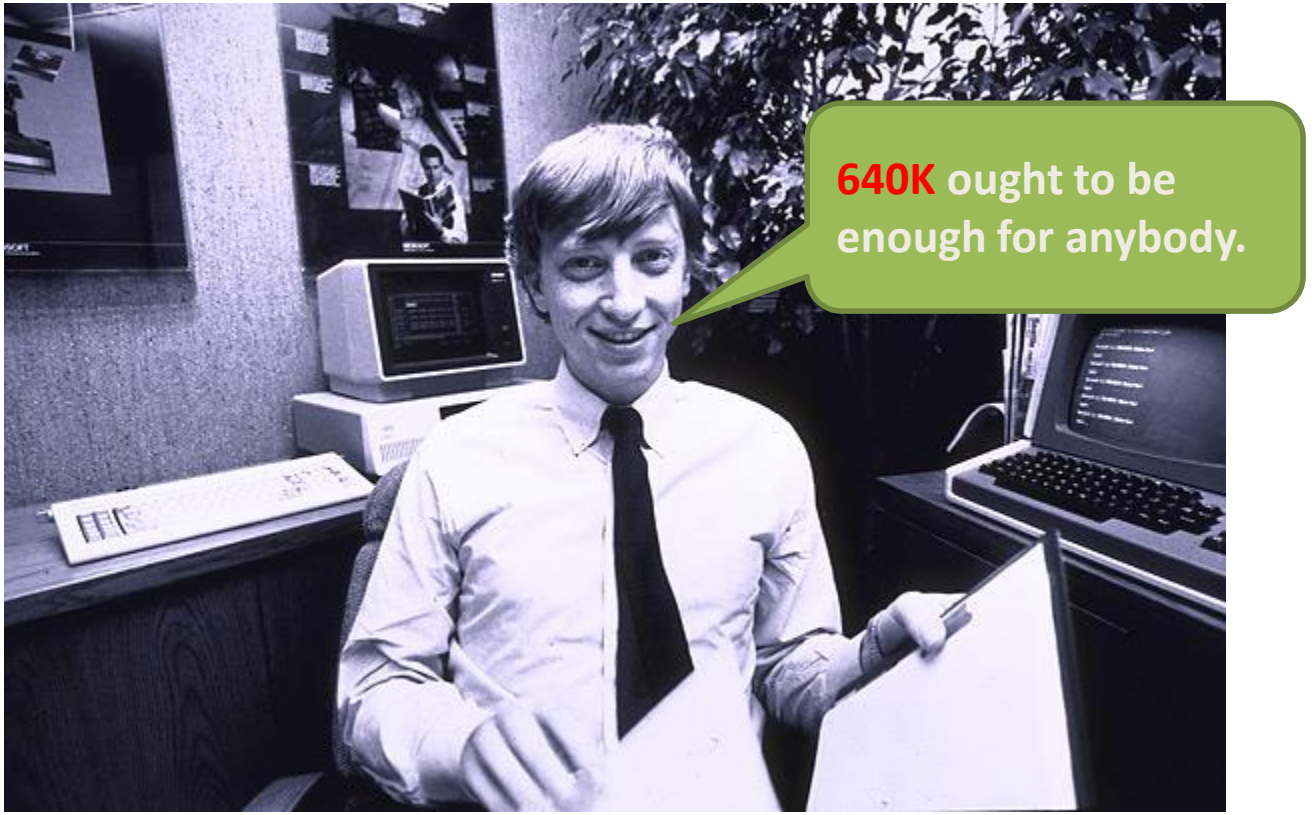
Cloud Computing

Big Data

Dell Zhang
Birkbeck, University of London
2018/19

The Challenges and Opportunities of Big Data

Why Big Data?



Big Data Everywhere!

- Lots of data are being collected and warehoused
  - Web data, e-commerce
  - Purchases at department/ grocery stores
  - Bank/Credit Card transactions
  - Social Network

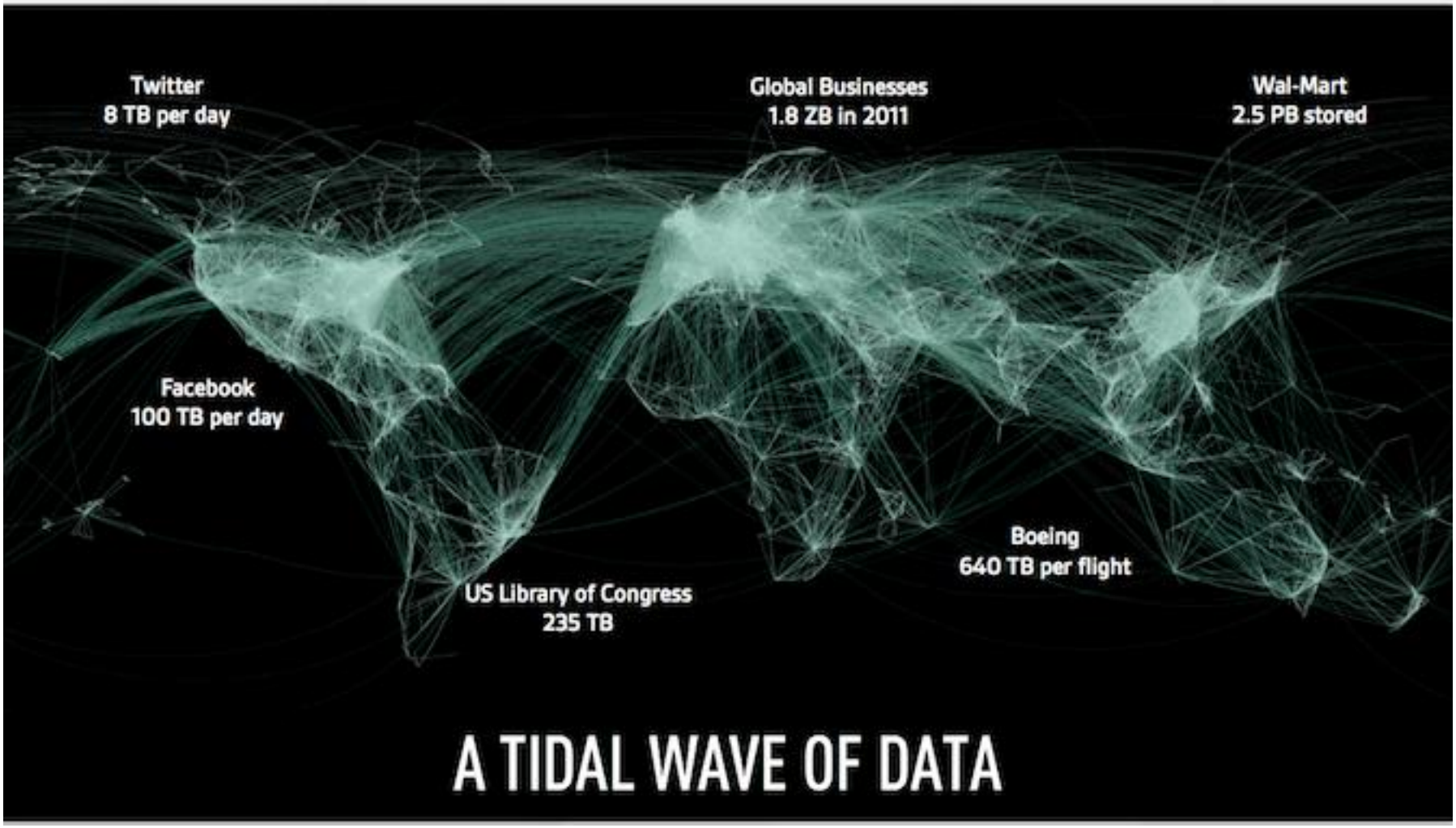
Big Data Challenges (3Vs)

- Big **V**olume
  - the quantity of generated and stored data
- Big **V**elocity
  - the speed at which the data is generated and processed
- Big **V**ariety
  - the type and nature of the data

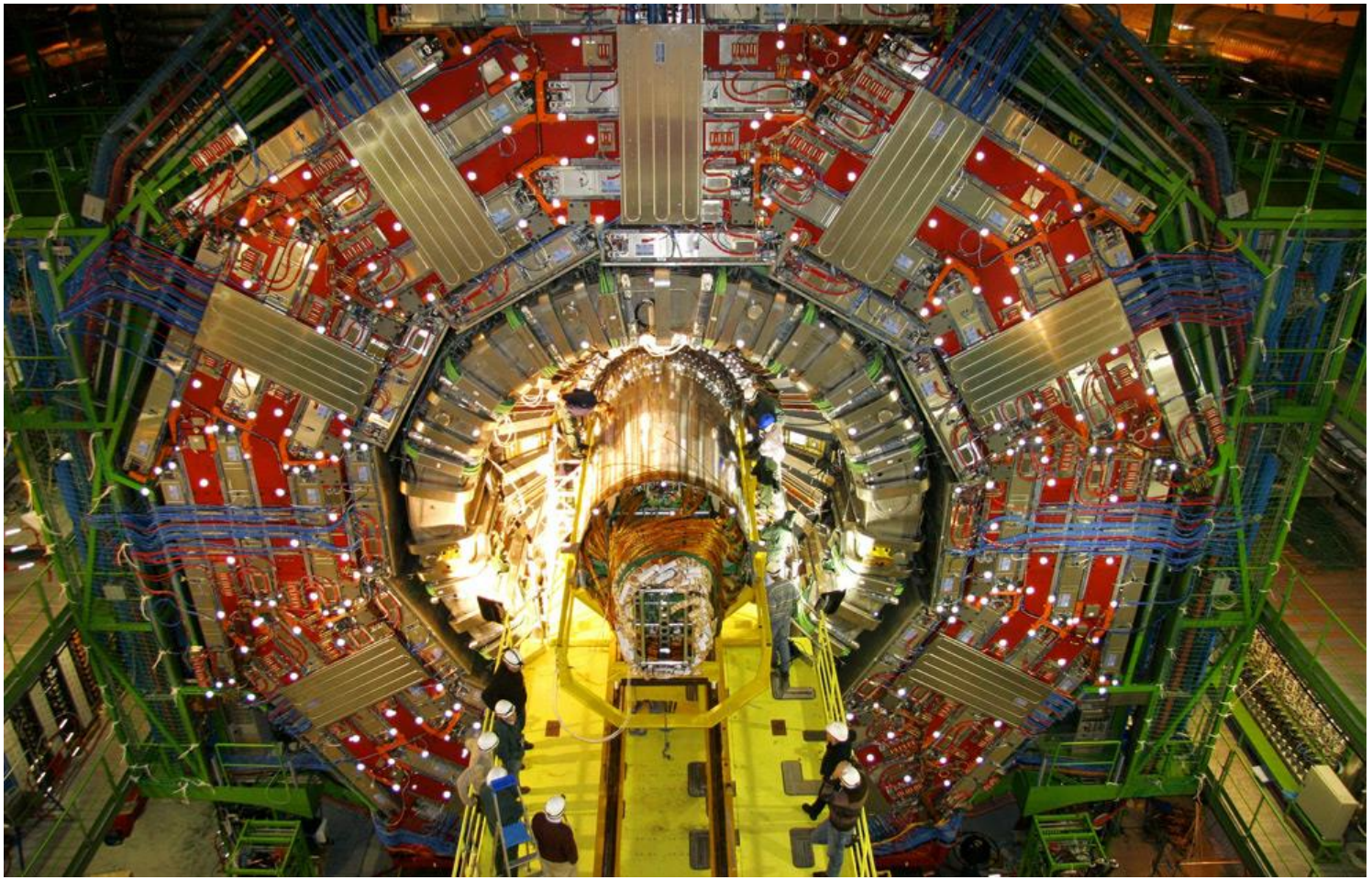

Don't forget to visit www.boyshome.ro

How Much Data?

- Google processes 20 PB a day (2008)
- Wayback Machine has 3 PB + 100 TB/month (3/2009)
- Facebook has 2.5 PB of user data + 15 TB/day (4/2009)
- eBay has 6.5 PB of user data + 50 TB/day (5/2009)
- CERN's Large Hydron Collider (LHC) generates 15 PB a year


Twitter
8 TB per day
Global Businesses
1.8 ZB in 2011
Wal-Mart
2.5 PB stored
Facebook
100 TB per day
Boeing
640 TB per flight
US Library of Congress
235 TB
A TIDAL WAVE OF DATA

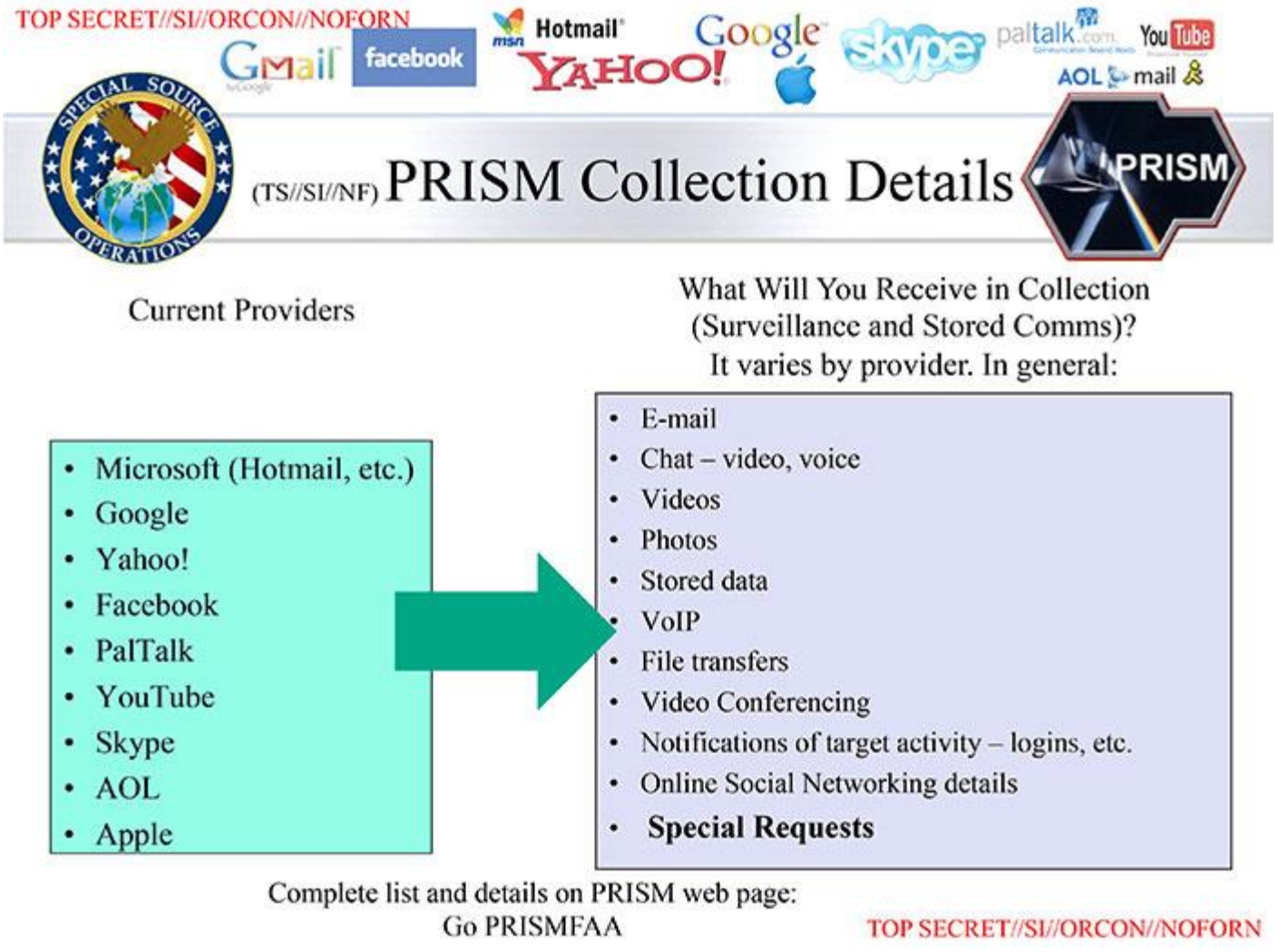
The Large Hadron Collider



Maximilien Brice, © CERN

The Earthscope Project

The PRISM


TOP SECRET//SI//ORCON//NOFORN
(TS//SI//NF) PRISM Collection Details
PRISM
Current Providers
What Will You Receive in Collection (Surveillance and Stored Comms)? It varies by provider. In general:
Microsoft (Hotmail, etc.)
Google
Yahoo!
Facebook
PalTalk
YouTube
Skype
AOL
Apple
E-mail
Chat – video, voice
Videos
Photos
Stored data
VoIP
File transfers
Video Conferencing
Notifications of target activity – logins, etc.
Online Social Networking details
Special Requests
Complete list and details on PRISM web page:
Go PRISMFAA
TOP SECRET//SI//ORCON//NOFORN

Boundless Informant

- It is a big data analysis and data visualization system used by the United States National Security Agency (NSA). [Wikipedia]
  - According to published slides, Boundless Informant leverages Free and Open Source Software — and is therefore "available to all NSA developers" — and corporate services hosted in the cloud.
  - The tool uses HDFS, MapReduce, and Accumulo (formerly Cloudbase) for data processing.

