Distributed Systems 1

CUCS Course 4113
https://systems.cs.columbia.edu/ds1-class/

Instructor: Roxana Geambasu

Large-Scale Software Systems Stacks

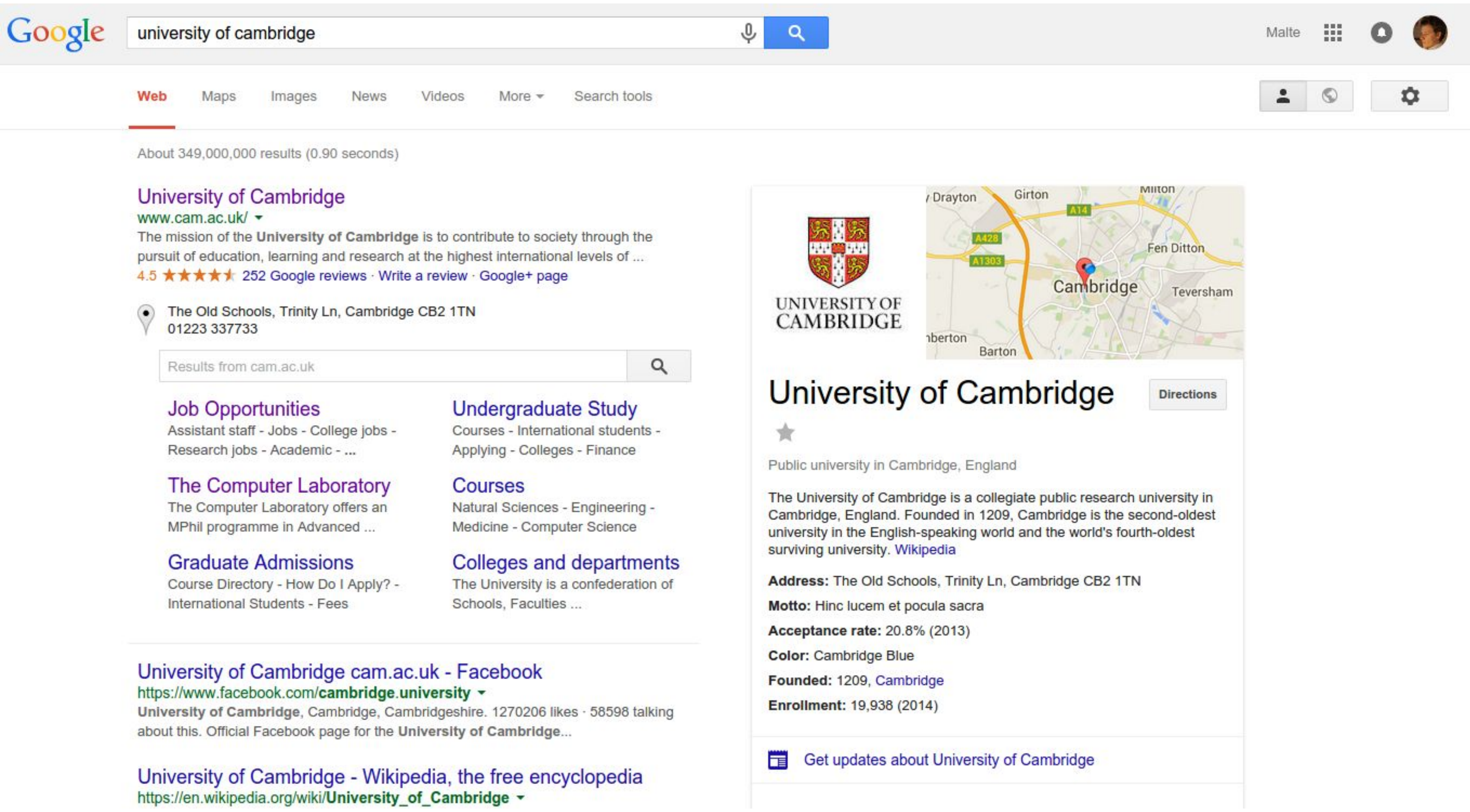
Lecture Theme

- We talked a lot about storage in this class, plus a bit about distributed computation. For storage, we focused on a particular type of interface (transactional databases).
- But there's a **vast range of infrastructural components** that are needed for building successful distributed applications. Large companies and open-source communities have such components available.
- This lecture aims to provide an **index of such components**. We won't give details about how these components are built, but pointers to where you can find out more.
- We'll also give pointers to valuable advice on skills and patterns useful for building large-scale systems.

Acknowledgements

- Because the course lecture is so broad, there’s a lot to acknowledge for the content provided here.
- Particularly important for these slides are two sources:
  - A 2015 talk by Malte Schwarzkopf on software systems stacks at large companies [1].
  - A couple of talks by Jeff Dean about experience and advice from building some key infrastructure systems at Google (original slides [2] and [3]).

“What It Takes to Build Google?”



Google | university of cambridge | Malte

Web | Maps | Images | News | Videos | More | Search tools

About 349,000,000 results (0.90 seconds)

University of Cambridge
www.cam.ac.uk/
The mission of the **University of Cambridge** is to contribute to society through the pursuit of education, learning and research at the highest international levels of ...
4.5 ★★★★★ 252 Google reviews · Write a review · Google+ page

The Old Schools, Trinity Ln, Cambridge CB2 1TN
01223 337733

Results from cam.ac.uk

Job Opportunities
Assistant staff - Jobs - College jobs - Research jobs - Academic - ...

Undergraduate Study
Courses - International students - Applying - Colleges - Finance

The Computer Laboratory
The Computer Laboratory offers an MPhil programme in Advanced ...

Courses
Natural Sciences - Engineering - Medicine - Computer Science

Graduate Admissions
Course Directory - How Do I Apply? - International Students - Fees

Colleges and departments
The University is a confederation of Schools, Faculties ...

University of Cambridge cam.ac.uk - Facebook
https://www.facebook.com/**cambridge**.**university**
University of Cambridge, Cambridge, Cambridgeshire. 1270206 likes · 58598 talking about this. Official Facebook page for the **University of Cambridge**...

University of Cambridge - Wikipedia, the free encyclopedia
https://en.wikipedia.org/wiki/**University_of_Cambridge**

University of Cambridge | Directions

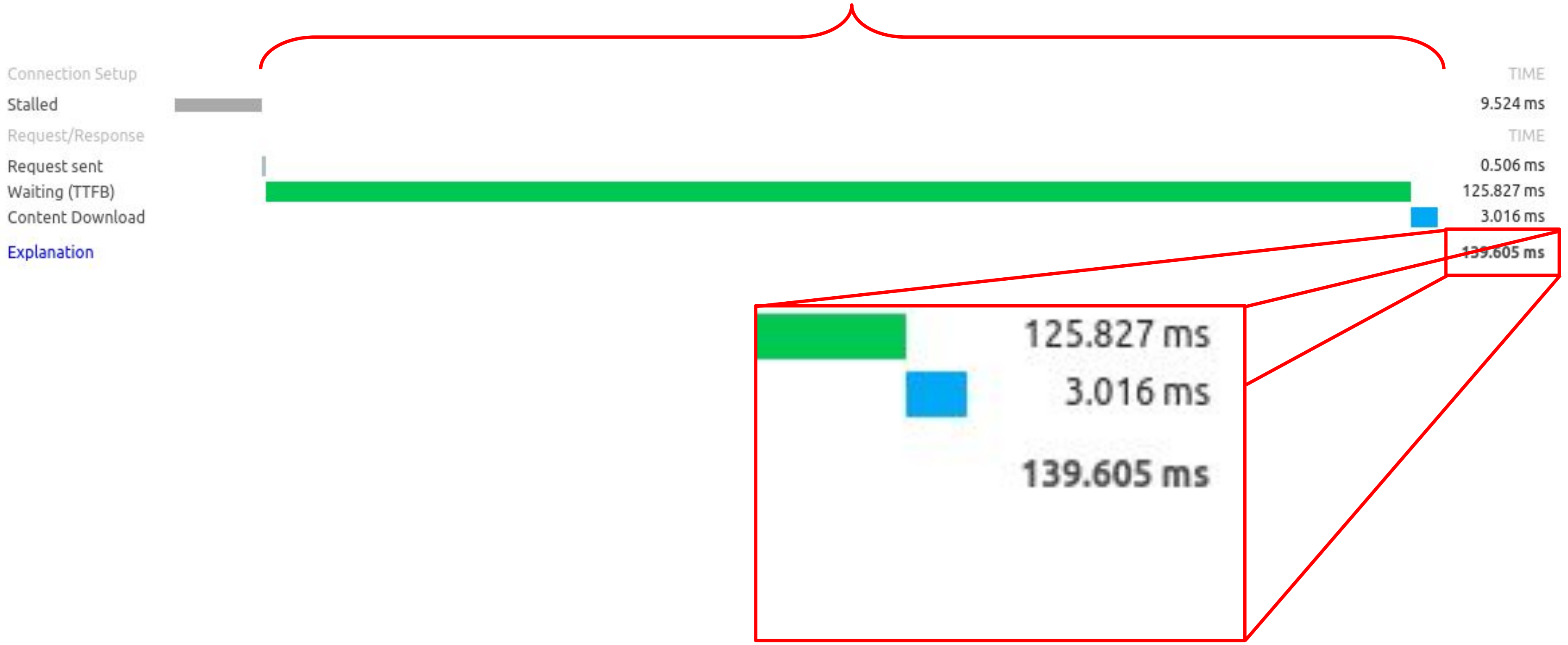
Public university in Cambridge, England

The University of Cambridge is a collegiate public research university in Cambridge, England. Founded in 1209, Cambridge is the second-oldest university in the English-speaking world and the world's fourth-oldest surviving university. Wikipedia

Address: The Old Schools, Trinity Ln, Cambridge CB2 1TN
Motto: Hinc lucem et pocula sacra
Acceptance rate: 20.8% (2013)
Color: Cambridge Blue
Founded: 1209, Cambridge
Enrollment: 19,938 (2014)

Get updates about University of Cambridge

What happens here?



Connection Setup TIME

Stalled 9.524 ms

Request/Response TIME

Request sent 0.506 ms

Waiting (TTFB) 125.827 ms

Content Download 3.016 ms

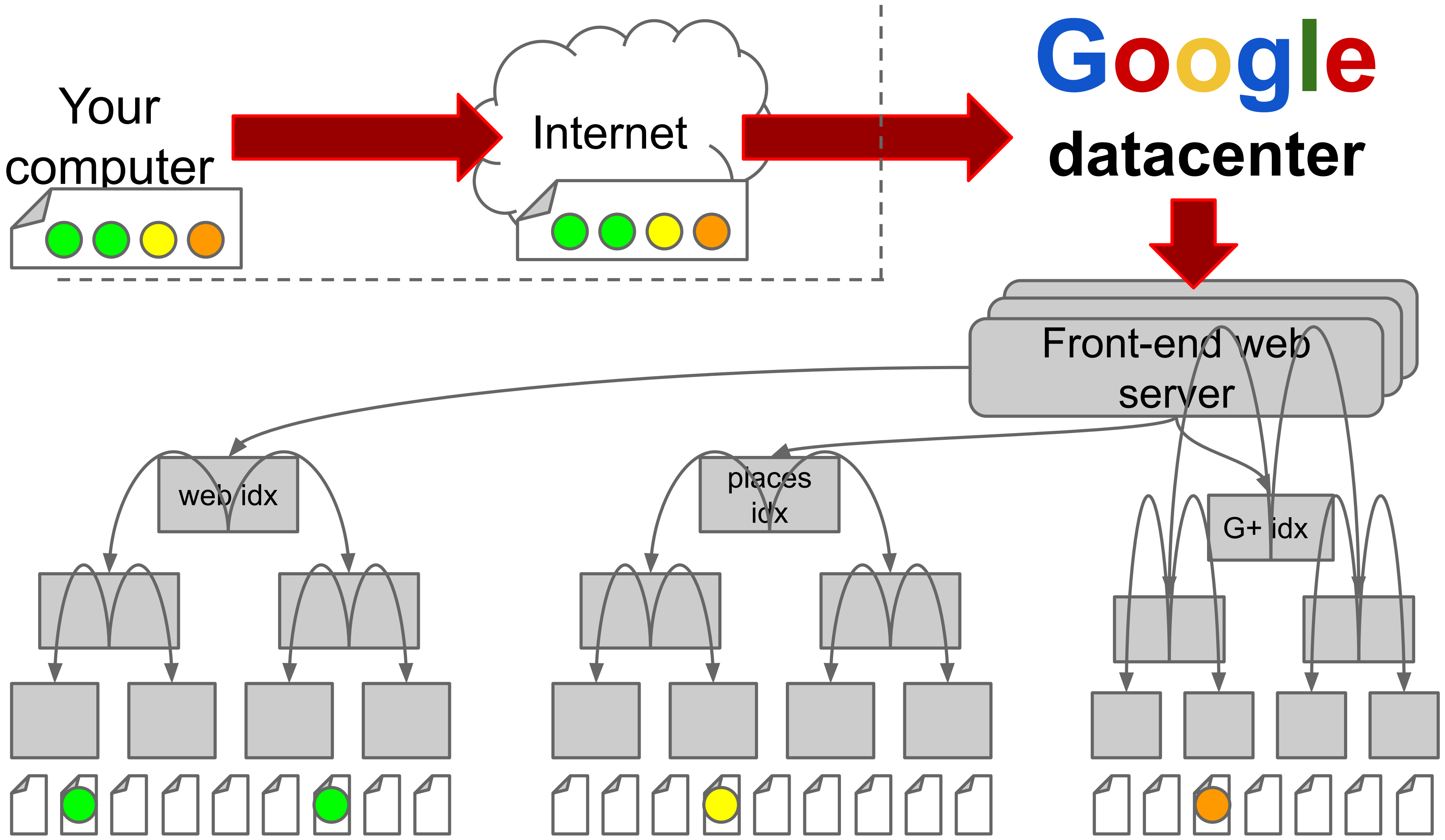
Explanation 139.605 ms

125.827 ms

3.016 ms

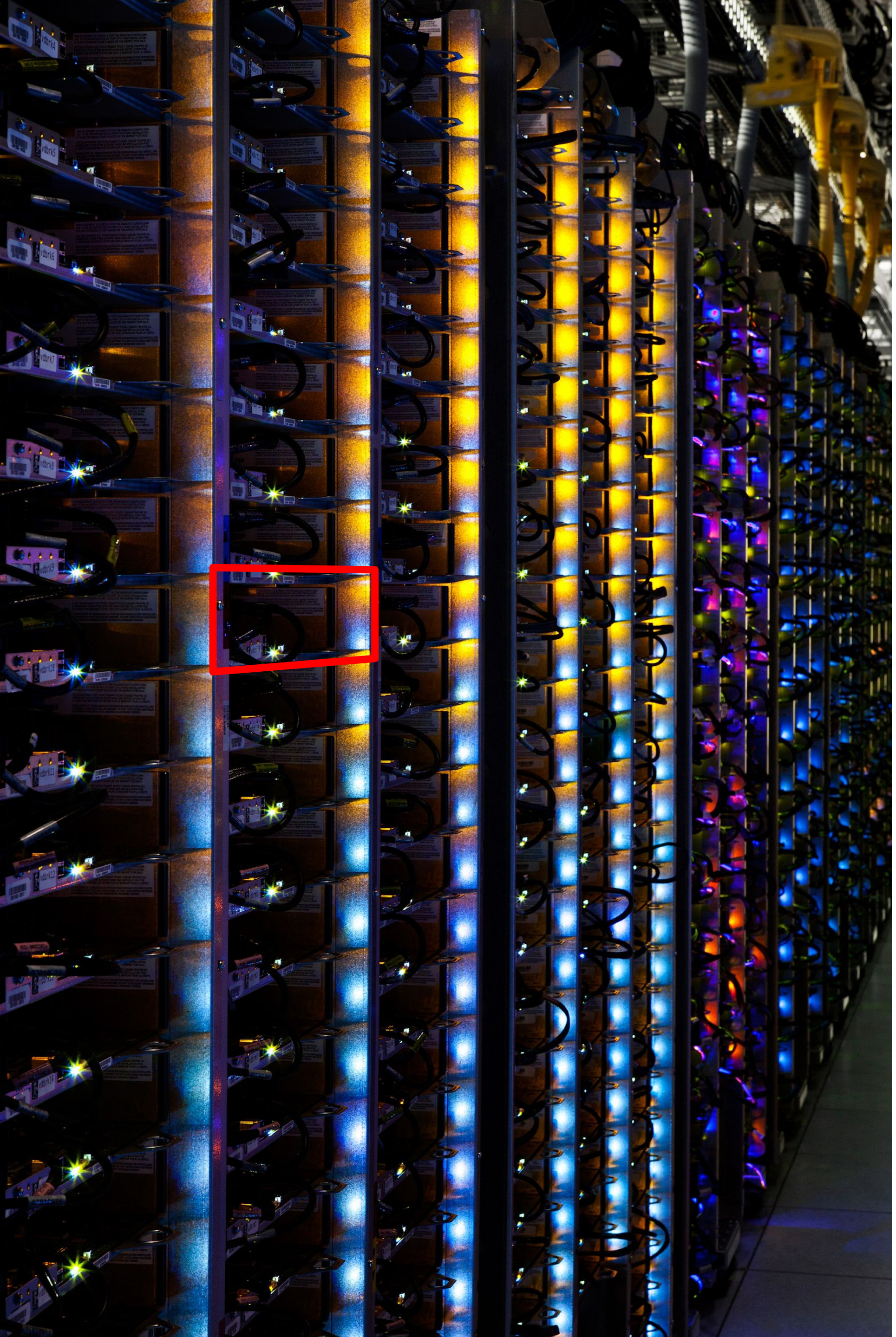
139.605 ms

What happens in those 139ms?



What we'll chat about

1. Datacenter hardware
2. Cluster failures
3. Scalable & fault tolerant software stacks
   a. Google
   b. Facebook
   c. Open source



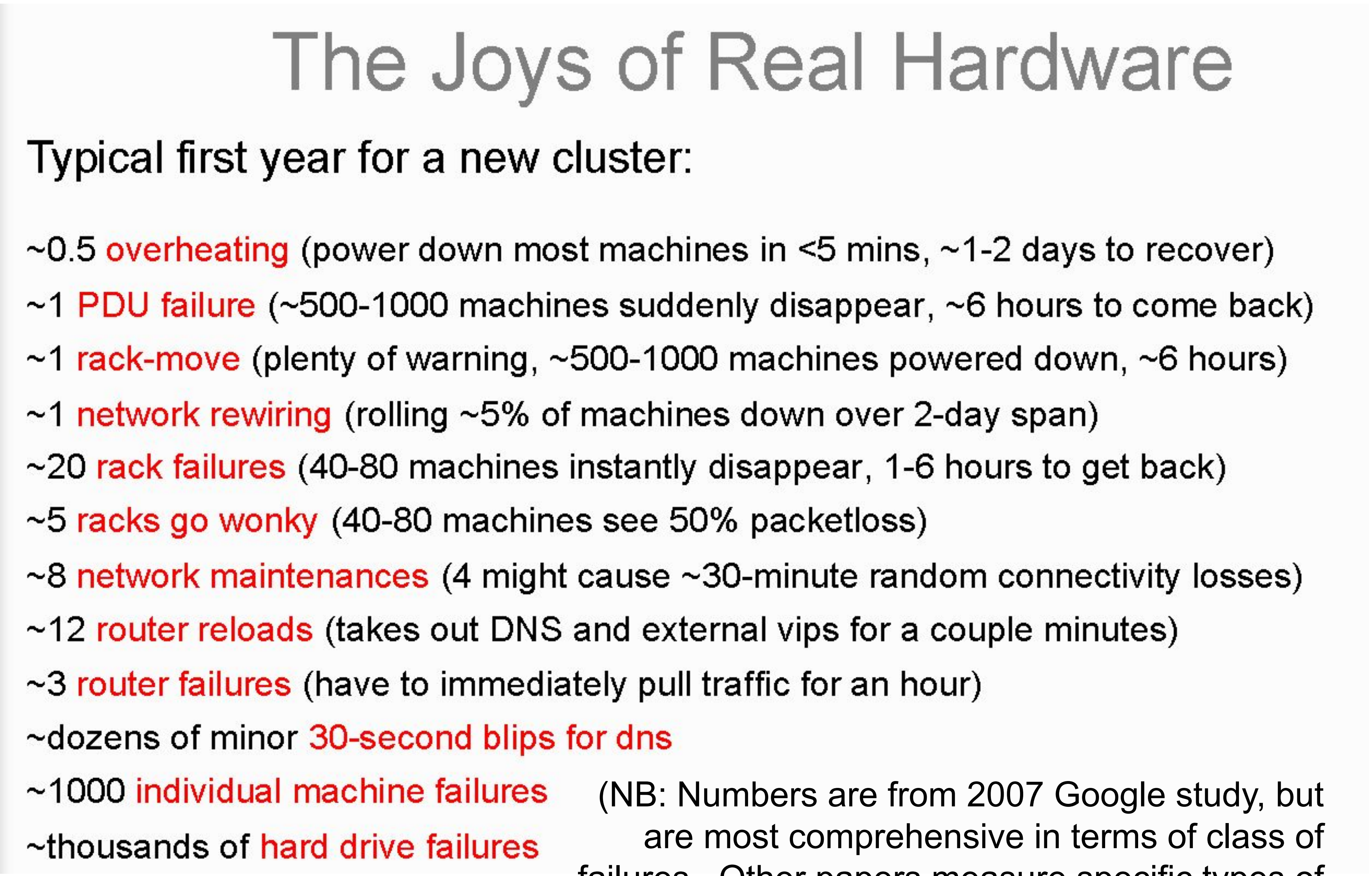
From Meta (as of 2022):

- O(1M) machines in total
- O(10s) regions
- O(1000s) interdependent services

- "Machine"
  - no chassis
  - DC battery
  - mostly custom-made

- Network
  - ToR switch
  - multi-path core

A video surveying a Google Datacenter (as of 2020) is here.



The Joys of Real Hardware

Typical first year for a new cluster:

~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% packetloss)

~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

(NB: Numbers are from 2007 Google study, but are most comprehensive in terms of class of failures. Other papers measure specific types of failure, such as this for disks and this for DRAM.)

Source: Jeff Dean
https://static.googleusercontent.com/media/research.google.com/en//people/jeff/stanford-295-talk.pdf, 2007.

Dealing with Scale and Failures

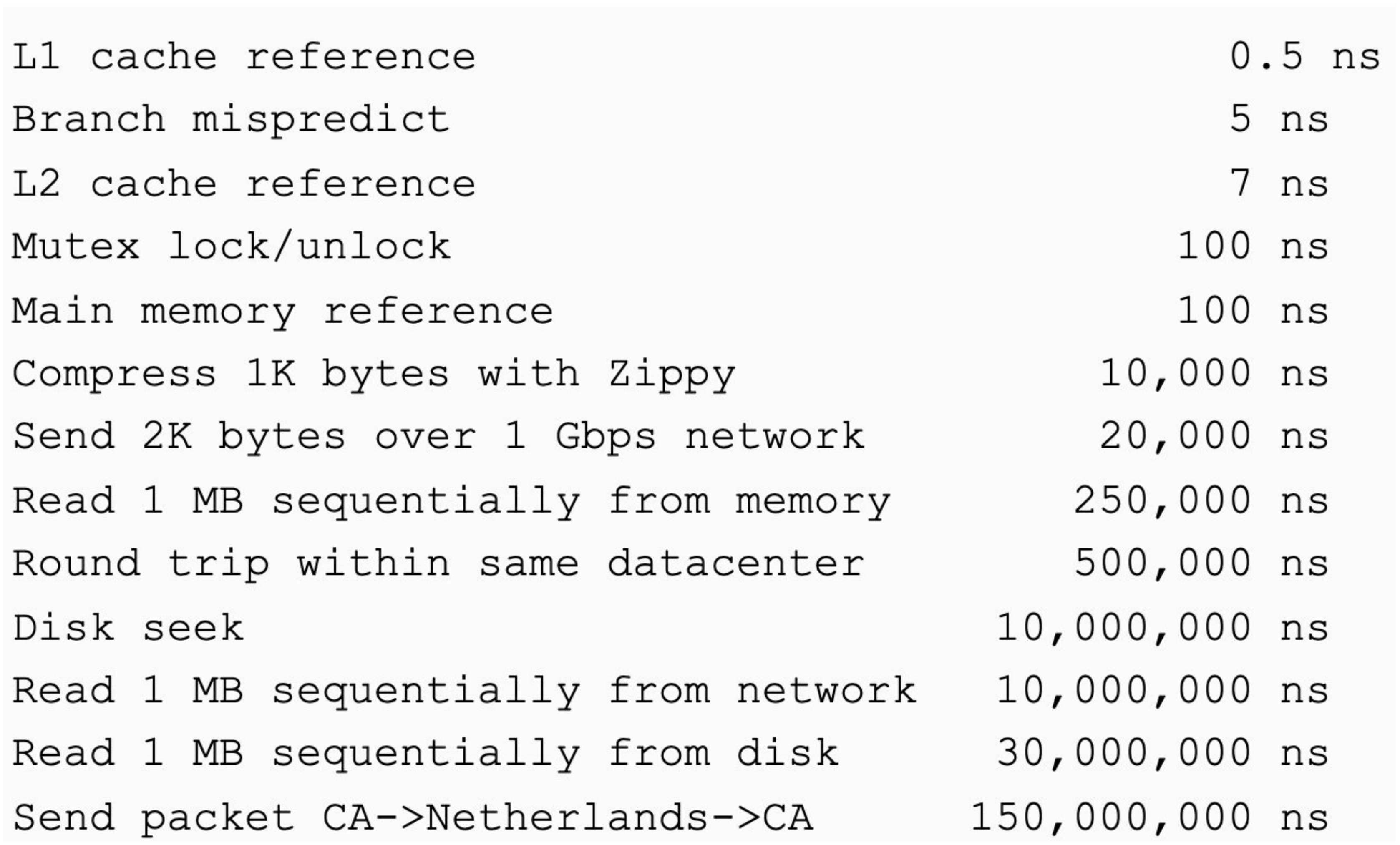
1. Leverage infrastructure systems that solve portions of your problem at scale and with fault-tolerance.
2. Follow engineering patterns for how to develop scalable, fault tolerant systems.
3. Reason about the space of design and try make design choices based on assessments of tradeoffs, either from back-of-the-envelope or from basic prototype evaluations.

Today: We’ll talk about the kinds of infrastructure systems that are often needed (and available) at companies or in the open-source community (#1 above).

