Indexing Large-Scale Data

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

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

November 16, 2010

Outline

What is it about?

The Web is a huge source of information: search engines (Google, Yahoo!) collect and store billions of documents – E-commerce web sites like Amazon manage hundreds of millions of customers – and Facebook, eBay, etc.

Very large datasets – commonly Petabytes, 10^{15} bytes, soon Exabytes (10^{18}), perhaps ultimately Zetabytes (10^{21}), the estimated size of the digital universe.

Distribution is the key to Web Scale data management – it brings scalability, but raises challenging problems (high risk of failure).

Specific requirement:

- low-latency algorithms for "point queries" (e.g., direct access to a few objects) over TBs datasets.

Outline

Guidelines and principles for distributed data management
$\Rightarrow$ in the present talk, focus on cluster-based approaches (vs P2P environments)

Indexing techniques for very large collections

- Hash-based techniques for (*key*, *value*) models.
  $\Rightarrow$ Illustration with the Dynamo system (Amazon)
- Tree-based structures supporting range queries
  $\Rightarrow$ Illustration with Bigtable (Google)

Indexing data in a distributed setting

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*:

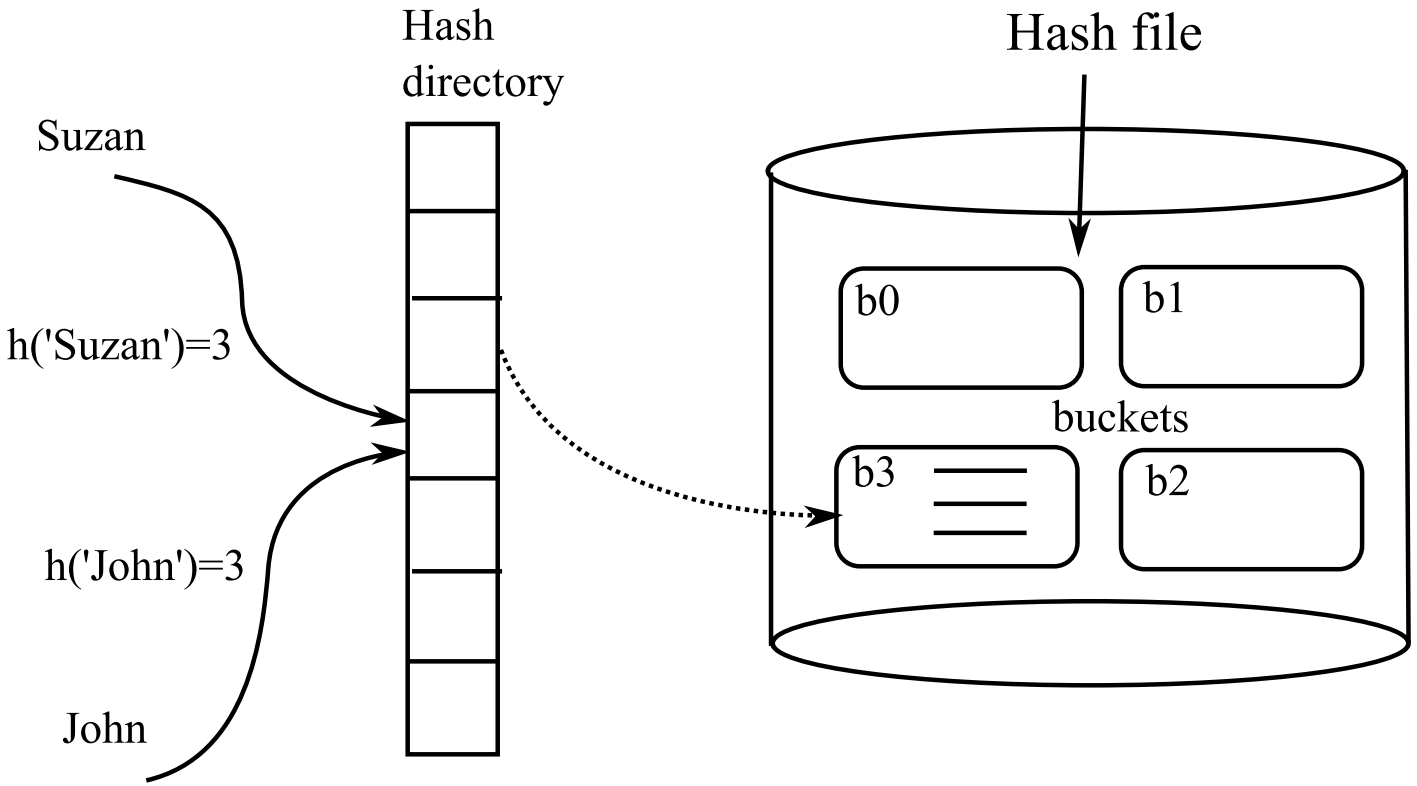
1. insertion *insert*(k, v),
2. deletion *delete*(k),
3. key search *search*(k): v.
4. (optional) range search *range*(k_1, k_2): $\{v\}$.

