storage + processing with distributed systems, at the age of big data

Erwan Le Merrer
erwan.lemerrer@technicolor.com

technicolor

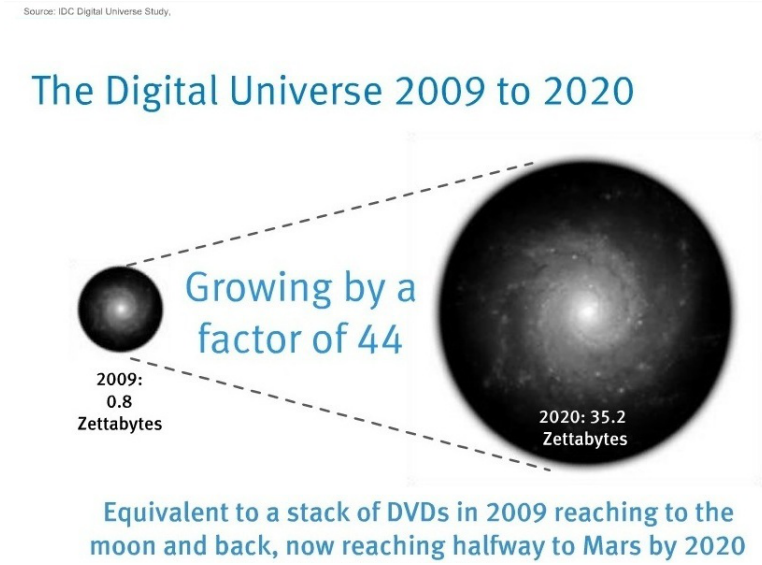
Outline

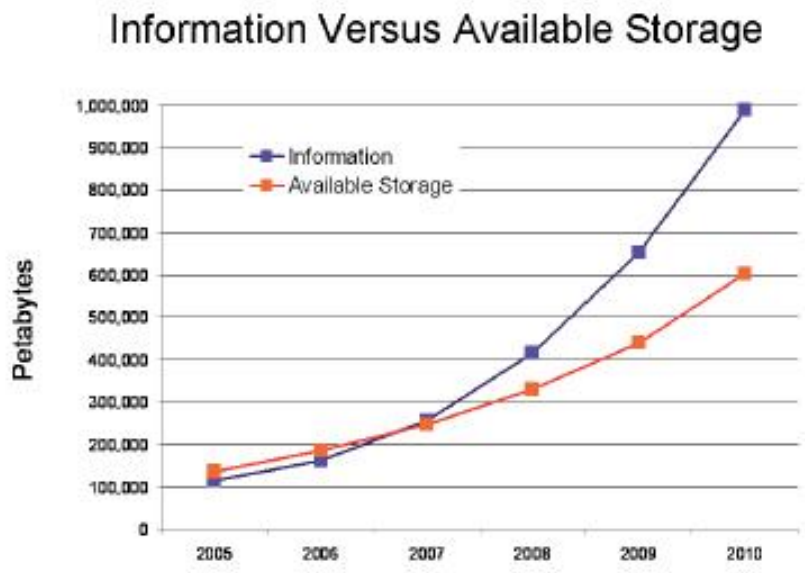
Outline

1. **Big Data, consequences, and some roadmap**
2. **Data storage**
   - NoSQL datastores
   - In memory storage
   - Efficient storage and repair
3. **Processing**
   - Batch processing
   - Incremental processing
   - Stream processing
4. **What's next ?**
   - In software/platform
   - In hardware
   - In application logic

technicolor

How big are big data?



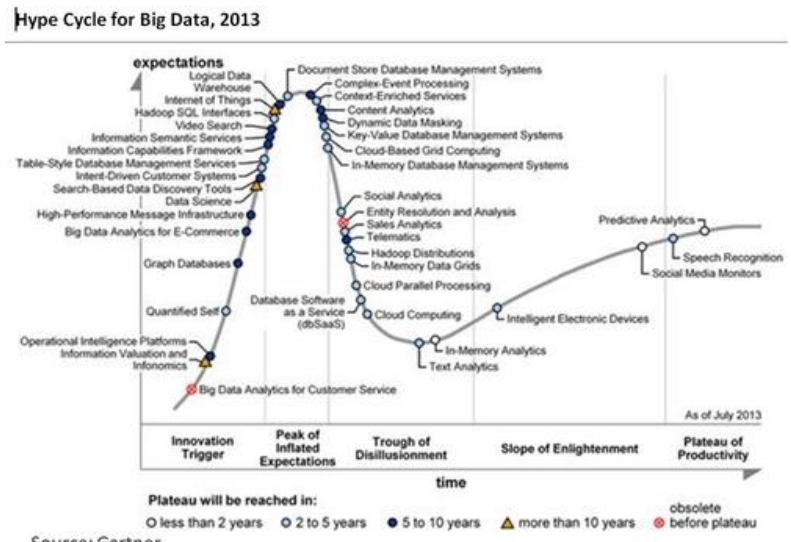


- 90% of all the data in the world created over the past 2 years
- 2010: enterprises + users: 13 exabytes of new data storage
- US will need around 190,000 workers with analytical skills + 1,5 millions managers aware of it due to big data applications
- Future contributors: IoT + ever increasing digitalization/tracking

0 next 3 slides: big data and its technical challenges, CACM'14

technicolor

But... what are we talking about exactly?



Source: Gartner

- Convergence around 3 V's (at least!)
  - Variety: heterogeneity of data types and representation
  - Volume: huge amount of data to process
  - Velocity: high arrival rate and rush to be exploited
- Interconnection + cheap devices = means to produce more and more data.
- Used for data-driven decision making

```
while true do:
   cat small_data >> big_data;
done
```

```
bin/hadoop jar hadoop-*-examples.jar
  \ randomwriter <out-dir>
```


technicolor

Real problem from real life ?

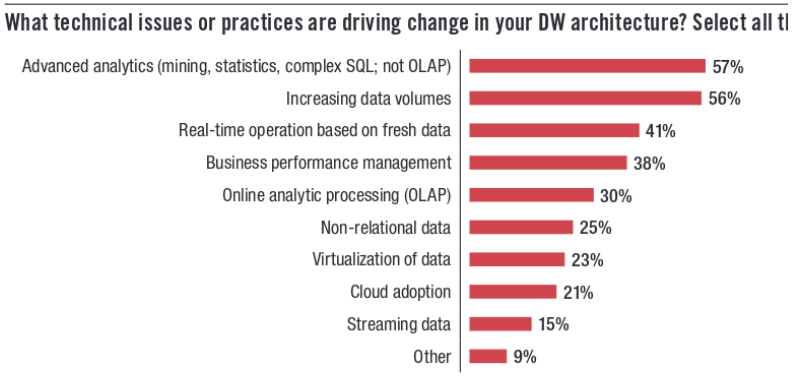


Figure 4. *Based on 1,688 responses from 538 respondents; 3.1 responses per respondent, on average.*

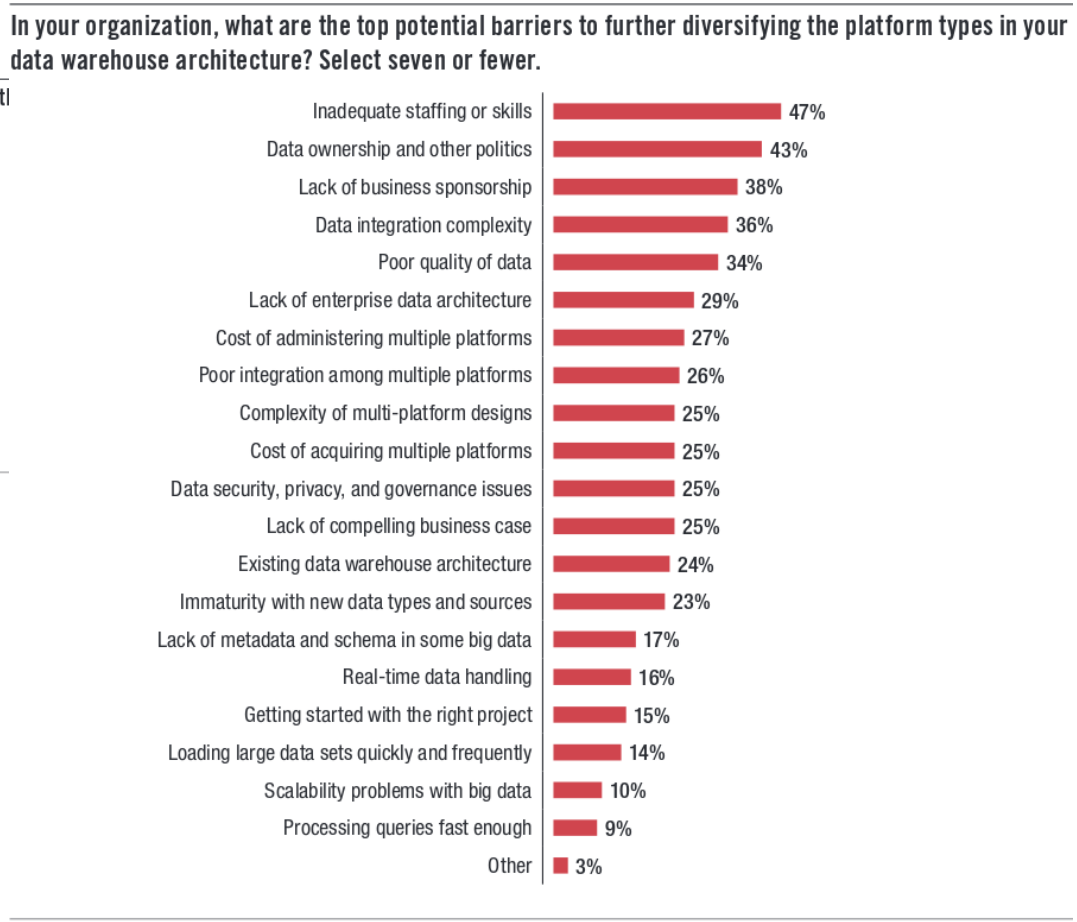


Figure 9. *Based on 2,758 responses from 538 respondents; 5.1 responses per respondent, on average.*

Plans in more details

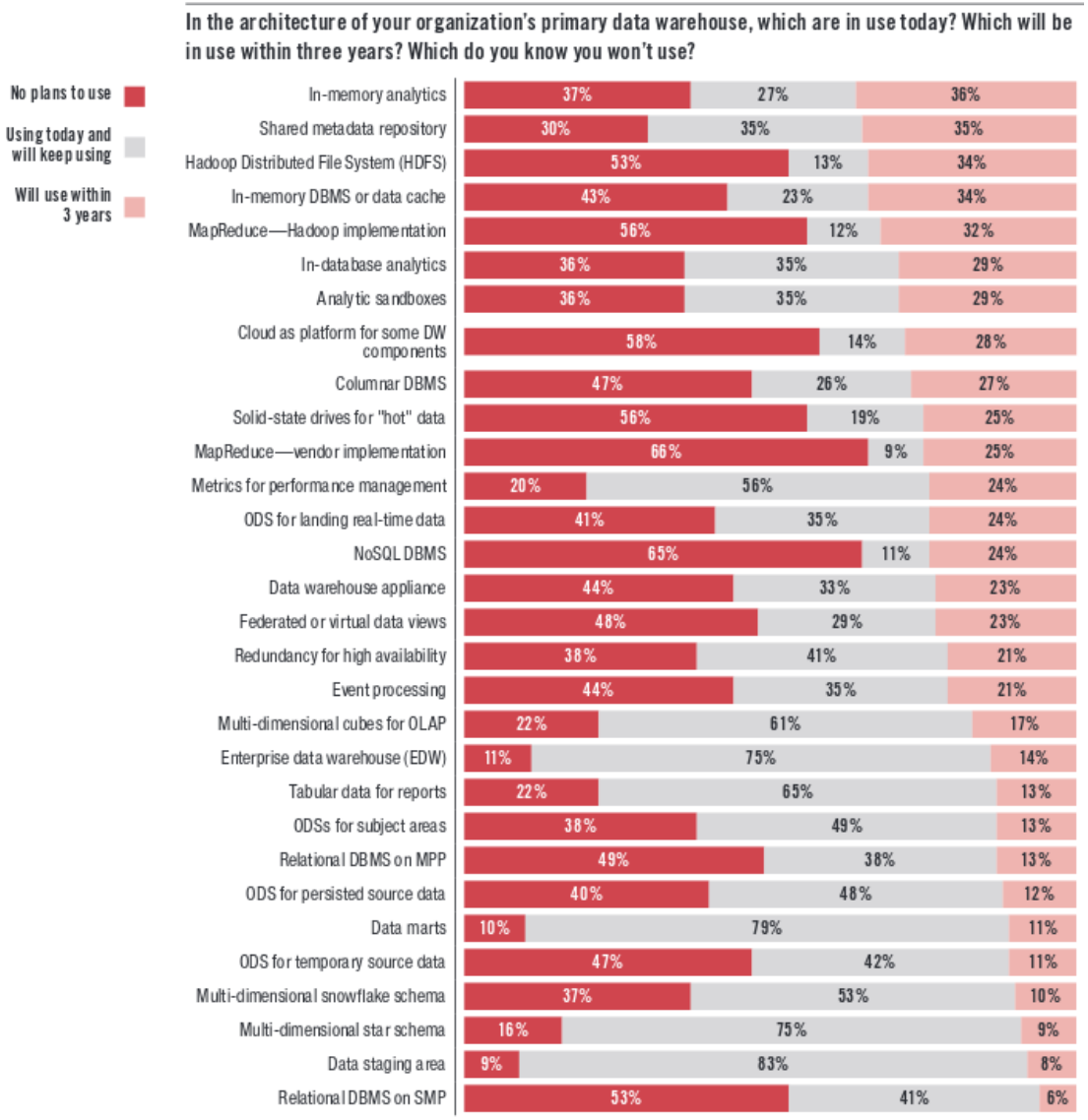
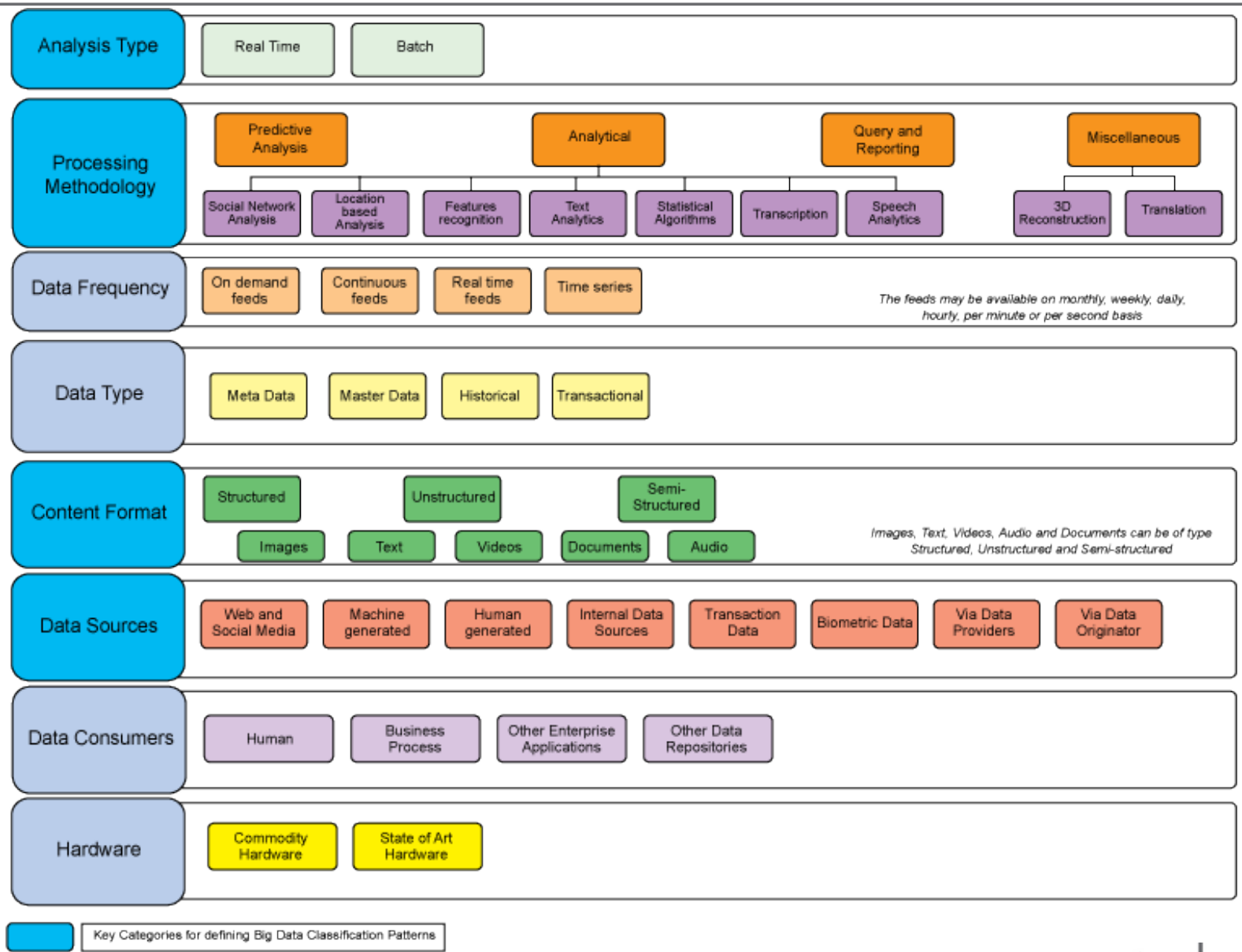


Figure 12. *Based on 538 respondents. The chart is sorted by the "will use within three years" column. This data set is also the basis for subsets charted in Figures 13, 14, and 15.*

technicolor

Classification and vocabulary



0 credit: IBM, 2013

technicolor

What to do with this data?

- Counting: fast increments, no data kept
- Aggregation: join needed, increase states required
- Research problems: keep a lot more, do not know what is useful

Example pipeline for aggregation or research:

0 Realtime Analytics for Big Data A Facebook Case Study, 2012. Picture: Oracle.

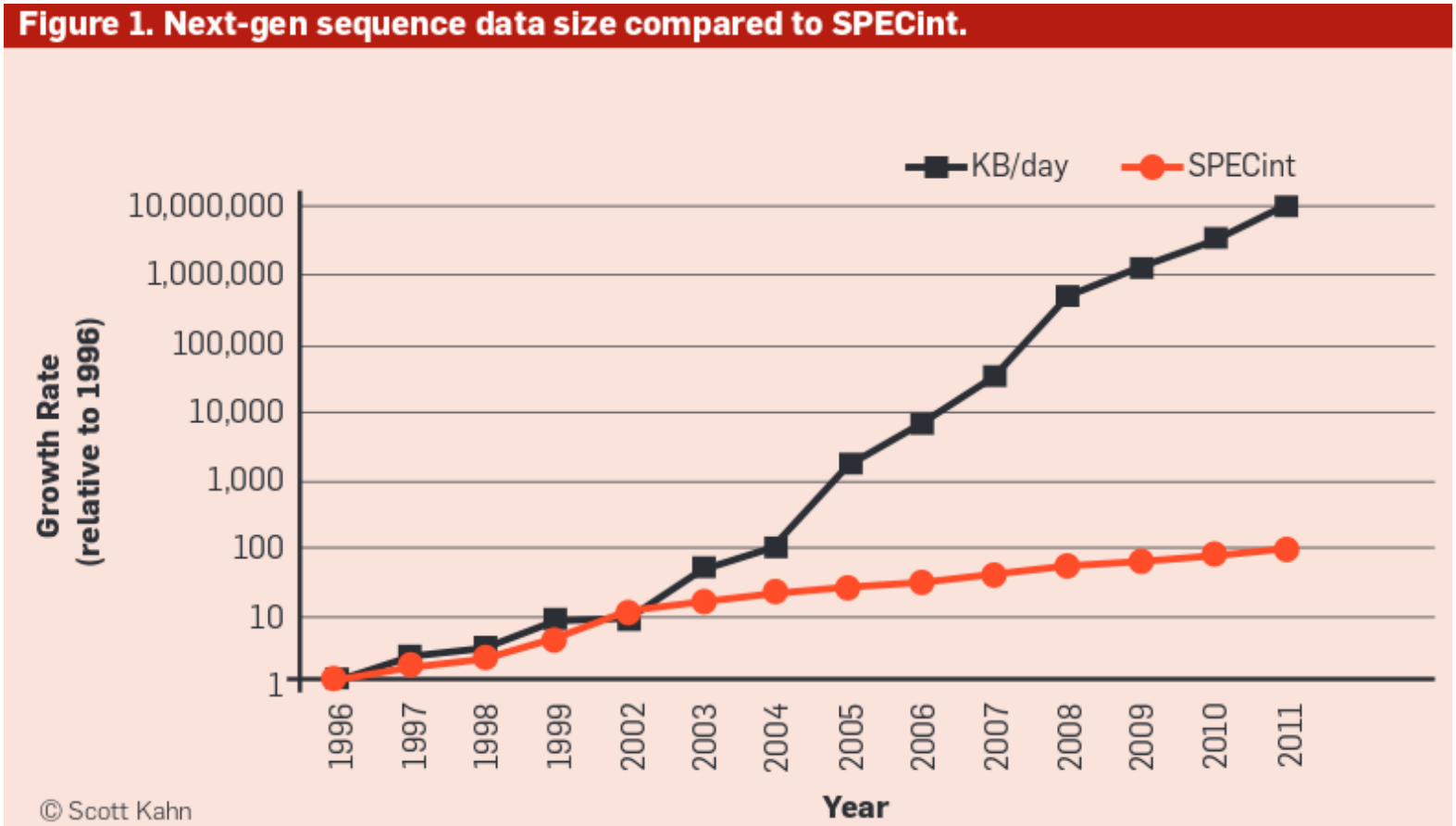
technicolor

Big data life cycle

Acquisition $\rightarrow$ extraction/cleaning $\rightarrow$ representation $\rightarrow$ modeling $\rightarrow$ interpretation

1. Filtering/compression of data before exploitation. Need for incremental ingestion
2. Extraction/cleaning are application dependant, and tolerance to faulty data due to unreliable data sources
3. Integration, aggregation and representation in convenient data structure to work with, in scalable data stores
4. Fit data into models / approximations to get good results without being overwhelmed by particular informations
5. Previous analysis results to be understood to (in turn) be leveraged. Do results have a meaning? If not, remodel and re-loop

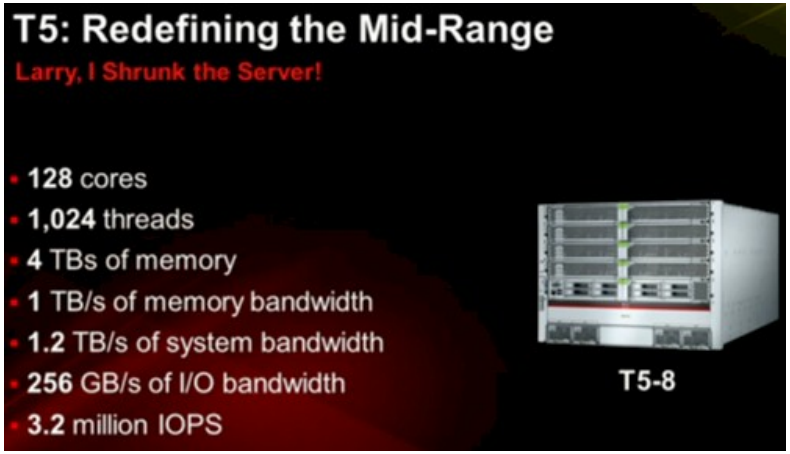
Problem: Moore's Law cannot help anymore



- e.g. Next Generation Sequencing: exponential data increase at a single machine. Outpaces CPU bench on that machine...

technicolor

So why not many cores/disks/... on **one** serveur ?



