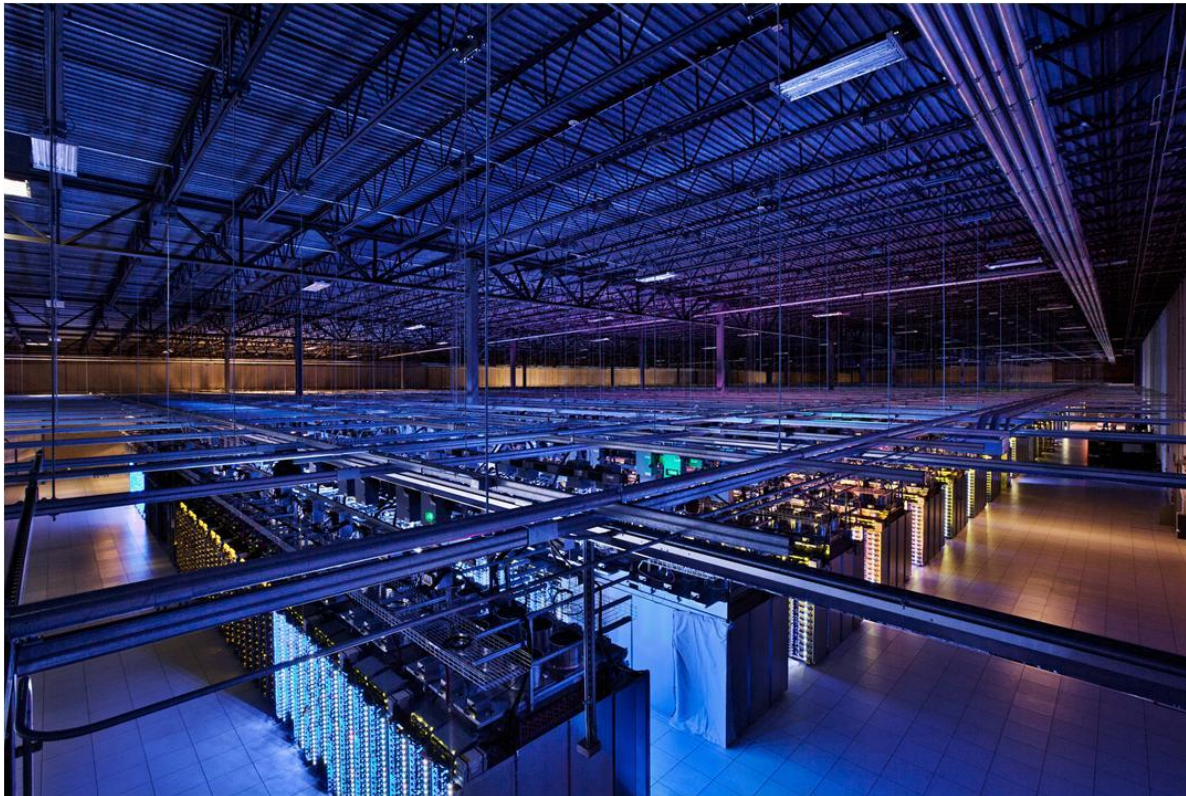


# Large-Scale Data Engineering

Introduction to cloud computing

+

Hadoop,  
HDFS &  
MapReduce

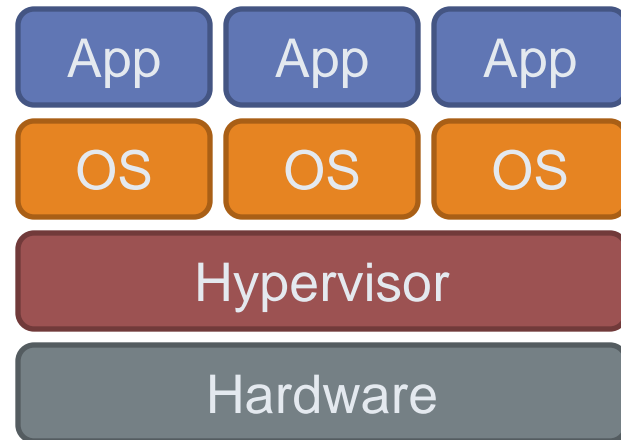
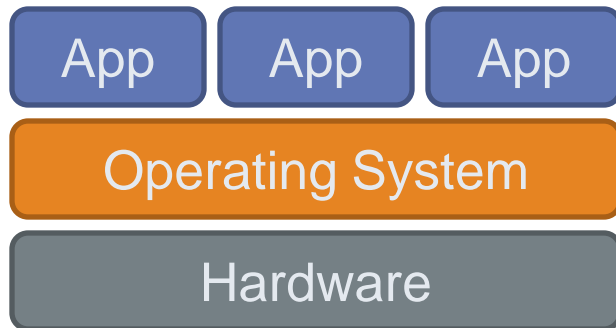


# COMPUTING AS A SERVICE

# Utility computing

- What?
  - Computing resources as a metered service (“pay as you go”)
  - Ability to dynamically provision virtual machines
- Why?
  - Cost: capital vs. operating expenses
  - Scalability: “infinite” capacity
  - Elasticity: scale up or down on demand
- Does it make sense?
  - Benefits to cloud users
  - Business case for cloud providers

# Enabling technology: virtualisation



# Everything as a service

- Utility computing = Infrastructure as a Service (IaaS)
  - Why buy machines when you can rent cycles?
  - Examples: Amazon's EC2, Rackspace
- Platform as a Service (PaaS)
  - Give me nice API and take care of the maintenance, upgrades
  - Example: Google App Engine
- Software as a Service (SaaS)
  - Just run it for me!
  - Example: Gmail, Salesforce

# Several Historical Trends

- Shared Utility Computing
  - 1960s – MULTICS – Concept of a Shared Computing Utility
  - 1970s – IBM Mainframes – rent by the CPU-hour. (Fast/slow switch.)
- Data Center Co-location
  - 1990s-2000s – Rent machines for months/years, keep them close to the network access point and pay a flat rate. Avoid running your own building with utilities!
- Pay as You Go
  - Early 2000s - Submit jobs to a remote service provider where they run on the raw hardware. Sun Cloud (\$1/CPU-hour, Solaris +SGE) IBM Deep Capacity Computing on Demand (50 cents/hour)
- Virtualization
  - 1960s – OS-VM, VM-360 – Used to split mainframes into logical partitions.
  - 1998 – VMWare – First practical implementation on X86, but at significant performance hit.
  - 2003 – Xen paravirtualization provides much perf, but kernel must assist.
  - Late 2000s – Intel and AMD add hardware support for virtualization.

# So, you want to build a cloud

- Slightly more complicated than hooking up a bunch of machines with an ethernet cable
  - Physical vs. virtual (or logical) resource management
  - Interface?
- A host of issues to be addressed
  - Connectivity, concurrency, replication, fault tolerance, file access, node access, capabilities, services, ...
- We'll tackle as many problems as we can
  - The problems are nothing new
  - Solutions have existed for a long time
  - However, it's the first time we have the challenge of applying them all in a single massively accessible infrastructure

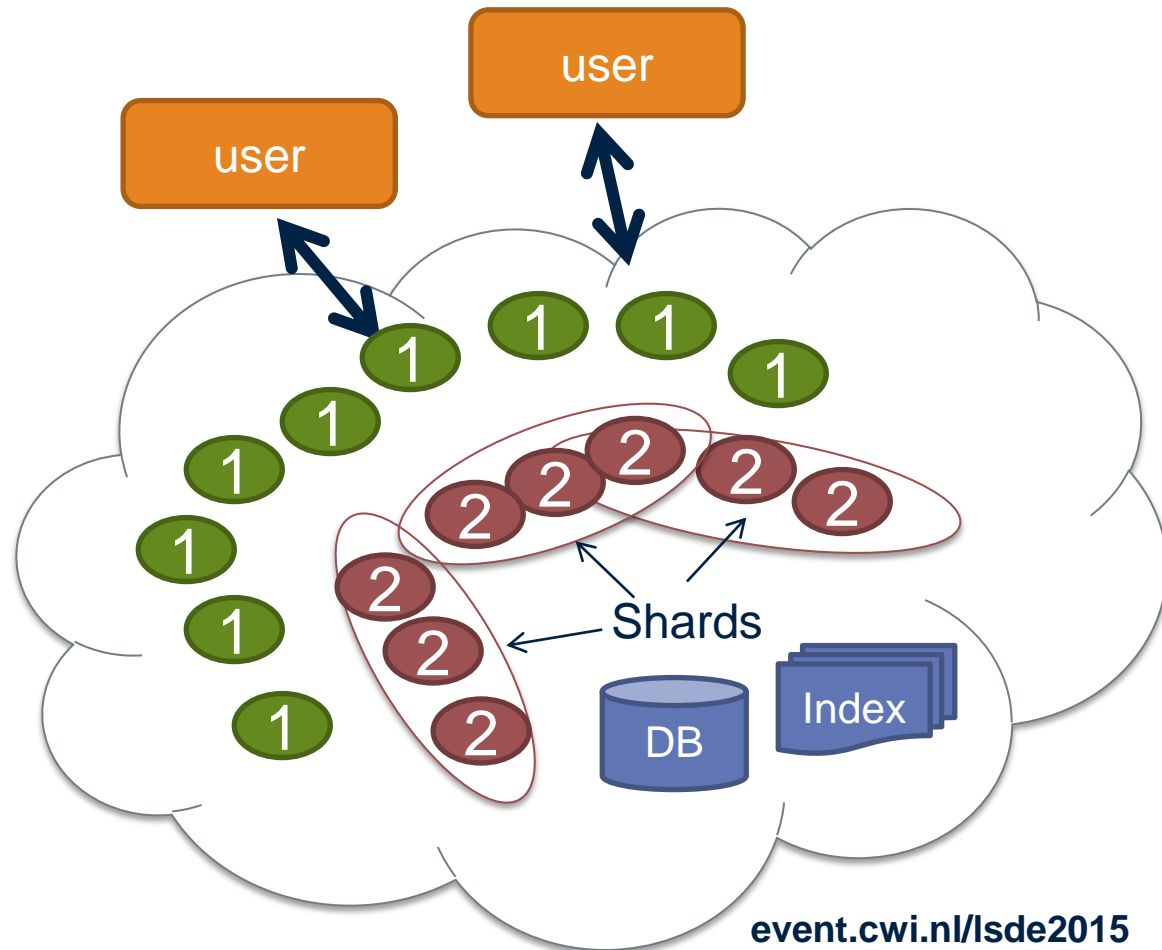
# How are clouds structured?

- Clients talk to clouds using web browsers or the web services standards
  - But this only gets us to the outer “skin” of the cloud data center, not the interior
  - Consider Amazon: it can host entire company web sites (like Target.com or Netflix.com), data (S3), servers (EC2) and even user-provided virtual machines!



# Big picture overview

- Client requests are handled in the first tier by
  - PHP or ASP pages
  - Associated logic
- These lightweight services are fast and very nimble
- Much use of caching: the second tier



# Many styles of system

- Near the edge of the cloud focus is on vast numbers of clients and rapid response
- Inside we find high volume services that operate in a pipelined manner, asynchronously
- Deep inside the cloud we see a world of virtual computer clusters that are
  - Scheduled to share resources
  - Run applications like MapReduce (Hadoop) are very popular
  - Perform the heavy lifting

# In the outer tiers replication is key

- We need to replicate
  - Processing
    - Each client has what seems to be a private, dedicated server (for a little while)
  - Data
    - As much as possible!
    - Server has copies of the data it needs to respond to client requests without any delay at all
  - Control information
    - The entire system is managed in an agreed-upon way by a decentralised cloud management infrastructure

# First-tier parallelism

- Parallelism is vital to speeding up first-tier services
- Key question
  - Request has reached some service instance X
  - Will it be faster
    - For X to just compute the response?
    - Or for X to subdivide the work by asking subservices to do parts of the job?
- Glimpse of an answer
  - Werner Vogels, CTO at Amazon, commented in one talk that many Amazon pages have content from 50 or more parallel subservices that run, in real-time, on the request!