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

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.

Issues with hash structures distribution

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

Two issues:

- 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 accurante 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 hash 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.

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.

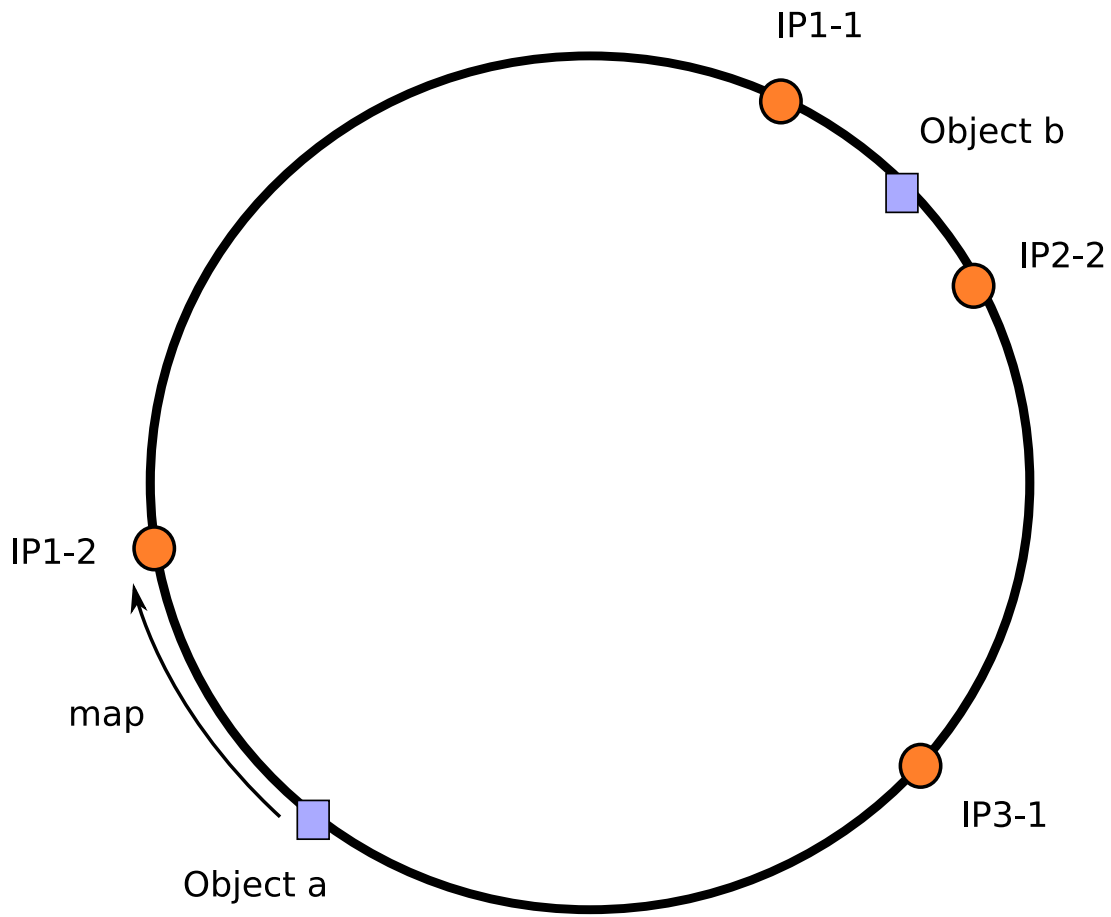
Fact: if N changes, or if a client uses an invalid value for N, the mapping becomes inconsistent.

