Distributed Indexing

Les Houches – May 18, 2010

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

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

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

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

Specific requirements:

1. efficient and reliable batch analysis of very large collections.
2. 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
⇒ 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.
  ⇒ Illustration with the Dynamo system (Amazon)
- Tree-based structures supporting range queries
  ⇒ Illustration with Bigtable (Google)
- Distributed batch processing: MapReduce.

Distributed computing techniques

- Example of a low-level programming model: MapReduce
- Expressive data processing languages (Pig Latin)

Data Management in Clouds – Illustration with Amazon EC2.

Outline

Distributed systems

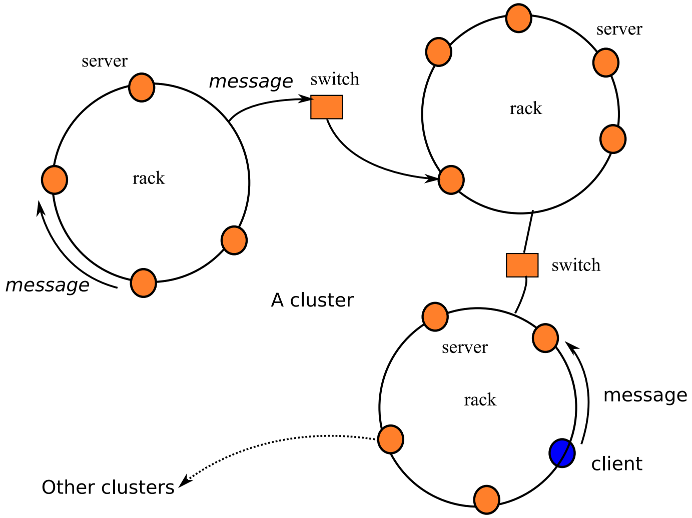
A *distributed system* is an application that coordinates the actions of several computers.

This coordination is achieved by exchanging *messages* which are pieces of data that convey some information.
⇒ "shared-nothing" architecture -> no shared memory

The system relies on a network that connects the computers and handles the routing of messages.
⇒ Local area networks (LAN), Peer to peer (P2P) networks, . . .

LAN-based infrastructure: clusters of machines

Three communication levels: "racks", clusters, and groups of clusters.



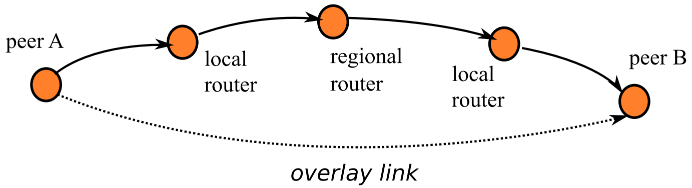
Example: Google (mid-2009)

1. $\approx$ 40 servers per rack;
2. $\approx$ 150 racks per data center (cluster)
3. how many clusters? Google's secret, and constantly evolving ...

Rough estimate: 150-200 data centers? 1,000,000 servers?

P2P infrastructure: Internet-based communication

Nodes, or "peers" communicate with messages sent over the Internet network.



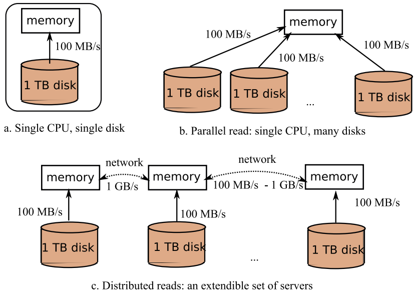
The physical route may consist of 10 or more forwarding messages, or "hops".

Performance

Type	**Latency**	**Bandwidth**
Disk	$\approx 5 \times 10^{-3}$s (5 millisec.);	At best 100 MB/s
LAN	$\approx 1 - 2 \times 10^{-3}$s (1-2 millisec.);	$\approx$ 1GB/s (single rack); $\approx$ 100MB/s (switched);
Internet	Highly variable. Typ. 10-100 ms.;	Highly variable. Typ. a few MBs.;

Distribution, why?

Sequential access. It takes 166 minutes (more than 2 hours and a half) to read a 1 TB disk.
Parallel access. With 100 disks, assuming that the disks work in parallel and sequentially: about 1mn 30s.
Distributed access. With 100 computers, each disposing of its own local disk: each CPU processes its own dataset.