# Read vs. write

- Parallelisation works fine, so long as we are reading
- If we break a large read request into multiple read requests for sub-components to be run in parallel, how long do we need to wait?
  - Answer: as long as the slowest read
- How about breaking a large write request?
  - Duh... we still wait till the slowest write finishes
- But what if these are not sub-components, but alternative copies of the same resource?
  - Also known as replicas
  - We wait the same time, but when do we make the individual writes visible?

Replication solves one problem but introduces another

# More on updating replicas in parallel

- Several issues now arise
  - Are all the replicas applying updates in the same order?
    - Might not matter unless the same data item is being changed
    - But then clearly we do need some agreement on order
  - What if the leader replies to the end user but then crashes and it turns out that the updates were lost in the network?
    - Data centre networks are surprisingly lossy at times
    - Also, bursts of updates can queue up
- Such issues result in inconsistency

# Eric Brewer's CAP theorem

- In a famous 2000 keynote talk at ACM PODC, Eric Brewer proposed that
  - “*You can have just two from Consistency, Availability and Partition Tolerance*”
- He argues that data centres need very fast response, hence availability is paramount
- And they should be responsive even if a transient fault makes it hard to reach some service
- So they should use cached data to respond faster even if the cached entry cannot be validated and might be stale!
- Conclusion: weaken consistency for faster response
- We will revisit this as we go along

# Is inconsistency a bad thing?

- How much consistency is really needed in the first tier of the cloud?
  - Think about YouTube videos. Would consistency be an issue here?
  - What about the Amazon “number of units available” counters. Will people notice if those are a bit off?
    - Probably not unless you are buying the last unit
    - End even then, you might be inclined to say “oh, bad luck”



# **CASE STUDY: AMAZON WEB SERVICES**

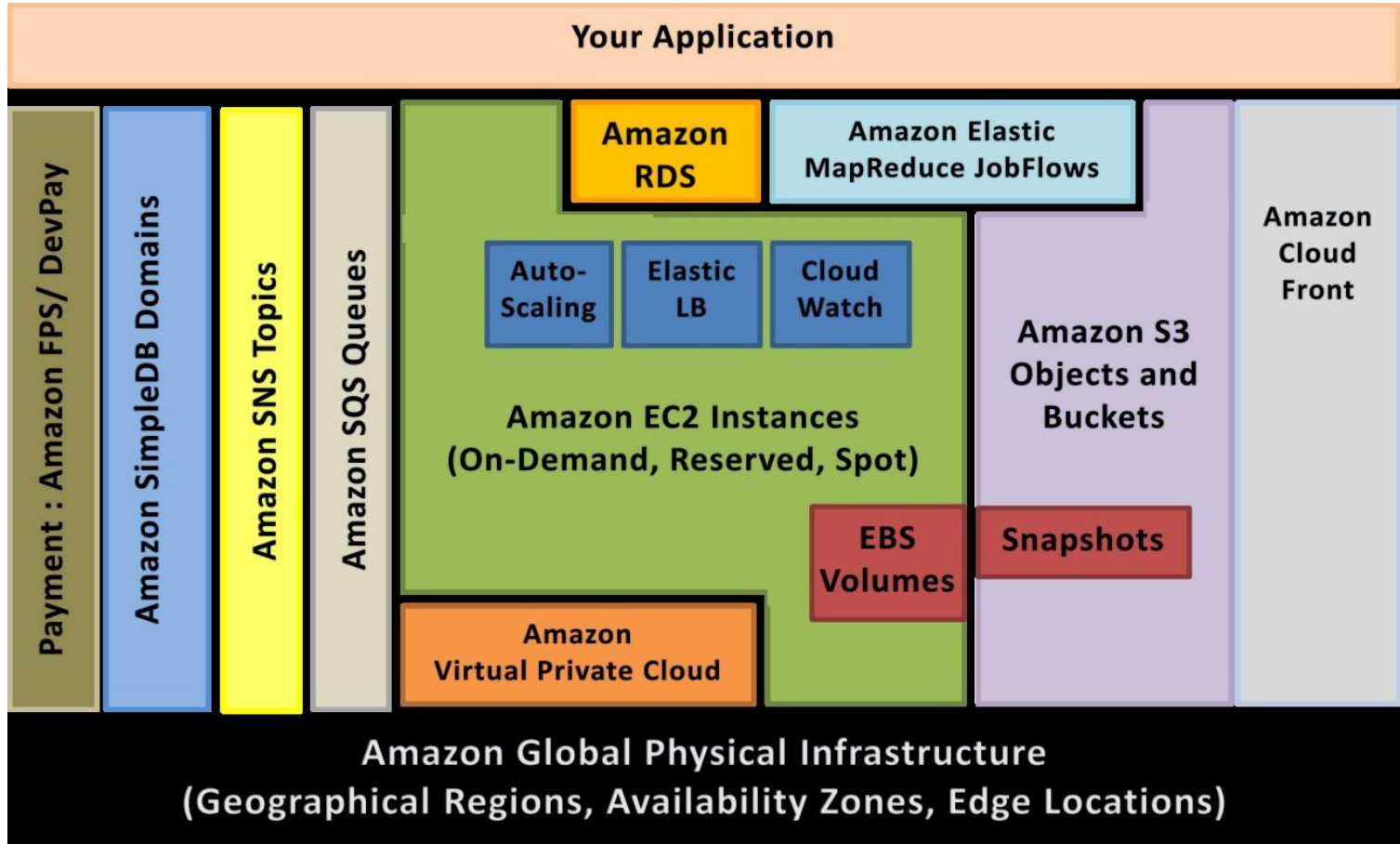
# Amazon AWS

- Grew out of Amazon's need to rapidly provision and configure machines of standard configurations for its own business.
- Early 2000s – Both private and shared data centers began using virtualization to perform “server consolidation”
- 2003 – Internal memo by Chris Pinkham describing an “infrastructure service for the world.”
- 2006 – S3 first deployed in the spring, EC2 in the fall
- 2008 – Elastic Block Store available.
- 2009 – Relational Database Service
- 2012 – DynamoDB

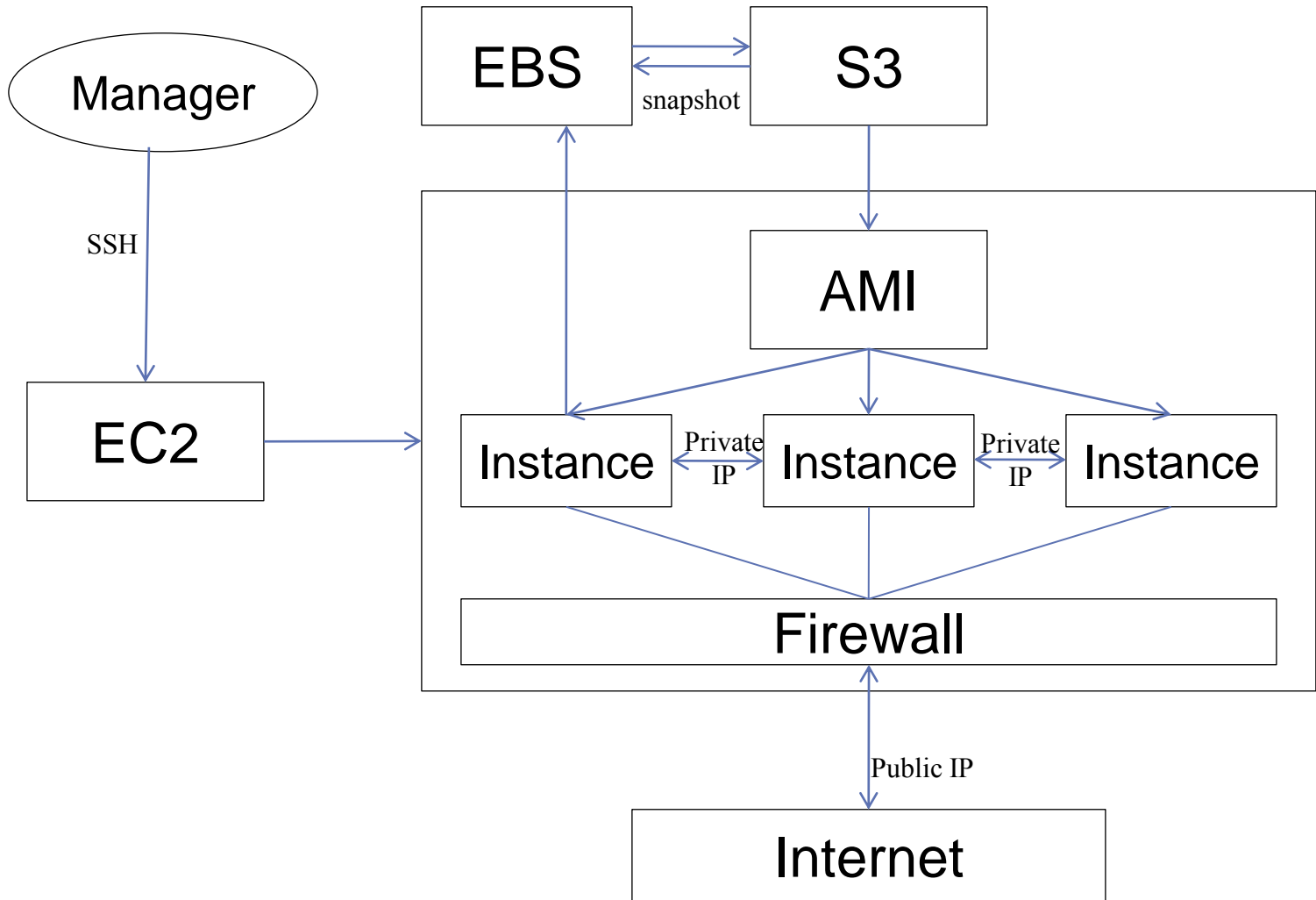
# Terminology

- Instance = One running virtual machine.
- Instance Type = hardware configuration: cores, memory, disk.
- Instance Store Volume = Temporary disk associated with instance.
- Image (AMI) = Stored bits which can be turned into instances.
- Key Pair = Credentials used to access VM from command line.
- Region = Geographic location, price, laws, network locality.
- Availability Zone = Subdivision of region the is fault-independent.

# Amazon AWS



# EC2 Architecture



Model	vCPU	CPU Credits / hour	Mem (GiB)	Storage (GB)
t2.micro	1	6	1	EBS Only
t2.small	1	12	2	EBS Only
t2.medium	2	24	4	EBS Only

Model	vCPU	Mem (GiB)	SSD Storage (GB)
c3.large	2	3.75	2 x 16
c3.xlarge	4	7.5	2 x 40
c3.2xlarge	8	15	2 x 80
c3.4xlarge	16	30	2 x 160
c3.8xlarge	32	60	2 x 320

### Use Cases

High performance front-end fleets, web-servers, on-demand batch processing, distributed analytics, high performance science and engineering applications, ad serving, batch processing, MMO gaming, video encoding, and distributed analytics.

Model	vCPU	Mem (GiB)	SSD Storage (GB)
m3.medium	1	3.75	1 x 4
m3.large	2	7.5	1 x 32
m3.xlarge	4	15	2 x 40
m3.2xlarge	8	30	2 x 80

Model	vCPU	Mem (GiB)	SSD Storage (GB)
r3.large	2	15.25	1 x 32
r3.xlarge	4	30.5	1 x 80
r3.2xlarge	8	61	1 x 160
r3.4xlarge	16	122	1 x 320
r3.8xlarge	32	244	2 x 320

### Use Cases

We recommend memory-optimized instances for high performance databases, distributed memory caches, in-memory analytics, genome assembly and analysis, larger deployments of SAP, Microsoft SharePoint, and other enterprise applications.

# EC2 Pricing Model

- Free Usage Tier
- On-Demand Instances
  - Start and stop instances whenever you like, costs are rounded up to the nearest hour. (Worst price)
- Reserved Instances
  - Pay up front for one/three years in advance. (Best price)
  - Unused instances can be sold on a secondary market.
- Spot Instances
  - Specify the price you are willing to pay, and instances get started and stopped without any warning as the market changes. (Kind of like Condor!)