What Kinds of Data?

- Structured Data
  - Tables/Transactions/Legacy Data, …
- Semi-Structured Data
  - XML, …
- Unstructured Data
  - Text Data: the Web, …
  - Graph Data: Social Networks, …

What to Do with Big Data?

- Aggregation and Statistics
  - Data Warehouse and OLAP
- Indexing, Searching, and Querying
  - Keyword based Search
  - Pattern Matching (XML/RDF)
- Knowledge Discovery
  - Data Mining
  - Statistical Modeling

What to Do with Big Data?

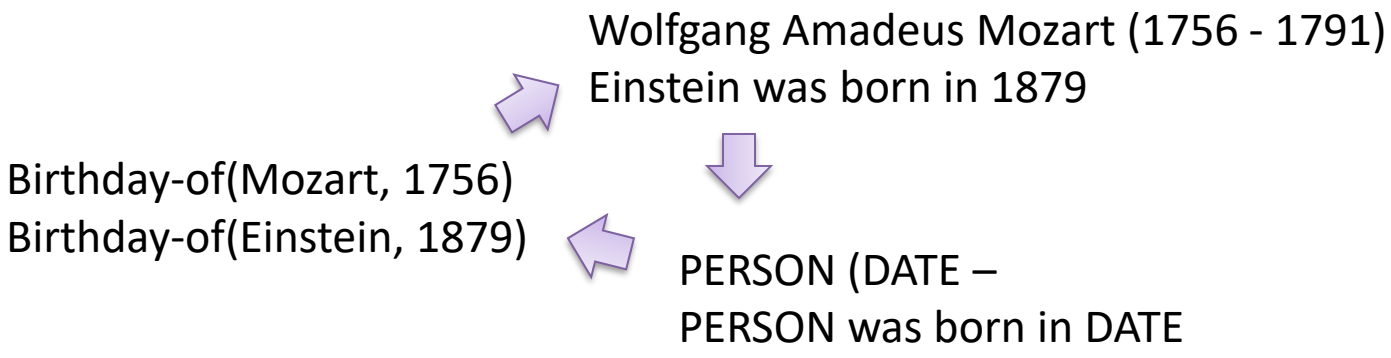
- Example: Answering factoid questions
  - Pattern matching on the Web
  - Works amazingly well

Who shot Abraham Lincoln? → search "* shot Abraham Lincoln"

(Brill et al., TREC 2001; Lin, ACM TOIS 2007)

What to Do with Big Data?

- Example: Learning relations
  - Start with seed instances
  - Search for patterns on the Web
  - Using patterns to find more instances



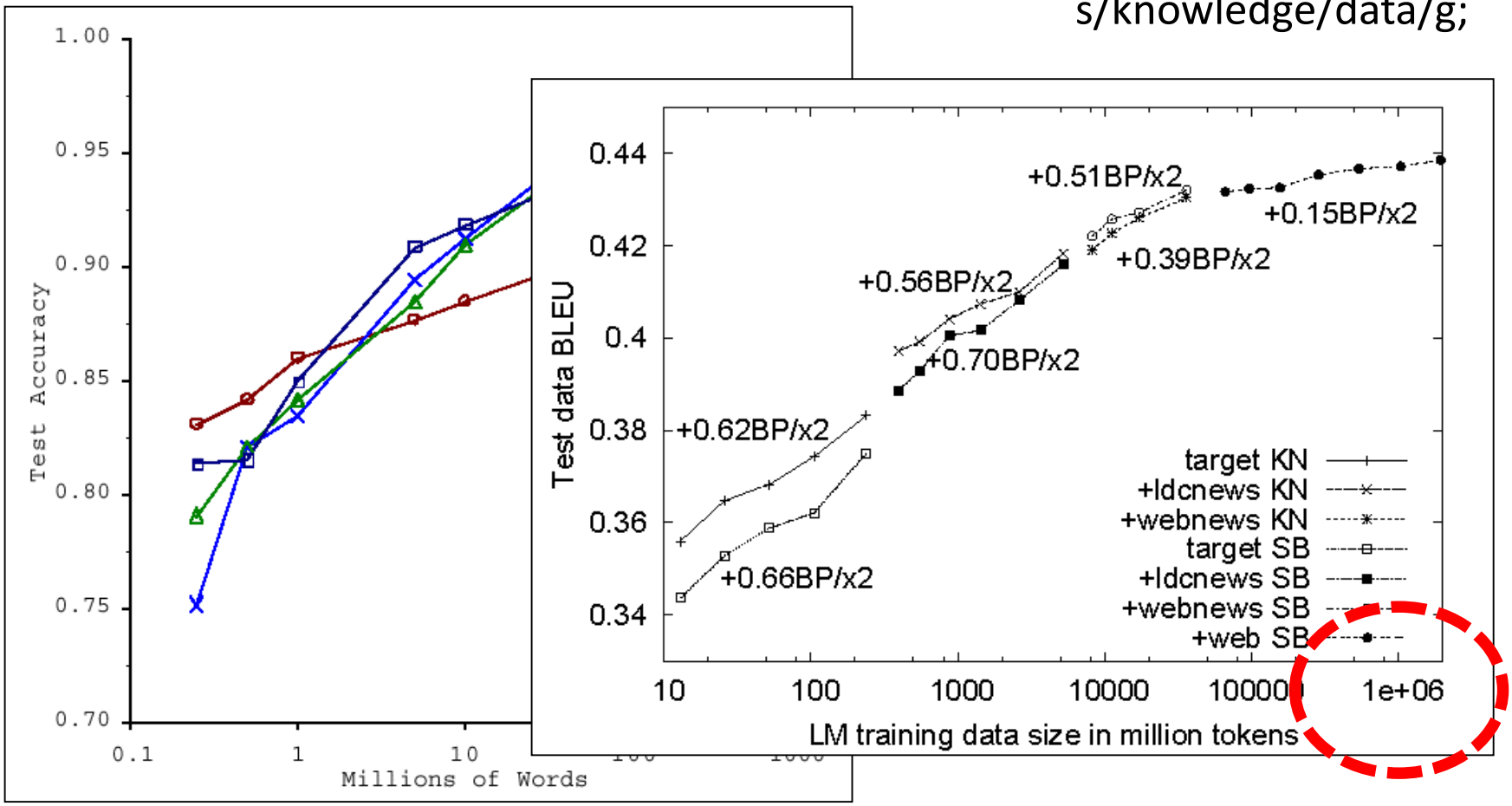
(Agichtein and Gravano, DL 2000; Ravichandran and Hovy, ACL 2002; …)

IBM’s Watson


$18,200
$21,440
$5,600
WATSON
BRAD
contempt
contemn
Despised Icon
97%
14%
10%

No Data Like More Data!

s/knowledge/data/g;



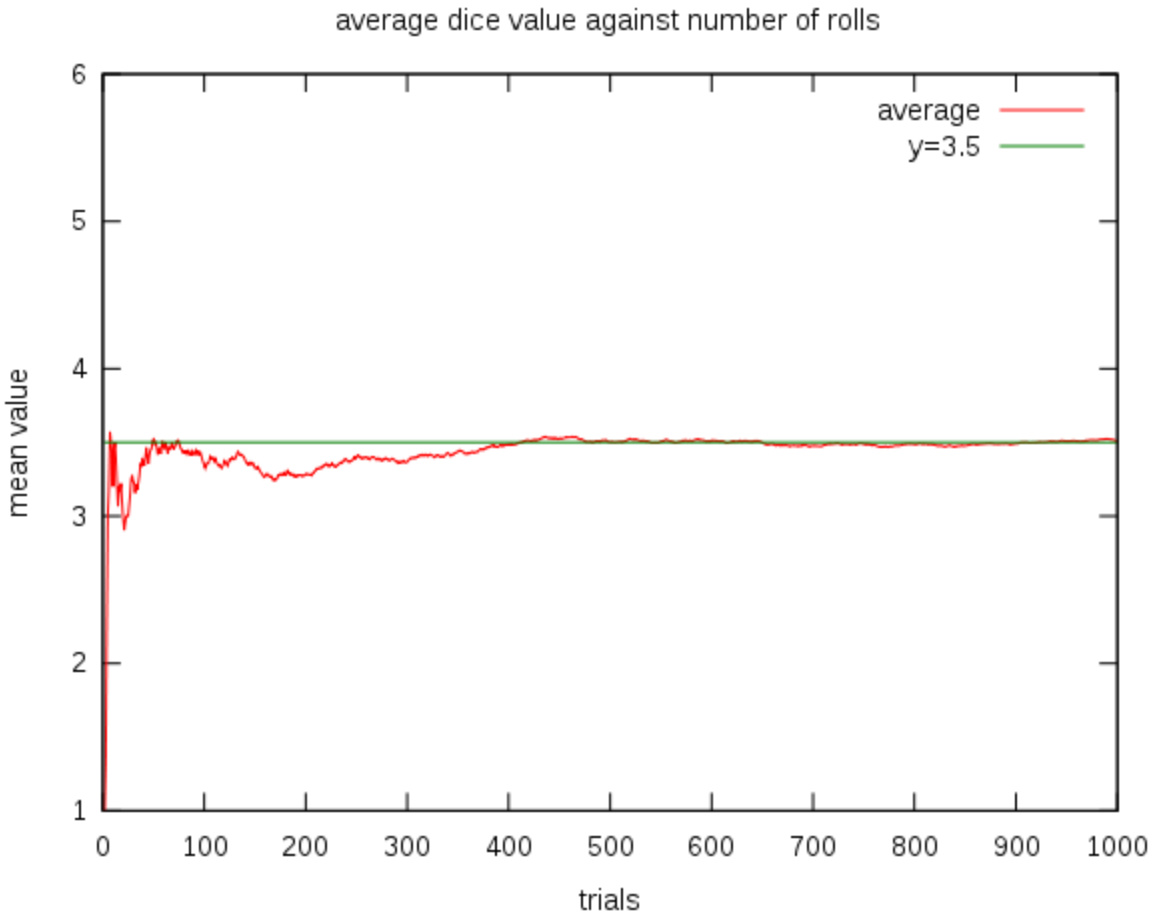
(Banko and Brill, ACL 2001)

(Brants et al., EMNLP 2007)

How do we get here if we're not Google?

The Law of Large Numbers

- In probability theory, the law of large numbers (LLN) is a theorem that describes the result of performing the same experiment a large number of times. According to the law, the average of the results obtained from a large number of trials should be close to the expected value, and will tend to become closer as more trials are performed. [Wikipedia]


average dice value against number of rolls
average
y=3.5
mean value
trials
1
2
3
4
5
6
0
100
200
300
400
500
600
700
800
900
1000

Average household income by programming language

Language	Average Household Income ($)	Data Points
Puppet	87,589.29	112
Haskell	89,973.82	191
PHP	94,031.19	978
CoffeeScript	94,890.80	435
VimL	94,967.11	532
Shell	96,930.54	979
Lua	96,930.69	101
Erlang	97,306.55	168
Clojure	97,500.00	269
Python	97,578.87	2314
JavaScript	97,598.75	3443
Emacs Lisp	97,774.65	355
C#	97,823.31	665
Ruby	98,238.74	3242
C++	99,147.93	845
CSS	99,881.40	527
Perl	100,295.45	990
C	100,766.51	2120
Go	101,158.01	231
Scala	101,460.91	243
ColdFusion	101,536.70	109
Objective-C	101,801.60	562
Groovy	102,650.86	116
Java	103,179.39	1402
XSLT	106,199.19	123
ActionScript	108,119.47	113

Kicker Careers Ranked by Make Percentage

Rank	Kicker	Make %	Number of Kicks
1	Garrett Hartley	87.7	57
2	Matt Stover	86.8	335
3	Robbie Gould	86.2	224
4	Rob Bironas	86.1	223
5	Shayne Graham	85.4	254
...	...	...	
51	Dave Rayner	72.2	90
52	Nick Novak	71.9	64
53	Tim Seder	71.0	62
54	Jose Cortez	70.7	75
55	Wade Richey	66.1	56

Meaningfulness of Answers

- A big data-mining risk is that you may "discover" patterns that are meaningless
  - Bonferroni's principle: (roughly) if you look in more places for interesting patterns than your amount of data will support, you are bound to find crap.