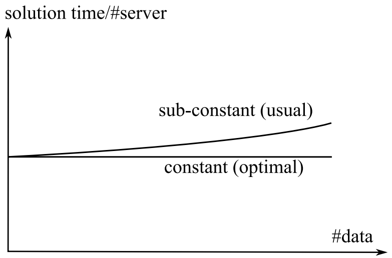


Scalability

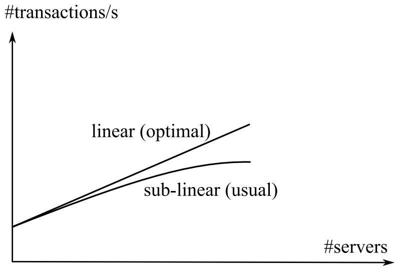
The latter solution is *scalable*, by adding new computing resources.

Properties of a distributed system: (i) scalability

Scalability refers to the ability of a system to continuously evolve in order to support an evergrowing amount of tasks.



Weak scaling: the solution time per server remains constant as the dataset grows



Strong scaling: the global throughput raises linearly with the number of servers (fixed problem size)

A scalable system should (i) distribute evenly the task load to all participants, and (ii) ensure a negligible distribution management cost.

Properties of a distributed system: (ii) efficiency

Two usual measures of its efficiency are the response time (or latency) which denotes the delay to obtain the first item, and the throughput (or bandwidth) which denotes the number of items delivered in a given period unit (e.g., a second).

Unit costs:

1. *number of messages* globally sent by the nodes of the system, regardless of the message size;
2. *size of messages* representing the volume of data exchanges.

Properties of a distributed system: (iii) availability and consistency

Availability is the capacity of of a system to limit as much as possible its latency.

Involves several aspects:

- Replication (caches):
- Quick restart on failures: monitor the participating nodes to detect failures as early as possible (usually via “heartbeat” messages);

Consistency: essentially, ensures that the system faithfully reflects the actions of a user.

- Strong consistency (ACID properties) – often requires a (slow) synchronous replication, and possibly heavy locking mechanisms.
- Weak consistency – accept to serve some requests with outdated data.
- Eventual consistency – same as before, but the system is guaranteed to converge towards a consistent state based on the last version.

Standard RDBMS favor consistency over availability – one of the reasons (?) of the 'NoSQL' trend.

Outline

Preliminaries

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 its (physical) address of v. It supports *dictionary operations*:

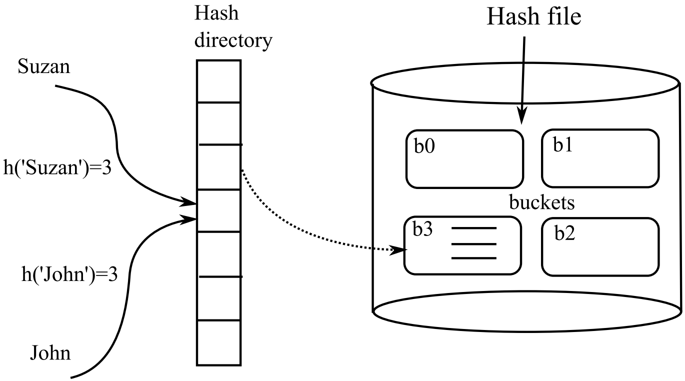
1. insertion *insert*(k, a),
2. deletion *delete*(k),
3. key search *search*(k)*:* a.
4. (optional) range search *range*$(k_1,\ k_2)$*:* $\{a\}$.

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 will.
- 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 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.

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.

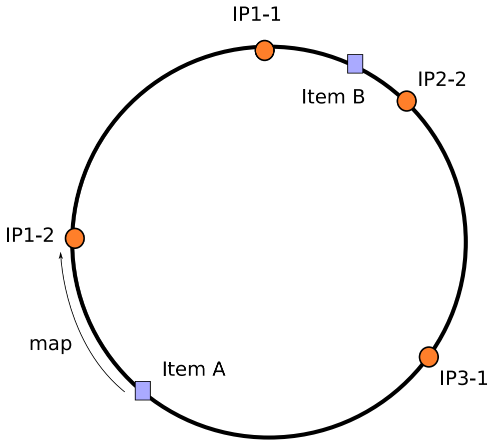
Fact: if N changes, or if a client uses an invalid value for N, the mapping becomes inconsistent.