Imagine a service implemented on a single super-computer; it becomes

- **very** expensive. Often vendor lock-in
- a performance **bottleneck** (no infinite scale-up!)
- a **single point of failure**
  - service should be **available** at all times, despite hard/soft failures
  - only way to provide availability is through **redundancy**
  - redundancy must be geographically **distributed** (within/outside datacenter)

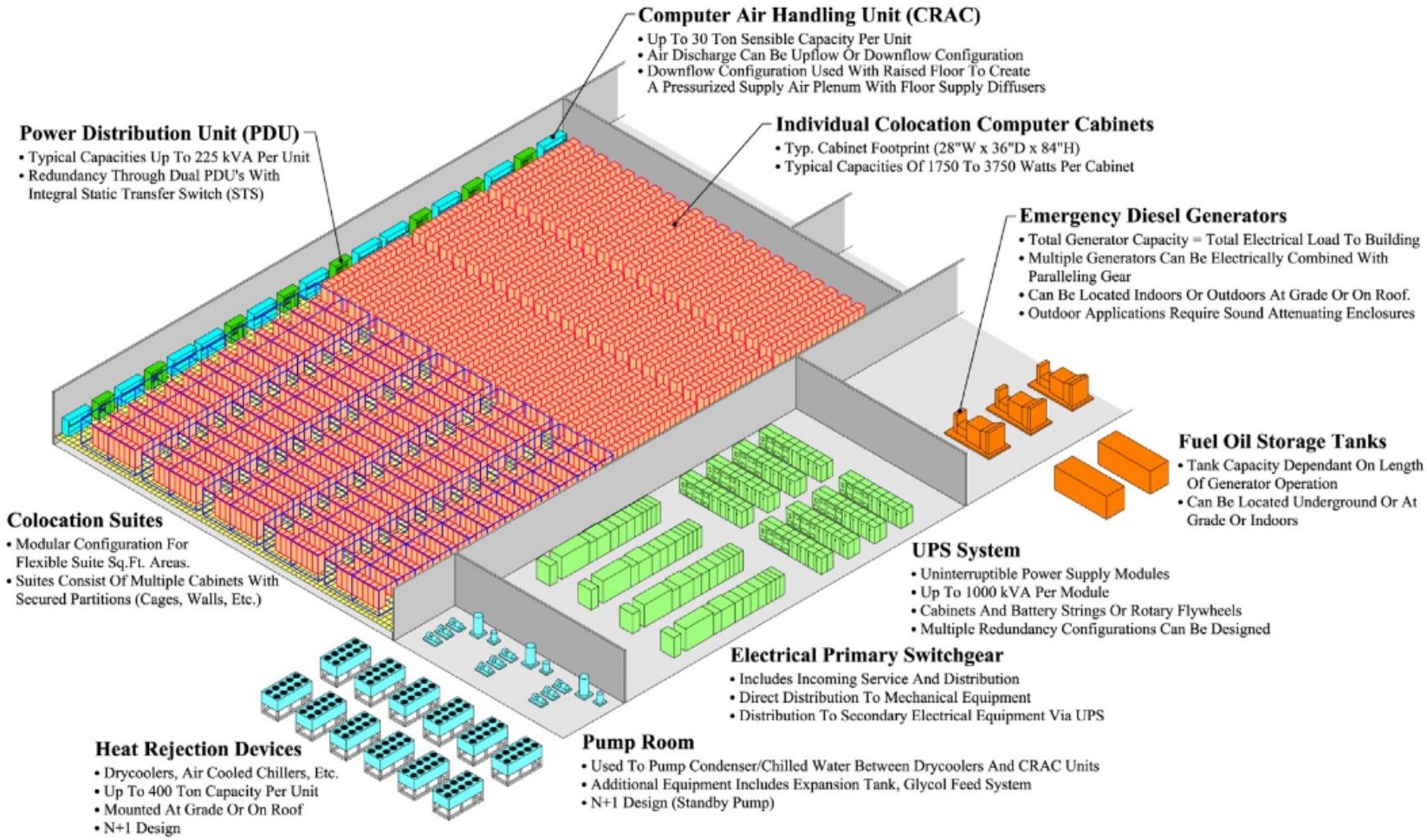
technicolor

Solution: distributed processing

- The classic setup: commodity hardware. Remove if problem (2-10%: disk annualized failure rate)
- (2010) Warehouse @ Google: 10,000+ machines, 1GB/s net., 6x1TB disk/machine
- What is new: cost/GB lower, machine failure impact higher, machine throughput higher
- Steady issues: network latency (RPC), disk throughput and latency
- Performance metrics for big systems
  - Performance (measurements on throughput)
  - Scalability (is system handling 10^x nodes more?)
  - Reliability (accountable system, persistence)
  - Availability (data accessible at time t?)

The new place for storing and computing

The datacenter



0 credit: BDL Associates

technicolor

Typical first year for a new cluster

At this scale, **failures are the norm**

~0.5 overheating (power down most machines in <5 mins, ~1-2 days to recover)

~1 PDU failure (~500-1000 machines suddenly disappear, ~6 hours to come back)

~1 rack-move (plenty of warning, ~500-1000 machines powered down, ~6 hours)

~1 network rewiring (rolling ~5% of machines down over 2-day span)

~20 rack failures (40-80 machines instantly disappear, 1-6 hours to get back)

~5 racks go wonky (40-80 machines see 50% packet loss)

~8 network maintenances (4 might cause ~30-minute random connectivity losses)

~12 router reloads (takes out DNS and external vips for a couple minutes)

~3 router failures (have to immediately pull traffic for an hour)

~dozens of minor 30-second blips for DNS

~1000 individual machine failures

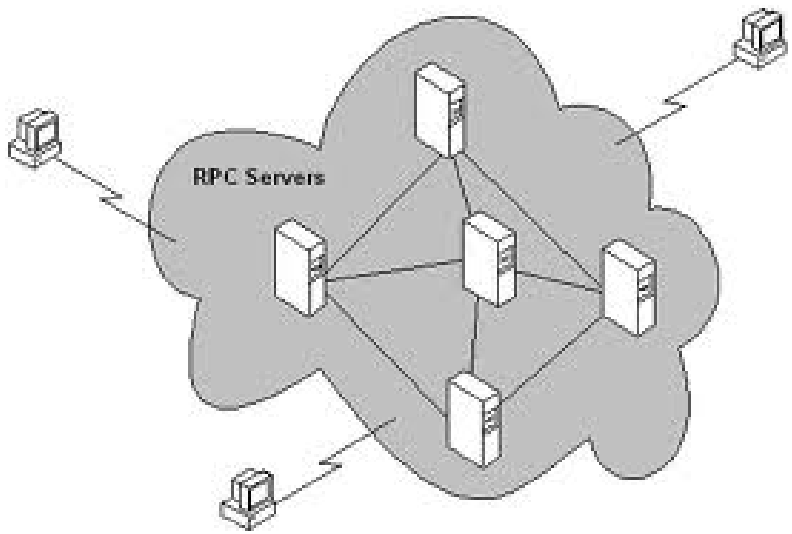
~thousands of hard drive failures

Slow disks, bad memory, misconfigured machines, flaky machines, etc.

0 credit: (Jeff Dean, Google)

Main characteristics of a distributed system

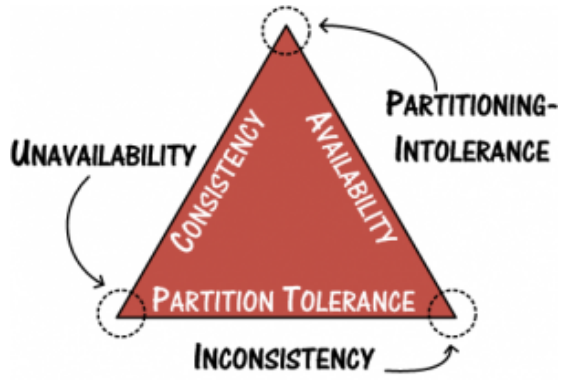
- Independant machines, processors, processes
- Message passing (no shared memory), newer models (*eg*, **BSP**)
- No shared clock
- Inpredictable and independant failures
- Basic operations
  - send message
  - receive message
  - compute locally



0

credit: Amr el Abbadi

Nice but ...



- CAP theorem
  - Desirable to have Consistency, High-Availability and Partition-tolerance in systems; **unfortunately no system can achieve all three at the same time**
  - CA: RDBMS; AP: NoSQL stores; CP: not an attractive option!
- Distributed processing is harder to apprehend
  - notions like time, ordering, synchronisation, ...

technicolor

What we will be talking about

Cloud storage/processing software for handling big data.

- But not about many other things
  - P2P or inter-cloud storage/processing
  - Specific algorithms for handling that data
  - data privacy
  - human perspective / collaboration
  - ...

technicolor

Outline

1 **Big Data, consequences, and some roadmap**

2 **Data storage**
- NoSQL datastores
- In memory storage
- Efficient storage and repair

3 **Processing**
- Batch processing
- Incremental processing
- Stream processing

4 **What's next ?**
- In software/platform
- In hardware
- In application logic

technicolor

GFS: the environment

- Racks: (2004) 30+ clusters, with 2000+ computers each
- (2010) Millions of very large files (>100MB), 10P !
  - Small files supported, but not optimized
- Writes are mostly appended (sequential)
  - Small write at arbitrary location not optimized
- Concurrency: essential, sync minimal

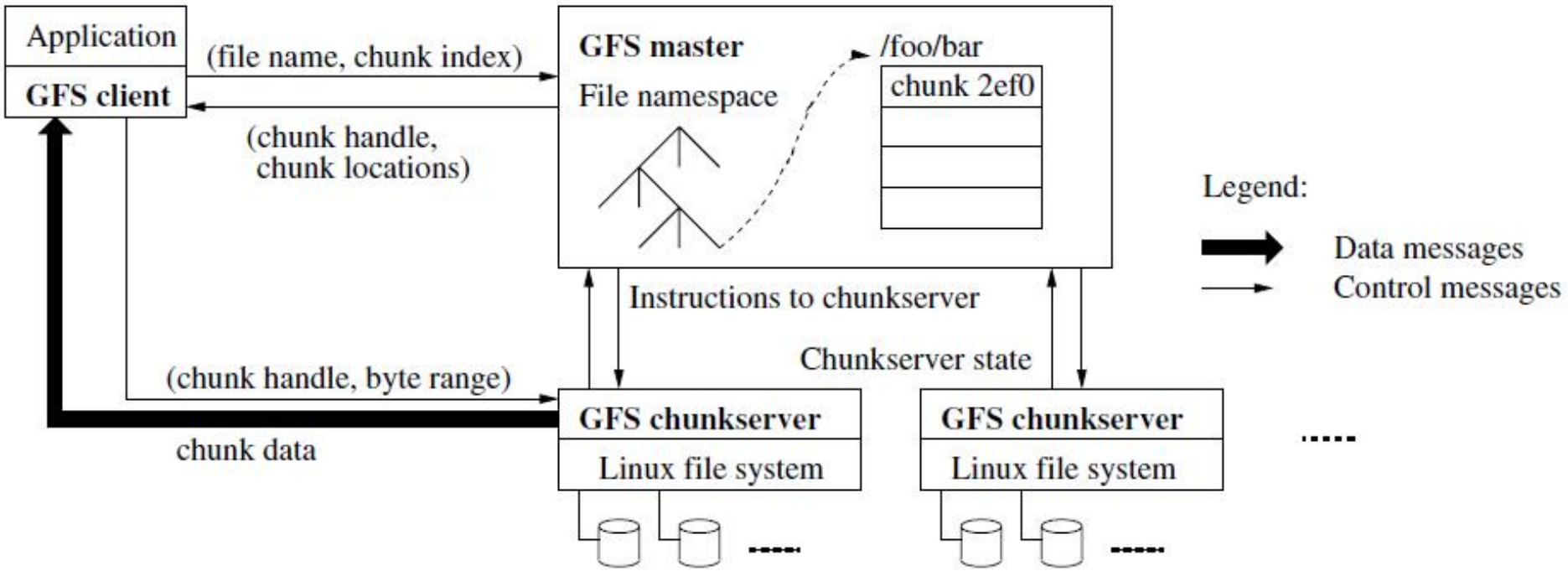
Architecture:

- Files divided into (large) chunks (64MB+64b metadata)
  - Less interactions/overhead from clients to server
  - Reduce size of metadata on main server
- Master / slaves model
  - Master: Maintains all File System metadata
  - Chunkservers: store chunks and replicates

Access from multiple clients: google apps

0 The google file system, SOSP 2003

GFS: the big picture



- Single master simplifies design and replication decision
  - Avoid bottleneck by relying on master only for lightweight operations (metadata in RAM)
- No caching on client or chunkservers
  - Remove cache coherence issues (sync minimal)
  - No penalty as most applications stream huge files

technicolor

HDFS: the free implementation

- Written in Java and is supported on major platforms
- Hadoop supports shell-like commands to interact with HDFS directly

```
hdfs dfs –mkdir /user/hadoop/dir1 /user/hadoop/dir2
hdfs dfs –put localfile /user/hadoop/hadoopfile
```

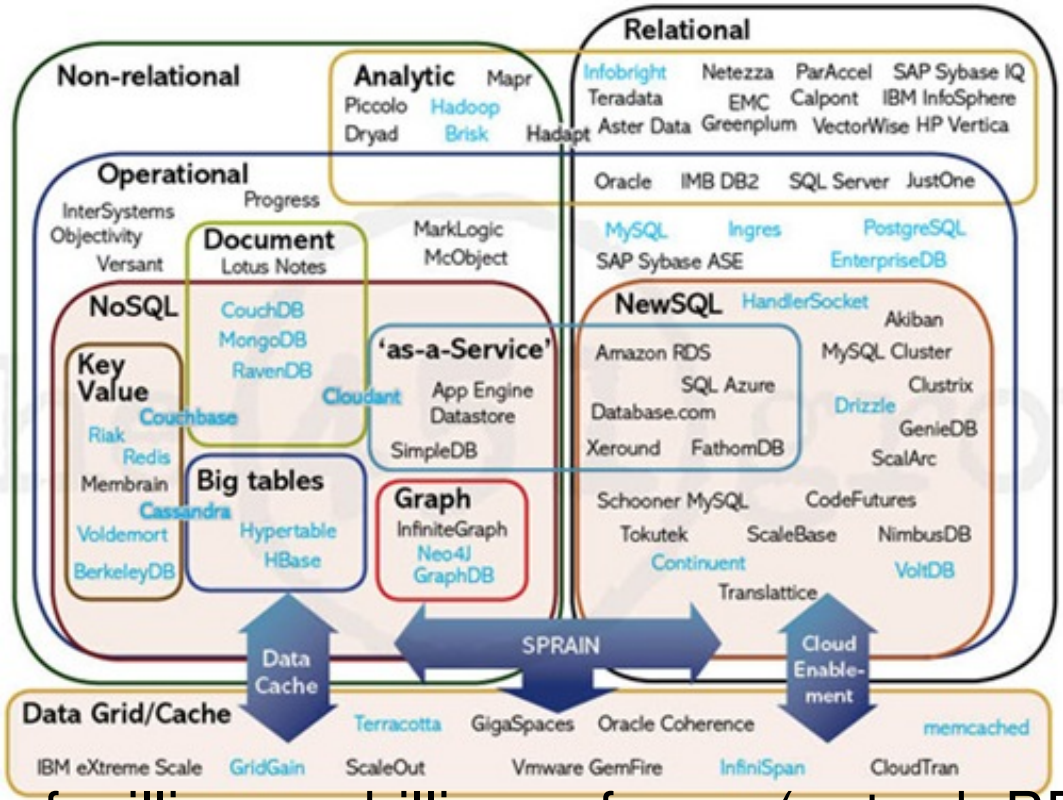
- write-once-read-many access model for files. A file once created, written, and closed need not be changed
  - simplifies data coherency issues and enables high throughput data access
  - (A MapReduce application or a web crawler application fits perfectly with this model!)

DEMO later on...

0 http://hadoop.apache.org/hdfs/

technicolor

NoSQL scalable stores



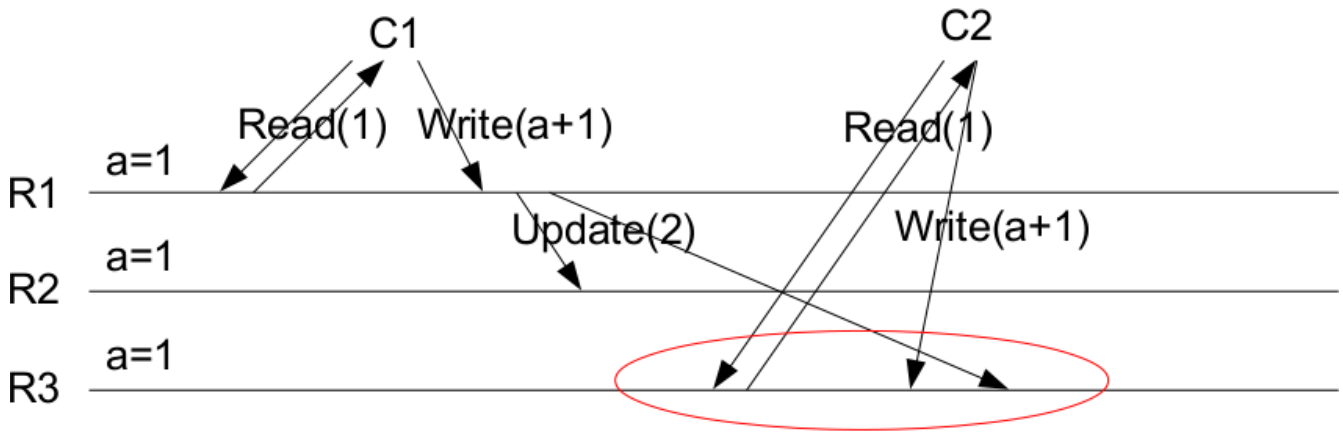
- Hundreds of millions or billions of rows (or trad. BDs are OK)
- Avoid join operations (very costly)
- No classic ACID prop. (atomicity/consistency/isolation/durability)
- The key-valued model
  - UID as the key
  - Only use key for consistent access to data

0 fig: http://cdi-mdm.blogspot.fr

technicolor

Consistency notions (sketch)

- Replication required to mask failures
- but triggers consistency issues. Is the following acceptable?



- well... depends on the application requirements
- Strong vs. relaxed consistency models
  - *Strong*: after a write, reads return updated value
  - *Weak*: no guarantee on value freshness, there is an inconsistency window
  - *Eventual*: after all updates to an object stop, then all subsequent reads return the updated value

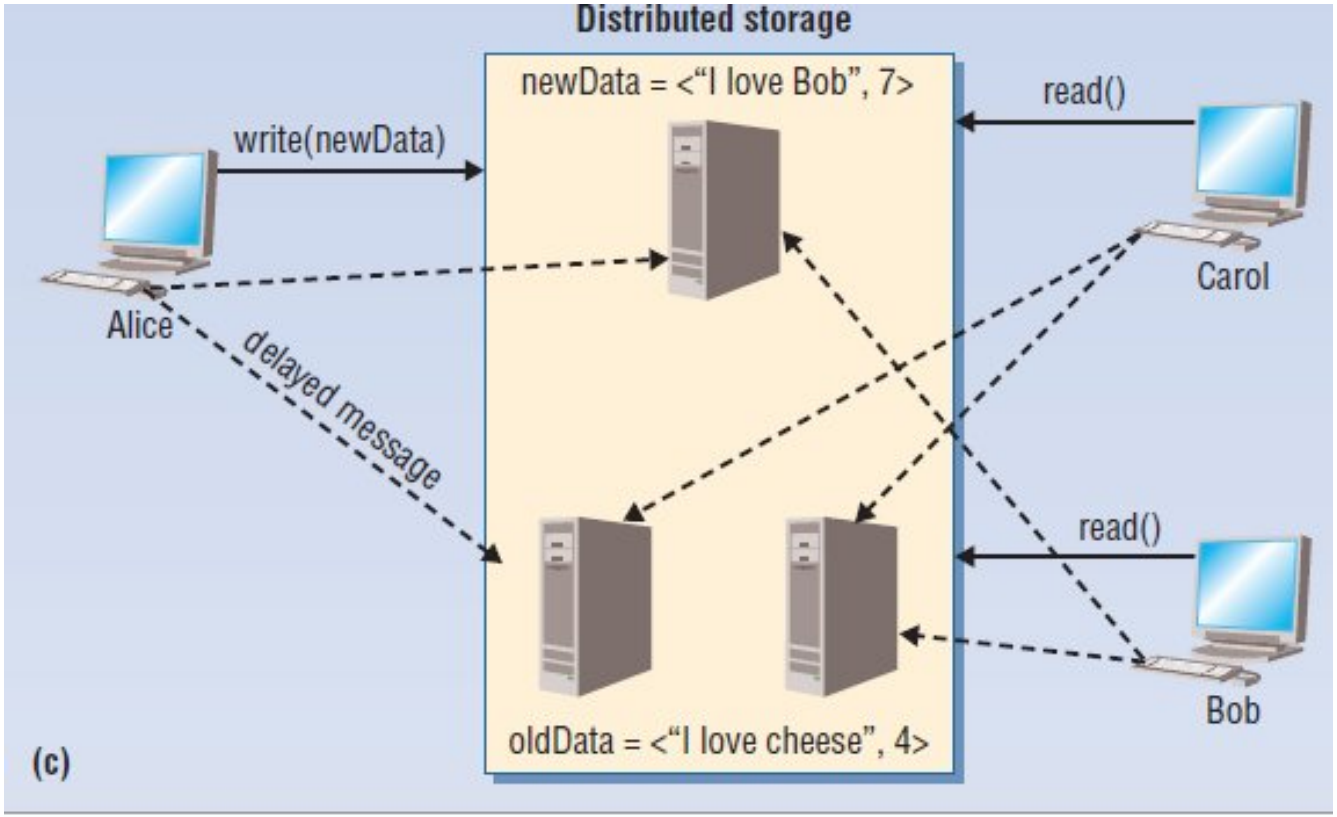
technicolor

ABD protocol: strong consistency

- Asynchronicity: crashed or slow node? Cannot distinguish!
- Example majority replication (strong)
- ABD protocol intuition: any two majority of replicas have a node in common, so a minority can crash
- **(value,timestamp)** pair stored at each storage node
- write
  - Get Phase: client asks replicas for their vt-pairs; wait a majority of responses. Choose a higher t that the higher received
  - Set Phase: client asks to store its vt-pair. Replicas do so if t > its own. Client waits for a majority of ACK (sent in any case); then OK
- read
  - Get Phase: client asks replicas for their vt-pairs; wait a majority of responses. Choose the highest vt-pair
  - Set Phase: client asks to store this vt-pair. Replicas do so if t > their own. Client waits for a majority of ACK (sent in any case); then read return v as a response

ABD protocol: strong consistency (cont.)

Set in Read is needed to avoid oscillation while a write is in progress:



- This simple and provable protocol can tolerate a minority of replica crashes, and works in purely asynchronous systems

0

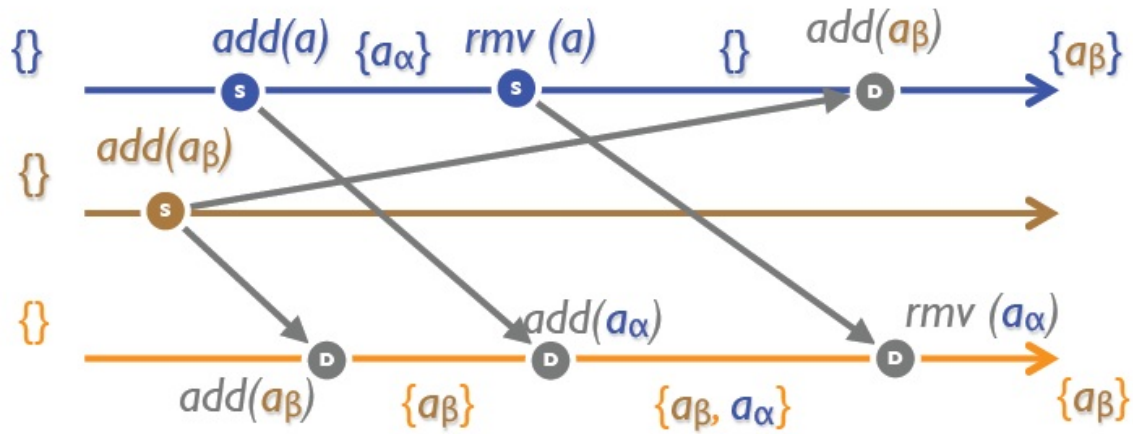
credit: Reliable distributed storage, IEEE 2008

technicolor

CRDTs: eventual consistency

In some (limited) cases, a radical simplification is possible:

- If concurrent updates to some data commute, and all of its replicas execute all updates in causal order, then the replicas converge **eventually**



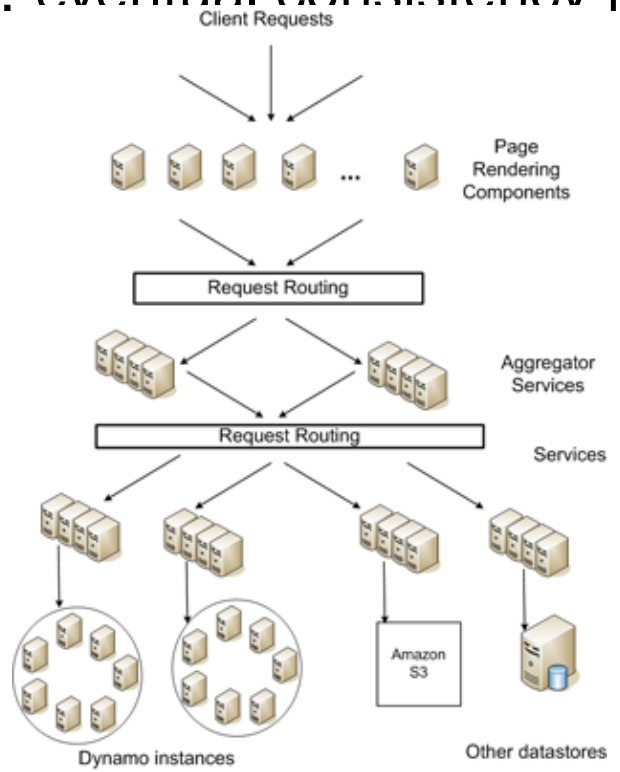
Weak, as read may not be up to date, but no sync/locks: system is highly available

0 CRDTs: Consistency without concurrency control, TR 2010

technicolor

NoSQL Key-Value Stores: Amazon's dynamo

- CAP $\rightarrow$ AP
- giant eCommerce: Shopping carts, Session management, Product catalog
- More decentralized than the GFS (no master, hotplug)
- For performance: eventual consistency in a production system!