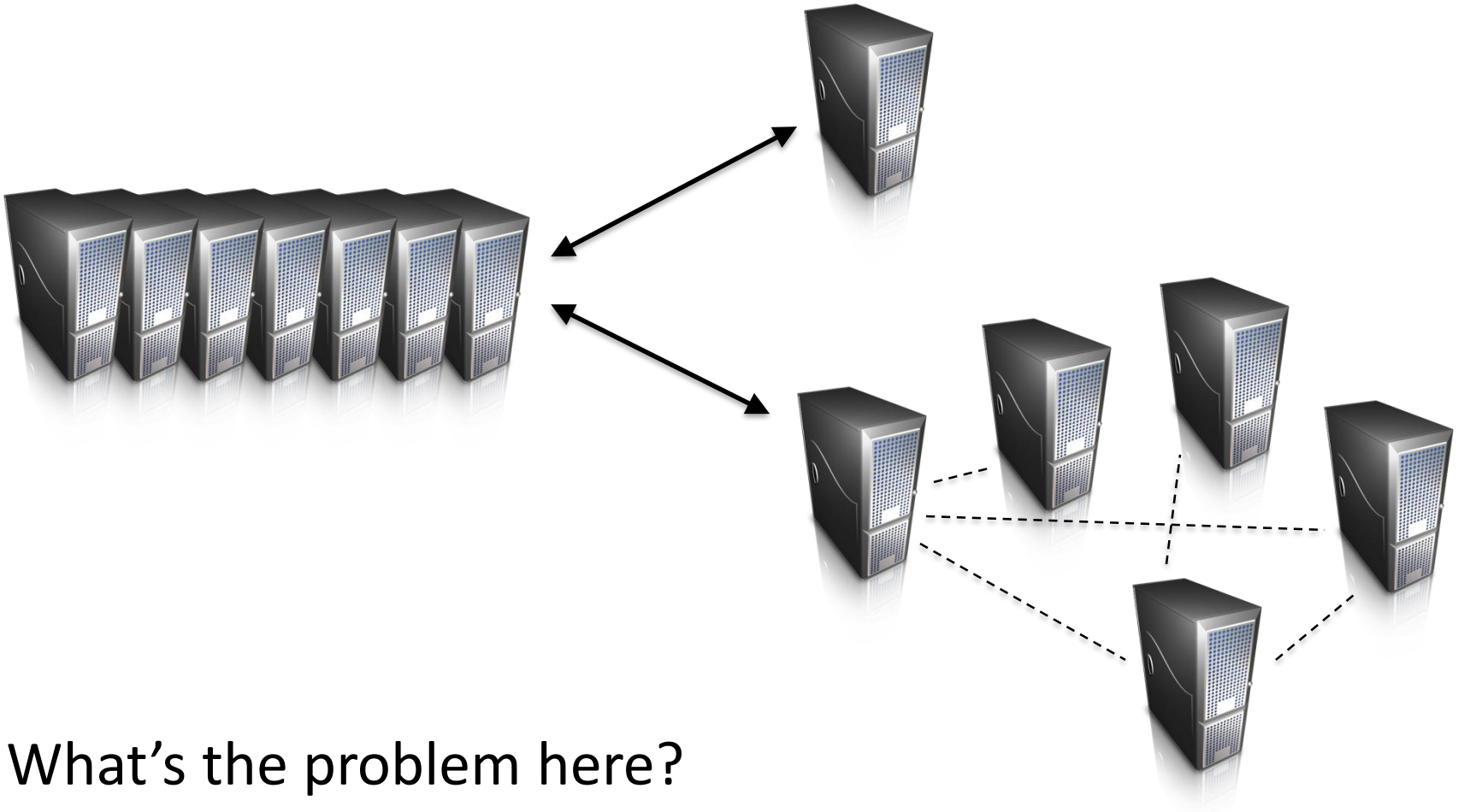
Meaningfulness of Answers

- Example: Rhine Paradox
  - Joseph Rhine was a parapsychologist in the 1950's who hypothesized that some people had Extra-Sensory Perception (ESP)
  - He devised an experiment where subjects were asked to guess 10 hidden cards: red or blue
  - He discovered that almost 1 in 1000 had ESP: they were able to get all 10 right!

Meaningfulness of Answers

- Example: Rhine Paradox
  - He told these people they had ESP and called them in for another test of the same type
  - Alas, he discovered that almost all of them had lost their ESP
  - What did he conclude?

How do we get data for computation?



What's the problem here?

How do we get data for computation?

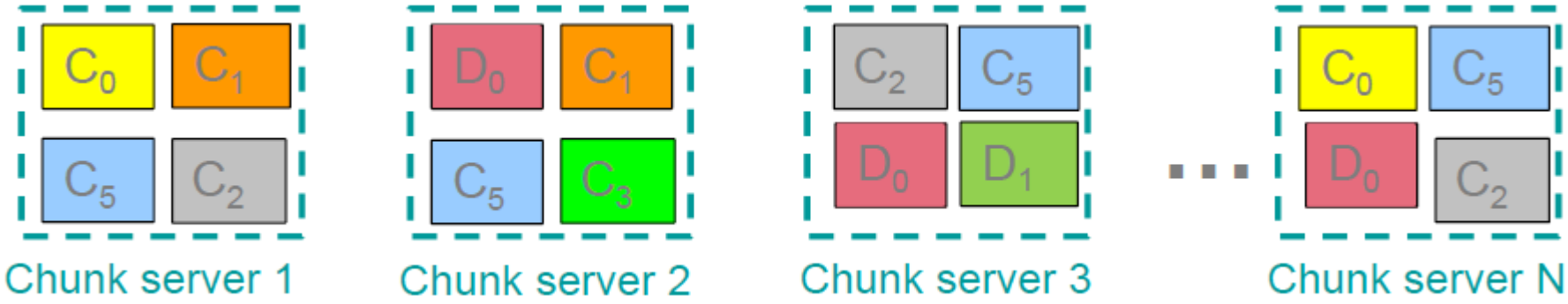
- Don’t move data to computation…, move computation to the data!
  - Store data on the local disks of nodes in the cluster
  - Start up the programs on the node that has the data local
- Why?
  - Not enough RAM to hold all the data in memory
  - Disk access is slow, but disk throughput is reasonable

Distributed File Systems

- A Distributed File System (that provides global file namespace) is the answer
  - GFS (Google File System) for Google's MapReduce
  - HDFS (Hadoop Distributed File System) for Hadoop
    - HDFS = GFS clone (same basic ideas)
- Typical usage pattern
  - Huge files (100s of GB to TB)
  - Data is rarely updated in place
  - Reads and appends are common

Distributed File Systems

- The problem of **reliability**: if nodes fail, how to store data persistently?
  - Data kept in "chunks" spread across machines
  - Each chunk replicated on different machines
  - Seamless recovery from disk or machine failure



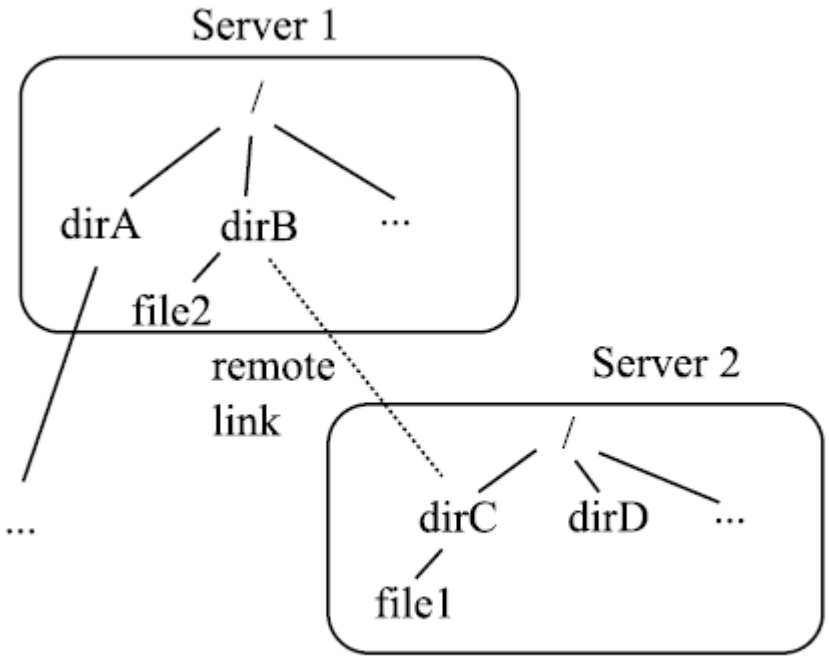
GFS: History

- Google File System, a paper published in 2003 by Google Labs at OSDI
  - Explains the design and architecture of a distributed system apt for serving very large data files (internally used by Google for storing documents collected from the Web).
  - Open Source versions have been developed at once: Hadoop File System (HDFS), Kosmos File System (KFS).

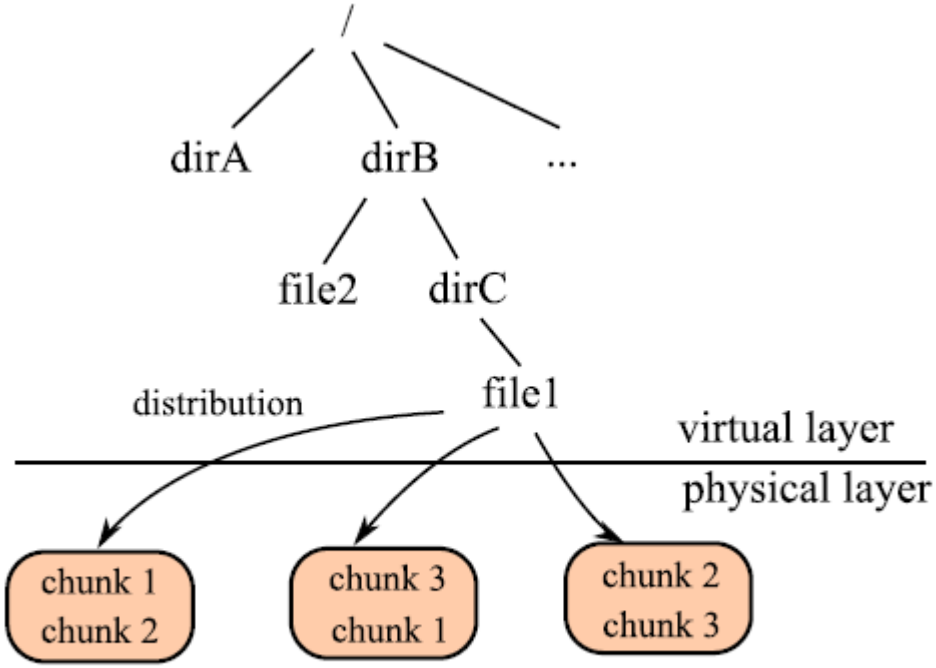
GFS: Motivation

- Why do we need a distributed file system in the first place?
  - Traditional Network File System (NFS) does not meet scalability requirements
    - What if *file1* gets really big?
  - A Large-Scale Distributed File System requires
    - A virtual file namespace
    - Partitioning of files in “chunks”.

GFS: Motivation



A traditional network file system



A large scale distributed file system

GFS: Assumptions

- Commodity hardware over “exotic” hardware
  - Scale “out”, not “up”
- High component failure rates
  - Inexpensive commodity components fail all the time
- “Modest” number of huge files
  - Multi-gigabyte files are common, if not encouraged
- Files are write-once, mostly appended to
  - Perhaps concurrently
- Large streaming reads over random access
  - High sustained throughput over low latency

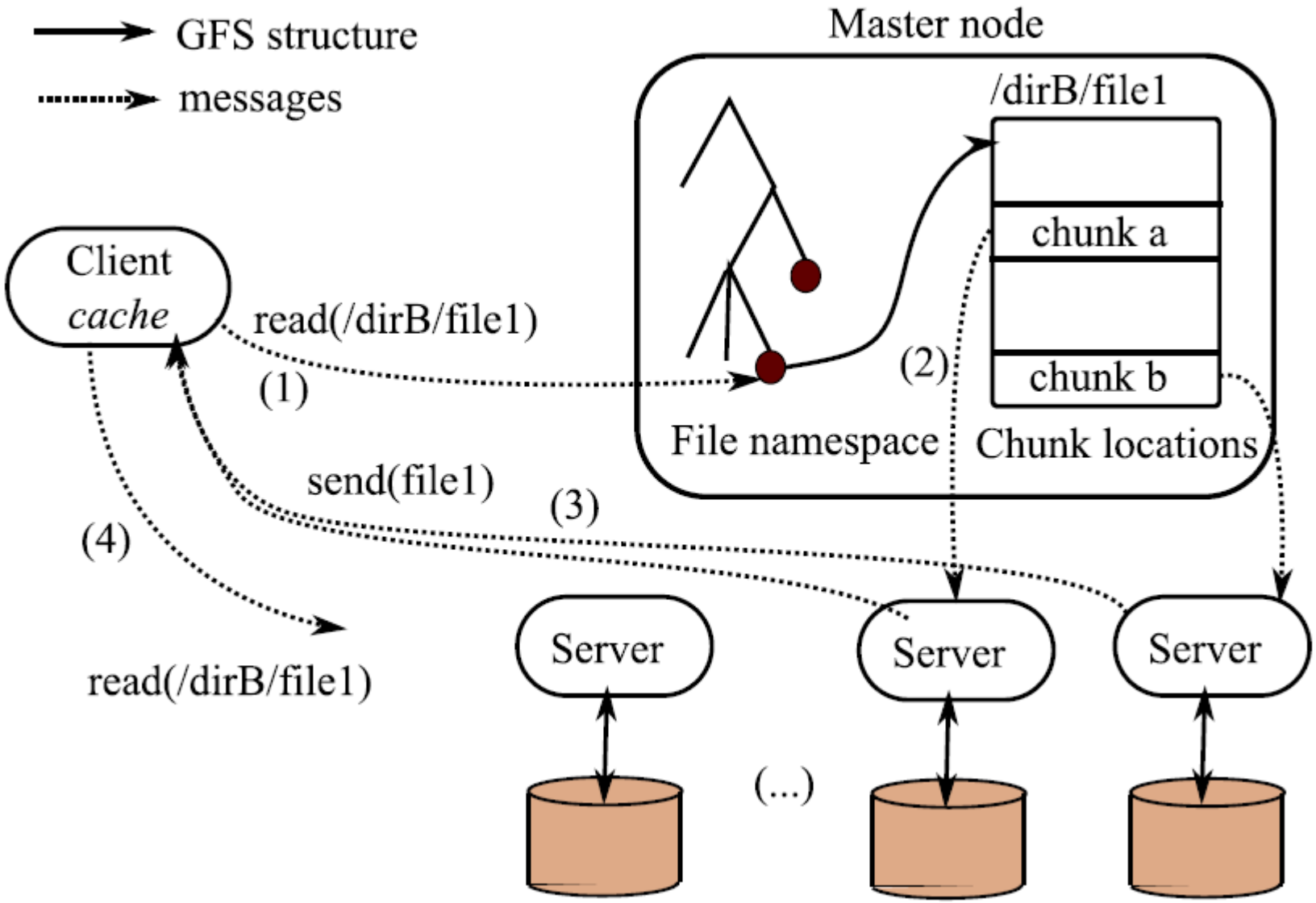
GFS: Design Decisions

- Chunk servers
  - File is split into equal-size contiguous chunks: typically 64MB
  - Each chunk is replicated (default 3x)
  - Try to keep replicas in different racks
- Master node
  - Stores metadata and coordinates access
  - Simple centralized management
  - Might be replicated