With Consistent hashing, addition or removal of an instance does not significa

- a simple, non-mutable hash function h maps both the keys to a 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 necessary.

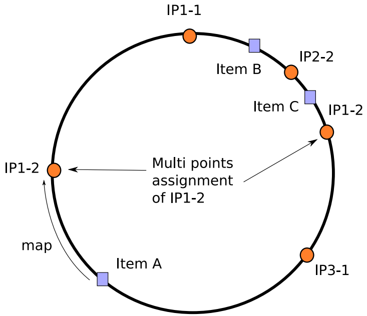
Some (really useful) refinements

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.
  $\Rightarrow$ probably not a scalable architecture.
- 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 $O(N)$ 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!).

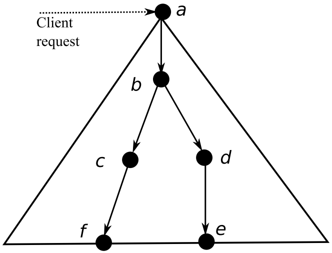
- Duplicates and maintains the hash directory at each node via gossiping – queries can be routed to the correct server with 1 message.
- The hosting server replicates N (application parameter) copies of its objects on the N distinct nodes that follow S on the ring.
- Propagates updates asynchronously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

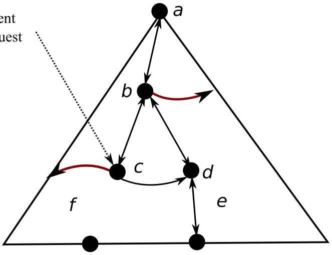
Outline

Issues with search trees distribution

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



Standard tree



With local routing nodes

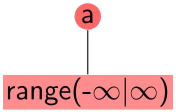
Solutions for distributed structures:

1. *caching* of the tree structure on the 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 1: BATON (P2P)

Conceptually: a standard binary search tree.

each node covers a range and contains all objects whose key belongs to the range.



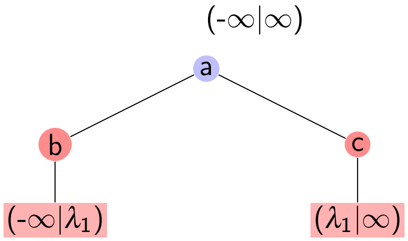
Case study 1: BATON (P2P)

Conceptually: a standard binary search tree.

When a server is added, a split occurs, and objects are evenly distributed.

A split generates a routing node and a data node – They can be allocated to a same server.

The range of a routing node covers its subtree.



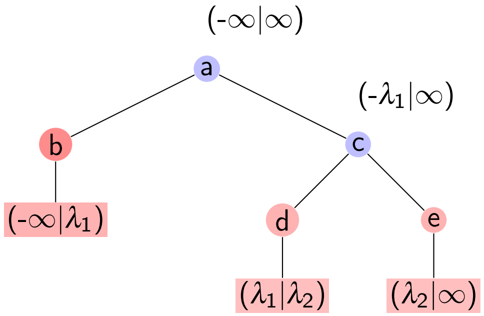
Case study 1: BATON (P2P)

Conceptually: a standard binary search tree.

The tree grows by splitting leaves and adding a local routing node.

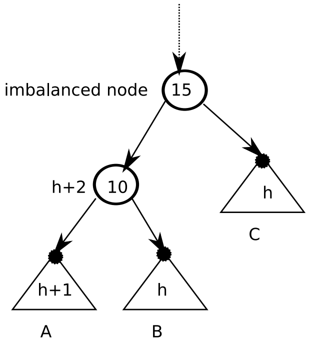
The tree is balanced iff, at each node, the subtrees heights do not differ by more than 1 (e.g., AVL trees).

With non-uniform datasets, split may lead to imbalance.