Refer to these slides [2] by Jeff Dean for DS design patterns and tradeoff analysis advice (#2 and #3 above). We’ll only include here one example back-of-the-envelope calculation. Note the final quiz may

Numbers Everyone Should Know

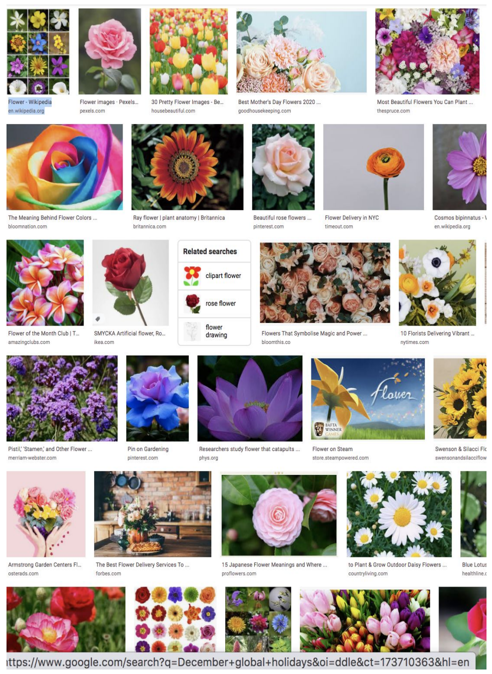
(NB: Numbers are outdated, keep searching for latest numbers online, e.g., [4])



L1 cache reference	0.5 ns
Branch mispredict	5 ns
L2 cache reference	7 ns
Mutex lock/unlock	100 ns
Main memory reference	100 ns
Compress 1K bytes with Zippy	10,000 ns
Send 2K bytes over 1 Gbps network	20,000 ns
Read 1 MB sequentially from memory	250,000 ns
Round trip within same datacenter	500,000 ns
Disk seek	10,000,000 ns
Read 1 MB sequentially from network	10,000,000 ns
Read 1 MB sequentially from disk	30,000,000 ns
Send packet CA->Netherlands->CA	150,000,000 ns

Source: Jeff Dean, https://static.googleusercontent.com/media/research.google.com/en//people/jeff/stanford-295-talk.pdf

Calculation 1: Thumbnail Page Generation


https://www.google.com/search?q=December+global+holidays&oi=ddle&ct=173710363&hl=en

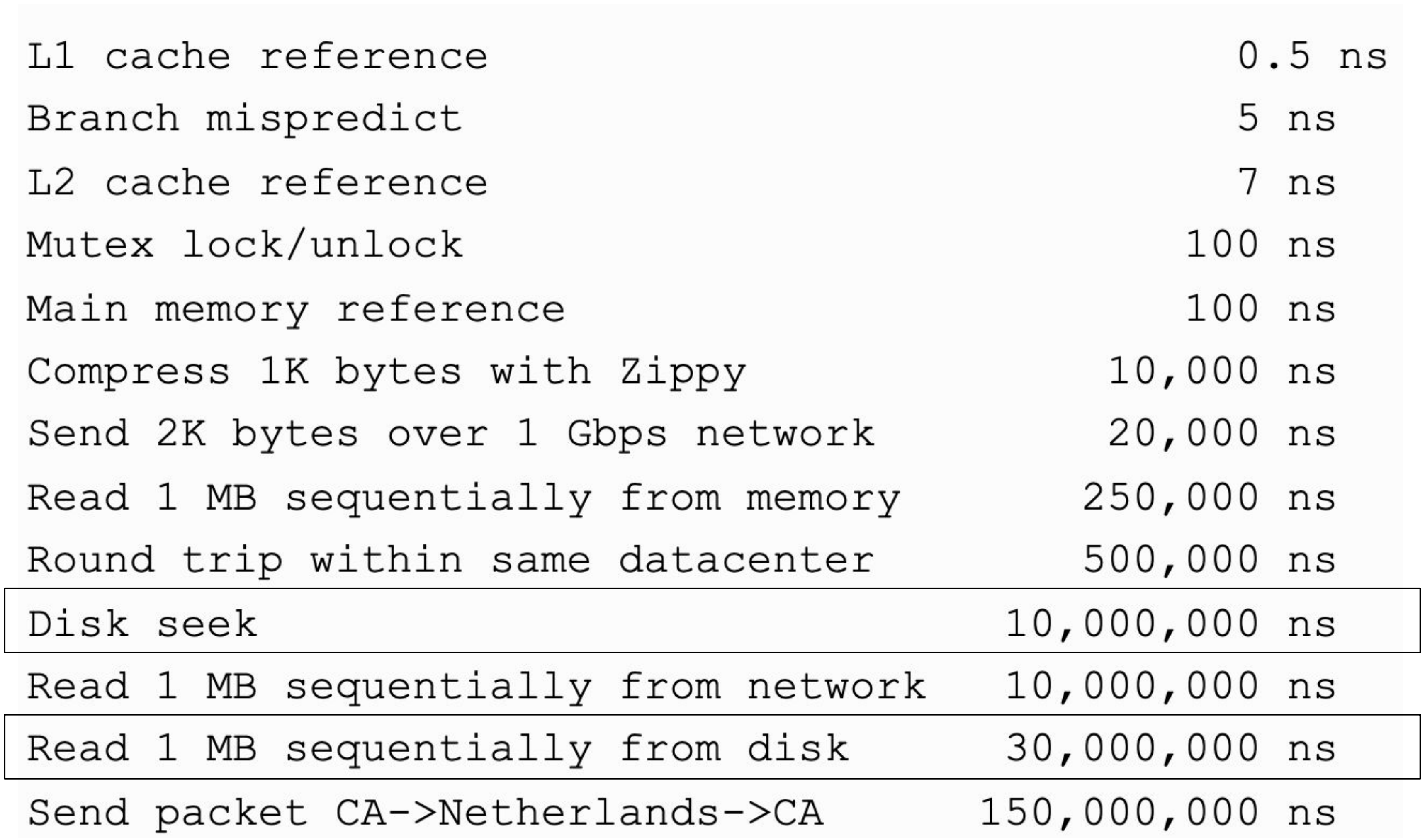
Question: How long to generate the image thumbnail page for an album of 30 pics (256KB/thumbnail)?

- Consider at least two designs for how the album app might interact with the file system to retrieve the thumbnails. Assume local application, no network/distribution.
- Use "Numbers Everyone Should Know" (previous slide) to give an order of magnitude estimation of the runtime under each design.
- In your answer sheet, briefly describe your options, give your assessment for runtime for each, and identify whether there is a clear winner?

https://tinyurl.com/back-of-the-envelope-activity

Numbers Everyone Should Know

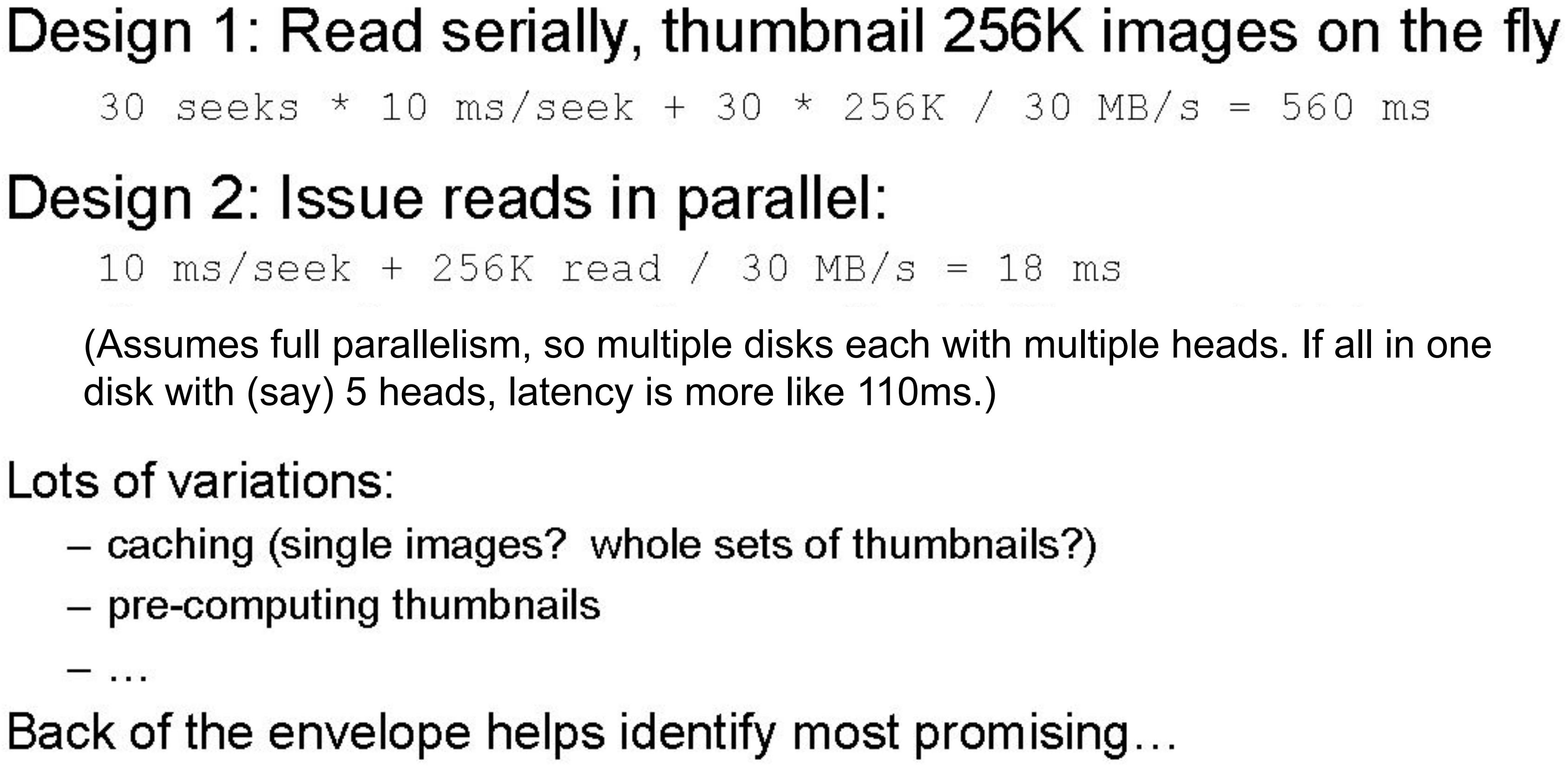
(NB: Numbers are outdated, keep searching for latest numbers online, e.g., [4])



L1 cache reference	0.5 ns
Branch mispredict	5 ns
L2 cache reference	7 ns
Mutex lock/unlock	100 ns
Main memory reference	100 ns
Compress 1K bytes with Zippy	10,000 ns
Send 2K bytes over 1 Gbps network	20,000 ns
Read 1 MB sequentially from memory	250,000 ns
Round trip within same datacenter	500,000 ns
Disk seek	10,000,000 ns
Read 1 MB sequentially from network	10,000,000 ns
Read 1 MB sequentially from disk	30,000,000 ns
Send packet CA->Netherlands->CA	150,000,000 ns

Source: Jeff Dean, https://static.googleusercontent.com/media/research.google.com/en//people/jeff/stanford-295-talk.pdf

Calculation 1: Thumbnail Page Generation



Design 1: Read serially, thumbnail 256K images on the fly

```
30 seeks * 10 ms/seek + 30 * 256K / 30 MB/s = 560 ms
```

Design 2: Issue reads in parallel:

```
10 ms/seek + 256K read / 30 MB/s = 18 ms
```

(Assumes full parallelism, so multiple disks each with multiple heads. If all in one disk with (say) 5 heads, latency is more like 110ms.)

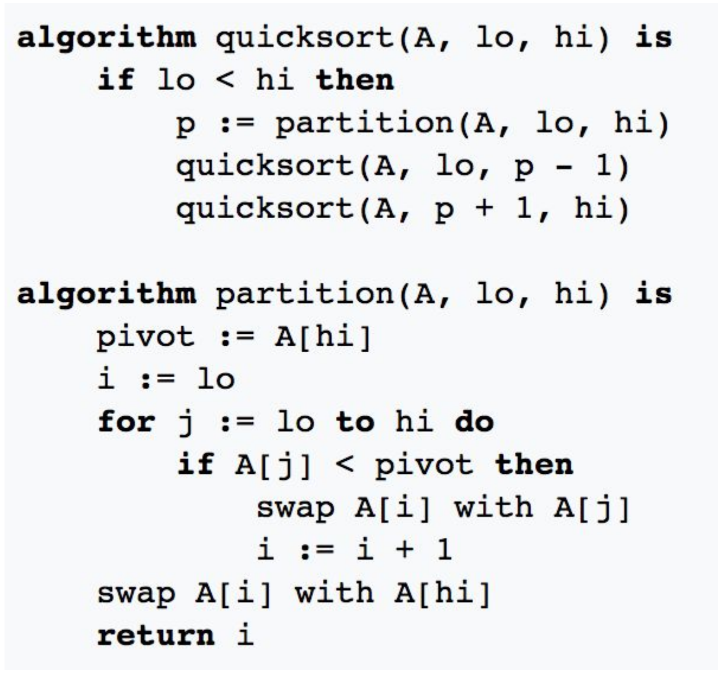
Lots of variations:

- caching (single images? whole sets of thumbnails?)
- pre-computing thumbnails
- …

Back of the envelope helps identify most promising…

Source: Jeff Dean, https://static.googleusercontent.com/media/research.google.com/en//people/jeff/stanford-295-talk.pdf

Calculation 2: Quicksort 1GB Numbers



```
algorithm quicksort(A, lo, hi) is
    if lo < hi then
        p := partition(A, lo, hi)
        quicksort(A, lo, p - 1)
        quicksort(A, p + 1, hi)

algorithm partition(A, lo, hi) is
    pivot := A[hi]
    i := lo
    for j := lo to hi do
        if A[j] < pivot then
            swap A[i] with A[j]
            i := i + 1
    swap A[i] with A[hi]
    return i
```

```
quicksort(A, 0, length(A) - 1).
```

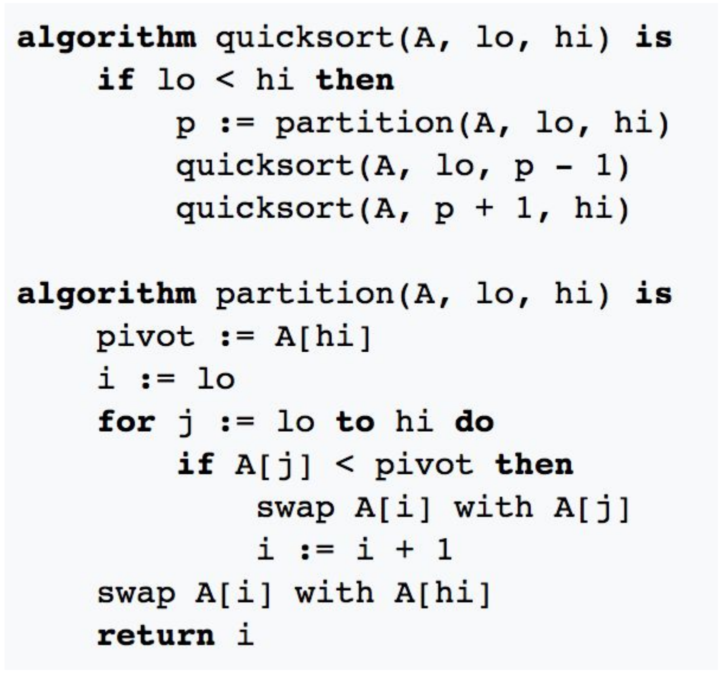
Question: How long to quicksort 1GB's worth of 4-byte numbers?

- Assume all numbers are in RAM.
- Think about how many numbers that would mean. → 2^28 numbers = n
- Remind yourselves of the algorithm and think of what the most expensive operations are likely to be. → 1) branch mispredictions (due to comparisons); 2) memory accesses
- For each expensive operation:
  - Approximate how many such ops on average.
    - 1) # comparisons: n log n = 2^28*28 ~= 2^33. Half mispredict: 2^32 branch mispredictions
    - 2) amount of memory accessed: 28 * 1GB = 28 GB RAM accessed
- Use "Numbers Everyone Should Know"

Mispredictions: 2^32 mispredivtios * 5ns = ~21 seconds
Memory component: 28GB @ 4GB/s ~= 7 seconds

---
30 seconds for sorting a 1GB-worth of numbers

Calculation 2: Quicksort 1GB Numbers



```
algorithm quicksort(A, lo, hi) is
    if lo < hi then
        p := partition(A, lo, hi)
        quicksort(A, lo, p - 1)
        quicksort(A, p + 1, hi)

algorithm partition(A, lo, hi) is
    pivot := A[hi]
    i := lo
    for j := lo to hi do
        if A[j] < pivot then
            swap A[i] with A[j]
            i := i + 1
    swap A[i] with A[hi]
    return i
```

```
quicksort(A, 0, length(A) - 1).
```

Question: How long to quicksort 1GB's worth of 4-byte numbers?

- Assume all numbers are in RAM.
- Think about how many numbers that would mean.
- Remind yourselves of the algorithm and think of what the most expensive operations are likely to be.
- For each expensive operation:
    - Approximate how many such ops on average.
    - Use "Numbers Everyone Should Know" to approximate the total cost of those ops.
- Then add things up and put your order of magnitude estimation in your answer sheet.

Numbers Everyone Should Know

(NB: Numbers are outdated, keep searching for latest numbers online, e.g., [4])

L1 cache reference	0.5 ns
Branch mispredict	5 ns
L2 cache reference	7 ns
Mutex lock/unlock	100 ns
Main memory reference	100 ns
Compress 1K bytes with Zippy	10,000 ns
Send 2K bytes over 1 Gbps network	20,000 ns
Read 1 MB sequentially from memory	250,000 ns
Round trip within same datacenter	500,000 ns
Disk seek	10,000,000 ns
Read 1 MB sequentially from network	10,000,000 ns
Read 1 MB sequentially from disk	30,000,000 ns
Send packet CA->Netherlands->CA	150,000,000 ns

Source: Jeff Dean, https://static.googleusercontent.com/media/research.google.com/en//people/jeff/stanford-295-talk.pdf

Calculation 2: Quicksort 1GB Numbers

```
algorithm quicksort(A, lo, hi) is
    if lo < hi then
        p := partition(A, lo, hi)
        quicksort(A, lo, p - 1)
        quicksort(A, p + 1, hi)

algorithm partition(A, lo, hi) is
    pivot := A[hi]
    i := lo
    for j := lo to hi do
        if A[j] < pivot then
            swap A[i] with A[j]
            i := i + 1
    swap A[i] with A[hi]
    return i
```

```
quicksort(A, 0, length(A) - 1).
```

Mispredictions = 2^32 misp * 5 ns =~ 21 sec

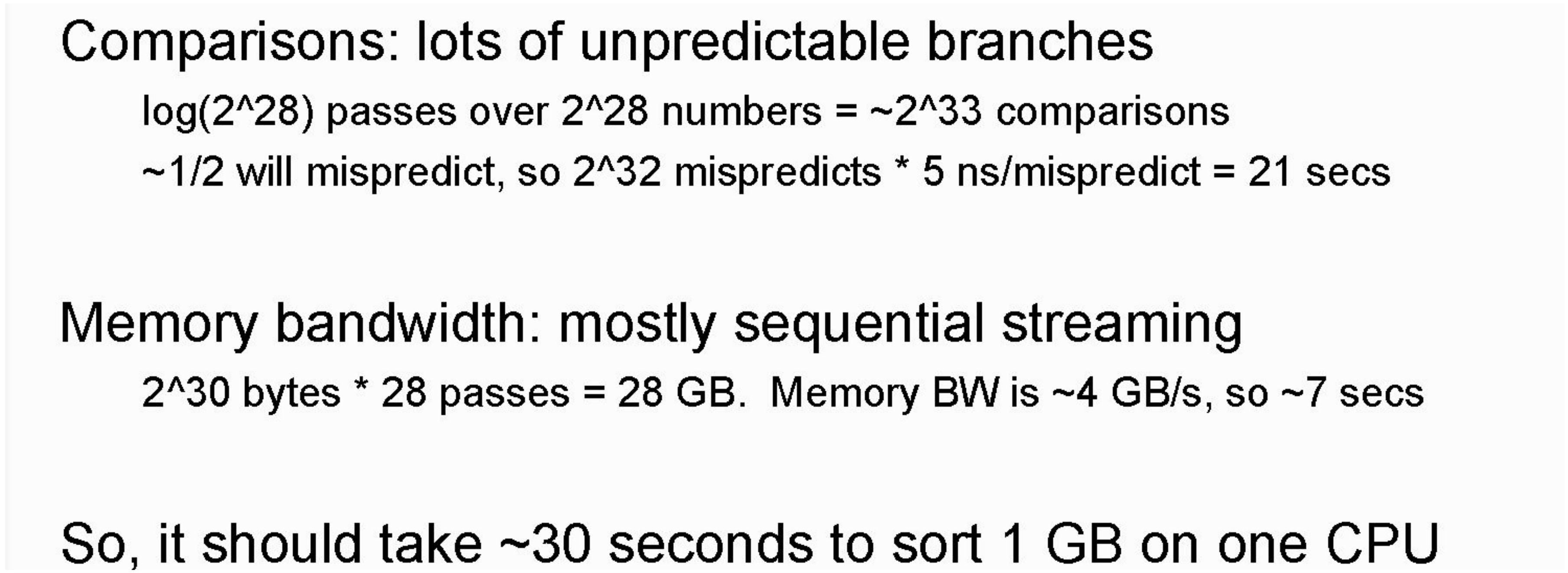
Memory: 28 GB @ 4GB/s =~ 7 sec

Question: How long to quicksort 1 GB of 4 byte numbers?

- Assume all numbers are in RAM.
- Think about how many numbers that would mean. → 2^28 numbers
- Remind yourselves of the algorithm and think of what the most expensive operations are likely to be. → comparisons, memory reads
- For each heavy operation:
  - Approximate how many such ops on average. → comparisons: log(2^28) passes over 2^28 numbers, or 2^33 comparisons. Half mispredict, so 2^32 mispredictions.

    → amount of memory read: 2^30 bytes for 28 passes.
  - Use "Numbers Everyone Should Know" to approximate the total cost of those ops.
- Then add things up and put your order of magnitude estimation in your answer sheet.

Calculation 2: Quicksort 1GB Numbers



Comparisons: lots of unpredictable branches

log(2^28) passes over 2^28 numbers = ~2^33 comparisons

~1/2 will mispredict, so 2^32 mispredicts * 5 ns/mispredict = 21 secs

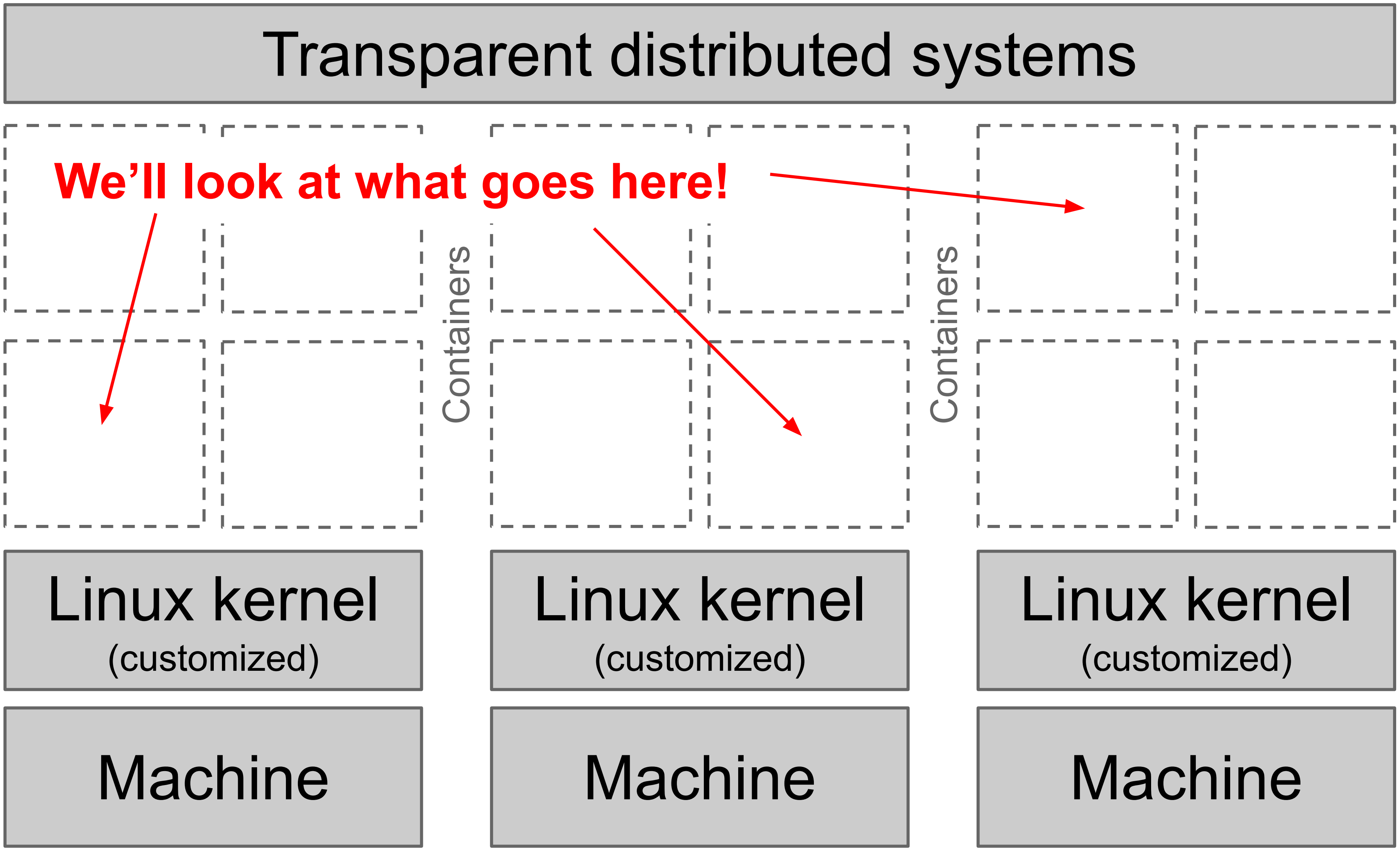
Memory bandwidth: mostly sequential streaming

2^30 bytes * 28 passes = 28 GB. Memory BW is ~4 GB/s, so ~7 secs

So, it should take ~30 seconds to sort 1 GB on one CPU

Source: Jeff Dean, https://static.googleusercontent.com/media/research.google.com/en//people/jeff/Stanford-DL-Nov-2010.pdf

Software Systems Stack


Transparent distributed systems
We'll look at what goes here!
Containers
Containers
Linux kernel (customized)
Linux kernel (customized)
Linux kernel (customized)
Machine
Machine
Machine

The Google Stack

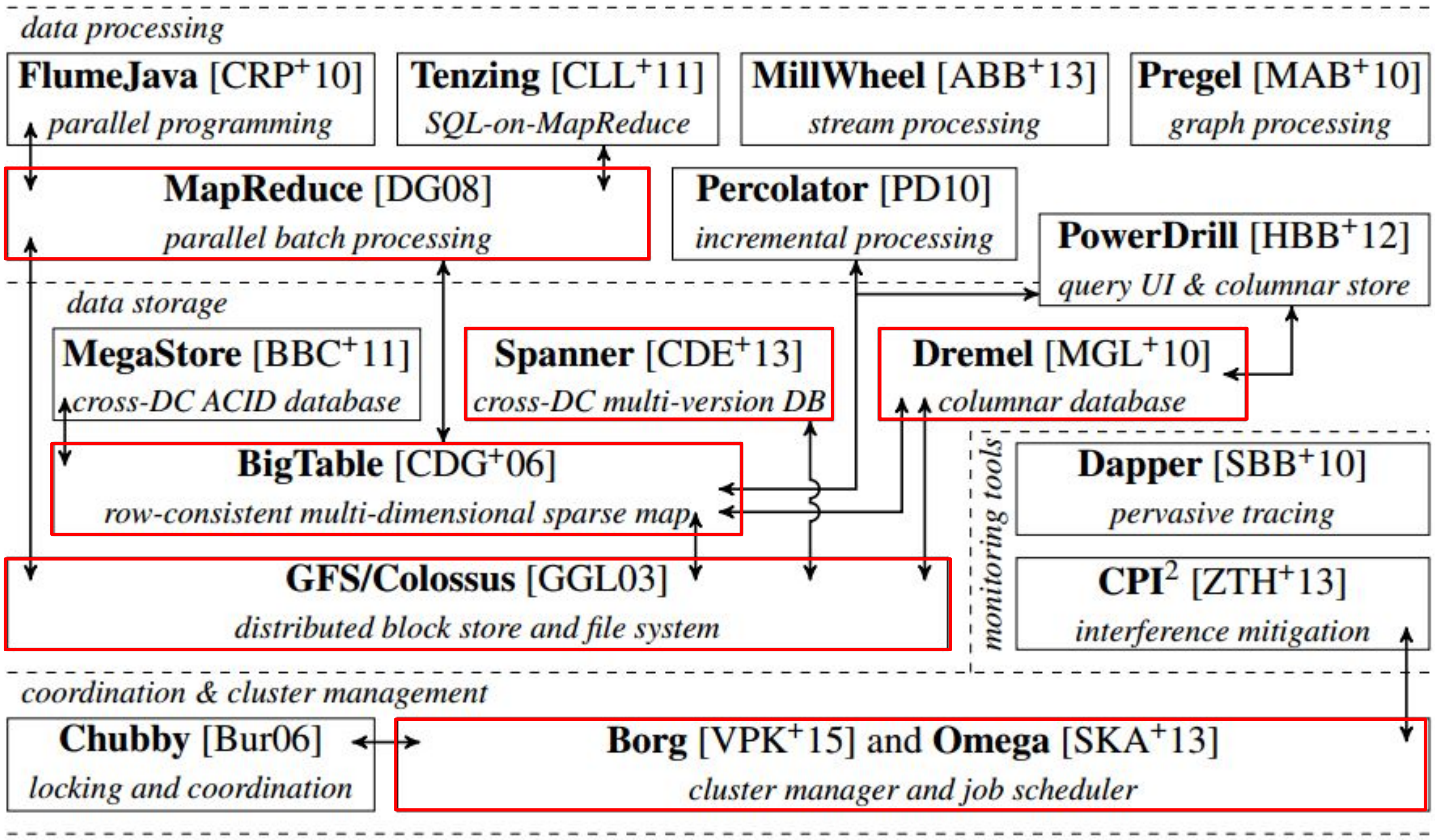


Figure from M. Schwarzkopf, “Operating system support for warehouse-scale computing”, PhD thesis, University of Cambridge, 2015 (to appear).

