

# Distributed systems

## Lecture 7: Replication in distributed systems, CAP, case studies

---

Dr Robert N. M. Watson

# Last time

---

- General issue of **consensus**:
  - How to get processes to agree on something
  - FLP says “impossible” in **asynchronous networks** with at least 1 failure ... but in practice we’re OK!
  - General idea useful for **leadership elections, distributed mutual exclusion**: relies on being able to detect failures
- **Distributed transactions**:
  - Need to commit a set of “sub-transactions” across multiple servers – want **all-or-nothing** semantics
  - Use **atomic commit** protocol like 2PC
- **Replication**:
  - **Performance, load-balancing, and fault tolerance**
  - Introduction to **consistency**

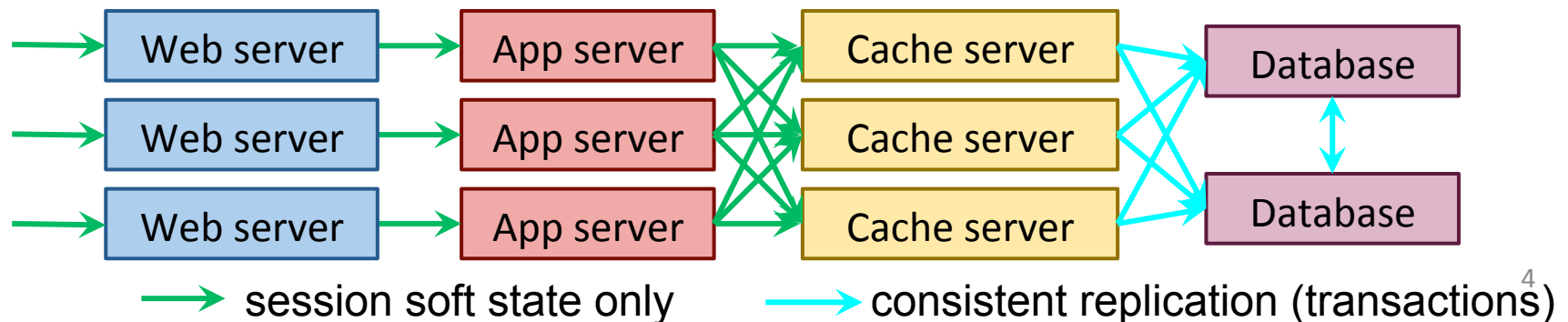
# Replication and Consistency

---

- Gets more challenging if clients can perform updates
- For example, imagine  $x$  has value 3 (in all replicas)
  - **C1** requests **write( $x$ , 5)** from **S4**
  - **C2** requests **read( $x$ )** from **S3**
  - What should occur?
- With **strong consistency**, the distributed system behaves as if there is no replication present:
  - i.e. in above, **C2** should get the value 5
  - requires coordination between all servers
- With **weak consistency**, **C2** may get 3 or 5 (or ...?)
  - Less satisfactory, but much easier to implement

# Replication for fault tolerance

- Replication for services, not just data objects
- Easiest is for a **stateless services**:
  - Simply duplicate functionality over  $k$  machines
  - Clients use any (e.g. closest), fail over to another
- Very few totally stateless services
  - But e.g. many web apps have per-session soft state
  - State generated per-client, lost when client leaves
- For example: multi-tier web farms (Facebook, ...):



# Passive Replication

---

- A solution for stateful services is **primary/backup**:
  - Backup server takes over in case of failure
- Based around **persistent logs** and **system checkpoints**:
  - Periodically (or continuously) checkpoint primary
  - If detect failure, start backup from checkpoint
- A few variants trade-off fail-over time:
  - **Cold-standby**: backup server must start service (software), load checkpoint & parse logs
  - **Warm-standby**: backup server has software running in anticipation – just needs to load primary state
  - **Hot-standby**: backup server mirrors primary work, but output is discarded; on failure, enable output

