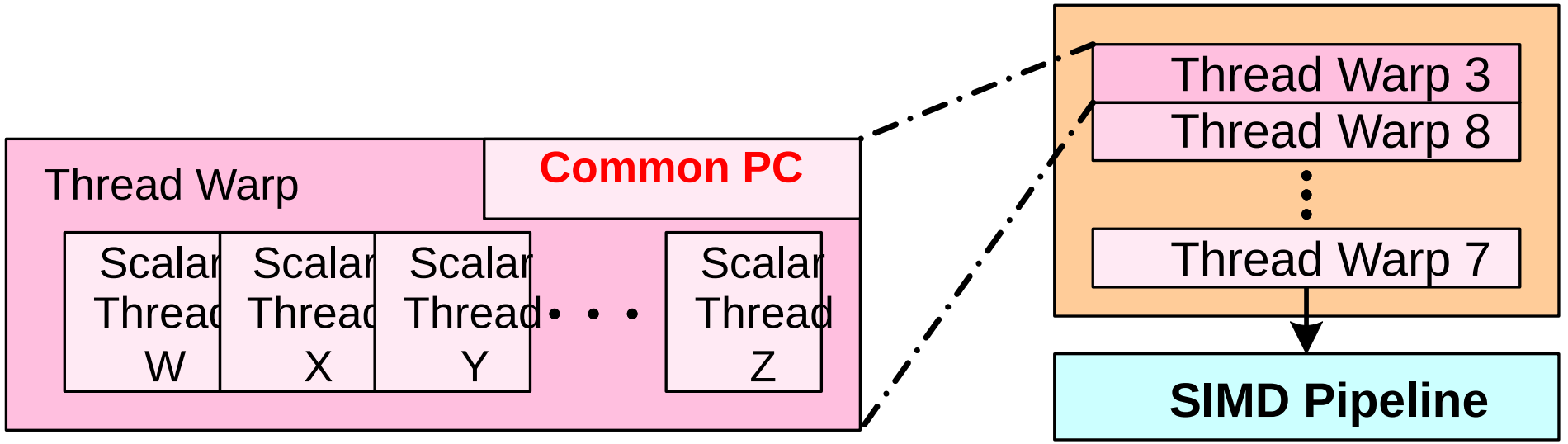
Outline

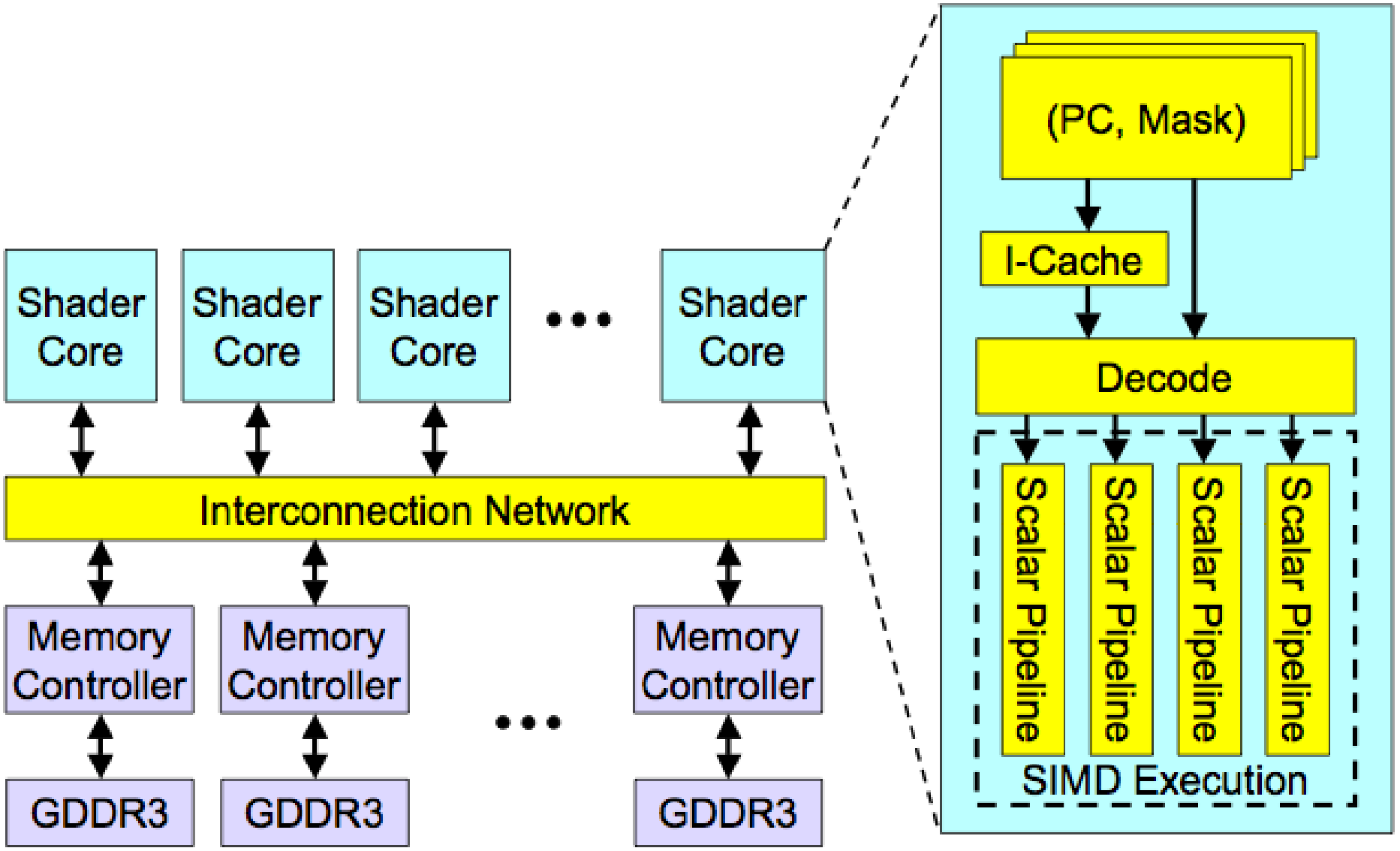
- Timeline
- GPU review
- Cloud (Warehouse scale) computing

Warps and Warp-Level Fine-Grain Multithreaded Execution

- Warp: A set of threads that execute the same instruction (on different data elements)
- All threads run the same code
  - Warp: The threads that run lengthwise in a woven fabric …



High-Level View of a GPU

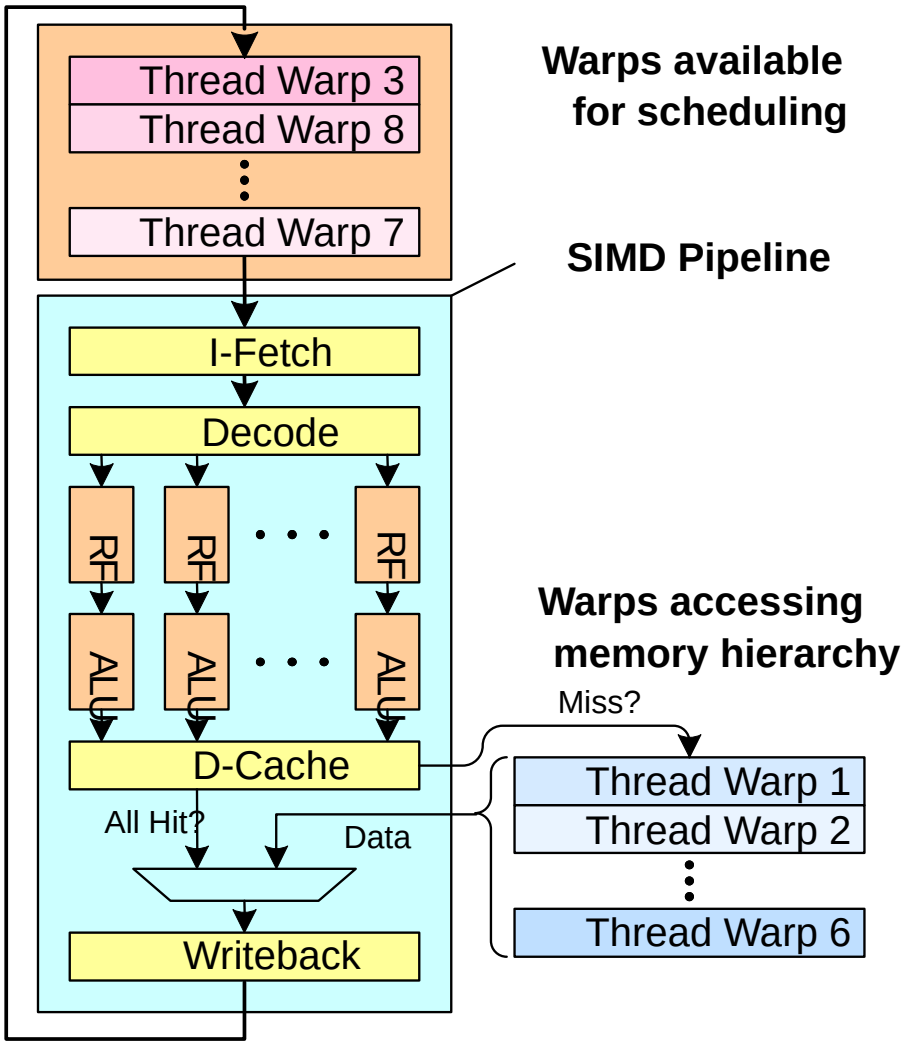


Latency Hiding via Warp-Level Fine Grain Multithreading

- Warp: A set of threads that execute the same instruction (on different data elements)

- Fine-grained multithreading
  - One instruction per thread in pipeline at a time (No interlocking)
  - Interleave warp execution to hide latencies

- Register values of all threads stay in register file
- Fine-Grain multithreading enables long latency tolerance
  - Millions of pixels

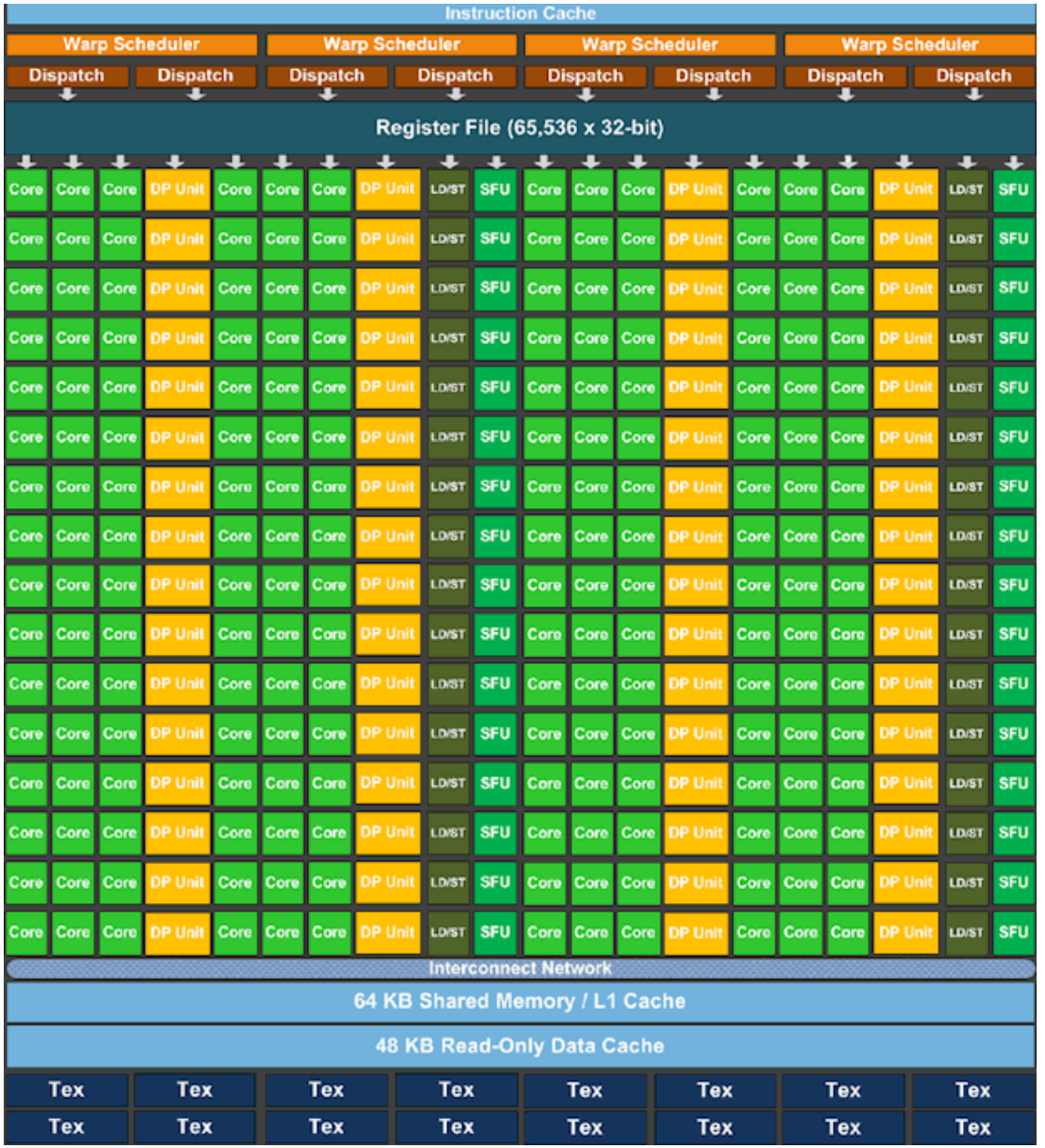


nVIDIA Kepler Architecture – GK104

- 7,08 billion transistors
- 2688 “cores”
- 384 bit memory bus (GDDR5)
- 288,40 GB/sec memory bandwidth
- 45000 Gflops single precision
- PCI Express 3.0

nVIDIA GK110 Streaming Multiprocessor (SMX)

- nVIDIA GK110
  - Fundamental thread block unit
  - 192 stream processors (SPs) (scalar ALU for threads)
  - *64 double-precision ALUs*
  - 32 super function units (SFUs) (cos, sin, log, ...)
  - 256 kB local register files (RFs)
  - 16 / 48 kB level 1 cache
  - 16 / 48 kB shared memory
  - *48 kB Read-Only Data Cache*
  - 1536 kB global level 2 cache

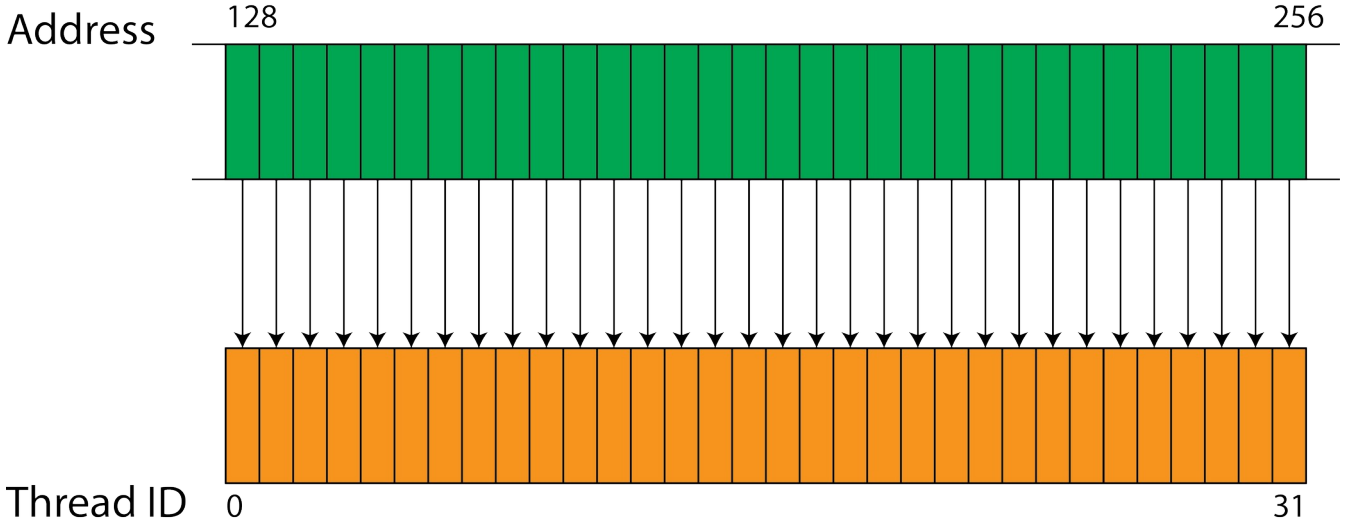


GPU Address Coalescing

- Coalesced memory access or memory coalescing refers to combining multiple memory accesses into a single transaction
  - On the K20 GPU, every successive 128 bytes (32 single precision words) memory can be accessed by a warp (32 consecutive threads) in a single transaction
- However, the following conditions may result in uncoalesced load, i.e., memory access becomes serialized:
  - memory is not sequential
  - memory access is sparse
  - misaligned memory access