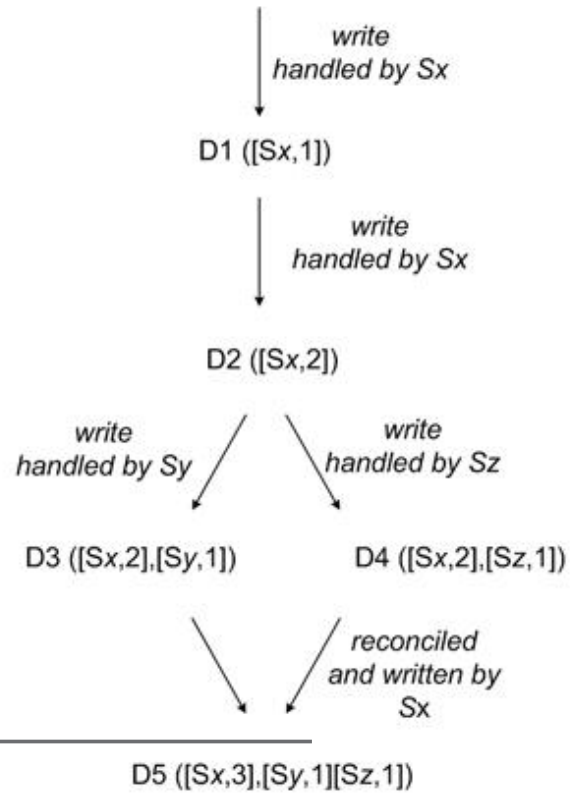


0 Dynamo: Amazon's Highly Available Key-value Store, SOSP 2007

technicolor

NoSQL KVS: Amazon's dynamo (cont.)

- Store and locate with <key,value> pairs
- Replicas are handled in relaxed consistency
  - Strong consistency may impose delays if some servers are overwhelmed; transactions must be possible at anytime
  - Eventual consistency allows temporary inconsistencies, that will be solved later on (most of time invisible to the user)
- A posteriori, **reconciliation** between replicas
- Needs vector clocks <node,cptr>
- Example: no causal relation bw D3&D4 $\rightarrow$ reconciliation D5
- In practice: "cart merging"
- Errors possible, but cheaper for Amazon to refund !



How to remove a central server? (I)

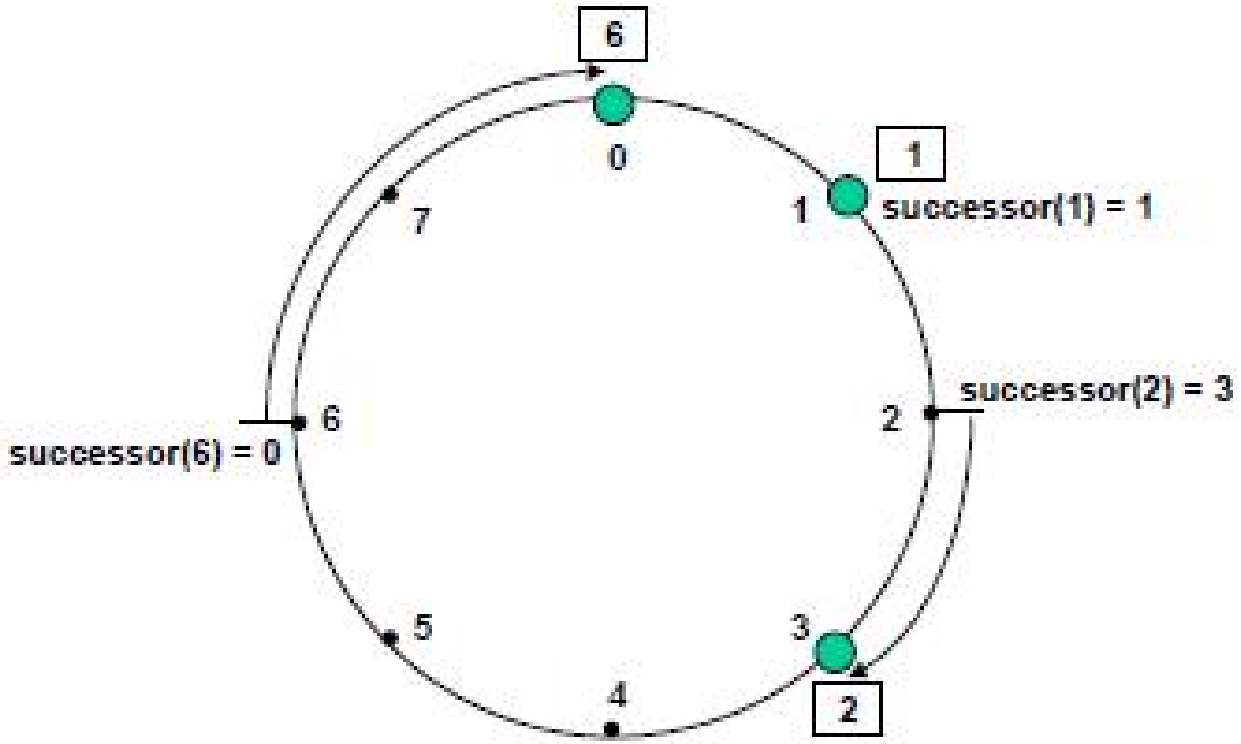
Chord: a Key Based Routing

- Store and locate <key,value> pairs
- Fully distributed mechanism (familly of structured network)
- Provably scalable ($O(\log n)$ with n the number of servers)
- Based on properties of consistent hashing
  - hash function assigns each server and data a m-bit identifier, by hashing IP, e.g. address:port

0 Chord: A Scalable P2PLookup Service for Internet Applications, Sigcomm '01

Chord: protocol sketch (II)

- IDs are ordered in a circle (modulo 2^n)
- Key is assigned to first node whose ID is equal/follows clockwise (called the successor)

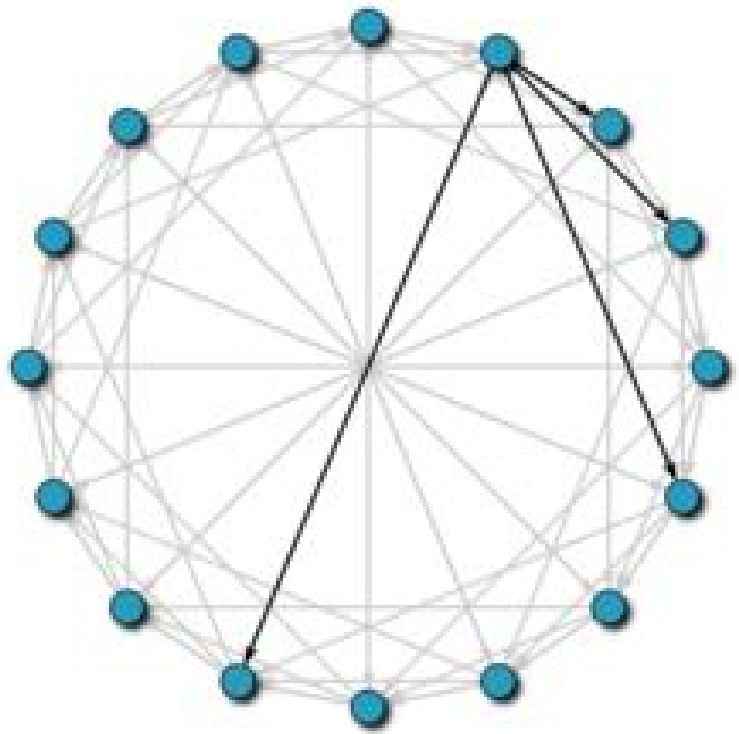


0

pics: wikipedia

Chord: protocol sketch (III)

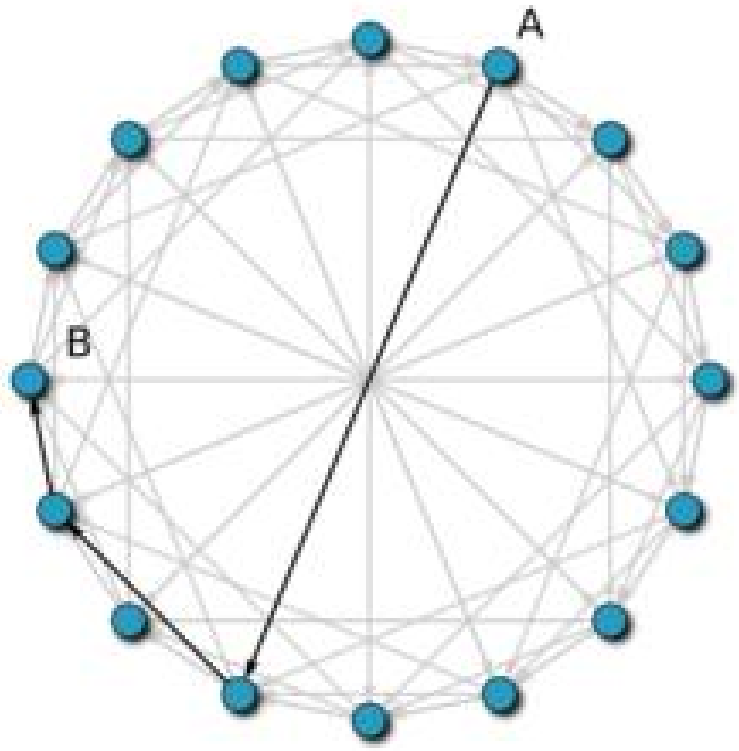
- Each node n maintains a finger table of m entries
- The k^{th} entry contains ID of the node that succeeds n by at least $n + 2^{k-1} \mod 2^m, 1 \leqslant k \leqslant m$



- Then outputs a well balanced structure to search on

Chord: protocol sketch (IV)

- Property: in a N node network, $O(\log N)$ nodes have to be contacted to find a responsible for a key w.h.p.
- Proof idea: remaining distance is halved at each step. After $\log N$ steps, distance is at most $2^m/N$, that is 1 ID lying in that space in expectation



technicolor

NoSQL: Google's Bigtable

- CAP $\rightarrow$ AP (asynchronous, eventually consistent replication)
- A table is a sparse, distributed and multi-dimensional map
- Data indexed on rows and columns
- row = unit of transactional consistency
- value (i.e. data) is not interpreted by the DB system

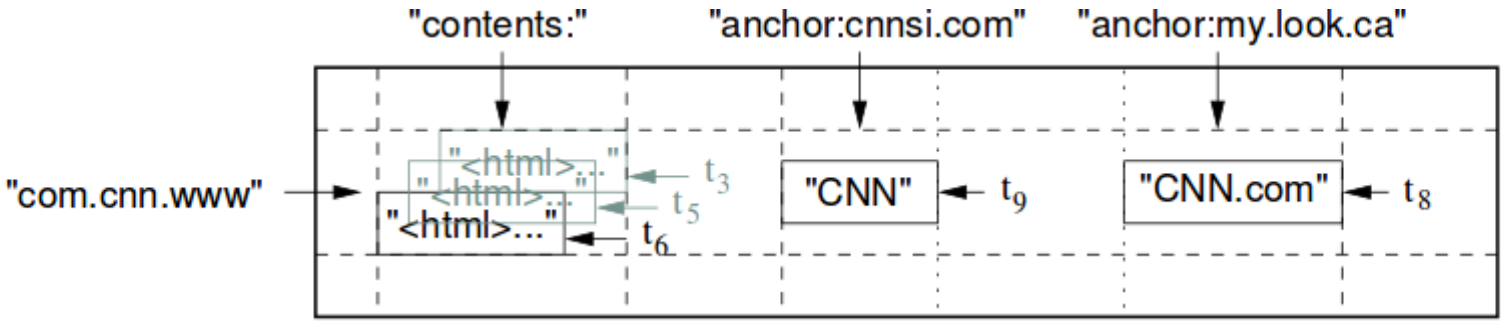


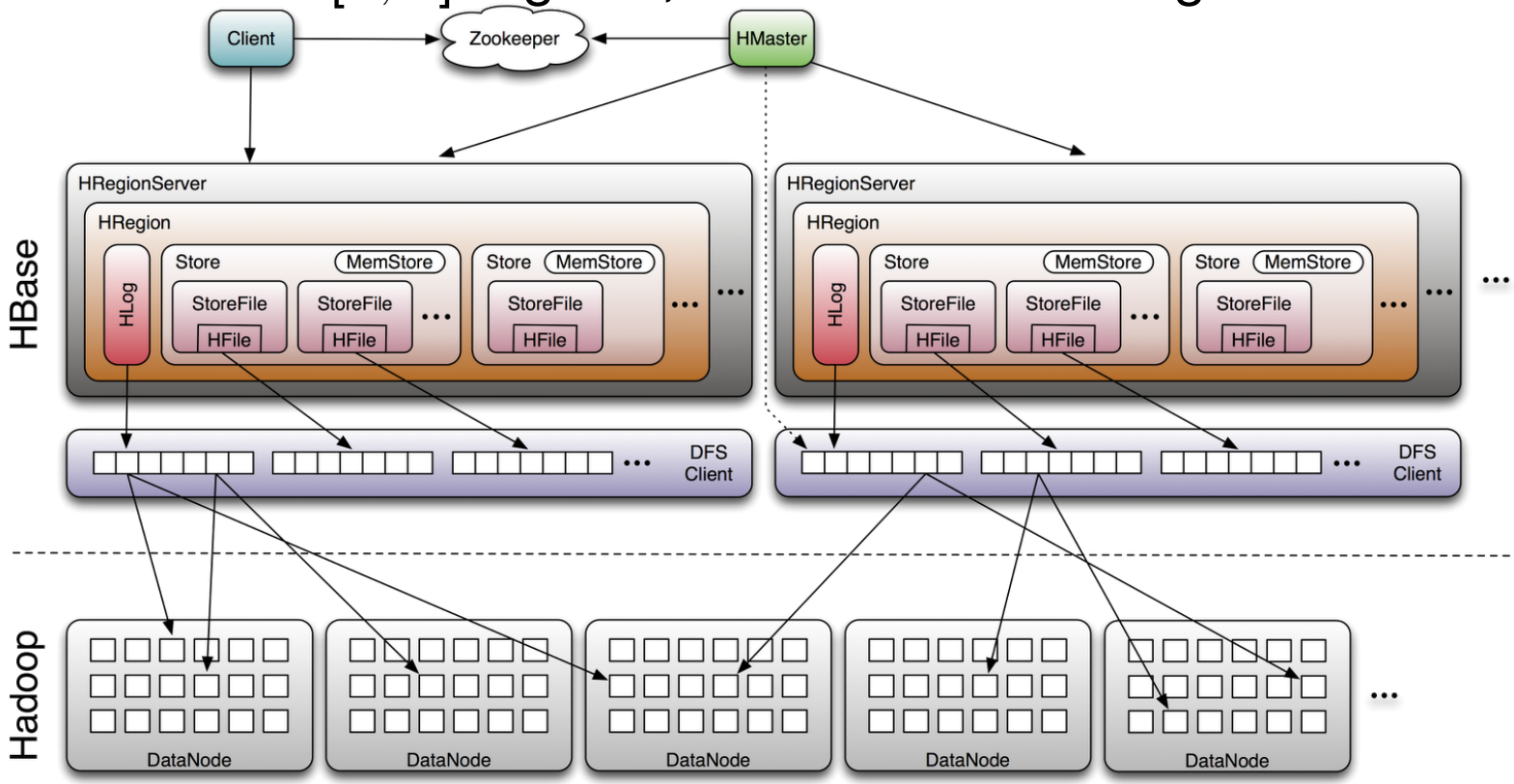
Figure 1: A slice of an example table that stores Web pages. The row name is a reversed URL. The contents column family contains the page contents, and the anchor column family contains the text of any anchors that reference the page. CNN's home page is referenced by both the Sports Illustrated and the MY-look home pages, so the row contains columns named anchor:cnnsi.com and anchor:my.look.ca. Each anchor cell has one version; the contents column has three versions, at timestamps t_3, t_5, and t_6.

0 Bigtable: A Distributed Storage System for Structured Data, SOSP 2006

technicolor

HBase: the free implementation

- vs HDFS: provides fast individual record lookups in files (tables)
- IS built on top of HDFS
- Strongly consistent reads and writes: CA ($\neq$ from Bigtable!).
- One table = [1, k] regions, distributed on HRegionServers



picture: toadworld.com

technicolor

HBase: some commands

- Create a table

```
hbase> create 'test', 'cf'
0 row(s) in 1.2200 seconds
```

- List info about that table

```
hbase> list 'test'
TABLE
test
1 row(s) in 0.0350 seconds
=> [``test'']
```

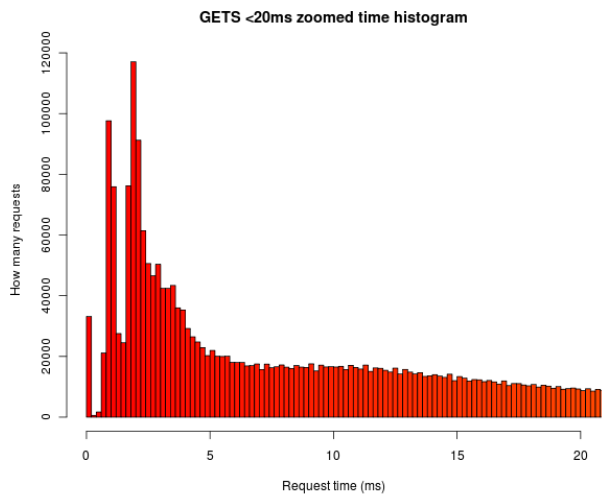
- Put data into it

```
hbase> put 'test', 'row1', 'cf:a', 'value1'
0 row(s) in 0.1770 seconds
```

- Get a single row of data

```
hbase> get 'test', 'row1'
COLUMN                CELL
 cf:a timestamp=1404, value=value1
1 row(s) in 0.0230 seconds
```

- Random R/W benchmark: 7 server cluster (8cores/32GB), 3 billion records (128-256 bytes per row, spread in 1 to 5 columns)

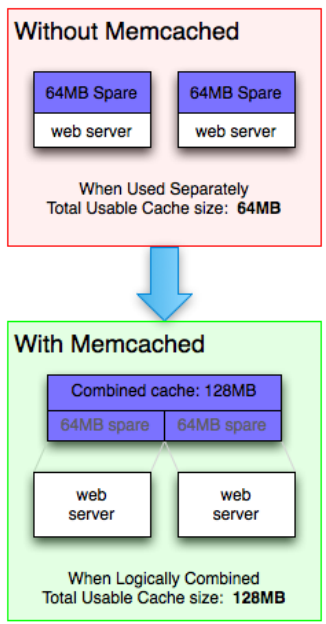


benchmark: http://hstack.org

technicolor

Memcached: object caching in memory

- Goal: un-load datastores used by web apps
- used by: Youtube, Flickr, Facebook, Twitter, WordPress.com, ...
- Virtual pool of memory. One object $\rightarrow$ one server
- $\rightarrow$ another KVS store, **in memory**. Ops: SET/GET/DELETE
  - client based hashing for retrieving object (hash-table)
  - no connection btw servers (no crosstalk/sync./broadcasting)
  - server policies for discarding objects (cache invalidation)




```
function get_foo(foo_id)
    foo = memcached_get(''foo:'' . foo_id)
    return foo if defined foo

    foo = fetch_foo_from_database(foo_id)
    memcached_set(''foo:'' . foo_id, foo)
    return foo
end
```


technicolor

Memcached @ Facebook

- Largest memcached install available: > billion rq/sec, storing trillions of items. AP: eventually consistent system.
- Graph data made persistant in MySQL cluster, but:
- Dominant factor: a data item is likely to be accessed if it has been recently created $\rightarrow$ good use case for a cache!
- A popular page load $\rightarrow$ avg of 521 items fetched from memcached:

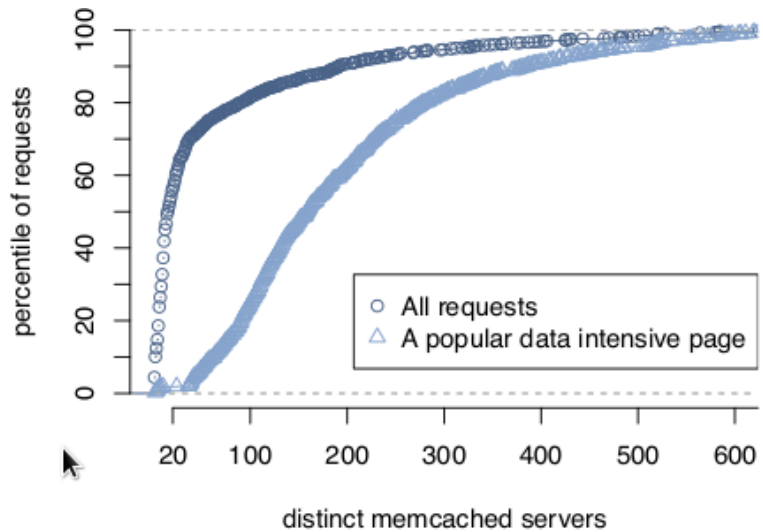


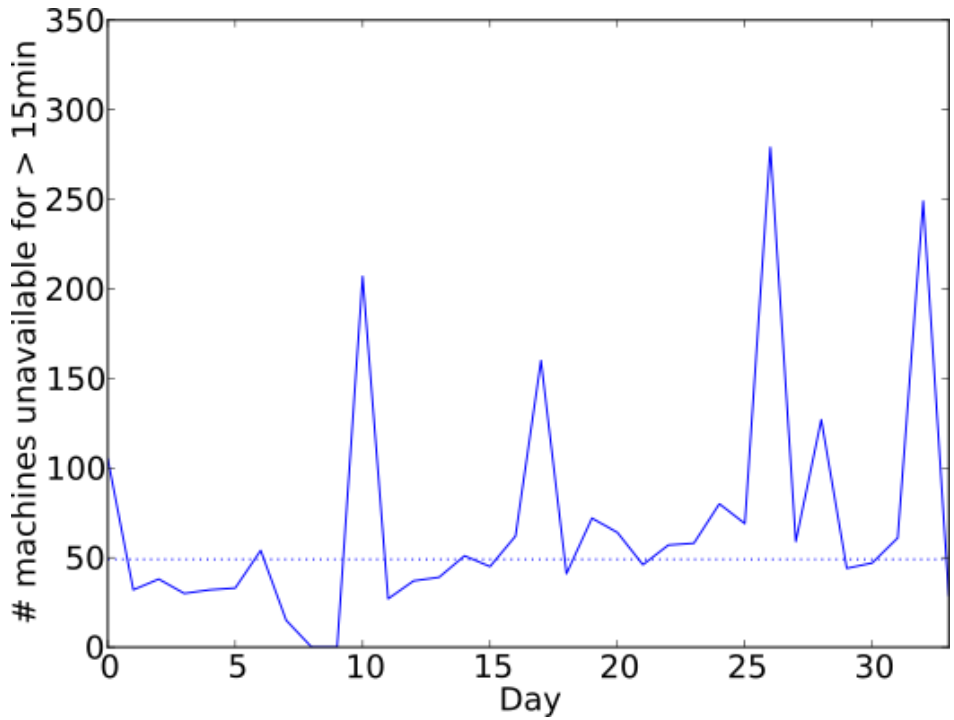
Figure 9: Cumulative distribution of the number of distinct `memcached` servers accessed

0 Scaling Memcache at Facebook, NSDI 2013

technicolor

The way to store data at low level ?

- Failure occurs in practice, e.g. at Facebook
  - 2 clusters for storage: each 100s PB = 10^4 TB
  - Thousands of machines (each 24-36TB)
  - Transient unavailabilities are failures too...



technicolor

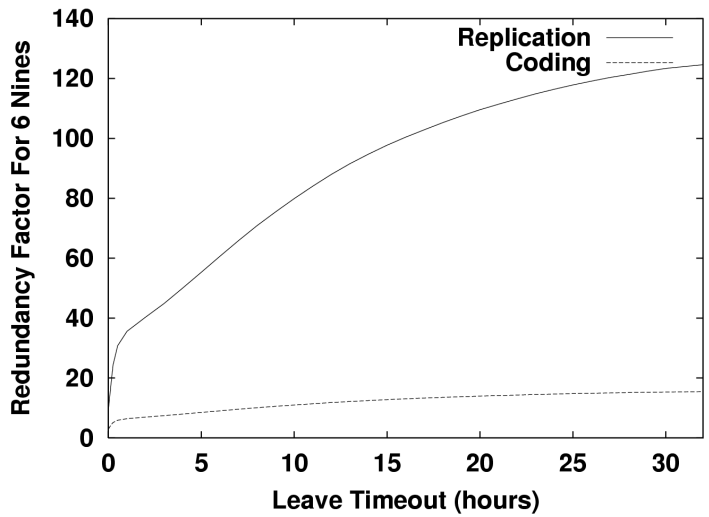
Beyound replication with erasure codes

- Need to “secure” data to prevent loss
- Plain replication (*triplication*) is **very** costly (overhead)

Scheme	Storage overhead	Repair traffic	MTTDL (days)
3-replication	2x	1x	$2.3079E+10$
RS $(10,4)$	0.4x	10x	$3.3118E+13$
LRC $(10,6,5)$	0.6x	5x	$1.2180E+15$

Table 1: Comparison summary of the three schemes. MTTDL assumes independent node failures.