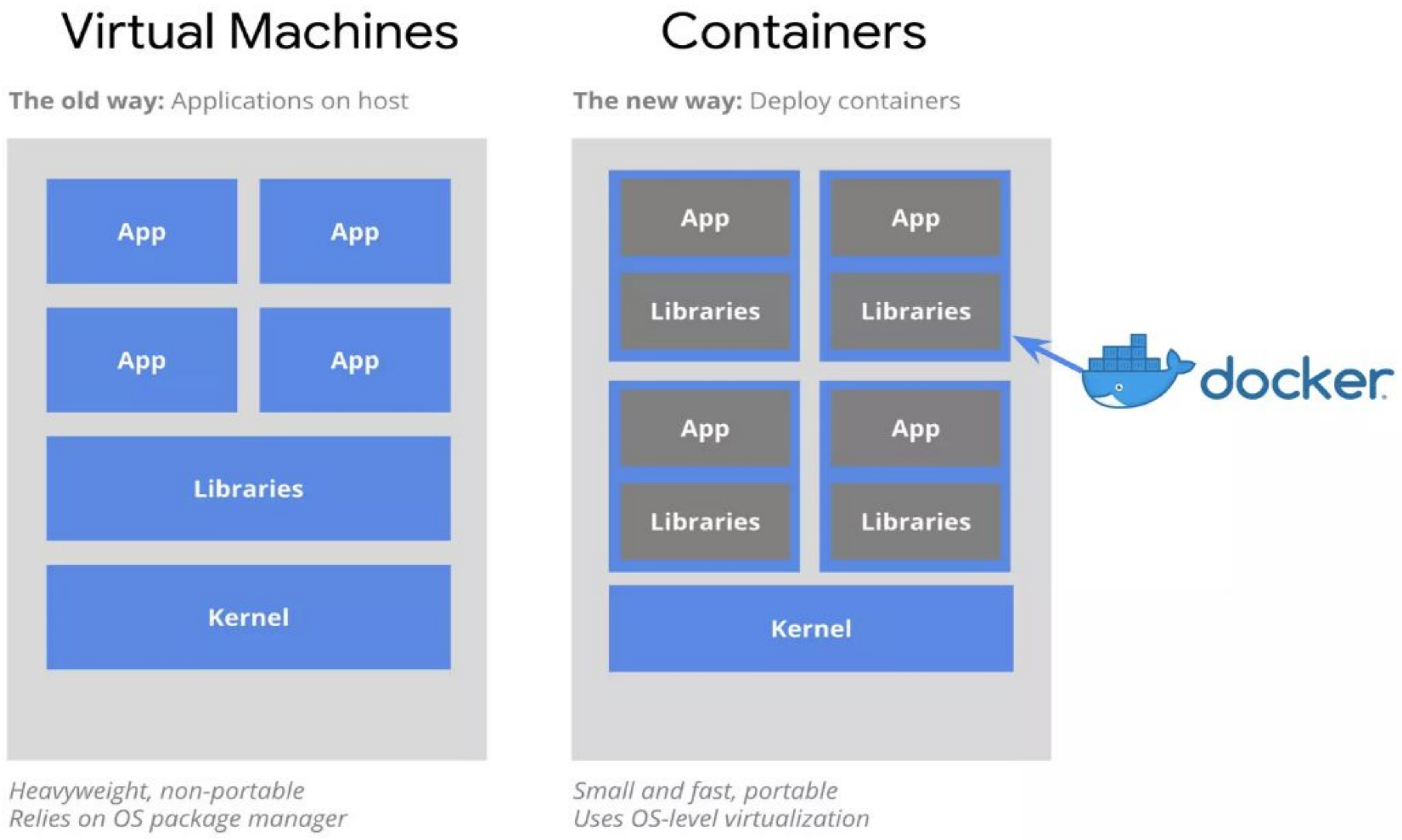
Example Infrastructure System: Kubernetes Cluster Orchestrator

Kubernetes (K8s)

https://kubernetes.io/

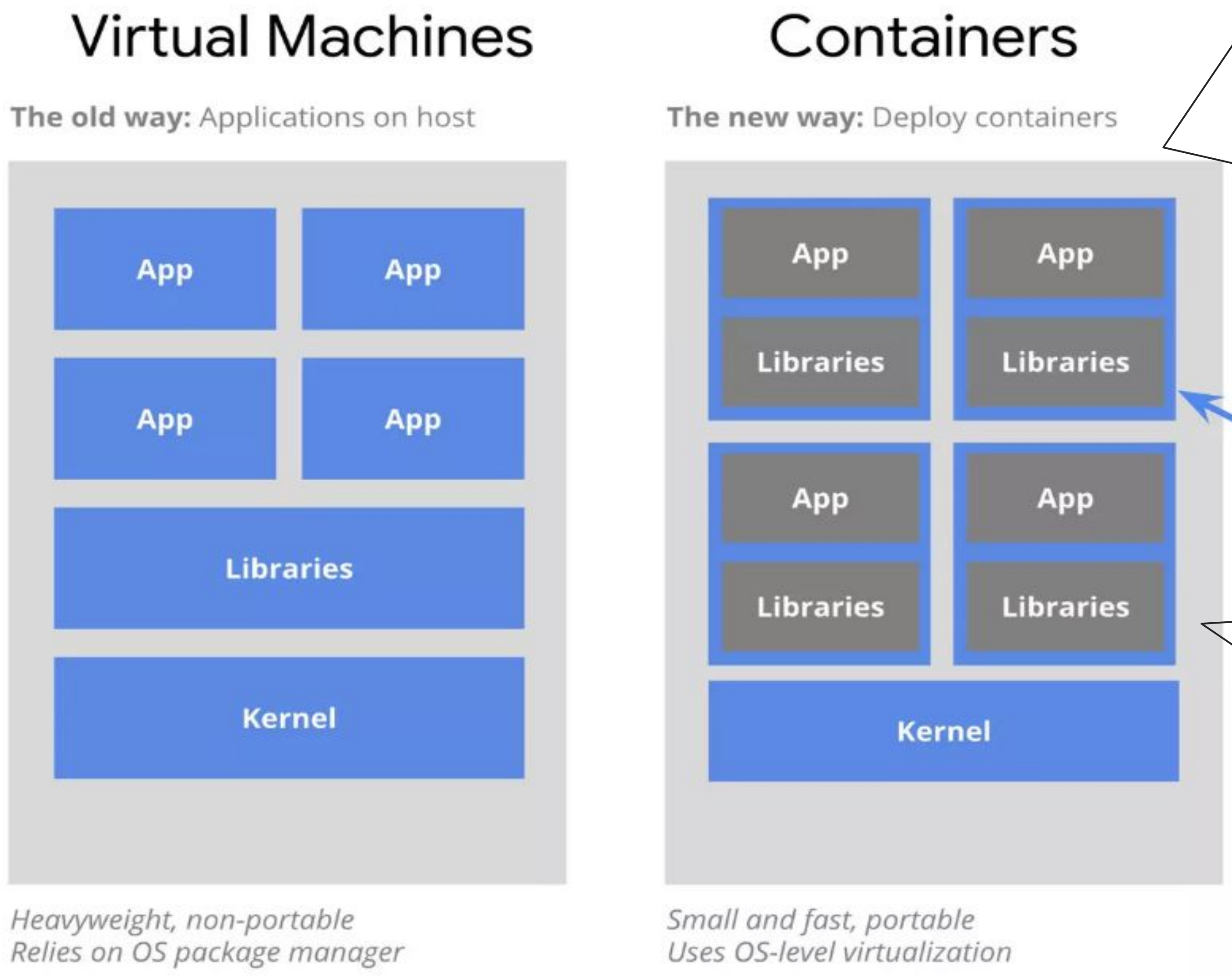
- Open-source system for automating deployment, scaling, and management of containerized applications.
- Groups containers that make up an application into logical units for easy management, scaling, and discovery.
- Builds upon 15 years of experience of running production workloads at Google, combined with best-of-breed ideas and practices from the community.

Background: Containers


Virtual Machines
The old way: Applications on host
App
App
App
App
Libraries
Kernel
Heavyweight, non-portable
Relies on OS package manager
Containers
The new way: Deploy containers
App
Libraries
App
Libraries
App
Libraries
App
Libraries
Kernel
docker
Small and fast, portable
Uses OS-level virtualization

source: kubernetes.io

Background: Containers


Virtual Machines
The old way: Applications on host
App
App
App
App
Libraries
Kernel
Heavyweight, non-portable
Relies on OS package manager
Containers
The new way: Deploy containers
App
App
Libraries
Libraries
App
App
Libraries
Libraries
Kernel
Small and fast, portable
Uses OS-level virtualization

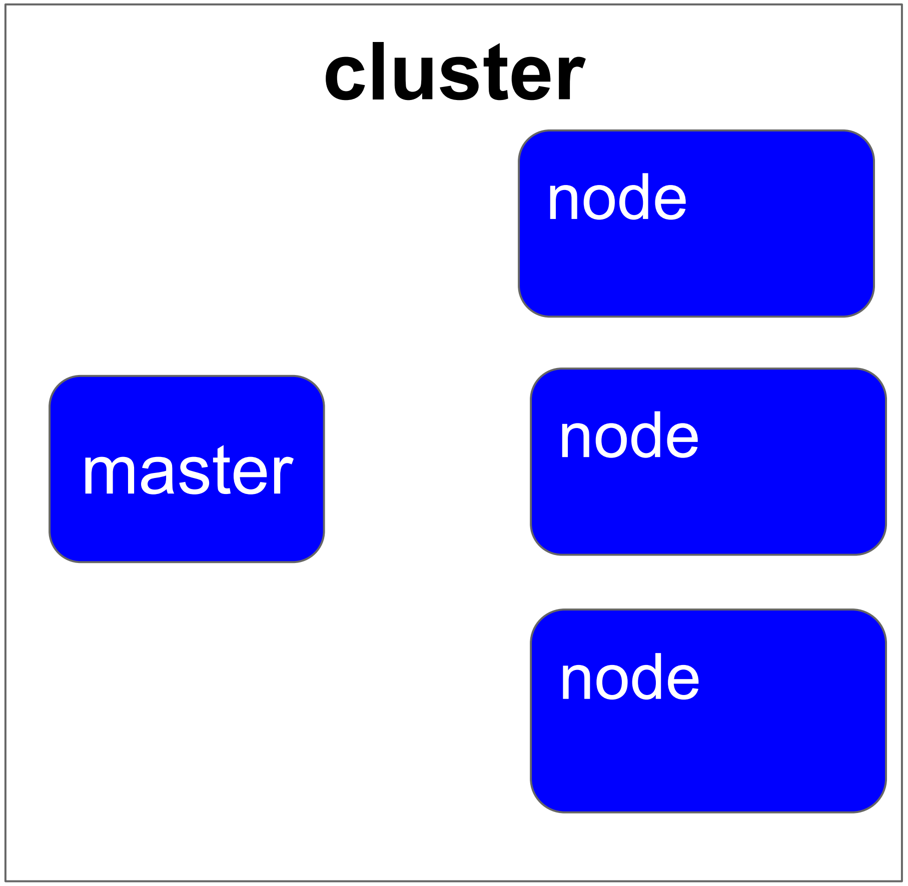
Google's been running this way for years! >2B containers launch per week (2019 source)!

Gmail, Search, Maps, Docs, …

source: kubernetes.io

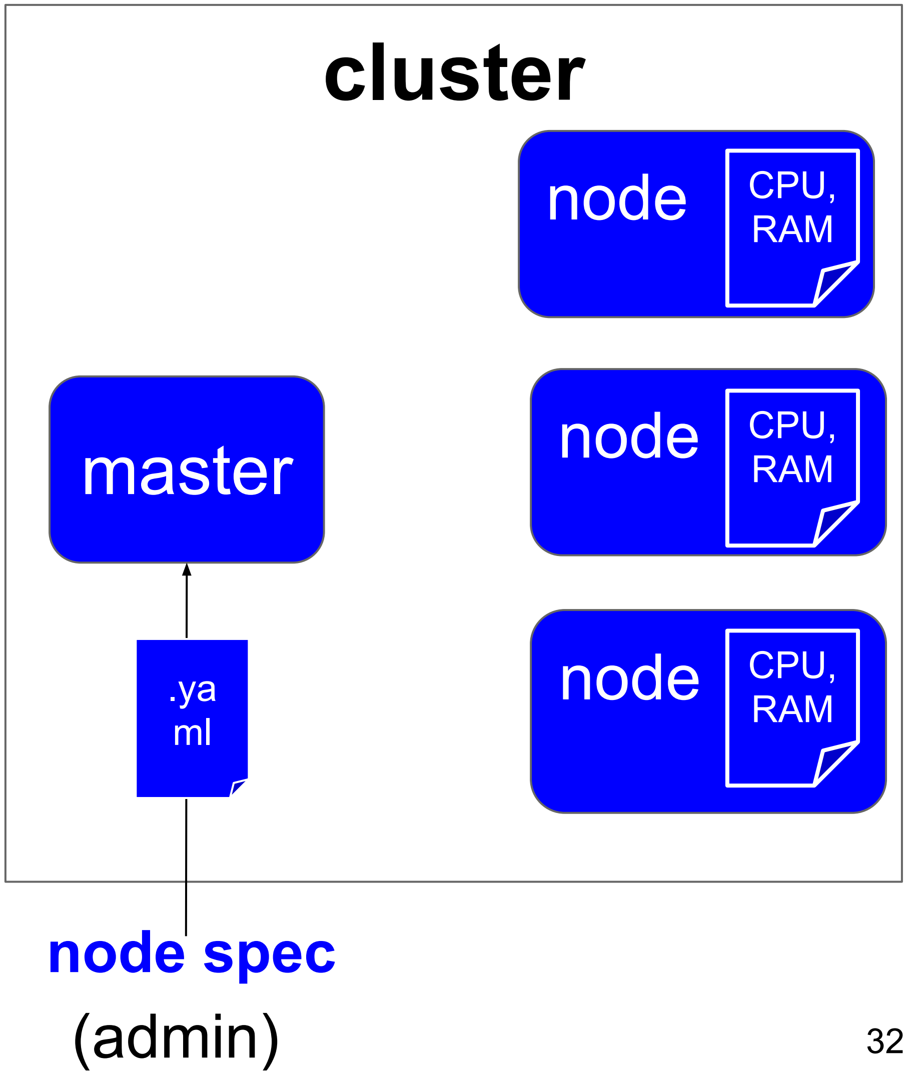
Key concepts

- K8s runs applications in a **cluster** of **nodes.**



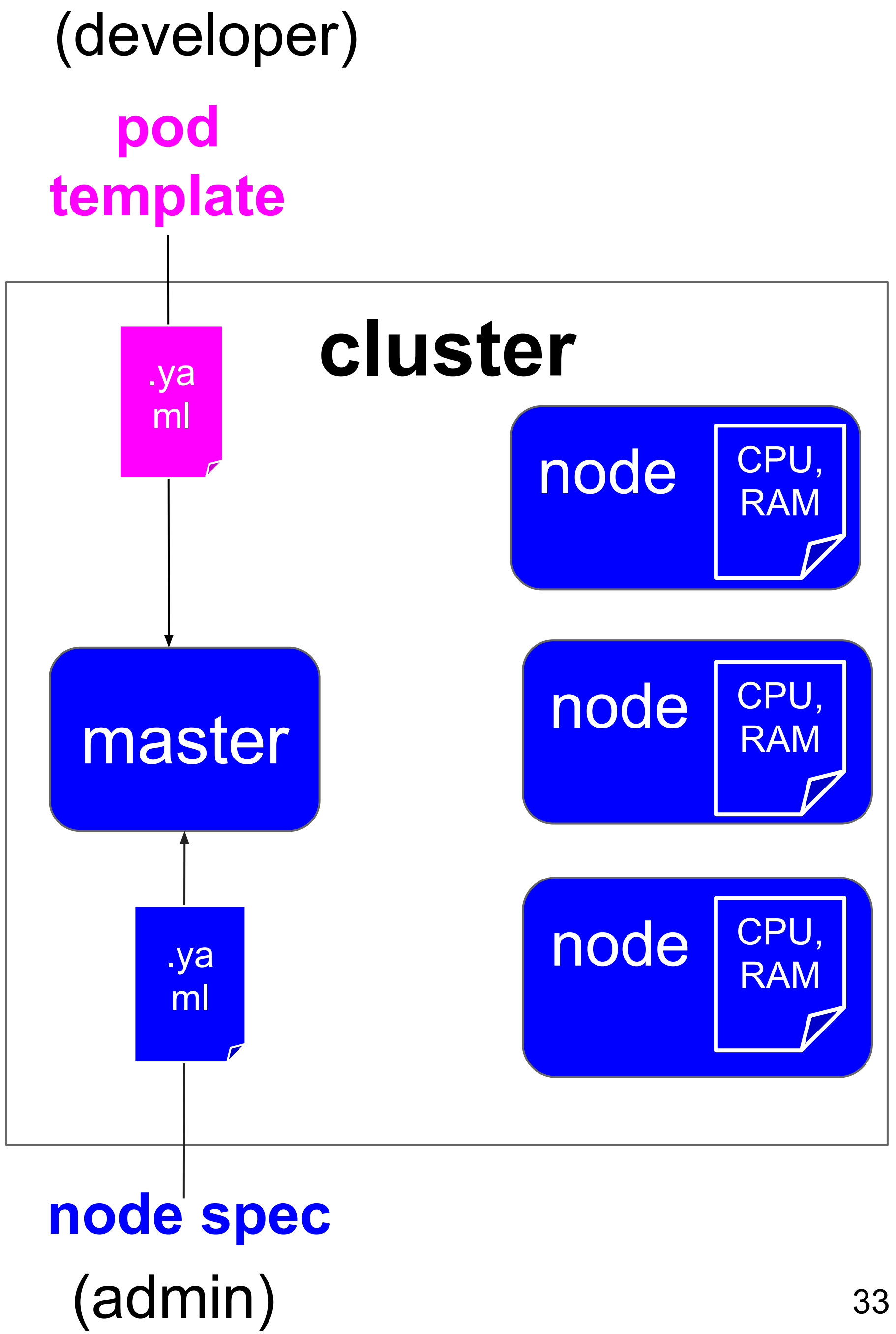
Key concepts

- K8s runs applications in a **cluster** of **nodes**.
- **Nodes** abstract out computing resources: can be physical machines or VMs; they are registered with specified amts of CPU, RAM, GPU, …



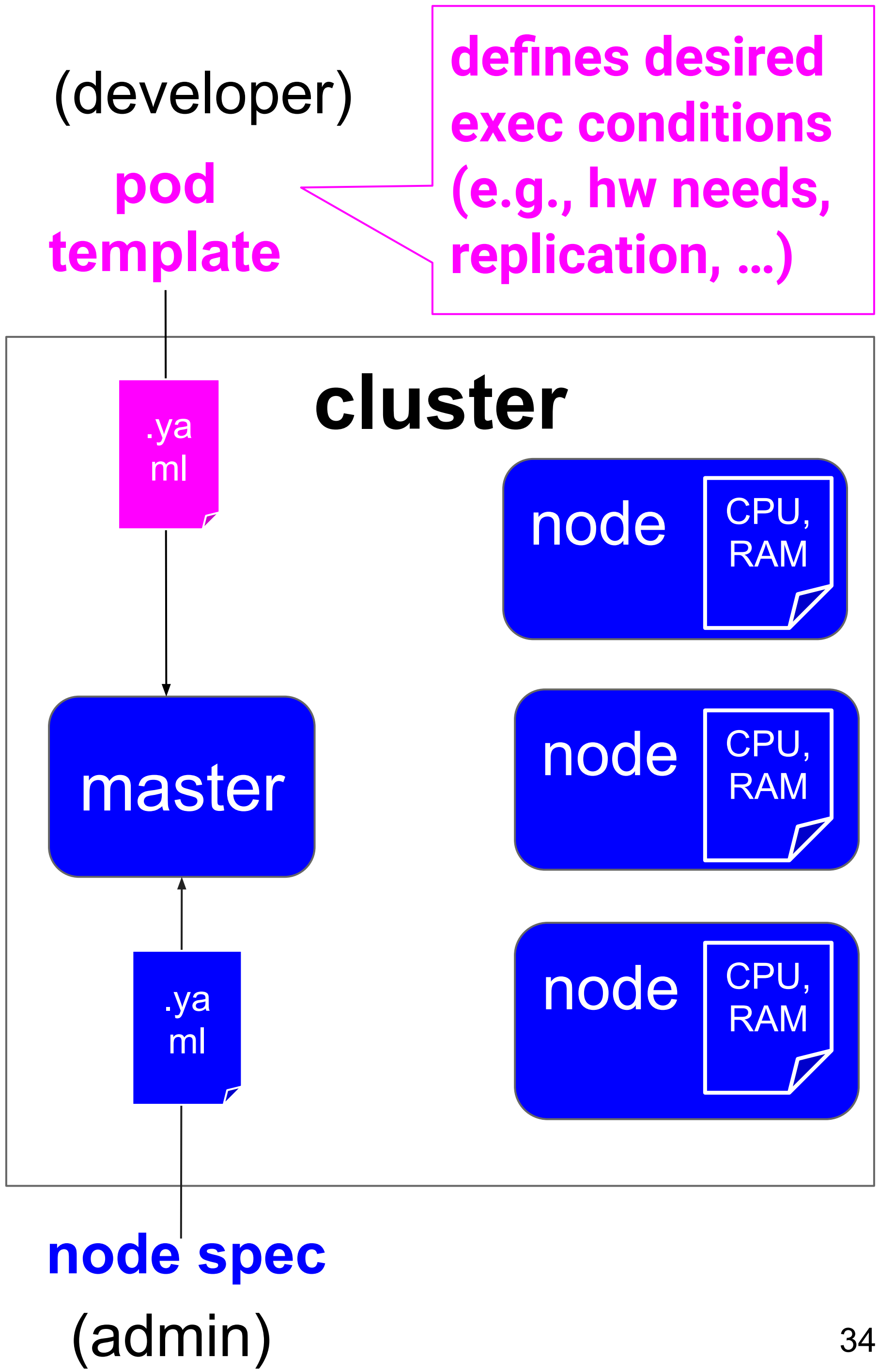
Key concepts

- K8s runs applications in a **cluster** of **nodes.**
- **Nodes** abstract out computing resources: can be physical machines or VMs; they are registered with specified amts of CPU, RAM, GPU, …
- Applications are called **pods** and consist of one or more containers, which the developer specifies in a .yaml file to k8s master.



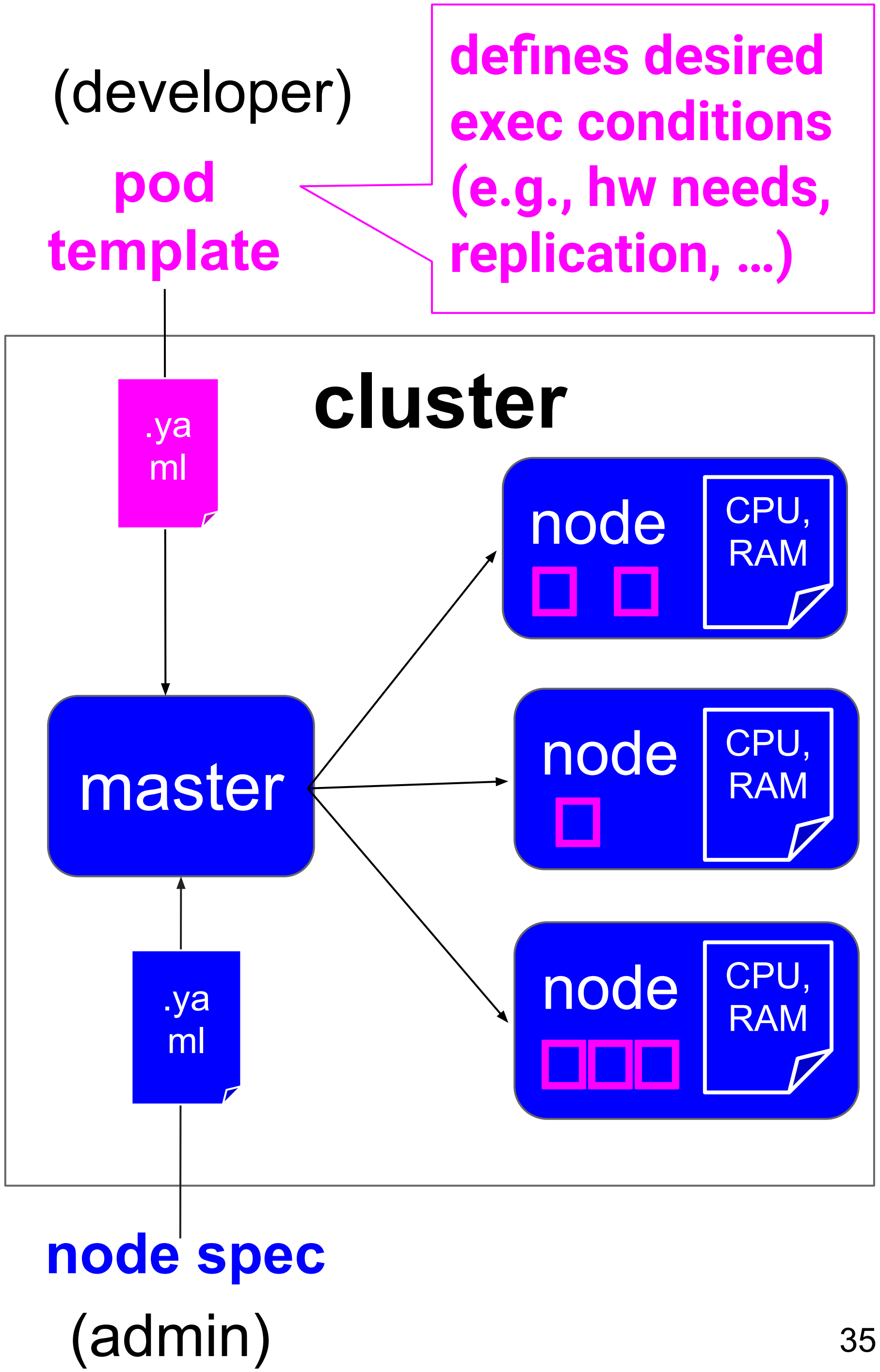
Key concepts

- K8s runs applications in a **cluster** of **nodes.**
- **Nodes** abstract out computing resources: can be physical machines or VMs; they are registered with specified amts of CPU, RAM, GPU, …
- Applications are called **pods** and consist of one or more containers, which the developer specifies in a .yaml file to k8s master.



K8s main functions

- Based on **pod templates**, selects suitable **nodes** and instantiates **pods** on them for execution.
- Continuously does that to ensure that, despite failures, the desired execution conditions for all pods are met.



Many more K8s functions

(from https://kubernetes.io/)

- Automated rollouts and rollbacks
- Service discovery and load balancing
- Storage orchestration
- Self-healing
- Automatic scheduling (bin packing)
- Secret and configuration management
- Batch execution
- Horizontal auto-scaling
- **Designed for extensibility**

Many more K8s functions

(from https://kubernetes.io/)

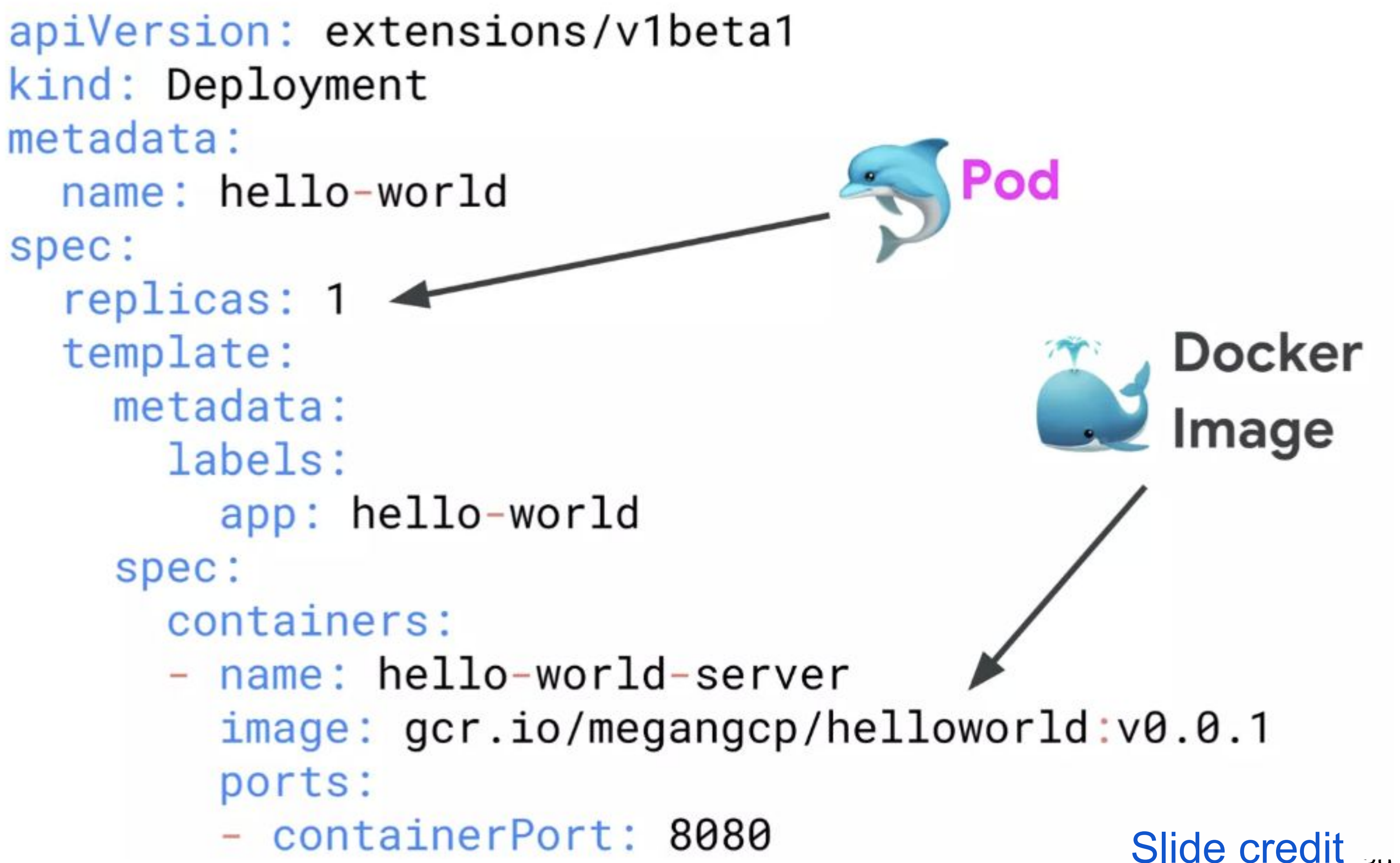
- Automated rollouts and rollbacks
- Service discovery and load balancing
- Storage orchestration
- Self-healing
- Automatic scheduling (bin packing)
- Secret and configuration management
- Batch execution
- Horizontal auto-scaling
- **Designed for extensibility**

EVEN MORE functions have been built outside of K8s, through its extension by third parties, demonstrating the value of extensible design for infra systems!