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_1 and S_2 are two adjacent servers on the ring: all the keys in range $]h(S_1), h(S_2)]$ are mapped to S_2.

Illustration

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



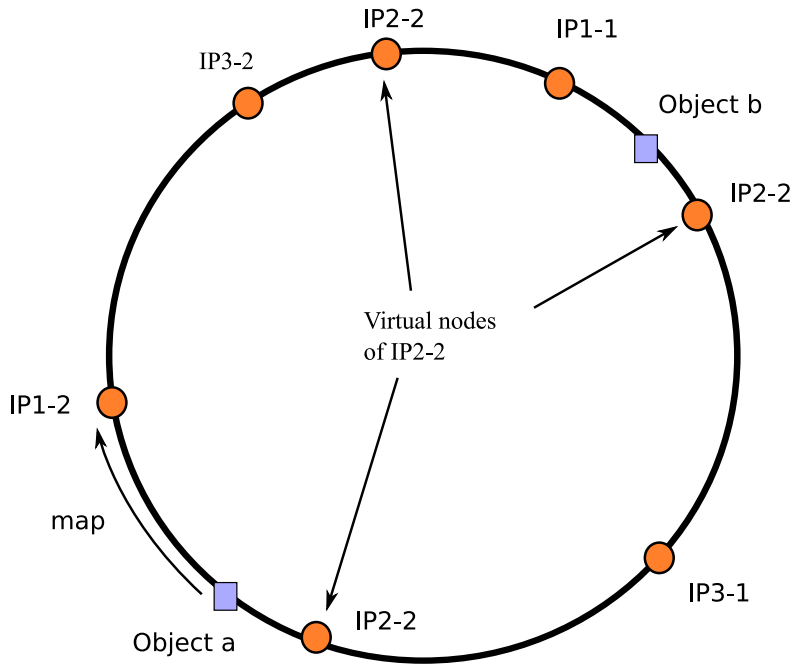
A server is added or removed? A local re-hashing is sufficient.

Some (really useful) refinements

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

Failure ⇒ use replication; put a copy on the next machine (on the ring), then on the next after the next, and so on.
Load balancing ⇒ 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, Chord system)
- 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!).

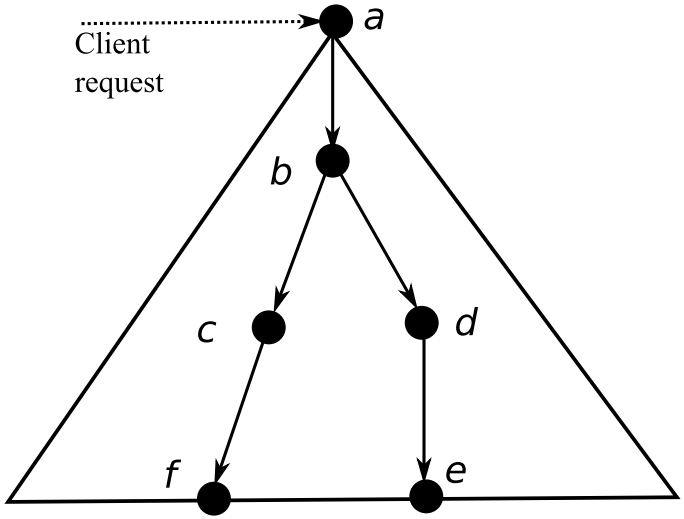
- Duplicates and maintains the hash directory at each node via gossiping – queries can be routed to the correct server with 1 message.
- The hosting server replicates N (application parameter) copies of its objects on the N distinct nodes that follow S on the ring.
- Propagates updates asynchronously $\Rightarrow$ 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/*

