted to the predecessor (OK if p exits gracefully, but what if p fails?)
- existing routing tables must be updated
- meawhile, the other peers must be ready to fail when they attempt to contact a friend

Two extensions allow to cope with failures:

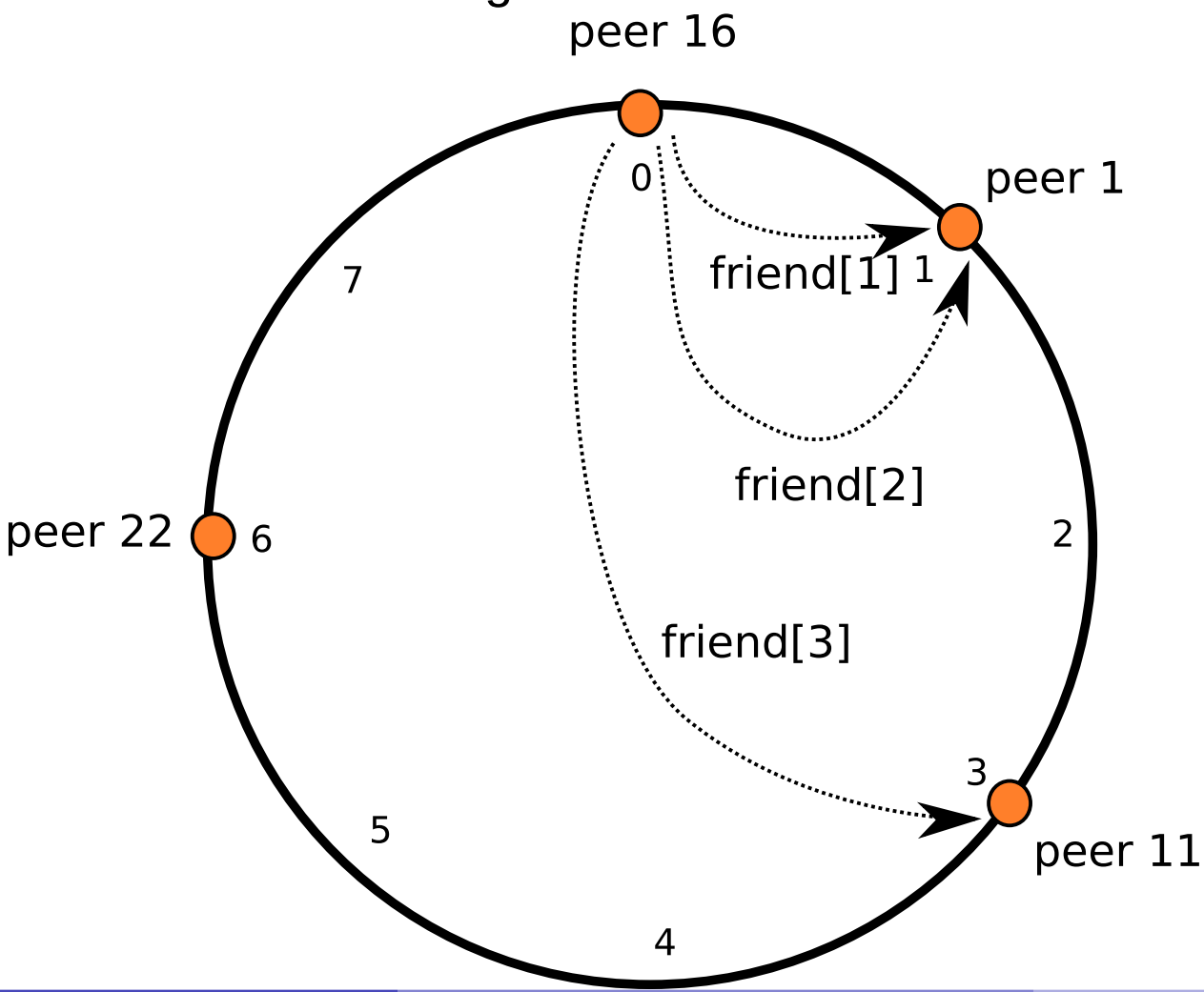
- **Predecessor list**: each peer maintains the list of its r immediate predecessor (r depends on the network churn).
  $\Rightarrow$ if a friend of p does not answer, the query is routed to a predecessor of p which chooses another route.
- **Replication**: items are replicated on the r predecessors.

Exercises

- Someone proposes the following solution to the problem of distributing the hash directory: each node will maintain the hash value and location of its successor. Discuss the advantage and downsides of this solution, and examine in particular the cost of the dictionary operation (insert, delete, search) and network maintenance operations (join and leave).
- Express the gap between $friends[i]$ and $friends[i+1]$ in the routing table of a CHORD peer.

Exercises

- Develop an example of the worst case for the *search()* operation in CHORD, with $m = 10$.
- Consider the CHORD ring below. What are the friends of $p11$, located at



3?