K8s outline

- Examples
  - Hello World
  - [Busybox]
  - [Nginx]
- System architecture (how it works)
- Extensibility
  - Argo workflows
  - Kubeflow pipelines
  - Ray on Kubernetes

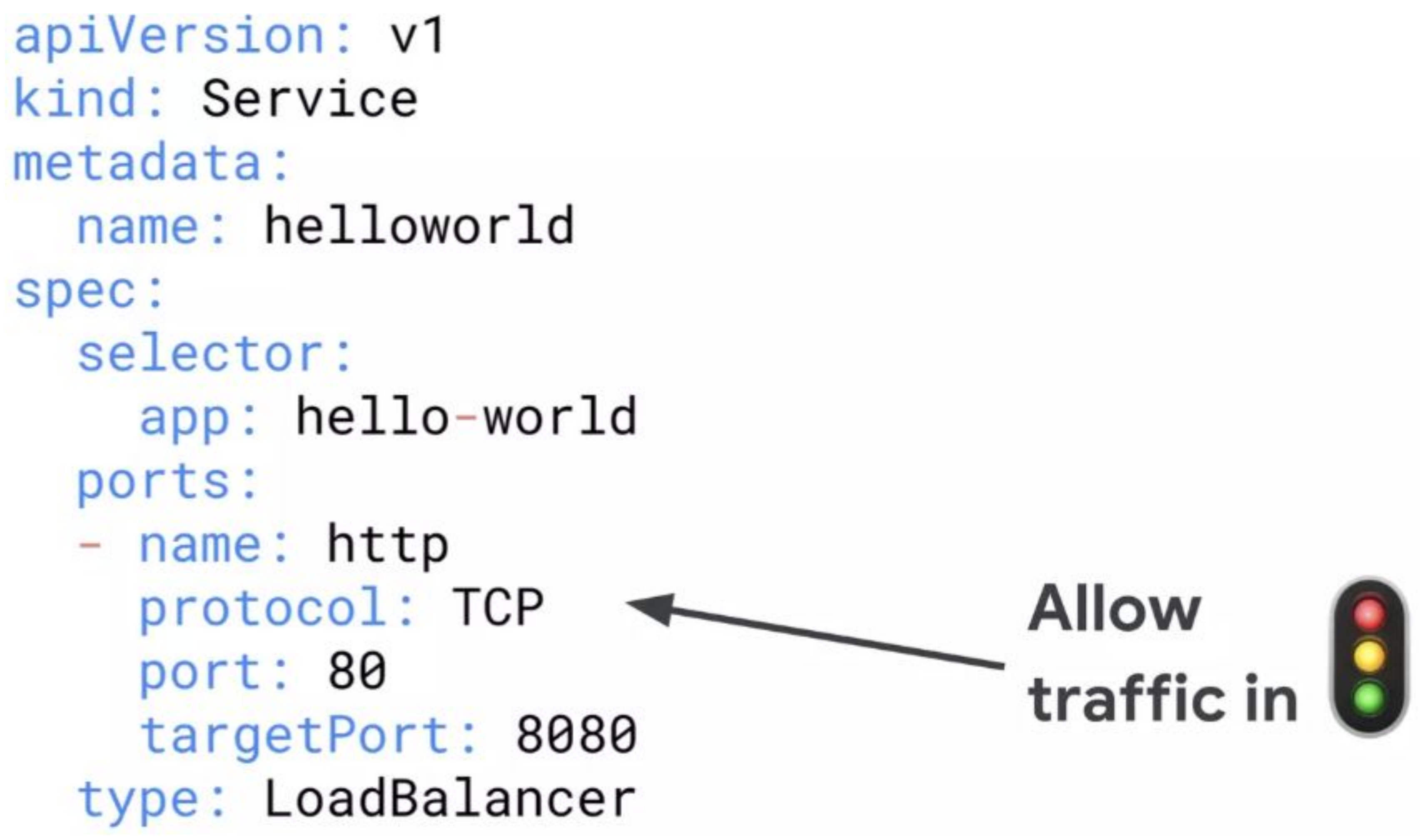
Example: Hello World



```yaml
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: hello-world
spec:
  replicas: 1
  template:
    metadata:
      labels:
        app: hello-world
    spec:
      containers:
      - name: hello-world-server
        image: gcr.io/megangcp/helloworld:v0.0.1
        ports:
        - containerPort: 8080
```

Slide credit

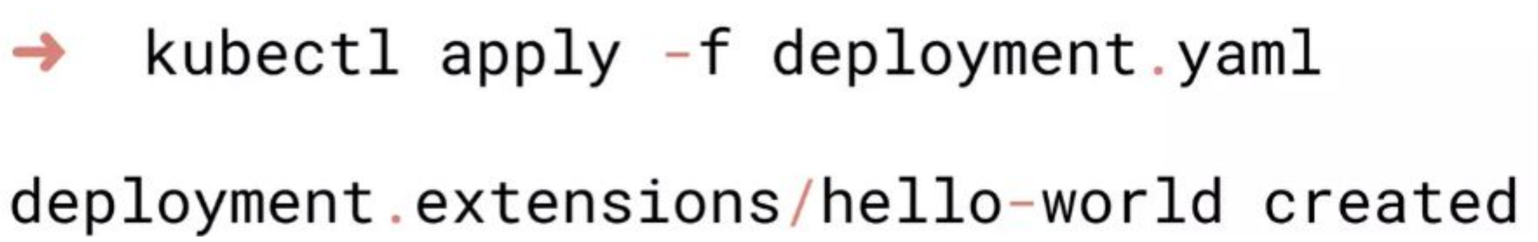
Example: Hello World (cont.)



```yaml
apiVersion: v1
kind: Service
metadata:
  name: helloworld
spec:
  selector:
    app: hello-world
  ports:
  - name: http
    protocol: TCP
    port: 80
    targetPort: 8080
  type: LoadBalancer
```

Slide credit

Example: Hello World (cont.)

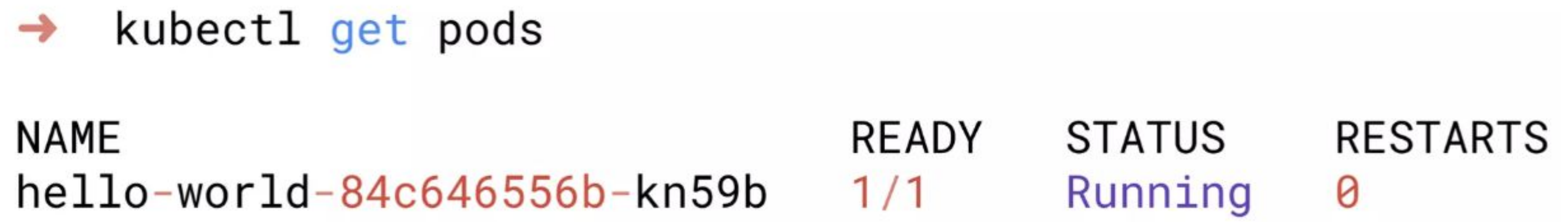


```
➜  kubectl apply -f deployment.yaml

deployment.extensions/hello-world created
```

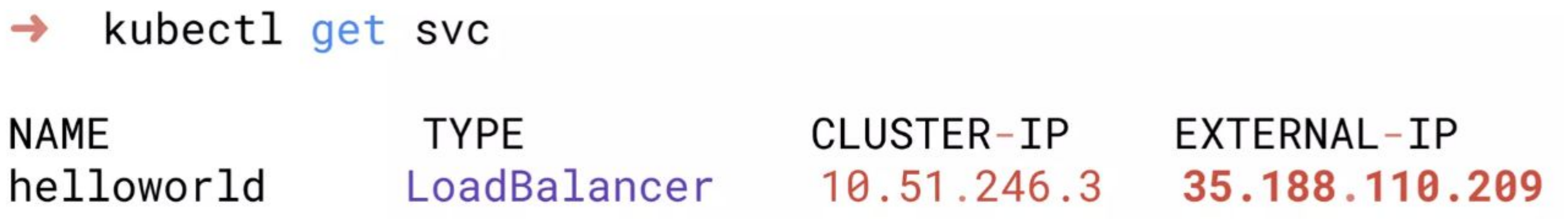
Slide credit

Example: Hello World (cont.)

After a while…



```
➜  kubectl get pods

NAME                              READY   STATUS    RESTARTS
hello-world-84c646556b-kn59b      1/1     Running   0
```

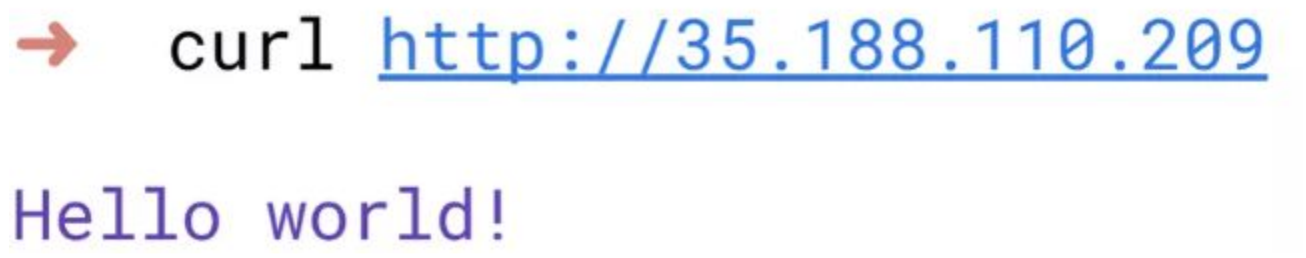


```
➜  kubectl get svc

NAME          TYPE           CLUSTER-IP     EXTERNAL-IP
helloworld    LoadBalancer   10.51.246.3    35.188.110.209
```

Slide credit

Example: Hello World (cont.)

After a while…



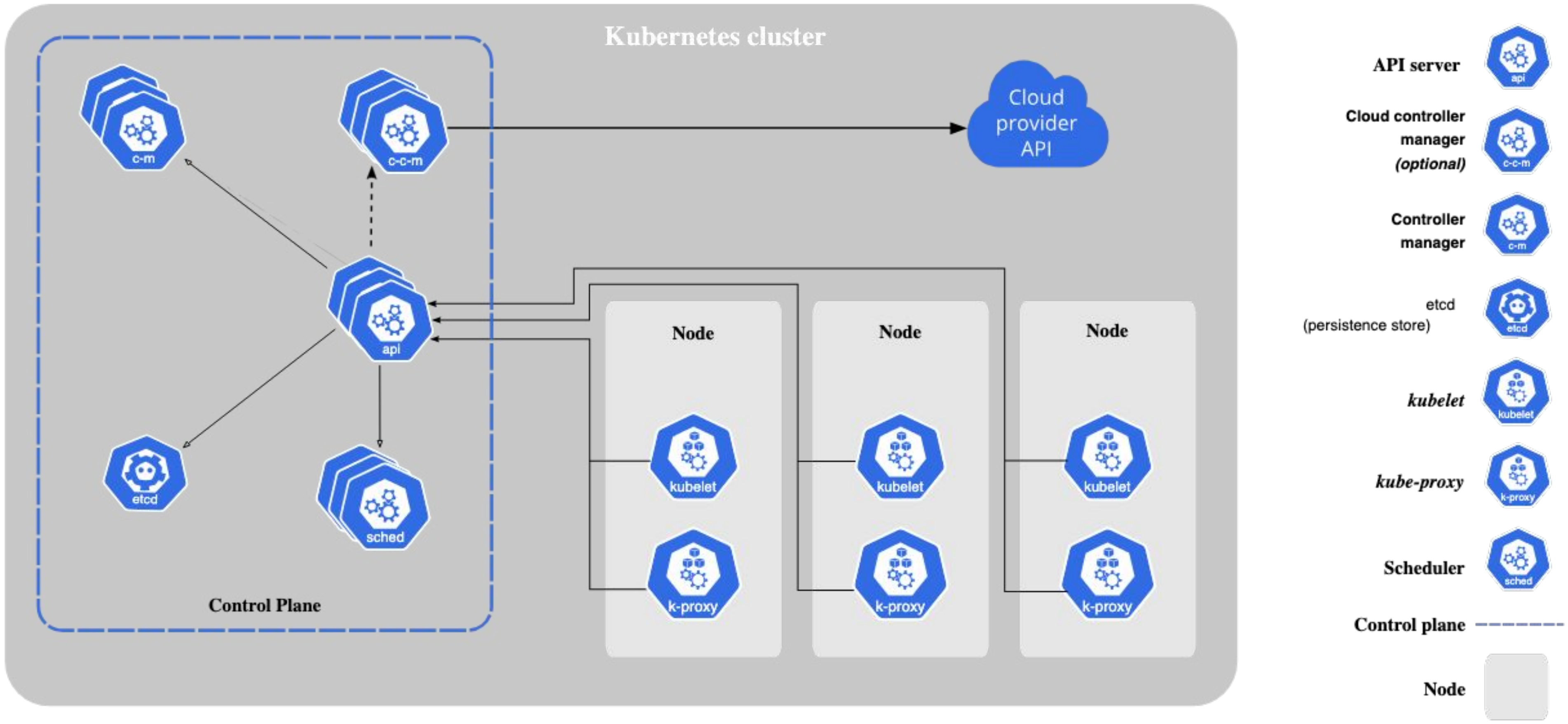
```
➜  curl http://35.188.110.209

Hello world!
```

Slide credit

K8s outline

- Examples
  - Hello World
  - Busybox (from doc)
  - Nginx (from doc)
- System architecture (how it works)
- Extensibility
  - Argo workflows
  - Kubeflow pipelines
  - Ray on Kubernetes

System architecture



Functionality detailed in: docs. RG describes the core aspects of the design.

K8s outline

- Examples
  - Hello World
  - Busybox (from doc)
  - Nginx (from doc)
- System architecture (how it works)
- Extensibility (from docs)
  - Argo workflows: overview, steps example, artifact passing example, dag example
  - Kubeflow pipelines: example
  - Ray on Kubernetes: docs

THE FOLLOWING SLIDES IN THIS PRESENTATION ARE NOT SUBJECT FOR THE EXAM.

The Google Stack

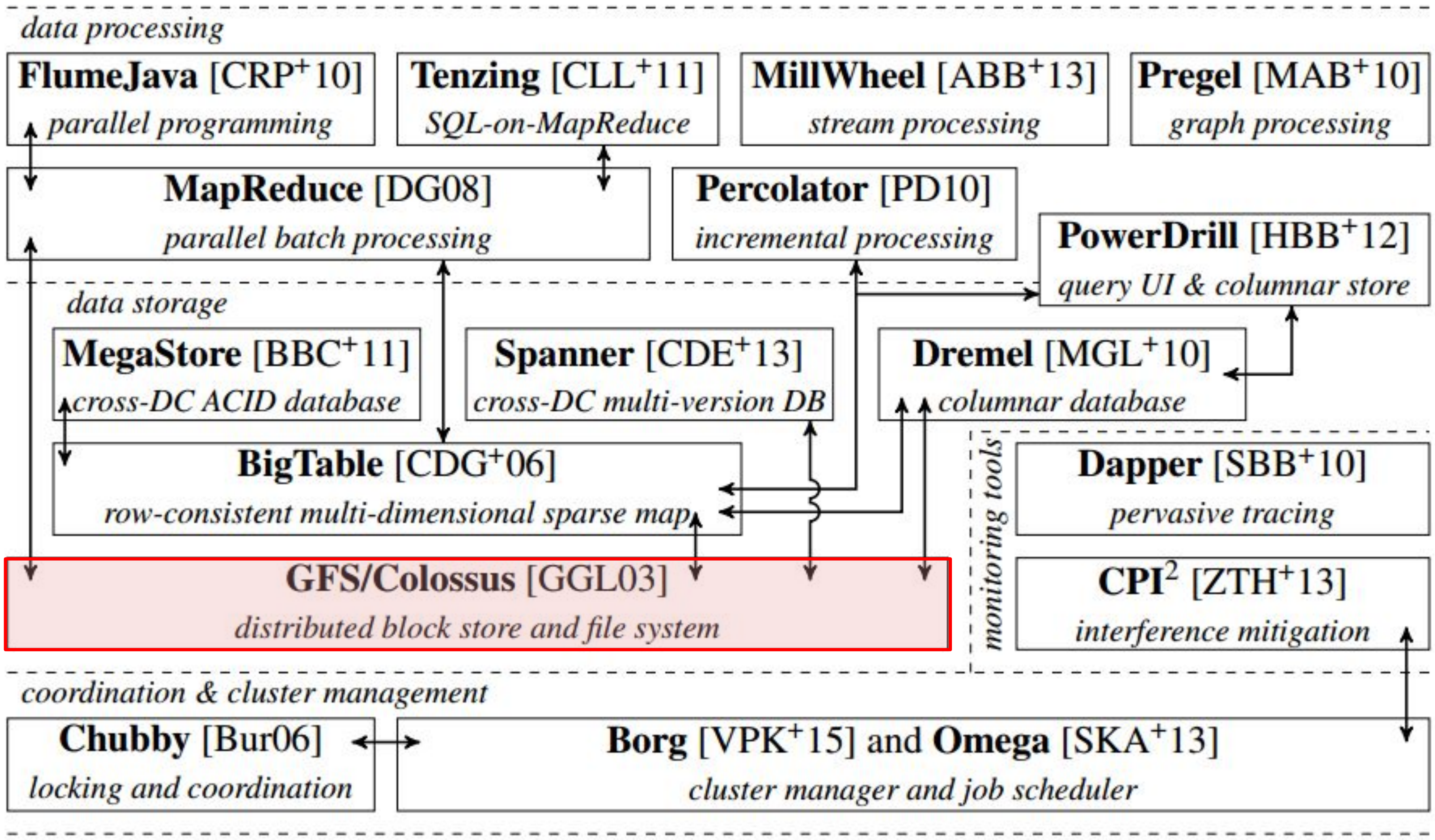
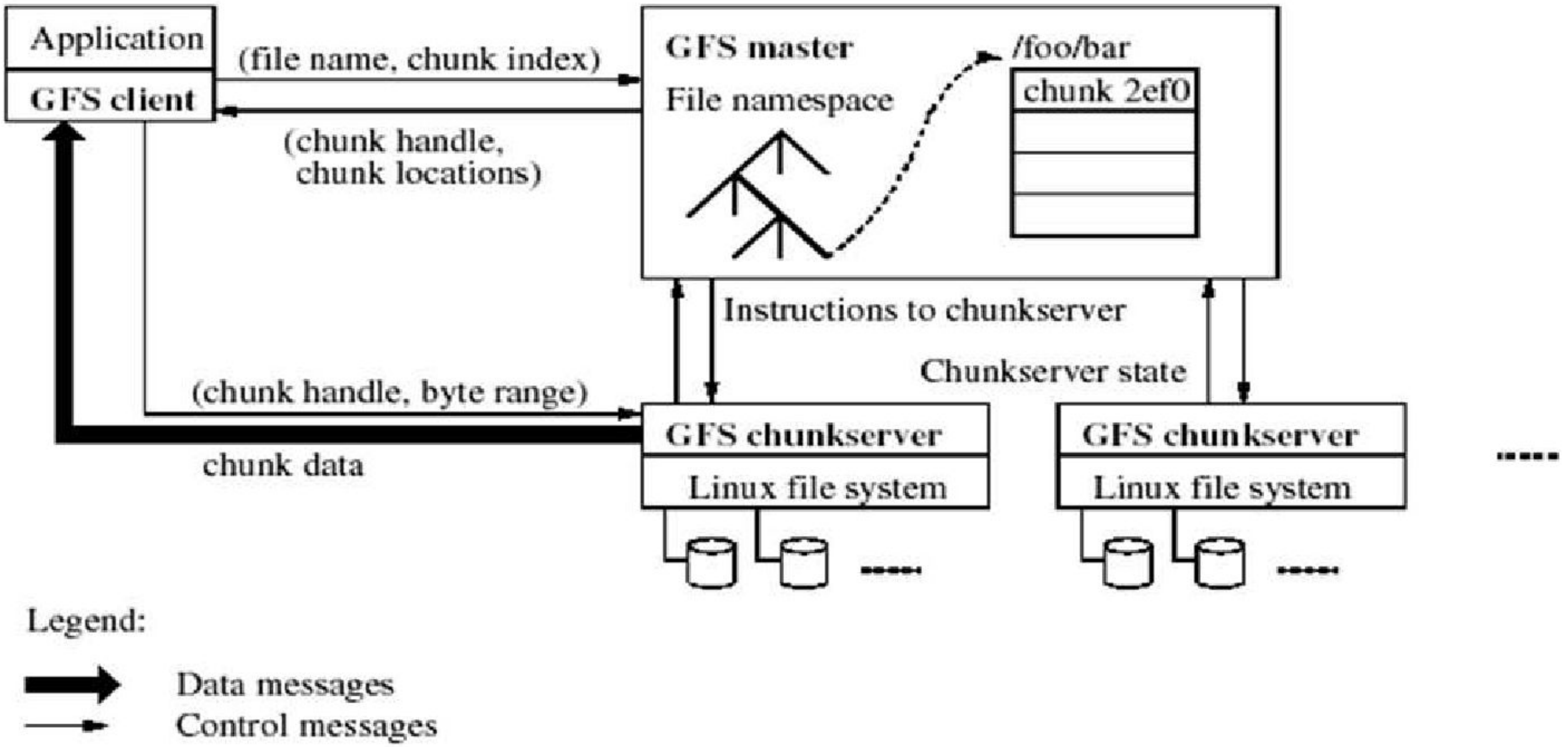


Figure from M. Schwarzkopf, "Operating system support for warehouse-scale computing", PhD thesis, University of Cambridge, 2015 (to appear).

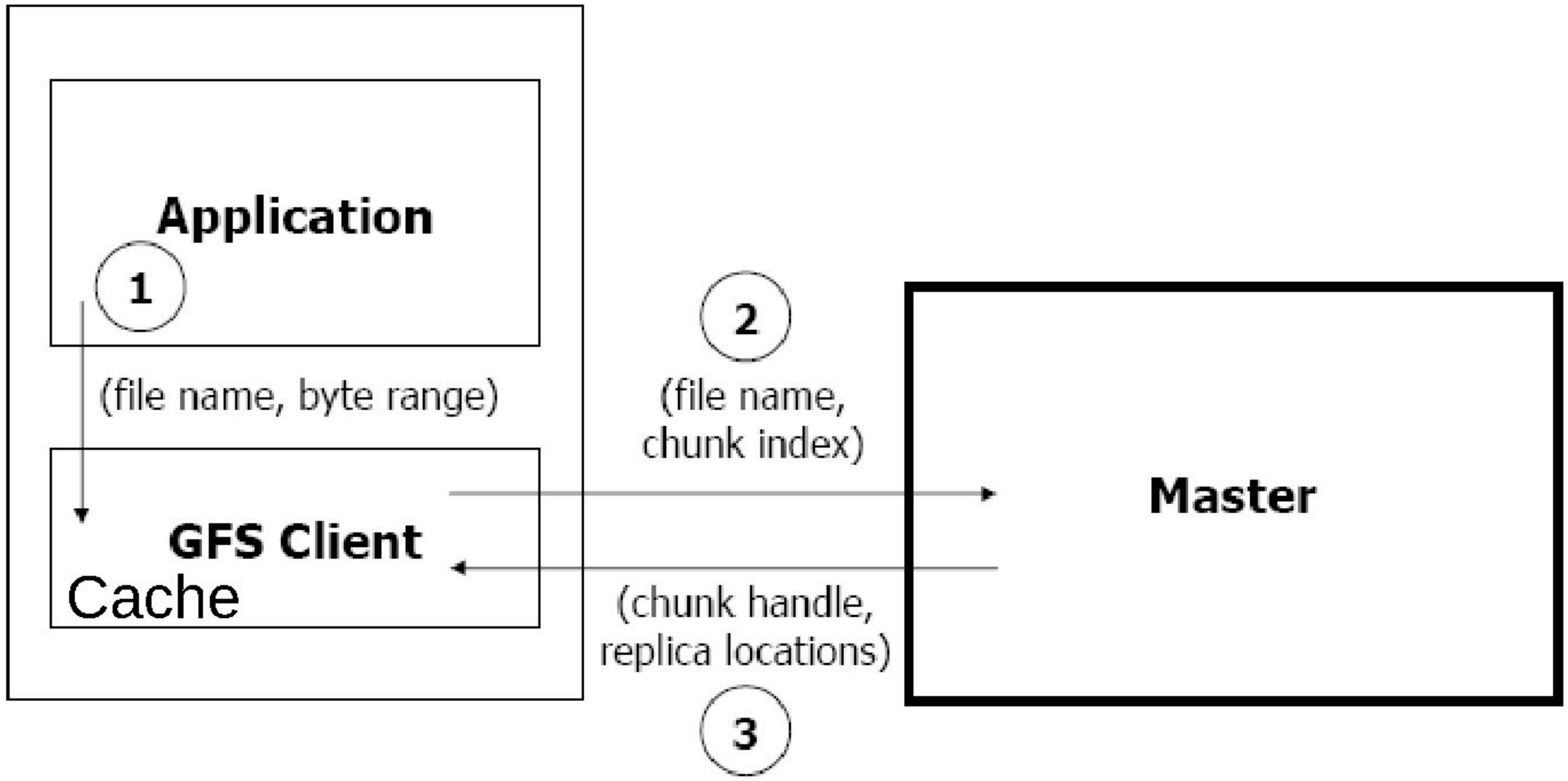
GFS/Colossus

- Bulk data block storage system
  - Optimized for large files (GB-size)
  - Supports small files, but not common case
  - **Read, write, record-append** modes
  - Record appends are the only one that gives clean semantics: **atomic append at least once.**
- **Colossus** = GFSv2, adds some improvements
  - e.g., Reed-Solomon-based erasure coding
  - better support for latency-sensitive applications
  - **sharded meta-data** layer, rather than single master