# Active Replication

---

- **Alternative**: have  $k$  replicas running at **all** times
- Front-end server acts as an **ordering node**:
  - Receives requests from client and forwards them to all replicas using totally ordered multicast
  - Replicas each perform operation and respond to front-end
  - Front-end gathers responses, and replies to client
- Typically require replicas to be “**state machines**”:
  - i.e. act deterministically based on input
  - Idea is that all replicas operate ‘in lock step’
- Active replication is expensive (in terms of resources)...
  - ... and not really worth it in the common case.
  - However valuable if consider **Byzantine failures**

# Achieving Strong Consistency

---

- Need to ensure any update propagates to all replicas before allow any subsequent reads
- One solution:
  - When  $S_i$  receives request to update  $x$ , first locks  $x$  at all other replicas
  - Once successful,  $S_i$  makes update, and propagates to all other replicas, who acknowledge
  - Finally,  $S_i$  instructs all replicas to unlock
  - Once  $C_j$  has an ACK for its write,  $C_x$  (including  $C_j$ ) will see update
- Need to handle failure (of replica, or network)
  - Add step to tentatively apply update, and only actually apply (“commit”) update if all replicas agree
- We’ve reinvented distributed transactions & 2PC!

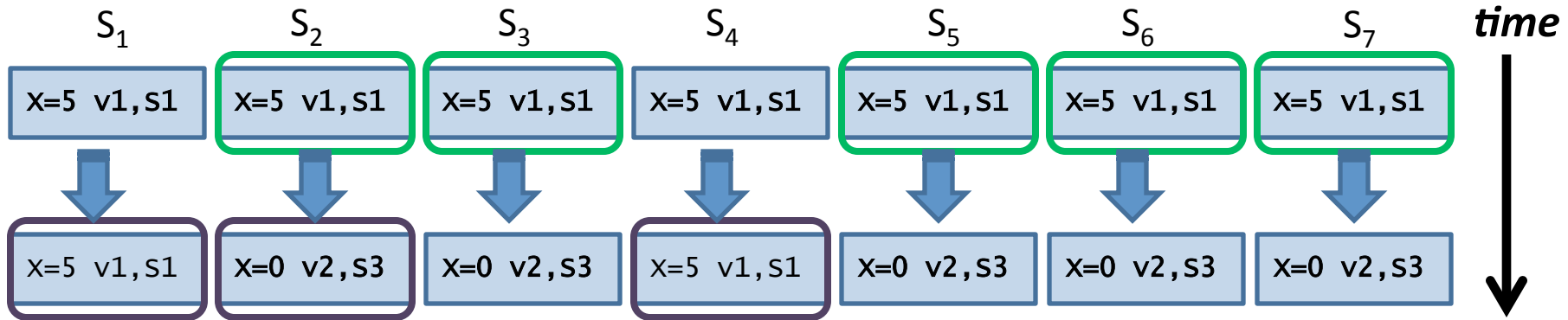
# Quorum Systems

---

- Transactional consistency works, but:
  - High overhead, and
  - Poor availability during update (worse if crash!)
- An alternative is a **quorum system**:
  - Imagine there are  $N$  replicas, a **write quorum**  $Q_w$ , and a **read quorum**  $Q_r$ , where  $Q_w > N/2$  and  $(Q_w + Q_r) > N$
- To perform a write, must update  $Q_w$  replicas
  - Ensures a majority of replicas have new value
- To perform a read, must read  $Q_r$  replicas
  - Ensures that we read *at least one* updated value



# Example



- Seven replicas ( $N=7$ ),  $Q_w = 5$ ,  $Q_r = 3$
- All objects have associated version ( $T, S$ )
  - $T$  is logical timestamp, initialized to zero
  - $S$  is a server ID (used to break ties)
- Any write will update at least  $Q_w$  replicas
- Performing a read is easy:
  - Choose replicas to read from until get  $Q_r$  responses
  - Correct value is the one with highest version

# Quorum Systems: Writes

---

- Performing a write is trickier:
  - Must ensure get entire quorum, or cannot update
  - Hence need a commit protocol (as before)
- In fact, transactional consistency is a quorum protocol with  $Q_w = N$  and  $Q_r = 1$ !
  - But when  $Q_w < N$ , additional complexity since must bring replicas up-to-date before updating
- Quorum systems are good when expect failures
  - Additional work on update, additional work on reads...
  - ... but increased **availability** during failure
- How might client-server traffic scale with  $Q_w/Q_r$ ?