GPU Address Coalescing (2)

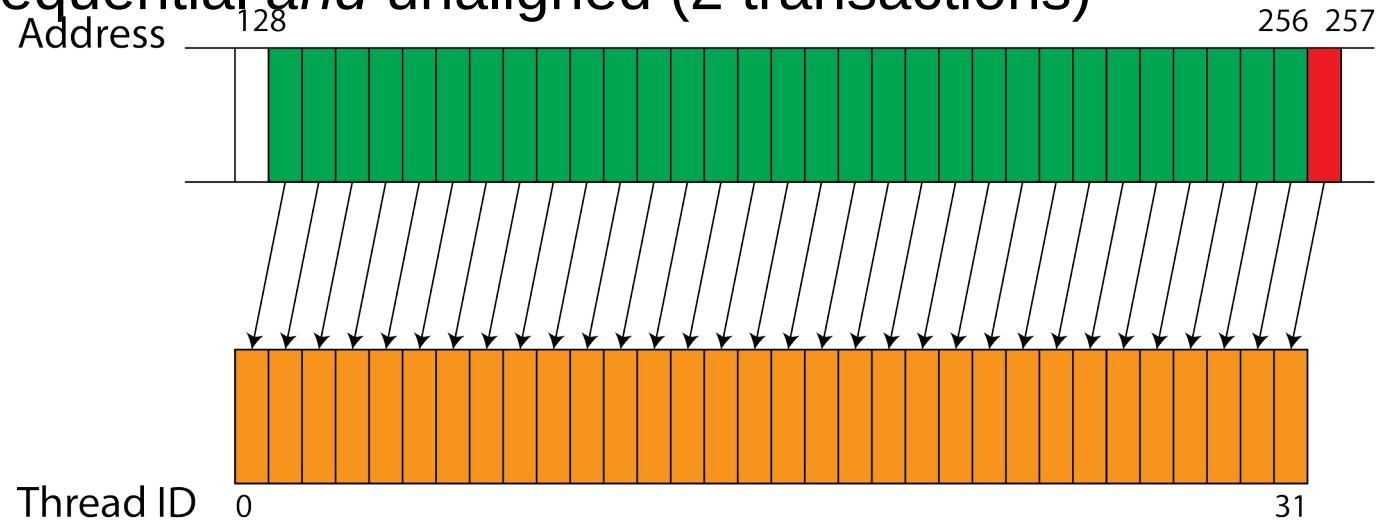
- Sequential *and* aligned



- Aligned *and not* Sequential

GPU Address Coalescing (3)

- Sequential *and* unaligned (2 transactions)



- A coalesced memory access instruction looks like this; transfer global memory (gmem), to shared memory (shmem):
  - shmem[threadIdx.x]=gmem[blockIdx.x*blockDim.x + threadIdx.x];
- The following instruction is not coalesced:

  stride=4;

  shmem[threadIdx.x]=gmem[stride*blockIdx.x*blockDim.x + threadIdx.x*stride];

GPU Conclusions

- GPUs gain efficiency from simpler cores and more parallelism
  - Very wide SIMD (SIMT) for parallel arithmetic and latency-hiding
- Heterogeneous programming with manual offload
  - CPU to run OS, etc. GPU for compute
- Massive (mostly data) parallelism required
  - Memory coalescing helps
- Threads in block share faster memory and barriers
  - Blocks in kernel share slow device memory and atomics

Warehouse scale computing (WSC)

- Provides Internet services
  - Search, social networking, online maps, video sharing, online shopping, email, cloud computing, etc.
- Differences with clusters:
  - Clusters have higher performance processors and network
  - Clusters emphasize thread-level parallelism, WSCs emphasize request-level parallelism
- Differences with datacenters:
  - Datacenters consolidate different machines and software into one location
  - Datacenters emphasize virtual machines and hardware heterogeneity in order to serve varied customers

WSC design requirements

- Cost-performance: *Small savings add up*
- Energy efficiency: Affects power distribution and cooling
- Dependability via redundancy
- Network I/O
- Interactive and batch processing workloads
- Ample computational parallelism is not important
  - Most jobs are totally independent: “Request-level parallelism”
- Operational costs count
  - Power consumption is a primary constraint when designing system
- Scale and its opportunities and problems
  - Can afford customized systems since WSC require volume purchase

Datacenter vs. desktop

- Massive Parallelism:
  - Request level parallelism (not process)
  - Data parallelism: Read only data
- Platform homogeneity
  - Simplifies cluster-level scheduling and load balancing and
  - Reduces the maintenance burden
- Workload churn
- Fault-free operation

Workloads

- Web search
  - High throughput, large data processing for each request
- Web mail
  - More disk intensive
  - More disks per node, fewer servers/request
- Mapreduce
  - Cluster for offline batch MapReduce jobs
  - Shared by several users

- Workloads are well-tuned and run at high activity levels

Modern (Google) workloads

- Typically, software developed in-house – `MapReduce`, `BigTable, ...`

- `MapReduce`: parallel operations performed on very large datasets, e.g., search on a keyword, aggregate a count over several documents

- `Hadoop` is an open-source implementation of the MapReduce framework (more later)

Parallelism / Fault Tolerance

- High levels of “natural” parallelism
  - Both inter-query and intra-query
  - Near-linear speedups
- Replication helps both parallelism and fault tolerance
  - Permits software-based fault tolerance (The requirements for fault tolerance aren’t very high)
- Most accesses are read-only
  - Combined w/ replication, simplifies updates
  - Queries are diverted during update

MapReduce

- Application-writer provides Map and Reduce functions operating on key-value pairs
- Each map function operates on a collection of records; a record is (say) a webpage or a facebook user profile
- The records are in the file system and scattered across several servers; thousands of map functions are spawned to work on all records in parallel
- The Reduce function aggregates and sorts the results produced by the Mappers, also performed in parallel

MapReduce Framework

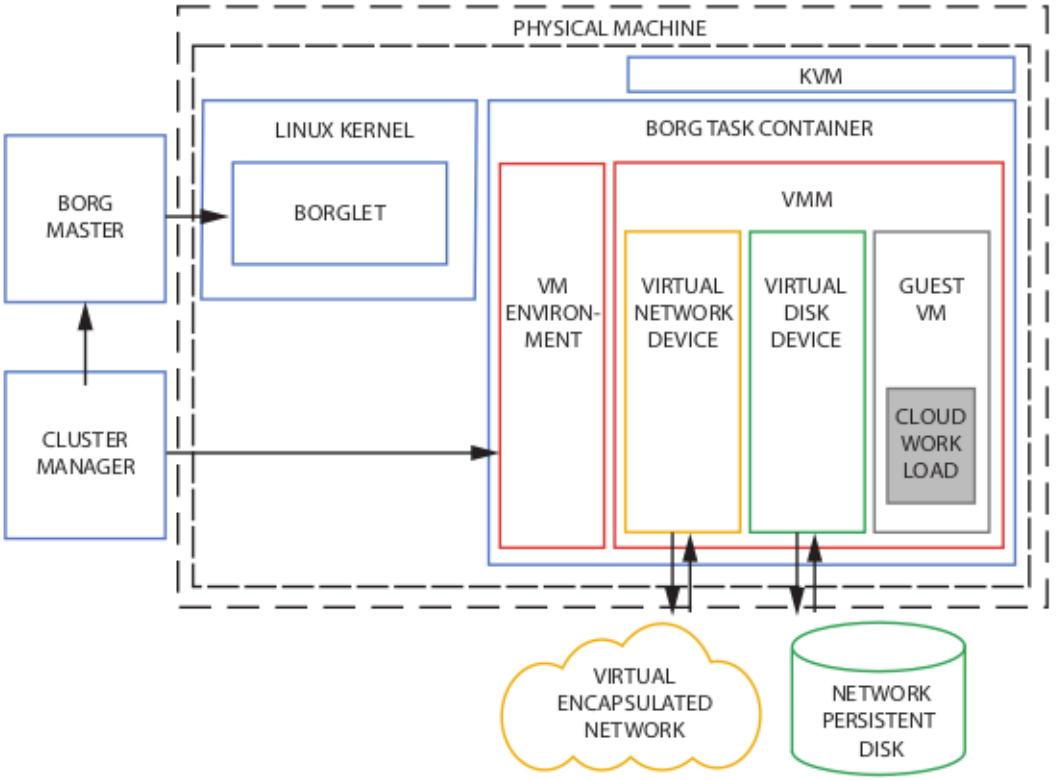
- Replicate data for fault tolerance
- Detect failed threads and re-start threads
- Handle variability in thread response times
- Use of MapReduce within Google has been growing every year:

  Aug'04 → Sep'09

  - Number of MR jobs has increased 100x+
  - Data being processed has increased 100x+
  - Number of servers per job has increased 3x

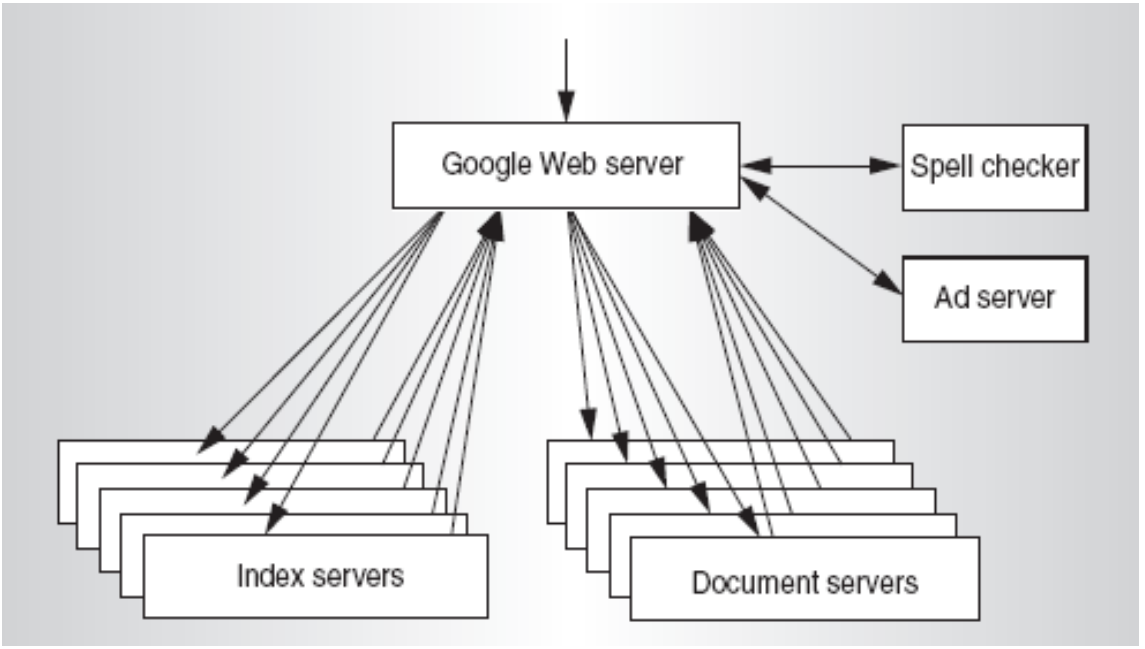
Cloud Computing

- Virtual Machine model
- I/O virtualization
- Large scale availability
- Resource “isolation” (distributed hosts)



Google server architecture

- Provide reliability in software
  - Use cluster of PCs
  - Not high-end servers
- Design system to maximize aggregate throughput
  - Not server response time (Can parallelize individual requests)



Machine crashes

- DRAM soft errors: 1.3% of all machines *per year*
- Disk errors: 2% and 4% in large field studies

Google server computer (ISA)

- Data is percent per instruction
  - Branch mispredict – 50 per 1K insts.
  - L1 I miss: 4 per 1K instructions
  - L1 D miss: 7 per 1K instructions
- CPI isn't bad
  - Branch mispredict rate is not very good
  - Data organized for optimal searches will likely have this property
- Chip multiprocessor or multithreading
  - ILP is limited (1 mispredict per 20 insts)
  - Pentium 4 has twice the CPI and similar branch prediction performance
  - Observe 30% speedup w/ hyperthreading (at high end of Intel projections)

Table 1. Instruction-level measurements on the index server.

Characteristic	Value
Cycles per instruction	1.1
Ratios (percentage)	
Branch mispredict	5.0
Level 1 instruction miss*	0.4
Level 1 data miss*	0.7
Level 2 miss*	0.3
Instruction TLB miss*	0.04
Data TLB miss*	0.7

* Cache and TLB ratios are per instructions retired.

Google server computer (memory)

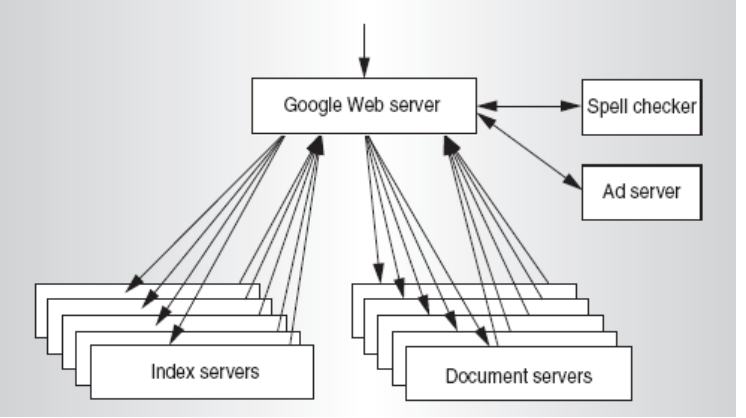
- Good temporal locality for instructions
  - Loops doing searches
- Data blocks
  - Very low temporal locality
  - Good spatial locality within index data block

    So data cache performance is relatively good.
- Memory bandwidth not a bottleneck
  - Memory bus about 20% utilized
  - Relatively high computation per data item

    (or maybe just high mis-predicts per data item)
  - Modest L2 cache lines seem to be a good design point

Google scheme

- Geographically distributed clusters
  - Each with a few thousand machines
- First perform Domain Name System (DNS) lookup
  - Maps request to nearby cluster
- Send HTTP request to selected cluster
  - Request serviced locally w/in that cluster
- Clusters consist of Google Web Servers (GWSes)
  - Hardware-based load balancer distributes load among GWSes



Google rack cluster

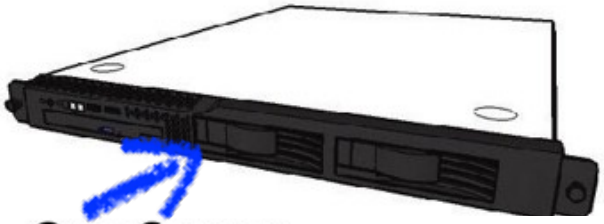
- Performance/Price beats pure Performance
  - Use dual CPU servers rather than quad
  - Use IDE rather than SCSI (cheaper)
- Use commodity PCs
  - Essentially mid-range PCs except for disks
  - No redundancy as in many high-end servers
- Use rack mounted clusters
  - Connect via Ethernet

Result is order of magnitude better performance/price than using high-end server components