# Free Usage Tier

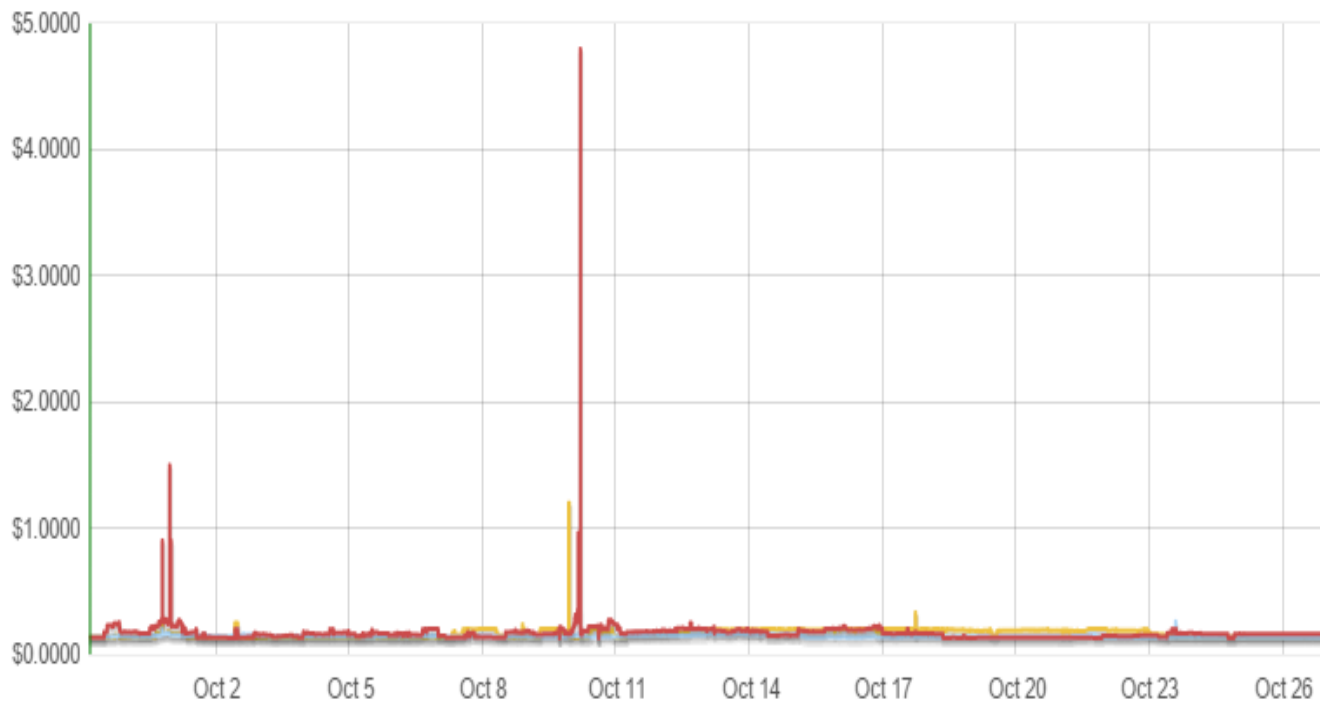
- 750 hours of EC2 running Linux, RHEL, or SLES t2.micro instance usage
- 750 hours of EC2 running Microsoft Windows Server t2.micro instance usage
- 750 hours of Elastic Load Balancing plus 15 GB data processing
- 30 GB of Amazon Elastic Block Storage in any combination of General Purpose (SSD) or Magnetic, plus 2 million I/Os (with Magnetic) and 1 GB of snapshot storage
- 15 GB of bandwidth out aggregated across all AWS services
- 1 GB of Regional Data Transfer





## Spot Instance Pricing History



Product : **Linux/UNIX** ▾ Instance type: **c3.4xlarge** ▾ Date range : **1 month** ▾ Availability zone: **All zones** ▾



Availability zone	Price
 us-west-2a	\$0.1389
 us-west-2b	\$0.1390
 us-west-2c	\$0.1322

Date September 28, 2014 9:08:53 AM UTC-4

# Simple Storage Service (S3)

- A **bucket** is a container for objects and describes location, logging, accounting, and access control. A bucket can hold any number of **objects**, which are files of up to 5TB. A bucket has a name that must be **globally unique**.
- Fundamental operations corresponding to HTTP actions:
  - `http://bucket.s3.amazonaws.com/object`
  - POST a new object or update an existing object.
  - GET an existing object from a bucket.
  - DELETE an object from the bucket
  - LIST keys present in a bucket, with a filter.
- A bucket has a **flat directory structure** (despite the appearance given by the interactive web interface.)

# S3 Weak Consistency Model

Direct quote from the Amazon developer API:

“Updates to a single key are **atomic**....”

“Amazon S3 achieves high availability by replicating data across multiple servers within Amazon's data centers. If a PUT request is successful, your data is safely stored. However, information about the changes must replicate across Amazon S3, which can take some time, and so you might observe the following behaviors:

- A process writes a new object to Amazon S3 and immediately attempts to read it. Until the change is fully propagated, Amazon S3 might report "key does not exist."
- A process writes a new object to Amazon S3 and immediately lists keys within its bucket. Until the change is fully propagated, the object might not appear in the list.
- A process replaces an existing object and immediately attempts to read it. Until the change is fully propagated, Amazon S3 might return the prior data.
- A process deletes an existing object and immediately attempts to read it. Until the deletion is fully propagated, Amazon S3 might return the deleted data.”

## Storage Pricing

Region: 

	Standard Storage	Reduced Redundancy Storage	Glacier Storage
First 1 TB / month	\$0.0300 per GB	\$0.0240 per GB	\$0.0100 per GB
Next 49 TB / month	\$0.0295 per GB	\$0.0236 per GB	\$0.0100 per GB
Next 450 TB / month	\$0.0290 per GB	\$0.0232 per GB	\$0.0100 per GB
Next 500 TB / month	\$0.0285 per GB	\$0.0228 per GB	\$0.0100 per GB
Next 4000 TB / month	\$0.0280 per GB	\$0.0224 per GB	\$0.0100 per GB
Over 5000 TB / month	\$0.0275 per GB	\$0.0220 per GB	\$0.0100 per GB

# Request Pricing

Region: US Standard

	Pricing
PUT, COPY, POST, or LIST Requests	\$0.005 per 1,000 requests
Glacier Archive and Restore Requests	\$0.05 per 1,000 requests
Delete Requests	Free †
GET and all other Requests	\$0.004 per 10,000 requests
Glacier Data Restores	Free ‡

† No charge for delete requests of Standard or RRS objects. For objects that are archived to Glacier, there is a pro-rated charge of \$0.03 per gigabyte for objects deleted prior to 90 days. [Learn more.](#)

‡ Glacier is designed with the expectation that restores are infrequent and unusual, and data will be stored for extended periods of time. You can restore up to 5% of your average monthly Glacier storage (pro-rated daily) for free each month. If you choose to restore more than this amount of data in a month, you are charged a restore fee starting at \$0.01 per gigabyte. [Learn more.](#)

# Data Transfer Pricing

The pricing below is based on data transferred "in" to and "out" of Amazon S3.

Region:

## Pricing

### Data Transfer IN To Amazon S3

All data transfer in	\$0.000 per GB
----------------------	----------------

### Data Transfer OUT From Amazon S3 To

Amazon EC2 in the Northern Virginia Region	\$0.000 per GB
--	----------------

Another AWS Region or Amazon CloudFront	\$0.020 per GB
---	----------------

### Data Transfer OUT From Amazon S3 To Internet

First 1 GB / month	\$0.000 per GB
--------------------	----------------

Up to 10 TB / month	\$0.120 per GB
---------------------	----------------

Next 40 TB / month	\$0.090 per GB
--------------------	----------------

Next 100 TB / month	\$0.070 per GB
---------------------	----------------

# Elastic Block Store

- An EBS volume is a **virtual disk** of a fixed size with a block read/write interface. It can be **mounted** as a filesystem on a running EC2 instance where it can be **updated incrementally**. Unlike an instance store, an EBS volume is **persistent**.
- (Compare to an S3 object, which is essentially a file that must be accessed in its entirety.)
- Fundamental operations:
  - CREATE a new volume (1GB-1TB)
  - COPY a volume from an existing EBS volume or S3 object.
  - MOUNT on one instance at a time.
  - SNAPSHOT current state to an S3 object.

## Amazon EBS Volume Types

Volume Type	EBS General Purpose (SSD)	EBS Provisioned IOPS (SSD)	EBS Magnetic
Use Cases	Boot volumes Small to Med DBs Dev and Test	I/O intensive Relational DBs NoSQL DBs	Infrequent Data Access
Storage Media	SSD-backed	SSD-backed	Magnetic disk-backed
Max Volume Size	1TB	1TB	1TB
Max IOPS/volume	3,000 (burst)	4,000	40 - 200
Max throughput/volume	128MBps	128MBps	40 - 90MBps
Max IOPS/instance	48,000	48,000	48,000
Max throughput/instance	800MBps	800MBps	800MBps
API Name	gp2	io1	standard
Price*	\$.10/GB - Month	\$.125/GB - Month \$.065/provisioned IOPS	\$.05/GB - Month \$.05/million I/O



EBS is approx. 3x more expensive by volume  
and 10x more expensive by IOPS than S3.

# Use Glacier for Cold Data

- Glacier is structured like S3: a **vault** is a container for an arbitrary number of **archives**. Policies, accounting, and access control are associated with vaults, while an archive is a single object.
- However:
  - All operations are asynchronous and notified via SNS.
  - Vault listings are updated once per day.
  - Archive downloads may take up to four hours.
  - Only 5% of total data can be accessed in a given month.
- Pricing:
  - Storage: \$0.01 per GB-month
  - Operations: \$0.05 per 1000 requests
  - Data Transfer: Like S3, free within AWS.
- S3 Policies can be set up to automatically move data into Glacier.

# **SOME MORE TIPS FROM ASSIGNMENT 1**


# Assignment 1: Querying a Social Graph



# LDBC Data generator

- Synthetic dataset available in different scale factors
  - SF100 ← for quick testing
  - SF3000 ← the real deal
- Very complex graph
  - Power laws (e.g. degree)
  - Huge Connected Component
  - Small diameter
  - Data correlations
    - Chinese have more Chinese names*
  - Structure correlations
    - Chinese have more Chinese friends*

← → ↻ | ldbcouncil.org/industry/members



**LDBC**  The graph & RDF benchmark reference



BENCHMARKS » INDUSTRY » PUBLIC » DEVELOPER » EVENTS TALKS PUBLICATIONS BLOG



Information about how the LDBC organization works



HOME » INDUSTRY » MEMBERS

Companies:

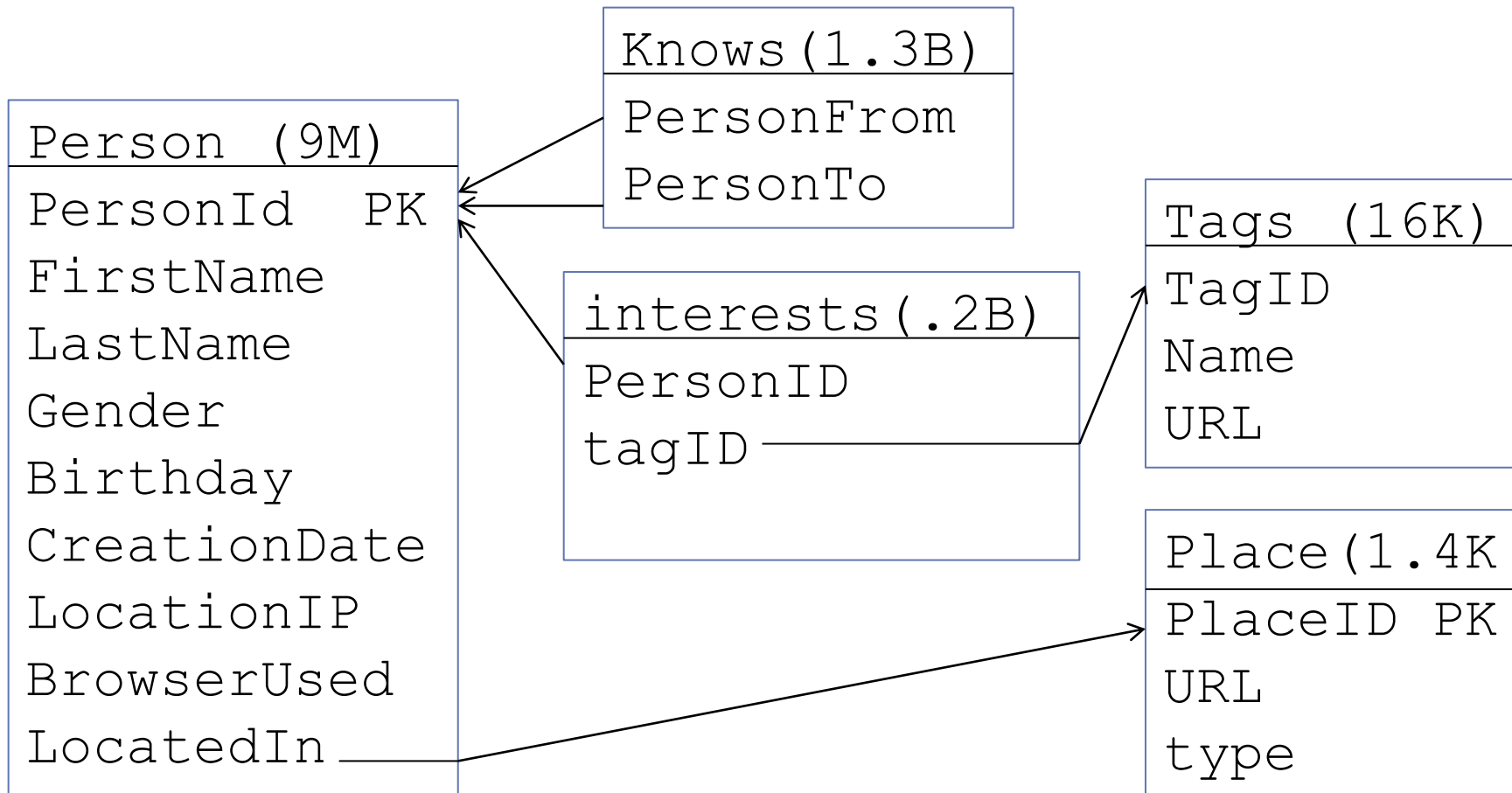



# CSV file schema

- See: [http://wikistats.ins.cwi.nl/lsde-data/practical\\_1](http://wikistats.ins.cwi.nl/lsde-data/practical_1)
- Counts for sf3000 (total 37GB)



# The Query

- The marketers of a social network have been data mining the musical preferences of their users. They have built statistical models which predict given an interest in say artists A2 and A3, that the person would also like A1 (i.e. rules of the form: A2 and A3  $\rightarrow$  A1). Now, they are commercially exploiting this knowledge by selling targeted ads to the management of artists who, in turn, want to sell concert tickets to the public but in the process also want to expand their artists' fanbase.
- The ad is a suggestion for people who already are interested in A1 to buy concert tickets of artist A1 (with a discount!) as a birthday present for a friend ("who we know will love it" - the social network says) who lives in the same city, who is not yet interested in A1 yet, but is interested in other artists A2, A3 and A4 that the data mining model predicts to be correlated with A1.

# The Query

*For all persons  $P$  :*

- *who have their birthday on or in between  $D1..D2$*
- *who do not like  $A1$  yet*

*we give a score of*

- *1 for liking any of the artists  $A2$ ,  $A3$  and  $A4$  and*
- *0 if not*

*the final score, the sum, hence is a number between 0 and 3.*

*Further, we look for friends  $F$ :*

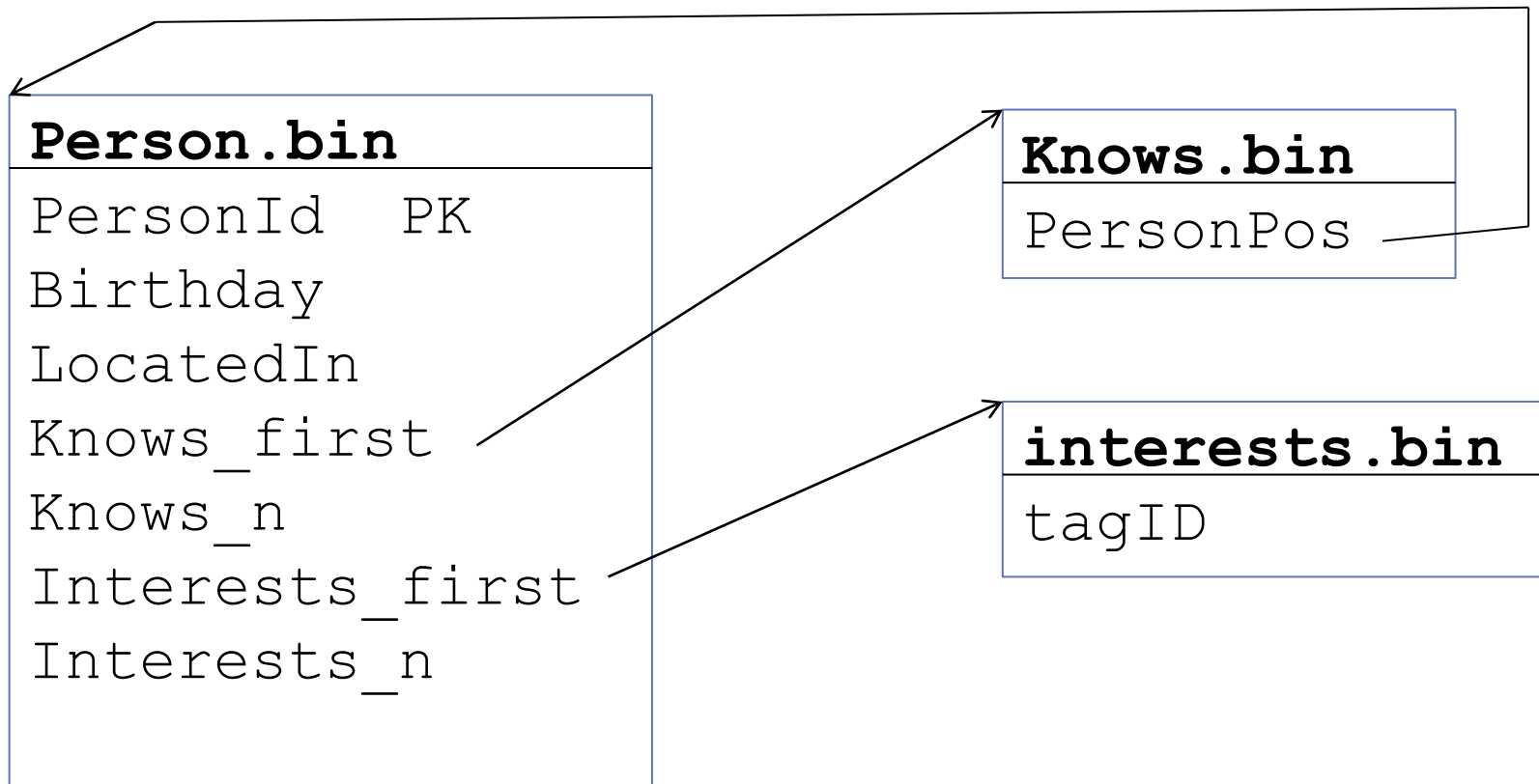
- *Where  $P$  and  $F$  who know each other mutually*
- *Where  $P$  and  $F$  live in the same city and*
- *Where  $F$  already likes  $A1$*

*The answer of the query is a table (score,  $P$ ,  $F$ ) with only scores  $> 0$*



# Binary files

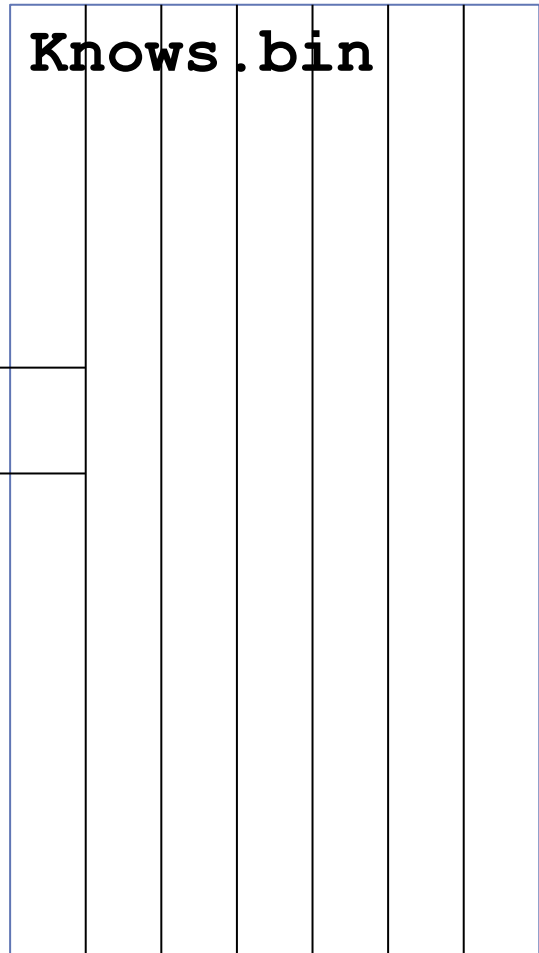
- Created by “loader” program in example github repo
- Total size: 6GB



# What it looks like

- Created by “loader” program in example github repo
- Total size: 6GB

*4bytes*  
*\* 1.3B*



*48bytes*  
*\* 8.9M*

knows\_first

knows\_n

*2bytes*  
*\* 204M*

# The Naïve Implementation

*The “cruncher” program*

*Go through the persons  $P$  sequentially*

- *counting how many of the artists  $A_2, A_3, A_4$  are liked as the score for those with  $\text{score} > 0$ :*

- *visit all persons  $F$  known to  $P$ .*

*For each  $F$ :*

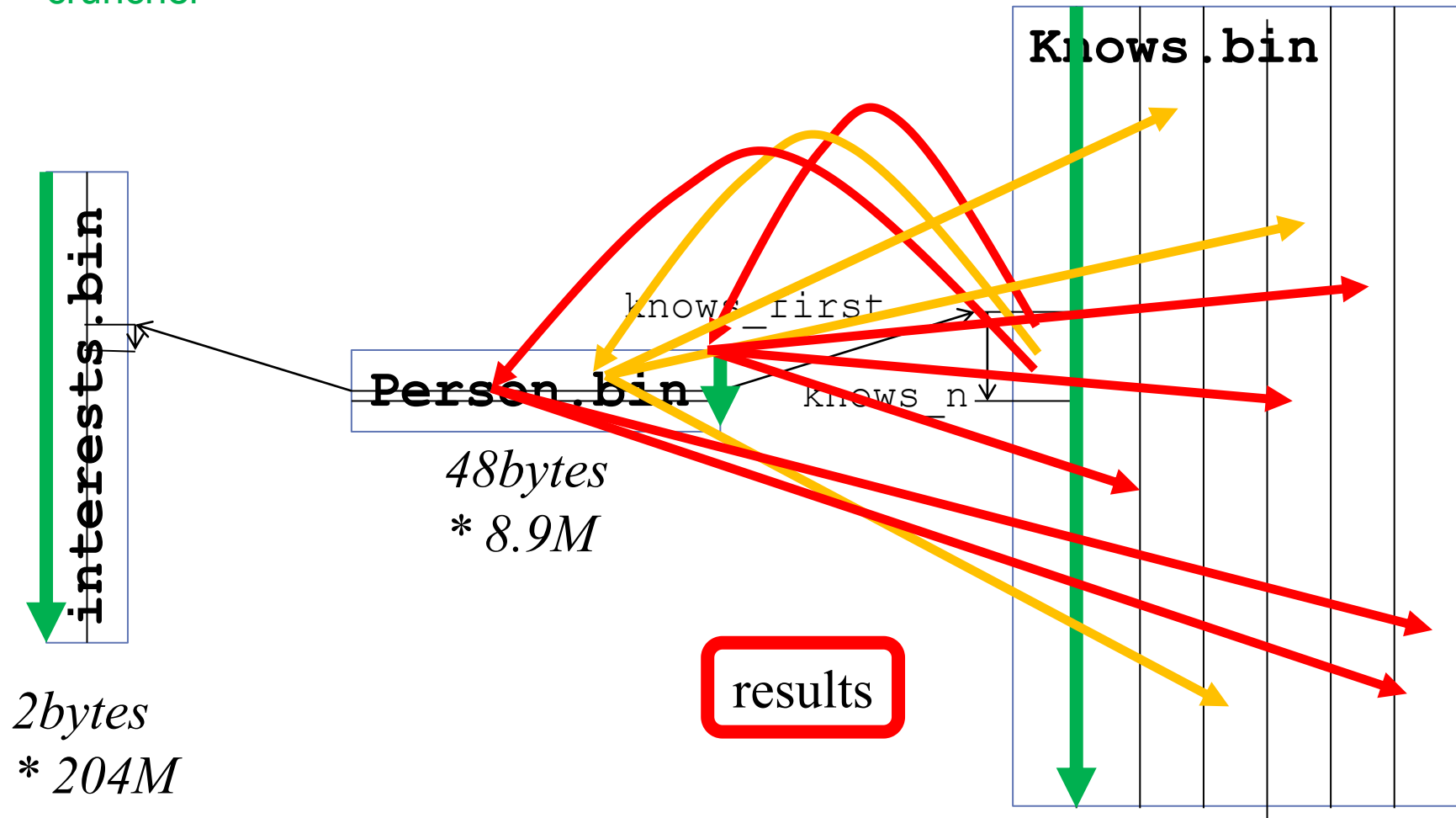
- *checks on equal location*
- *check whether  $F$  already likes  $A_1$*
- *check whether  $F$  also knows  $P$*