- Based on the fact that *fragmentation increases availability*

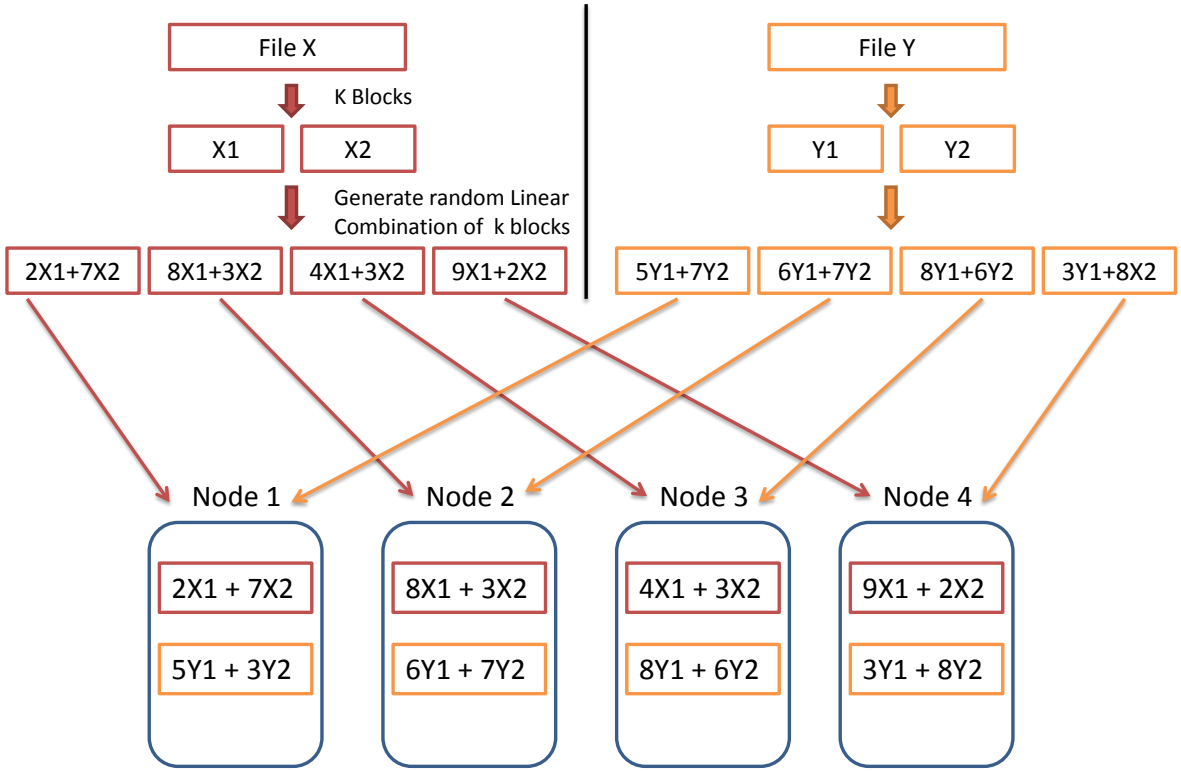


[0] XORing Elephants: Novel Erasure Codes for Big Data, VLDB 2013

technicolor

Example erasure codes: random codes

- chunk file into $k \rightarrow$ make n linear combinations $\rightarrow$ save those

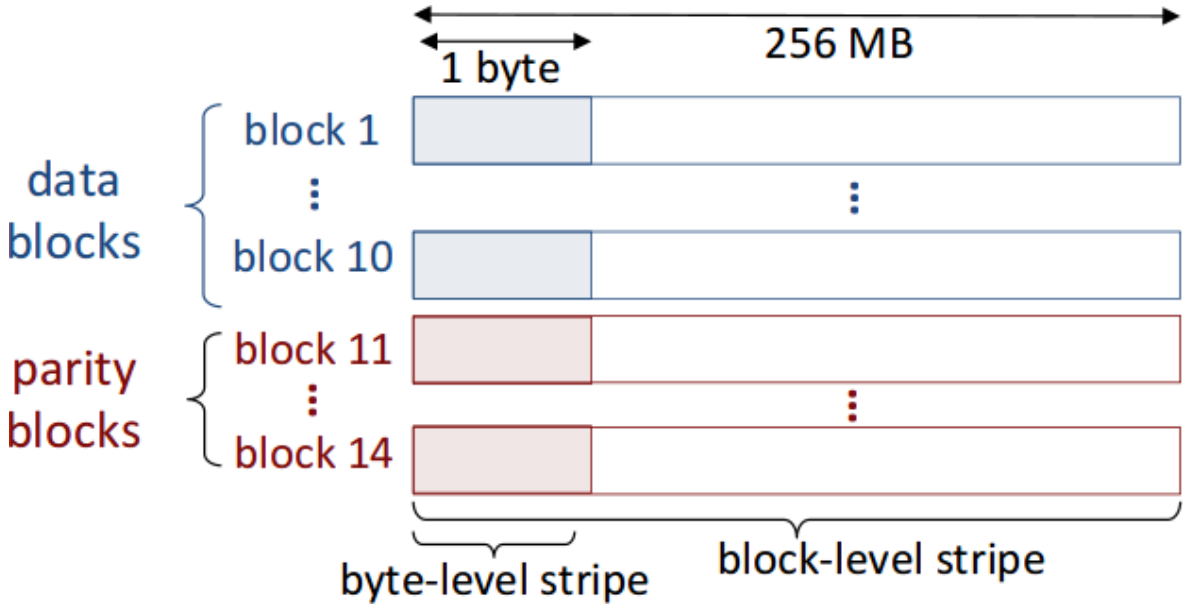


- Restore a file, by decoding **any** k blocks (here 2)
- Optimal in terms of storage/availability
- Very easy, but not efficient for read access

technicolor

Erasure codes: Reed-Solomon codes at Facebook

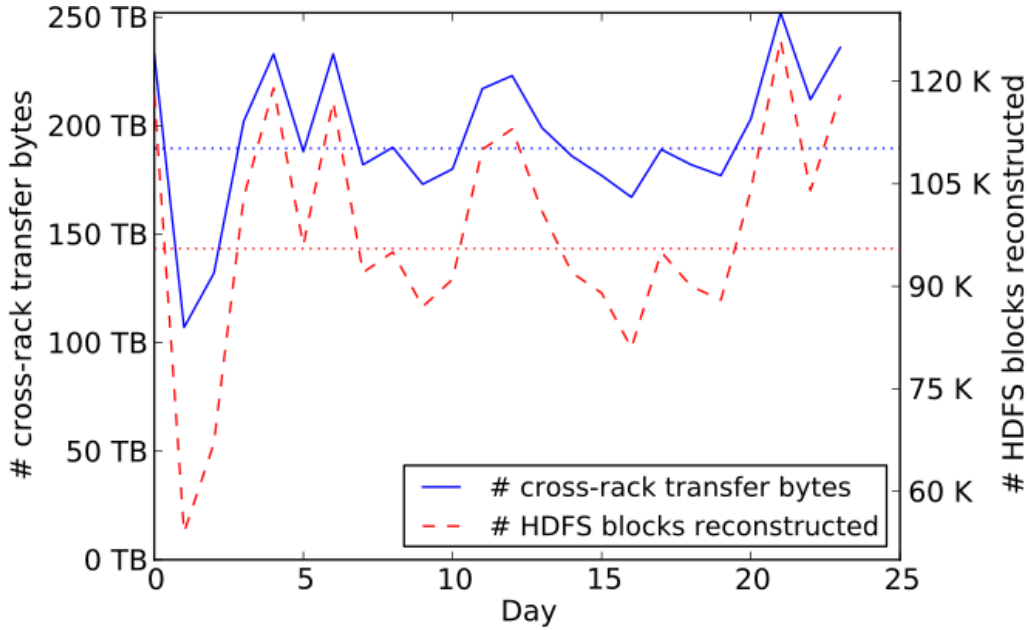
- 2 storage clusters of 100s PB, thousands of machines (each 24-36TB)
- Hot data is triplicated for performance
- After 3 months, un-used data stored with RS coding
- 256MB blocks stored in different racks



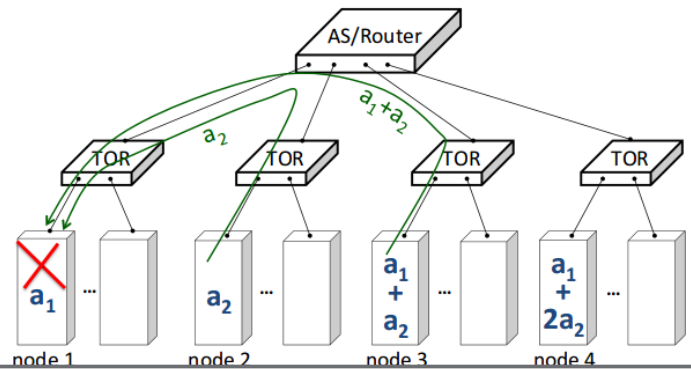
- Tolerates any 4 simultaneous server failures (unavailabilities)

but repair is a real problem in clusters

- Huge number of blocks repaired every day → of data transfered



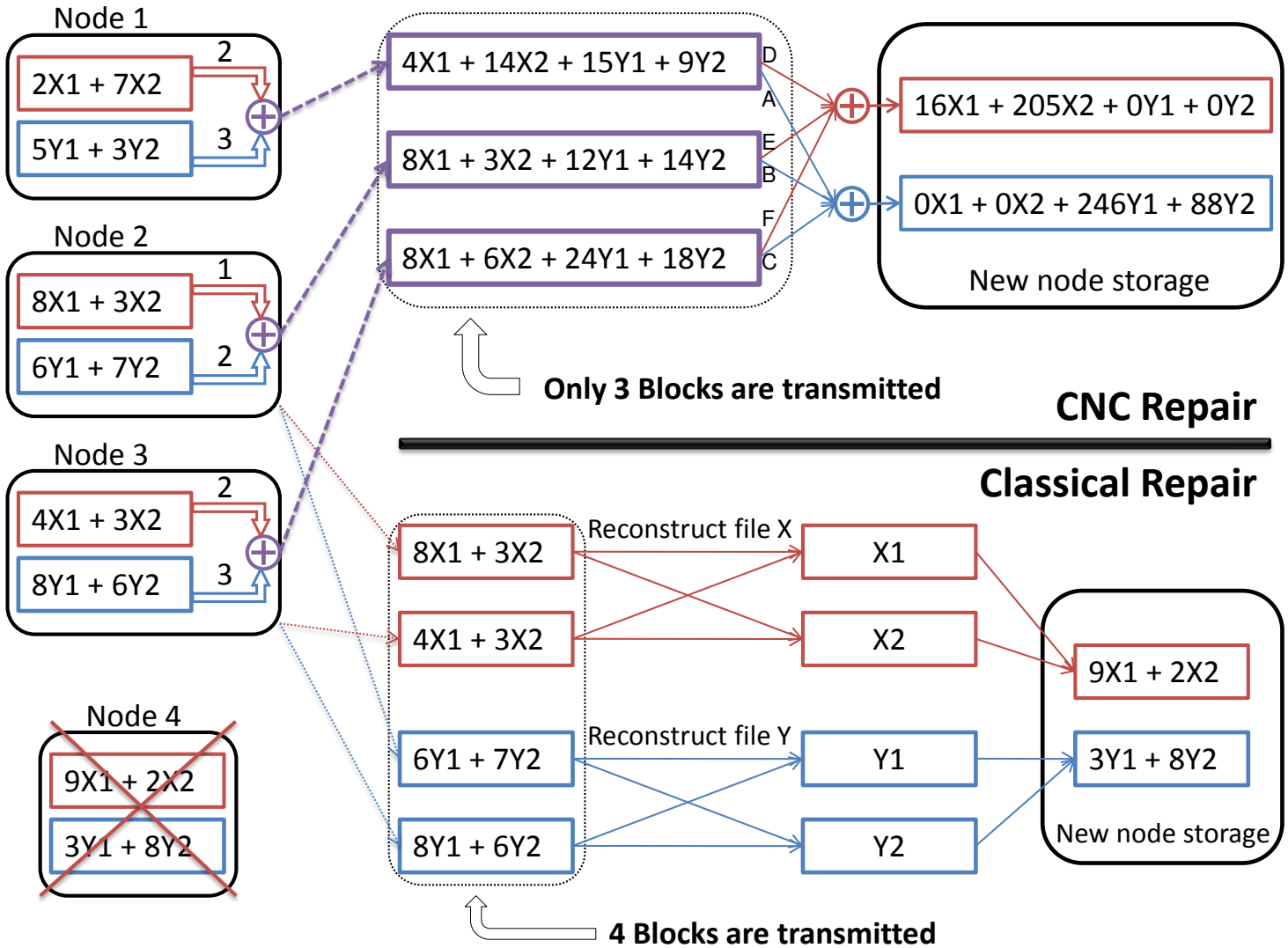
- Bandwidth is critical in datacenters, due to topology



technicolor

Efficient repair of erasure codes

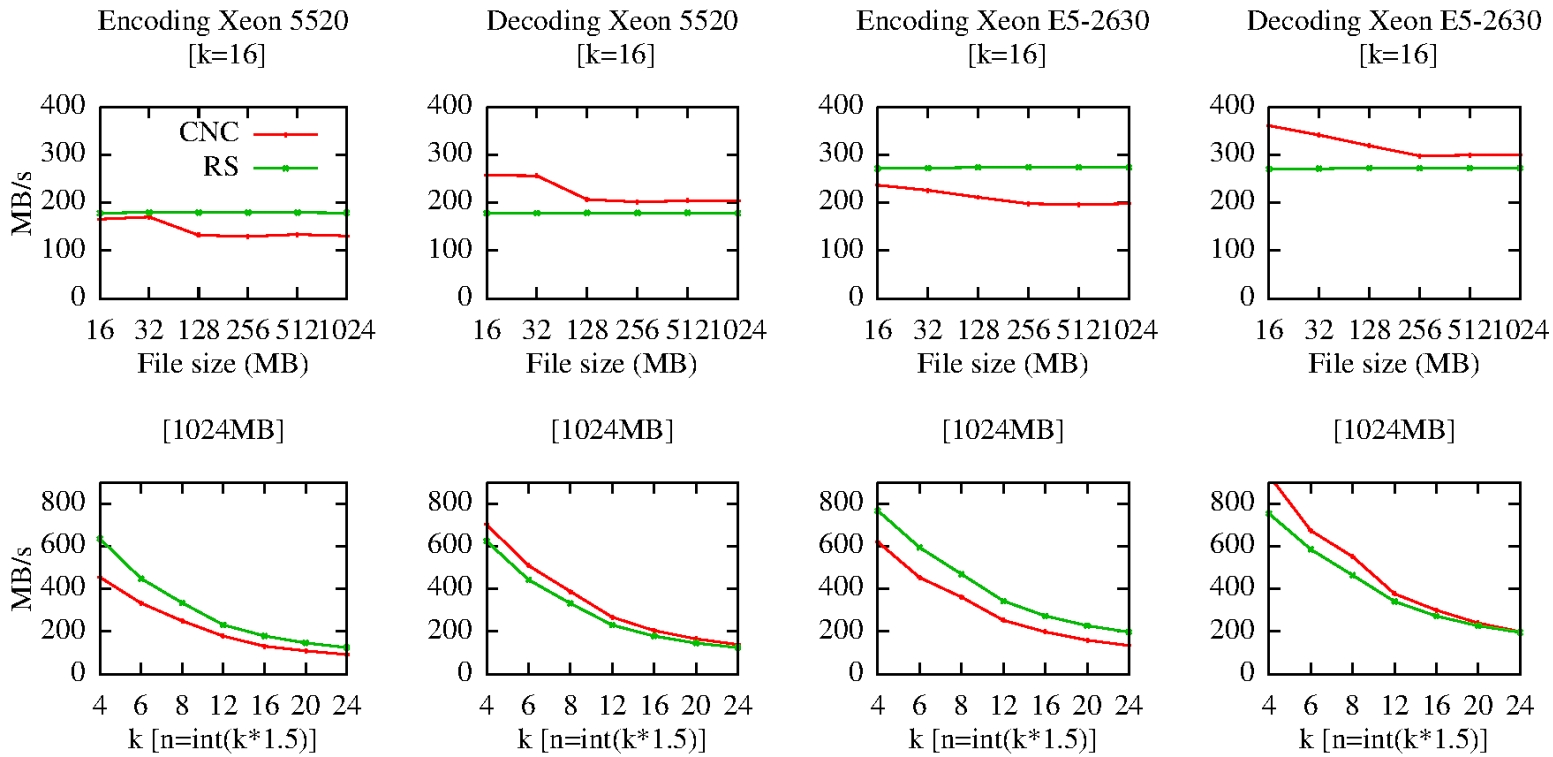
- A solution: transfer **less** blocks upon repair



technicolor

Encoding/decoding performances

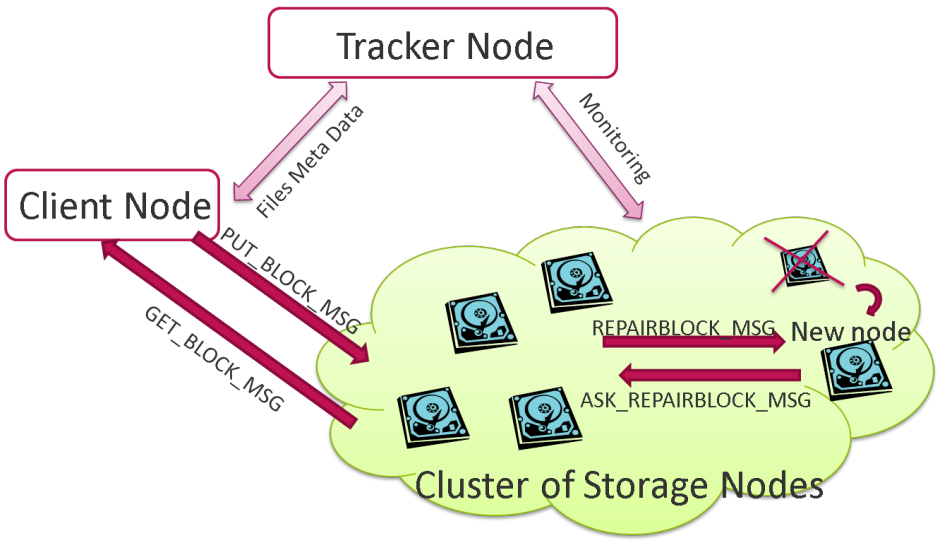
- Similar results for CNC and RS
- Gigabit Network saturated (1 core: can decode > 2Gb/s)

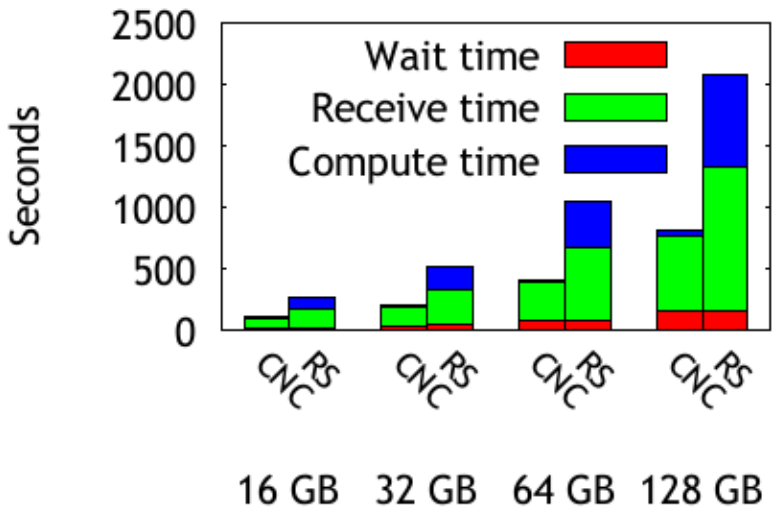


technicolor

Repair time

- Crucial to avoid bursty losses or network congestion
- 60% reduction due to avoiding decoding operations and transferring less data





technicolor

To code or not to code

- It totally depends on the application

	Replication	Reed-Solomon	CNC
Fault tolerance w.r.t storage overhead	✗	✓ (optimal)	
Efficient file access	✓	✓ (applied across files)	✗
Low repair bandwidth	✓	✗ (whole file)	✓ (only half)
Reintegration	✓	✗	✓

Outline

1 Big Data, consequences, and some roadmap

2 Data storage
- NoSQL datastores
- In memory storage
- Efficient storage and repair

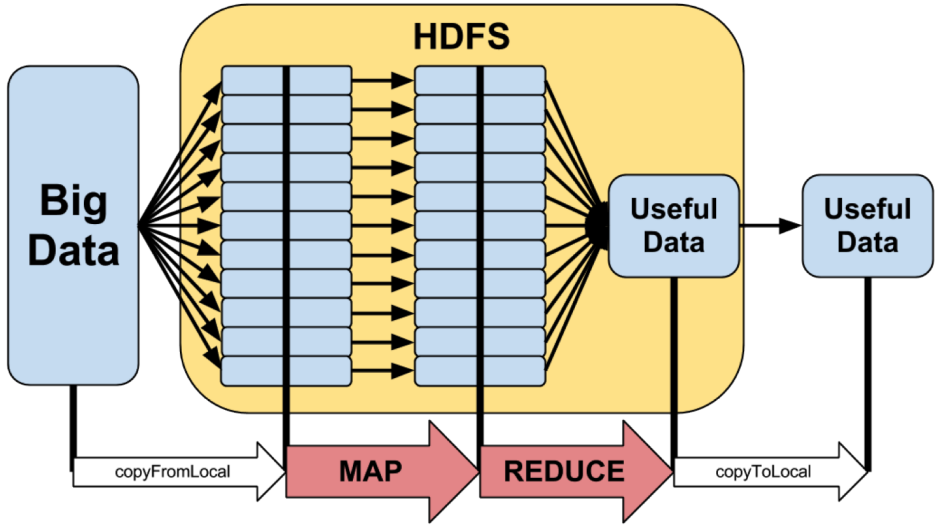
3 Processing
- Batch processing
- Incremental processing
- Stream processing

4 What's next ?
- In software/platform
- In hardware
- In application logic

technicolor

Google's MapReduce

- High Performance Computing here since long... But:
- MR hides parallelization, fault tolerance, data distribution & load balancing behind a simple library for cluster computing



- To compute typically now (has evolved since 2004)
  - Indexes (e.g. at Facebook)
  - grep / sorts, log analysis (see http://wiki.apache.org/hadoop/PoweredBy)
  - NOT ANYMORE: graph operations, machine learning

0 MapReduce: simplified data processing on large clusters, OSDI 2004

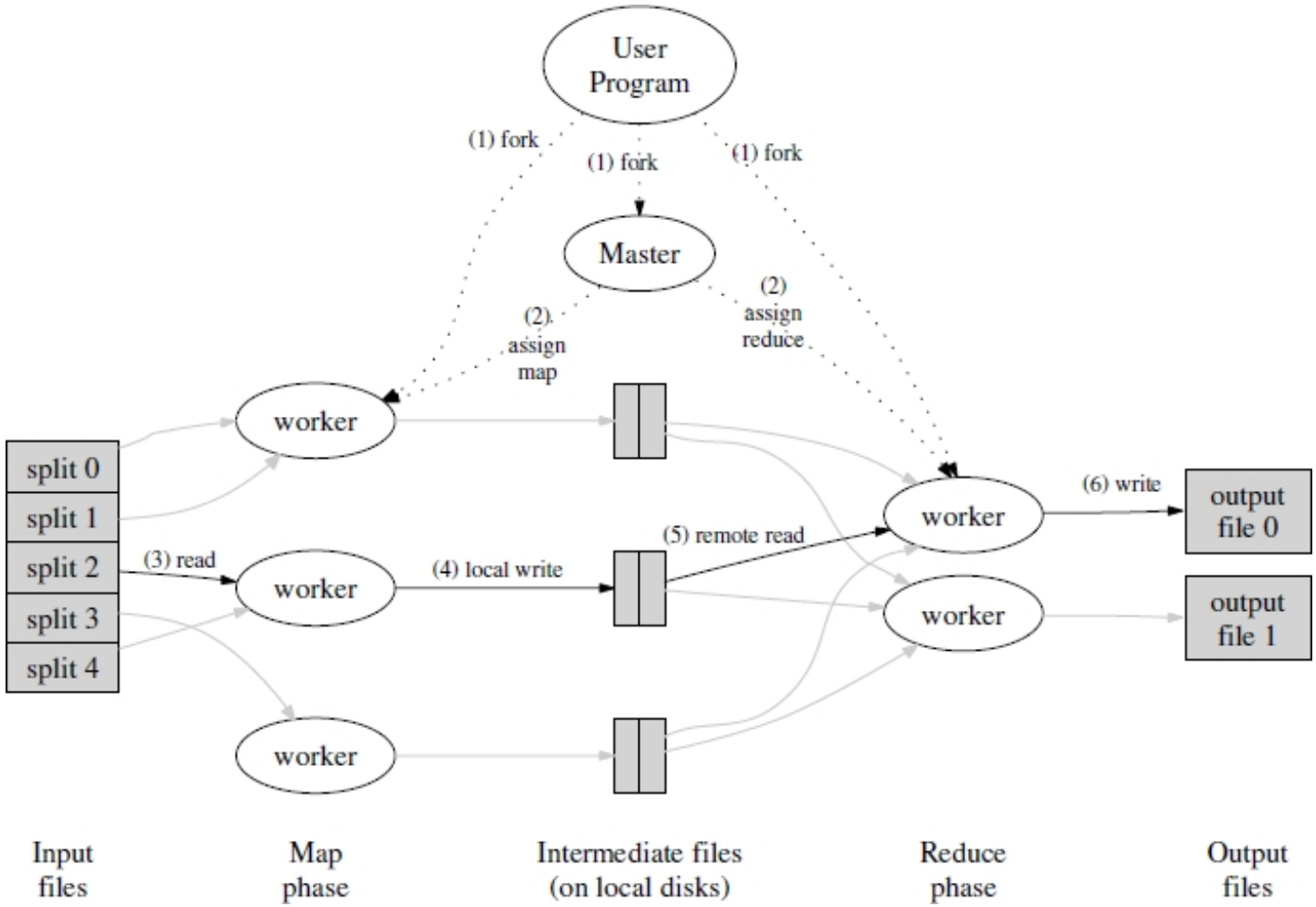
technicolor

MapReduce: programming model

- Input & Output: each a set of key/value pair
- Programmer specifies two functions (cf functional prog.)
  - map (in_key, in_value) $\rightarrow$ list(out_key, intermediate_value)
    - Processes input key/value pair
    - Produces set of intermediate pairs
  - reduce (out_key, list(intermediate_value)) $\rightarrow$ list(out_value)
    - Combines all intermediate values for a particular key
    - Produces a set of merged output values (usually just one)
  - Semantically, map/shuffle phase moves data and reduce processes it

```
map(String key, String value):
    //key: document name
    //value: document contents
    for each word w in value:
        EmitIntermediate(w, ''1'');

reduce(String key, Iterator values):
    //key: a word
    //values: a list of counts
    int result = 0;
    for each v in values:
        result += ParseInt(v);
    Emit(AsString(result));
```