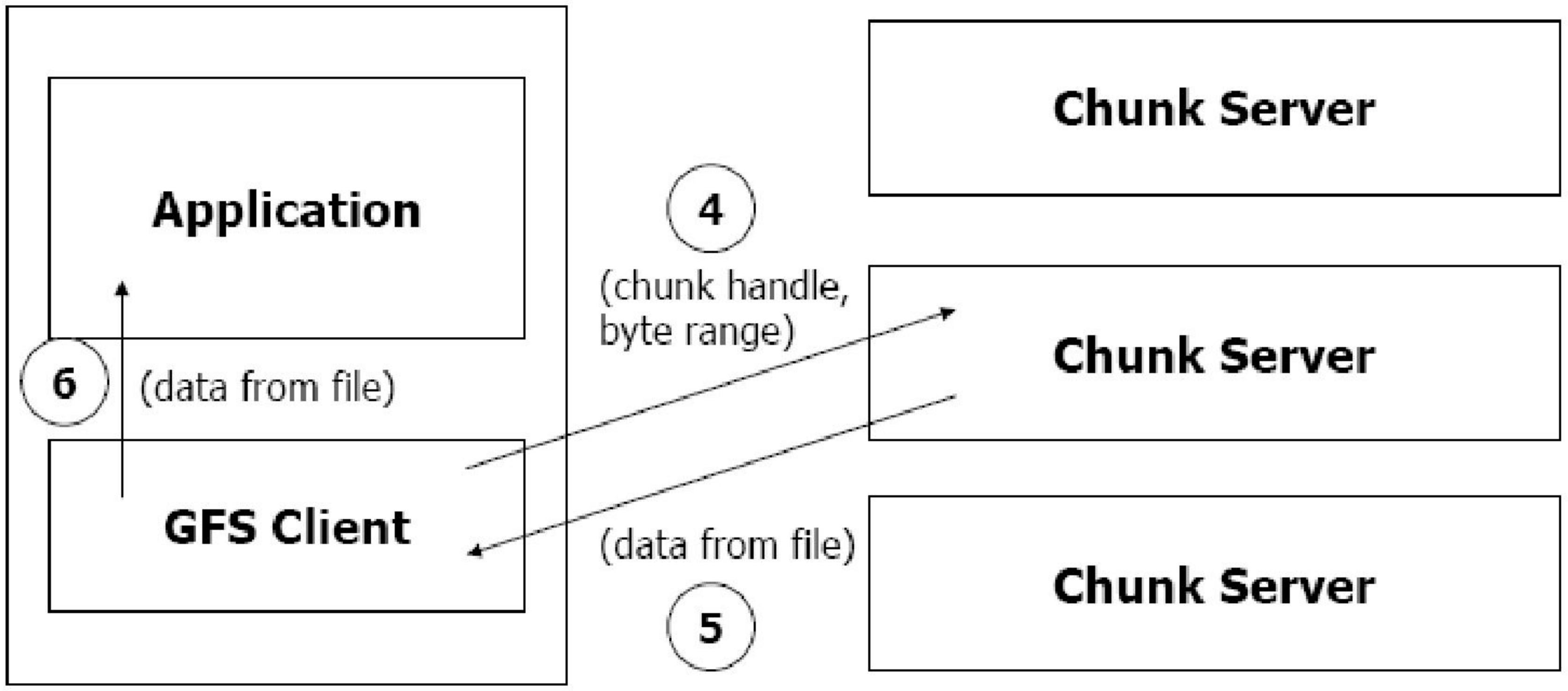
GFS/Colossus: architecture



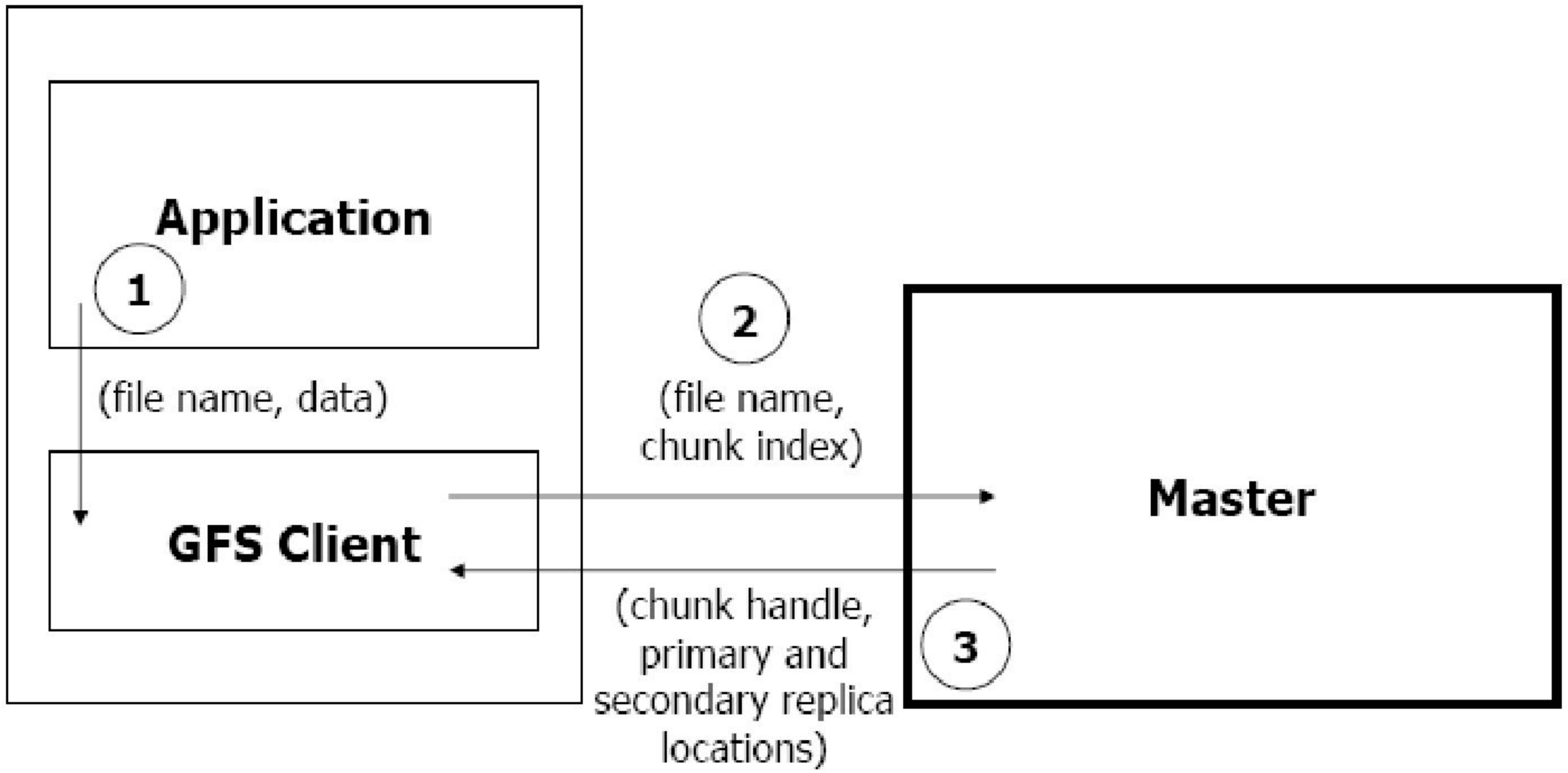
Read Protocol



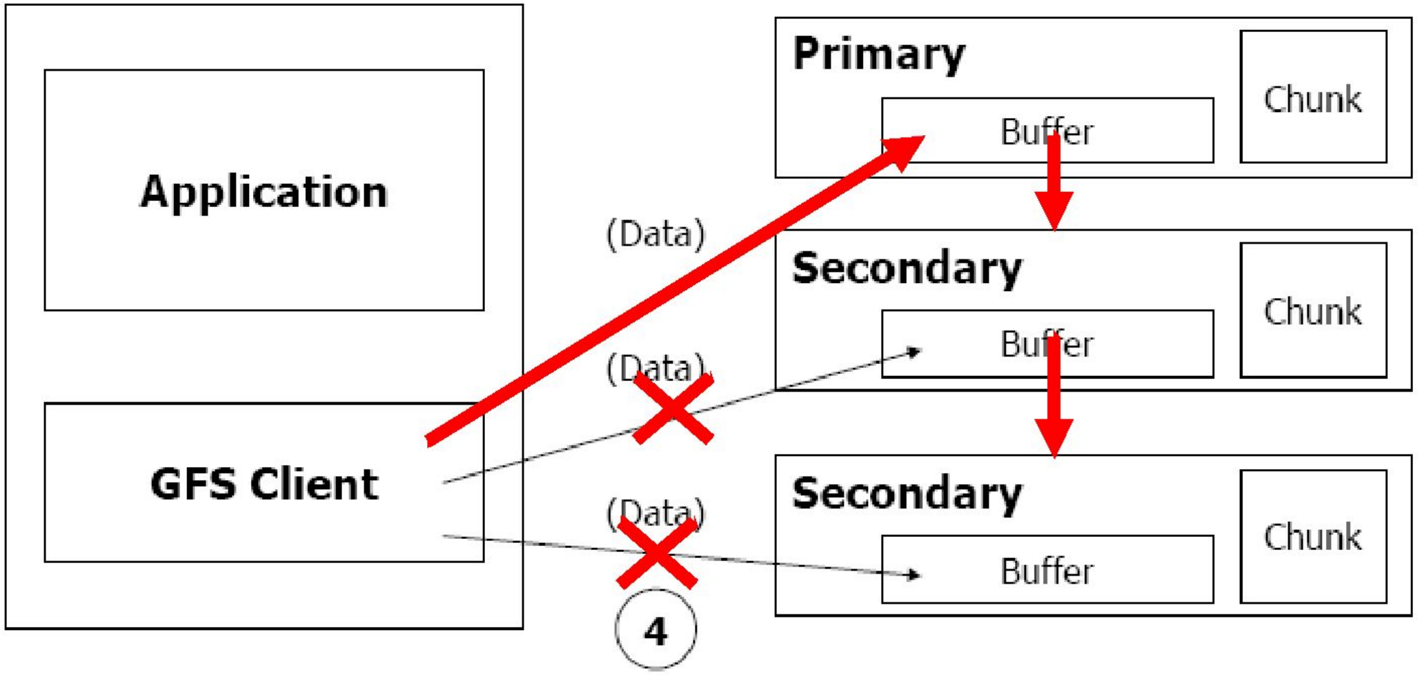
Read Protocol



Write Protocol



Write Protocol



Primary enforces one order across all writes to a file.
Thus, block writes are consistent but undefined in GFS.

Record Append Protocol

- The client specifies only the data, not the file offset
  - File offset is chosen by the primary
  - Why do they have this?

Record Append Protocol

- The client specifies only the data, not the file offset
  - File offset is chosen by the primary
  - Why do they have this?
- To provide meaningful semantic: at least once atomically
  - Because FS is not constrained Re: where to place data, it can get atomicity without sacrificing concurrency
- Rough mechanism:
  - If record fits in chunk, primary chooses the offset and communicates it to all replicas *offset is arbitrary*
  - If record doesn’t fit in chunk, the chunk is padded and client gets failure *file may have blank spaces*
  - If a record append fails at any replica, the client retries the

Detailed algo

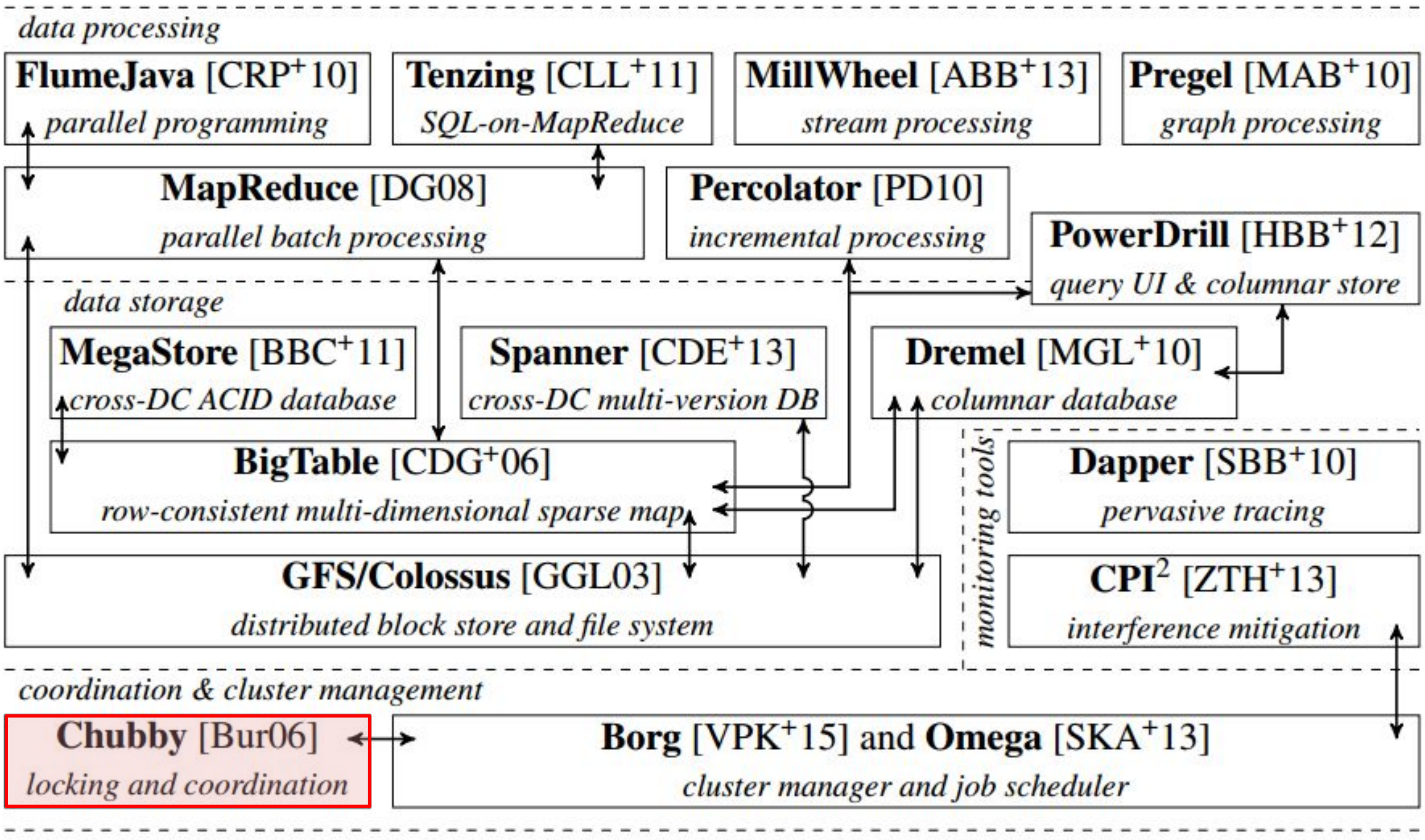
Application originates record append request.

2. GFS client translates request and sends it to master.
3. Master responds with chunk handle and (primary + secondary) replica locations.
4. Client pushes write data to all locations.
5. Primary checks if record fits in specified chunk.
6. If record does not fit, then:
    - The primary pads the chunk, tells secondaries to do the same, and informs the client.
    - Client then retries the append with the next chunk.
7. If record fits, then the primary:
    - appends the record at some offset in chunk,
    - tells secondaries to do the same (specifies offset),
    - receives responses from secondaries,

Implications of weak semantics

- Relying on appends rather on overwrites
- Writing self-validating records
  - Checksums to detect and remove *padding*
- Self-identifying records
  - Unique Identifiers to identify and discard *duplicates*
- Hence, applications need to adapt to GFS and be aware of its inconsistent semantics
- BUT: You can implement a (transaction) log replication protocol on it, so it’s a useful building block toward a stronger-semantic system.

The Google Stack



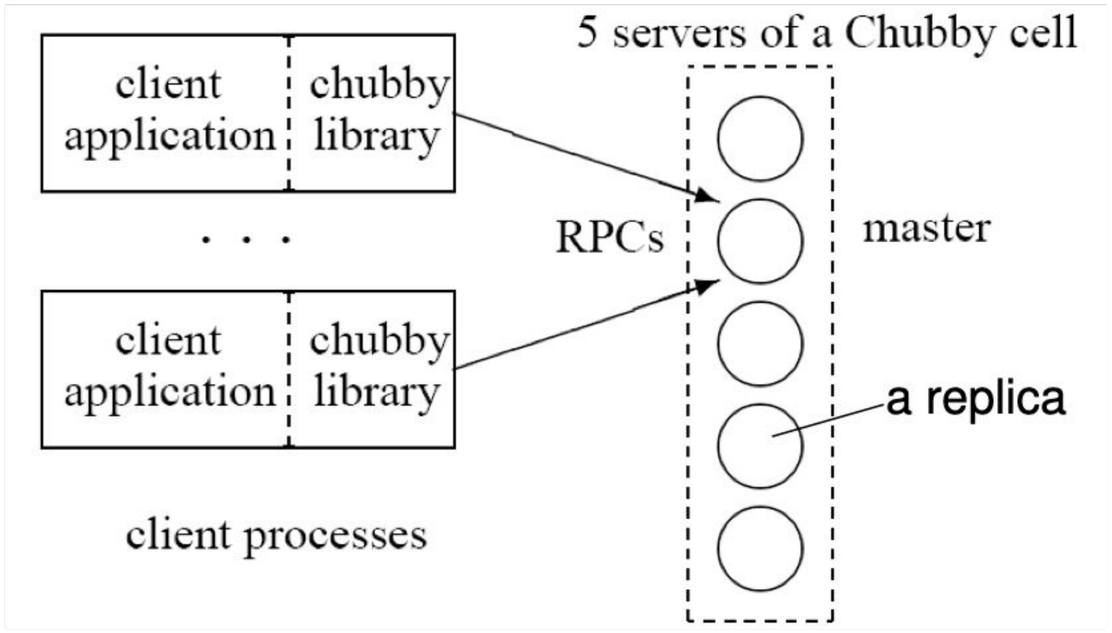
Details & Bibliography: http://malteschwarzkopf.de/research/assets/google-stack.pdf

Figure from M. Schwarzkopf, "Operating system support for warehouse-scale computing", PhD thesis, University of Cambridge, 2015.

Chubby (2004)

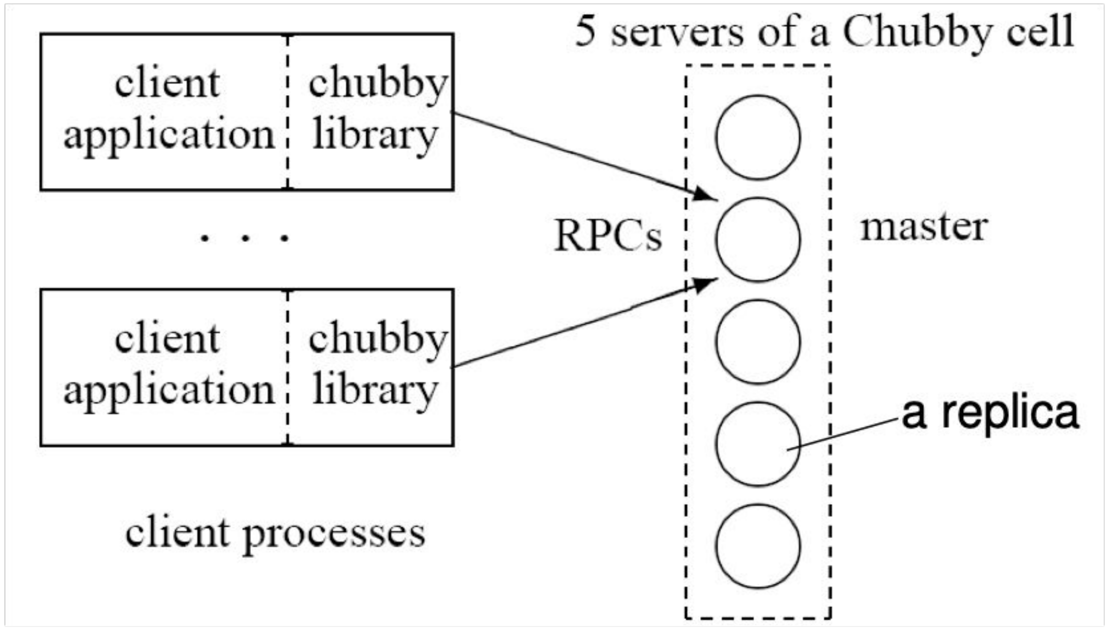
- Lock Service
- UNIX-like file system interface
- Reliability and availability
- Chubby uses Paxos for everything
  - Propagate writes to a file
  - Choosing a Master
  - Even for adding new Chubby servers to a Chubby cell
- Used by many services at Google (Colossus, Bigtable)
- Open-source version is called Zookeeper, also used as building block in many systems

System Architecture



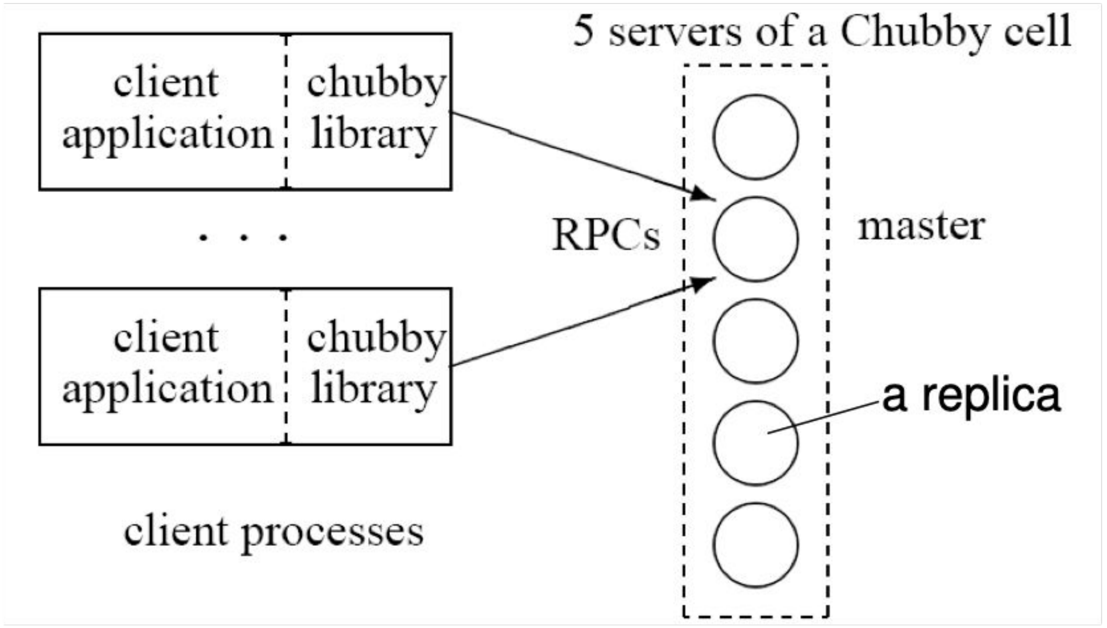
- A chubby cell consists of a small set of servers (replicas)
  - Placed in different racks, so as to minimize chance of correlated failures
- A master is elected from the replicas via Paxos
  - Master lease: several seconds
  - If master fails, a new one will be elected, but only after master leases expire
- Client talks to the master via the chubby library
  - All replicas are listed in DNS; clients discover master by talking to any replica

System Architecture (2)



- Replicas maintain copies of a simple database
- Clients send read/write requests only to the master
- For a write:
  - The master propagates it to replicas via Paxos
  - Replies after the write reaches a majority of replicas
- For a read:
  - The master satisfies the read alone

System Architecture (3)



- If a replica fails and does not recover for a long time (a few hours)
  - A fresh machine is selected to be a new replica, replacing the failed one
  - It updates the DNS
  - Obtains a recent copy of the database
  - The current master polls DNS periodically to discover new replicas
  - Integrating the new replica into the group is another Paxos run

Interface

- Supports a hierarchical namespace for lock files.
  - /ls/foo/OurPrimaryServer.lck
    - First component (ls): lock service (common to all names)
    - Second component (foo): the chubby cell (used in DNS lookup to find the Chubby master)
    - The rest: lock file name inside the cell
- Supports:
  - Atomic create, delete, atomic read of full contents, atomic write of full contents, etc.
  - Reader and writer locks
  - Clients can subscribe to events (modifications of Chubby

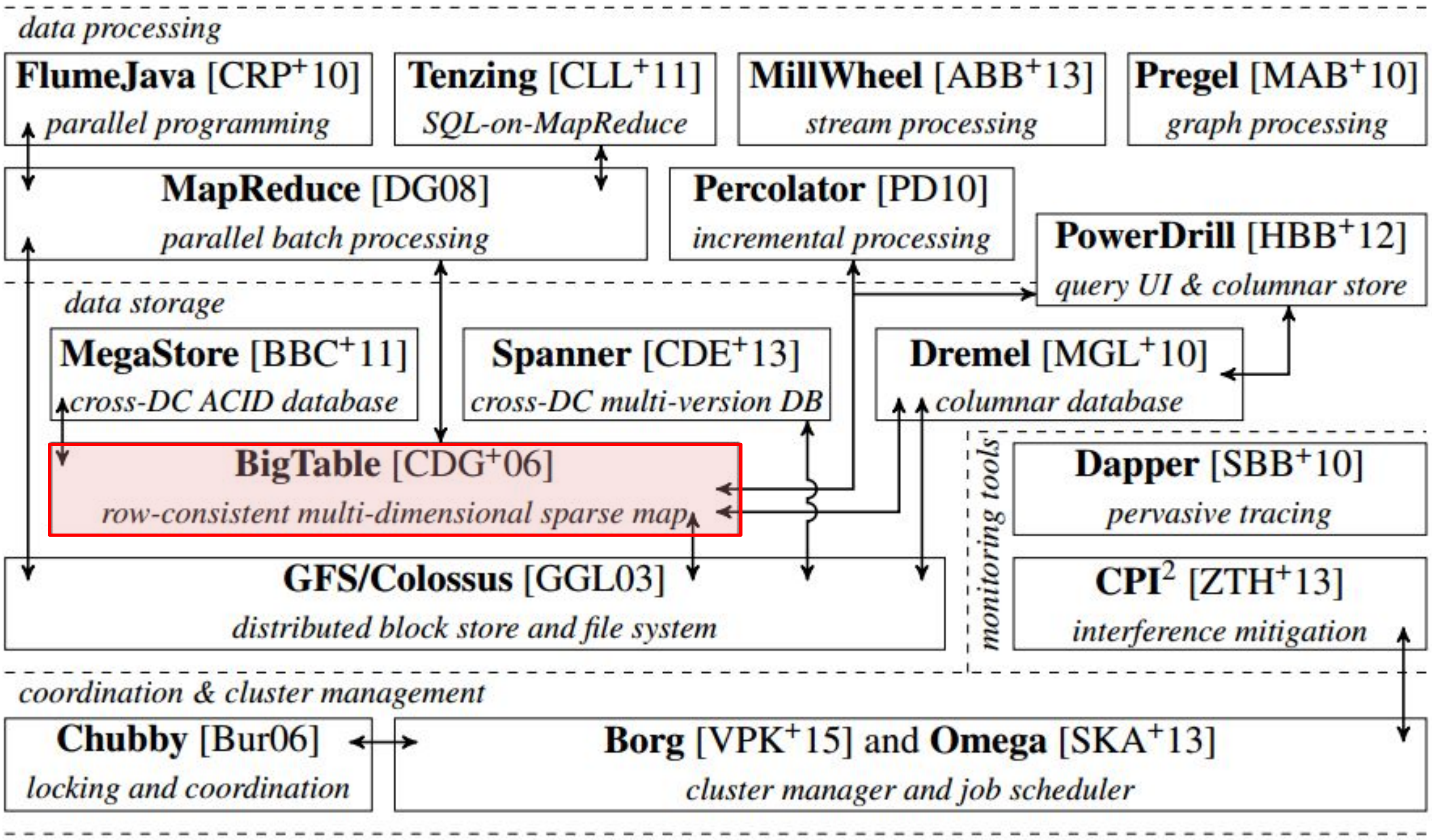
APIs

- Open()
  - Mode: read/write/change ACL; Events; Lock-delay
  - Create new file or directory?
- Close()
- GetContentsAndStat(), GetStat(), ReadDir()
- SetContents(): set all contents; SetACL()
- Delete()
- Locks: Acquire(), TryAcquire(), Release()
- Sequencers: GetSequencer(), SetSequencer(), CheckSequencer()

Example: Primary Election

```
Open("/ls/foo/OurServicePrimary", "write mode");
if (successful) {
    // primary
    SetContents(primary_identity);
} else {
    // replica
    Open("/ls/foo/OurServicePrimary", "read mode",
            "file-modification event");
    when notified of file modification:
            primary = GetContentsAndStat();
}
```

The Google Stack



Details & Bibliography: http://malteschwarzkopf.de/research/assets/google-stack.pdf

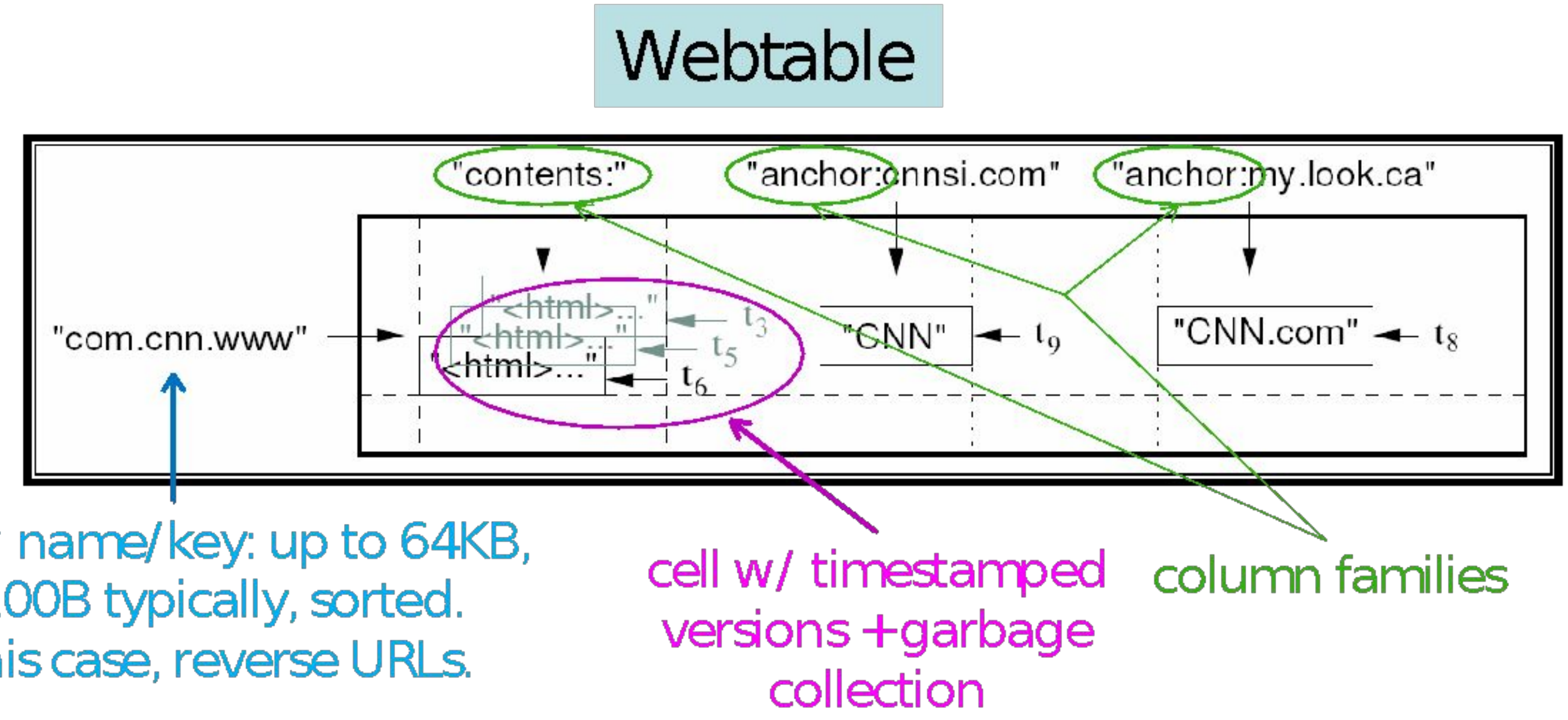
Figure from M. Schwarzkopf, "Operating system support for warehouse-scale computing", PhD thesis, University of Cambridge, 2015.