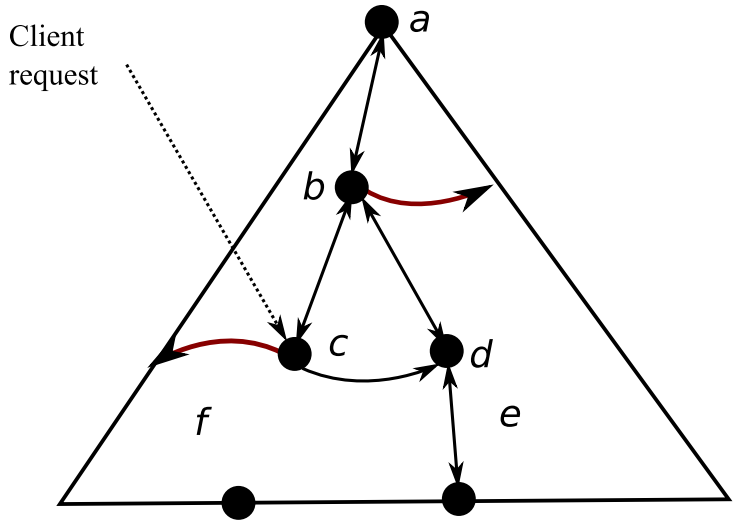
Outline

Issues with search trees distribution

All operations follow a top-down path ⇒ potential factor of non-scalability



Standard tree



With local routing nodes

Solutions for distributed structures:

1. *caching* of the tree structure on the Client node
2. *replication* of parts of the tree
3. *routing tables*, stored at each node, enabling horizontal navigation in the tree.

Case study: Bigtable

Can be seen as a distributed *map* structure, with features taken from B-trees, and from non-dense indexed files.

Context:

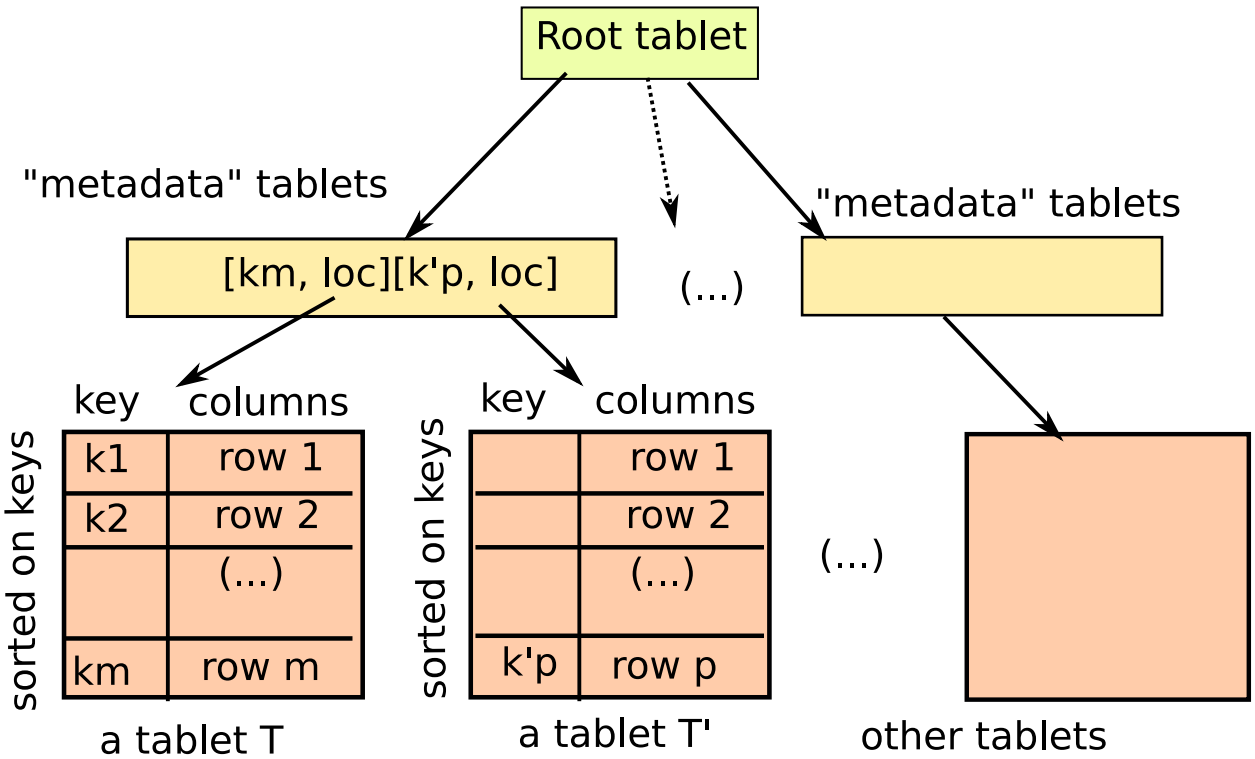
- a controlled environment, with homogeneous servers located in a Data Center;
- a stable organization, with long-term storage of large structured data;
- a data model (column-oriented tables with versioning)

Design:

- close to a B-tree, with large capacity leaves
- scalability is achieved by a cache maintained by Client nodes.