MapReduce Workflow

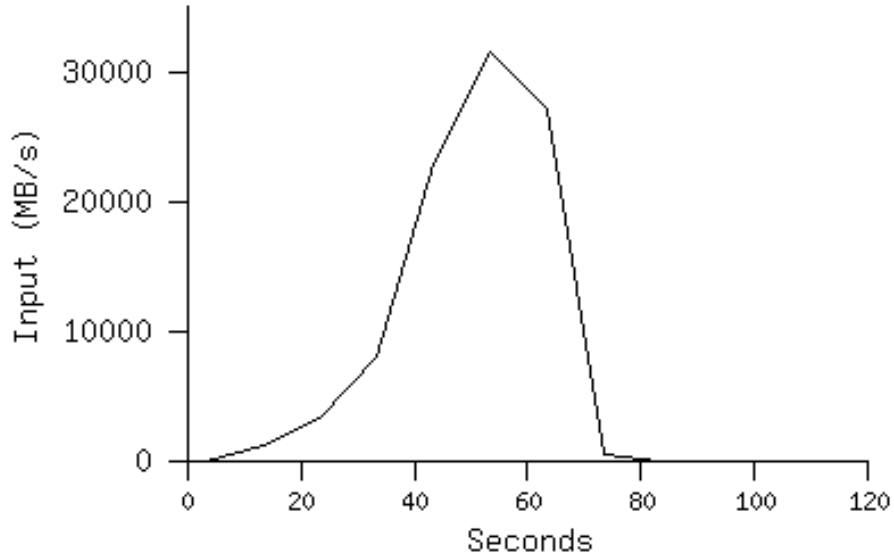


- Possibly many iterations of this workflow

technicolor

MapReduce example performance

- Scan 10^{10} 100-byte records to extract matchs for rare pattern. Record cut into 15000*64MB. 1800 machines (2Ghz Xeons, GB ethernet)

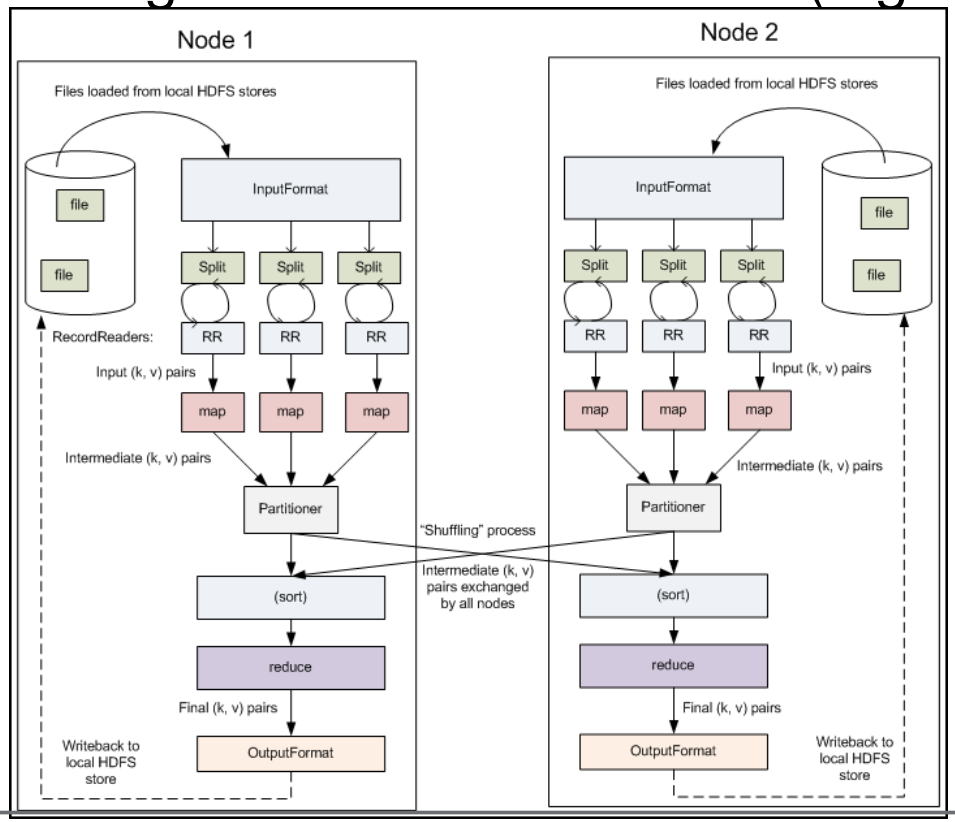


- 1800 machines read 1 TB of data at peak of 31 GB/s, with locality improvements
- Rack switches would limit to 10 GB/s without

technicolor

Hadoop: the free implementation

- Used by 100s: Yahoo! 10^5 CPUS, Facebook PBs of data...
- Hadoop framework in Java
- Higher level ways of interacting: Pig, Hive, Chukwa ...
- Workflow managers as DAGs of actions (e.g. oozie)



technicolor

A sample list of MR compatible applications

Web crawling
- Crawl Blog posts and later process them
- Process documents from a continuous web crawl and distributed training of support vector machines
- Image-based video copyright protection
- Image processing and conversions

Search Indexes
- Parses and indexes mail logs for search

Research on natural language processing and machine learning
- Particle physics, genomics, disease research, astronomy (NASA), etc.
- Crawling, processing, and log analysis
- Store copies of internal log and dimension data sources and use as sources for reporting/analytics and machine learning

Visitor behavior
- Recommender system for behavioral targeting
- Session analysis and report generation
- Analyzing similarities of user's behavior
- Filtering and indexing listing, processing log analysis, for recommendation data
- Storage, log analysis, and pattern analysis
- Logs/Streaming Queues
- Filter and index listings, removing exact duplicates and grouping similar ones

Analyze and index textual information

Image processing
- Image content-based advertising and auto-tagging for social media
- Facial similarity and recognition
- Gathering WWW DNS data to discover content distribution networks and configuration issues

Web log analysis
- Charts calculation and web log analysis
- Process clickstream and demographic data to create web analytic reports
- Research for Ad Systems and Web Search
- Analyze user's actions, click flow, and links
- Process data relating to people on the web
- Aggregate, store, and analyze data related to in-stream viewing of Internet video

Scientific

Data mining
- Build scalable machine learning algorithms like canopy clustering, k-means, Naïve Bayes, etc.

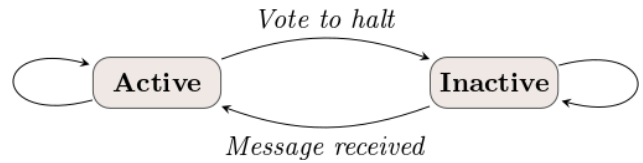
- OKAY for machine learning, but not very efficient as we shall see

0

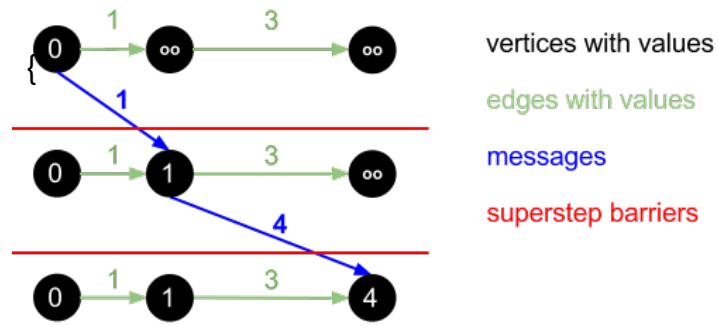
credit: Neil Raden Hired Brains, Inc., 2012

The Bulk Synchronous Model (in a nutshell)

- Generalizing MR, $\neq$ Parallel Random Access Machine model
- Asynch. concurrent computation up to sync barrier: iteration
- e.g. Google (Pregel), Facebook (Giraph) for graph computation
- “think like a vertex” computation type
- Sync barriers at supersteps, state machine for vertices:




```
public void compute(Iterable<DoubleWritable> messages) {
  double minDist = Double.MAX_VALUE;
  for (DoubleWritable message : messages) {
    minDist = Math.min(minDist, message.get());
  }
  if (minDist < getValue().get()) {
    setValue(new DoubleWritable(minDist));
    for (Edge<LongWritable, FloatWritable> edge : getEdges()) {
      double distance = minDist + edge.getValue().get();
      sendMessage(edge.getTargetVertexId(),
                  new DoubleWritable(distance));
    }
  }
  voteToHalt();
}
```

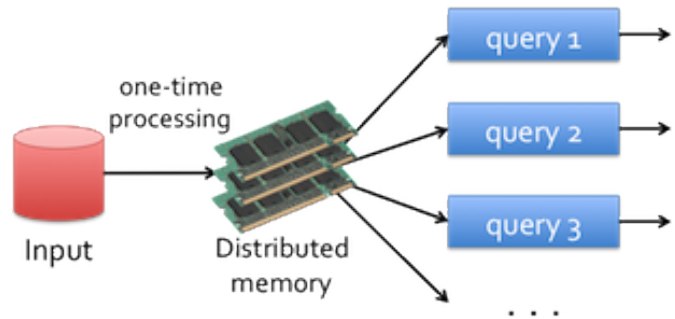


example from http://giraph.apache.org

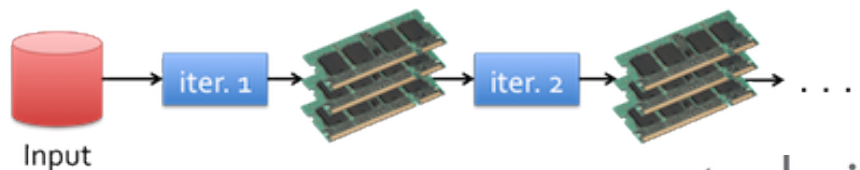
technicolor

Apache's Spark

- Greater flexibility for iterative jobs
  - In-memory computing primitives (no writes on disk to persist data between two iterations as in MapReduce)
  - DAG execution engine that supports complex data flow
- Available platform, program in Java/Scala/Python
- Works with HDFS-enabled systems (HBase...)
- Interactive mode !



(a) Low-latency computations (queries)



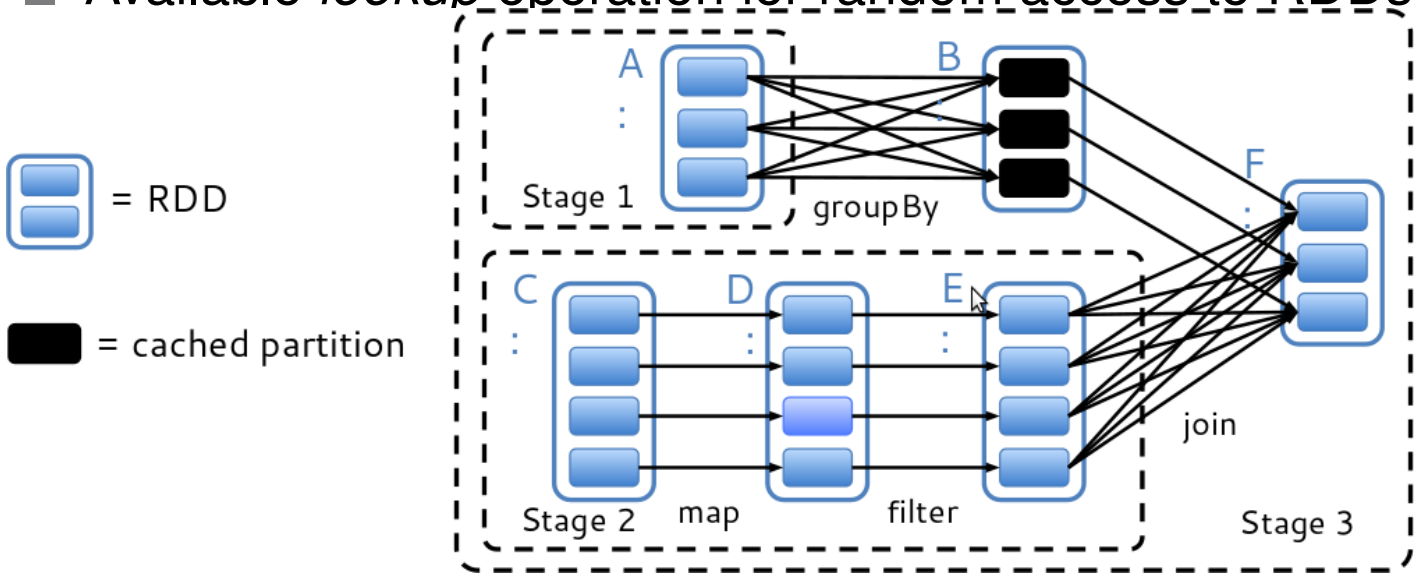
(b) Iterative computations

0 Spark: Cluster Computing with Working Sets, Hotcloud 2010

technicolor

Spark: Resilient Distributed Datasets

- Technical shift: RO partitioned collection of objects in RAM
  - Data does not have to fit on a single machine
  - RDD partition automatically rebuilt upon failure
    - No replication: history of transformations maintained
    - e.g. map $\rightarrow$ filter $\rightarrow$ join
  - Scheduler builds a DAG of stages to execute
  - Scheduler assigns tasks to machines based on data (RDD) locality
  - Available *lookup* operation for random access to RDDs

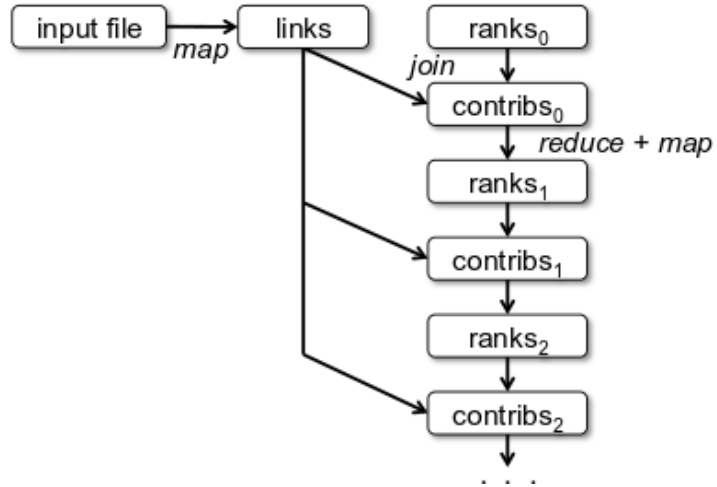


0 Resilient Distributed Datasets ..., NSDI 2012

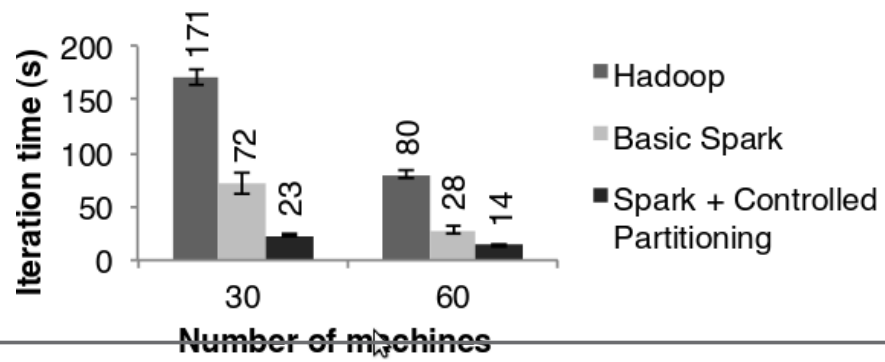
technicolor

Spark: Pagerank example

```
// Load graph as an RDD of (URL, outlinks) pairs
val links = spark.textFile(...).map(...).persist()
var ranks = // RDD of (URL, rank) pairs

for (i <- 1 to ITERATIONS) {
  // Build an RDD of (targetURL, float) pairs
  // with the contributions sent by each page
  val contribs = links.join(ranks).flatMap {
    (url, (links, rank)) =>
      links.map(dest => (dest, rank/links.size))
  }
  // Sum contributions by URL and get new ranks
  ranks = contribs.reduceByKey((x,y) => x+y)
    .mapValues(sum => a/N + (1-a)*sum)
}
```

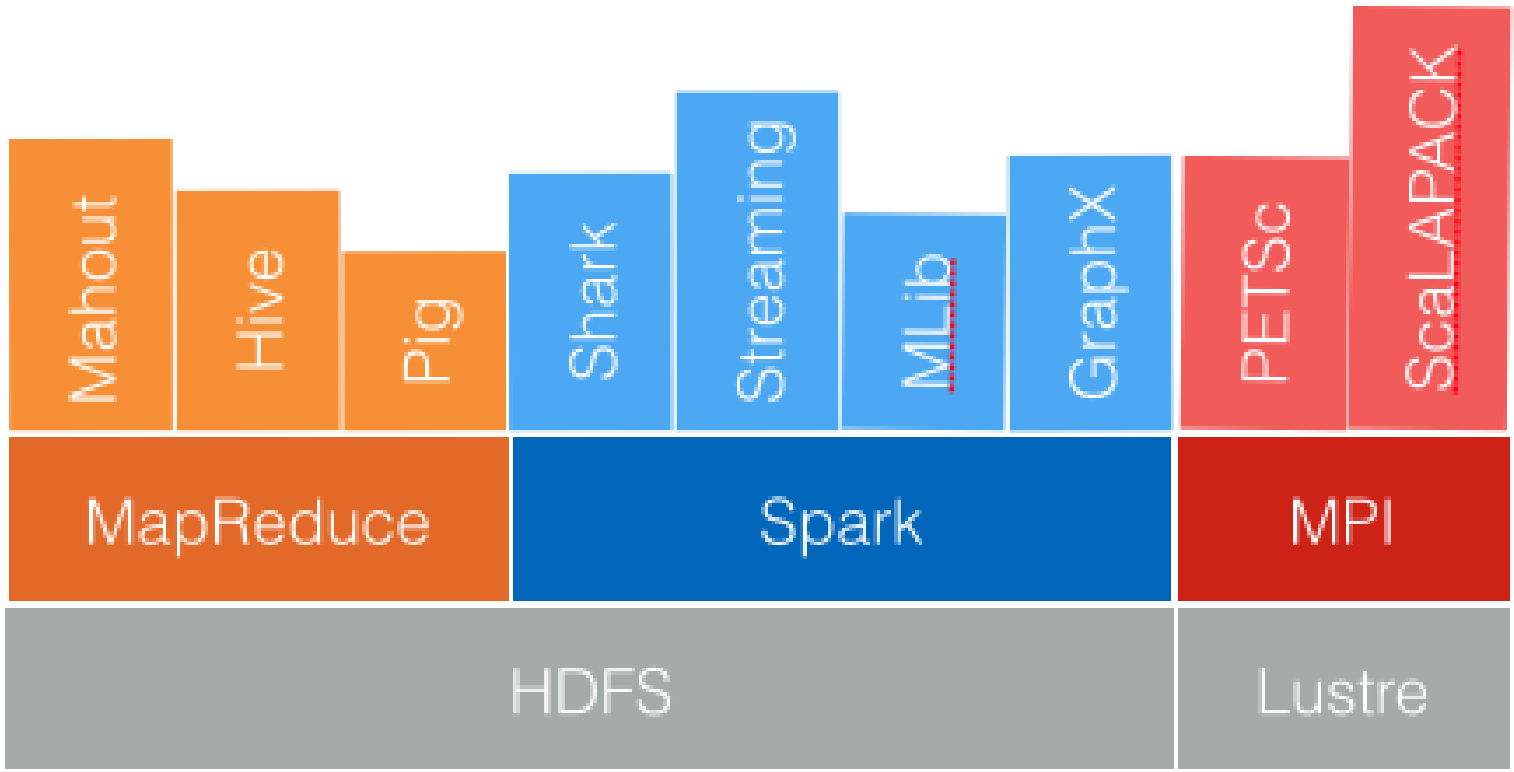


- No required replication: rebuilt from *map* on input file blocks
- Controlled data partitioning: locality for links → less communication for *join* operation



technicolor

Short situation recap & demo

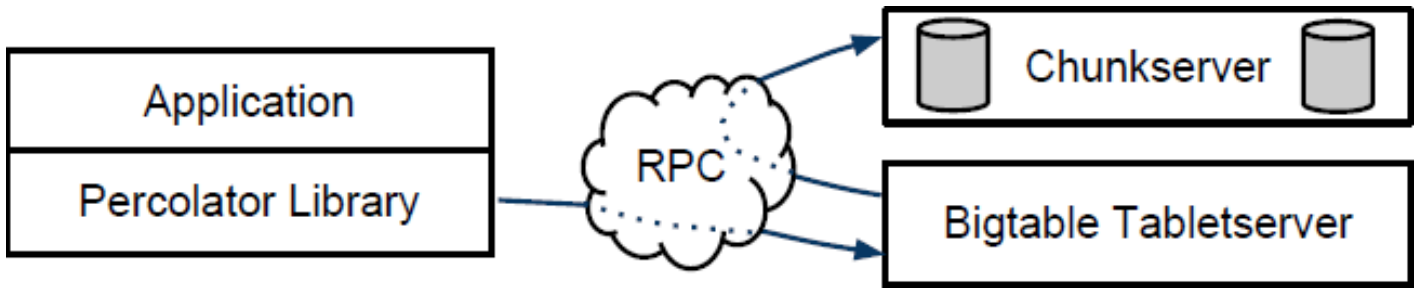


- Quick Demo: HDFS, Hadoop, Spark.

technicolor

Google's Percolator

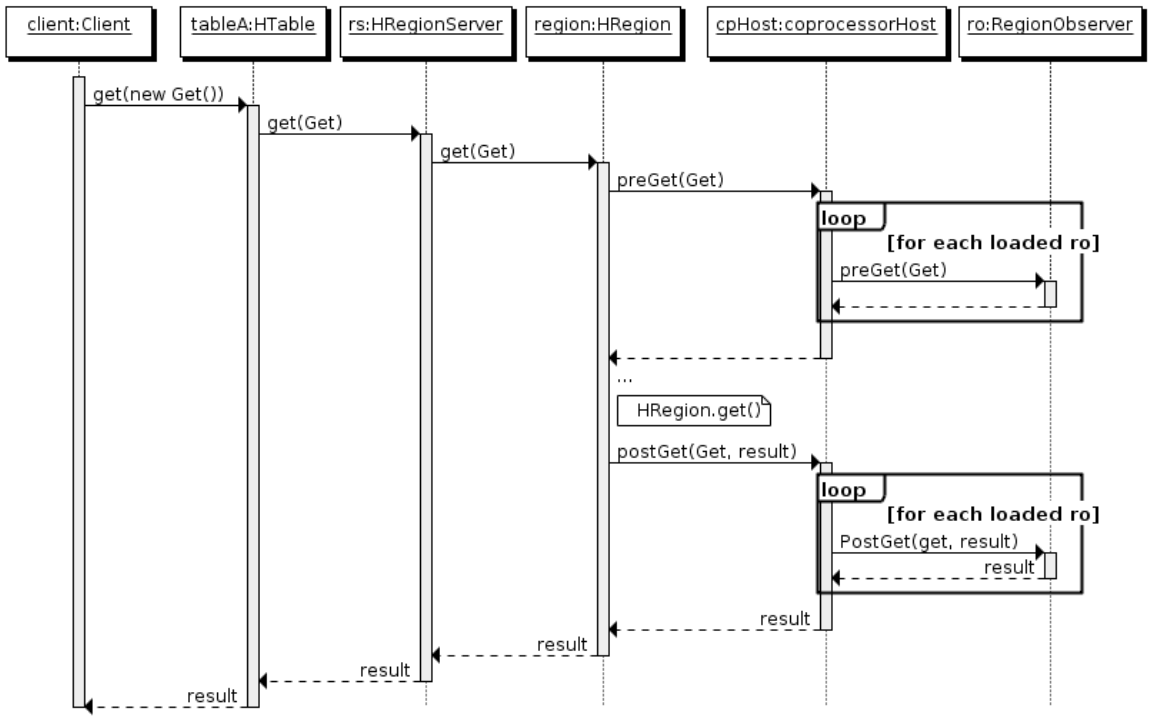
- Moving data from storage to processing cluster is expensive
- If computation in-place OK (e.g. aggregates): incremental actions



- Scan table for changes & invokes observers on Bigtable update
- use case: Google's web search index
  - Update after small re-crawling? Re-MapReduce whole dataset ?
  - Instead: *trigger* when webpage added, and update index
  - Result: insertion in index is 100x faster

Hbase's coprocessors: the free implementation

- callback executed from HBase code when certain events occur
- 3 observer interfaces
  - regionObserver: hooks for data manipulation events, Get, Put, Delete, Scan, ...
  - WALObserver: hooks for write-ahead log operations
  - MasterObserver: hooks for create, delete, modify table, ..