BigTable (2006)

- "A Bigtable is a sparse, distributed, persistent multi-dimensional sorted map"

  (row:string, column:string, timestamp:int64) → string

- Example: the (simplified) schema of the Webtable:

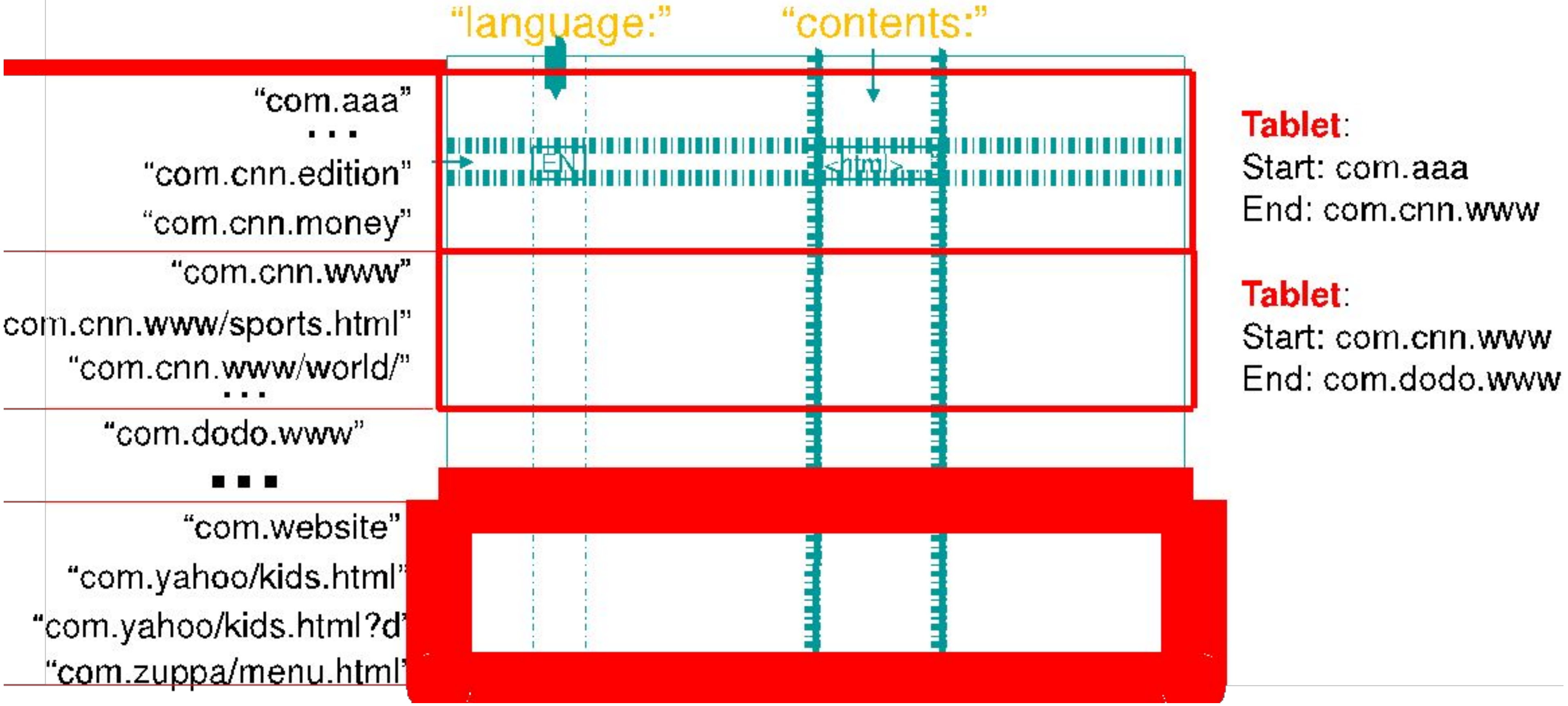

Webtable
"contents:"
"anchor:cnnsi.com"
"anchor:my.look.ca"
"com.cnn.www"
"<html>..."
t3
t5
t6
"CNN"
t9
"CNN.com"
t8
Row name/key: up to 64KB, 10-100B typically, sorted. In this case, reverse URLs.
cell w/ timestamped versions +garbage collection
column families

Key Ideas

- Distributed **tablets** hold shards of the map
- Reads & writes within a row are **transactional**
  - Independently of the number of columns touched
  - **But:** no cross-row transactions possible
  - Turns out users find this hard to deal with
- Example of good principles for DS design:
  - **stateless design** (stores all state in Colossus, Chubby)
  - **layered design** (relies on other services and structures)
  - **recursive design** (tablet server locations are stored in Bigtable itself)

Tablets

- A Bigtable table is partitioned into many tablets based on row keys
  - Tablets (100-200MB each) are stored in a particular structure in GFS
- Each tablet is served by one tablet server
  - Tablets are stateless (all state is in GFS), hence they can restart at any time



The Bigtable API

- Metadata operations
  - Create/delete tables, column families, change metadata

- Writes: Single-row, atomic
  - Set(): write cells in a row
  - DeleteCells(): delete cells in a row
  - DeleteRow(): delete all cells in a row

- Reads: Scanner abstraction
  - Allows to read arbitrary cells in a Bigtable table
    - Each row read is atomic
    - Can restrict returned rows to a particular range
    - Can ask for just data from 1 row (getter), all rows (scanner), etc.
    - Can ask for all columns, just certain column families, or specific columns
    - Can ask for certain timestamps only

API Examples: Write

```
// Open the table
Table *T = OpenOrDie("/bigtable/web/webtable");

// Write a new anchor and delete an old anchor
RowMutation r1(T, "com.cnn.www");
r1.Set("anchor:www.c-span.org", "CNN");
r1.Delete("anchor:www.abc.com");
Operation op;
Apply(&op, &r1);
```

atomic row modification

No support for (RDBMS-style) multi-row transactions

Servers

- Library linked into every client
- One master server
  - Assigns/load-balances tablets to tablet servers
  - Detects up/down tablet servers
  - Garbage collects deleted tablets
  - Coordinates metadata updates (e.g., create table, …)
  - Does **NOT** provide tablet location (we'll see how this is gotten)
  - Master is stateless – state is in Chubby and… Bigtable (recursively)!

- Many tablet servers
  - Tablet servers handle R/W requests to their tablets
  - Split tablets that have grown too large
  - Tablet servers are also stateless – their state is in GFS!

Chubby & Colossus State

Chubby state:

```
/ls/bt/master-server
/ls/bt/live-tablet-servers/
   /ID1
   /ID2
  …
/ls/bt/first-metadata-server
```

Colossus state:

```
/fs/bt/tabletID1/
  /log
  /SS1
  /SS2
  …
/fs/bt/tabletID2/
   …
```

Write on whiteboard

Tablet Assignment

- 1 Tablet => 1 Tablet server
- Master
  - keeps tracks of set of live tablet serves and unassigned tablets
  - Master sends a tablet load request for unassigned tablet to the tablet server

- Bigtable uses Chubby to keep track of tablet servers

- On startup a tablet server:
  - Tablet server creates and acquires an exclusive lock on uniquely named file in Chubby directory
  - Master monitors the above directory to discover tablet servers

- Tablet server stops serving tablets if it loses its exclusive lock
  - Tries to reacquire the lock on its file as long as the file still exists

Tablet Assignment

- If the file no longer exists, tablet server not able to serve again and kills itself

- Master is responsible for finding when tablet server is no longer serving its tablets and reassigning those tablets as soon as possible.

- Master detects by checking periodically the status of the lock of each tablet server.
  - If tablet server reports the loss of lock
  - Or if master could not reach tablet server after several attempts.

Tablet Assignment

- Master tries to acquire an exclusive lock on server's file.
  - If master is able to acquire lock, then chubby is alive and tablet server is either dead or having trouble reaching chubby.
  - If so master makes sure that tablet server never can server again by deleting its server file.
  - Master moves all tablets assigned to that server into set of unassigned tablets.
- If Chubby session expires, master kills itself.
- When master is started, it needs to discover the current tablet assignment.

Master Startup Operation

- Grabs unique master lock in Chubby
  - Prevents server instantiations
- Scans directory in Chubby for live servers
- Communicates with every live tablet server
  - Discover all tablets
- Scans METADATA table to learn the set of tablets
  - Unassigned tables are marked for assignment

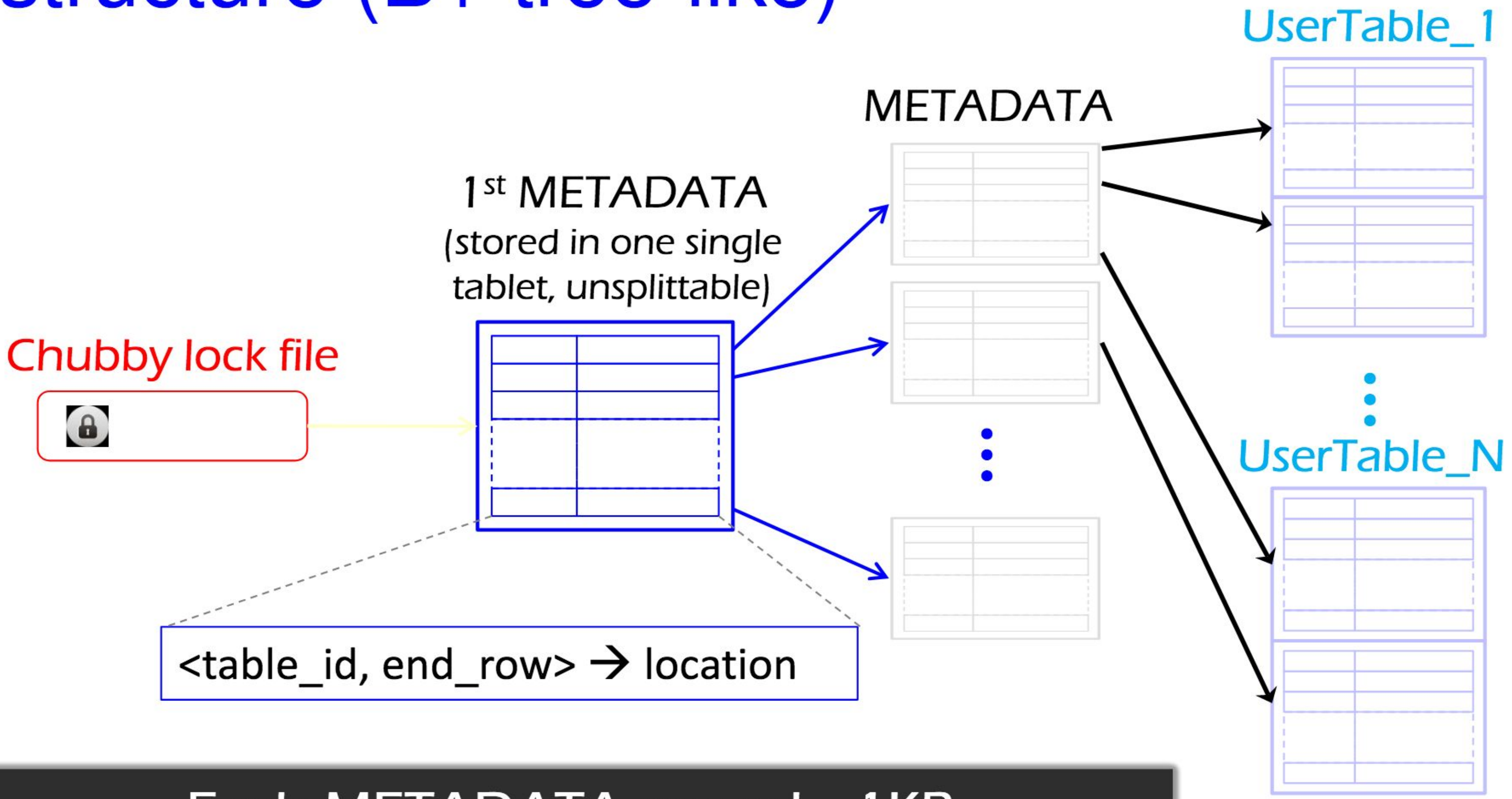
Locating Tablets

- Since tablets move around from server to server, given a row, how do clients find the right machine?
  - Tablet properties: startRowIndex and endRowIndex
  - Need to find tablet whose row range covers the target row

- One approach: could use the Bigtable master
  - Central server almost certainly would be bottleneck in large system

- Instead: store special tables containing tablet location info in the Bigtable cell itself (recursive design ☺)

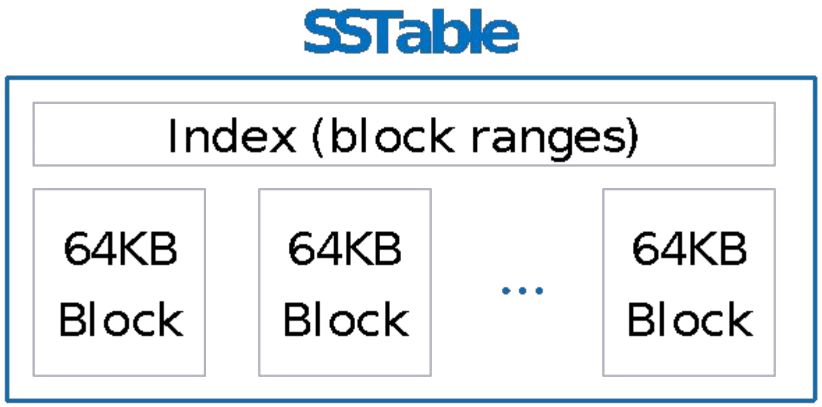
Tablets are located using a hierarchical structure (B+ tree-like)


UserTable_1
METADATA
1st METADATA
(stored in one single tablet, unsplittable)
Chubby lock file
UserTable_N
<table_id, end_row> → location

Each METADATA record ~1KB
Max METADATA table = 128MB
Addressable table values in Bigtable = 2^{21} TB

Tablet storage and R/W operation

- Uses Google SSTables, a key building block
- Without going into much detail, an SSTable:
  - Is an immutable, sorted file of key-value pairs
  - SSTable files are stored in GFS
  - Keys are: <row, column, timestamp>
  - SSTables allow only appends, no updates (delete possible)
    - Why do you think they don't use something that supports updates?



Read/Write Operations

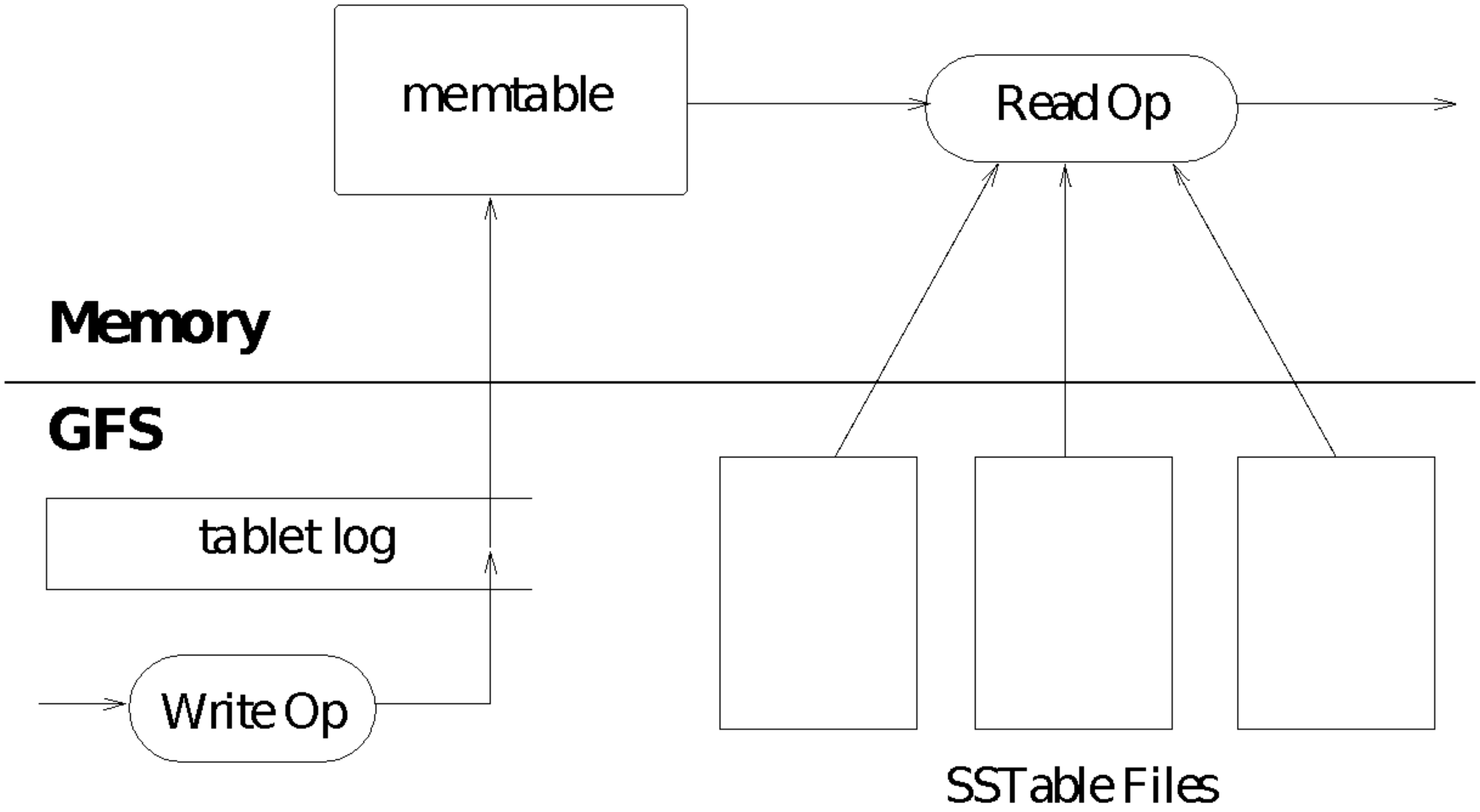


Figure 5: Tablet Representation

The Google Stack

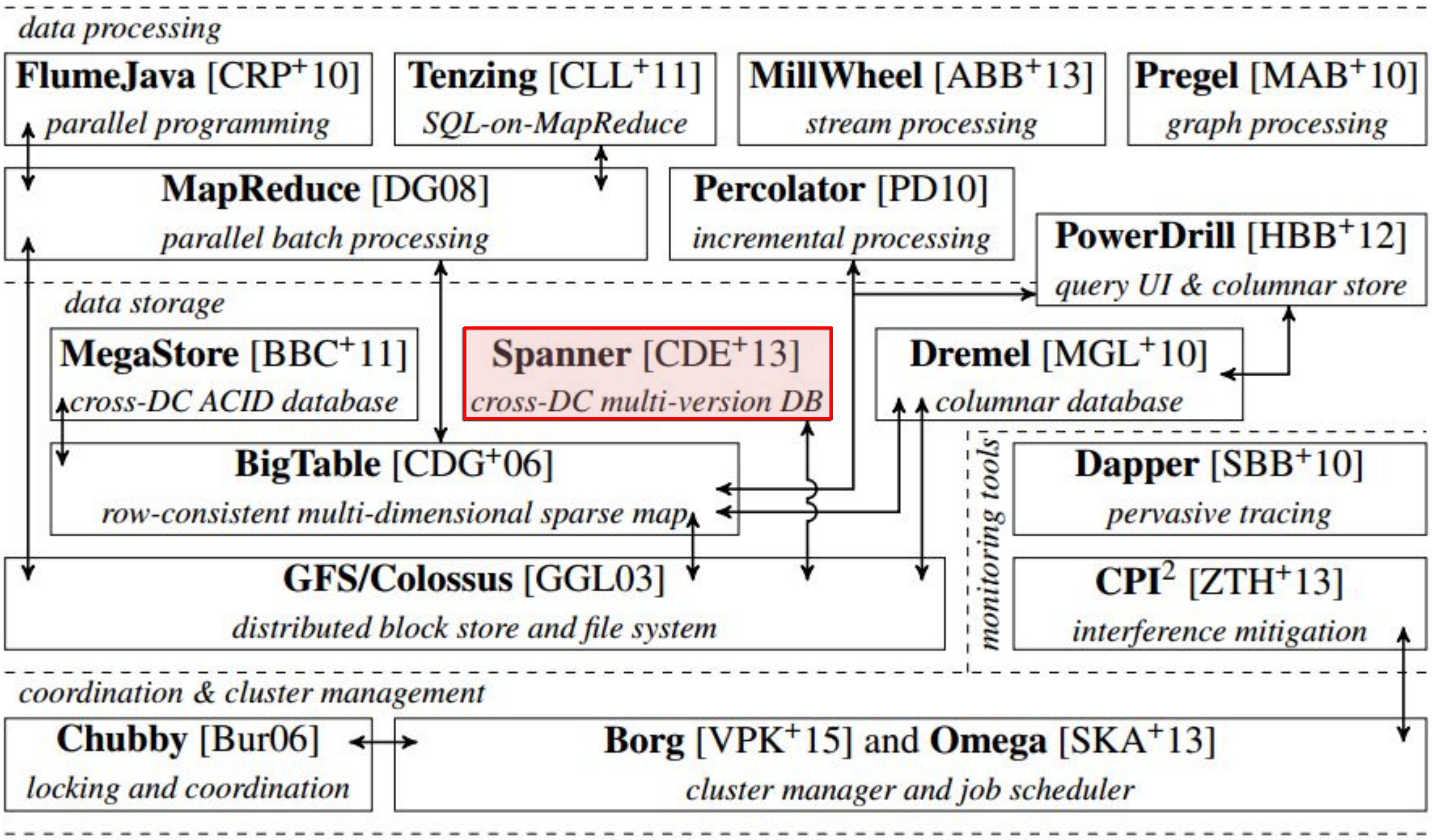


Figure from M. Schwarzkopf, "Operating system support for warehouse-scale computing", PhD thesis, University of Cambridge.

Spanner (2012)

- BigTable insufficient for some consistency needs
- Often have transactions across >1 data centres
  - May buy app on Play Store while travelling in the U.S.
  - Hit U.S. server, but customer billing data is in U.K.
  - Or may need to update several replicas for fault tolerance

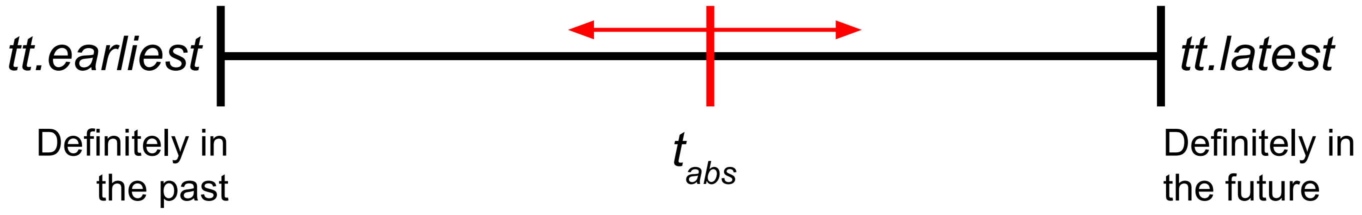
- Wide-area consistency is hard
  - due to long delays and clock skew
  - no **global, universal notion of time**
  - NTP not accurate enough, PTP doesn’t work (jittery links)

Spanner (2012)

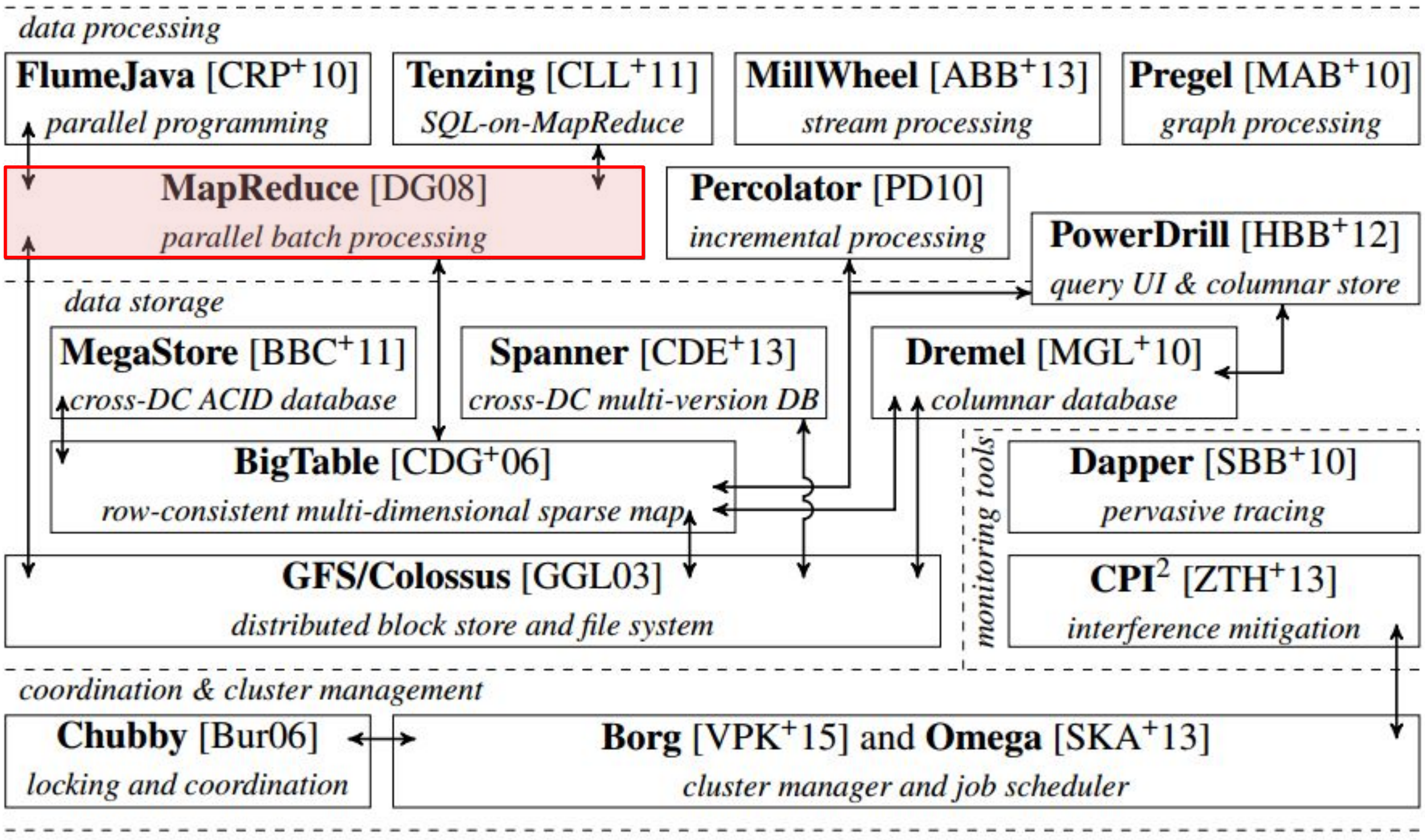
- Spanner offers **transactional** consistency: full RDBMS power, ACID properties, at global scale!