More Google rack

- A rack can hold 48 1U servers
  - 1U is 1.75 inches high and is the maximum height for a server unit
- A rack switch is used for communication within and out of a rack; an array switch connects an array of racks
- Latency grows if data is fetched from remote DRAM or disk (300us vs. 0.1us for DRAM and 12ms vs. 10ms for disk)
- Bandwidth within a rack is much higher than between arrays → software must be aware of data placement and locality

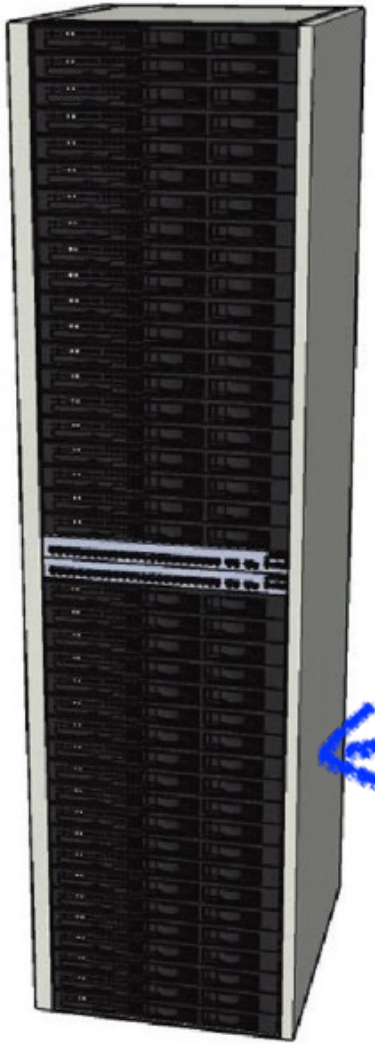
Rack diagram

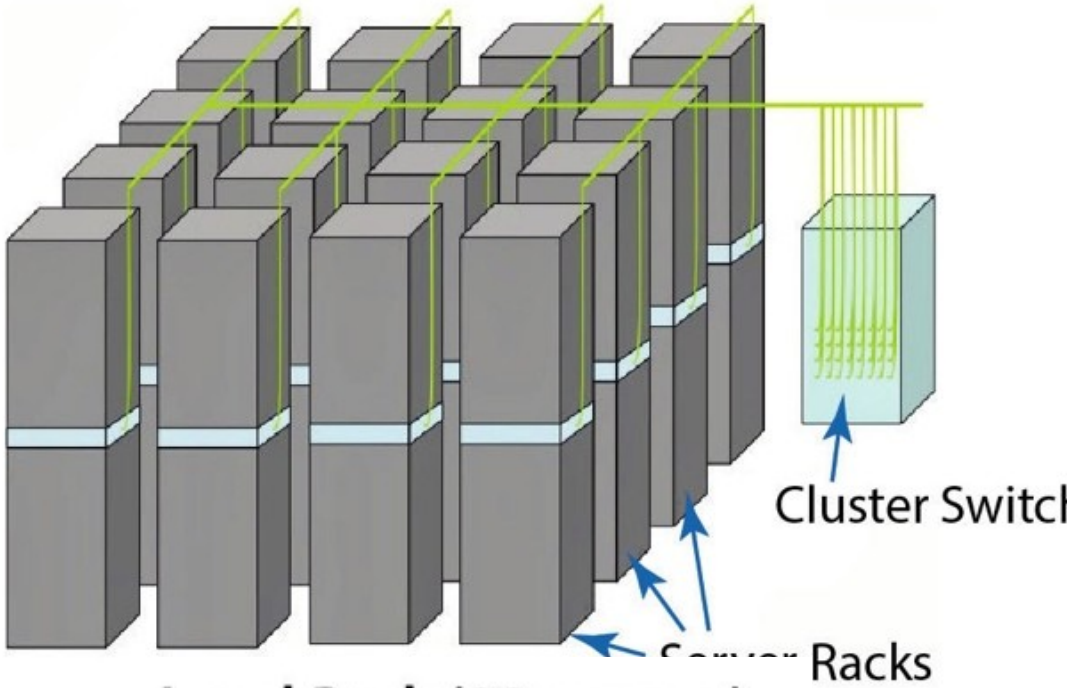


One Server

DRAM: 16 GB, 100 ns, 20 GB/s
Disk: 2T B, 10 ms, 200 MB/s
Flash: 128 GB, 100 us, 1 GB/s

2 sockets/server, 2 cores/socket.



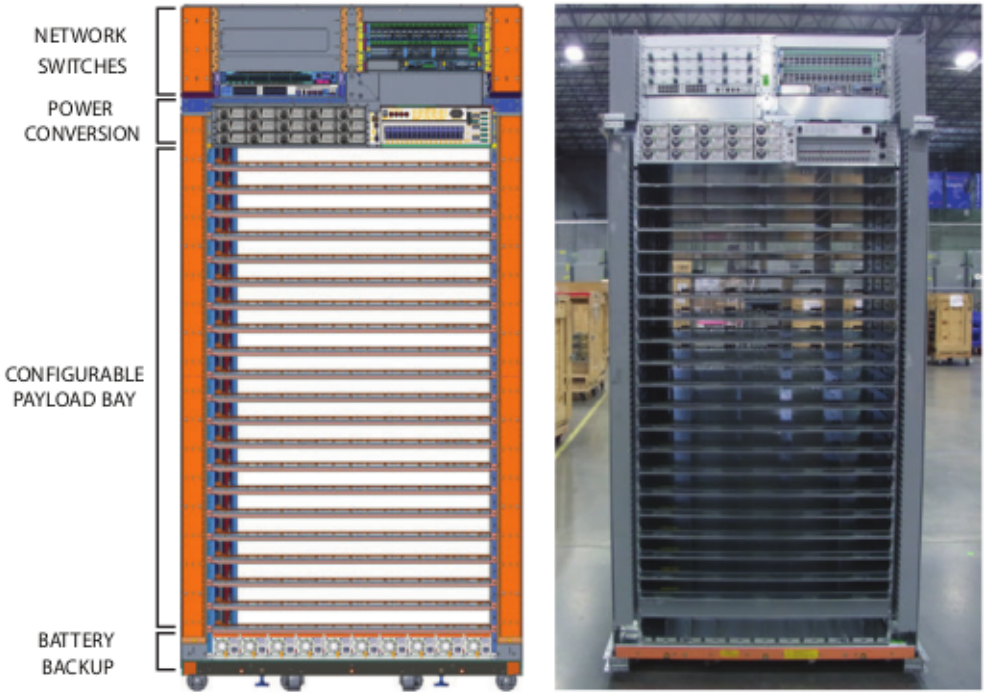


Local Rack (80 servers)

DRAM: 1 TB, 300 us, 100 MB/s
Disk: 160 TB, 11 ms, 100 MB/s
Flash: 20 TB, 400 us, 100 MB/s

30$/port GigE network switch

Rack schematic



Array of Racks

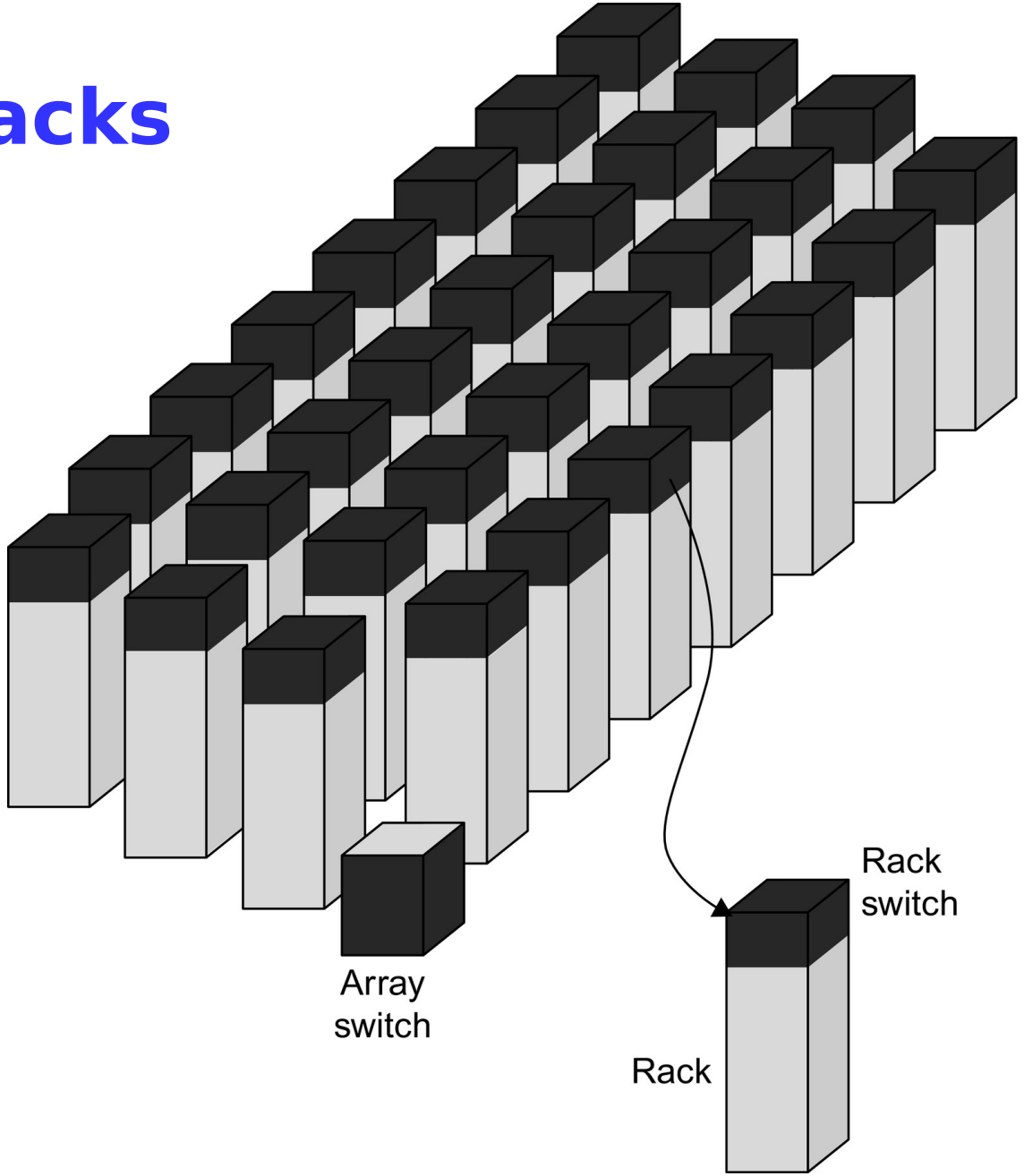


Figure 6.5 Hierarchy of switches in a WSC. Based on Figure 1.1 in Barroso, L.A., Clidaras, J., Hölzle, U., 2013. The datacenter as a computer: an introduction to the design of warehouse-scale machines. Synth. Lect. Comput. Architect. 8 (3), 1–154.

© 2019 Elsevier Inc. All rights reserved.

Older WCS Network Structure

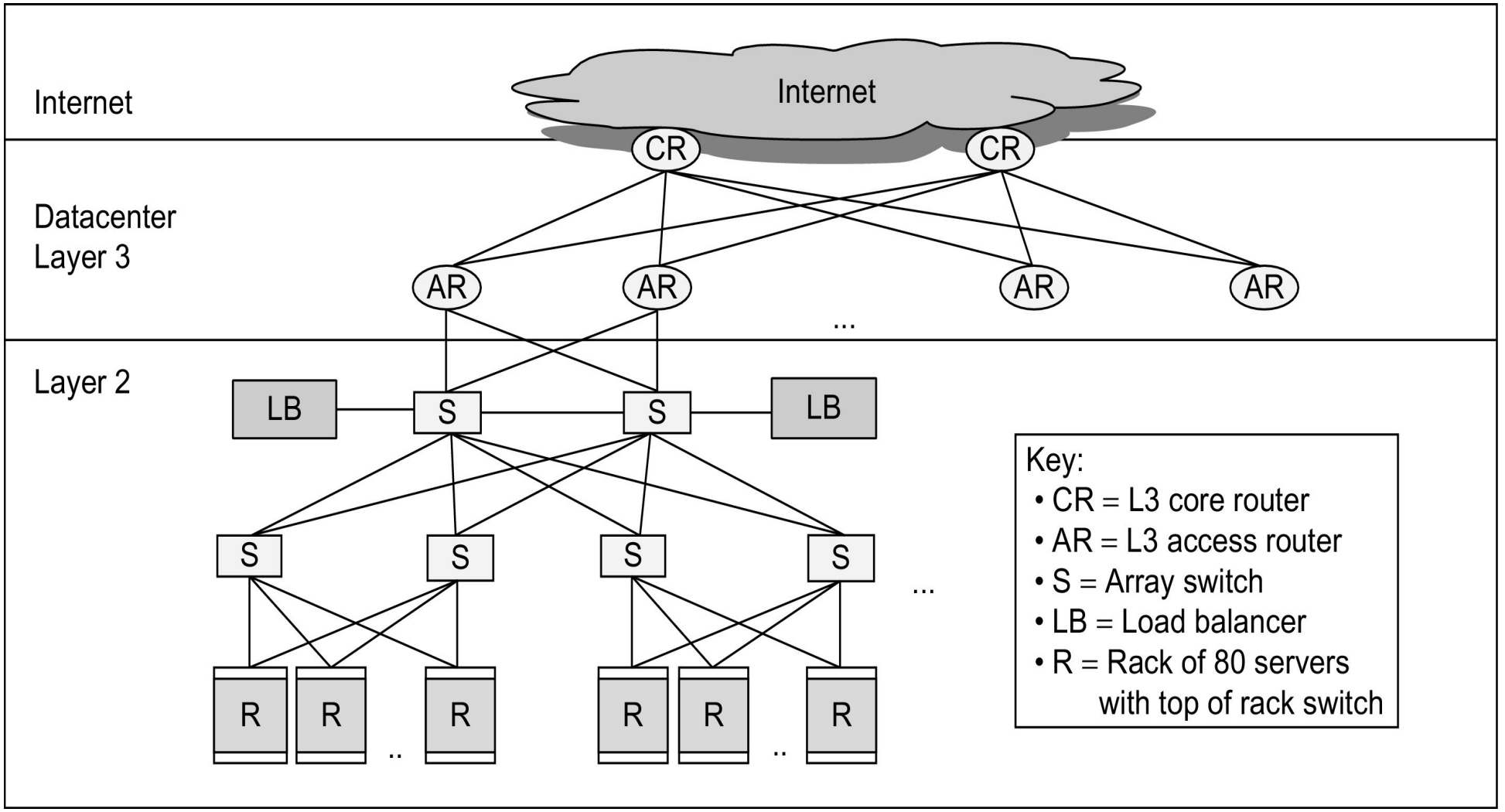


Figure 6.8 A Layer 3 network used to link arrays together and to the Internet (Greenberg et al., 2009). A load balancer monitors how busy a set of servers is and directs traffic to the less loaded ones to try to keep the servers approximately equally utilized. Another option is to use a separate *border router* to connect the Internet to the data center Layer 3 switches. As we will see in Section 6.6, many modern WSCs have abandoned the conventional layered networking stack of traditional switches.

© 2019 Elsevier Inc. All rights reserved.