- Example: return a token only if availability counter positive

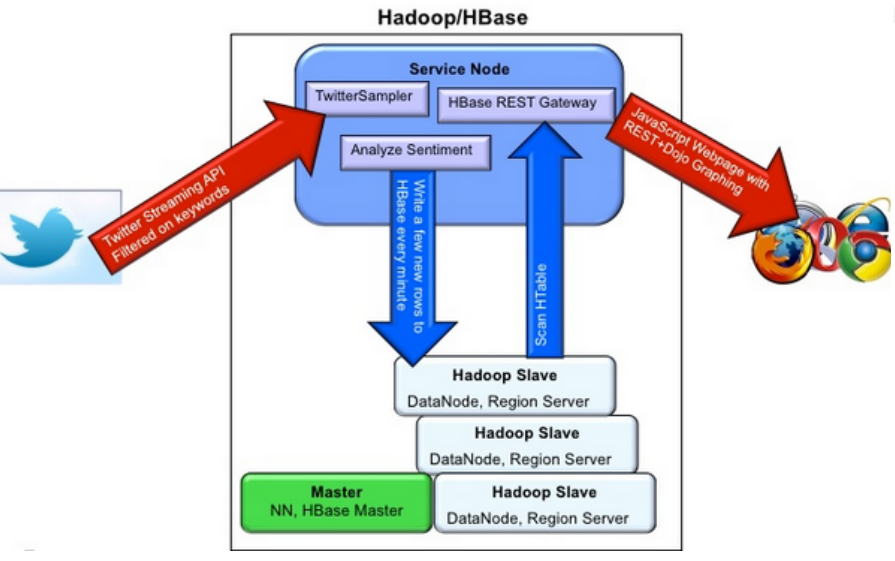
technicolor

Hbase's coprocessors: example of a dynamic RPC

```
// client side
results = table.coprocessorExec(ColumnAggregationProtocol.class, scan,
    new BatchCall<ColumnAggregationProtocol,Long>() {
      public Integer call(ColumnAggregationProtocol instance)throws IOException{
        return instance.sum(TEST_FAMILY, TEST_QUALIFIER);
      }
    });

// server side: protocol for performing aggregation at regions.
public static class ColumnAggregationEndpoint extends BaseEndpointCoprocessor
implements ColumnAggregationProtocol {
  @Override
  public long sum(byte[] family, byte[] qualifier)
  throws IOException {
    // aggregate at each region
    Scan scan = new Scan();
    scan.addColumn(family, qualifier);
    long sumResult = 0;
    InternalScanner scanner = getEnvironment().getRegion().getScanner(scan);
    try {
      List<KeyValue> curVals = new ArrayList<KeyValue>();
      boolean hasMore = false;
      do {
    curVals.clear();
    hasMore = scanner.next(curVals);
    KeyValue kv = curVals.get(0);
    sumResult += Bytes.toLong(kv.getValue());
      } while (hasMore);
    } finally {scanner.close();}
    return sumResult;
  }
}
```

HBase example: real time sentiment analysis

- Trig computation based on recent Tweets, on insertion into HBase




```
private void writeToHBase() {
        Calendar cal = Calendar.getInstance();
        String calStr = String.format("%04d", (cal.get(Calendar.YEAR)))
                + ":" + String.format("%02d", cal.get(Calendar.MONTH) + 1)
                + ":" + String.format("%02d", cal.get(Calendar.DAY_OF_MONTH))
                + ":" + String.format("%02d", cal.get(Calendar.HOUR_OF_DAY))
                + ":" + String.format("%02d", cal.get(Calendar.MINUTE));
        String rowKey = keyword + ":" + calStr;
        Put put = new Put(rowKey.getBytes());
        put.add(COLFAM1.getBytes(), "NEUTRAL".getBytes(), tracker.getNeutralCount().getBytes());
        put.add(COLFAM1.getBytes(), "POSITIVE".getBytes(), tracker.getPositiveCount().getBytes());
        put.add(COLFAM1.getBytes(), "NEGATIVE".getBytes(), tracker.getNegativeCount().getBytes());
        try {
                table.put(put);
        } catch (Exception ex) {
```

Row	NEUTRAL	POSITIVE	NEGATIVE
obama:2012:06:04:13:34	1	4	0
romney:2012:06:04:13:34	2	3	1
davebarry:2012:06:04:13:34	0	9	0

0

credit: Jagane Sundar, Hadoop Summit 2012

technicolor

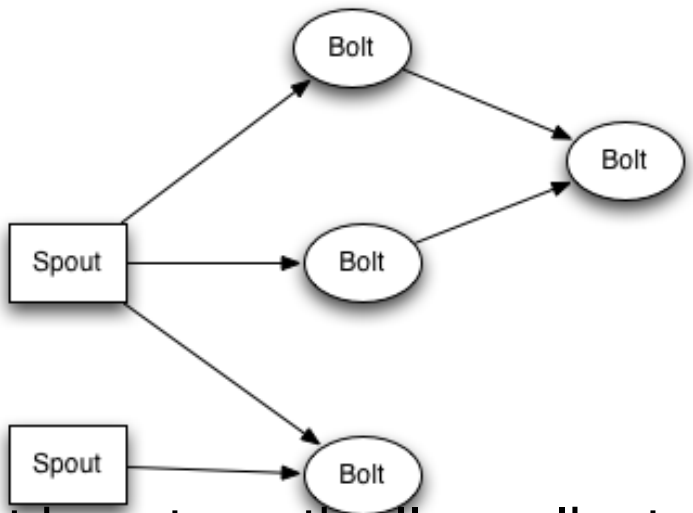
Stream processing: on the fly computation

- Sometimes data has value mostly at reception time
- When one knows *exactly* what to process
- and that the computation *can be* immediate / incremental
- When an event arrives at the datacenter:

filter $\rightarrow$ process $\rightarrow$ update $\rightarrow$ trash event

The Storm platform

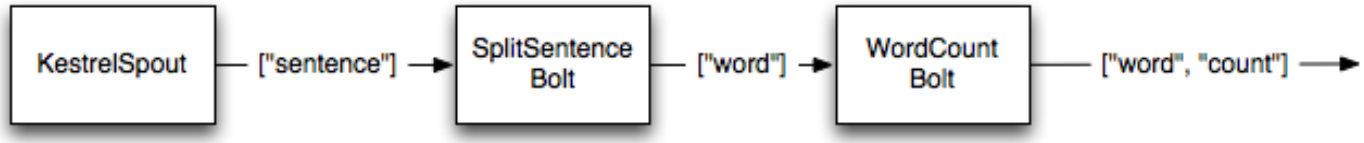
- Open source platform (Java), used by Twitter and many others
- Event=tuple: named list of values
- 2 building blocks for application logic (=instances)
  - Spouts: sources of streams
    - e.g. connected to the Twitter API to emit tweets
  - Bolts: units of computation
    - e.g. filter tweets based on keywords / identify trends
- Design a topology that encodes the system logic



- Each component is automatically replicated for scaling

technicolor

Storm example



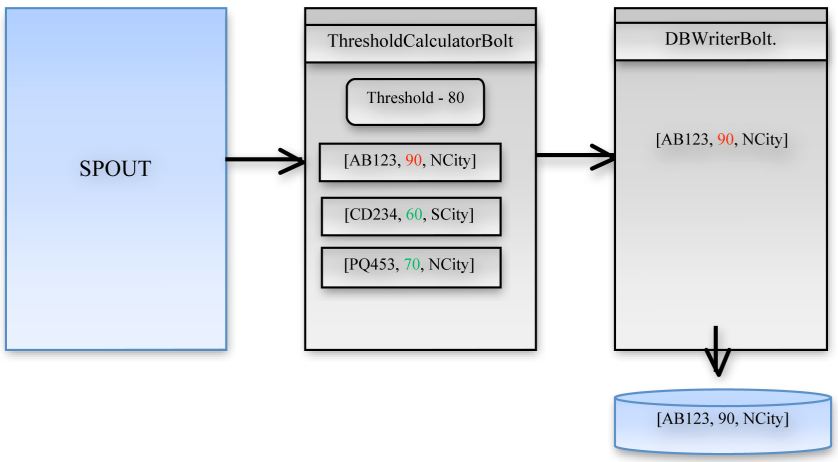

```
// build the topologyy
TopologyBuilder builder = new TopologyBuilder();
builder.setSpout(''spout'', new RandomSentenceSpout(), 5);
builder.setBolt(''split'', new SplitSentence(), 8).shuffleGrouping(''spout'');
builder.setBolt(''count'', new WordCount(), 12).fieldsGrouping(''split'', new Fields(''word''));

// split arriving sentences, e.g. in python
class SplitSentenceBolt(storm.BasicBolt):
    def process(self, tup):
        words = tup.values[0].split('' '')
        for word in words:
          storm.emit([word])

// finally, count & emit
    public static class WordCount extends BaseBasicBolt {
        Map counts = new HashMap();
        public void execute(Tuple tuple, BasicOutputCollector collector) {
            String word = tuple.getString(0);
            Integer count = counts.get(word);
            if(count==null) count = 0;
            count++;
            counts.put(word, count);
            collector.emit(new Values(word, count));
        }
        public void declareOutputFields(OutputFieldsDeclarer declarer) {
            declarer.declare(new Fields(''word'', ''count''));
        }
    }
```


technicolor

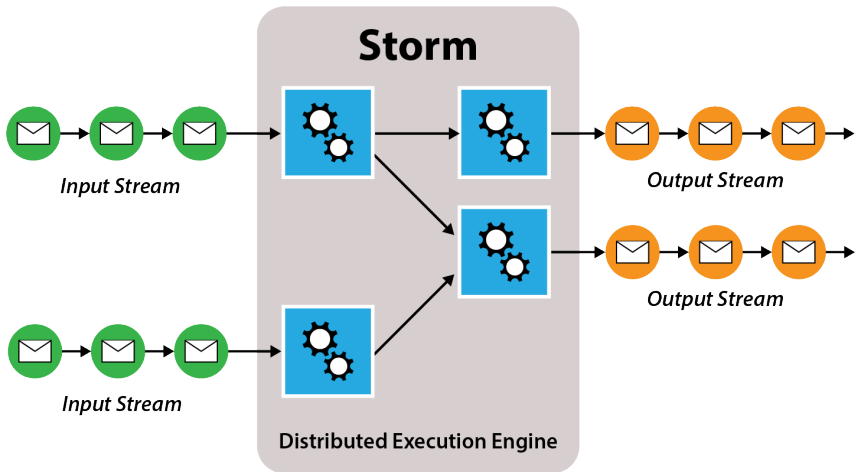
Storm example use case

- Real time analytics for the Internet of Things
- Real time processing intermediate layer

Speed tracking/alert for your fancy car :(



Online censorship :(



0

credits: Dr.Dobb's, ScaleOut Software

technicolor

The stream processing (more formal) model

- An example stream model: the *cash register* model
  - consider a n-dimensional vector $\mathbf{a}$
  - $a_i(0) = 0, \forall i = 1, ..n$
  - at time t: $a(t) = [a_1(t), ...a_i(t), ...a_n(t)]$
  - the tth update is (i_t, c_t), meaning that

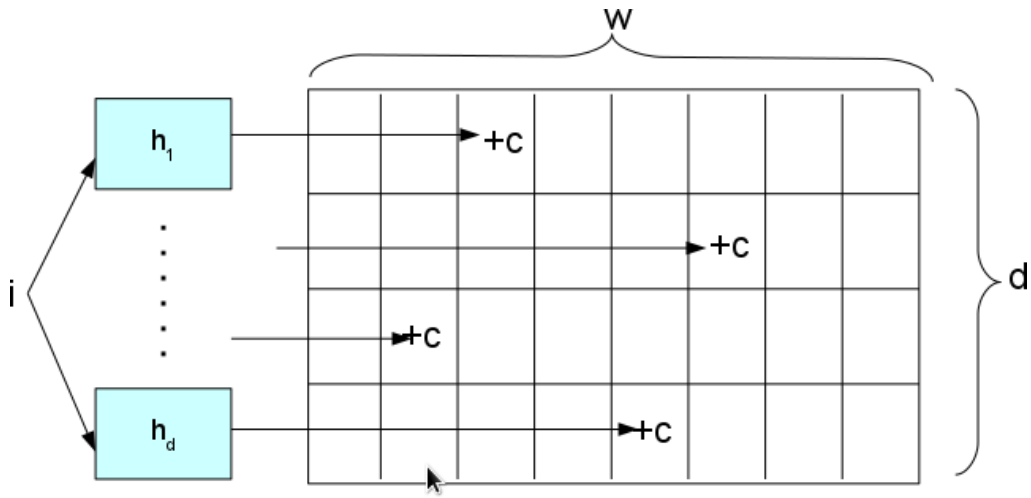
$$a_{i_t}(t) = a_{i_t}(t-1) + c_t$$

- Encodes the incremental nature of stream processing, plus small memory footprint

0 An Improved Data Stream Summary: The Count-Min Sketch and its Applications, J. of Algorithms 2005

Stream app.: an algorithm for finding heavy hitters

- Count-min sketch: 2-D array, probabilistic data structure
- Error in ϵ with probability β
- $d = [\ln 1/\beta]$ (depth), $w = [e/\epsilon]$ (width)



CM sketch update

- Assuming d hash functions (chosen uniformly/independently)
  - On event i, update a_{i_t} with (i_t, c_t), $\forall 1 \leqslant j \leqslant d$

$$count[j, h_j(i_t)] \leftarrow count[j, h_j(i_t)] + c_t$$

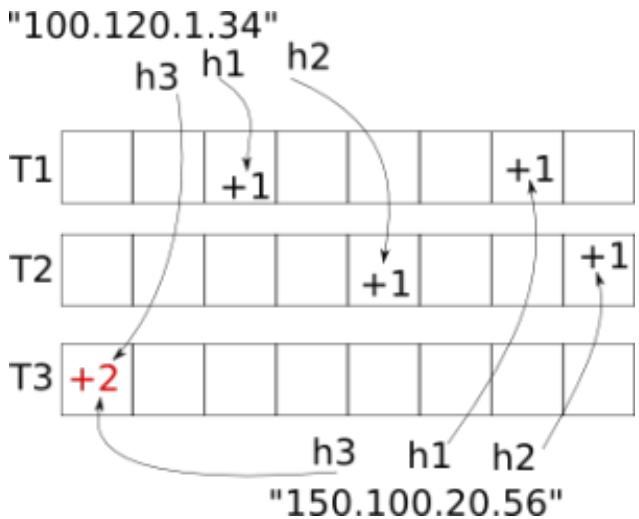
technicolor

Stream app.: an algorithm for finding heavy hitters (2)

- The ϕ-heavy hitters in **a**: $\{i : a_i > \phi \|a\|_1\}$, w. $\|a\|_1 = \Sigma_{i=1}^{n} a_i(t)$
- Approximate heavy hitters: $\{a_i \geqslant (\phi - \epsilon)\|a\|_1\}$, w. chosen $\epsilon < \phi$

AHH set $\leftarrow \hat{a}_i$ s.t. $\hat{a}_i = \min_j count[j, h_j(i)] > \phi\|a\|_1$ in the sketch

- (not complete algorithm, but you get the point)
- e.g. counting IP packets from sources, and finding heavy users

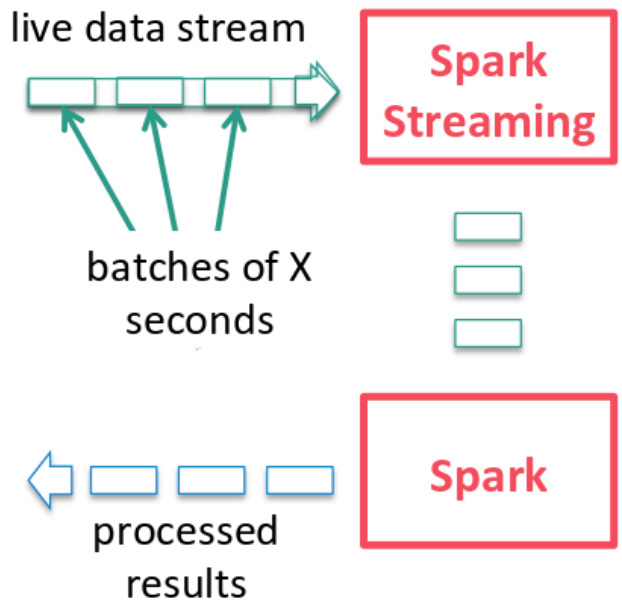


technicolor

Back to Spark with Spark streaming

Run a streaming computation as a series of very small, deterministic batch jobs

- Chop up the live stream into batches of X seconds
- Spark treats each batch of data as RDDs and processes them using RDD operations
- Finally, the processed results of the RDD operations are returned in batches

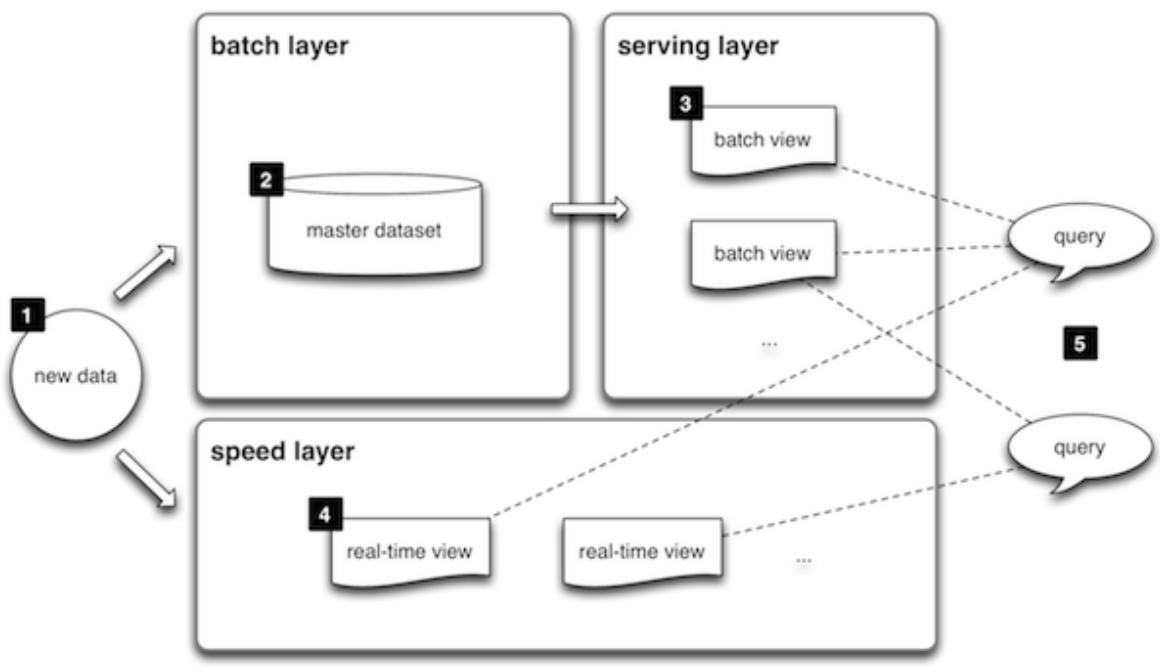


- thus “streaming”, but using windows of events, not single events

technicolor

Lambda architecture: batch + stream

- Of course, to get best of both worlds, combine them



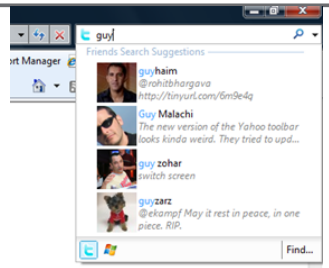
$\rightarrow$ fault tolerance, and low latency when needed (and possible)

- Perfect way for having
  - having fast analytics to display trends, and
  - more complex data-intensive analytics

0 credit: Nathan Marz, 2014

technicolor

A service architecture: Twitter's query suggestion

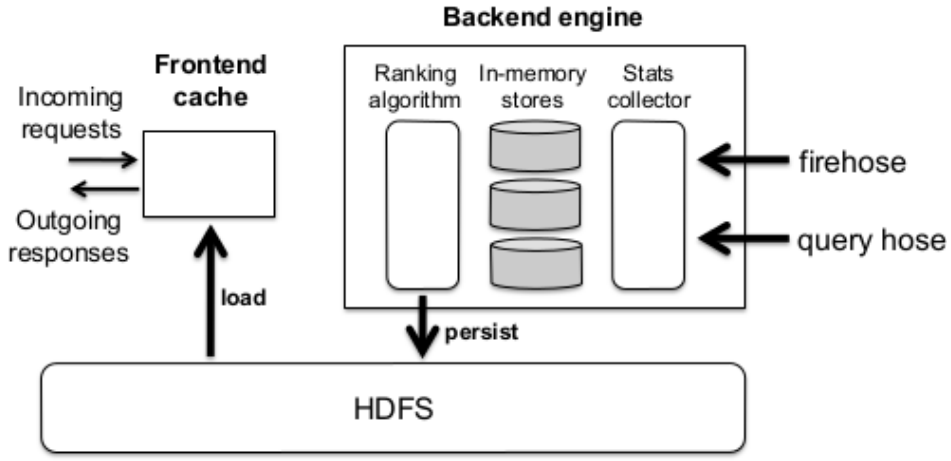


- Service: query suggestion (instant search suggestion)
- Idea: if queries A & B appear in same context $\rightarrow$ related. Recommend accordingly
- What workflow/architecture HAS failed for computation
  - Gather/aggregate logs (client events, 1TB compressed/day)
  - Move sanitized logs to the data warehouse
    - Per hour dir. (e.g. /**logs**/**cat**/**YYY**/**MM**/**DD**/**HH**/) (small # large files)
  - Start workflow manager (Pig/Hadoop jobs) on each dir.
- Problems
  - Lag btw log generation and movement to data warehouse (hours!)
  - 15-20 mn for Hadoop to process 1 hour of log data
    - Burden of the compute cluster: 10,000s jobs per day
  - Hadoop stragglers problem (mean running time $<<$ max)
- Best case: 10s minute latencies from end to end $\rightarrow$ redesign!

0 ... Twitter's real-time related query suggestion architecture, SIGMOD 2013

technicolor

A service architecture: Twitter's query suggestion (2)

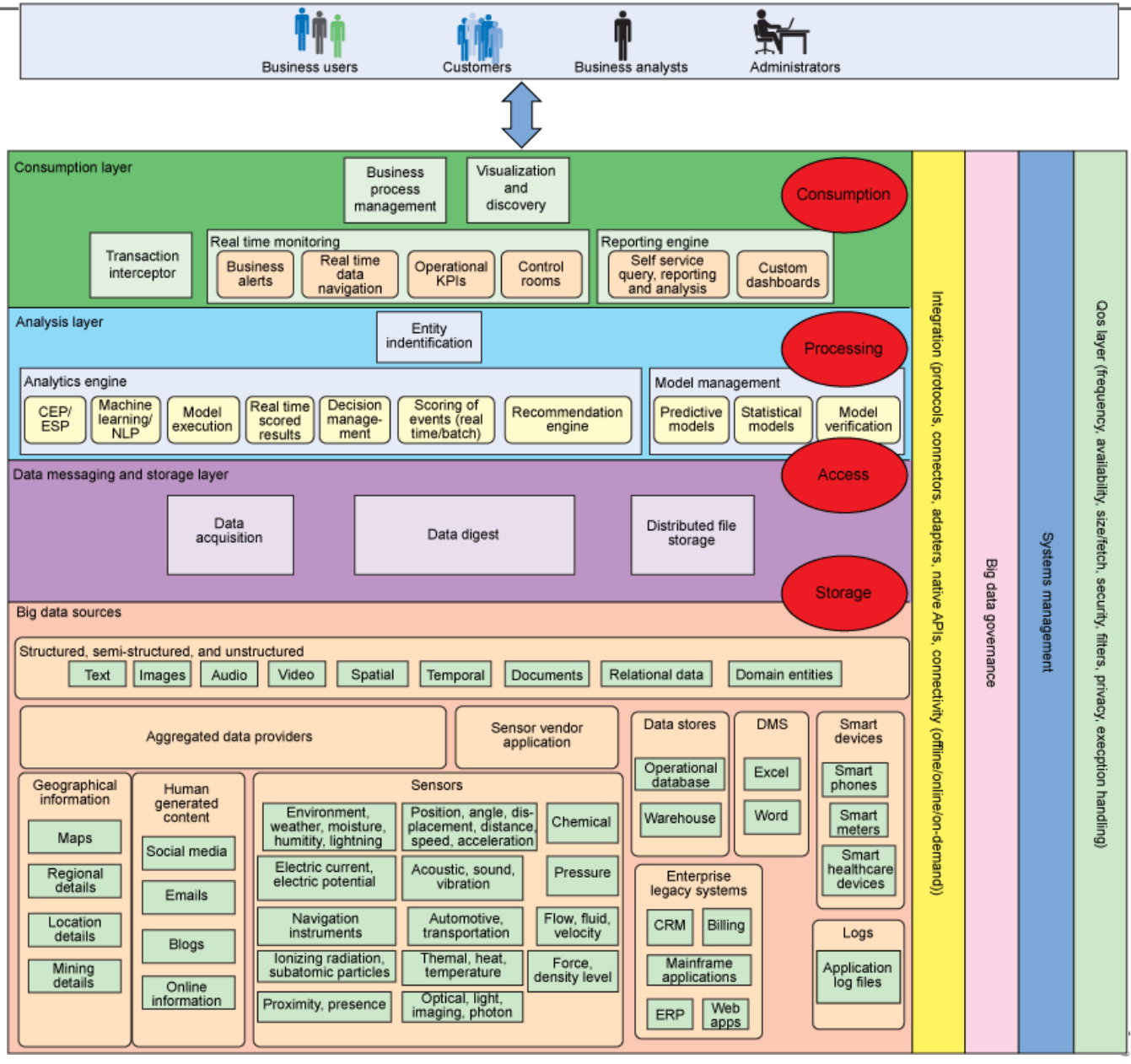


- New architecture components
  - Hoses: stream of Tweets and queries
  - In memory stores: counts, statistics, query cooccurrences
  - Ranking: sorts most relevant coocurrences using stores
  - Frontend: exposed to clients, scales out with increasing load
- HDFS persists computations ($\forall$ 5mn). Retrospective analysis OK
- Fast analytics on pairs of coocurrences (decaying data in sparse struct.)