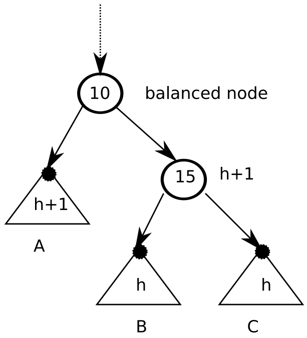


Balancing the tree

When th tree gets imbalanced A *rotation* is required (still similar to AVL trees):



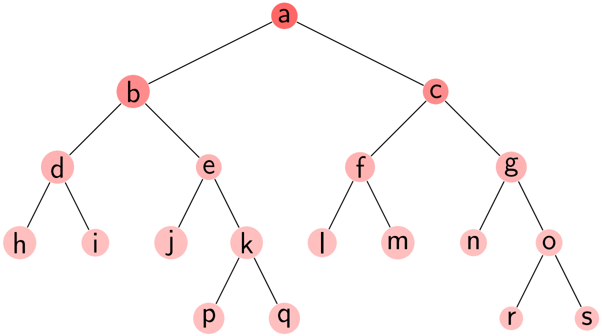
A split occurred in A



After a rotation

The approach is still non scalable – every path goes through the root.

A complete example



If we do not add some information: node a receives all the messages, node b receives half of the messages, node d 1/4 of the messages, etc (for uniform query distr.).

⇒ we will partially replicate the tree structure at each node to balance the query load.

Routing tables

Each node stores *routing tables*, that consist of:

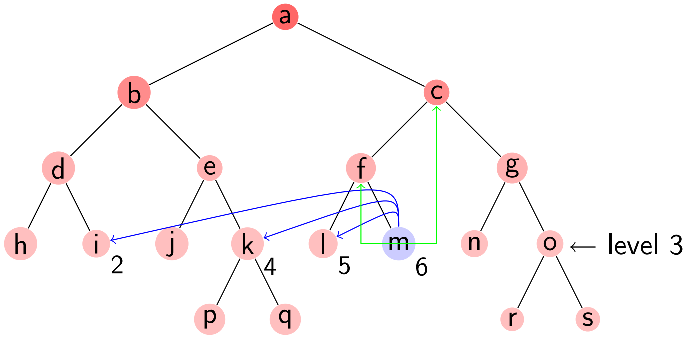
1. parent, left child and right child addresses;
2. previous and next adjacent nodes in in-order traversal;
3. left and right routing tables, that reference nodes at the same level and at position $pos + / - 2^i, i = 0, 1, 2, \ldots$.

Ideas

1. the amount of replication is limited (each node knows a number of “friends" which is logarithmic in the total number of nodes)
2. each node knows better the nodes which are close, than nodes which are far.

Routing tables: example

The left routing table (blue edges) refers to nodes at respective positions $6-2^0=5$, $6-2^1=4$, and $6-2^2=2$.



Note that the gaps between two friends f_i and f_{i+1} gets larger as i increases ($2^{i+1}-2^i=2^i$).
The number of friends is $\log N$, N being the number of nodes in the considered level.

The routing table of node m

Node m must maintain the following information

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	$[o_{min}, o_{max}]$

$\Rightarrow$ heavy work when something changes in the network.

Search operations

A *search(k)* request is sent by a Client node to any peer p in the structure. Two steps:

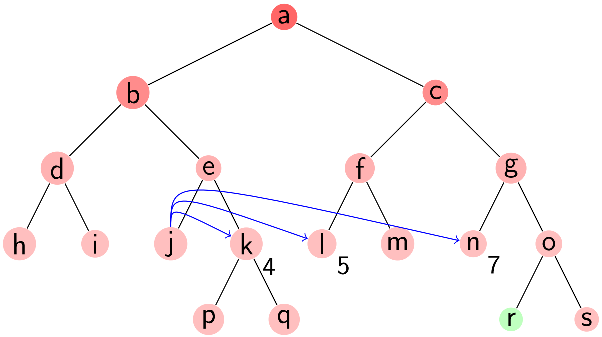
- (horizontal) p looks in its routing table for a node p' at the same level that covers k
  $\rightarrow$ p' is not a friend of p? then there is a friend of p that knows p' better than p.
- (top-down) from p', a standard top-down path is followed.

Procedure: p chooses the farthest friends p" whose lower bound is smaller than k

Search space halved at each step $\Rightarrow$ ensures that p' is found after at most $\log N$ iterations.