# Weak Consistency

---

- Maintaining strong consistency has costs:
  - Need to coordinate updates to all (or  $Q_w$ ) replicas
  - Slow... and will block other accesses for the duration
- **Weak consistency** provides fewer guarantees:
  - e.g. **C1** updates (replica of) object **x** at **S3**
  - **S3** lazily propagates changes to other replicas
  - Other clients can potentially read old (“stale”) value
- Considerably **more efficient**:
  - Write is simpler, and doesn’t need to wait for communication with lots of other replicas...
  - ... hence is also **more available** (i.e. fault tolerant)

# FIFO Consistency

---

- As with group communication primitives, various ordering guarantees possible
- **FIFO consistency**: all updates at  $S_i$  occur in the same order at all other replicas
  - As with FIFO multicast, can buffer for as long as we like!
  - But says nothing about how  $S_i$ 's updates are interleaved with  $S_j$ 's at another replica (may put  $S_j$  first, or  $S_i$ , or mix)
- Still useful in some circumstances
  - e.g. single user accessing different replicas at disjoint times
  - I.e., client will see its writes serialised
  - Essentially primary replication with primary=last accessed

# Eventual Consistency

---

- FIFO consistency doesn't provide very nice semantics:
  - E.g. **C1** writes **V<sub>1</sub>** of file **f** to **S<sub>1</sub>**
  - Later **C1** reads **f** from **S<sub>2</sub>**, and writes **V<sub>2</sub>**
  - Much later, **C1** reads **f** from **S<sub>3</sub>** and gets **V<sub>1</sub>** – changes lost!
- What happened?
  - **V<sub>1</sub>** arrived at **S<sub>3</sub>** after **V<sub>2</sub>**, thus overwrote it (stooooopid **S<sub>3</sub>**)
- A desirable property in weakly consistent systems is that they converge to a more correct state
  - I.e. in the absence of further updates, every replica will eventually end up with the same latest version
- This is called **eventual consistency**

# Implementing Eventual Consistency

---

- Servers  $S_i$  keep a **version vector**  $V_i(\mathbf{O})$  for each object
  - For each update of  $\mathbf{O}$  on  $S_i$ , increment  $V_i(\mathbf{O})[i]$
  - (essentially a vector clock reused as a version number)
- Servers synchronize pair-wise from time to time
  - For each object  $\mathbf{O}$ , compare  $V_i(\mathbf{O})$  to  $V_j(\mathbf{O})$
  - If  $V_i(\mathbf{O}) < V_j(\mathbf{O})$ ,  $S_i$  gets an up-to-date copy from  $S_j$ ;  
if  $V_j(\mathbf{O}) < V_i(\mathbf{O})$ ,  $S_j$  gets an up-to-date copy from  $S_i$ .
- If  $V_i(\mathbf{O}) \sim V_j(\mathbf{O})$  we have a **write-conflict**:
  - Concurrent updates have occurred at 2 or more servers
  - Must apply some kind of reconciliation method
  - (similar to revision control systems, and equally painful)

# Amazon's Dynamo [2007]

---

- Storage service used within Amazon's web services
- Designed to prioritise availability above consistency:
  - SLA to give bounded response time 99.99% of the time
  - if customer wants to add something to shopping basket and there's a failure... still want addition to 'work'
  - Even if get (temporarily) inconsistent view... fix later!
- Built around notion of a so-called **sloppy quorum**:
  - Have  $N$ ,  $Q_w$ ,  $Q_r$  as we saw earlier... but don't actually require that  $Q_w > N/2$ , or that  $(Q_w + Q_r) > N$
  - Instead make tunable: **lower Q values = higher availability** (i.e. read/write) throughput
  - Also let system continue during failure