No general guidelines, application tailored design!

technicolor

Wrapping-up



credit: IBM, 2013

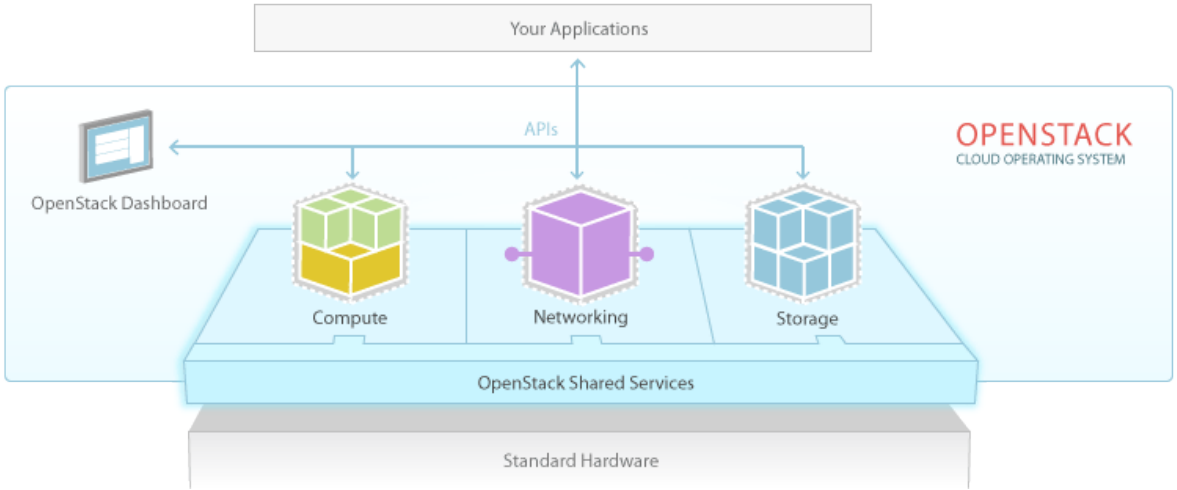
technicolor

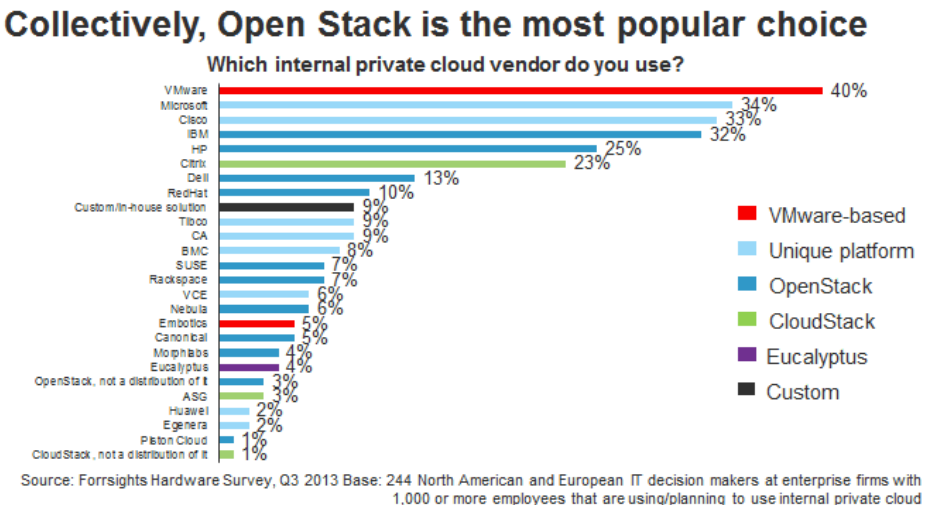
Outline

1. **Big Data, consequences, and some roadmap**
2. **Data storage**
   - NoSQL datastores
   - In memory storage
   - Efficient storage and repair
3. **Processing**
   - Batch processing
   - Incremental processing
   - Stream processing
4. **What's next ?**
   - In software/platform
   - In hardware
   - In application logic

technicolor

The OpenStack initiative





- "open source cloud OS" (Apache 2.0 license)
- Founded by Rackspace Hosting and NASA, 18000+ contributors
- ODISEA pushed for erasure codes in the storage layer

technicolor

Towards rackscale computing

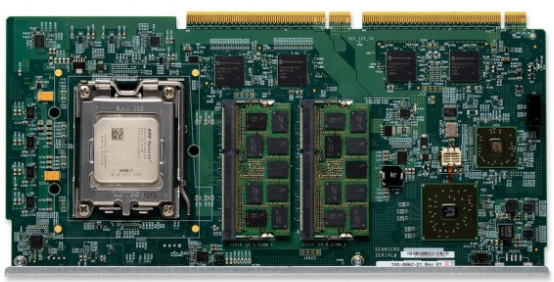
- Commodity hardware is now the norm unit for processing
- Then more power = more machines
- To boost datacenter on demand, few years ago, idea was:

- Apparently did not quite work
- so, what is a cool new unit of processing for better performance ?

Towards rackscale computing

- The rack is the new unit of deployment?
- In between a server and a cluster

	Today's rack	2020 Rack-scale Computer	2014 Rack-scale Computer
#Cores (#servers)	~100s (20-40)	~100,000s (1,000s)	*~1,000s (100s-1000s)*
Memory	~1 TB	~100s TB	*~10 TBs*
Storage	~100 TB (SSD/HDD)	~100s PB (NVM)	*~10s PBs (SSD/HDD)*
Network	10 Gbps / server	1 Tbps / server	*~10s -100s Gbps / server*



0 Towards Rack scale Computing Challenges and Opportunities, by P. Costa, 2014

Rackscale: hardware characteristics

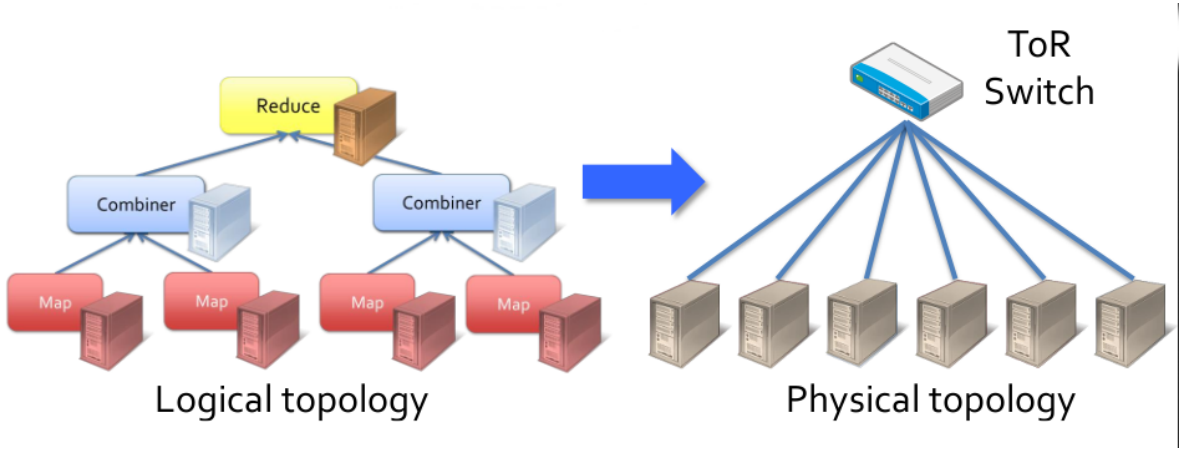
System-on-chip (SoC) integration combines cores, caches and network interfaces in a single die. SoCs are widespread on mobile platforms as they save power and space, and the same advantages also apply in the server domain. SoCs enable vendors to build micro-servers: extremely small server boards containing computation, memory, network interfaces, and sometimes flash storage. For instance, the Calxeda ECX-1000 SoC hosts four ARM cores, a memory controller, a SATA interface, and a fabric switch onto a single die.

Fabric integration connects such micro-servers into a high-bandwidth low-latency network. Typically this is done by using a “distributed switch” architecture; micro-servers are connected via point-to-point links into a multi-hop direct-connect topology. The point-to-point links can simply be traces on a PCB back-plane and can run custom physical and link protocols that are not exposed to the micro-servers. They can thus offer high bandwidth (10-100 Gbps) and low per-hop latency (100- 500 ns). A “fabric controller” on each node provides a NIC interface to the micro-server and also forwards packets for other micro-servers. Any topology that has a small number of links per node can be used; 2D and 3D torus are popular choices adapted from supercomputing architectures.

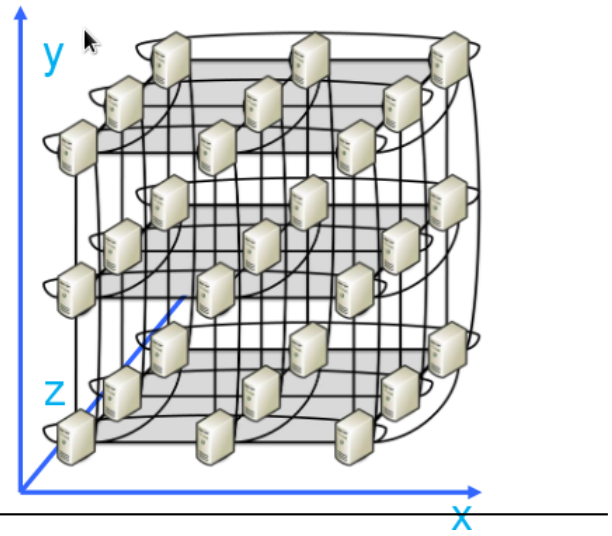
0 extracted from “Rethinking the Network Stack for Rack-scale Comp.”, HPDC 2014

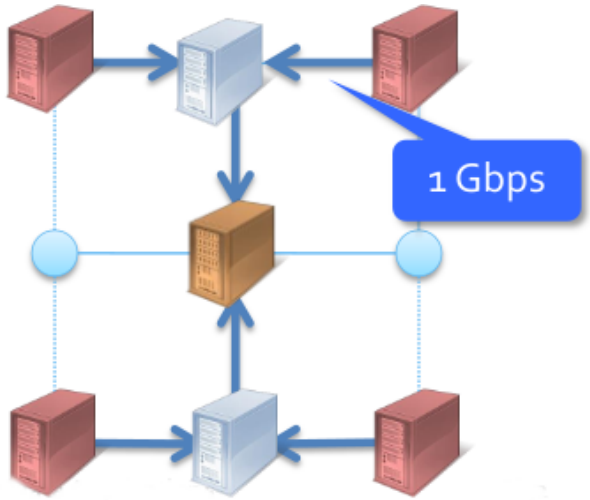
Rackscale: nice, but needs adapted software

- Bottlenecks due to physical topology when aggregating



- → map logical/physical topology



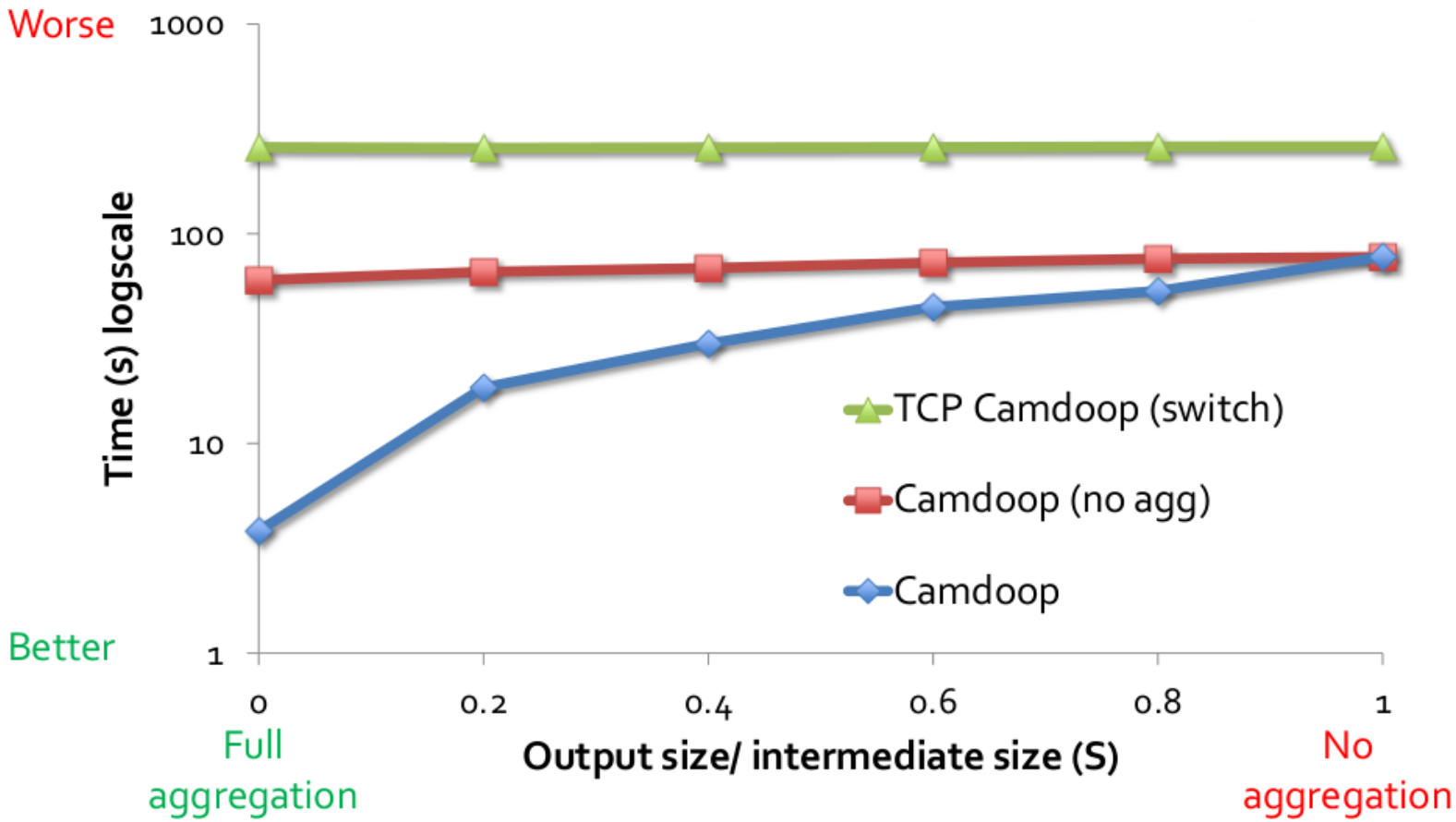


Camcube, NSDI 2012

technicolor

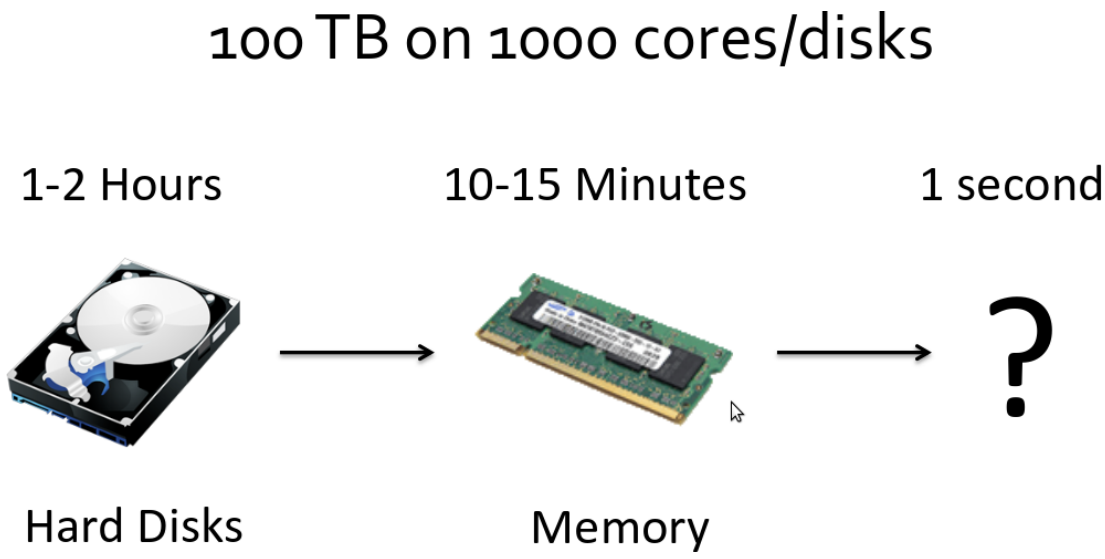
Rackscale: adapting Hadoop

- Camdoop: adapting Hadoop to the rack, e.g. in rack aggregation



technicolor

Trading accuracy with response time

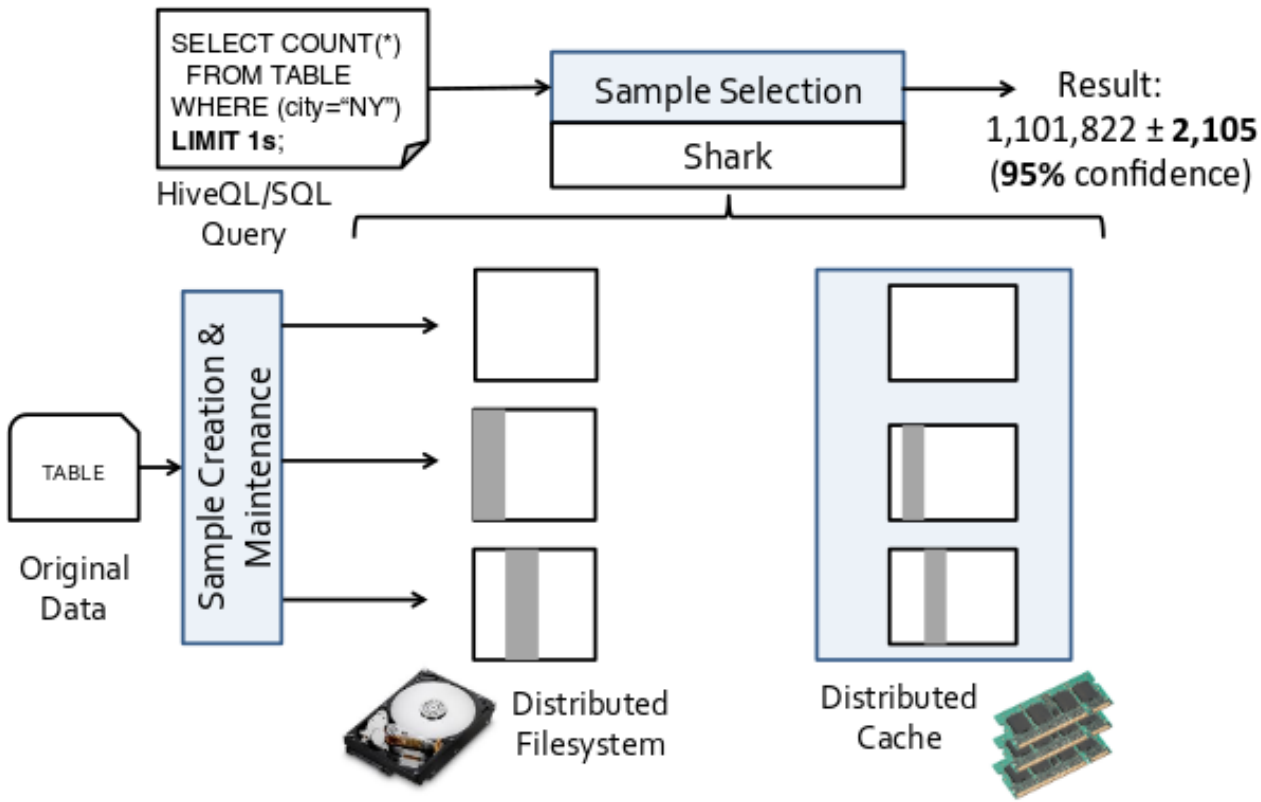


- scanning 200-300GB RAM may take 10 seconds
- some queries may involve that much data: not efficient !
- exact answers not needed all the time $\rightarrow$ bounded error

0 BlinkDB: queries with bounded errors and bounded response times on very large data, Eurosys 2013

BlinkDB: constrained queries

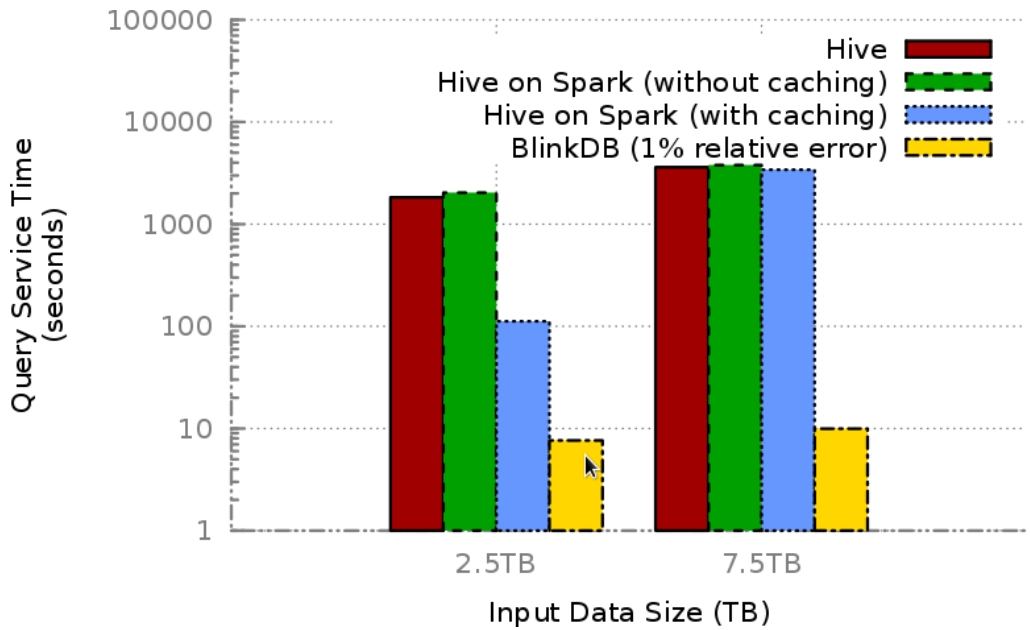
```
SELECT COUNT(*)
FROM Sessions
WHERE Genre = 'western'
GROUP BY OS
ERROR WITHIN 10% AT CONFIDENCE 95%
```



technicolor

BlinkDB: time improvement

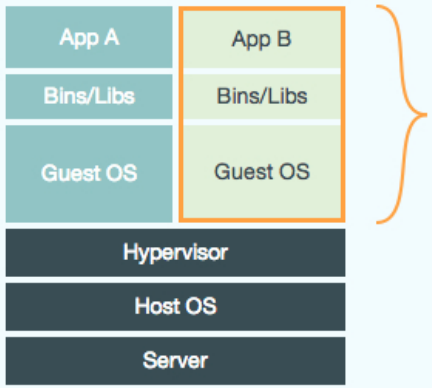
- Assumes that colums used for queries are stable over time
- Sample creation module
- Sample selection module (best fit for current query/constraints)



- In general: so much data that sometimes does not make any sense to have a 100% exact result, if error can be bounded and lots of time saved

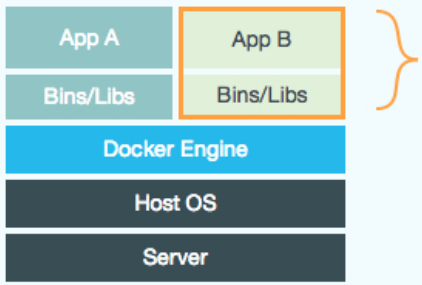
technicolor

Distributed system from containers: Docker



Virtual Machines

Each virtualized application includes not only the application - which may be only 10s of MB - and the necessary binaries and libraries, but also an entire guest operating system - which may weigh 10s of GB.



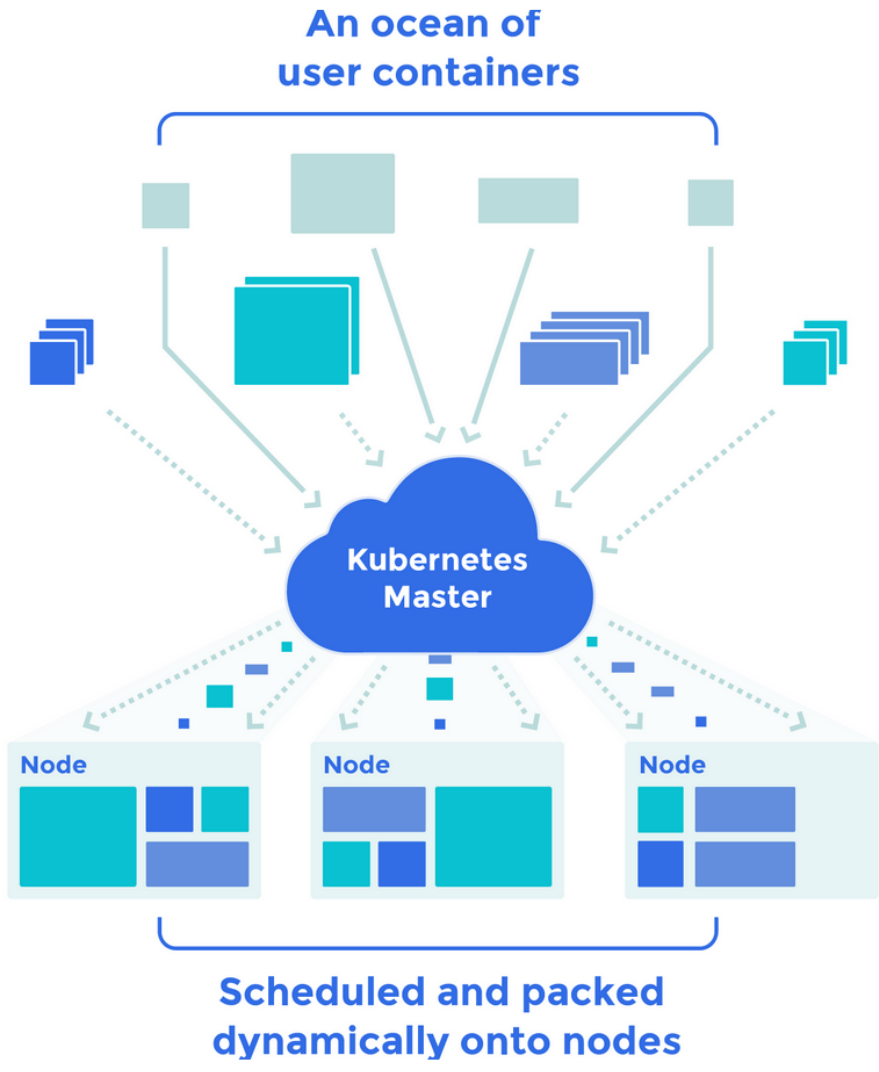
Docker

The Docker Engine container comprises just the application and its dependencies. It runs as an isolated process in userspace on the host operating system, sharing the kernel with other containers. Thus, it enjoys the resource isolation and allocation benefits of VMs but is much more portable and efficient.