Example of search

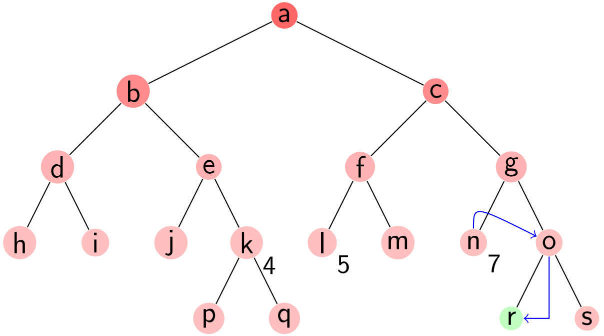
Assume a request sent to node j for a key that belongs to node r



Blue edges: the (right) friends of j; so j must forward the request to n, its farthest friends whose lower bound is smaller than k.

Example of search

Now *n* looks in its own routing table to forward the search.



n knows this part of the tree better than *j*: it finds *o*, the ancestor of *r*, and a downward path is then initiated.

Case study 2: Bigtable

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

Context: very different from Baton.

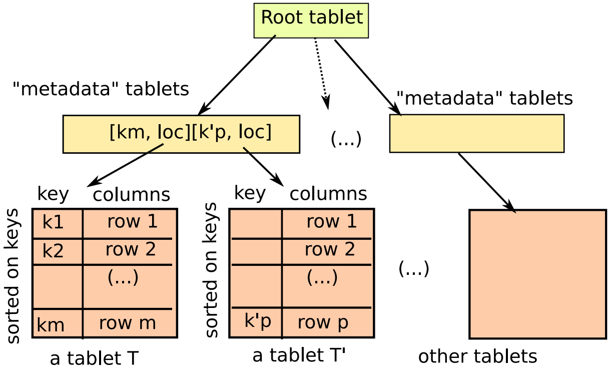
- a controlled environment, with homogeneous servers located in a Data Center;
- a stable organization, with long-term storage of large structured data;
- a data model (column-oriented tables with versioning)

Design: very different as well

- close to e 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.

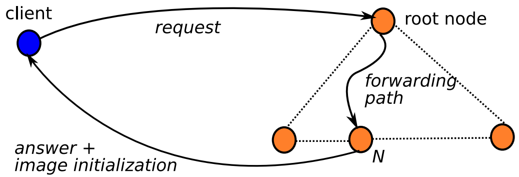


The table is partitioned in "tablets", and tablets are indexed by upper levels.

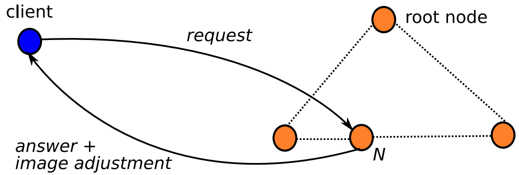
Full tablets are split, with upward adjustment.