# Session Guarantees

---

- Eventual consistency seems great, but how can you program to it?
  - Need to know something about what guarantees are provided to the client
- These are called **session guarantees**:
  - Not system wide, just for one (identified) client
  - Client must be a more active participant, e.g. client maintains version vectors of objects it has read & written
- Example: **Read Your Writes (RYW)**:
  - if  $C_i$  writes a new value to  $x$ , a subsequent read of  $x$  should see this update ... even if  $C_i$  is now reading from a different replica
  - Need  $C_i$  to remember highest id of any update it made
  - Only read from a server if it has seen that update



# Session Guarantees + Availability

---

- There are many variations on session guarantees
  - All deal with allowable state on replica given history of accesses by a specific client
- Session guarantees are weaker than strong consistency, but stronger than ‘pure’ weak consistency:
  - But this means that they **sacrifice availability**
  - i.e. choosing not to allow a read or write if it would break a session guarantee means not allowing that operation!
  - ‘pure’ weak consistency would allow the operation
- Can we get the best of both worlds?

# Consistency, Availability & Partitions

---

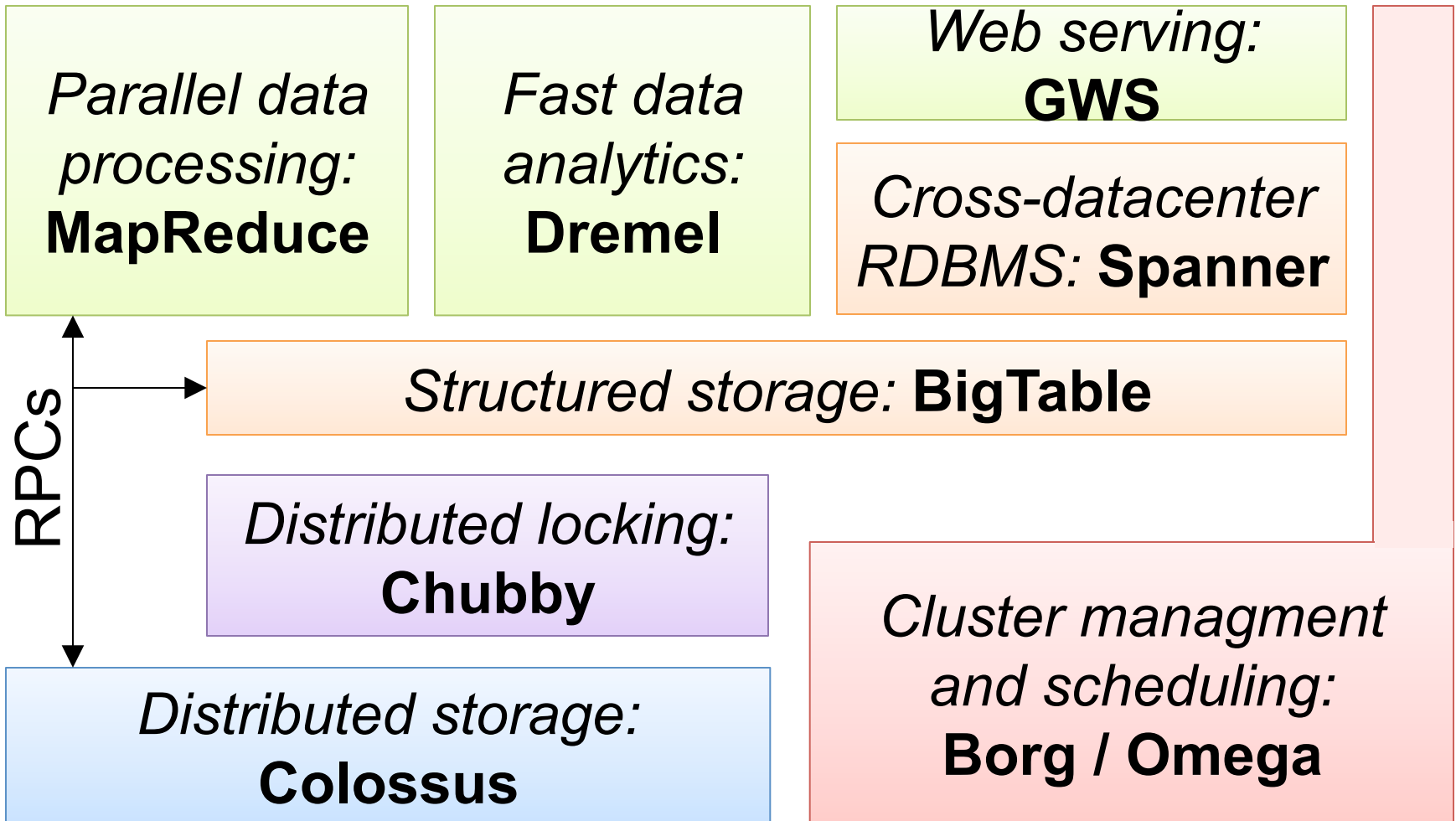
- Short answer: No ;-)
- The **CAP Theorem** (Brewer 2000, Gilbert & Lynch 2002) says you can only guarantee two of:
  - **Consistent data, Availability, Partition-tolerance**
- ... in a single system.
- In local-area systems, can sometimes drop partition-tolerance by using redundant networks
- In the wide-area, this is not an option:
  - **Must choose between consistency & availability**
  - Most Internet-scale systems ditch consistency
- **NB:** this doesn't mean that things are always inconsistent, just that they're not always **guaranteed** to be consistent