- Secret sauce: **hardware-assisted clock sync**
  - Using GPS and atomic clocks in data centres

- Use global timestamps and Paxos to reach consensus
  - Still have a period of uncertainty for write TX: **wait it out**!
  - Each timestamp is an **interval**:



The Google Stack



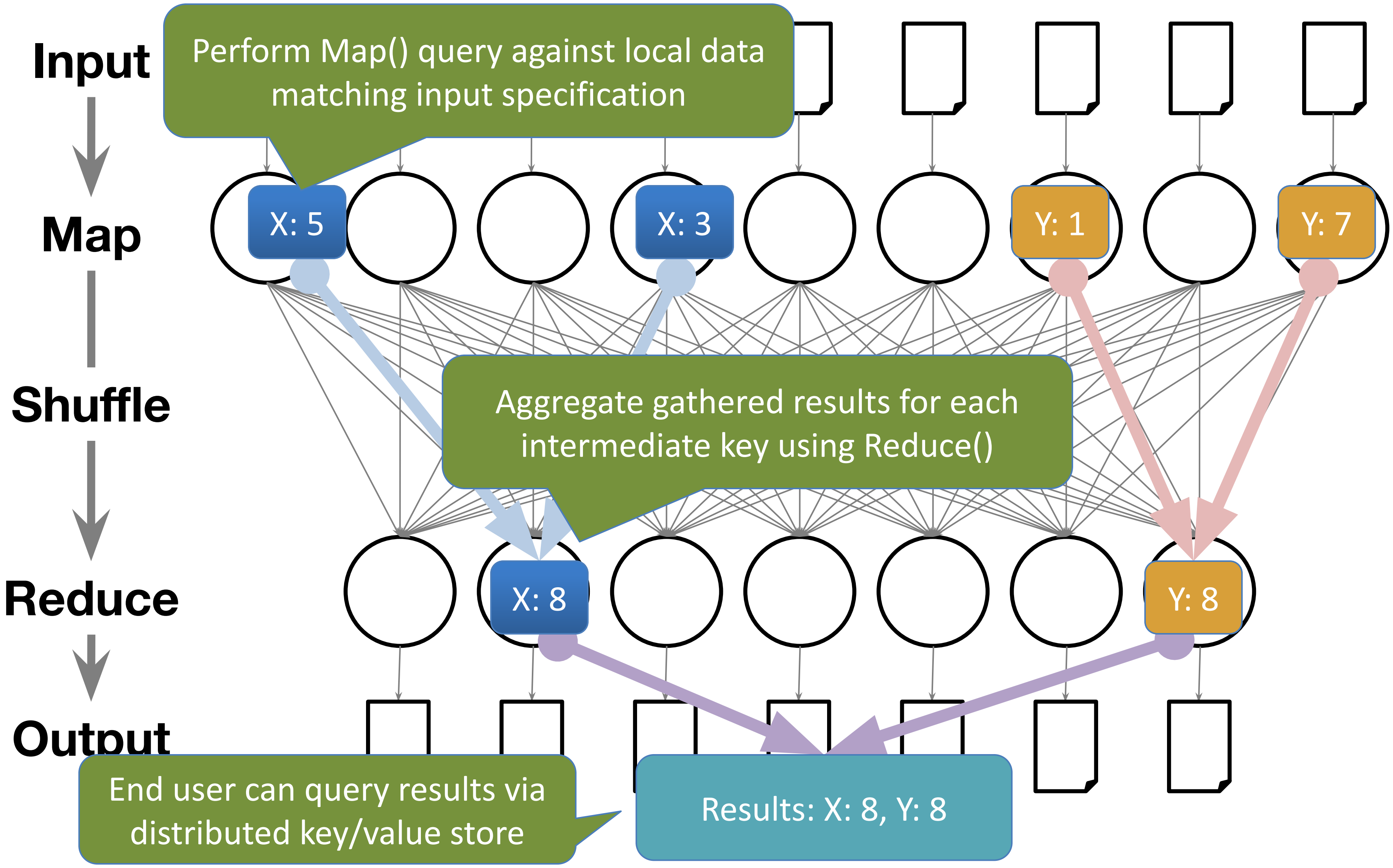
Details & Bibliography: http://malteschwarzkopf.de/research/assets/google-stack.pdf

Figure from M. Schwarzkopf, "Operating system support for warehouse-scale computing", PhD thesis, University of Cambridge, 2015.

MapReduce (2004)

- **Parallel programming framework** for scale
  - Run a program on 100’s to 10,000’s machines
- Framework takes care of:
  - Parallelization, distribution, load-balancing, scaling up (or down) & fault-tolerance
- **Accessible:** programmer provides two methods ;-)
  - map(key, value) → list of <key’, value’> pairs
  - reduce(key’, value’) → result
  - Inspired by functional programming

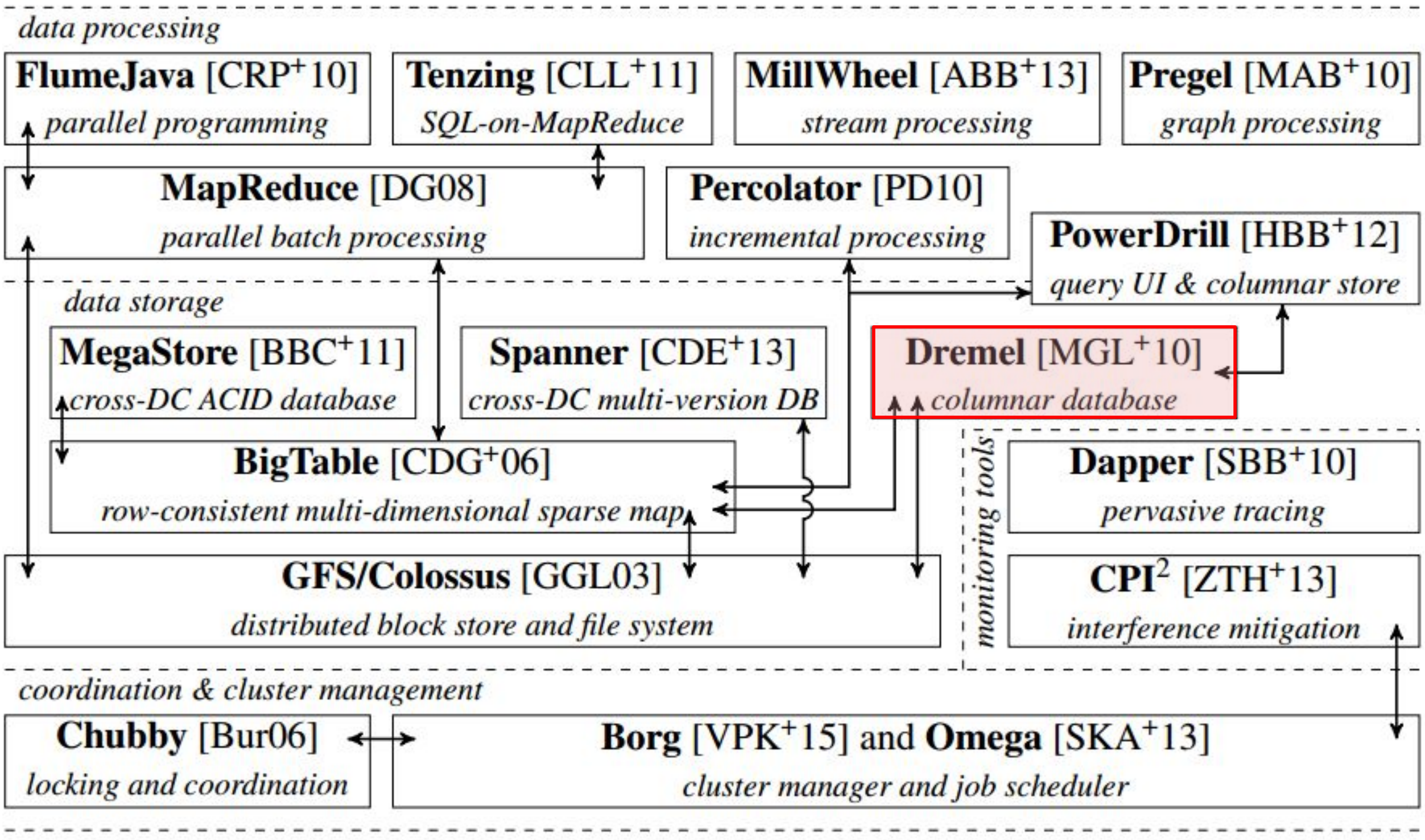
MapReduce


Input
Map
Shuffle
Reduce
Output
Perform Map() query against local data matching input specification
X: 5
X: 3
Y: 1
Y: 7
Aggregate gathered results for each intermediate key using Reduce()
X: 8
Y: 8
End user can query results via distributed key/value store
Results: X: 8, Y: 8

MapReduce: Pros & Cons

- **Extremely simple**, and:
  - Can **auto-parallelize** (since operations on every element in input are independent)
  - Can **auto-distribute** (since rely on underlying Colossus/BigTable distributed storage)
  - Gets **fault-tolerance** (since tasks are idempotent, i.e. can just re-execute if a machine crashes)
- Doesn't really use **any** sophisticated distributed systems algorithms (except storage replication)
- However, not a panacea:
  - Limited to batch jobs, and computations which are expressible as a `map()` followed by a `reduce()`

The Google Stack

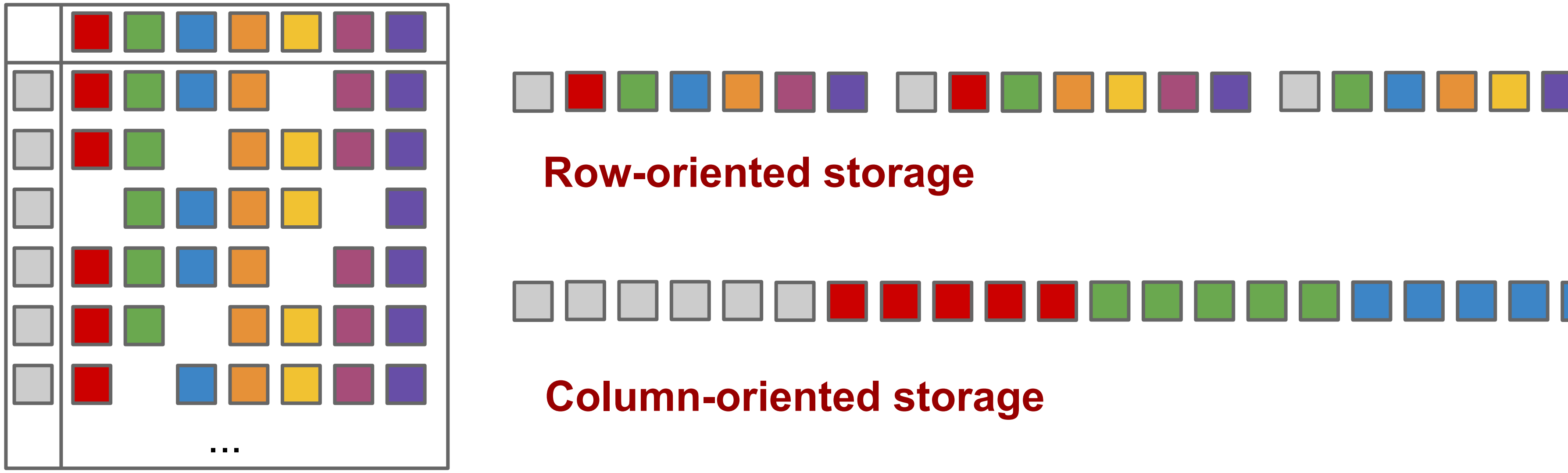


Details & Bibliography: http://malteschwarzkopf.de/research/assets/google-stack.pdf

Figure from M. Schwarzkopf, "Operating system support for warehouse-scale computing", PhD thesis, University of Cambridge, 2015.

Dremel (2010)

- **Column-oriented** store
    - For quick, interactive queries



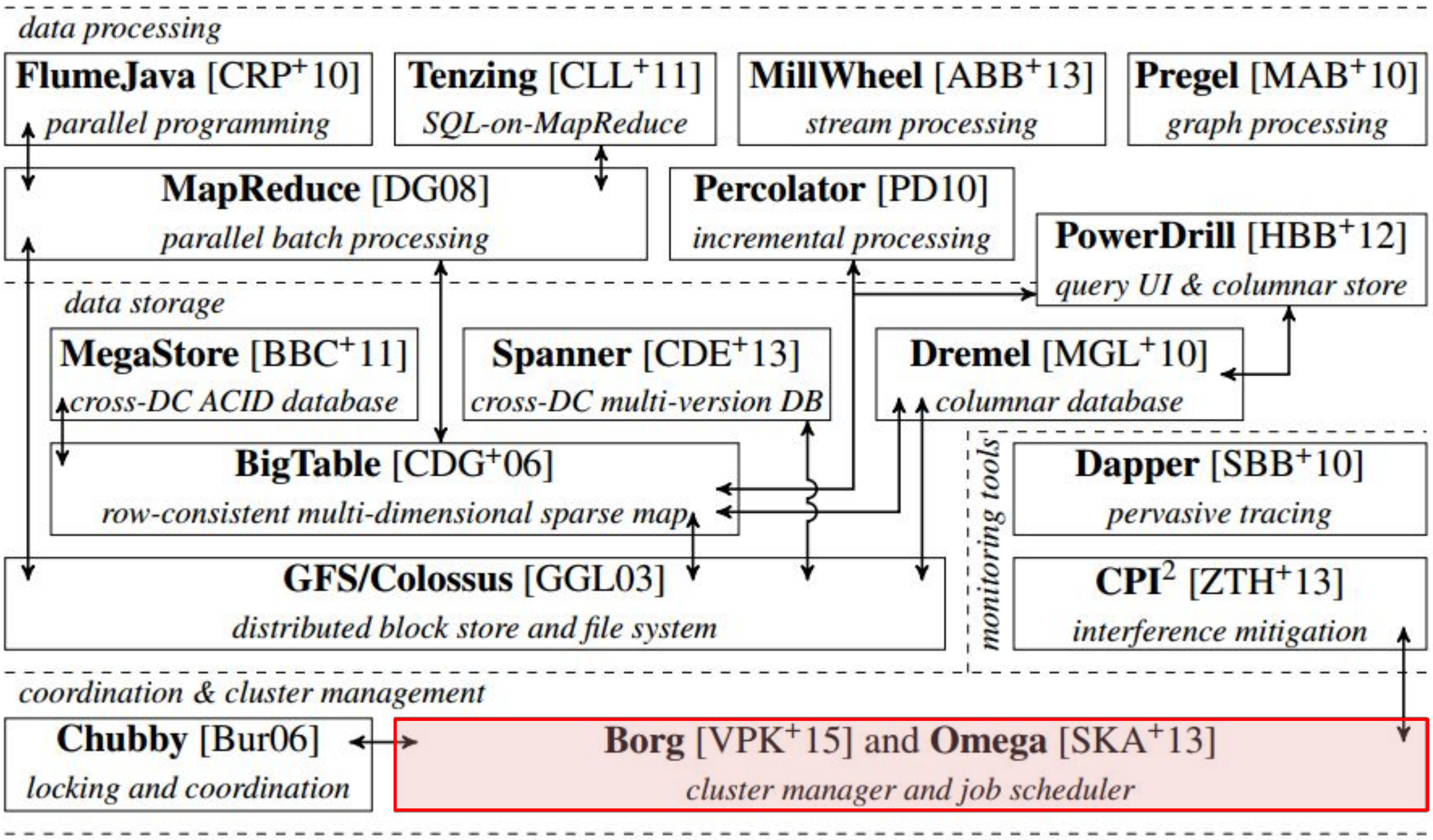
Dremel (2010)

- Stores **protocol buffers**
  - Google's universal serialization format
  - Nested messages → nested columns
  - Repeated fields → repeated records

- Efficient encoding
  - Many **sparse records**: don't store NULL fields

- Record re-assembly
  - Need to put results back together into records
  - Use a **Finite State Machine** (FSM) defined by protocol buffer structure

The Google Stack



Details & Bibliography: http://malteschwarzkopf.de/research/assets/google-stack.pdf

Figure from M. Schwarzkopf, "Operating system support for warehouse-scale computing", PhD thesis, University of Cambridge, 2015 (to appear).

Borg

- **Cluster manager** and scheduler
  - Tracks machine and task liveness
  - Decides where to run what

- Consolidates workloads onto machines
  - Efficiency gain, cost savings
  - Need fewer clusters

- Watch Borg EuroSys'14 talk by John Wilkes: https://www.youtube.com/watch?v=7MwxA4Fj2l4

Borg

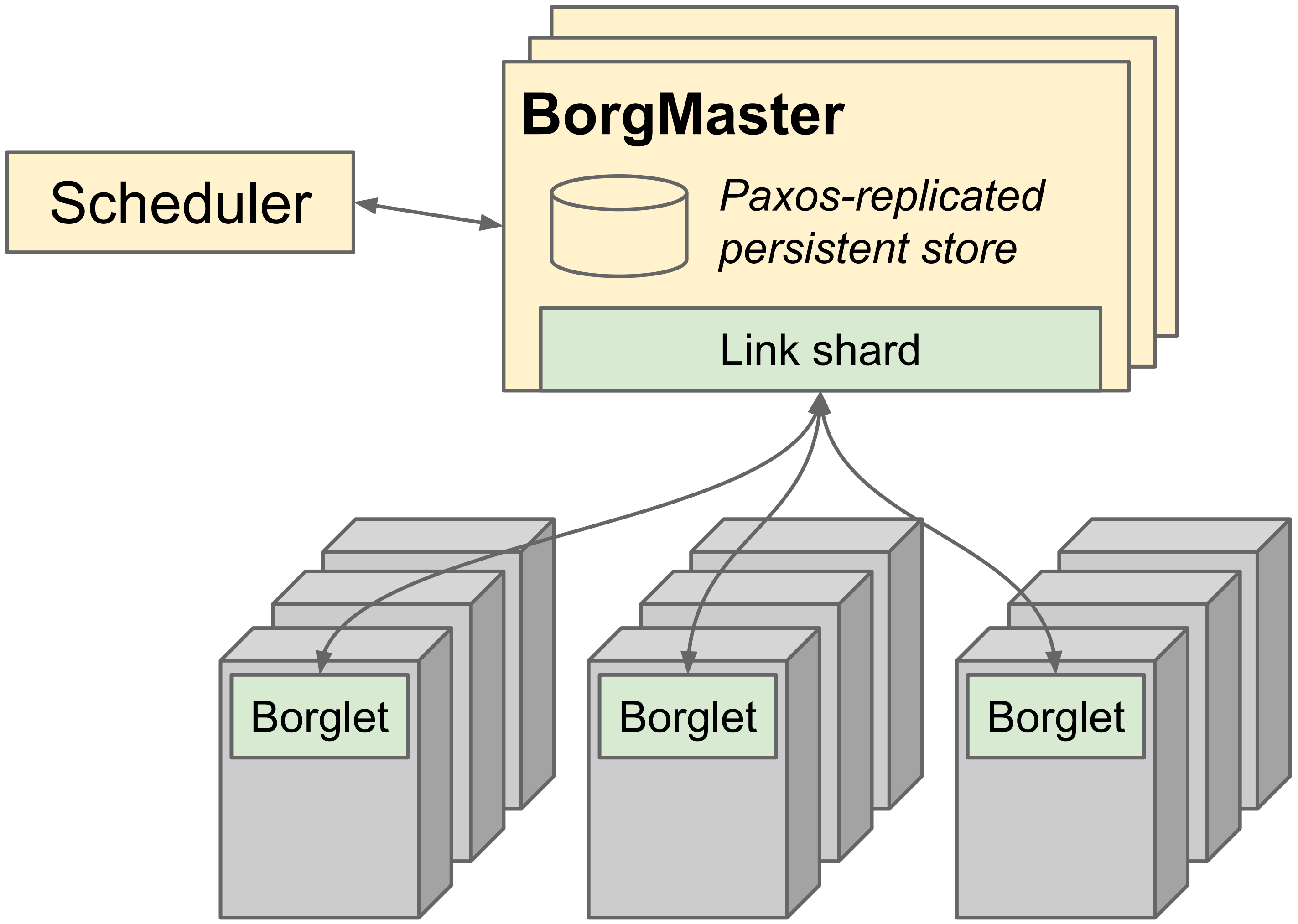


Figure reproduced after A. Verma et al., “Large-scale cluster management at Google with Borg”, Proceedings of EuroSys 2015.

Borg: workloads

Cluster A

Medium size

Medium utilization

Cluster B

Large size

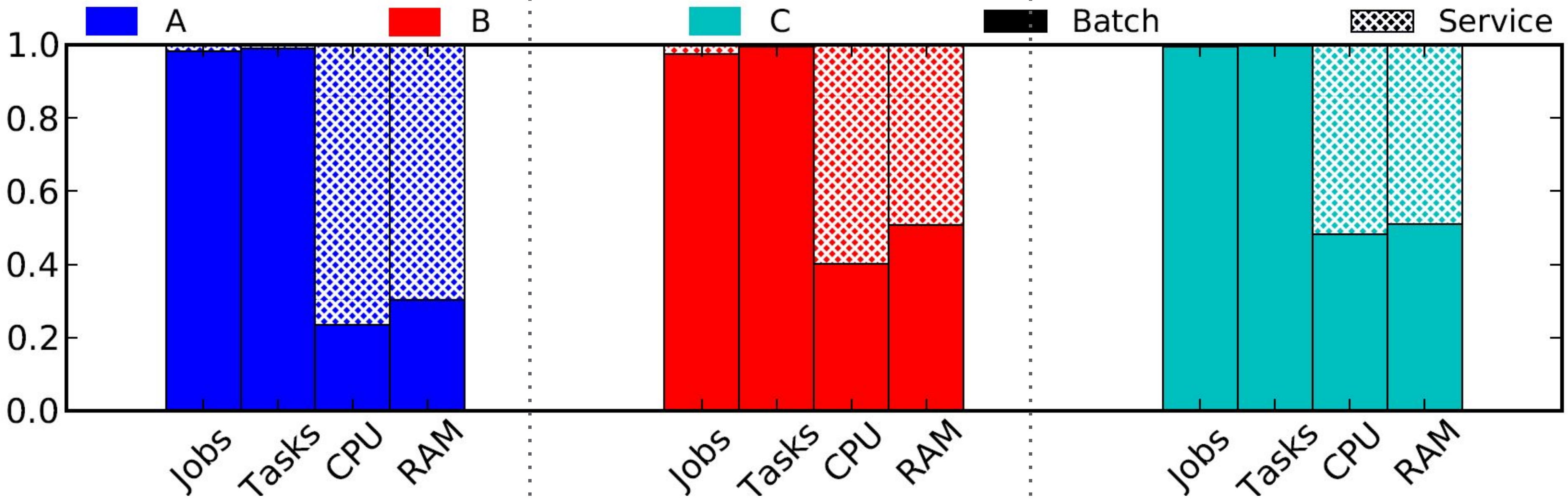
Medium utilization

Cluster C

Medium (12k mach.)

High utilization

Public trace



Jobs/tasks:	counts
CPU/RAM:	resource seconds [i.e. resource * job runtime in sec.]

Figure from M. Schwarzkopf et al., “Omega: flexible, scalable schedulers for large compute clusters”, Proceedings of EuroSys 2013.

Borg: workloads

Service jobs run for much longer than batch jobs:
long-term user-facing services vs. one-off analytics.

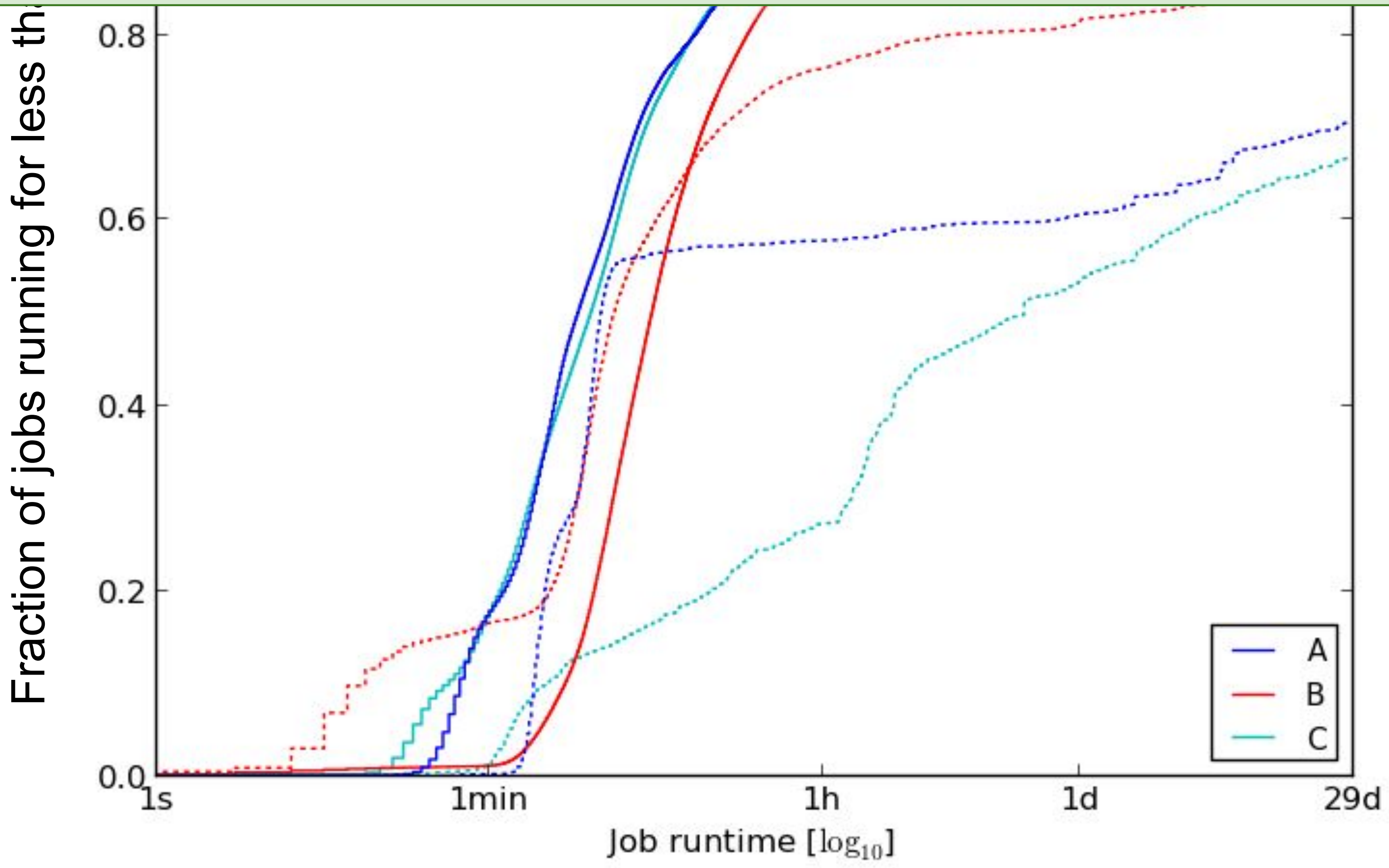


Figure from M. Schwarzkopf et al., "Omega: flexible, scalable schedulers for large compute clusters", Proceedings of EuroSys 2013.

Borg: workloads

Batch jobs arrive more frequently than service jobs:
more numerous, shorter duration, fail more.