Array Switch

- Switch that connects an array of racks
  - Array switch should have 10 X the bisection bandwidth of rack switch
  - Cost of n-port switch grows as n^2
  - Often utilize content addressible memory chips and FPGAs

Newer Clos Network Structure

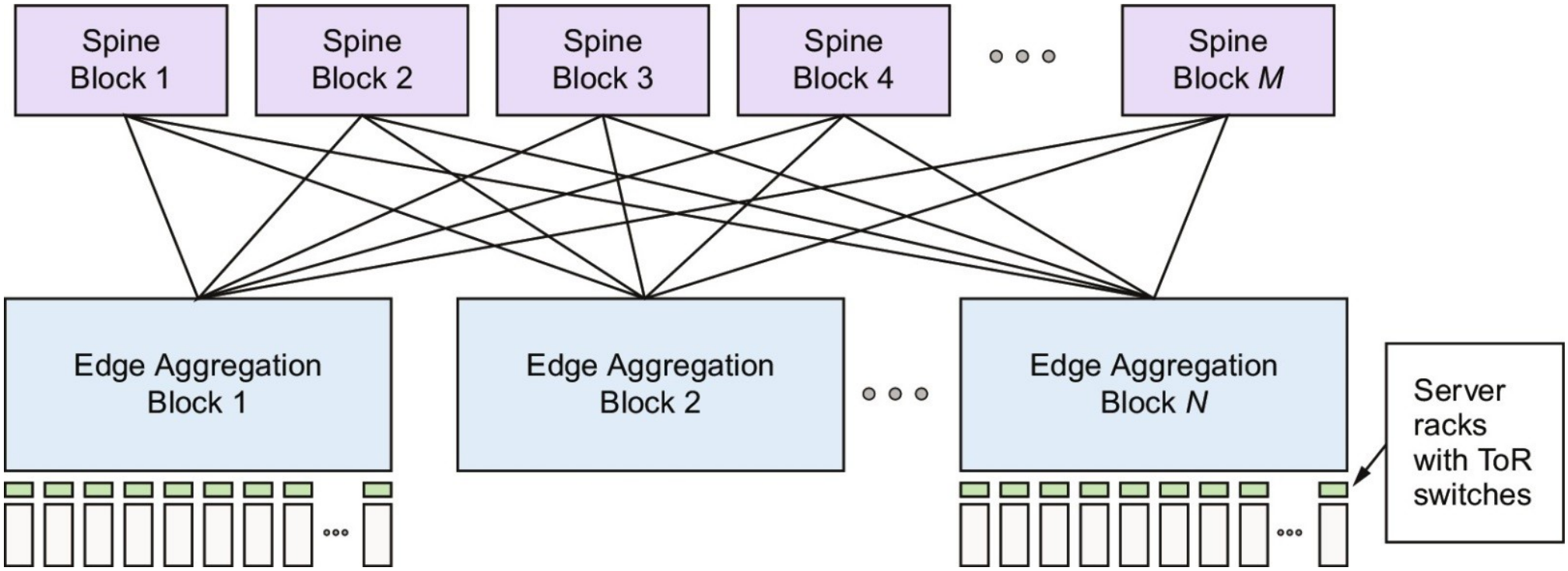
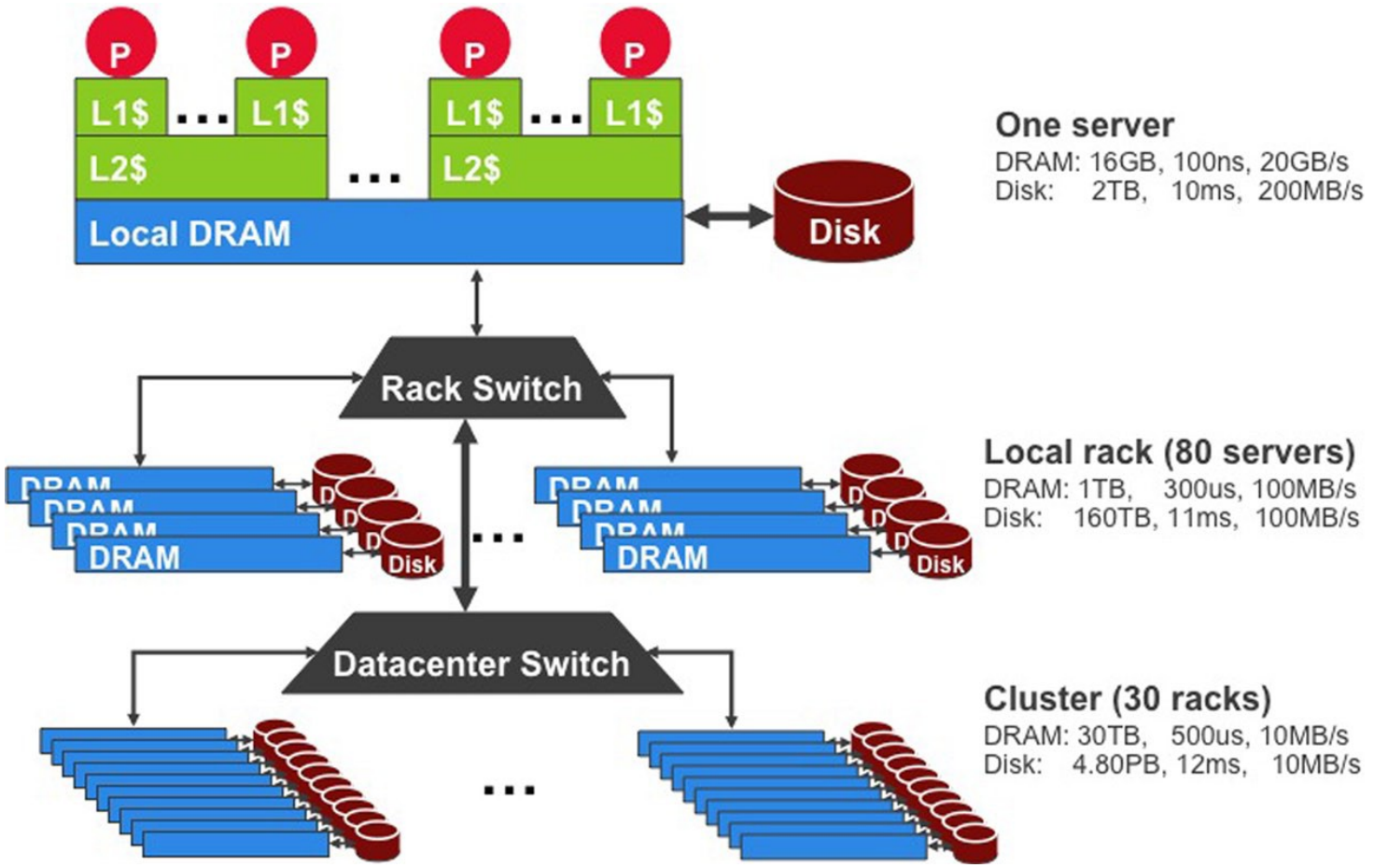


Figure 6.31 A Clos network has three logical stages containing crossbar switches: ingress, middle, and egress. Each input to the ingress stage can go through any of the middle stages to be routed to any output of the egress stage. In this figure, the middle stages are the *M* Spine Blocks, and the ingress and egress stages are in the *N* Edge Activation Blocks. Figure 6.22 shows the changes in the Spine Blocks and the Edge Aggregation Blocks over many generations of Clos networks in Google WSCs.

© 2019 Elsevier Inc. All rights reserved.

Storage hierarchy



WSC Memory Hierarchy

- **Servers can access DRAM and disks on other servers using a NUMA-style interface**

	Local	Rack	Array
DRAM latency (µs)	0.1	300	500
Flash latency (µs)	100	400	600
Disk latency (µs)	10,000	11,000	12,000
DRAM bandwidth (MB/s)	20,000	100	10
Flash bandwidth (MB/s)	1000	100	10
Disk bandwidth (MB/s)	200	100	10
DRAM capacity (GB)	16	1024	31,200
Flash capacity (GB)	128	20,000	600,000
Disk capacity (GB)	2000	160,000	4,800,000

Storage options

- Use disks inside the servers, or
- Network attached storage through Infiniband

- WSCs generally rely on local disks
- Google File System (GFS) uses local disks and maintains at least three replicas

Latency, bandwidth and capacity

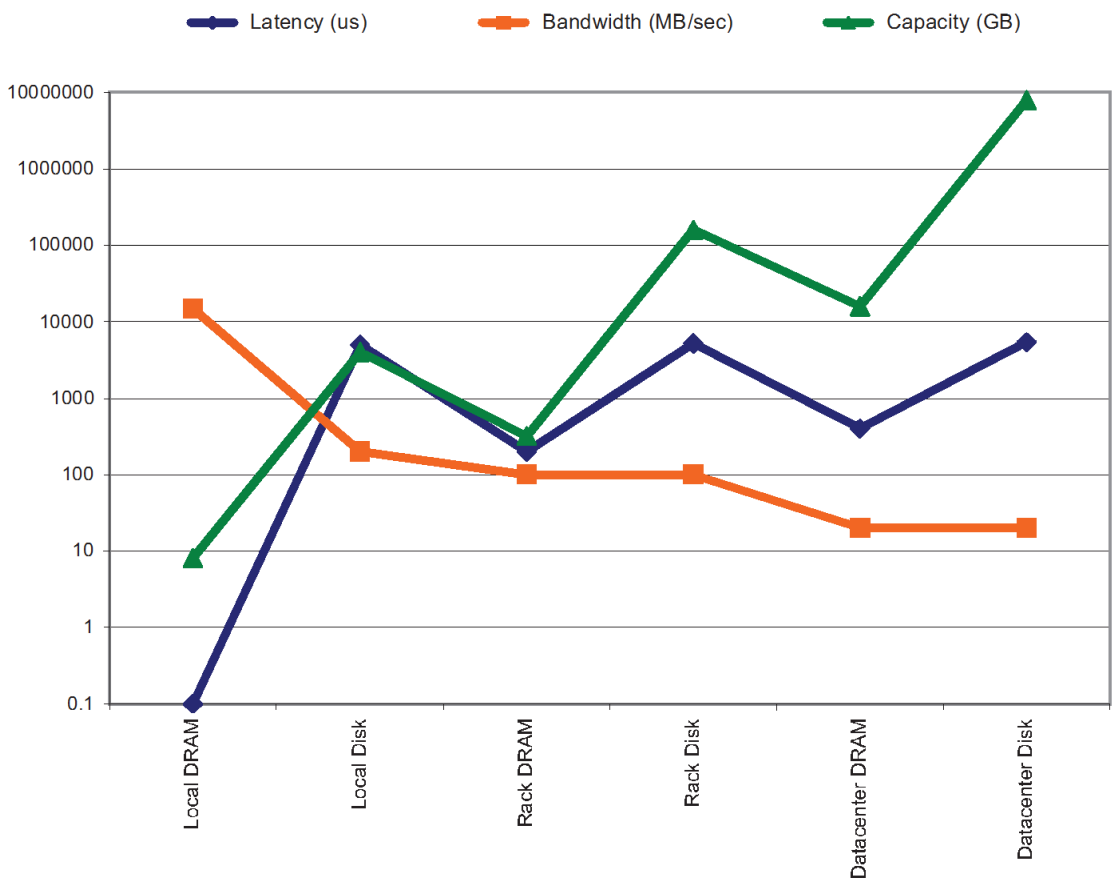


FIGURE 1.3: Latency, bandwidth, and capacity of a WSC.

Power and cooling requirements

- Cooling system also uses water (evaporation and spills)
  - E.g. 70,000 to 200,000 gallons per day for an 8 MW facility
- Power cost breakdown:
  - Chillers: 30-50% of the power used by the computing equipment
  - Air conditioning: 10-20% of the computing power, mostly due to fans
- How many servers can a WSC support?
  - Each server:
    - “Nameplate power rating” gives maximum power consumption
    - To get actual, measure power under actual workloads
  - Oversubscribe cumulative server power by 40%, but monitor power closely

Datacenter construction

- Datacenters are expensive
  - Construction cost rivals that of the energy cost over a datacenter's lifetime
  - Operating as close to maximum capacity is important to amortize construction cost
- Over-provisioning allows them to stretch to the power budget (With safety valves in case they mess up)
- Three main complications:
  1) Actual power consumption of machines is less than advertised
  2) Power varies with workload activity
  3) Different workloads use different amounts of power

Datacenter layout

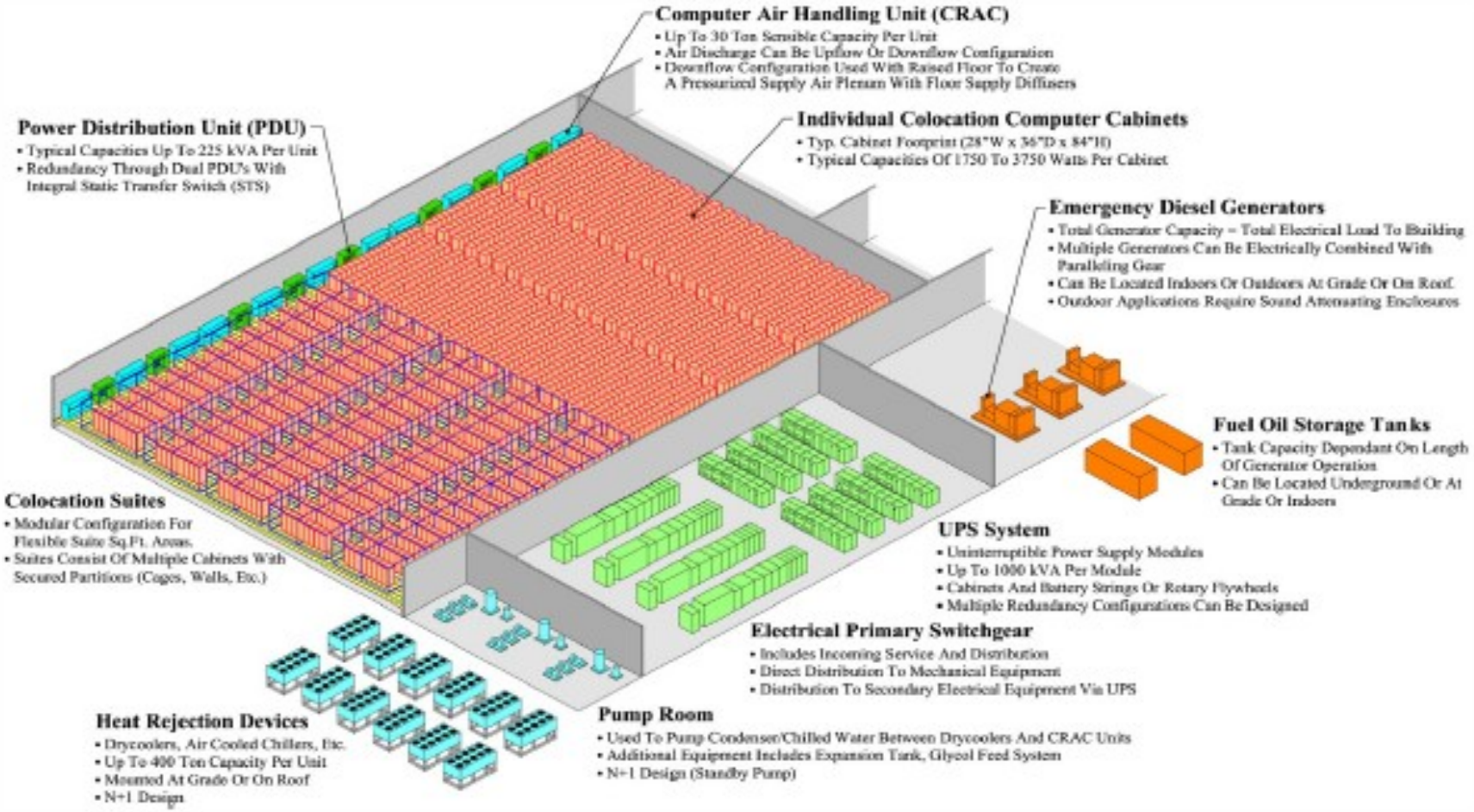


FIGURE 4.1: The main components of a typical datacenter (image courtesy of DLB Associates [23]).