Architecture: one Master - many Servers

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



a) A new client contacts a distributed system

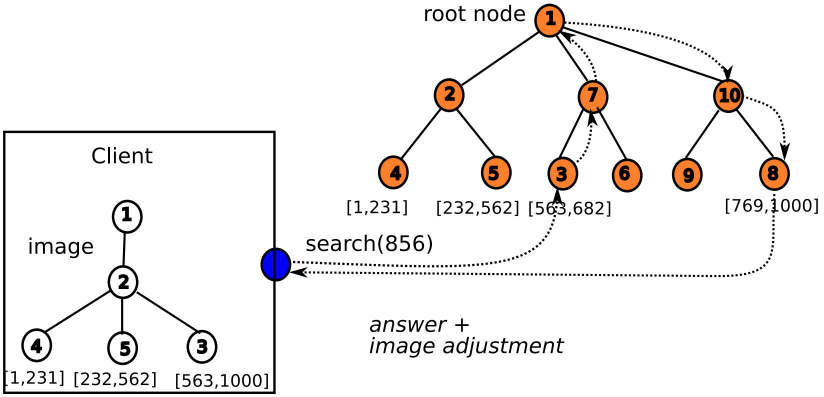


b) Using its image, the client directly contacts N

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 adjutment

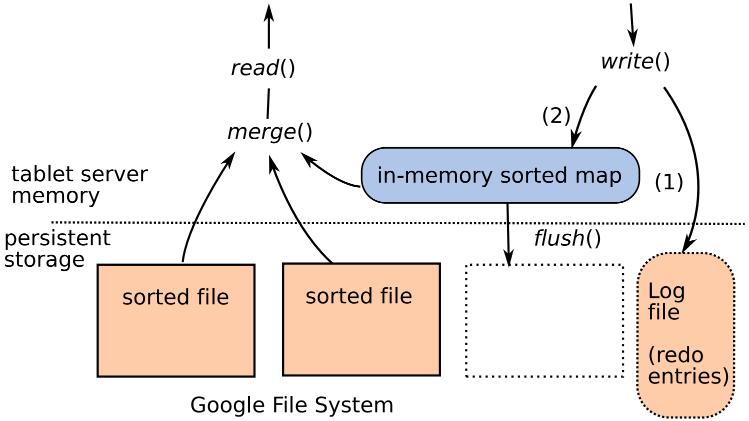
A Client request may fail, due to an out-of-date image of the tree.



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

Persistence management in Bigtable

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



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: availability. Use replication and monitor the system. Issues:

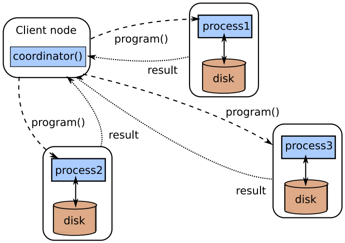
- Synchronous replication is costly: choose between strong consistency and strong availability.
  $\Rightarrow$ design a conflict resolution protocol in case of weak consistency.
- Always be ready to face a failure somewhere.
  $\Rightarrow$ location of replica shoud make a quick restart easy.

Outline

History and development of MapReduce

Published by Google Labs in 2004 at OSDI [DG04]. Implemented in Hadoop, widely used (Yahoo!, Amazon EC2).

A programming model (inspired by standard functional programming operators) to facilitate the development and execution of distributed tasks.



Main idea: "push the program near the data". The programmer defines two functions; MapReduce takes in charge distribution aspects.

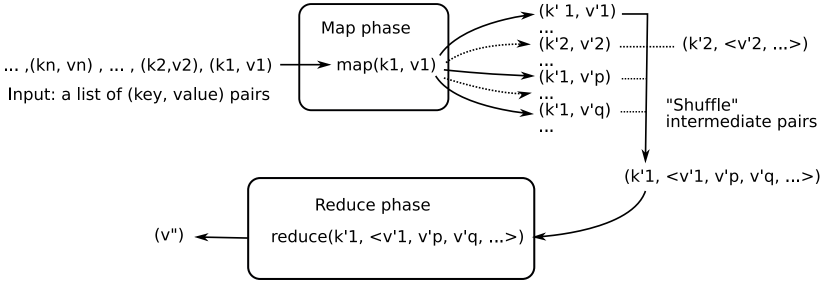
The programming model of MapReduce

Used to process data flows of $(key, value)$ pairs.

1. *map()* takes as input a list of pairs $(k, v) \in K_1 \times V_1$ and produces (for each pair), another *list* of pairs $(k', v') \in K_2 \times V_2$, called *intermediate pairs*.
   Example: take a pair $(uri, document)$, produce list of pairs $(term, count)$
2. (shuffle) the MapReduce execution environment groups intermediate pairs on the key, and produces grouped instances of type $(K_2, list(V_2))$
   Example: intermediate pairs for term 'job' are grouped as $('job', < 1, 4, 2, 8 >)$
3. *reduce()* operates on grouped instances of intermediate pairs $(k'_1, < v'_1, \cdots, v'_p, \cdots, v'_q, \cdots >)$; Each instance processed by the procedure outputs a result, usually a single value v''.
   Example: take a grouped instance $('job', < 1, 4, 2, 8 >)$ and output the sum: 15

Job workflow in MapReduce

Important: each pair, at each phase, is processed independently from the other pairs.


... ,(kn, vn) , ... , (k2,v2), (k1, v1)
Input: a list of (key, value) pairs
Map phase
map(k1, v1)
(k' 1, v'1)
...
(k'2, v'2)
...
(k'1, v'p)
...
(k'1, v'q)
...
(k'2, <v'2, ...>)
"Shuffle" intermediate pairs
(k'1, <v'1, v'p, v'q, ...>)
Reduce phase
reduce(k'1, <v'1, v'p, v'q, ...>)
(v")

Network and distribution are transparently managed by MapReduce.