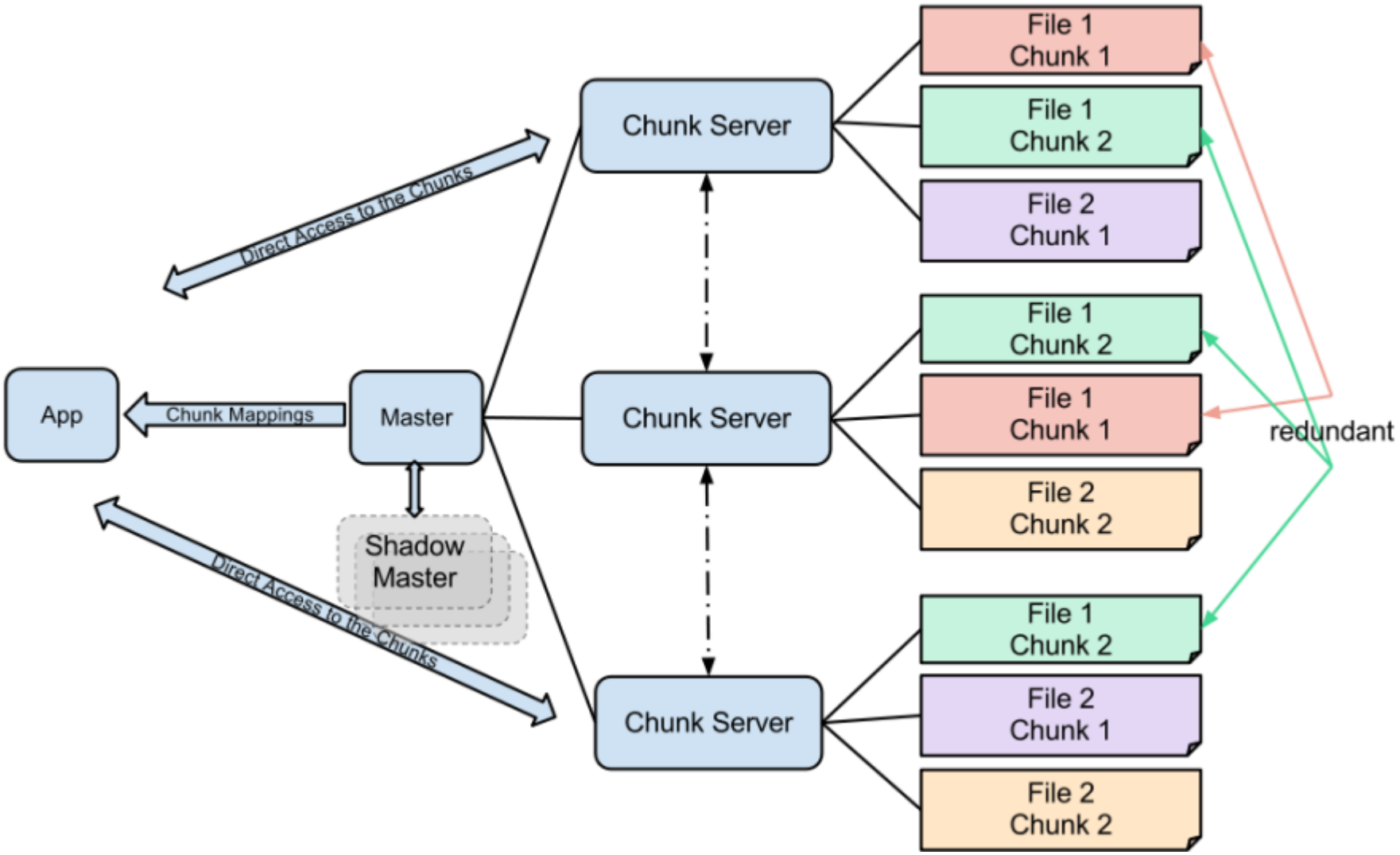
GFS: Design Decisions

- Client library for file access
  - Talks to master node to find chunk servers
  - Connects directly to chunk servers to access data
  - Maintains a cache with the chunks' locations (but not the chunks themselves)
    - No data caching: little benefit due to large datasets, streaming reads.
  - Simplify the API: some of the issues are pushed onto the client (e.g., data layout)

GFS: Architecture



GFS Architecture



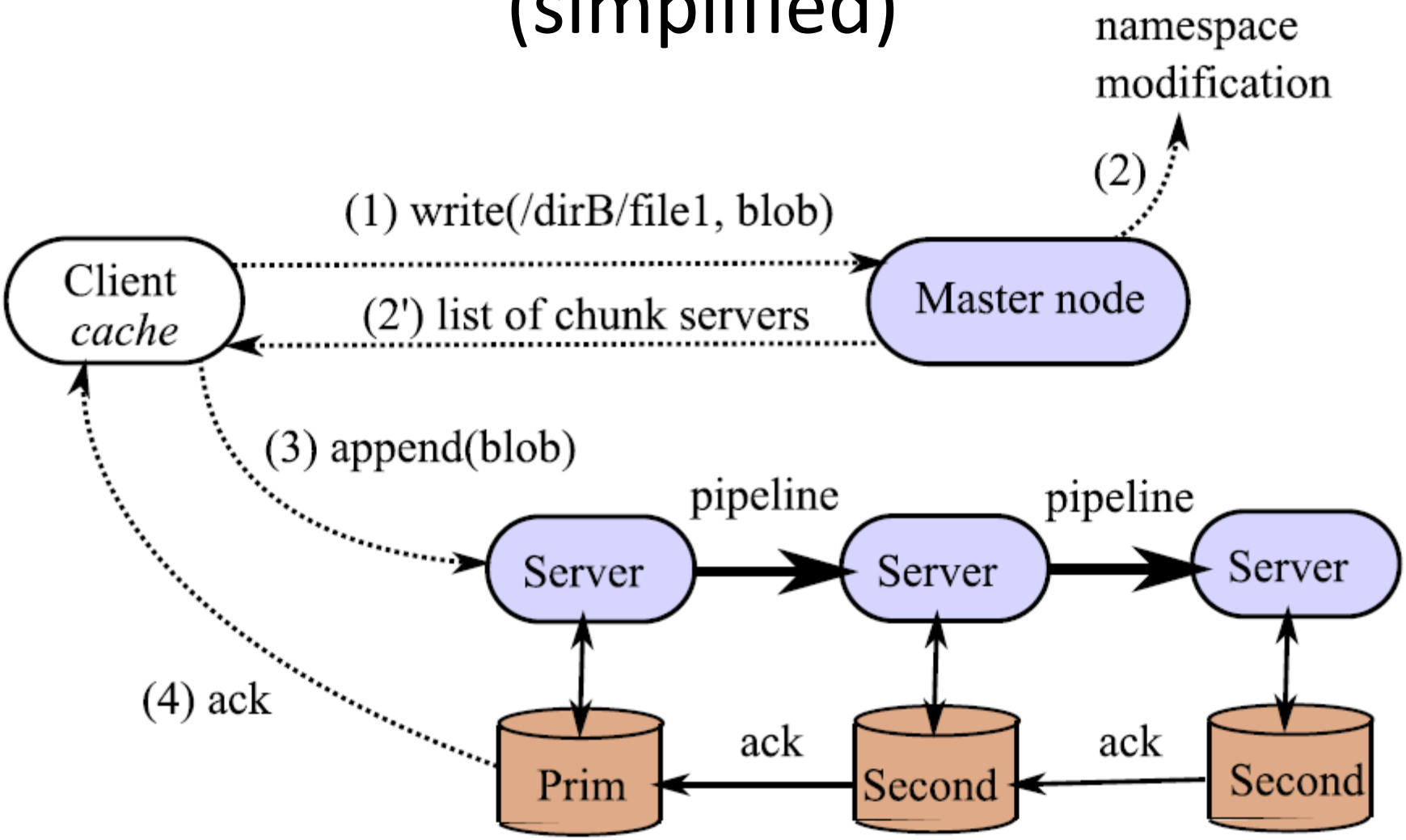
GFS: Technical Details

- The architecture works best for very large files (e.g., several GBs), divided in large (e.g., 64 MBs) chunks
  - This limits the metadata information served by the Master
- Each server implements recovery & replication techniques
  - Default: 3 replicas

GFS: Technical Details

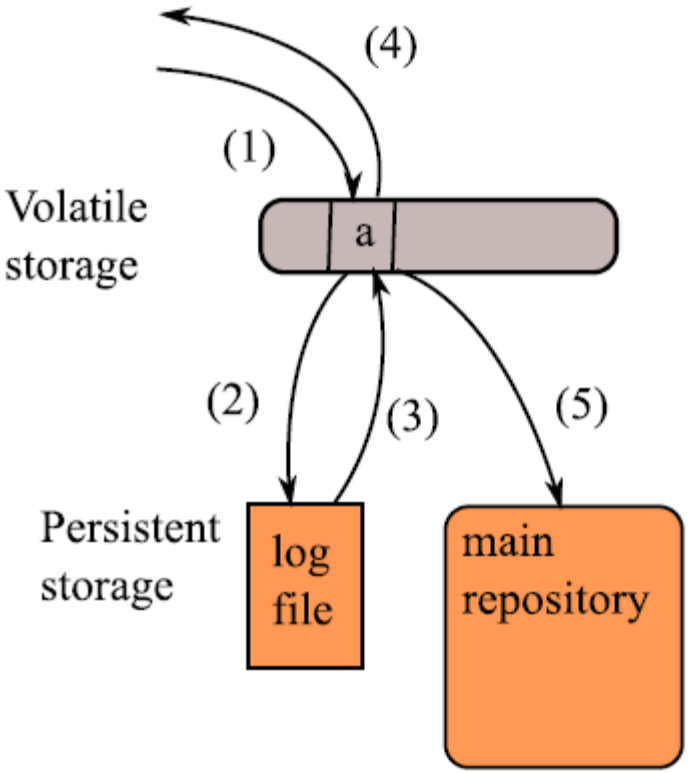
- Availability
  - The Master sends heartbeat messages to servers, and initiates a replacement when a failure occurs.
- Scalability
  - The Master is a potential single point of failure; its protection relies on distributed recovery techniques for all changes that affect the file namespace.

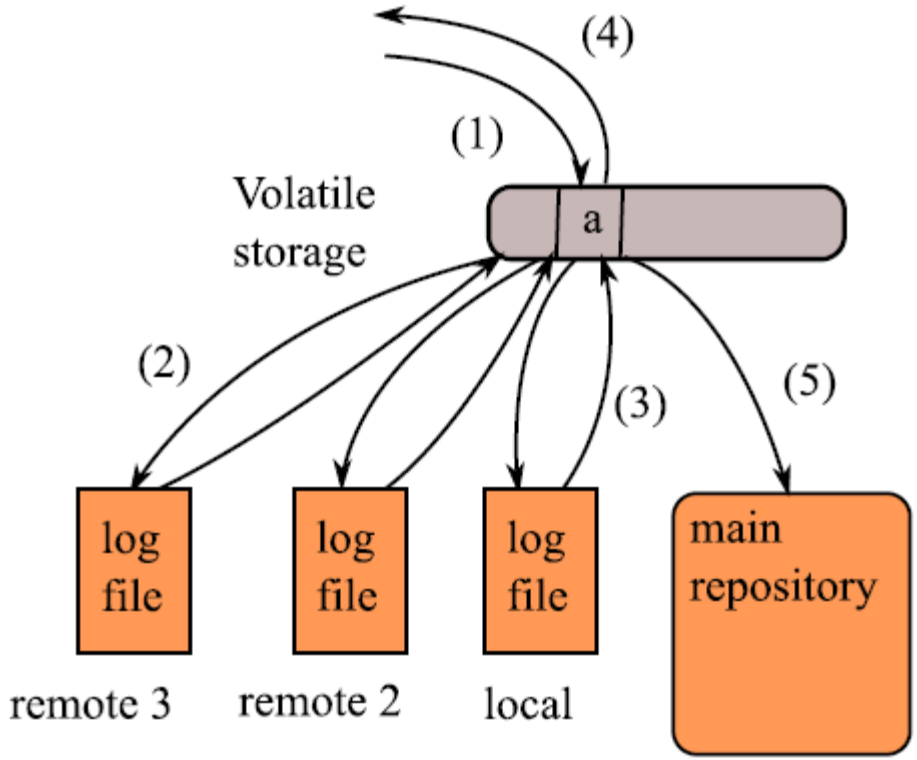
Workflow of a *write()* operation (simplified)



Write (append) in GFS (simplified to non-concurrent operations)