Overview of Bigtable structure

Leaf level: a "table" organized in "rows" indexed by a key. Rows are stored in lexicographic order on the key values.

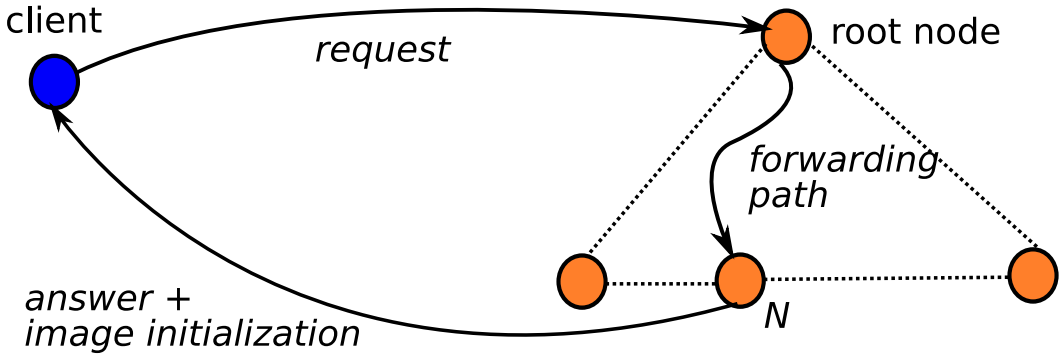


The table is partitioned in "tablets", and tablets are indexed by upper levels.

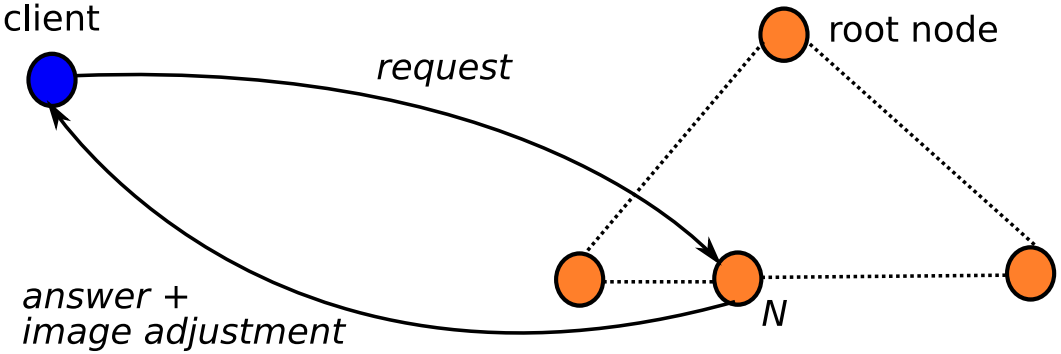
Full tablets are split, with upward adjustment.

Architecture: one Master - many Servers

The Master maintains the root node and carries out administrative tasks.