Power efficiency

115kv

0.3% loss
99.7% efficient

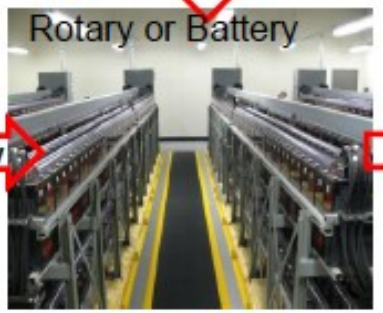
13.2kv

8% distribution loss
.997^3*.94*.99 = 92.2%

2.5MW Generator
~180 Gallons/hour

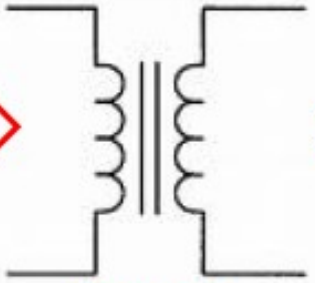
13.2kv

UPS:
Rotary or Battery



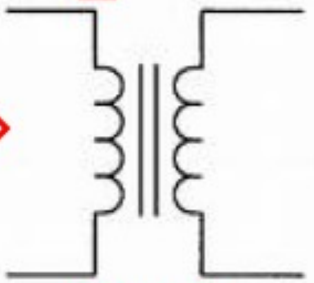
6% loss
94% efficient, >97% available

13.2kv



0.3% loss
99.7% efficient

480V



0.3% loss
99.7% efficient

208V

IT LOAD

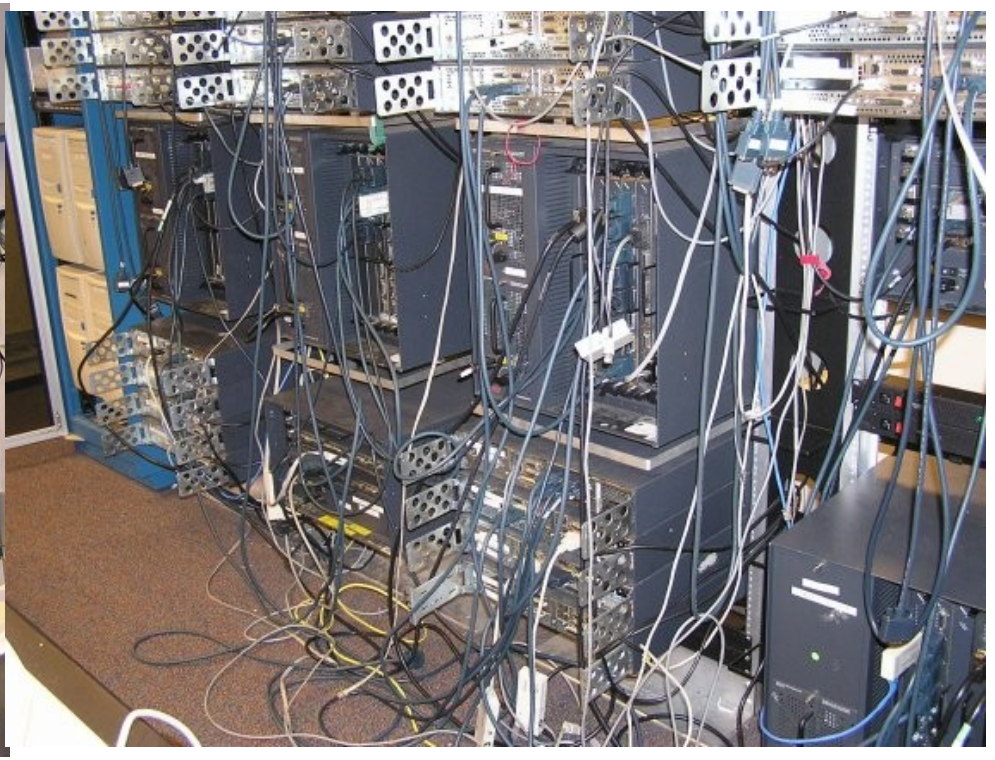
~1% loss in switch
Gear and conductors

Networking and Power

- Within Datacenter racks, network equipment often the “hottest” components in the hot spot
- Network opportunities for power reduction
  - Transition to higher speed interconnects (10 Gbs) at DC scales and densities
  - High function/high power assists embedded in network element

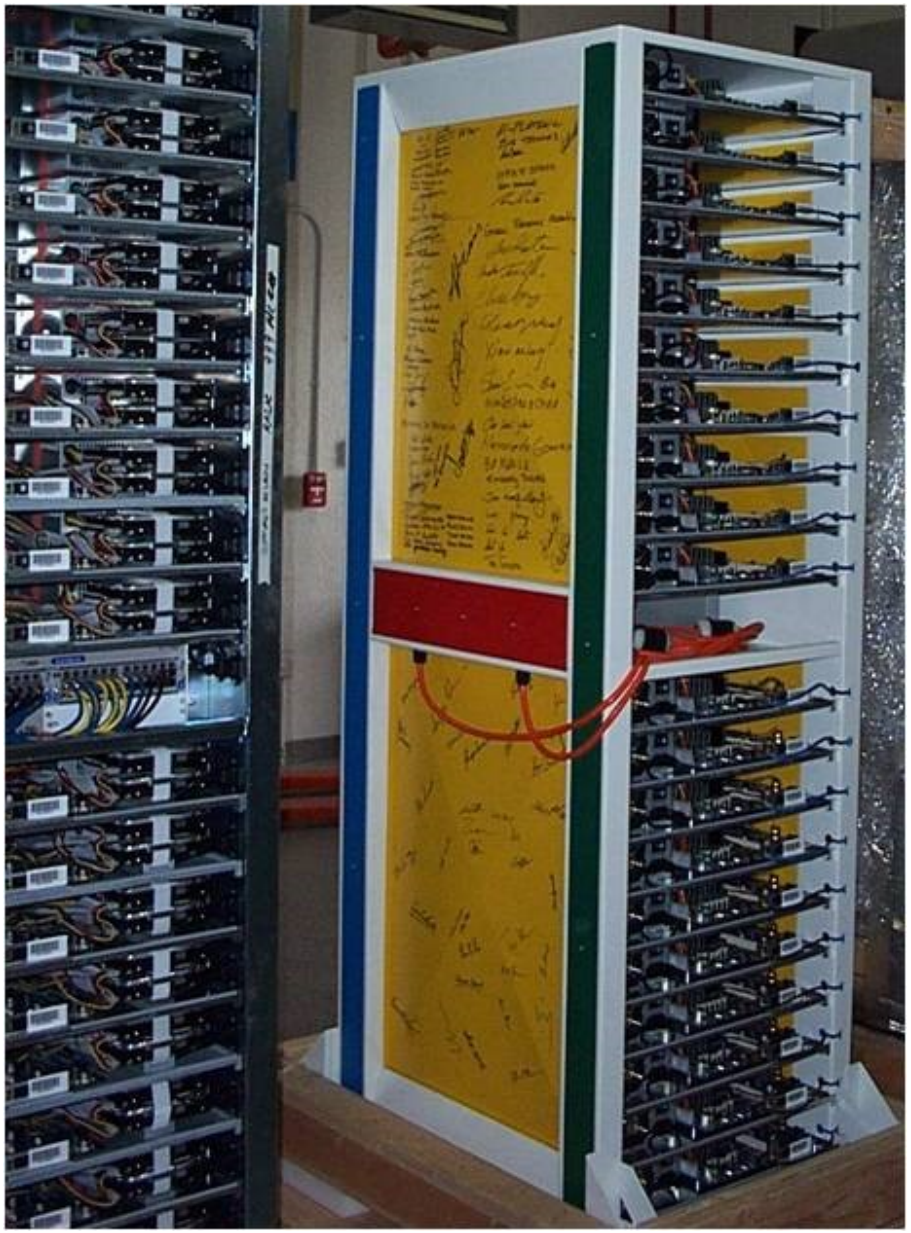
Networking and power

- 96 x 1 Gbit port Cisco datacenter switch consumes around 15 kW -- approximately 100x a typical dual processor Google server @ 145 W
- High port density drives network element design, but such high power density makes it difficult to tightly pack them with servers

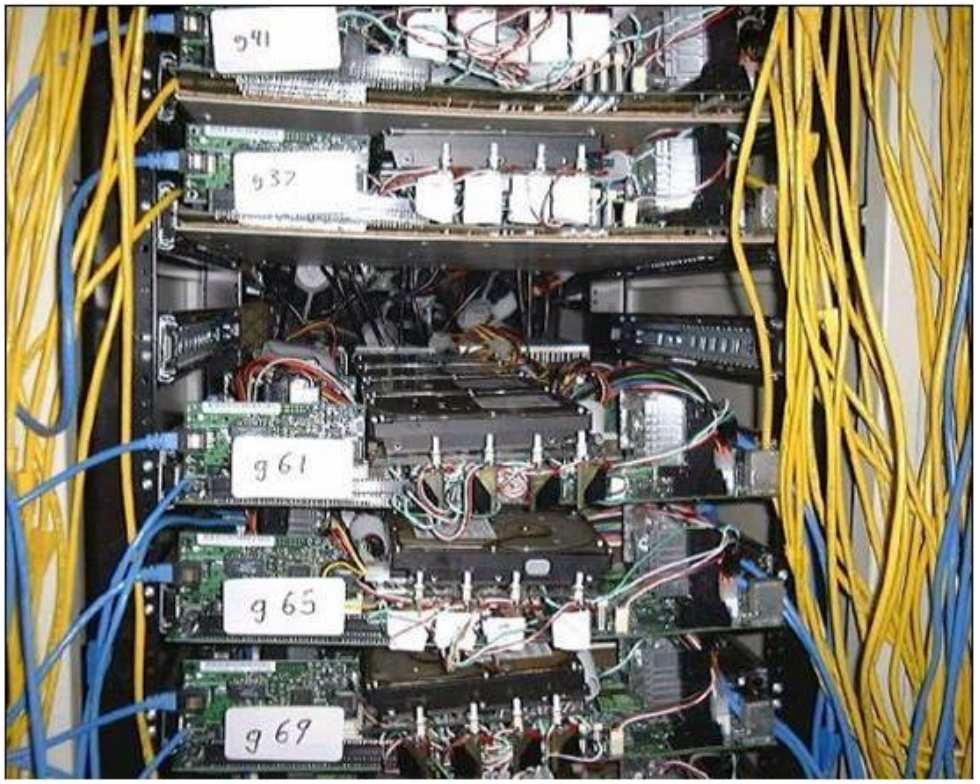


Google 1997

Early Google Server Rack



Google 2001



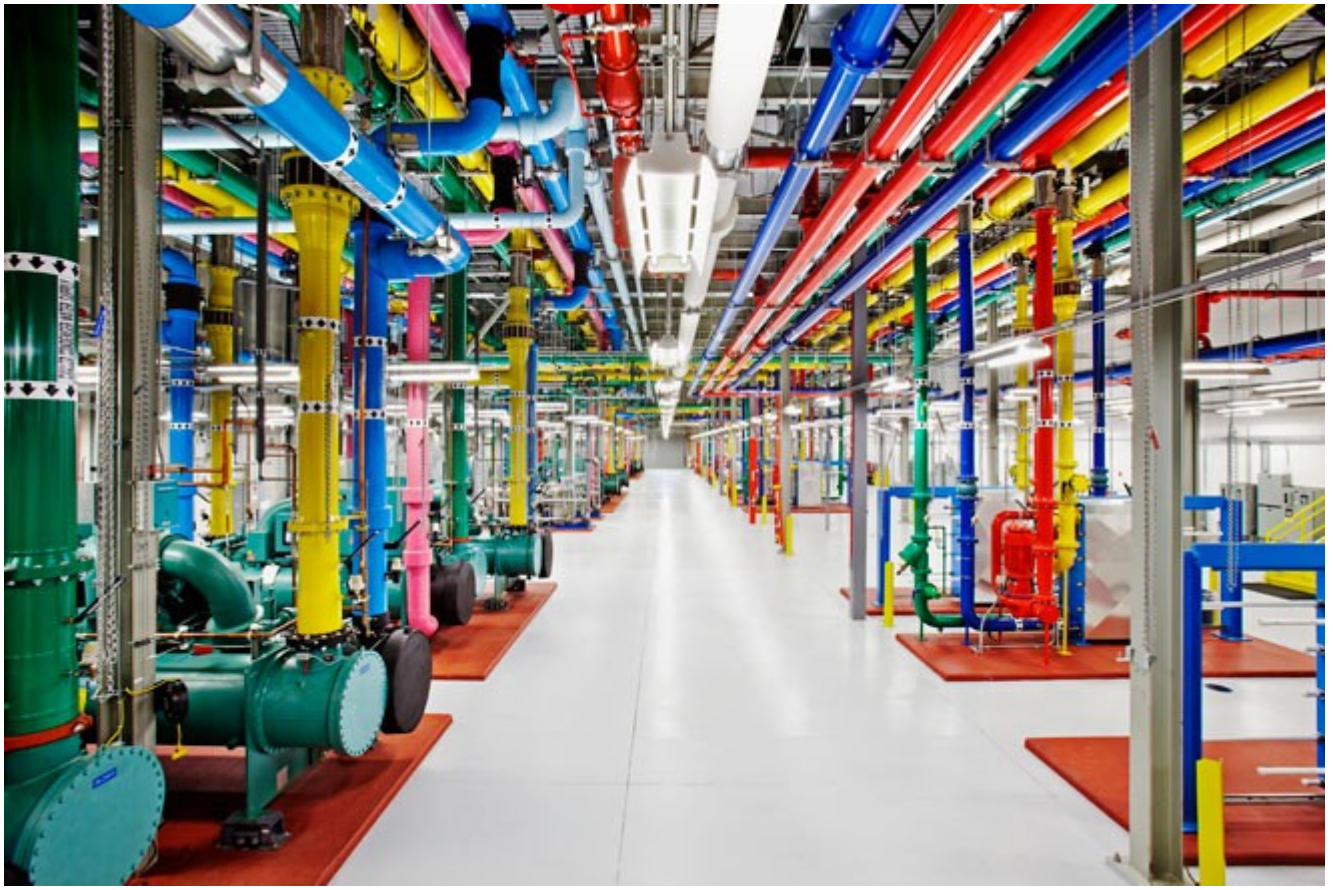
Commodity CPUs

Lots of disks

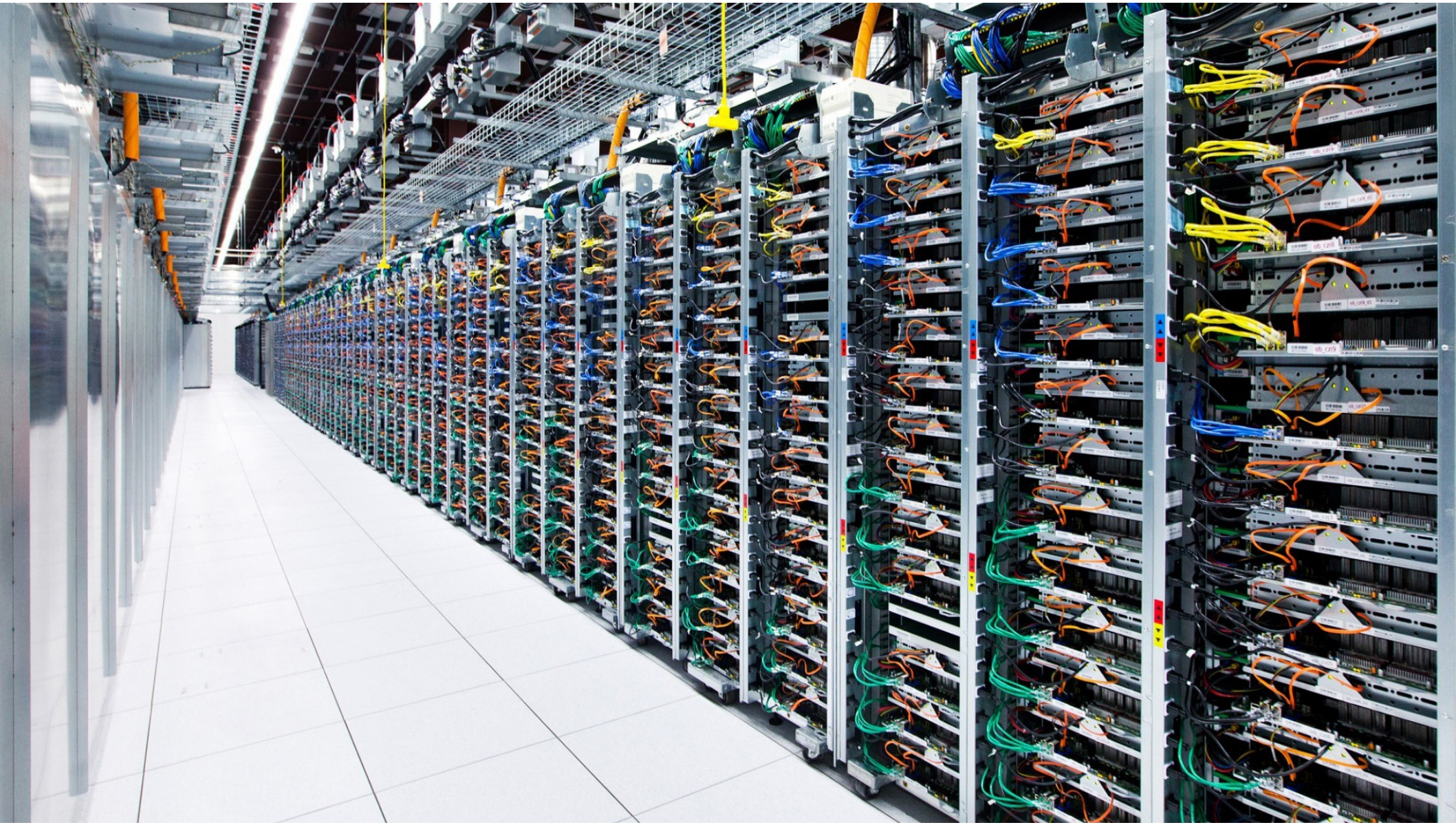
Low bandwidth network

Cheap !