*if all this succeeds  $(\text{score}, P, F)$  is added to a result table.*

# Naïve Query Implementation

- “cruncher”

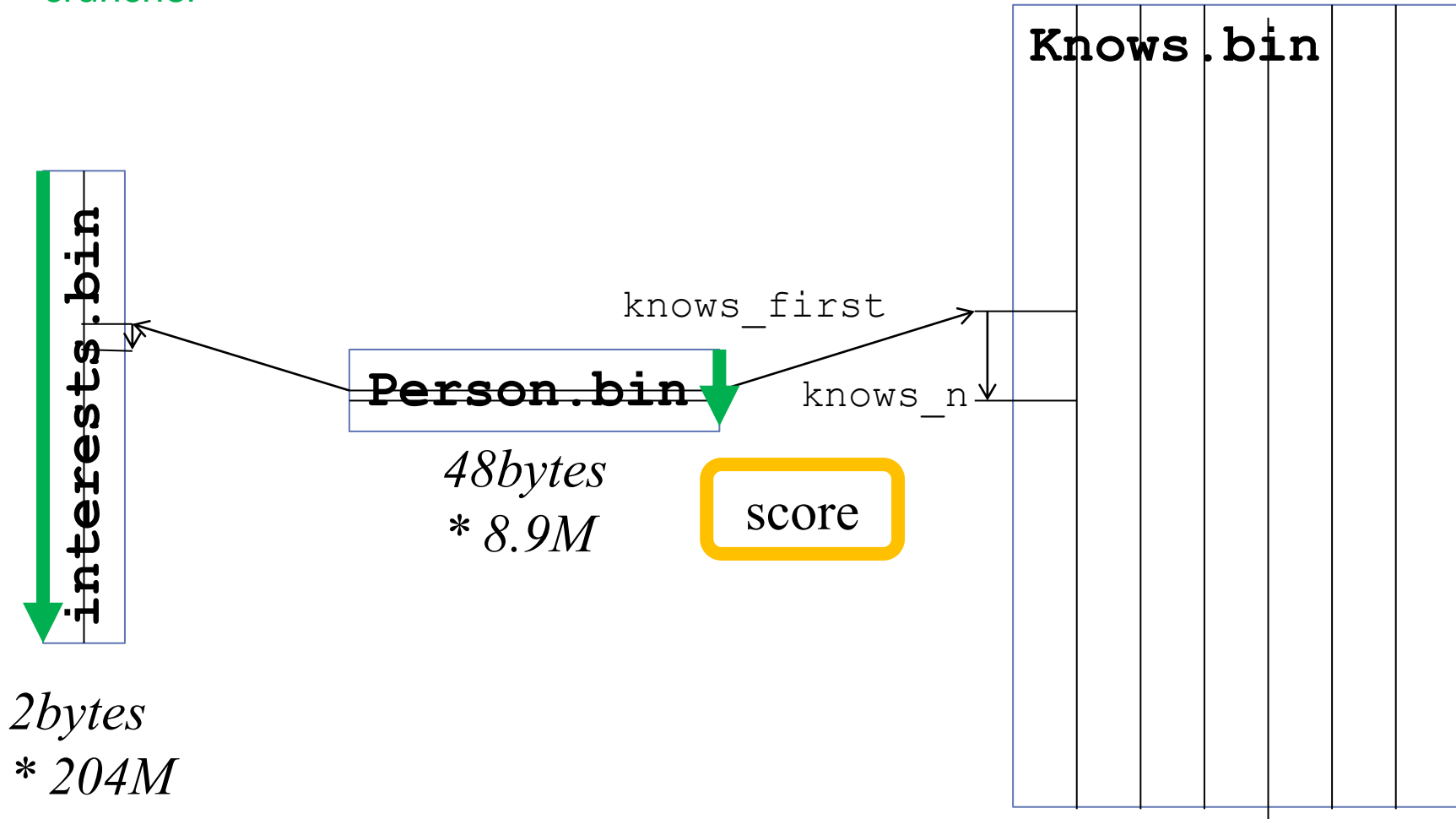
4bytes  
\* 1.3B



# Sequential Query Implementation

4bytes  
\* 1.3B

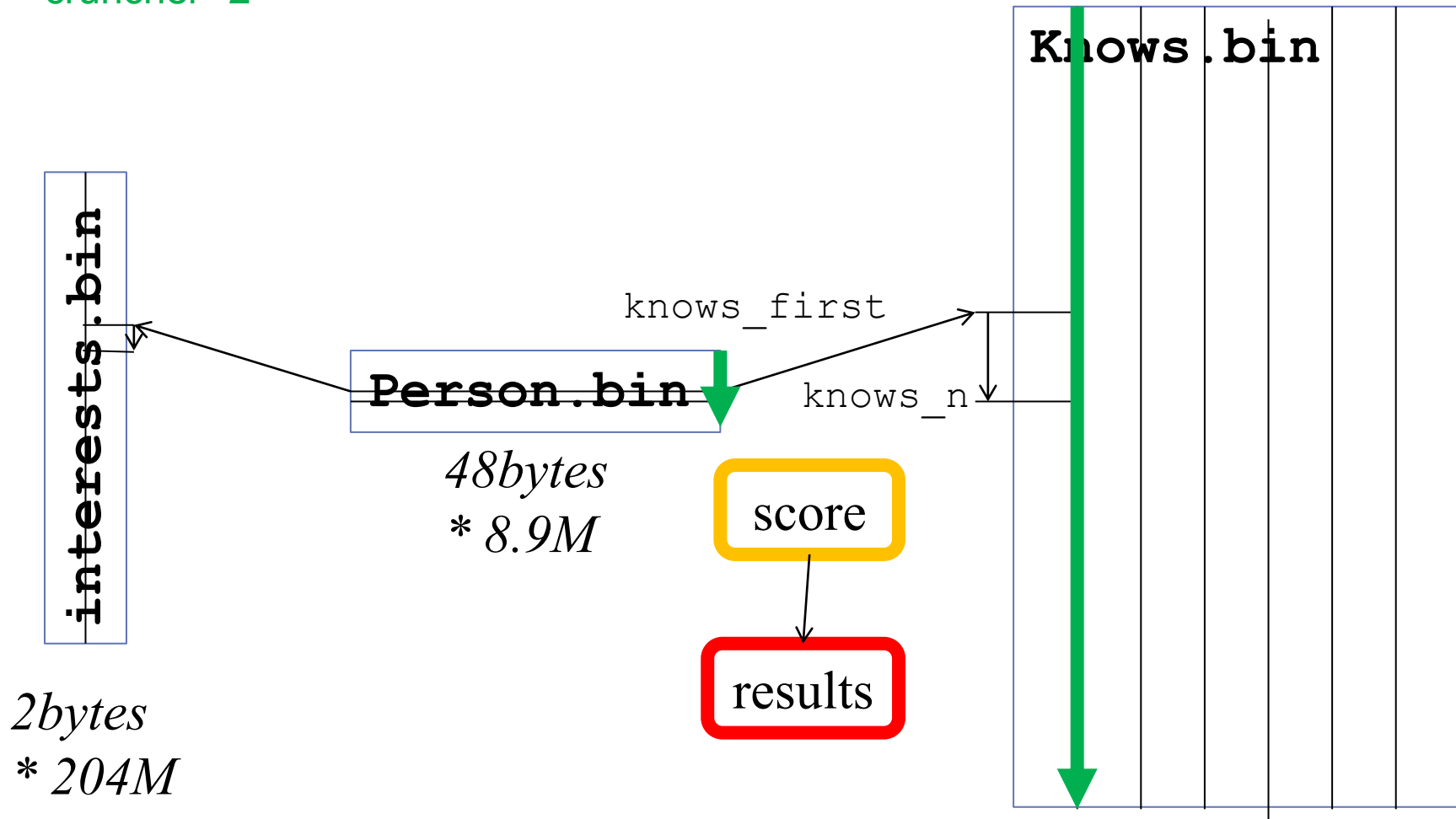
- “cruncher”