Namespace Updates: Distributed Recovery Protocol



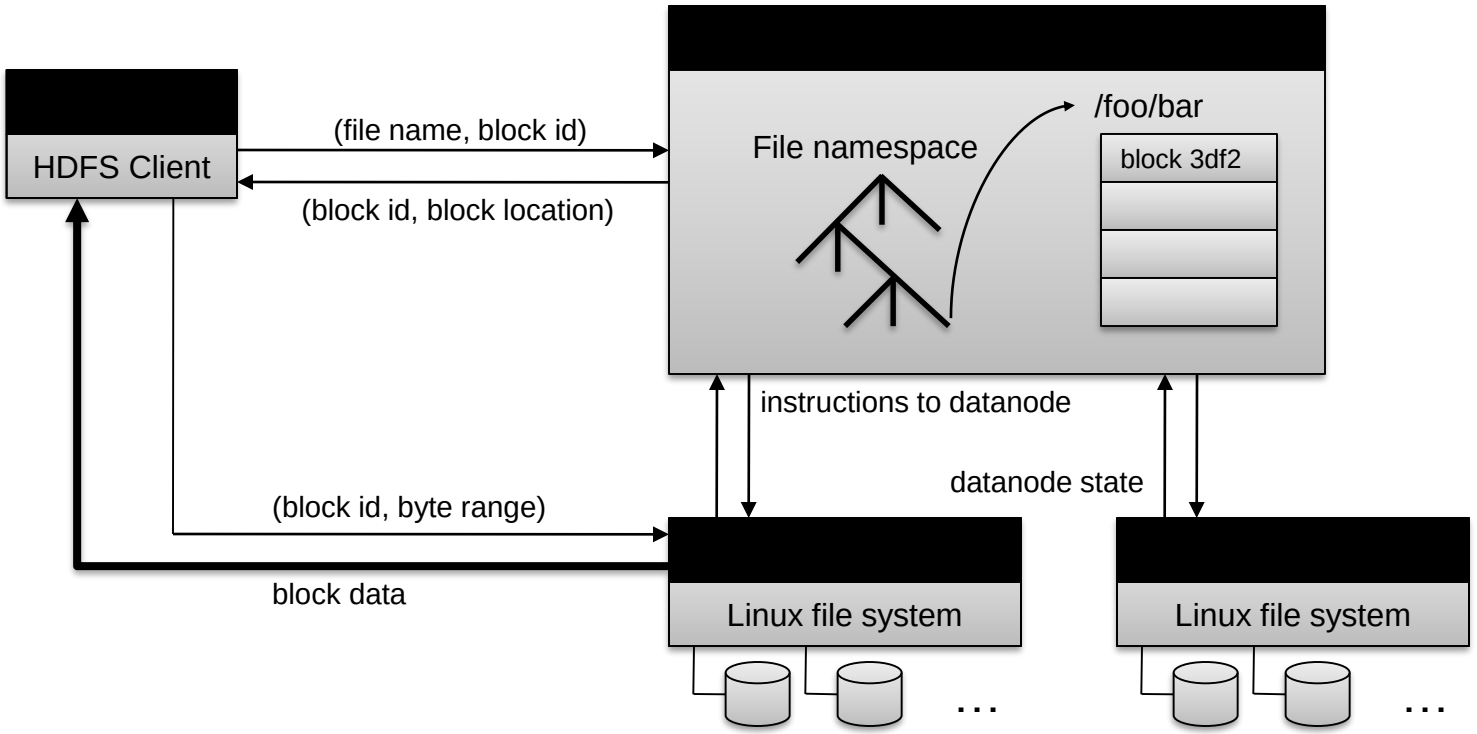


From GFS to HDFS

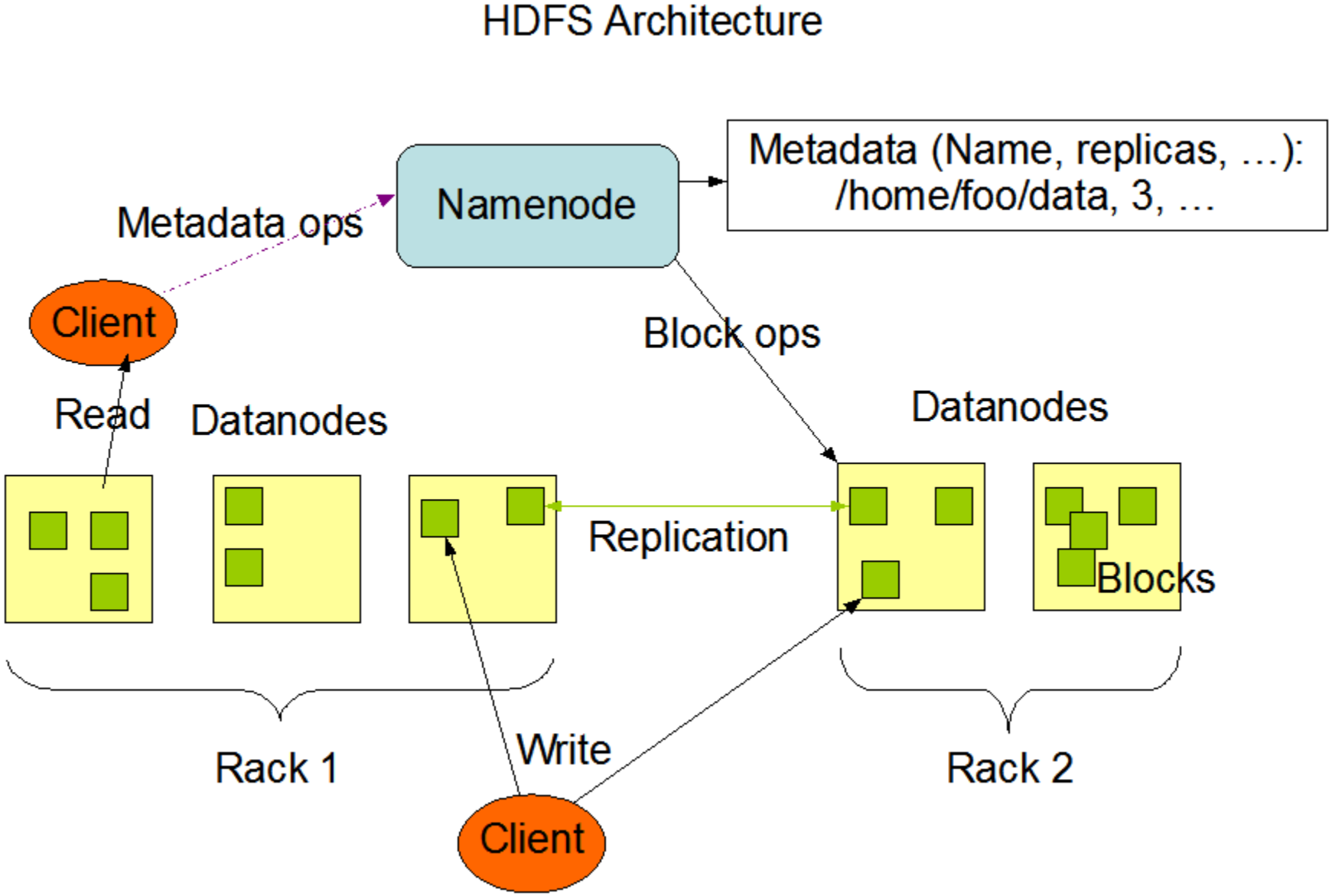
- Terminology differences:
  - GFS master = Hadoop namenode
  - GFS chunkservers = Hadoop datanodes
- Functional differences:
  - No file appends in HDFS (planned feature)
  - HDFS performance is (likely) slower

For the most part, we'll use the Hadoop terminology…

HDFS: Architecture



Adapted from (Ghemawat et al., SOSP 2003)


HDFS Architecture
Namenode
Metadata (Name, replicas, …):
/home/foo/data, 3, …
Metadata ops
Client
Block ops
Read
Datanodes
Datanodes
Replication
Blocks
Rack 1
Write
Rack 2
Client

HDFS: NameNode Metadata

- Meta-data in Memory
  - The entire metadata is in main memory
  - No demand paging of meta-data
- Types of Metadata
  - List of files
  - List of blocks for each file
  - List of DataNodes for each block
  - File attributes, e.g creation time, replication factor
- A Transaction Log
  - Records file creations, file deletions. etc

HDFS: Namenode Responsibilities

- Managing the file system namespace:
  - Holds file/directory structure, metadata, file-to-block mapping, access permissions, etc.
- Coordinating file operations:
  - Directs clients to datanodes for reads and writes
  - No data is moved through the namenode
- Maintaining overall health:
  - Periodic communication with the datanodes
  - Block re-replication and rebalancing
  - Garbage collection

HDFS: DataNode

- Block Server
  - Stores data in the local file system (e.g. ext3)
  - Stores meta-data of a block (e.g. CRC)
  - Serves data and meta-data to Clients
- Block Report
  - Periodically sends a report of all existing blocks to the NameNode
- Facilitates Pipelining of Data
  - Forwards data to other specified DataNodes

HDFS: Block Placement

- Current Strategy
  - One replica on local node
  - Second replica on a remote rack
  - Third replica on the same remote rack
  - Additional replicas are randomly placed
- Clients read from nearest replica
- Would like to make this policy pluggable

HDFS: Data Correctness

- Use checksums to validate data
  - CRC32
- File creation
  - Client computes checksum per 512 byte
  - DataNode stores the checksum
- File access
  - Client retrieves the data and checksum from DataNode
  - If validation fails, Client tries other replicas

HDFS: NameNode Failure

- A single point of failure
- Transaction Log stored in multiple directories
  - A directory on the local file system
  - A directory on a remote file system (NFS/CIFS)
- Need to develop a real high availability solution

Distributed Access Structures

- Indexing
  - Hashing based techniques
    - Consistent Hashing
  - Tree based techniques
    - Distributed B-Tree

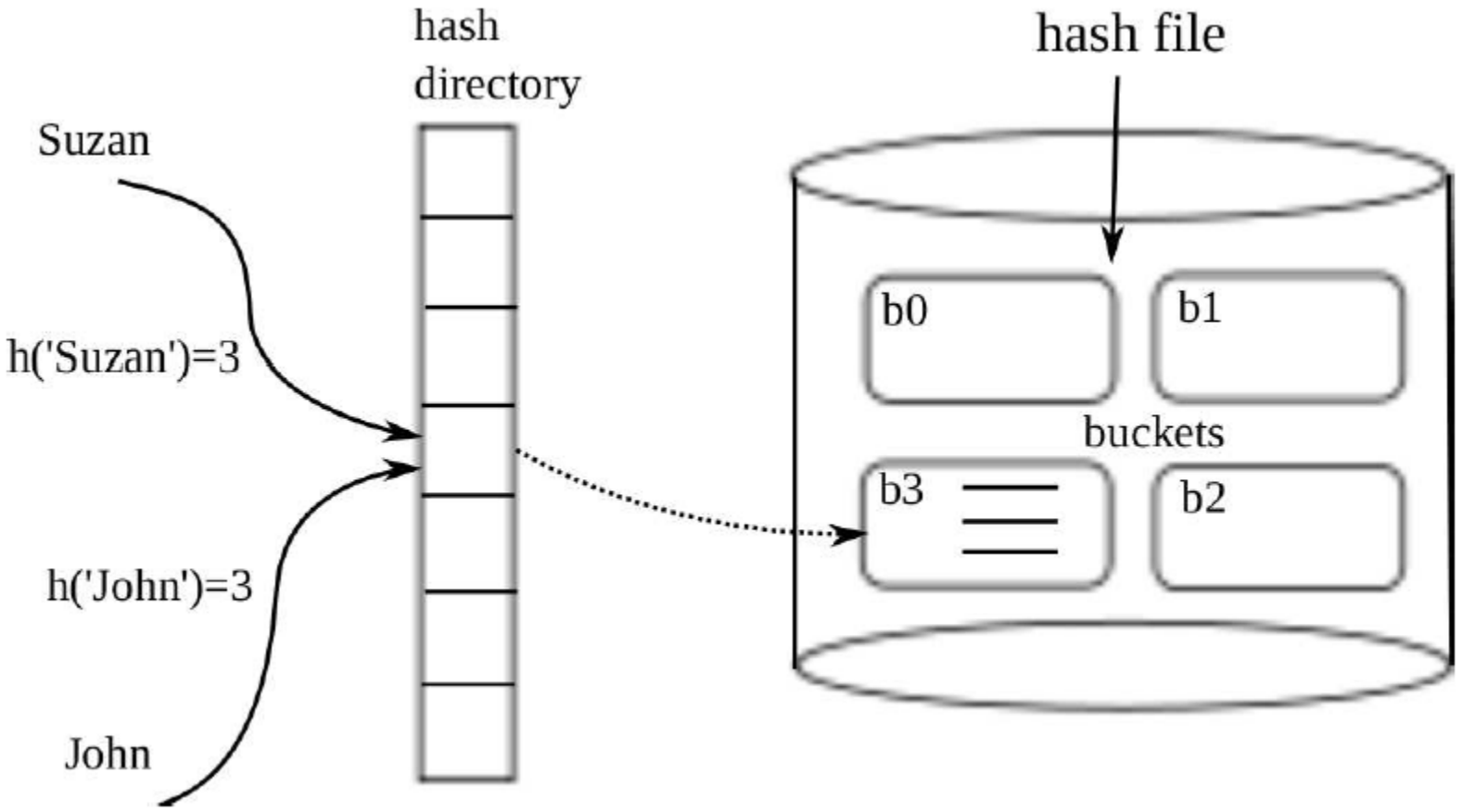
Indexing

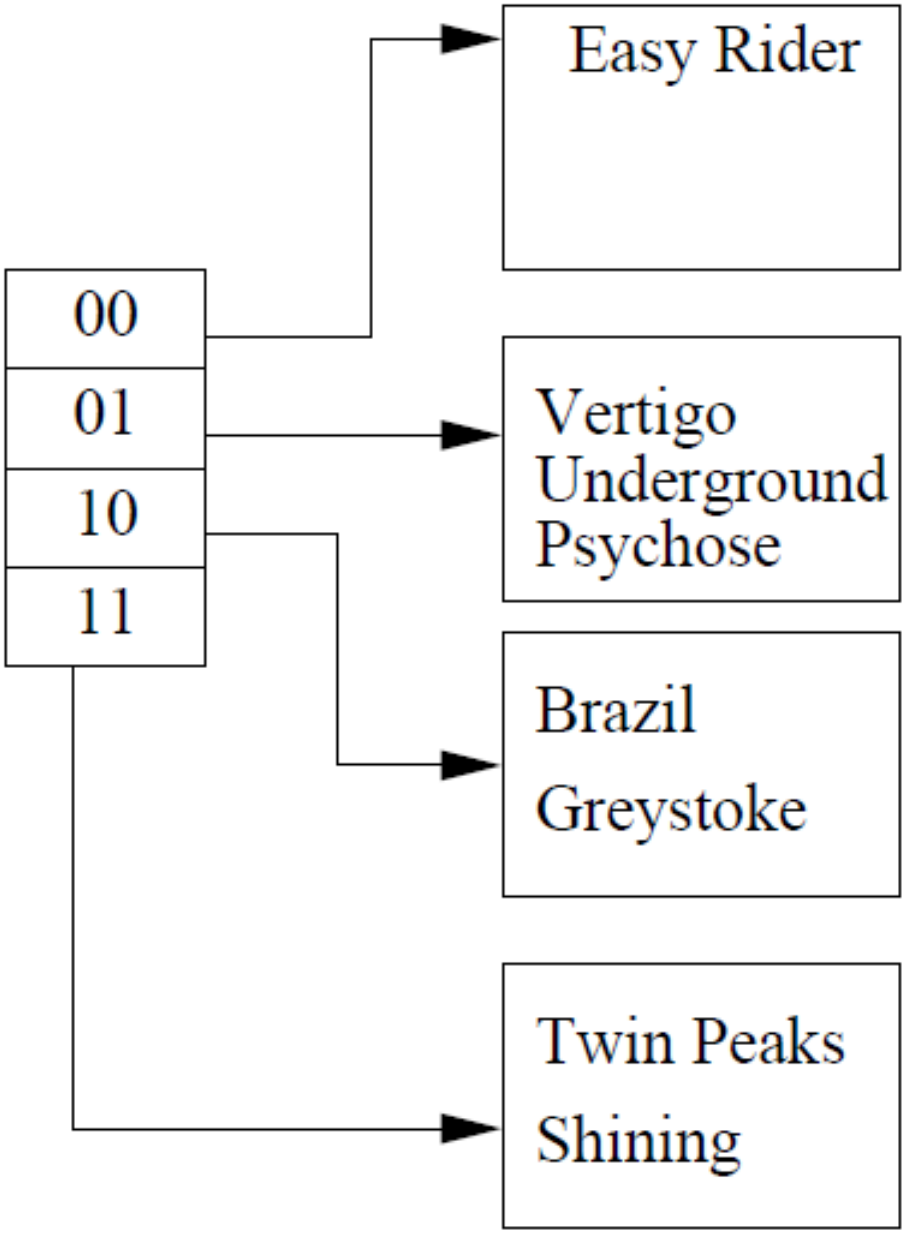
- We assume a (very) large collection C of pairs (k,v), where k is a key and v is the value of an object (seen as row data).
- An index on C is a structure that associates the key k with the (physical) address of v.

Indexing

- An index supports *dictionary operations*:
  - insertion $\text{insert}(k,v)$
  - deletion $\text{delete}(k)$
  - key search $\text{search}(k)$: v
  - range search $\text{range}(k_1, k_2)$: $\{v\}$ [optional]
- In a distributed index, one should also consider:
  - (node) leave and (node) join operations

Hashing (Centralised)




00
01
10
11
Easy Rider
Vertigo
Underground
Psychose
Brazil
Greystoke
Twin Peaks
Shining

Hashing (Distributed)