Example: Counting terms occurrences in documents

The *map()* function:

```
mapCW(String key, String value):
  // key: document name
  // value: document contents
  for each term t in value:
    return (t, 1);
```

Note: we do not even need to count the occurrences of *t* in *value*.

Counting terms occurrences in documents (cont')

The *reduce()* function. It takes as input a grouped instance on a *key*. The nested list can be scanned with an iterator.

```
reduceCW(String key, Iterator values):
  // key: a term
  // values: a list of counts
  int result = 0;

  // Loop on the values list; cumulate in result
  for each v in values:
    result += v;

  // Send the result
  return result;
```

And, finally, the Job Driver program which submits both functions.

```
// A specification object for MapReduce execution
MapReduceSpecification spec;

// Define input files
  MapReduceInput* input = spec.add_input();
  input->set_filepattern("/movies/*.xml");
  input->set_mapper_class("MapWC");

// Specify the output files:
MapReduceOutput* out = spec.output();
out->set_filebase("/gfs/freq");
out->set_num_tasks(100);
out->set_reducer_class("ReduceWC");

// Now run it
MapReduceResult result;
if (!MapReduce(spec, &result)) abort();
  // Done: 'result' structure contains result info
   return 0;
```

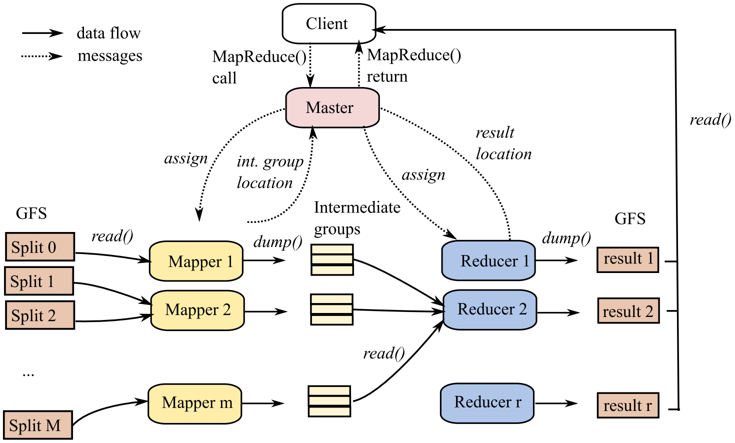
Processing *map()* and *reduce()* as a MapReduce job.

A MapReduce *job* takes care of the distribution, synchronization and failure handling. Specifically:

- the input is split in M groups; each group is assigned to a *mapper* (assignment is based on the data locality principle);
- each mapper processes a group and stores the intermediate pairs locally;
- grouped instances are assigned to *reducers* thanks to a hash function.
- (Shuffle) intermediate pairs are sorted on their key by the reducer;
- one obtains grouped instances, submitted to the *reduce()* function;

NB: the data locality does no longer hold for the Reduce phase, since it reads from the mappers.

Distributed execution of a MapReduce job.



Failure management

In a distributed setting, the specific job handled by a machine is only a minor part of the overall computing task.

Moreover, because the task is distributed on hundreds or thousands of machines, the chances that a problems occurs somewhere are much larger; starting the job from the beginning is not a valid option.

The Master periodically checks the availability and reacheability of the "Workers"

1. if a reducer fails, its task may be reassigned;
2. if a mapper fails, its task must be started from scratch, even in the Reduce phase (it holds intermediate groups).
3. if the Master fails? Then the whole job should be re-initiated.

Pig Latin

Motivation: define high-level languages that use MapReduce as an underlying data processor.

A Pig Latin statement is an operator that takes a relation as input and produces another relation as output.

Pig Latin statements are generally organized in the following manner:

1. A **LOAD** statement reads data from the file system as a *relation* (list of tuples).
2. A series of "transformation" statements process the data.
3. A **STORE** statement writes output to the file system; or, a **DUMP** statement displays output to the screen.

Statements are executed as composition of MapReduce jobs.