# Sequential Query Implementation

4bytes  
\* 1.3B

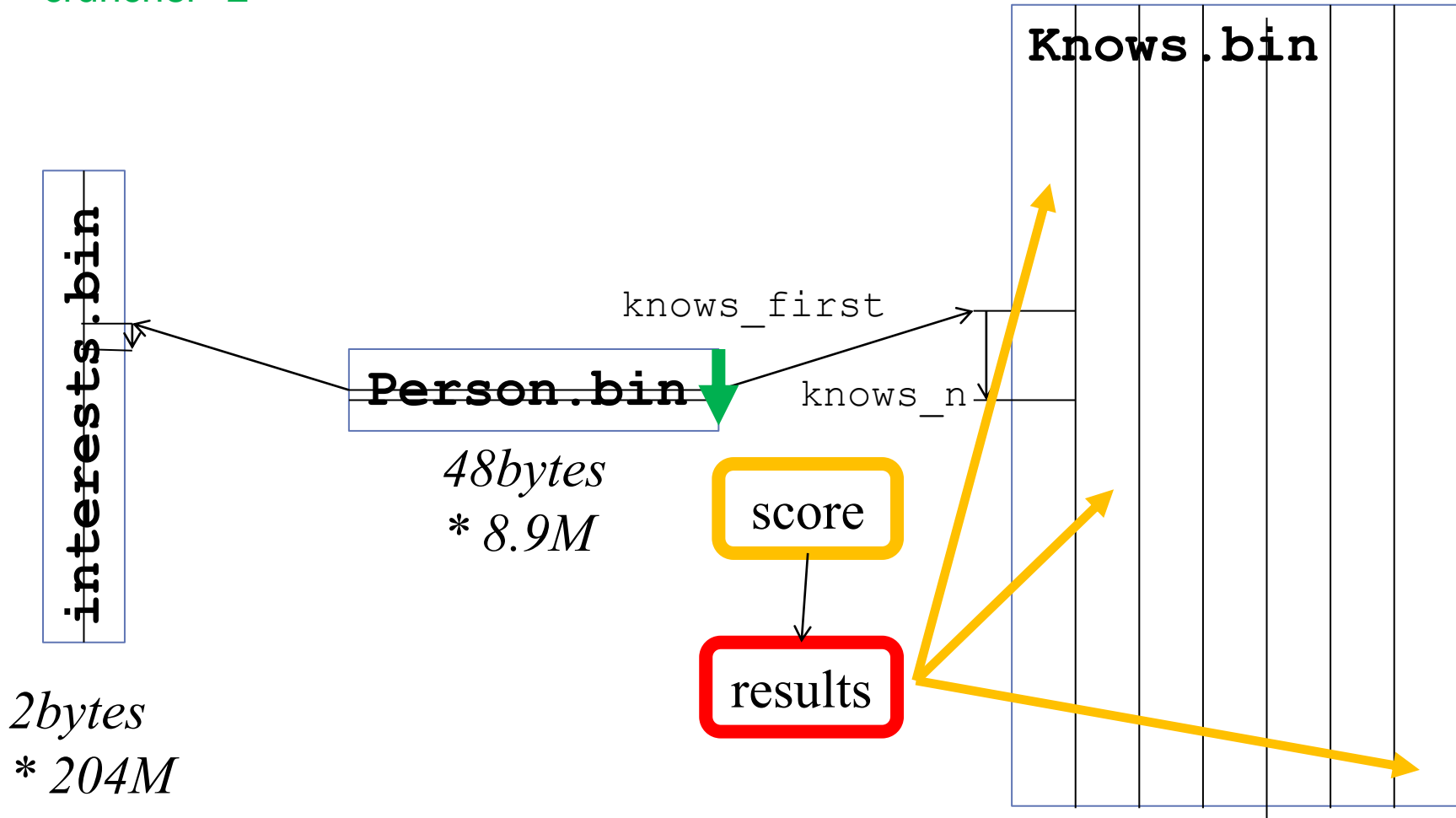
- “cruncher”-2



# Sequential Query Implementation

4bytes  
\* 1.3B

- “cruncher”-2



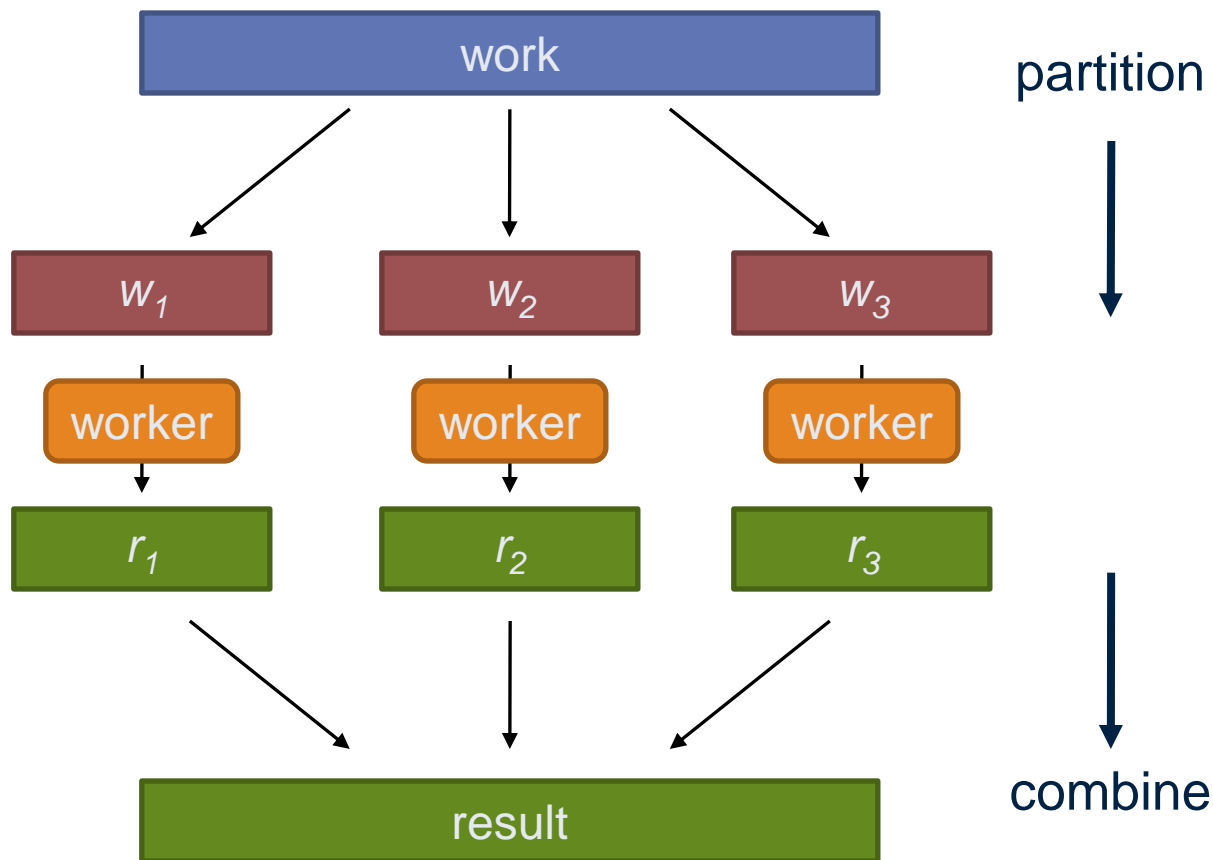
# Improving Bad Access Patterns

- Minimize Random Memory Access
  - Apply filters first. Less accesses is better.
- Denormalize the Schema
  - Remove joins/lookups, add looked up stuff to the table (but.. makes it bigger)
- Trade Random Access For Sequential Access
  - perform a 100K random key lookups in a large table
    - ➔ put 100K keys in a hash table, then
      - scan table and lookup keys in hash table
- Try to make the randomly accessed region smaller
  - Remove unused data from the structure
  - Apply data compression
  - Cluster or Partition the data (improve locality) ...hard for social graphs
- If the random lookups often fail to find a result
  - Use a Bloom Filter



# SETTING UP WORKFLOWS

# Key premise: divide and conquer



# Parallelisation challenges

- How do we assign work units to workers?
- What if we have more work units than workers?
- What if workers need to share partial results?
- How do we aggregate partial results?
- How do we know all the workers have finished?
- What if workers die?

What's the common theme of all of these problems?

# Common theme?

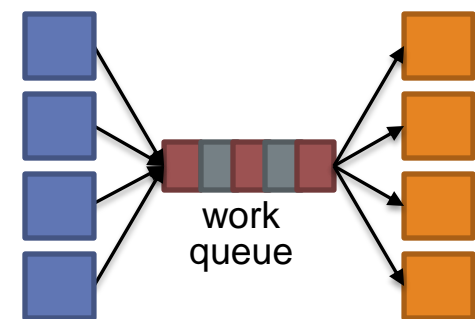
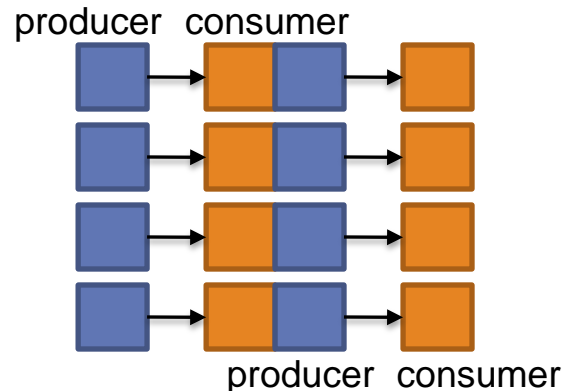
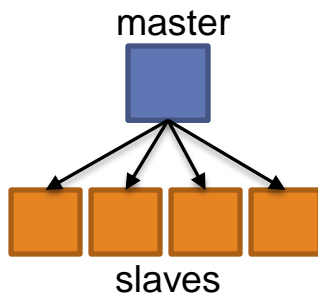
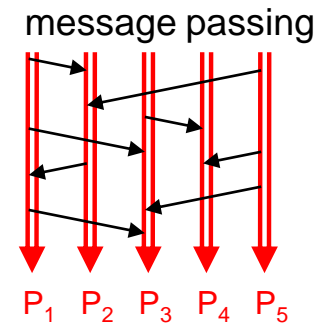
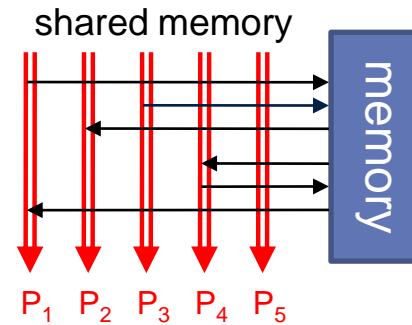
- Parallelization problems arise from:
  - Communication between workers (e.g., to exchange state)
  - Access to shared resources (e.g., data)
- Thus, we need a synchronization mechanism

# Managing multiple workers

- Difficult because
  - We don't know the order in which workers run
  - We don't know when workers interrupt each other
  - We don't know when workers need to communicate partial results
  - We don't know the order in which workers access shared data
- Thus, we need:
  - Semaphores (lock, unlock)
  - Conditional variables (wait, notify, broadcast)
  - Barriers
- Still, lots of problems:
  - Deadlock, livelock, race conditions...
  - Dining philosophers, sleeping barbers, cigarette smokers...
- Moral of the story: be careful!

# Current tools

- Programming models
  - Shared memory (pthreads)
  - Message passing (MPI)
- Design patterns
  - Master-slaves
  - Producer-consumer flows
  - Shared work queues



# Where the rubber meets the road

- Concurrency is difficult to reason about
- Concurrency is even more difficult to reason about
  - At the scale of datacenters and across datacenters
  - In the presence of failures
  - In terms of multiple interacting services
- Not to mention debugging...
- The reality:
  - Lots of one-off solutions, custom code
  - Write you own dedicated library, then program with it
  - Burden on the programmer to explicitly manage everything
  - The MapReduce runtime alleviates this

# What's the point?

- It's all about the right level of abstraction
  - Moving beyond the von Neumann architecture
  - We need better programming models
- Hide system-level details from the developers
  - No more race conditions, lock contention, etc.
- Separating the what from how
  - Developer specifies the computation that needs to be performed
  - Execution framework (aka runtime) handles actual execution

The data centre *is* the computer!





**Here's your new toy**

# MAPREDUCE AND HDFS

# Big data needs big ideas

- Scale “out”, not “up”
  - Limits of SMP and large shared-memory machines
- Move processing to the data
  - Cluster has limited bandwidth, cannot waste it shipping data around
- Process data sequentially, avoid random access
  - Seeks are expensive, disk throughput is reasonable, memory throughput is even better
- Seamless scalability
  - From the mythical man-month to the tradable machine-hour
- Computation is still big
  - But if efficiently scheduled and executed to solve bigger problems we can throw more hardware at the problem and use the same code
  - Remember, the datacentre is the computer

# Typical Big Data Problem

- Iterate over a large number of records
- Extract something of interest from each
- Shuffle and sort intermediate results
- Aggregate intermediate results
- Generate final output