a) A new client contacts a distributed system

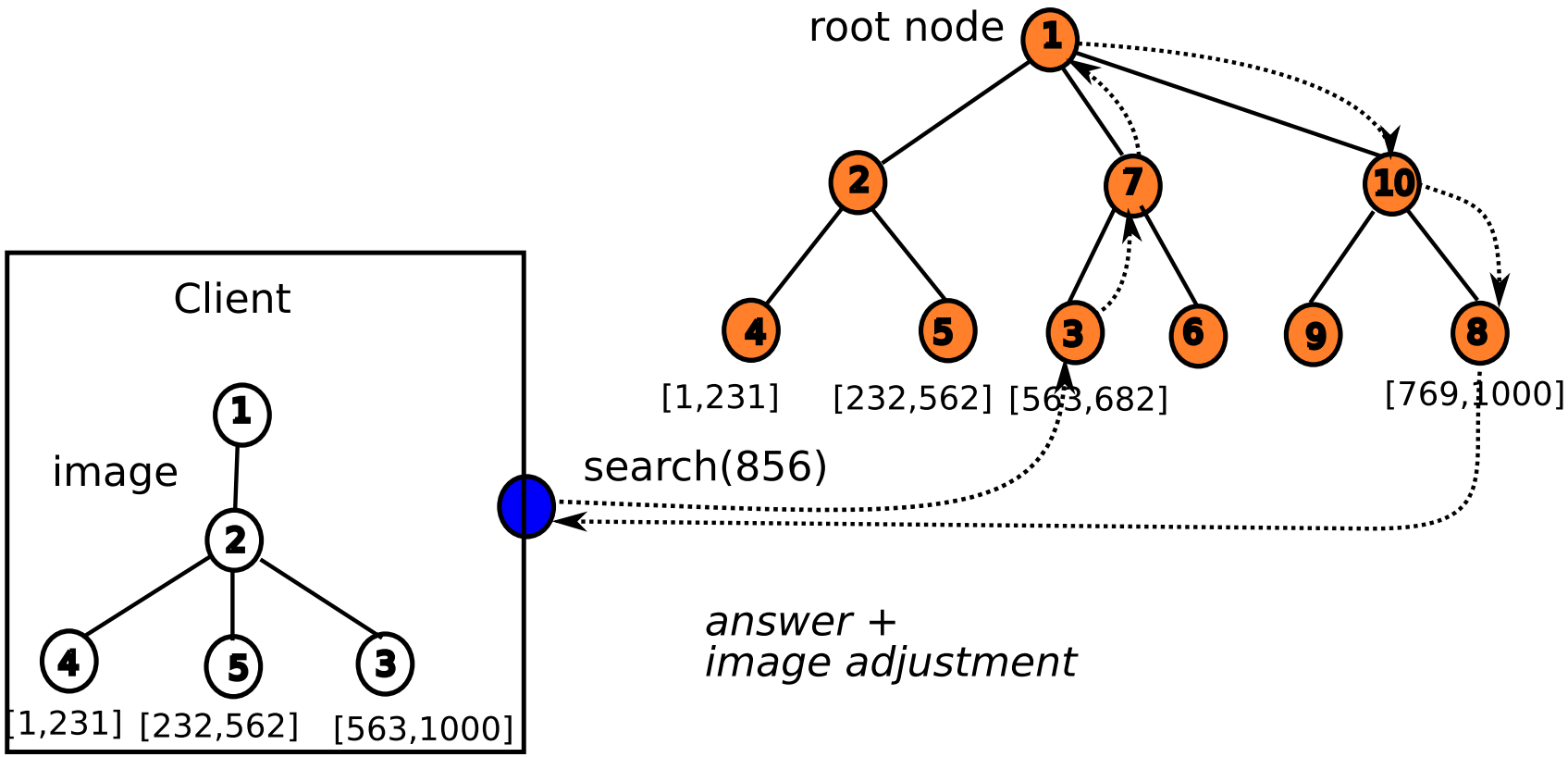


b) Using its image, the client directly contacts N

Scalability is obtained with Client cache that stores a (possibly outdated) image of the tree.