Olde Google datacenter cooling



Olde Google datacenter

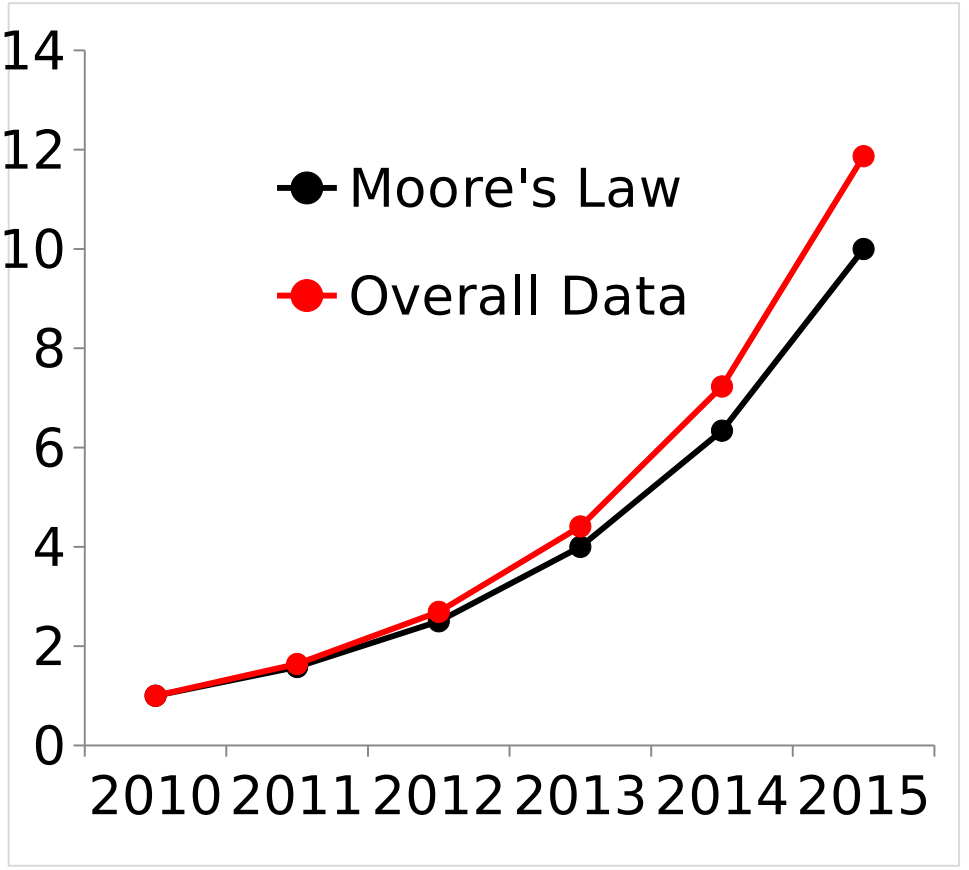


Datacenter evolution

Facebook’s daily logs: 60 TB

1000 genomes project: 200 TB

Google web index: 10+ PB



(IDC report*)

Slide from Ion Stoica

Case study

- 8 MW facility : facility cost: $88M, server/networking cost: $79M

- Monthly expense: $3.8M. Breakdown:
  - Servers 53% (amortized CapEx)
  - Networking 8% (amortized CapEx)
  - Power/cooling infrastructure 20% (amortized CapEx)
  - Other infrastructure 4% (amortized CapEx)

  - Monthly power bill 13% (true OpEx)
  - Monthly personnel salaries 2% (true OpEx)

Other metrics

- Performance does matter, especially latency
- Analysis of Bing search engine: Given a 200ms delay, the next click by the user is delayed by 500ms → a poor response time amplifies the user's non-productivity
- Reliability (MTTF) and Availability (MTTF/MTTF+MTTR) are very important, given the large scale
- A server with MTTF of 25 years : 50K servers → 5 server failures a day; Annual disk failure rate is 2-10% → 1 disk failure every hour

Datacenter Arms Race

Amazon, Google, Microsoft, Facebook, … race to build next-generation mega-datacenters

- **Industrial-scale Information Technology**
- **100,000+ servers**
- **Located where land, water, fiber-optic connectivity, and cheap power are available**

Microsoft Quincy

- **43600 sq. ft. (10 football fields), sized for 48 MW**
- **Also Chicago, San Antonio, Dublin @$500M each**

Google:

- **The Dalles OR, Pryor OK, Council Bluffs, IW, Lenoir NC, Goose Creek , SC**

Amazon EC2

Machine	Memory (GB)	Compute Units (ECU)	Local Storage (GB)	Cost / hour
t1.micro	0.615	2	0	$0.02
m1.xlarge	15	8	1680	$0.48
cc2.8xlarge	60.5	88 (Xeon 2670)	3360	$2.40

1 ECU = CPU capacity of a 1.0-1.2 GHz 2007 Opteron or 2007 Xeon processor

The Joys of Real Hardware

Typical first year for a new cluster:

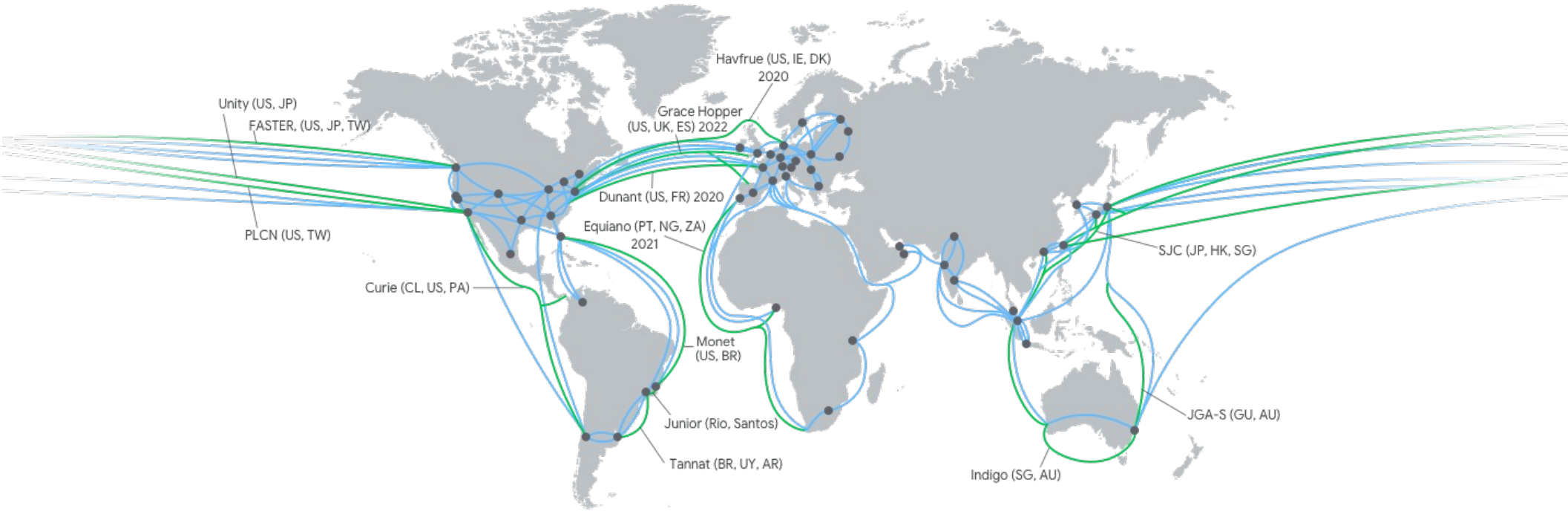
~0.5 overheating (power down most machines in <5 mins, ~1-2 days to recover)
~1 PDU failure (~500-1000 machines suddenly disappear, ~6 hours to come back)
~1 rack-move (plenty of warning, ~500-1000 machines powered down, ~6 hours)
~1 network rewiring (rolling ~5% of machines down over 2-day span)
~20 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
slow disks, bad memory, misconfigured machines, flaky machines, etc.

Long distance links: wild dogs, sharks, dead horses, drunken hunters, etc.

Jeff Dean @ Google

Google worldwide datacenters

Google worldwide network



Google, The Dalles, Oregon

Datacenter Reality

Google data center in The Dalles, Oregon

Capacity:
~10000 machines

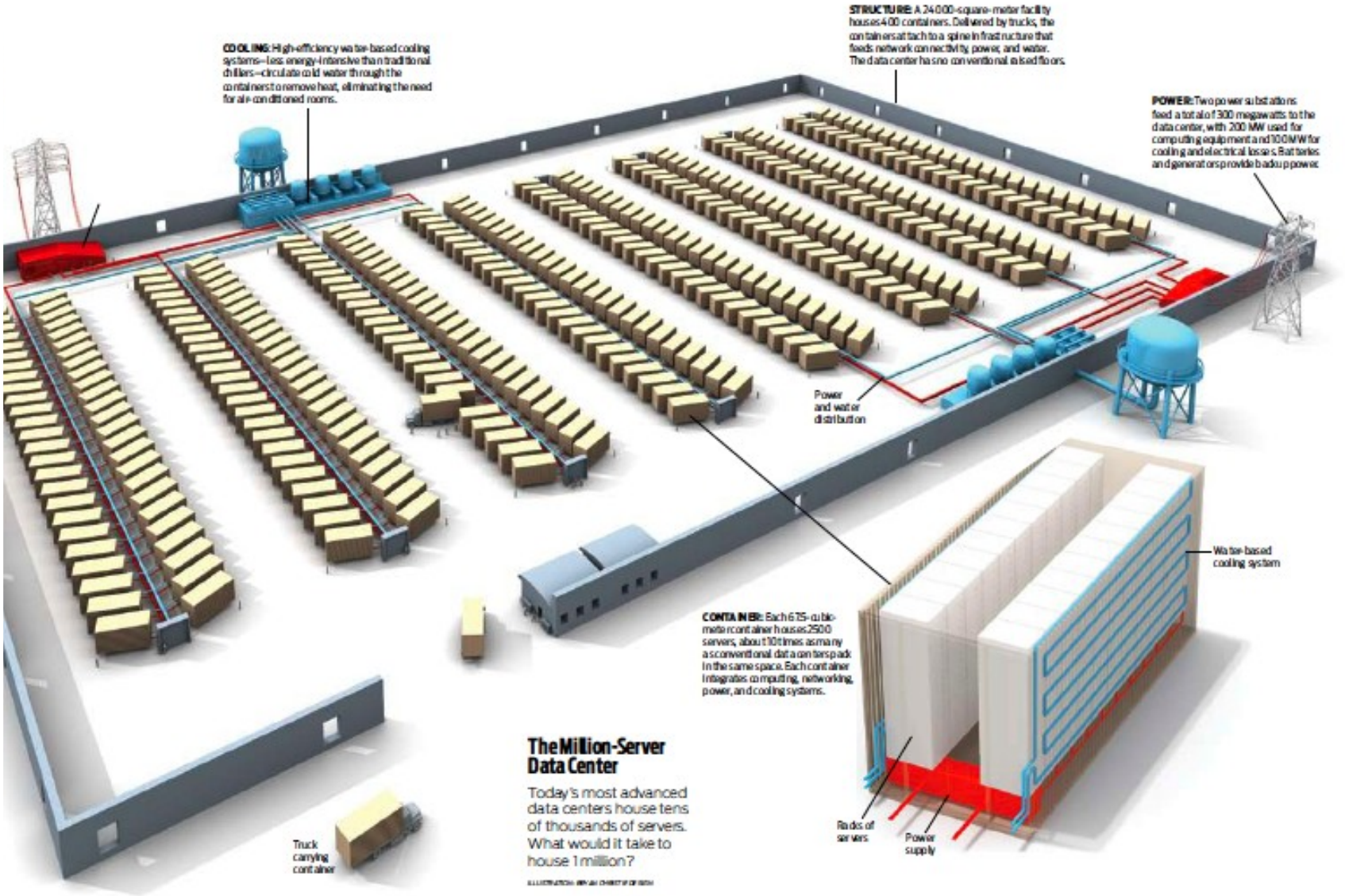
Bandwidth:
12-24 disks per node

Latency:
256GB RAM cache

Quincy, WA datacenters

Microsoft Datacenter (bing,linkedin,box,office)

Microsoft Chicago



Programming Models

Message Passing Models (MPI)

Fine-grained messages + computation

Hard to deal with disk locality, failures, stragglers

1 server fails every 3 years →

10K nodes see 10 faults/day

Programming Models

Data Parallel Models

Restrict the programming interface

Automatically handle failures, locality etc.

"Here's an operation, run it on all of the data"

- I don't care *where* it runs (you schedule that)
- In fact, feel free to run it; *retry* on different nodes

MapReduce

MapReduce: Simplified Data Processing on Large Clusters

Jeffrey Dean and Sanjay Ghemawat

jeff@google.com, sanjay@google.com

Google, Inc.

Abstract

ce is a programming model and an associ-
entation for processing and generating large
sers specify a *map* function that processes a
ir to generate a set of intermediate key/value
reduce function that merges all intermediate
iated with the same intermediate key. Many
sks are expressible in this model, as shown

given day, etc. Most such computations a
ally straightforward. However, the input da
large and the computations have to be distri
hundreds or thousands of machines in orde
a reasonable amount of time. The issues of
allelize the computation, distribute the data
failures conspire to obscure the original sir
tation with large amounts of complex code
these issues.