Map

Reduce

Key idea: provide a functional abstraction for these two operations

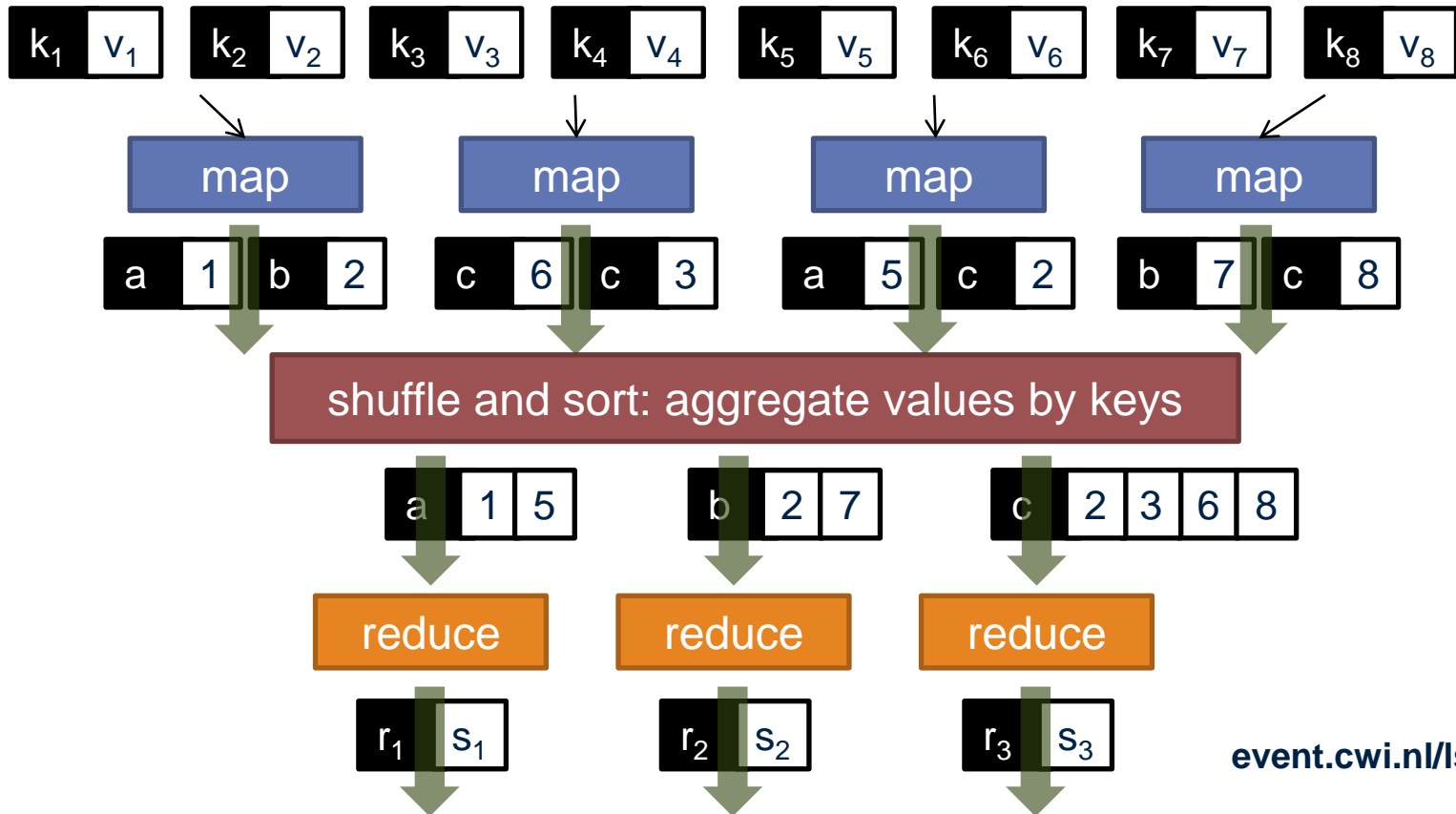
# MapReduce

- Programmers specify two functions:

**map**  $(k_1, v_1) \rightarrow [k_2, v_2]$

**reduce**  $(k_2, [v_2]) \rightarrow [k_3, v_3]$

- All values with the same key are sent to the same reducer



# MapReduce runtime

- Orchestration of the distributed computation
- Handles scheduling
  - Assigns workers to map and reduce tasks
- Handles data distribution
  - Moves processes to data
- Handles synchronization
  - Gathers, sorts, and shuffles intermediate data
- Handles errors and faults
  - Detects worker failures and restarts
- Everything happens on top of a distributed file system (more information later)

# MapReduce

- Programmers specify two functions:

**map**  $(k, v) \rightarrow \langle k', v' \rangle^*$

**reduce**  $(k', v') \rightarrow \langle k', v' \rangle^*$

– All values with the same key are reduced together

- The execution framework handles everything else
- This is the minimal set of information to provide
- Usually, programmers also specify:

**partition**  $(k', \text{number of partitions}) \rightarrow \text{partition for } k'$

– Often a simple hash of the key, e.g.,  $\text{hash}(k') \bmod n$

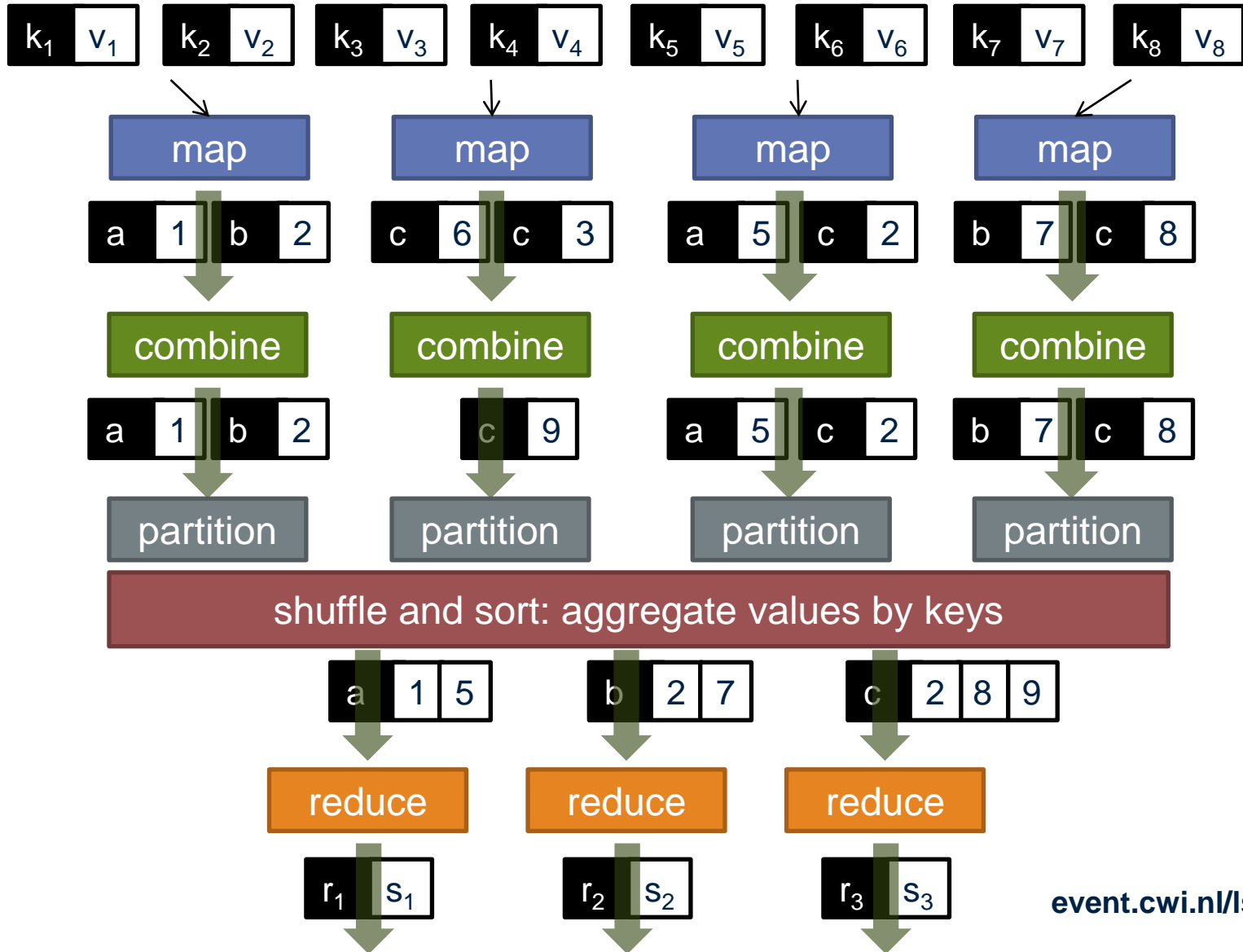
– Divides up key space for parallel reduce operations

**combine**  $(k', v') \rightarrow \langle k', v' \rangle^*$

– Mini-reducers that run in memory after the map phase

– Used as an optimization to reduce network traffic

# Putting it all together





# Two more details

- Barrier between map and reduce phases
  - But we can begin copying intermediate data earlier
- Keys arrive at each reducer in sorted order
  - No enforced ordering *across* reducers

# “Hello World”: Word Count

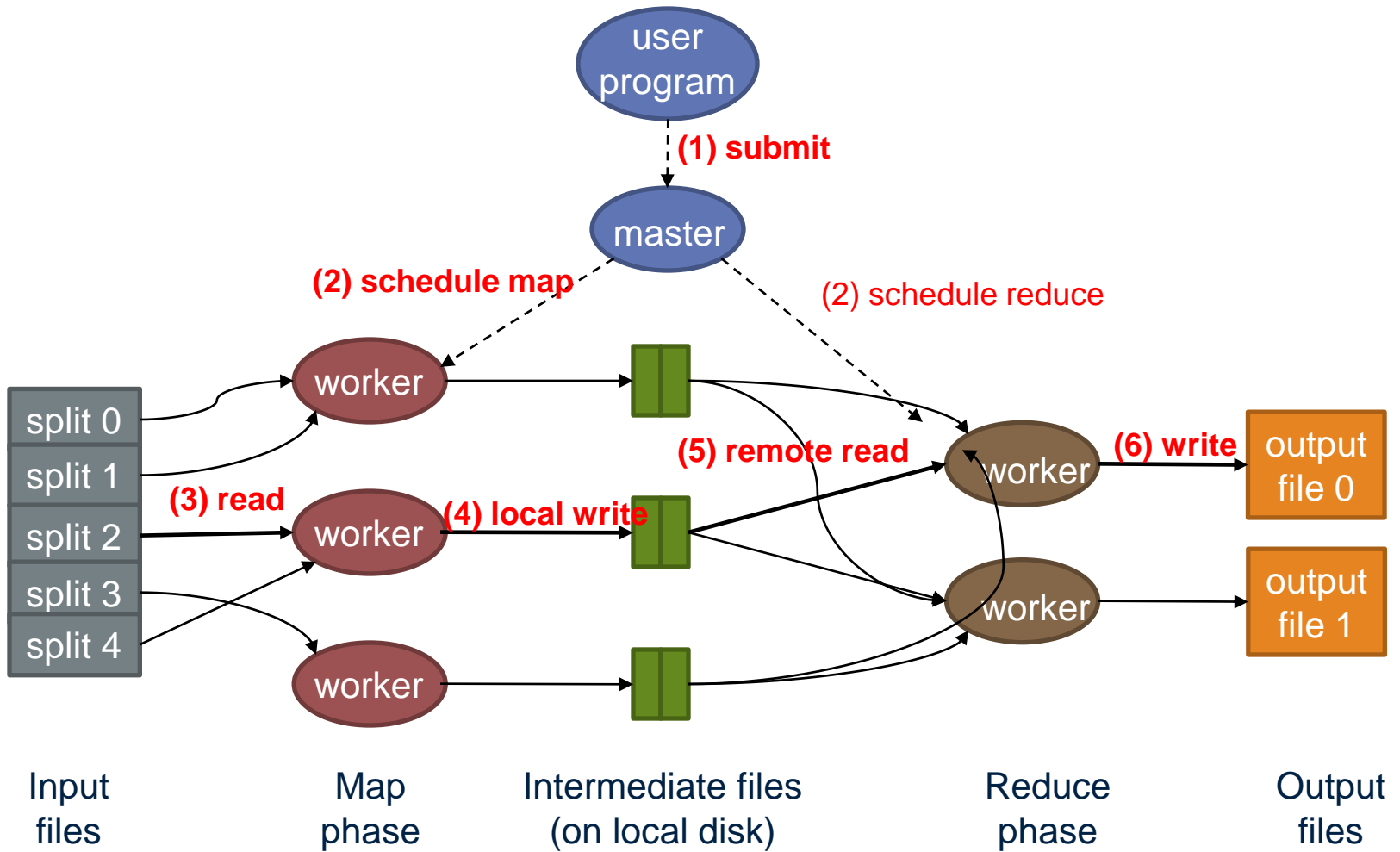
```
Map(String docid, String text):  
  for each word w in text:  
    Emit(w, 1);
```

```
Reduce(String term, Iterator<Int> values):  
  int sum = 0;  
  for each v in values:  
    sum += v;  
  Emit(term, sum);
```