An example

Example: load file from meteo sensors; compute the maximal temperature by year

```
-- Load files as a relation
records = LOAD 'meteo.txt'
       AS (year: char, temperature: int, quality: int);

filtered_records = FILTER RECORD
       BY quality = 1 OR quality = 4;

grouped_records = GROUP filtered_records BY year;

max_temp = FOREACH grouped_records GENERATE group,
        MAX (filtered_records.temperature);

DUMP max_temp;
```

What you should remember on distr. computing

MapReduce is a simple model for batch processing of very large collections.
$\Rightarrow$ good for data analytics; not good for point queries (high latency).

The systems brings robustness against failure of a component and transparent distribution.
$\Rightarrow$ more expressive languages required (Pig); could MapReduce be used as an infrastructure for very large scale database distribution (e.g., HadoopDB)?

Parallel databases exist for a long time (TeraData); they use distribution for scalability.

Overview of existing systems

- Google File System (GFS). Distr. storage for very large, unstructured files. Open-source: HDFS (Hadoop)
- Dynamo. Internal data store of Amazon. Open source: Voldemort project.
- Bigtable, sorted distributed storage. Open source: HTable (Hadoop).
- Amazon proposes a Cloud environment, EC2, with: S3 and Hadoop/MapReduce.

The end!

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

http://webdam.inria.fr/textbook

Processing the *WordCount()* example (1)

Let the input consists of documents, say, one million 100-terms documents of approximately 1 KB each.

The split operation distributes these documents in groups of 64 MBs: each group consist of 64,000 documents. Therefore $M : \lceil 1,000,000/64,000 \rceil \approx 16,000$ groups.

We assume a global count of 1,000 distinct terms in the chunk; the (local) Map phase produces 6,400,000 pairs (t, c).

Let $hash(t) = t \mod 1,000$. Each intermediate group $i, 0 \leq i < 1000$ contains 6,400 pairs, each with 6-7 distinct terms t such that $hash(t) = i$.

Processing the *WordCount()* example (2)

Assume that $hash('call') = hash('mine') = hash('blog') = i = 100$. We focus on three Mappers M^p, M^q and M^r:

1. $G_i^p =(<\ldots, ('mine', 1), \ldots, ('call',1), \ldots, ('mine',1), \ldots, ('blog', 1) \ldots>$
2. $G_i^q =(< \ldots, ('call',1), \ldots, ('blog',1), \ldots>$
3. $G_i^r =(<\ldots, ('blog', 1), \ldots, ('mine',1), \ldots, ('blog',1), \ldots>$

R_i reads G_i^p, G_i^p and G_i^p from the three Mappers, sorts their unioned content, and groups the pairs with a common key:

$\ldots, ('blog', <1, 1, 1, 1>), \ldots, ('call', <1, 1>), \ldots, ('mine', <1, 1, 1>)$

Our $reduce_{WC}()$ function is then applied by R_i to each element of this list. The output is *('blog', 4), ('call', 2) and ('mine', 3)*.

Amazon's Elastic MapReduce

Relies on Hadoop running on the web-scale infrastructure of Amazon EC2 – uses Amazon S3 for distributed storage.
`aws-mrjob`

References

Azza Abouzeid, Kamil Bajda-Pawlikowski, Daniel J. Abadi, Alexander Rasin, and Avi Silberschatz.
HadoopDB: An Architectural Hybrid of MapReduce and DBMS Technologies for Analytical Workloads.
Proceedings of the VLDB Endowment (PVLDB), 2(1):922–933, 2009.

Hung chih Yang, Ali Dasdan, Ruey-Lung Hsiao, and Douglas Stott Parker Jr.
Map-reduce-merge: simplified relational data processing on large clusters.
In *SIGMOD*, pages 1029–1040, 2007.

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.

Alan Gates, Olga Natkovich, Shubham Chopra, Pradeep Kamath, Shravan Narayanam, Christopher Olston, Benjamin Reed, Santhosh Srinivasan, and Utkarsh Srivastava.
Building a HighLevel Dataflow System on top of MapReduce: The Pig Experience.
Proceedings of the VLDB Endowment (PVLDB), 2(2):1414–1425, 2009.

Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy.
Hive - A Warehousing Solution Over a Map-Reduce Framework.
Proceedings of the VLDB Endowment (PVLDB), 2(2):1626–1629, 2009.