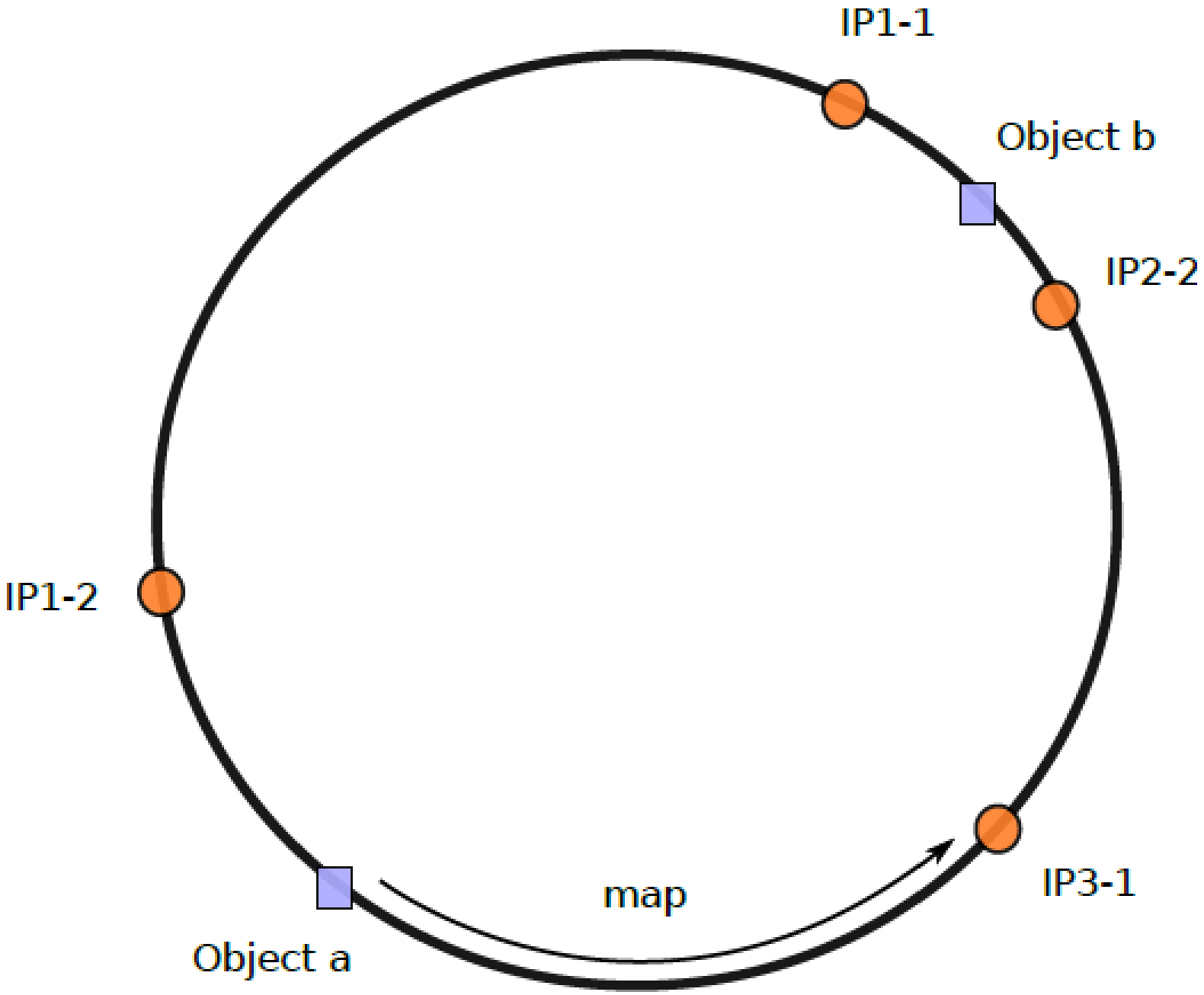
- The straightforward idea
  - Everybody uses the same hash function, and buckets are replaced by *servers*.
- Issues with distribution
  - Dynamicity: at Web scale, we must be able to add or remove servers at any moment.
  - Inconsistencies: it is very hard to ensure that all participants share an accurate view of the system (e.g., the hash function)

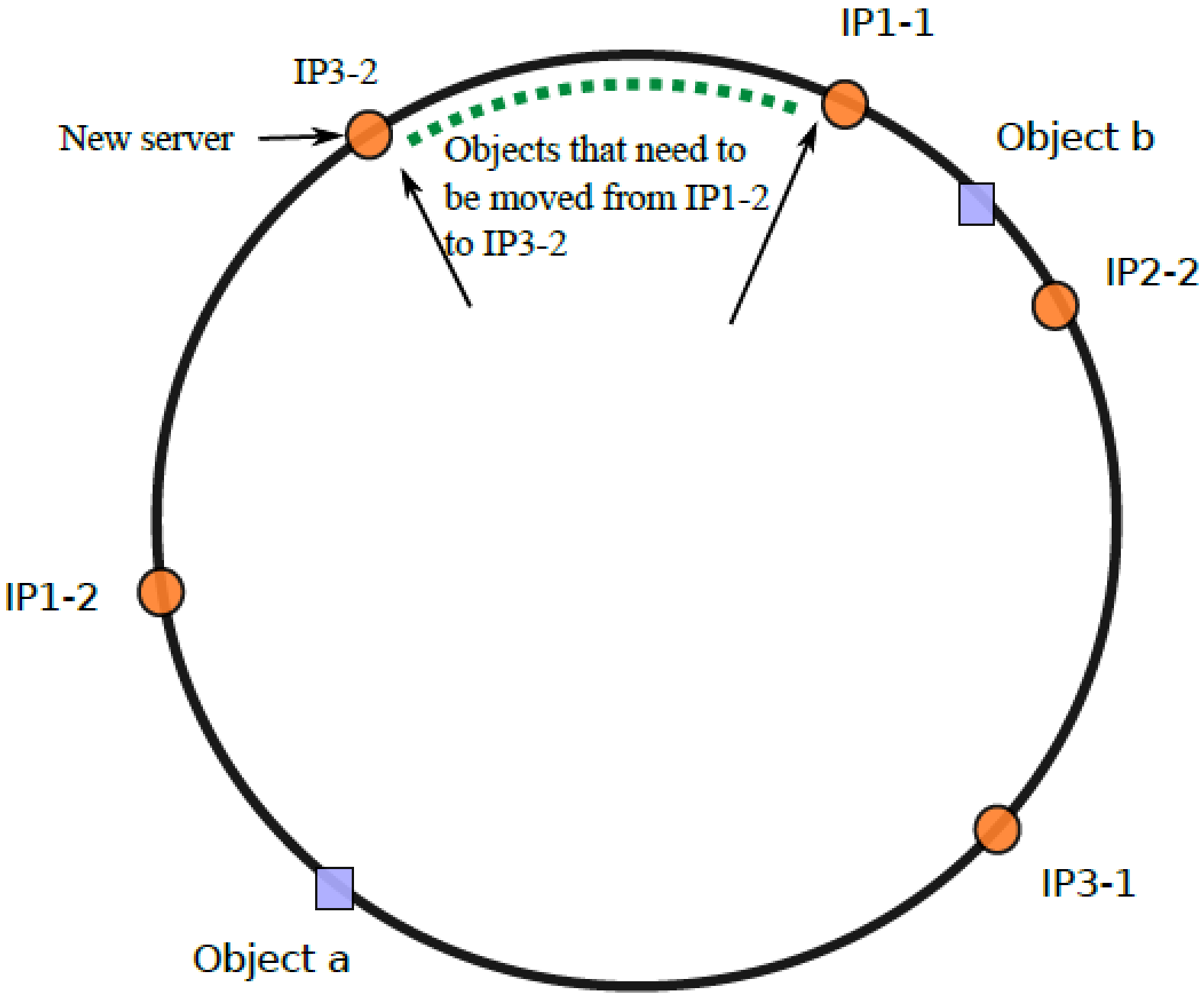
Hashing (Distributed)

- A naïve solution
  - Let N be the number of servers.
  - The function $modulo(h(k), N) = i$ maps a pair (k,v) to server Si .
  - Fact: if N changes, or if a client uses an invalid value for N, the mapping becomes inconsistent.

Consistent Hashing

- 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 and the servers' IPs to a large address space A (e.g., $[0, 2^{64}-1]$) organized as a ring 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.


IP1-1
Object b
IP2-2
IP1-2
IP3-1
map
Object a

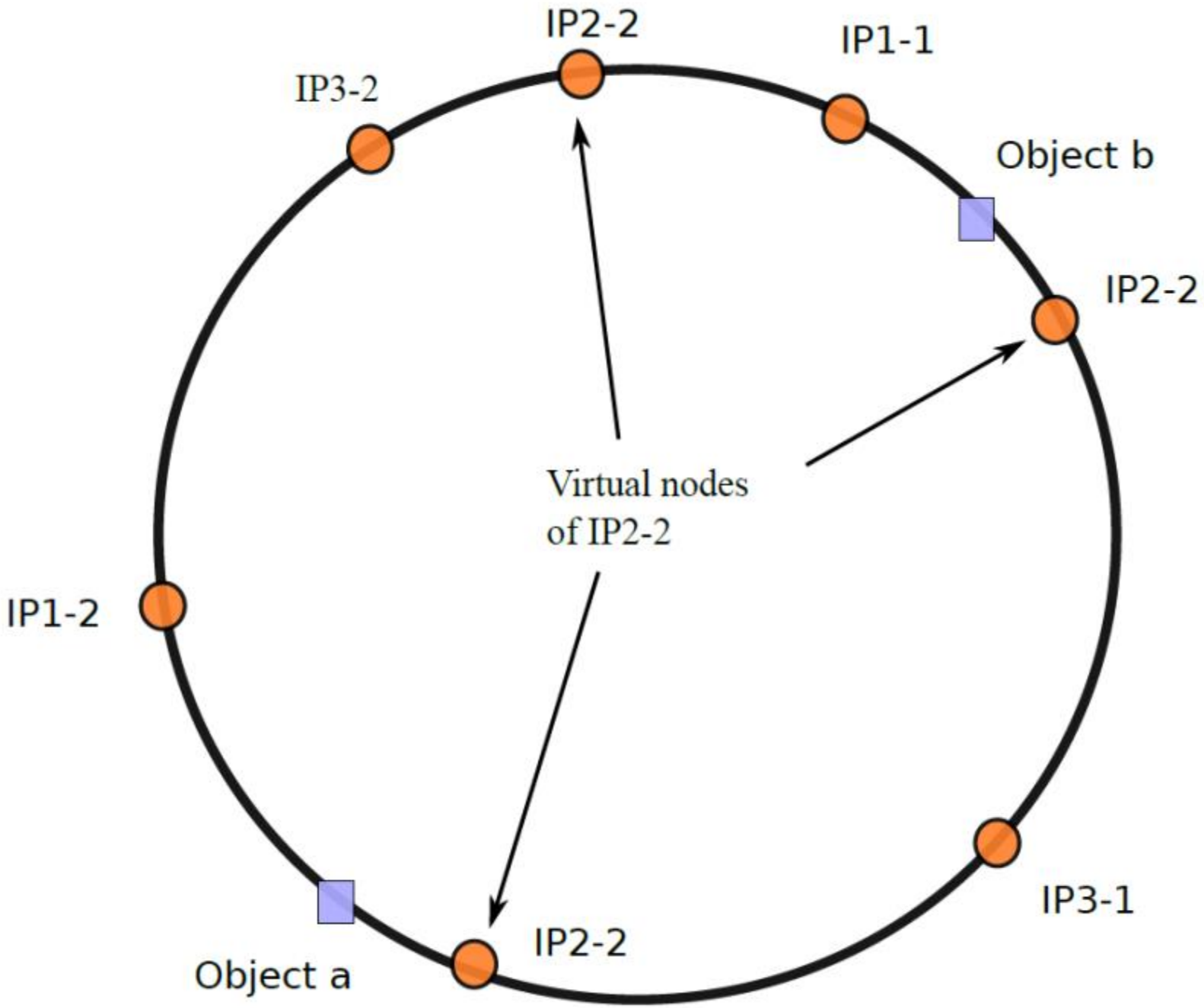

IP1-1
IP3-2
New server
Objects that need to
be moved from IP1-2
to IP3-2
Object b
IP2-2
IP1-2
IP3-1
Object a

Some (Really Useful) Refinements

- What if a server fails?
  - Use replication:
    - put a copy on the next machine (on the ring),
    - then on the next after the next,
    - and so on.

Some (Really Useful) Refinements

- How can we balance the load?
  - 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 relocation is more evenly distributed.
    - also useful in case of heterogeneity (the rule in large-scale systems).


IP2-2
IP1-1
IP3-2
Object b
IP2-2
Virtual nodes
of IP2-2
IP1-2
IP3-1
IP2-2
Object a

Where is the Hash Directory?

- On a specific (“Master") node, acting as a load balancer (e.g., in caching systems)
  - raises scalability issues
- Each node records its successor on the ring
  - may require $\mathrm{O}(N)$ messages for routing queries: not resilient to failures

Where is the Hash Directory?

- Each node records $\log N$ carefully chosen other nodes (e.g., in P2P)
  - ensures $O(\log N)$ messages for routing queries
  - convenient trade-off for highly dynamic networks
- Full duplication of the hash directory at each node (e.g., in Dynamo)
  - ensures 1 message for routing
  - heavy maintenance protocol which can be achieved through gossiping

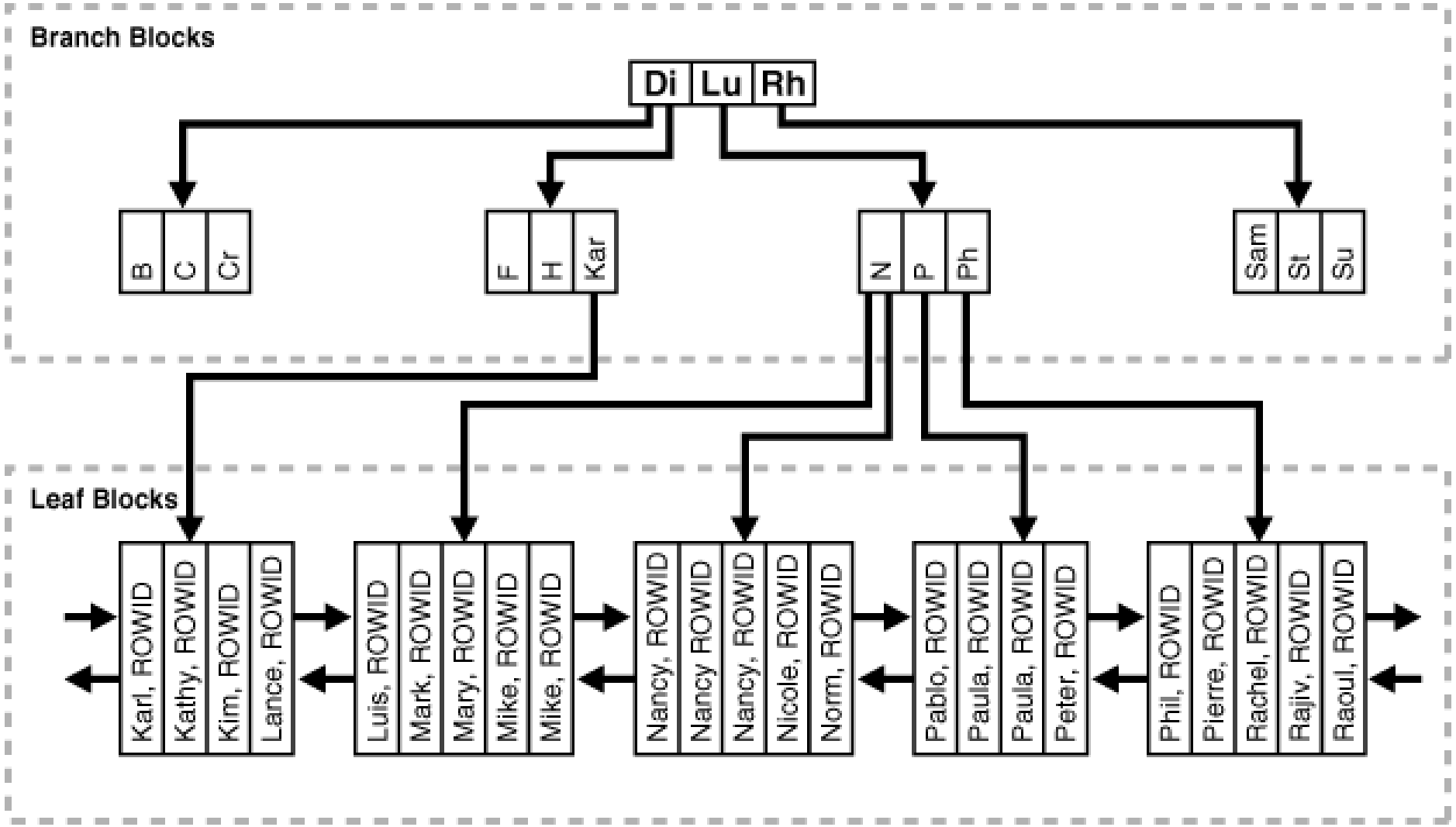
Amazon Dynamo

- A distributed storage system that targets high availability
  - Your shopping cart is stored there!
- An open-source version: Voldemort
  - It is used by LinkedIn

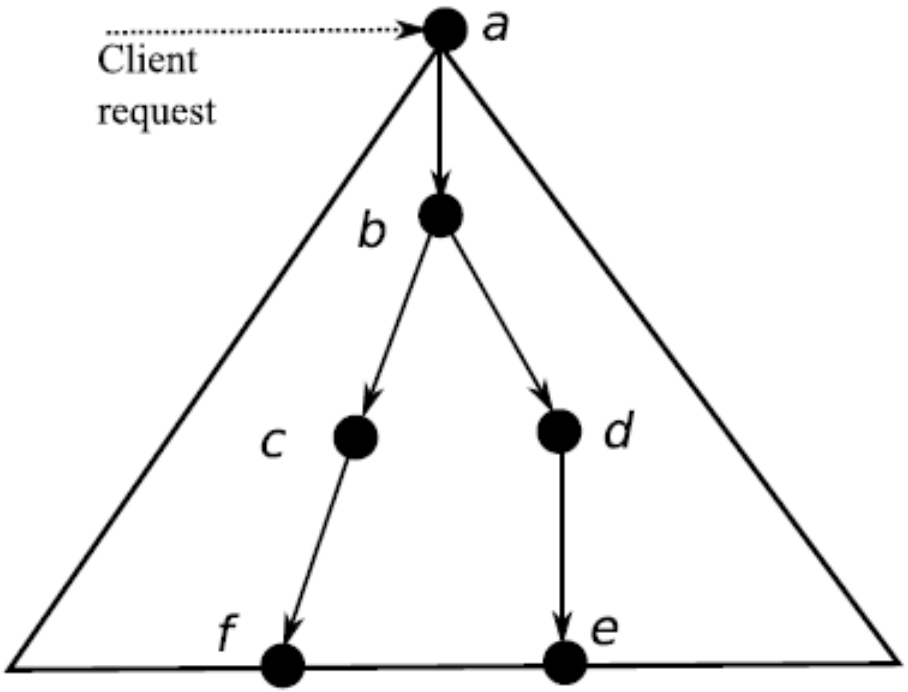
Amazon Dynamo

- Features
  - 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: 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.