# A Google Datacentre

---

- Very brief overview of:
  - MapReduce
  - BigTable
  - Spanner

# Google: Architecture Overview



# Google's MapReduce [2004]

---

- 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
  - **Locality**: compute close to (distributed) data
- Programmer implements two methods
  - **map**(key, value) → list of <key', value'> pairs
  - **reduce**(key', value') → result
  - Inspired by functional programming
- E.g., for every word, count documents using word(s):
  - First, extract words from local documents in **map**() phase
  - Then, aggregate and generate sums in **reduce**() phase

# MapReduce: The Big Picture

Perform **Map()** query against local data matching input spec ;  
write new keys/values (e.g., 5 instances of X found here)

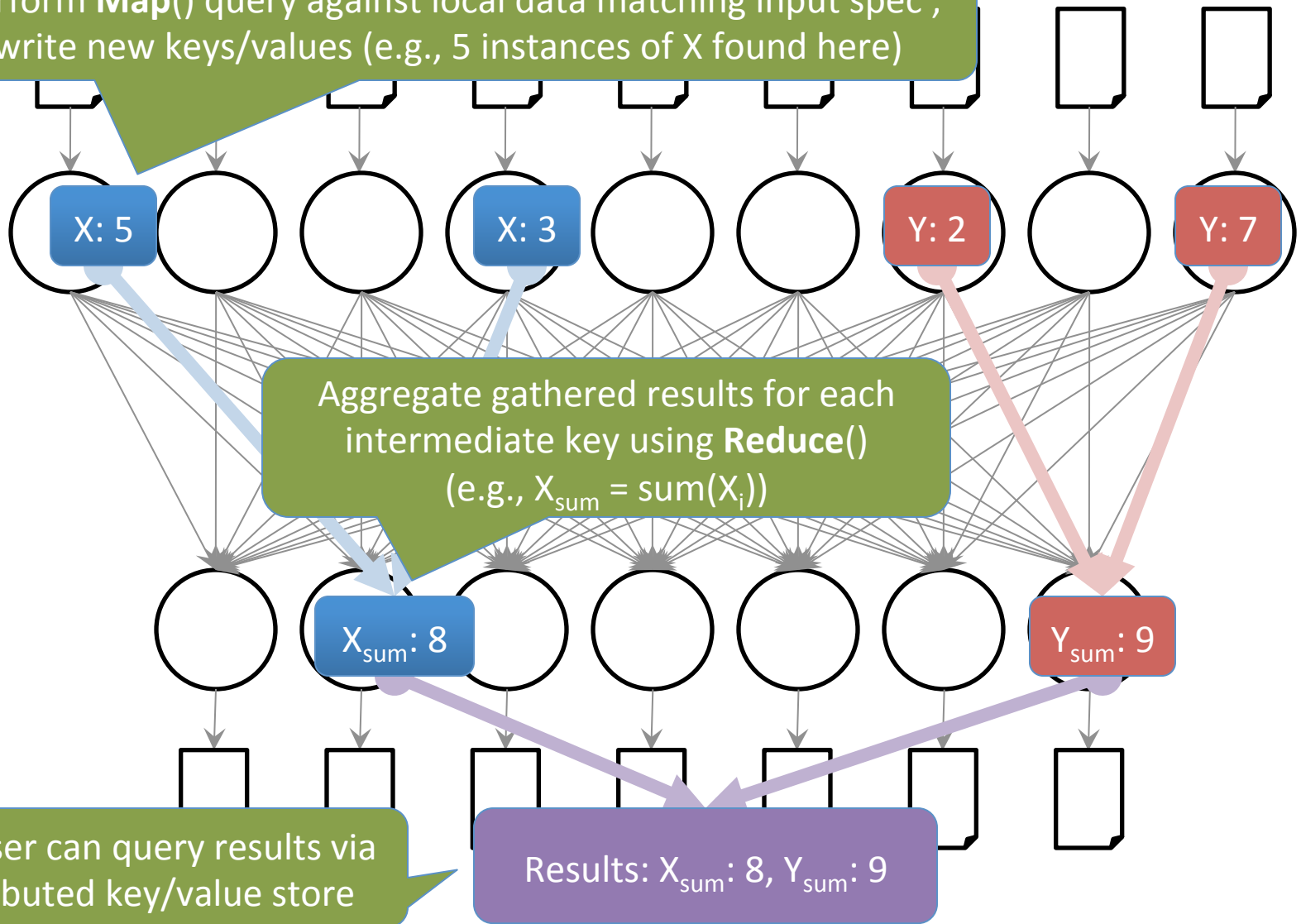
**Input**

**Map**

**Shuffle**

**Reduce**

**Output**



# MapReduce Example Programs

---

- **Sorting** data is trivial (map, reduce both identity function)
  - Works since the shuffle step essentially sorts data
- **Distributed grep** (search for words)
  - **map**: emit a line if it matches a given pattern
  - **reduce**: just copy the intermediate data to the output
- **Count URL access frequency**
  - **map**: process logs of web page access; output <URL, 1>
  - **reduce**: add all values for the same URL
- **Reverse web-link graph**
  - **map**: output <target, source> for each link to *target in a page*
  - **reduce**: concatenate the list of all source URLs associated with a target. Output <target, list(source)>