- Lightweight shipping of applications in one container, as opposed to Virtual Machines

technicolor

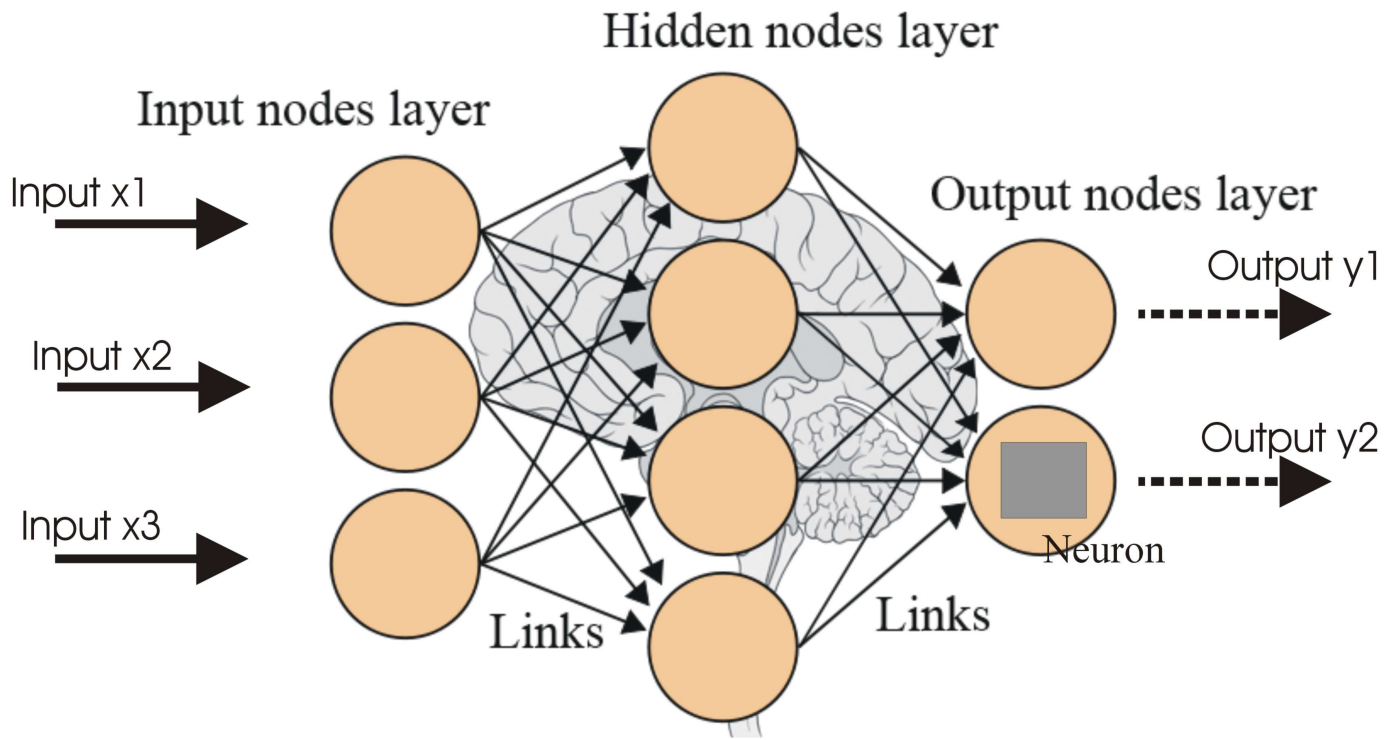
Distributed system from containers: orchestration



- Building and managing a large distrbuted system of containers

0 http://googlecloudplatform.blogspot.com/2015_01_01_archive.html

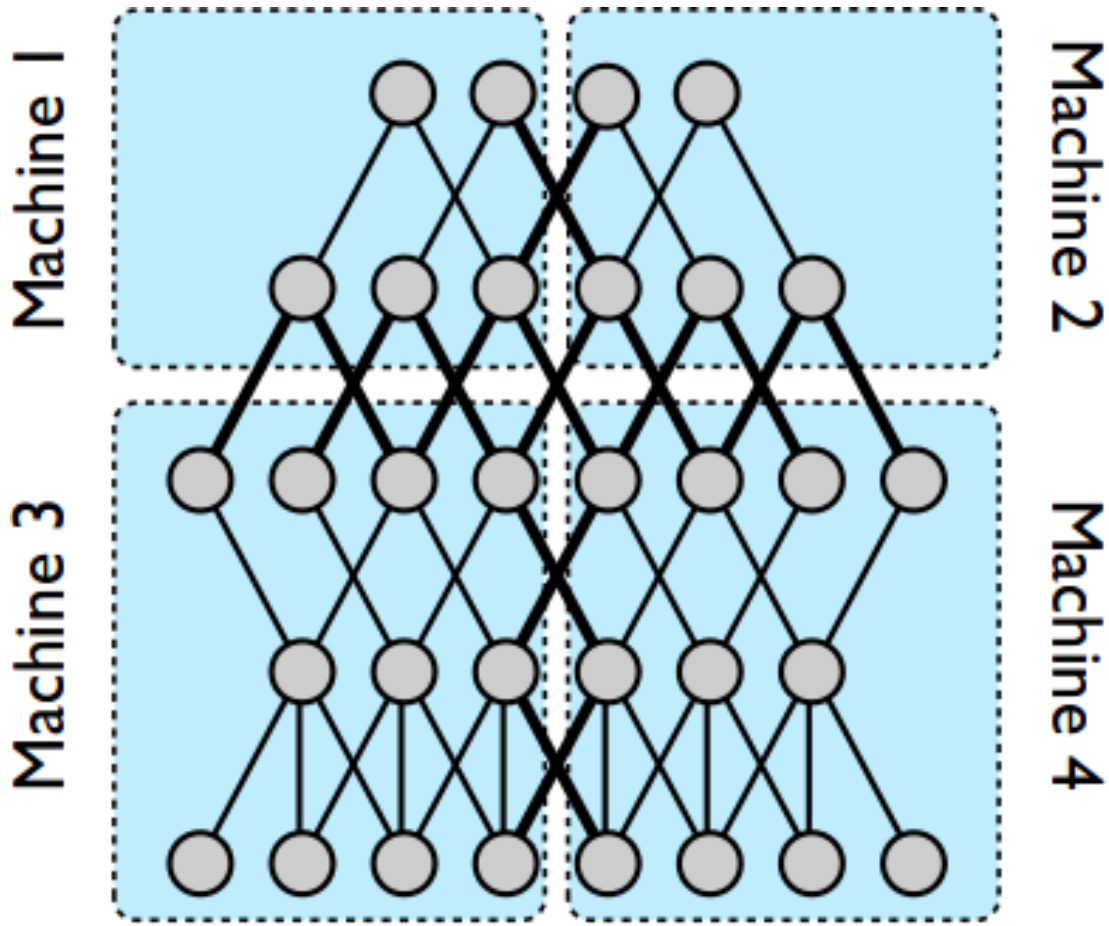
Deep neural nets: powerful machine learning



- Here a single layer neural network

technicolor

Deep neural nets: basic distribution

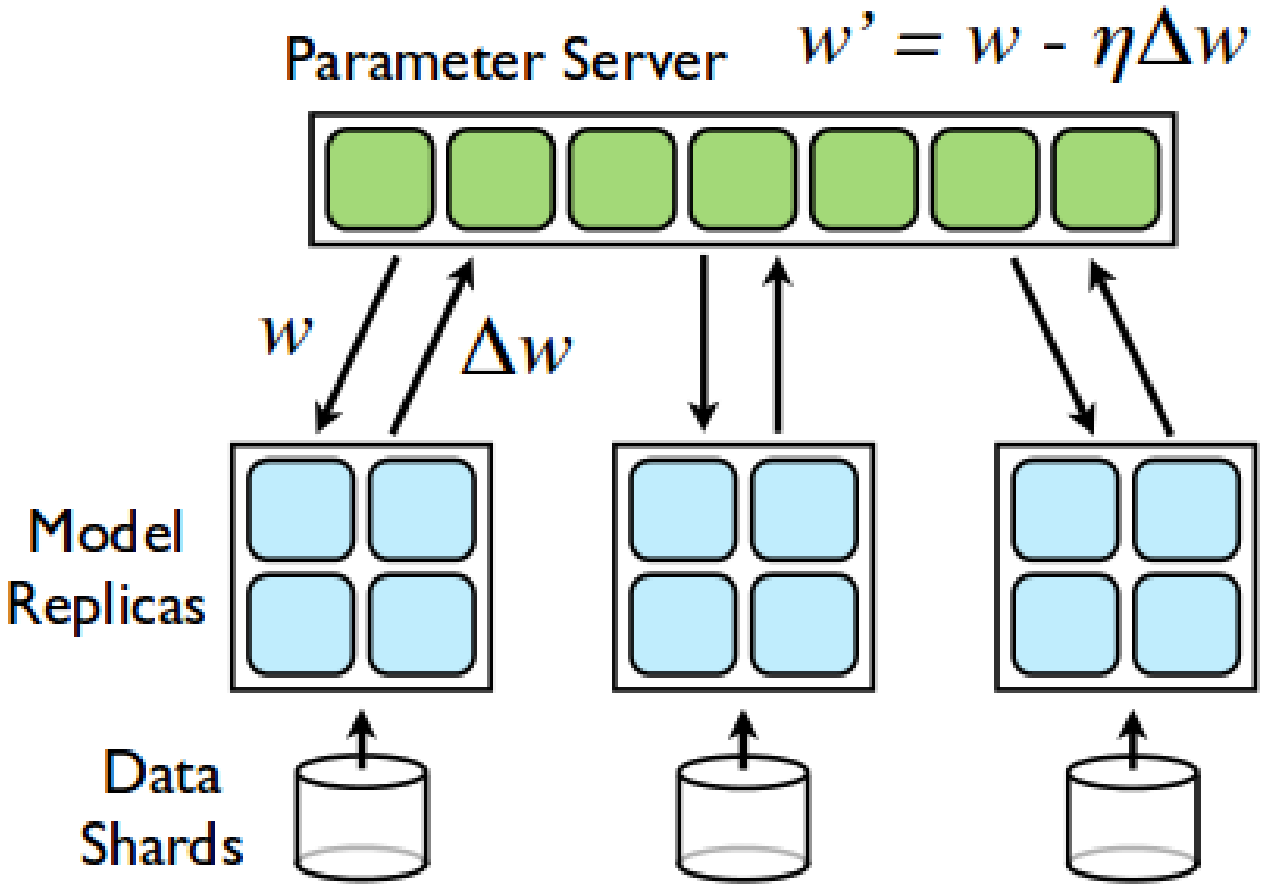


- Multi-layer “deep” network, distributed on 4 machines
- High latency occurs on “cut” links

0 Google’s deepdist, NIPS 2012

technicolor

Deep neural nets: model for massive parallelization



- Model pulled from server on replicas, learn on local data shard and then push model updates to parameter server

0 Google's deepdist, NIPS 2012

bitcoin: A P2P electronic cash system

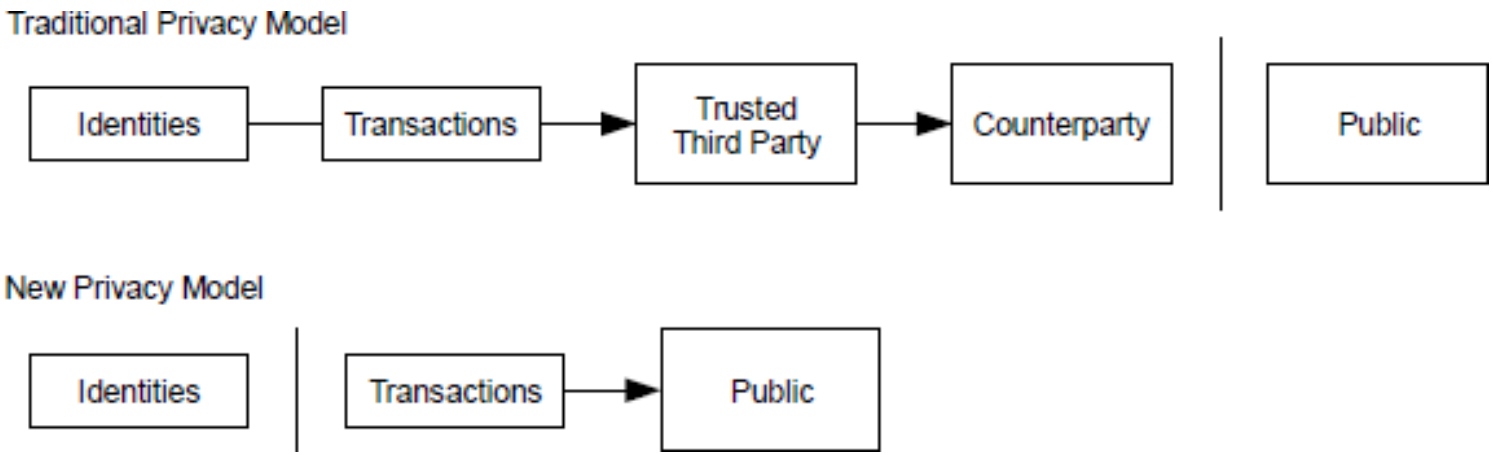
- Internet relies on financial institutions serving as **trusted** parties to process electronic payments
- Bitcoin: **rely on cryptographic proofs instead of trust**, to avoid a central third party
- Claim: secure if nodes collectively control more CPU power than attacker nodes

0 Bitcoin: A Peer-to-Peer Electronic Cash System, draft

technicolor

bitcoin: privacy model

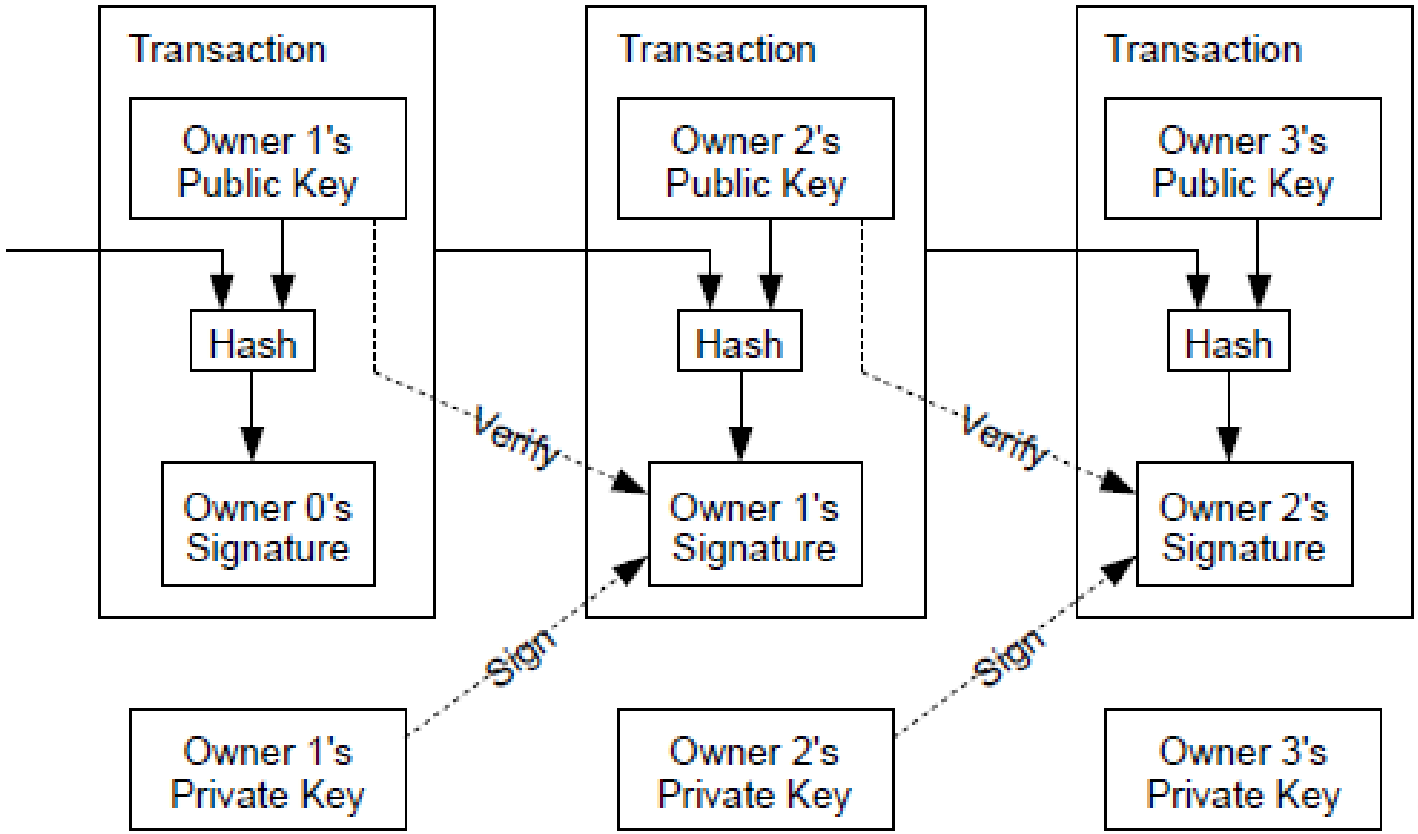
- Traditional banks: limiting access to information to involved parties only
- Bitcoin: rely on “unlinkable” pair of keys for each transaction (no way to put a name on a particular key)
- Transactions are public, but without linking any real people



technicolor

bitcoin: Coins and transactions

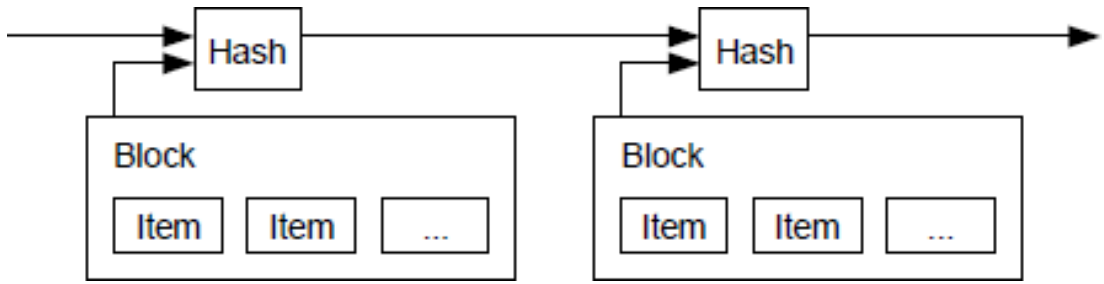
- An electronic coin = a chain of digital signatures
- A user transfers a coin to someone by
  - signing the hash of the previous transaction, and
  - the hash of payee public key
- The payee can check the chain of ownership



technicolor

bitcoin: Coins and transactions (2)

- Problem: as there is no central authority, payee cannot check if the coin was **double-spent**
- Solution: every participating node should be aware of **all** transactions, so that a majority can assess a transaction is “correct” (a given coin is used once)
- Assessment possible as nodes agree on a single history of order of transactions. Possible using timestamps on transactions
- Hashing blocks of items to be timestamped:

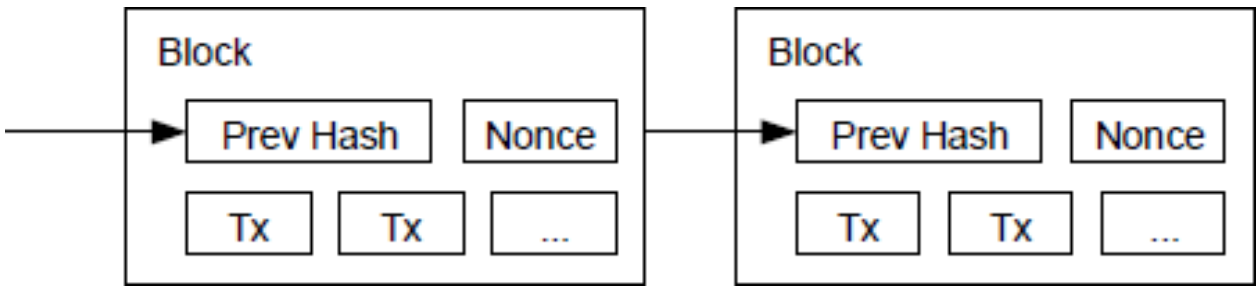


- Each timestamp includes previous one to form a chain
- A hash **proves** that transaction has existed at the time

technicolor

bitcoin: Coins and transactions (3)

- To avoid attacker nodes to break this, a proof-of-work is needed
  - Hard computationally, but quick to check (e.g. a captcha)
  - Here incrementing a **nounce** in a block, until the hash starts with a given number of zeros (exponential work w.r.t. that number of zeros)

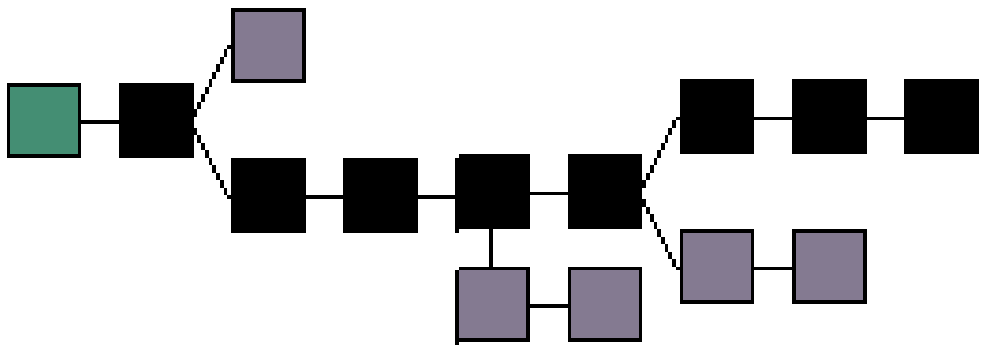


- **If a majority of CPU power is controlled by honest nodes, this honest chain will grow faster and outpace chains from attackers**
  (e.g. an attacker cannot reuse an already given coin because it cannot recreate a longer “false” chain)

bitcoin: network protocol

1) New transactions are broadcast to all nodes.
2) Each node collects new transactions into a block.
3) Each node works on finding a difficult proof-of-work for its block.
4) When a node finds a proof-of-work, it broadcasts the block to all nodes.
5) Nodes accept the block only if all transactions in it are valid and not already spent.
6) Nodes express their acceptance of the block by working on creating the next block in the chain, using the hash of the accepted block as the previous hash.

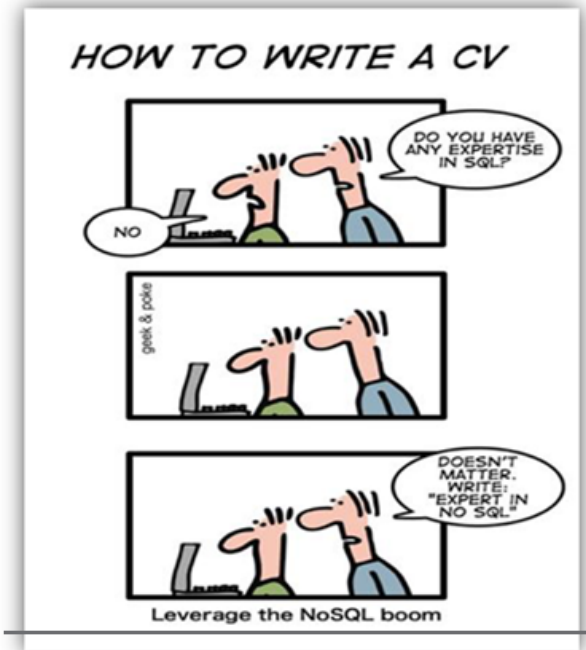
- Concurrency is handled by simply always choosing the longest received chain



- The blockchain concept is expanding for code execution (smart contracts, cf Ethereum), DNS systems, …

Conclusion

- Fast changing area (technology, needs, science)
- Unfortunately, or fortunately for system designers: no one size fits all for storage or/and processing. Carefull design needed based on the app requirements
- Be SURE you need big data solutions!
  - NoSQL: at least hundreds of millions / billions of rows
  - Hadoop: consider it if data does not fit on one disk (10sTB), if data forms a single format/table, and latency ultra critical



Contact me for **any** question or concern:
erwan.lemerrer@technicolor.com

technicolor