Google 2004

Build search index

Compute PageRank

Hadoop: Open-source
at Yahoo, Facebook

MapReduce Programming Model

Data type: Each record is (key, value)

Map function:

$$(K_{in}, V_{in}) \rightarrow \text{list}(K_{inter}, V_{inter})$$

Reduce function:

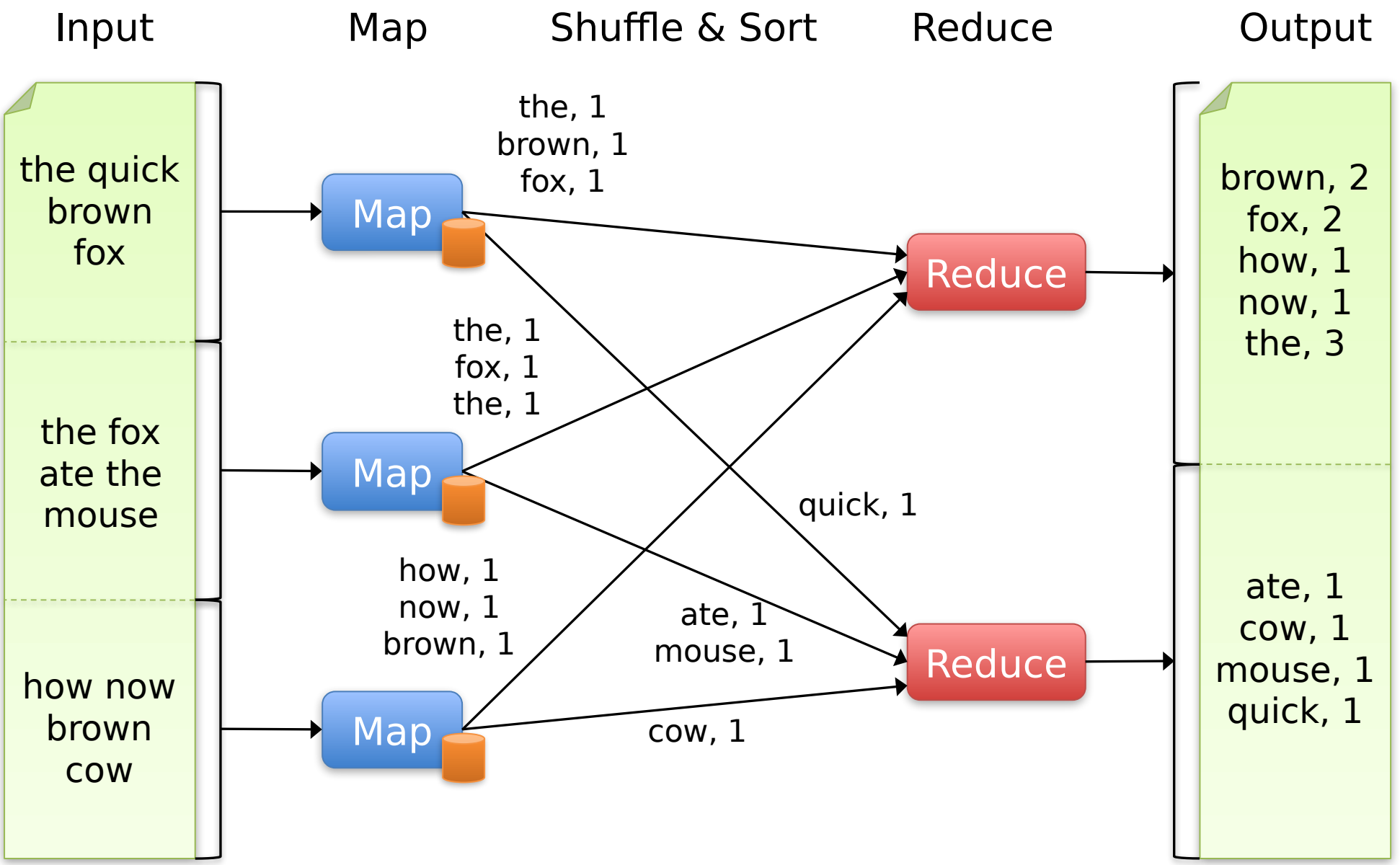
$$(K_{inter}, \text{list}(V_{inter})) \rightarrow \text{list}(K_{out}, V_{out})$$

Example: Word Count

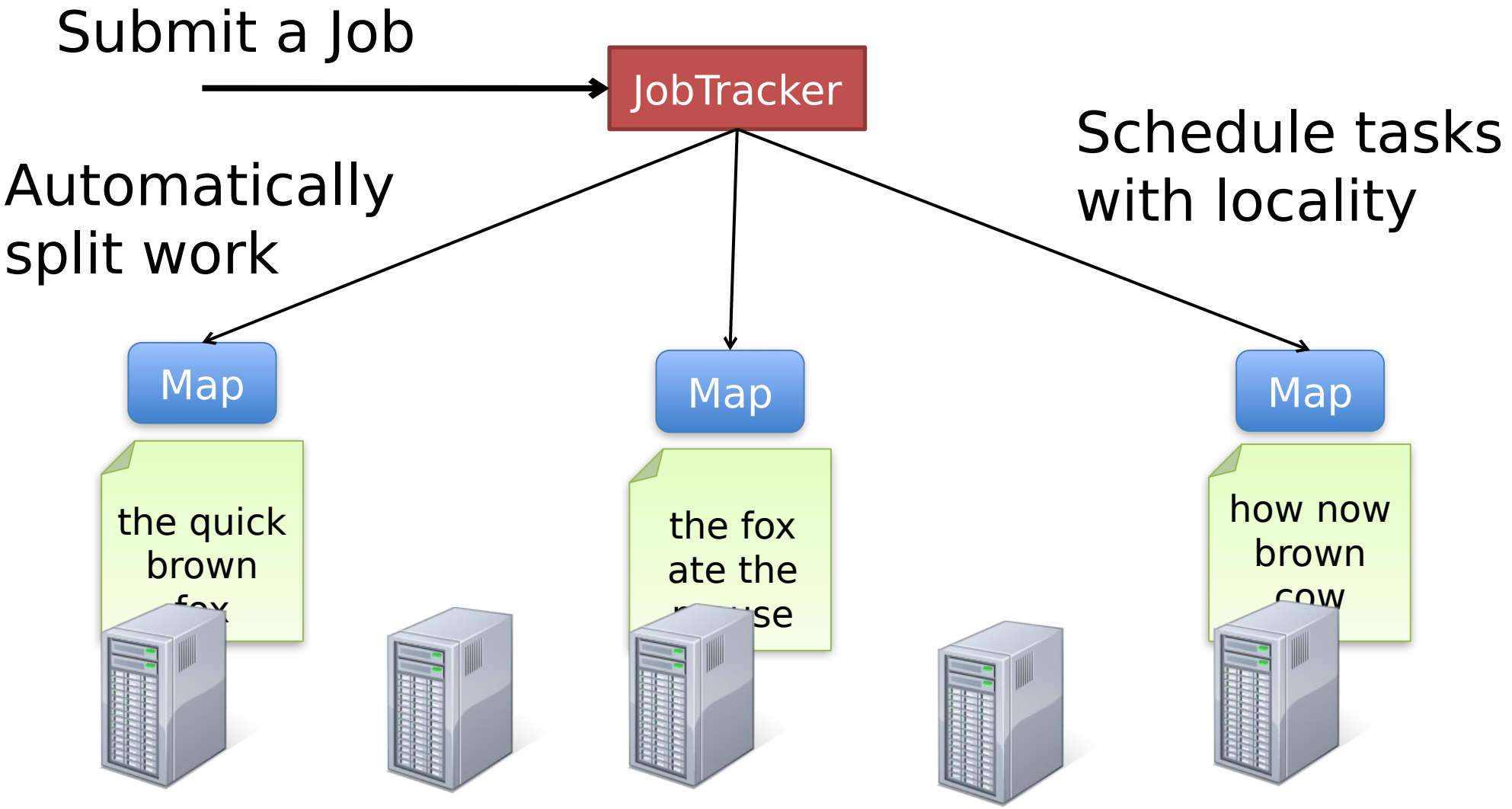
```
def mapper(line):
    for word in line.split():
        output(word, 1)


def reducer(key, values):
    output(key, sum(values))
```

Word Count Execution



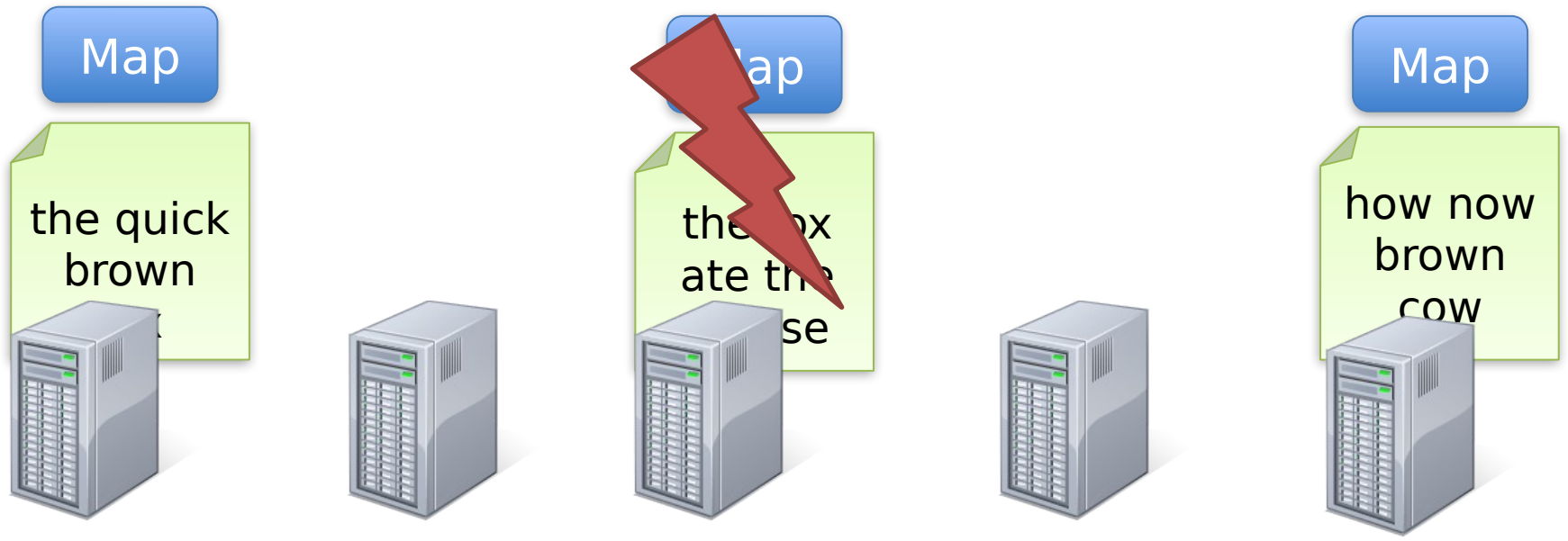
Word Count Execution



Fault Recovery

If a task crashes:

- Retry on another node
- If the same task repeatedly fails, end the job

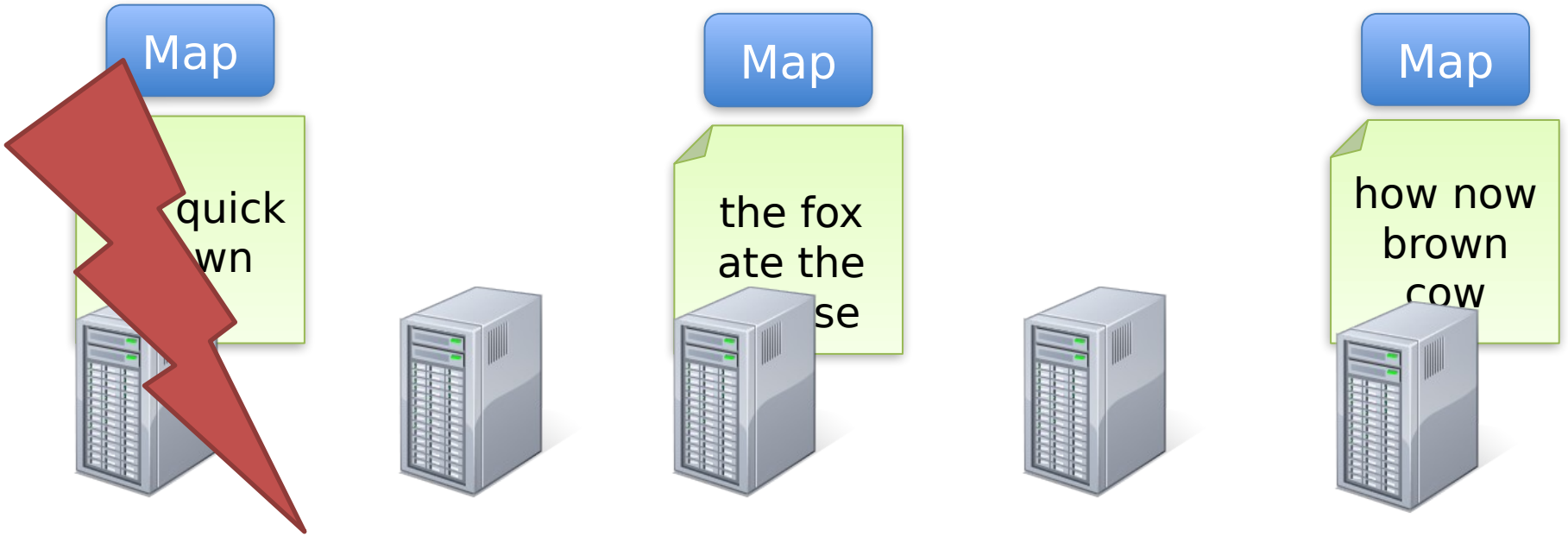


Requires user code to be **deterministic**

Fault Recovery

If a node crashes:

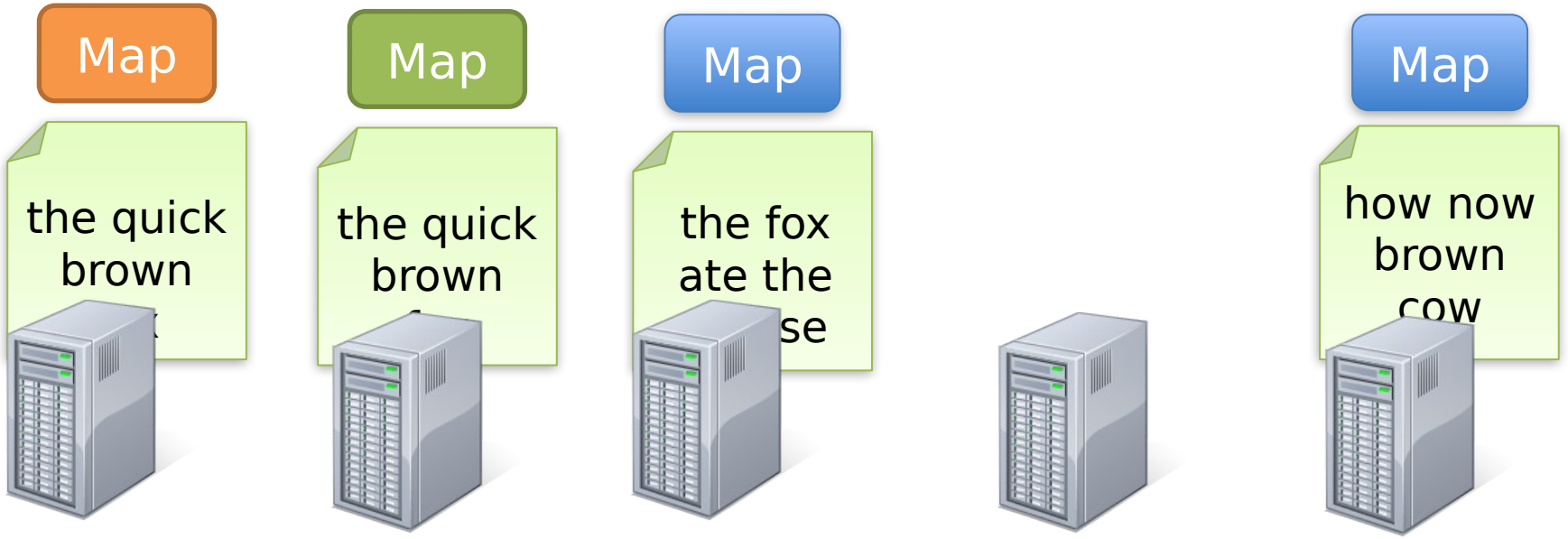
- Relaunch its current tasks on other nodes
- Relaunch tasks whose outputs were lost



Fault Recovery

If a task is going slowly (straggler):

- Launch second copy of task on another node
- Take the output of whichever finishes first