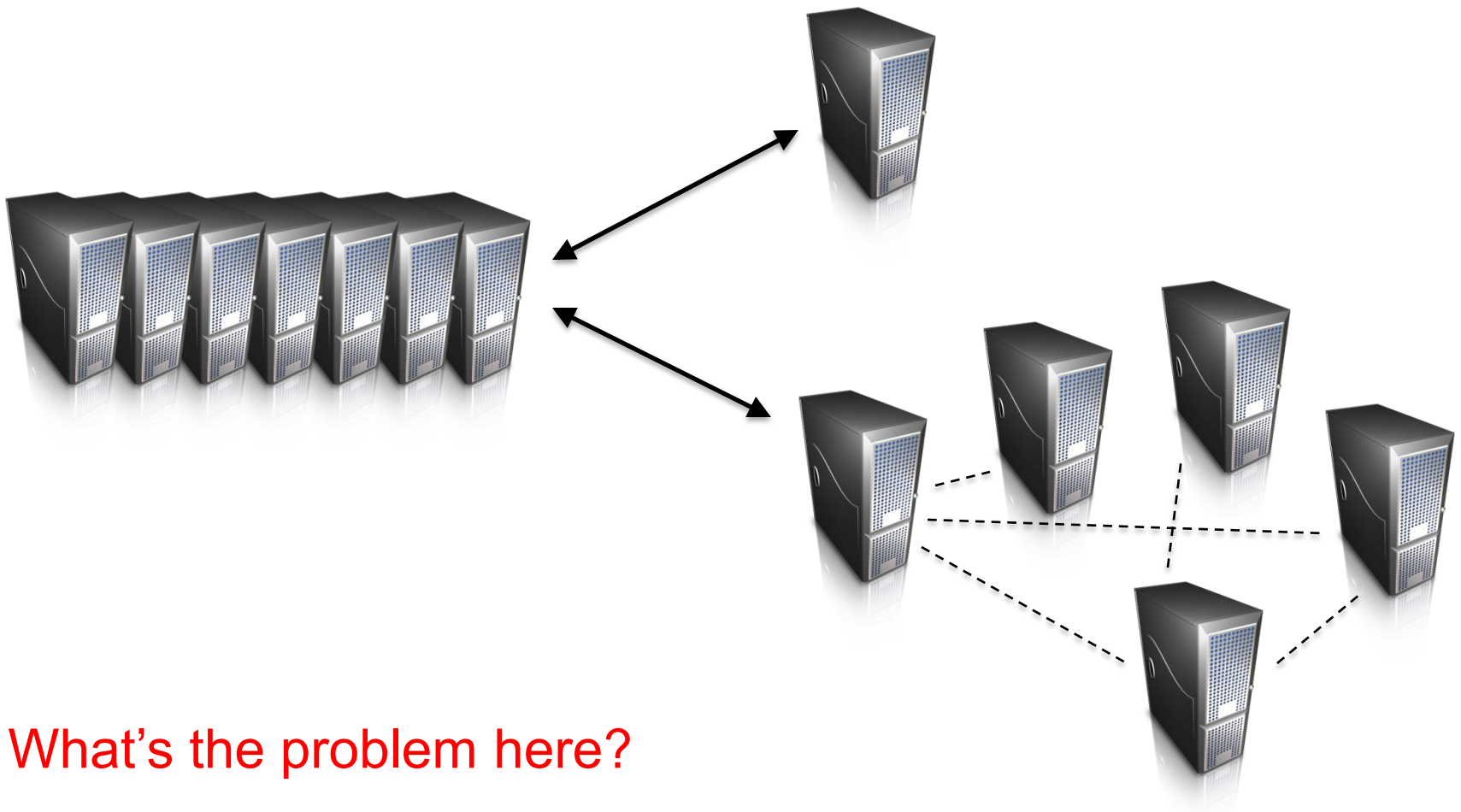
# MapReduce Implementations

- Google has a proprietary implementation in C++
  - Bindings in Java, Python
- Hadoop is an open-source implementation in Java
  - Development led by Yahoo, now an Apache project
  - Used in production at Yahoo, Facebook, Twitter, LinkedIn, Netflix, ...
  - The *de facto* big data processing platform
  - Rapidly expanding software ecosystem
- Lots of custom research implementations
  - For GPUs, cell processors, etc.





# How do we get data to the workers?



What's the problem here?

# Distributed file system

- Do not move data to workers, but move workers to the data!
  - Store data on the local disks of nodes in the cluster
  - Start up the workers on the node that has the data local
- Why?
  - Not enough RAM to hold all the data in memory
  - Disk access is slow, but disk throughput is reasonable
- A distributed file system is the answer
  - GFS (Google File System) for Google's MapReduce
  - HDFS (Hadoop Distributed File System) for Hadoop

# GFS: Assumptions

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

# GFS: Design Decisions

- Files stored as chunks
  - Fixed size (64MB)
- Reliability through replication
  - Each chunk replicated across 3+ chunkservers
- Single master to coordinate access, keep metadata
  - Simple centralized management
- No data caching
  - Little benefit due to large datasets, streaming reads
- Simplify the API
  - Push some of the issues onto the client (e.g., data layout)

**HDFS = GFS clone (same basic ideas)**

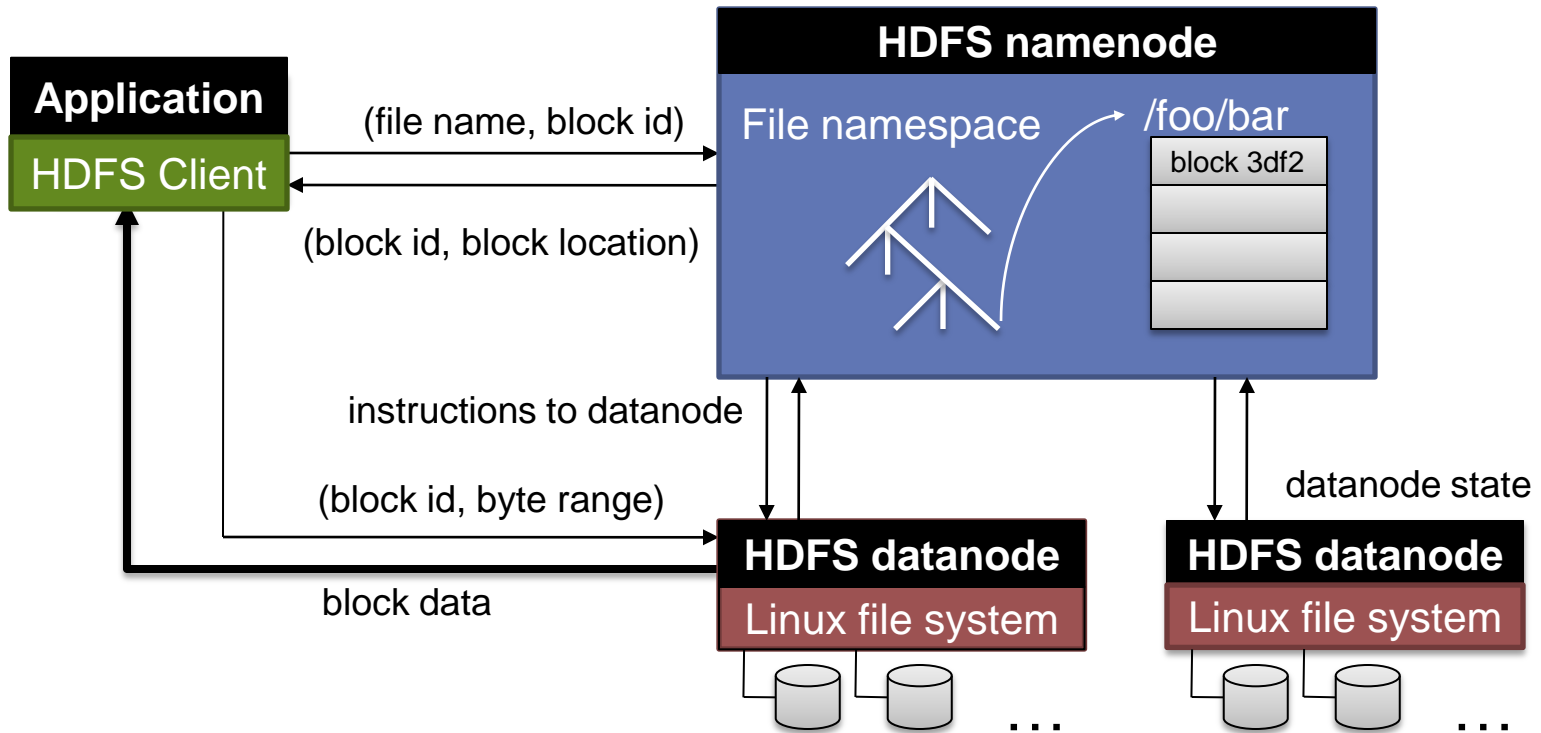


# From GFS to HDFS

- Terminology differences:
  - GFS master = Hadoop namenode
  - GFS chunkservers = Hadoop datanodes
- Differences:
  - Different consistency model for file appends
  - Implementation
  - Performance

For the most part, we'll use Hadoop terminology

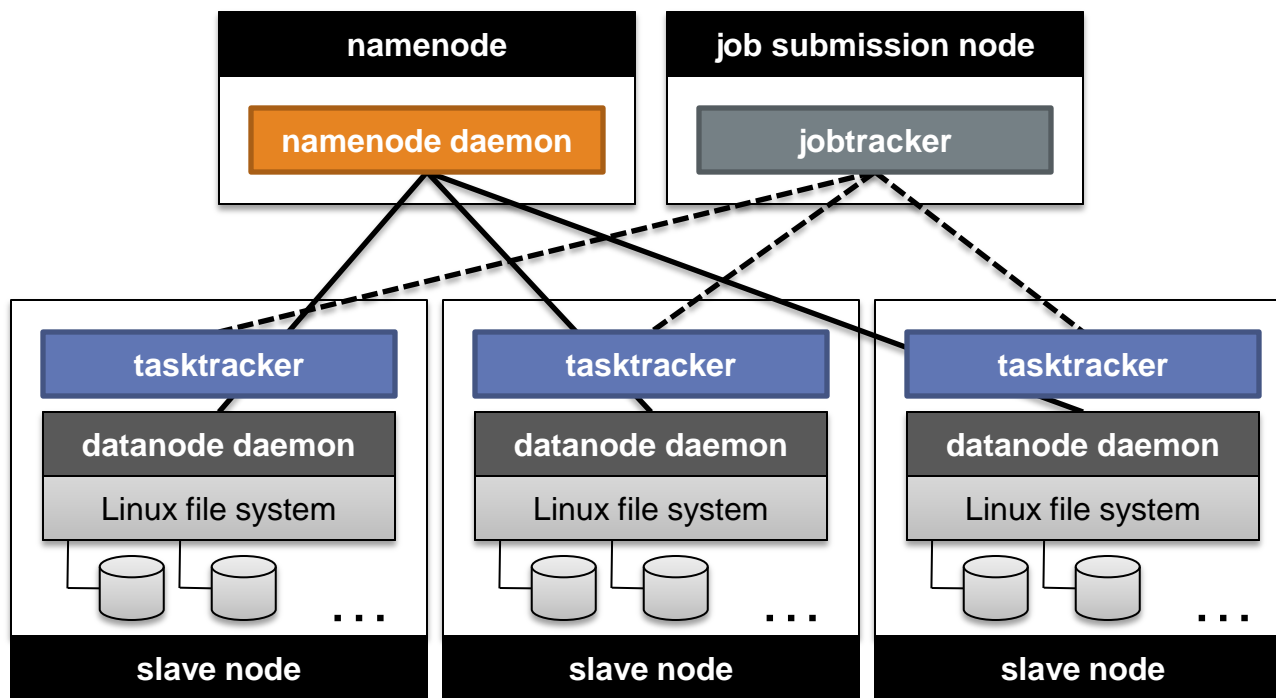
# HDFS architecture



# Namenode responsibilities

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

# Putting everything together



# Summary

- Introduced the notion of utility computing
- Introduced cloud computing and the need for infrastructure
- Presented some of the tools necessary for manipulating Big Data
- We will next turn to the internals of such platforms