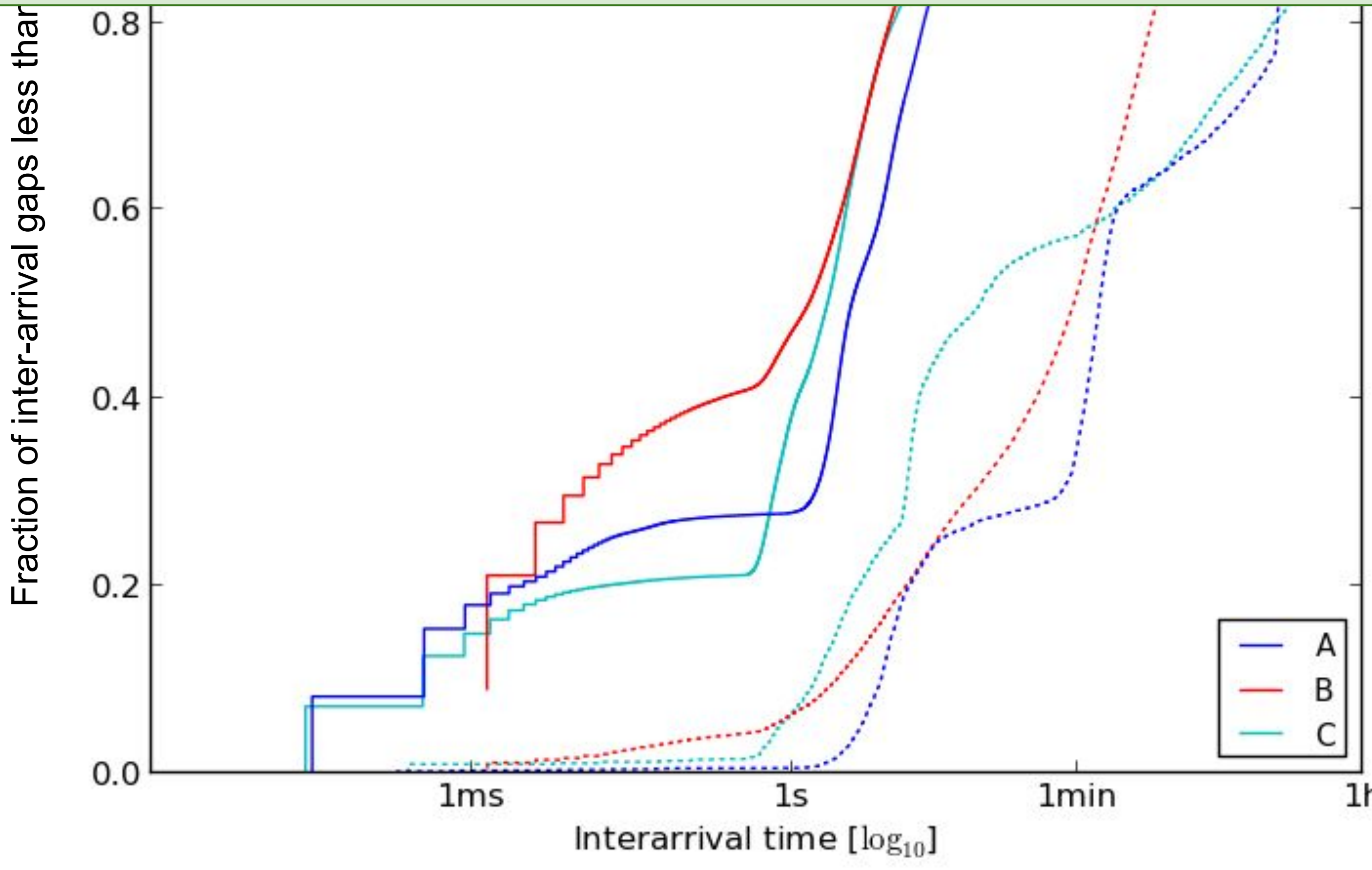
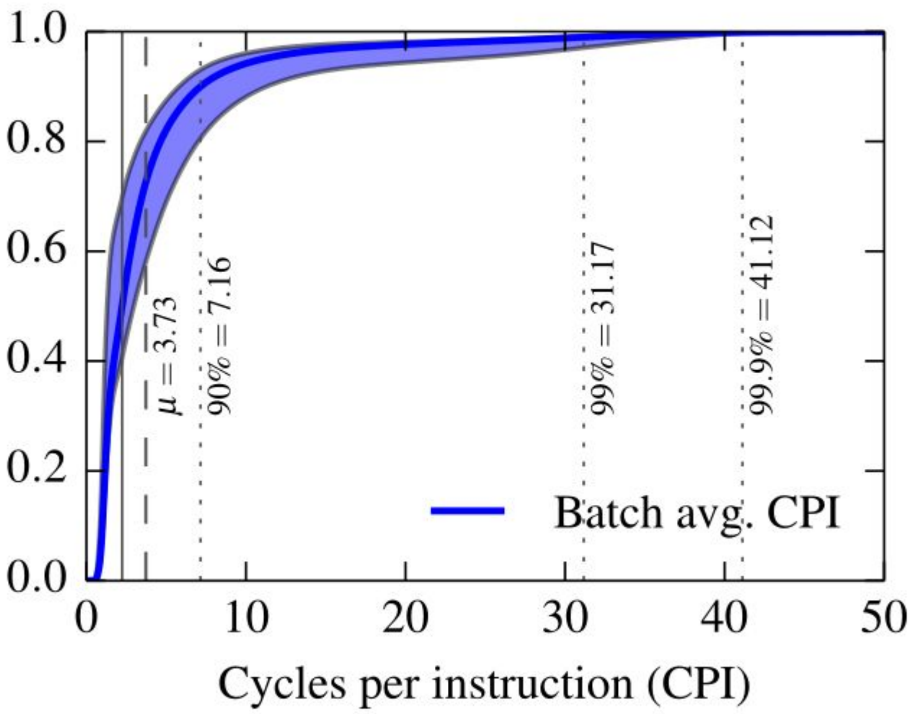
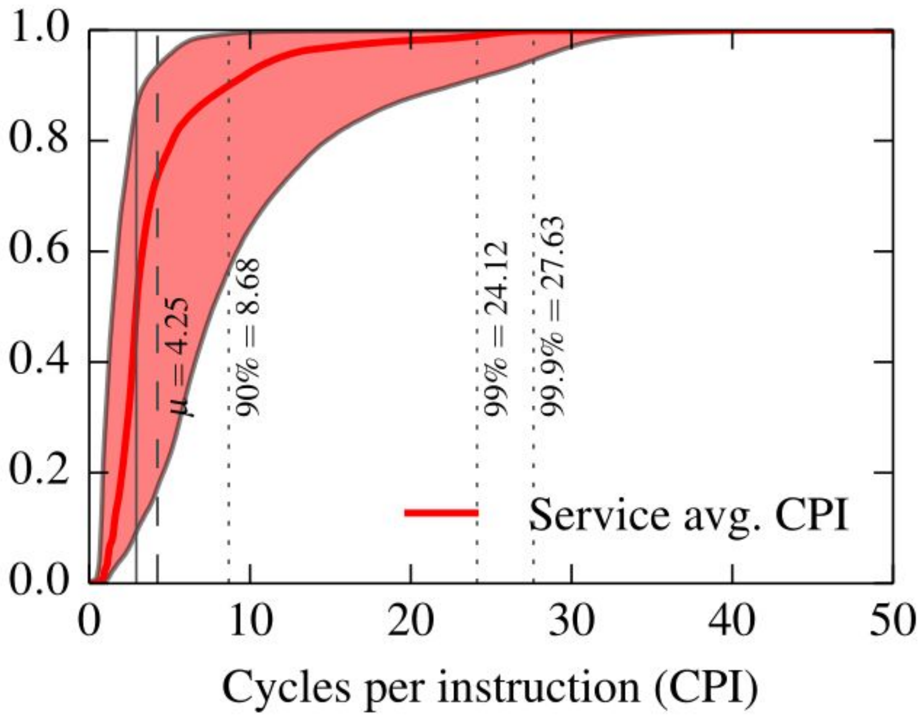


Figure from M. Schwarzkopf et al., “Omega: flexible, scalable schedulers for large compute clusters”, Proceedings of EuroSys 2013.

Borg: workloads

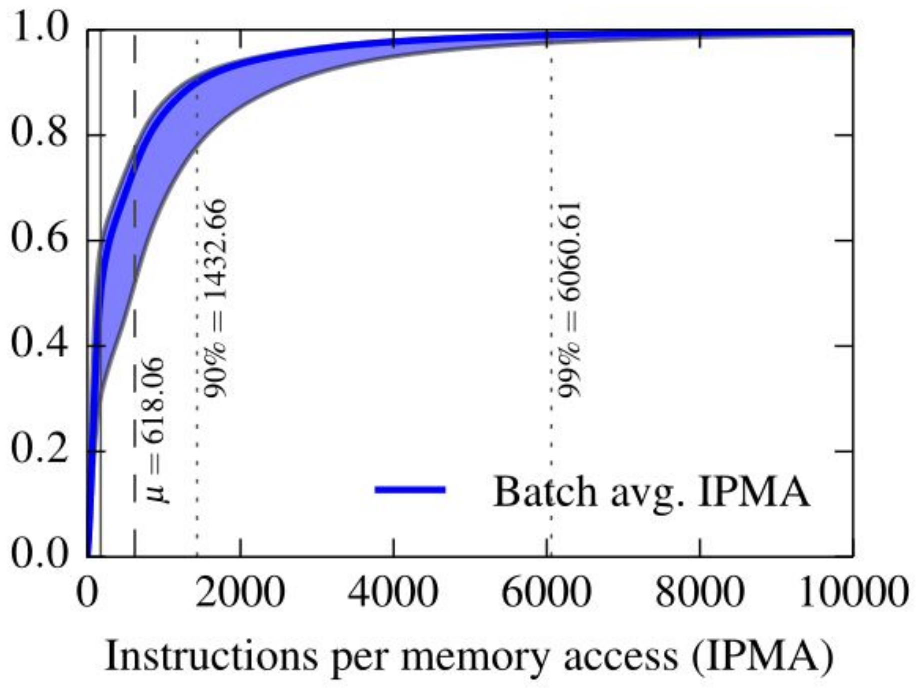


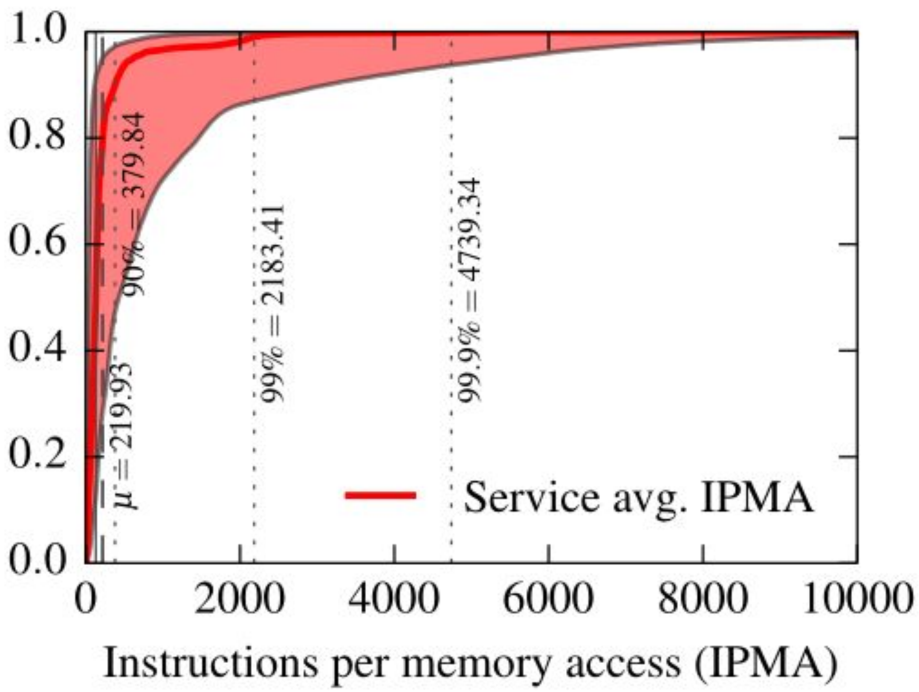


Batch jobs have a longer-tailed CPI distribution:
lower scheduling priority in kernel scheduler.

Figures from M. Schwarzkopf, “Operating system support for warehouse-scale computing”, PhD thesis, University of Cambridge, 2015.

Borg: workloads

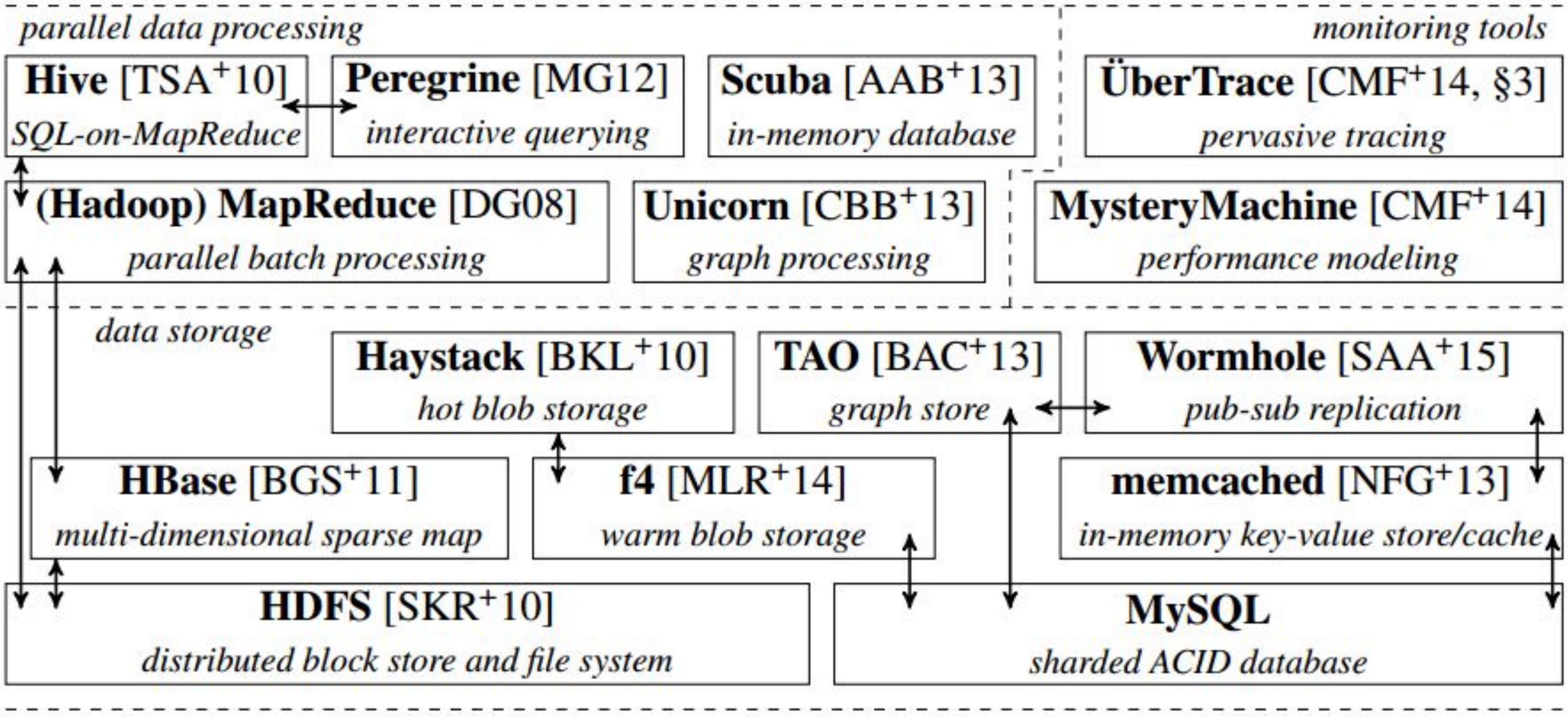




Service workloads access memory more frequently:
larger working sets, less I/O.

Figures from M. Schwarzkopf, "Operating system support for warehouse-scale computing", PhD thesis, University of Cambridge, 2015.

The facebook Stack

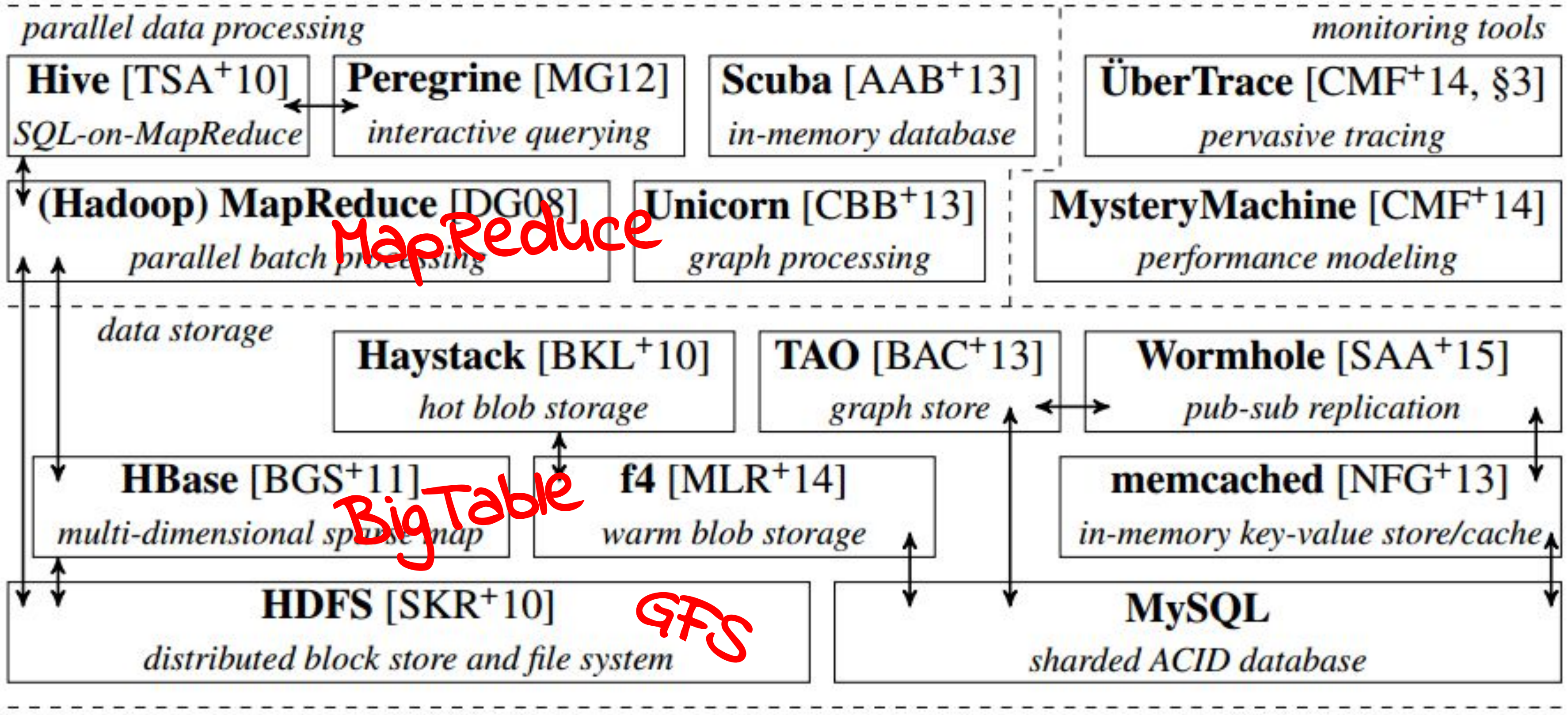


parallel data processing
monitoring tools
Hive [TSA+10] SQL-on-MapReduce
Peregrine [MG12] interactive querying
Scuba [AAB+13] in-memory database
ÜberTrace [CMF+14, §3] pervasive tracing
(Hadoop) MapReduce [DG08] parallel batch processing
Unicorn [CBB+13] graph processing
MysteryMachine [CMF+14] performance modeling
data storage
Haystack [BKL+10] hot blob storage
TAO [BAC+13] graph store
Wormhole [SAA+15] pub-sub replication
HBase [BGS+11] multi-dimensional sparse map
f4 [MLR+14] warm blob storage
memcached [NFG+13] in-memory key-value store/cache
HDFS [SKR+10] distributed block store and file system
MySQL sharded ACID database

Details & Bibliography: http://malteschwarzkopf.de/research/assets/facebook-stack.pdf

Figure from M. Schwarzkopf, "Operating system support for warehouse-scale computing", PhD thesis, University of Cambridge, 2015.

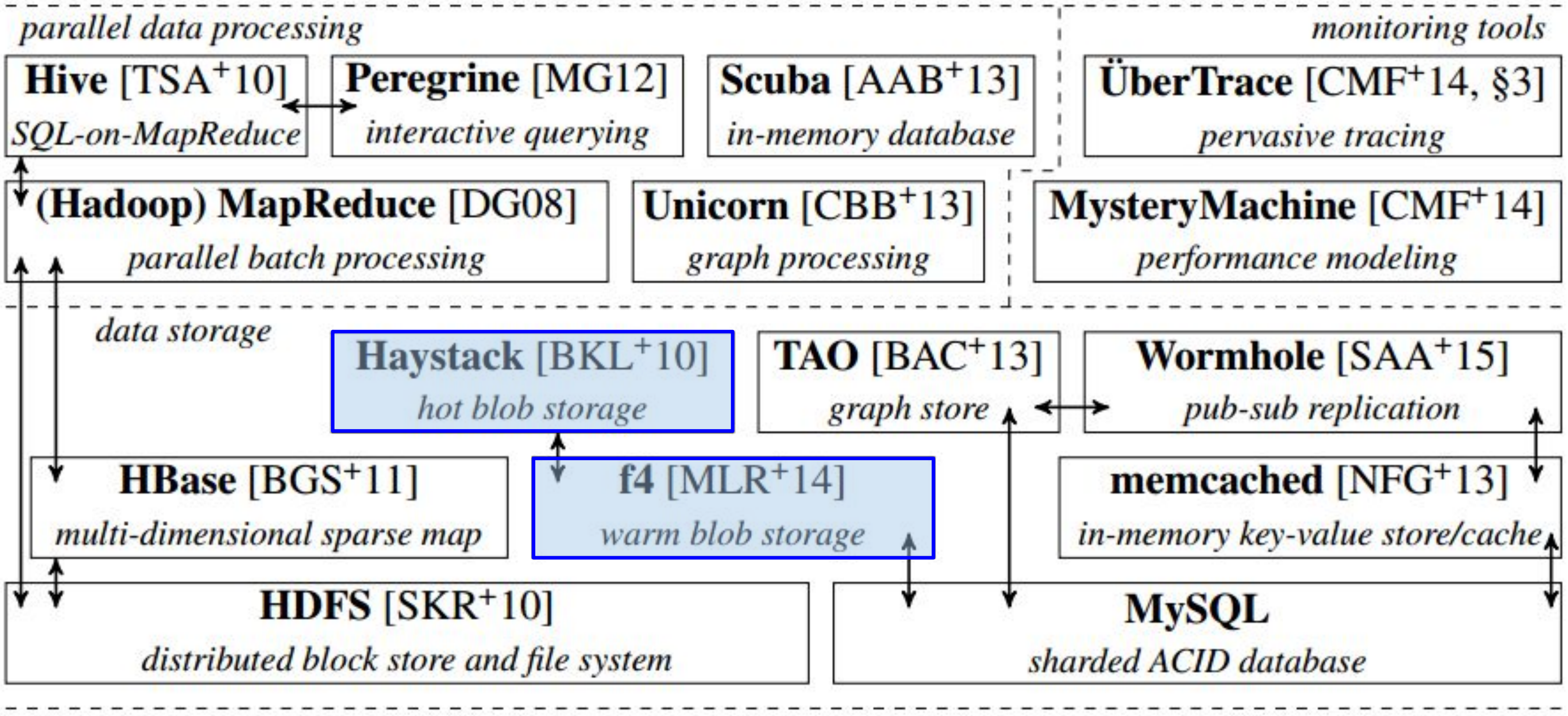
The facebook Stack



Details & Bibliography: http://malteschwarzkopf.de/research/assets/facebook-stack.pdf

Figure from M. Schwarzkopf, “Operating system support for warehouse-scale computing”, PhD thesis, University of Cambridge, 2015.

The facebook Stack



Details & Bibliography: http://malteschwarzkopf.de/research/assets/facebook-stack.pdf

Figure from M. Schwarzkopf, “Operating system support for warehouse-scale computing”, PhD thesis, University of Cambridge, 2015.

Haystack & f4

- **Blob stores**, hold photos, videos
  - **not:** status updates, messages, like counts

- Items have a level of **hotness**
  - How many users are currently accessing this?
  - Baseline “cold” storage: MySQL

- Want to **cache close to users**
  - Reduces network traffic
  - Reduces latency
  - But cache capacity is limited!
  - Replicate for performance, not resilience

What about
other companies’ stacks?

How about other companies?

- Very similar stacks.
    - Microsoft, Yahoo, Twitter all similar in principle.
- Typical set-up:
    - Front-end serving systems and fast back-ends.
    - Batch data processing systems.
    - Multi-tier structured/unstructured storage hierarchy.
    - Coordination system and cluster scheduler.
- Minor differences owed to business focus
    - e.g., Amazon focused on inventory/shopping cart.

Open source software

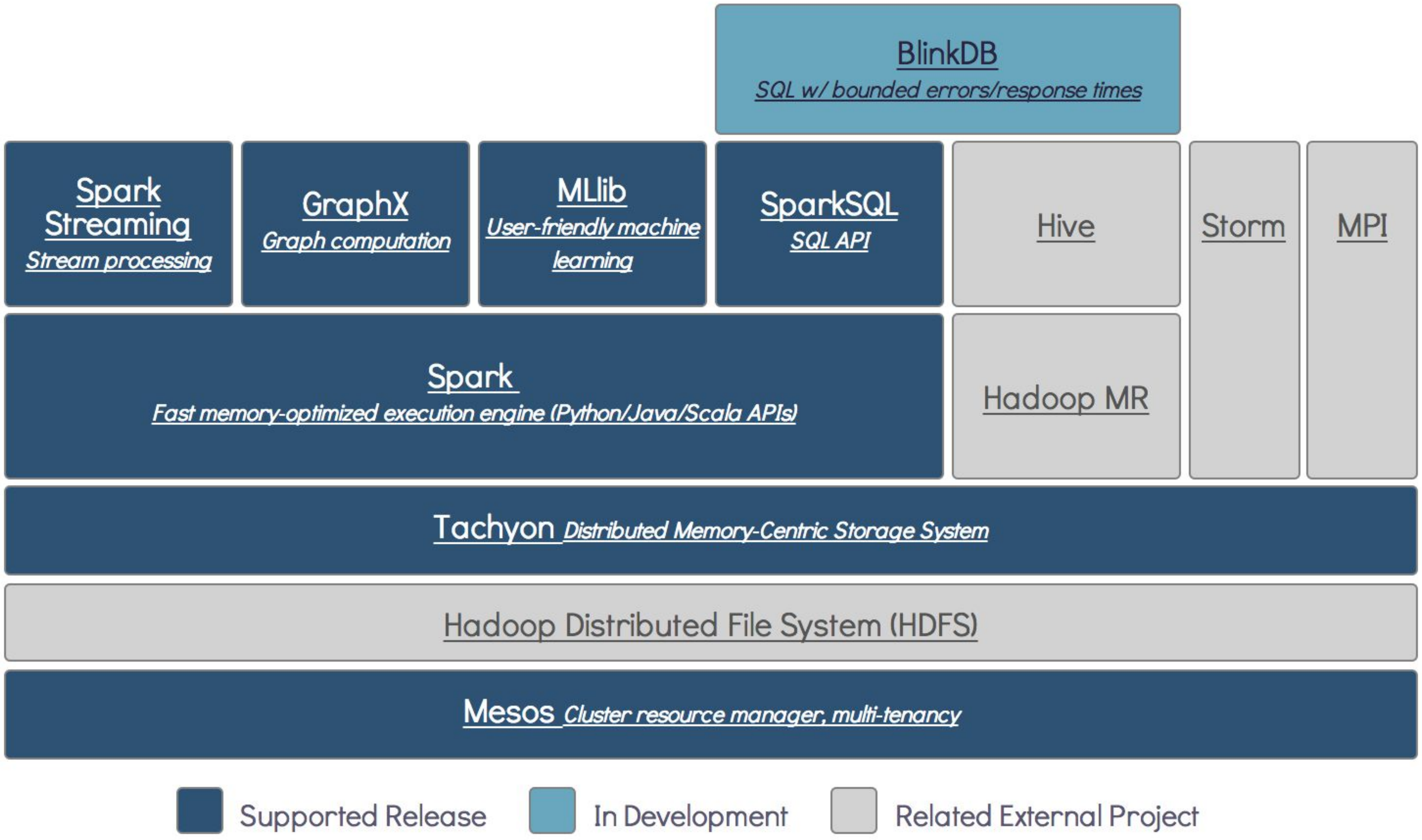
Lots of open-source implementations!

- MapReduce → Hadoop, Spark, Metis
- GFS → HDFS
- BigTable → HBase, Cassandra
- Borg → Mesos, Firmament
- Chubby → Zookeeper

But also some releases from companies…

- Presto (Facebook)
- Kubernetes (Google Borg)

The Spark Stack


BlinkDB
SQL w/ bounded errors/response times
Spark Streaming
Stream processing
GraphX
Graph computation
MLlib
User-friendly machine learning
SparkSQL
SQL API
Hive
Storm
MPI
Spark
Fast memory-optimized execution engine (Python/Java/Scala APIs)
Hadoop MR
Tachyon Distributed Memory-Centric Storage System
Hadoop Distributed File System (HDFS)
Mesos Cluster resource manager, multi-tenancy
Supported Release
In Development
Related External Project

Newer Stacks

- Lots of new support for machine learning
  - Google: Tensorflow, Tensorflow Serving, Tensorflow Extended (TFX)
  - Uber: Michelangelo
  - Spark/Berkeley Data Stack (BDAS): MLBase, MLlib, Clipper

References

[1] Malte Schwartzkopf. “What does it taketo make Googlework at scale?” 2015. https://docs.google.com/presentation/d/1OvJStE8aohGeI3y5BcYX8bBHwoHYCPu99A3KTTZElr0/edit#slide=id.p.

[2] Jeff Dean. “Software Engineering Advice from Building Large-Scale Distributed Systems,” 2007. https://static.googleusercontent.com/media/research.google.com/en//people/jeff/stanford-295-talk.pdf.

[3] Jeff Dean. “Building Software Systems at Google and Lessons Learned,” 2010. https://static.googleusercontent.com/media/research.google.com/en//people/jeff/Stanford-DL-Nov-2010.pdf.

[4] Colin Scott. “Latency Numbers Every Programmer Should Know.” https://colin-scott.github.io/personal_website/research/interac