Application: Search

Input: (lineNumber, line) records

Output: lines matching a given pattern

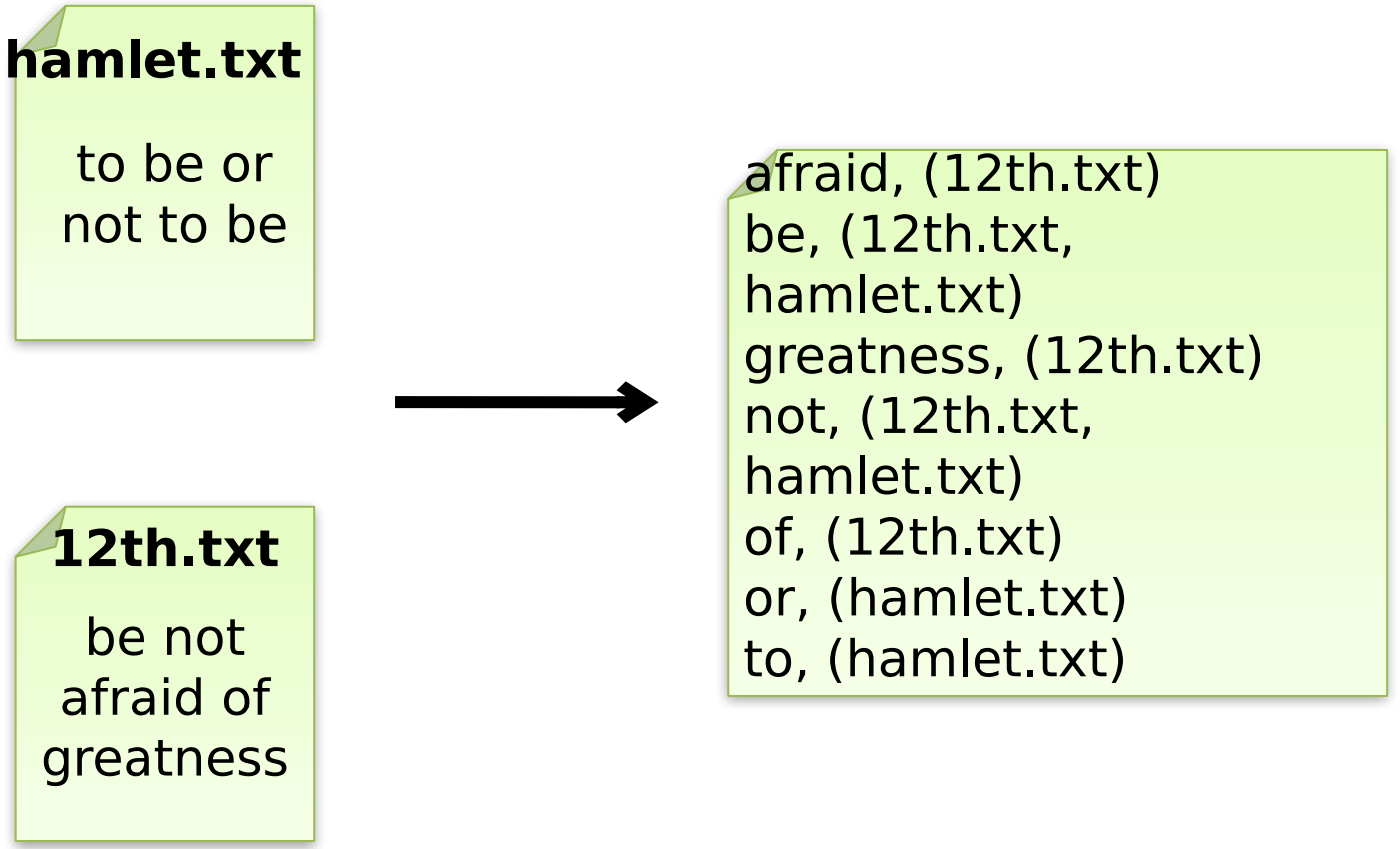
Map:

```
if(line matches pattern):
    output(line)
```

Reduce: Identity function

- Alternative: no reducer (map-only job)

Application: Inverted Index



Application: Inverted Index

Input: (filename, text) records

Output: list of files containing each word

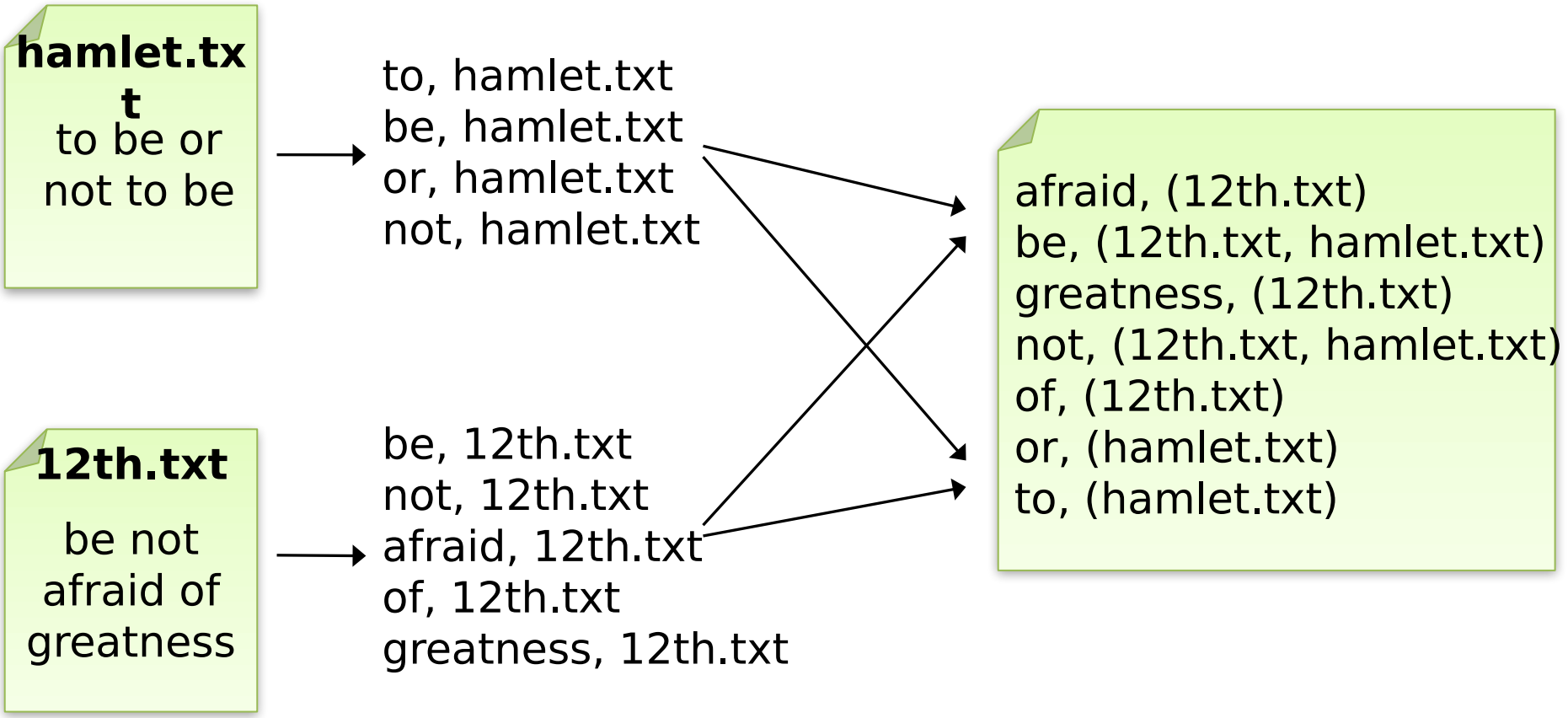
Map:

```
foreach word in text.split():
    output(word, filename)
```

Reduce:

```
def reduce(word, filenames):
    output(word, unique(filenames))
```

Application: Inverted Index



MPI vs. MapReduce

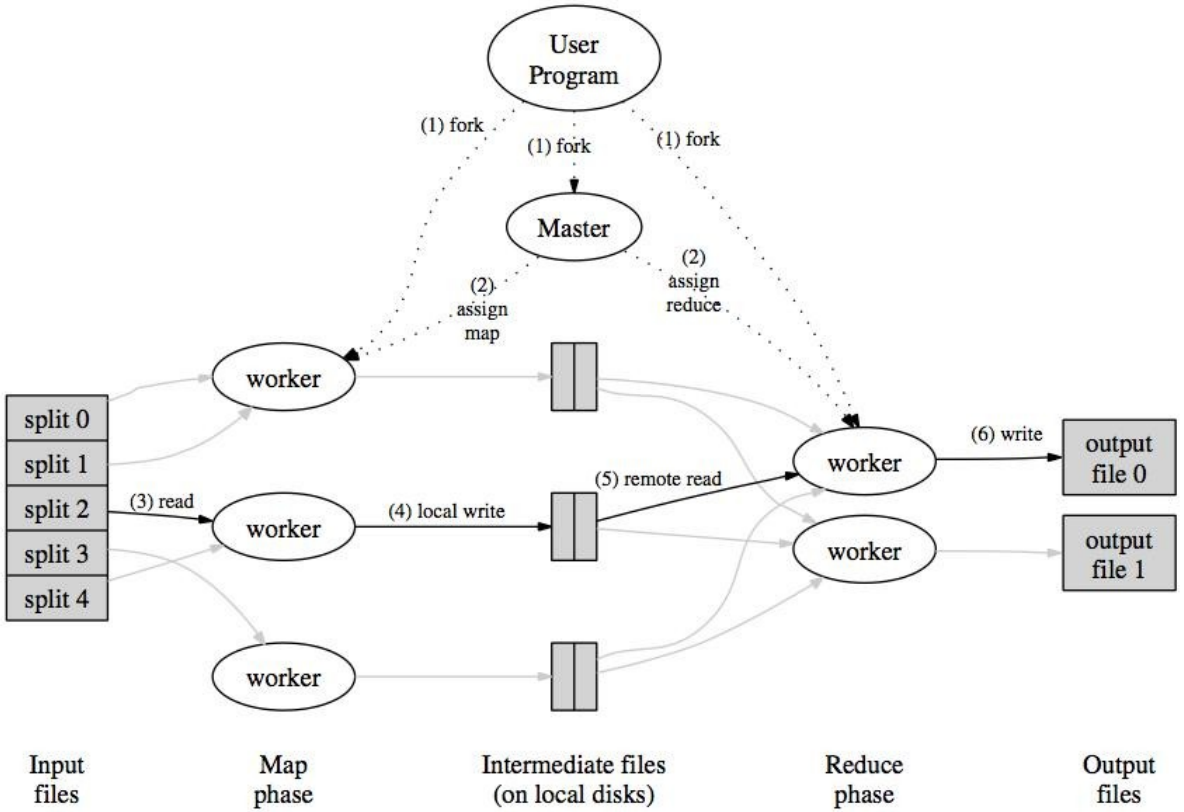
MPI

- Parallel process model
- Fine grain control
- High Performance

MapReduce

- High level data-parallel
- Automate locality, data transfers
- Focus on fault tolerance

Google MapReduce



(J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. *OSDI '04, pages 137–150, 2008)*

Summary

MapReduce data-parallel model

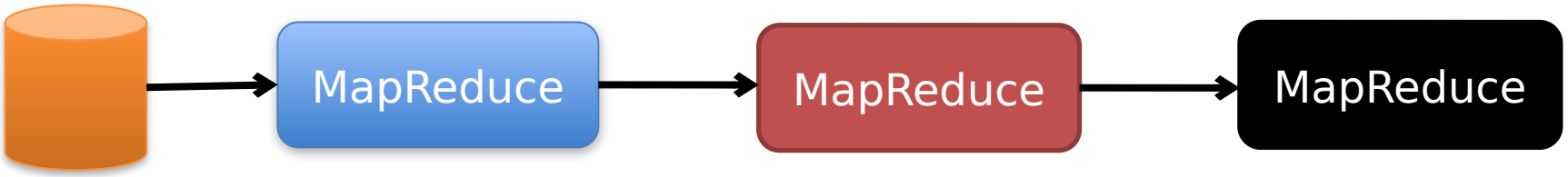
Simplified cluster programming

Automates

- Division of job into tasks
- Locality-aware scheduling
- Load balancing
- Recovery from failures & stragglers

When an Abstraction is Useful…

People want to compose it!



Most real applications require multiple MapReduce steps

- Google indexing pipeline: 21 steps
- Analytics queries (e.g. sessions, top K): 2-5 steps
- Iterative algorithms (e.g. PageRank): tens of steps

Programmability

Multi-step jobs create spaghetti code

- 21 MapReduce steps → 21 mapper and reducer classes

Lots of boilerplate wrapper code per step

API doesn’t provide type safety

Performance

MapReduce only provides one pass of computation

- Must write out data to file system in-between

Expensive for apps that need to *reuse* data

- Multi-step algorithms (e.g. PageRank)
- Interactive data mining

Hadoop

- Hadoop Distributed File System (HDFS): This stores files in a Hadoop-native format and parallelizes them across a cluster. It manages the storage of large sets of data across a Hadoop Cluster. Hadoop can handle both structured and unstructured data.
- YARN: YARN is Yet Another Resource Negotiator. It is a schedule that coordinates application runtimes.
- MapReduce: It is the algorithm that actually processes the data in parallel to combine the pieces into the desired result.
- Hadoop Common: It is also known as Hadoop Core and it provides support to all other components it has a set of common libraries and utilities that all other modules depend on

Applications using MapReduce

- Distributed Grep:
  - Input: large set of files
  - Output: lines that match pattern
    - Map – *Emits a line if it matches the supplied pattern*
    - Reduce – *Copies the intermediate data to output*
- Count of URL access frequency
  - Input: Log of accessed URLs, e.g., from proxy server
  - Output: For each URL, % of total accesses for that URL
    - Map – *Process web log and outputs <URL, 1>*
    - Multiple Reducers - *Emits <URL, URL_count>*
    - Chain another MapReduce job after above one
    - Map – *Processes <URL, URL_count>* and outputs *<1, (<URL, URL_count>)>*
    - One Reducer – Does two passes over input. First sums up *URL_count's* to calculate overall_count.
      - *Emits multiple <URL, URL_count/overall_count>*

MapReduce Programming

- Externally: For user
  1) Write a Map program (short), write a Reduce program (short)
  2) Specify number of Maps and Reduces (parallelism level)
  3) Submit job; wait for result
  4) Need to know very little about parallel/distributed programming!
- Internally: For the Paradigm and Scheduler
  1) Parallelize Map (embarrassingly parallel)
  2) Transfer data from Map to Reduce ("Shuffle")
     - All Map output records with same key assigned to same Reduce task
     - use partitioning function, e.g., hash(key)%number of reducers
  3) Parallelize Reduce (embarrassingly parallel)
  4) Implement Storage for Map input, Map output, Reduce input, and Reduce output

     Ensure that no Reduce starts before all Maps are finished: ***barrier*** between Map phase and Reduce phase

     Map input, Reduce output: Distributed File System (HDFS, GFS)

     Map output: stored locally, Reduce input: fetched remotely (from Map machine)

Spark

- Apache Spark Core: It is responsible for functions like scheduling, input and output operations, task dispatching, etc.
- Spark SQL: This is used to gather information about structured data and how the data is processed.
- Spark Streaming: This component enables the processing of live data streams.
- Machine Learning Library: The goal of this component is scalability and to make machine learning more accessible.
- GraphX: This has a set of APIs that are used for facilitating graph analytics tasks.

Hadoop vs. Spark

- Hadoop reads and writes files to HDFS, Spark processes data in *RAM* using a concept known as an RDD, Resilient Distributed Dataset.
- Spark can run either in stand-alone mode, with a Hadoop cluster serving as the data source, or in conjunction with Mesos (open source cluster).
- Spark is structured around Spark Core, the engine that drives the scheduling, optimizations, and RDD abstraction, as well as connects Spark to the correct filesystem (HDFS, S3, RDBMS, or Elasticsearch). There are several libraries that operate on top of Spark Core, including Spark SQL, MLLib for machine learning, GraphX for graph problems, and streaming which allows for the input of continually streaming log data.