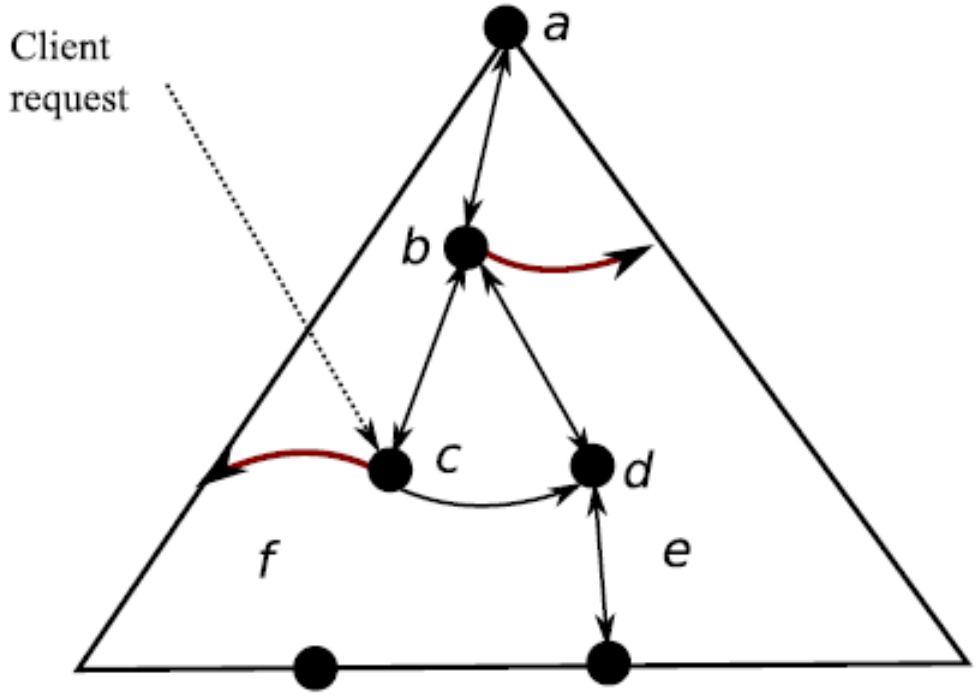
B-Tree (Centralised)



B-Tree (Distributed)



Standard tree



With local routing nodes

B-Tree (Distributed)

- Issues with distribution
  - All operations follow a top-down path: potential factor of non-scalability
- Possible solutions
  - *caching* of the tree structure, or part of it, on the Client node
  - *replication* of the upper levels of the tree
  - *routing tables*, stored at each node, enabling horizontal and vertical navigation in the tree

BigTable

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

BigTable

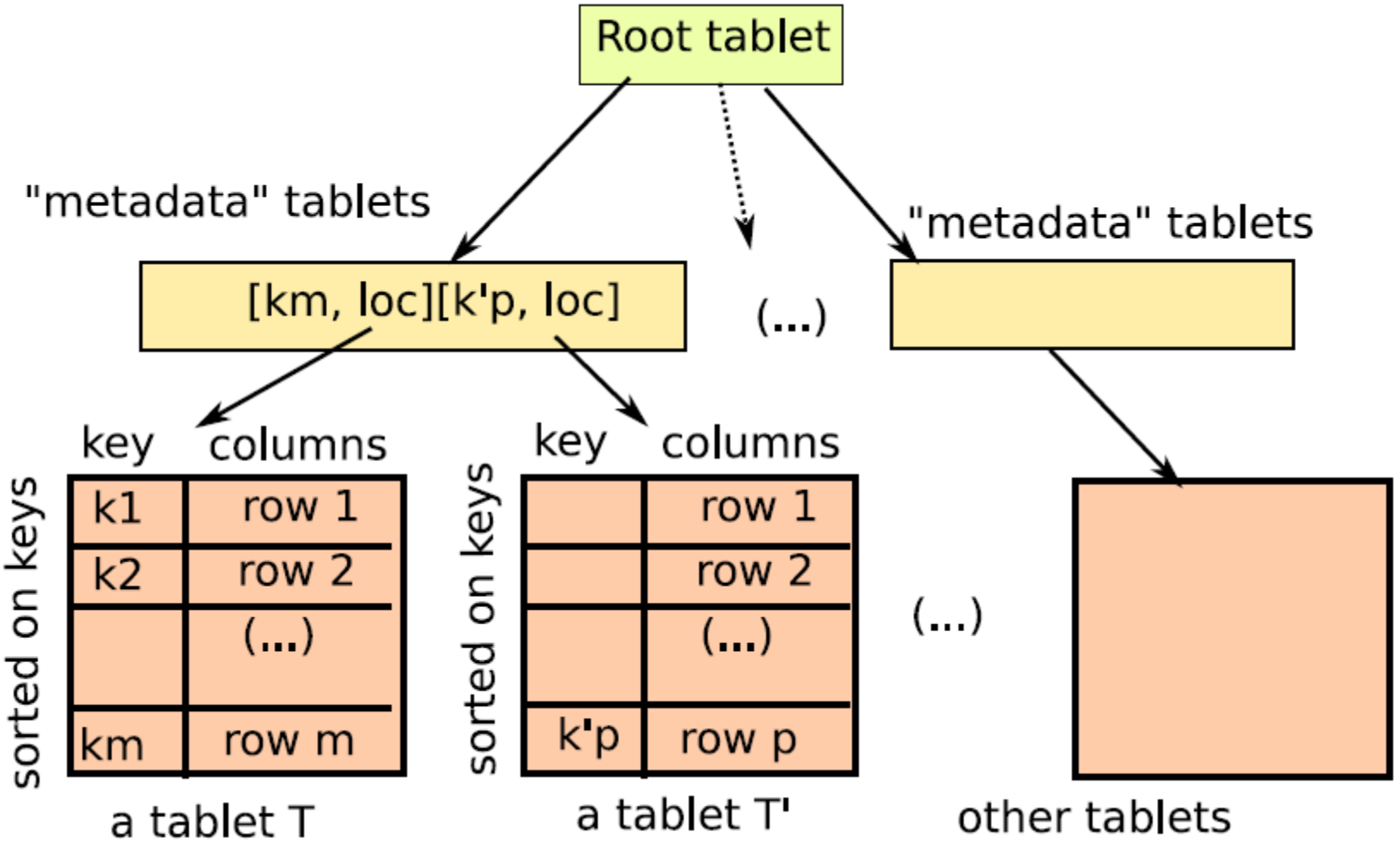
- Context
  - A controlled environment, with homogeneous servers located in a data centre;
  - A stable organization, with long-term storage of large structured data;
  - A data model (column-oriented tables with versioning)

BigTable

- Design
  - Close to a B-tree, with large capacity leaves
  - Scalability is achieved by a cache maintained by Client nodes

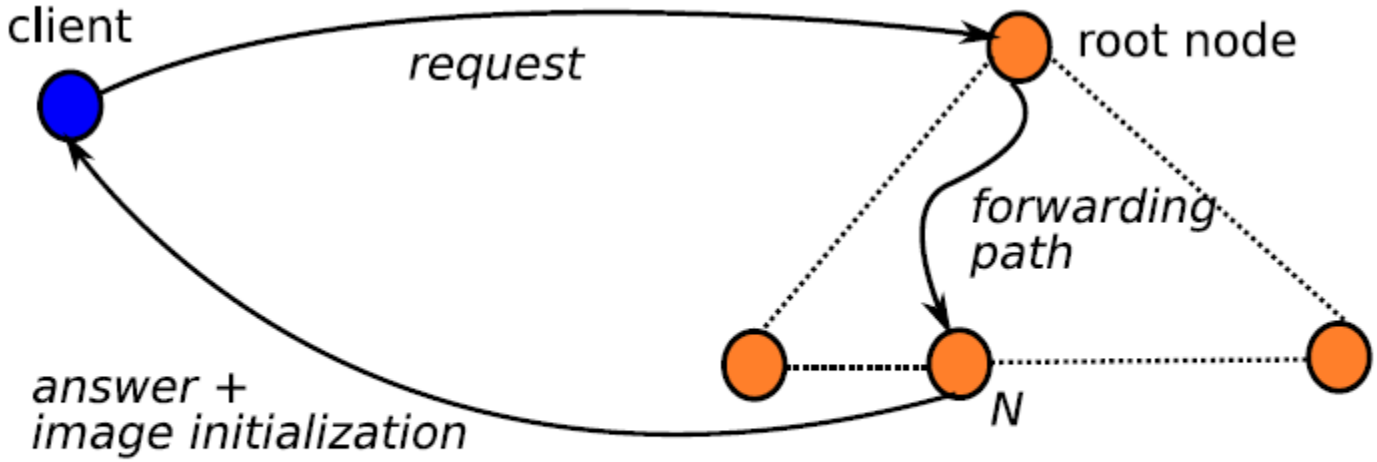
BigTable

- Structure Overview
  - The table is partitioned in “tablets”
    - Tablets are indexed by upper levels
    - Full tablets are split, with upward adjustment
  - Leaf level:
    - A “tablet” organized in “rows” indexed by a key
    - Rows are stored in lexicographic order on the key values

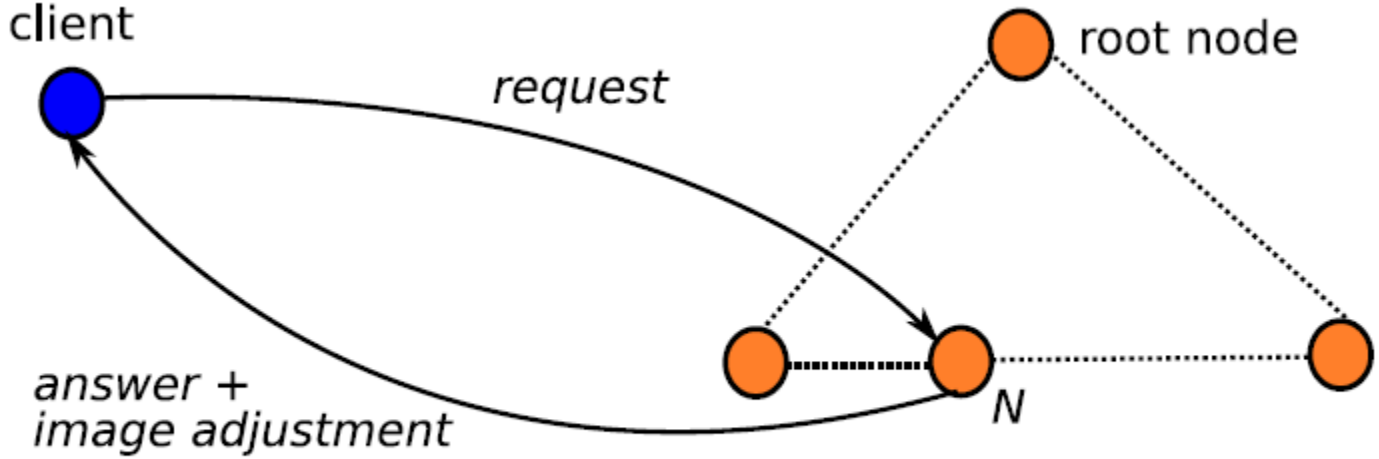

Root tablet
"metadata" tablets
[km, loc][k'p, loc]
(...)
"metadata" tablets
key
columns
sorted on keys
k1
row 1
k2
row 2
(...)
km
row m
a tablet T
key
columns
sorted on keys
row 1
row 2
(...)
k'p
row p
a tablet T'
(...)
other tablets