Example of an out-of-range request followed by an adjtment

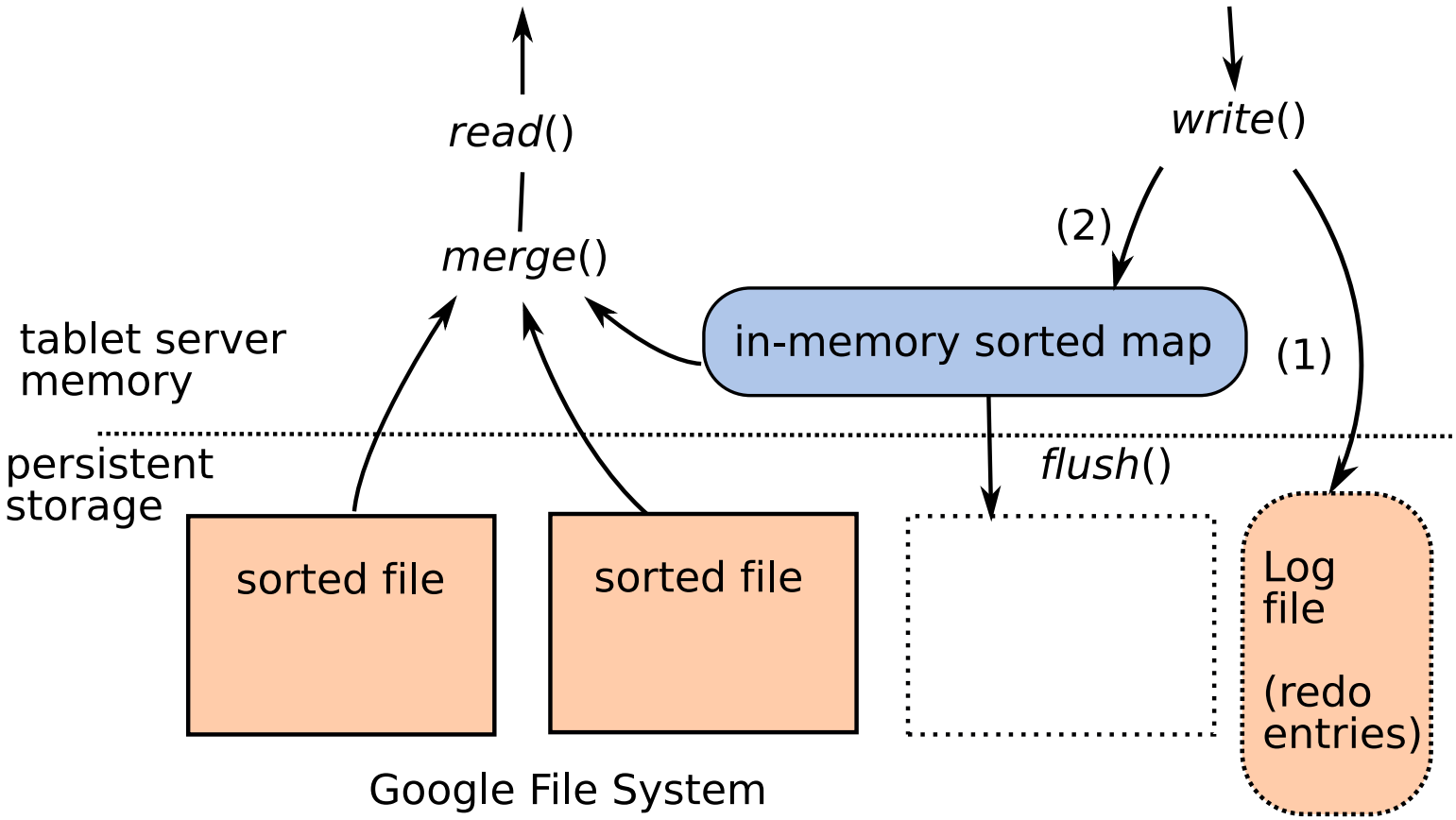
A Client request may fail, due to an out-of-date image of the tree.



An adjustment requires at most *height(Tree)* rounds of messages.

Persistence management in Bigtable

Problem: how can we maintain the sorted structure of tablets?



Outline

Distributed indexing: what you should remember

Key point: Scalability. No single point of failure; even load distribution over all the nodes. Technical means:

- Distribute (and maintain) routing information.
  $\Rightarrow$ trade-off between maintenance cost and operations cost.
- Cache an image of the structure (e.g., in the Client).
  $\Rightarrow$ design a convergence protocol if the image gets outdated.

Key point: efficiency. Clearly depends on the amount of information replicated at each node or at the Cient.

- Stable systems: the structure can be duplicated at each node. Allows $O(1)$ cost – low maintenance.
- Highly dynamic systems: very hard to maintain a consistent view of the structure for each participant.

Always: be ready to face a failure somewhere; detect failures, use and replication and deal with it.

The end!

For references, slides, and public chapters of the book:

http://webdam.inria.fr/textbook

Contact the authors for login/password.

References

americonarticle Jeffrey Dean and Sanjay Ghemawat.
MapReduce: Simplified Data Processing on Large Clusters.
In *Intl. Symp. on Operating System Design and Implementation (OSDI)*, pages 137–150, 2004.