# MapReduce: Pros and Cons

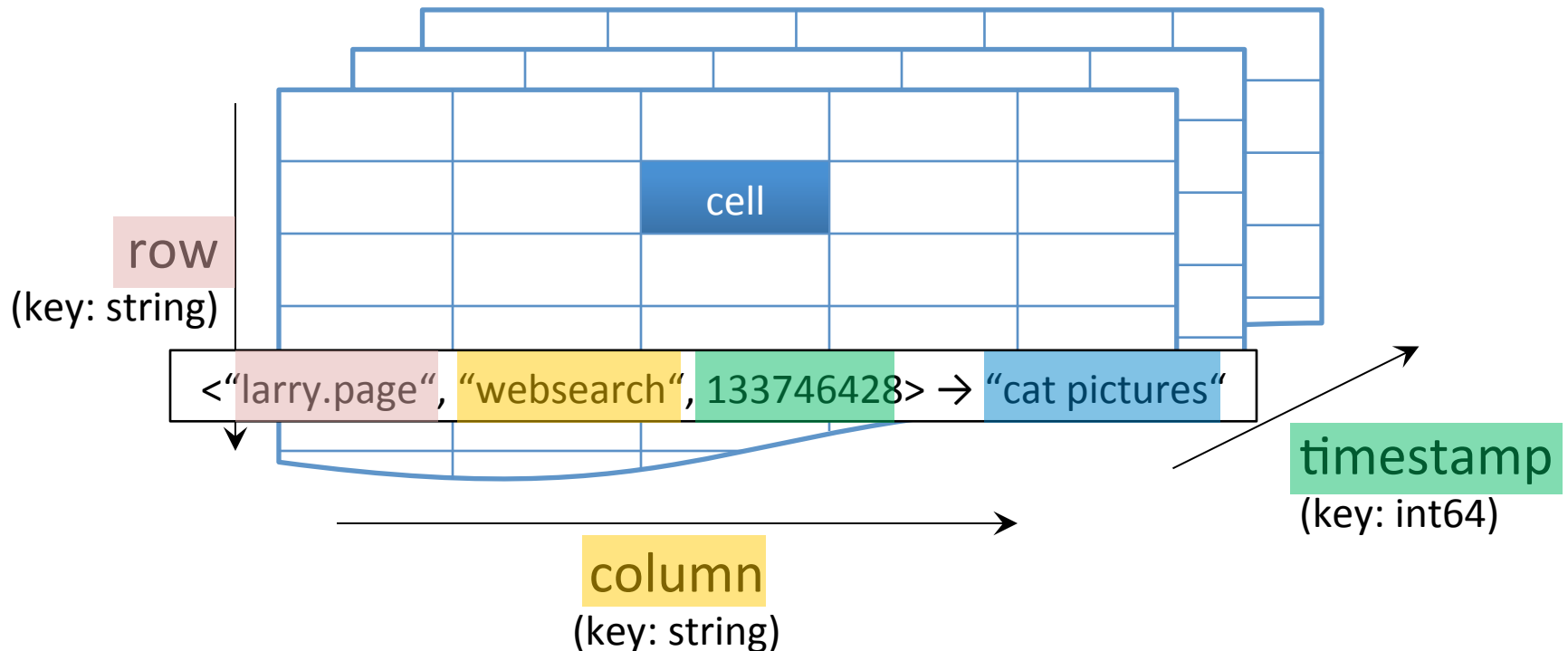
---

- **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** of the sophisticated algorithms we've seen (except storage replication)
- However not a panacea:
  - Limited to batch jobs, and computations which are expressible as a map() followed by a reduce()



# Google's BigTable [2006]

- 'Three-dimensional' structured key-value store:
  - $\langle \text{row key}, \text{column key}, \text{timestamp} \rangle \rightarrow \text{value}$
- Effectively a distributed, sorted, sparse map



# Google's BigTable [2006]

---

- Distributed **tablets** (~1 GB max) hold subsets of map
  - Adjacent rows have user-specifiable locality
  - E.g., store pages for a particular website in the same tablet
- On top of Colossus, which handles replication and fault tolerance: *only one (active) server per tablet!*
- Reads & writes within a row are **transactional**
  - Independently of the number of columns touched
  - **But:** no cross-row transactions possible
- META0 tablet is “root” for name resolution
  - Filesystem meta stored in BigTable itself
- Use **Chubby** to elect master (META0 tablet server), and to maintain list of tablet servers & schemas
  - 5-way replicated **Paxos consensus** on data in Chubby

# Google's Spanner [2012]

---

- BigTable insufficient for some consistency needs
- Often have transactions across >1 datacentres
  - May buy app on Play Store while travelling in the U.S.
  - Hit U.S. server, but customer billing data is in U.K.
- Spanner offers **transactional** consistency: full RDBMS power, ACID properties, at global scale!
- Wide-area consistency is hard
  - due to long delays and clock skew
- Secret sauce: **hardware-assisted clock sync**
  - Using GPS and atomic clocks in datacentres
  - Use global timestamps and Paxos to reach consensus
  - Still have a period of uncertainty for write TX: **wait it out!**

# Comparison

---

**Dynamo**

**BigTable**

**Spanner**

*Consistency*

eventual

weak(ish)

strong

*Availability*

high throughput,  
low latency

low throughput,  
high latency

*Expressivity*

simple key-value

row transactions

full transactions

# Next time

---

- Security [for distributed systems]
- Distributed-filesystem case studies++
- Very briefly
  - Message-Oriented Middleware (MOM)
  - Publish-Subscribe (PubSub)
- Summary and some advice on exam questions