BigTable

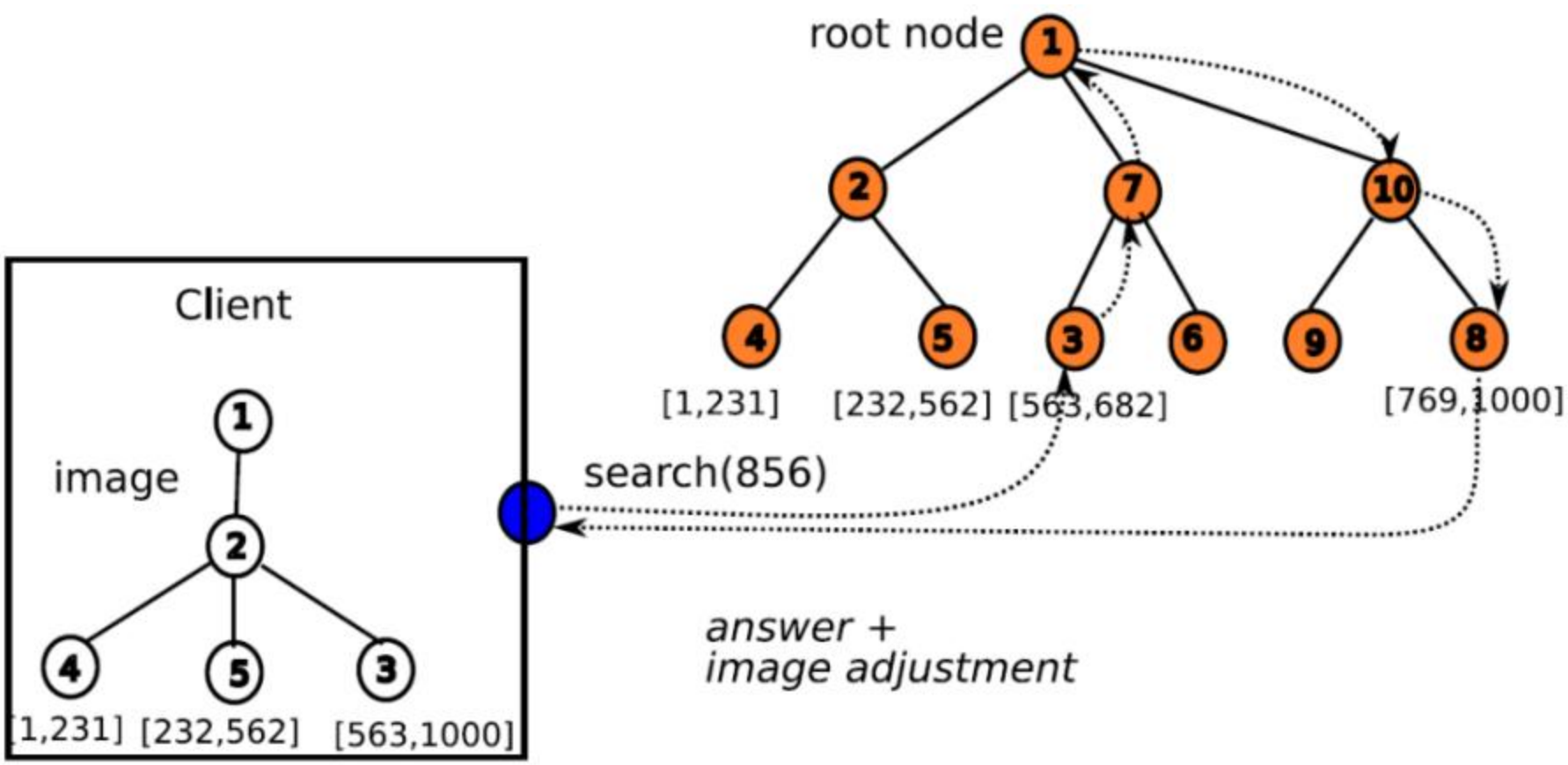
- Architecture: one Master - many Servers
  - The Master maintains the root node and carries out administrative tasks
  - Scalability is obtained with Client cache that stores a (possibly outdated) image of the tree
    - A Client request may fail, due to an out-of-date image of the tree
    - An adjustment requires at most *height(Tree)* rounds of message

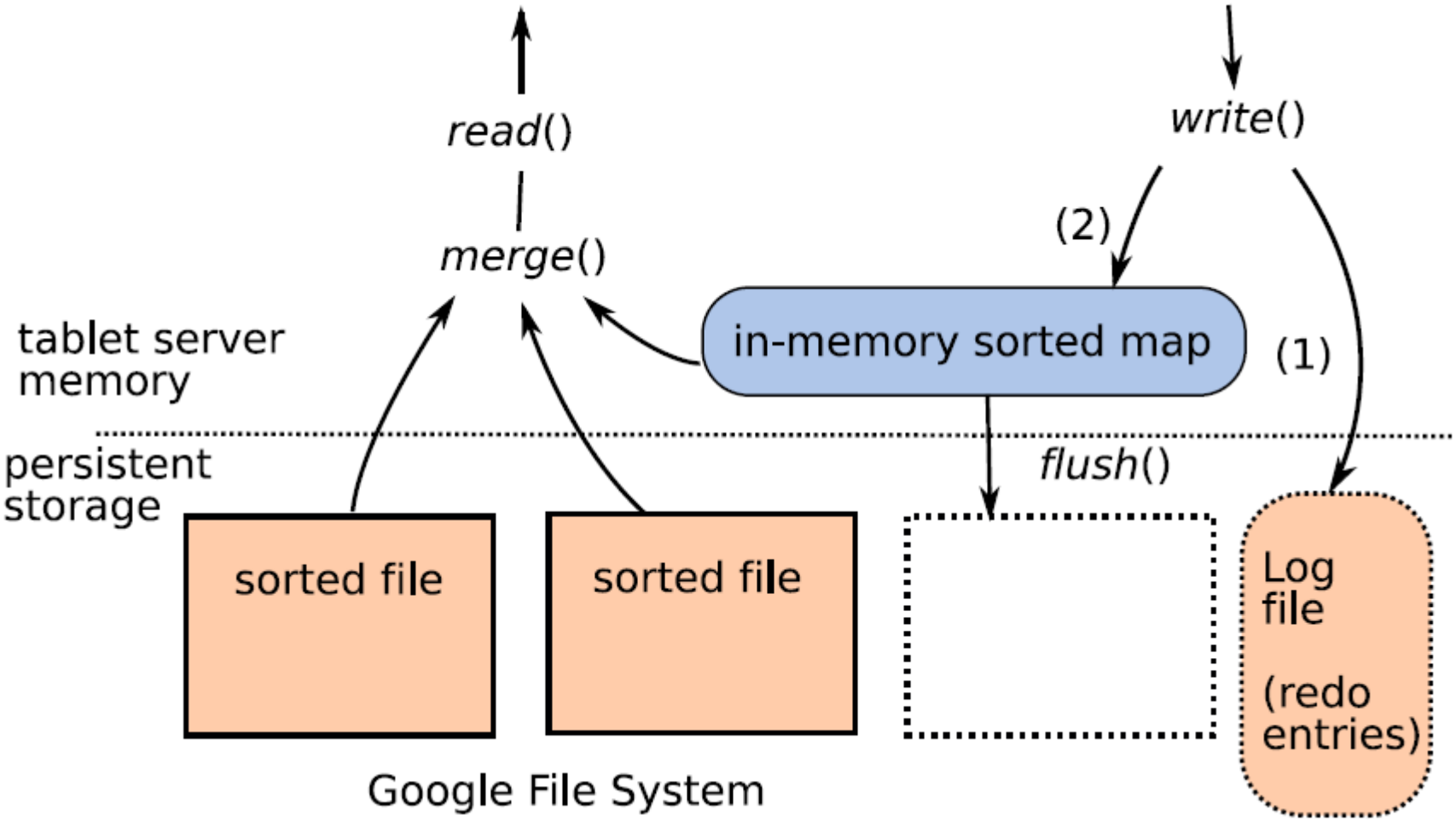


a) A new client contacts a distributed system



b) Using its image, the client directly contacts N


root node
1
2
7
10
4
5
3
6
9
8
[1,231]
[232,562]
[563,682]
[769,1000]
Client
image
1
2
4
5
3
[1,231]
[232,562]
[563,1000]
search(856)
answer +
image adjustment



Persistence Management

Apache Cassandra

- BigTable data model + Dynamo infrastructure
  - Initially developed by Facebook
  - Now used by Twitter and Digg, etc.

Distributed Access Structures

- Key points that you should remember
  - Reliability: always be ready to face a failure somewhere
    - Detect failures and deal with it
    - Use replication

Distributed Access Structures

- Key points that you should remember
  - Scalability: no single point of failure; even load distribution over all the nodes.
    - Distribute (and maintain) routing information: trade-off between maintenance cost and operations cost.
    - Cache an image of the structure (e.g., in the Client): design a convergence protocol if the image gets outdated.

Distributed Access Structures

- Key points that you should remember
  - Efficiency: clearly depends on the amount of information replicated at each node or at the Client
    - Stable systems: the structure can be duplicated at each node, and allows O(1) cost – low maintenance
    - Highly dynamic systems: very hard to maintain a consistent view of the structure for each participant

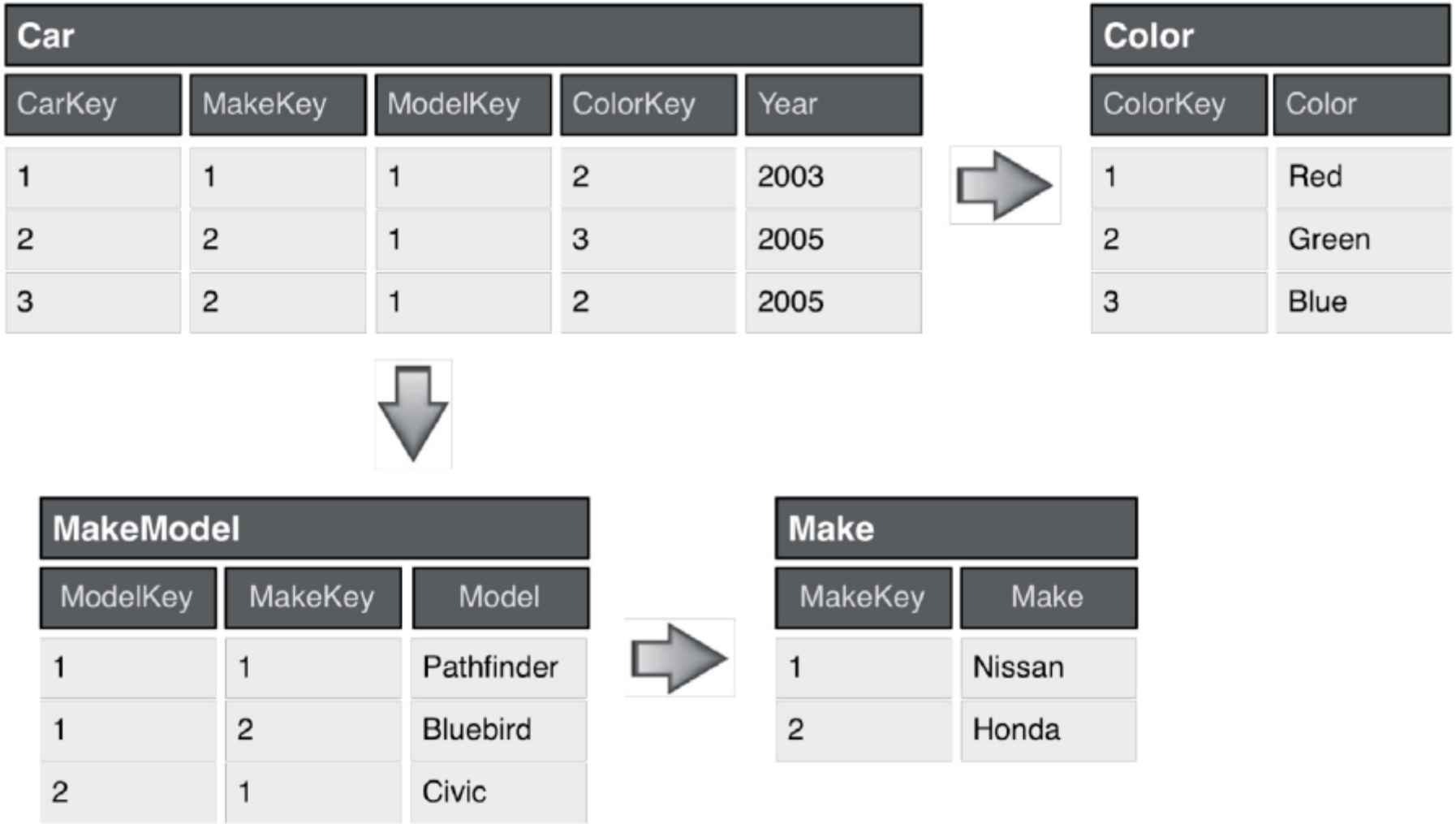
NoSQL Databases

- NoSQL ("not only SQL") is a broad class of database management systems identified by non-adherence to the widely used relational model
  - Relational tables → structured storage (typically key-value pairs)
  - Scaling horizontally:
    - Breaking some of the fundamental guarantees of SQL in order to reach massive scale

Relational Tables

- A Relational Database Management System (RDBMS) stores data as a collection of tables, and provides relational operators to manipulate the data in tabular form.
- All modern commercial relational databases employ SQL as their query language, therefore they are also called SQL databases.

Relational Tables



Car

CarKey	MakeKey	ModelKey	ColorKey	Year
1	1	1	2	2003
2	2	1	3	2005
3	2	1	2	2005

Color

ColorKey	Color
1	Red
2	Green
3	Blue

MakeModel

ModelKey	MakeKey	Model
1	1	Pathfinder
1	2	Bluebird
2	1	Civic

Make

MakeKey	Make
1	Nissan
2	Honda

Relational Tables

- The challenge is *scalability*: SQL databases are hard to be expanded by adding more servers as data have to be replicated across the new servers or partitioned between them
  - Data shipping: time-consuming and expensive
  - Nearly impossible to guarantee maintenance of referential integrity (any field in a table that's declared a foreign key can contain only values from a parent table's primary key or a candidate key)

Structured Storage

- May not require fixed table schemas
- Usually avoid *join* operations
  - if a join is needed that depends on shared tables, then replicating the data is hard and blocks easy scaling
- Often provide weak consistency guarantees
  - such as eventual consistency and transactions restricted to single data items
  - in most cases, you can impose full ACID guarantees by adding a supplementary middleware layer

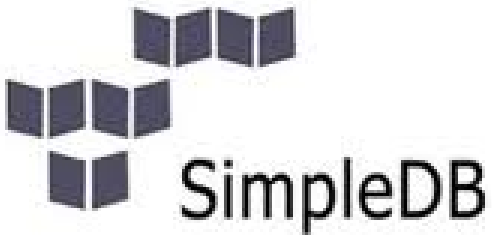
Key-Value Pairs

- Key-value databases are item-oriented
  - All relevant data relating to an item are stored in that item. Therefore data are commonly duplicated between items in a table
  - A table can contain very different items
  - It radically improves scalability by eliminating the need to join data from multiple tables

Key-Value Pairs

Car	
Key	Attributes
1	Make: Nissan Model: Pathfinder Color: Green Year: 2003
2	Make: Nissan Model: Pathfinder Color: Blue Year: 2005 Trans: Automatic

API call	API functional description
`CreateDomain`	Creates a domain that contains your dataset.
`DeleteDomain`	Deletes a domain.
`ListDomains`	Lists all domains.
`DomainMetadata`	Retrieves information about creation time for the domain, storage information both as counts of item names and attributes, and total size in bytes.
`PutAttributes`	Adds or updates an item and its attributes, or adds attribute-value pairs to items that exist already. Items are automatically indexed as they're received.
`BatchPutAttributes`	For greater overall throughput of bulk writes, performs up to 25 `PutAttribute` operations in a single call.
`DeleteAttributes`	Deletes an item, an attribute, or an attribute value.
`GetAttributes`	Retrieves an item and all or a subset of its attributes and values.
`Select`	Queries the data set in the familiar "Select target from *domain_name* where *query_expression*" syntax. Supported value tests are =, `!=`, <, >, <=, >=, `like`, `not like`, `between`, `is null`, `isn't null`, and `every()`. Example: `select * from mydomain where every(keyword) = "Book"`. Orders results using the `SORT` operator, and counts items that meet the condition(s) specified by the predicate(s) in a query using the `Count` operator.



SQL vs NoSQL

Database use	Challenges faced with a cloud database
Transactional support and referential integrity	Applications using cloud databases are largely responsible for maintaining the integrity of transactions and relationships between tables.
Complex data access	Cloud databases (and ORM in general) excel at single-row transactions: get a row, save a row, and so on. But most nontrivial applications have to perform joins and other operations that cloud databases can't.
Business Intelligence	Application data has value not only in terms of powering applications but also as information that drives business intelligence. The dilemma of the pre-relational database, in which valuable business data was locked inside impenetrable application data stores, isn't something to which business will willingly return.

Take Home Messages

- The challenges and opportunities of big data
- Large-scale distributed file systems
  - GFS (HDFS)
- Large-scale distributed access structures
  - Dynamo (Voldemort), BigTable (HBase)
- NoSQL databases