# Distributed Systems

## Fall 2018 Exam 3 Review

Paul Krzyzanowski

Rutgers University

Fall 2018

# Question 1

In the Google cluster architecture, a search query is sent to a specific Google data center via:

(a) Hardware load balancers.

(b) Routing rules defined at each ISP.

(c) DNS-based load balancing.

(d) The address that is present in the URL.

---

(a) — That directs the query to a specific web server once it hits the data center
(b) — Routing defines the path, not the destination
(d) — The URL only identifies *google.com*

DNS (Domain Name System) servers provide *domain name → IP address*
By providing different IP addresses for different queries, DNS servers can control where clients will address their messages

# Question 2

A principle *not* employed in the Google search cluster is:

(a) Merging is efficient: divide the search among multiple systems.

(b) Use commodity PCs instead of highly-reliable computers.

(c) The price/performance ratio of a computer is more important than its peak performance.

(d) Design computers to minimize downtime by using fault-tolerant hardware.

---

(a) A query is distributed among lots of index servers; results are merged
Same principle in MapReduce: lots of *map* workers work on shards of data
Results are sorted at each *map* worker and merged at *reduce* worker

(b) — Yes, that's part of the price/performance equation
(c) — Yes, use more systems if it's more cost-effective
(d) — No.

# Question 3

If many *map* workers in MapReduce generate huge amounts of data that all share the same key:

(a) The *(key, value)* sets can be processed in parallel by multiple reduce workers.

(b) All *(key, value)* sets must be processed by a single reduce worker.

(c) It will result in an error since each *map* worker must generate data with unique keys.

(d) Multiple iterations of MapReduce may be run to process the data.

---

(a) No: That's not how MapReduce is designed to work
All values related to the same key are expected to be presented at once to a *reduce* function. In theory, you can define a partitioning function that breaks this (e.g., `random() % num_reduce_workers`)

(b) Yes

(c) No: lots of workers can generate the same key

(d) No: you can run multiple iterations but that doesn't solve this issue

# Question 4

In MapReduce, a user's *reduce* function is called once for each:

(a) *Map* worker.

(b) Unique *(key, value)* pair.

(c) *Reduce* worker.

(d) Unique key.

---

(a) No — You can have arbitrary numbers: *M map* workers and *N reduce* workers
(b) No — you'd like at most one *reduce* worker per unique *key*
(c) No — a *reduce* worker calls the function once for each unique key
(d) Yes: each *key* and all the values associated with that key

# Question 5

The *partitioning function* in MapReduce:

(a) Determines which chunks of input data will be processed by which map workers.

(b) Determines which keys will be handled by which reduce workers.

(c) Dynamically load balances the ratio of map and reduce workers.

(d) Breaks up the available servers among the concurrent users running MapReduce jobs.

---

(a) No — that's input splitting (sharding)
(b) Yes, each map worker runs a *partitioning* function on a key that determines which *reduce* worker should handle that key
(c) No — a key should be targeted to a single reduce worker
(d) No — there's no consideration for concurrent MapReduce jobs

# Question 6

MapReduce's *shuffle* phase:

(a) Gets the needed data to each reduce worker.

(b) Randomly distributes key processing among reduce workers to ensure a balanced load.

(c) Breaks up large sets of data that share the same key among multiple reduce workers.

(d) Allocates any available additional map servers to speed up processing.

---

(a) Yes — this phase includes
map worker: {output from *map* function} → partition & sort
reduce worker: copy & merge → {input to *reduce* function}
(b) No — key distribution is not random
(c) No — All keys go to the same reduce worker
(d) No — The master is responsible for allocating *map* and *reduce* workers

# Question 7

When a *reduce* worker dies in MapReduce, another one takes its place and:

(a) All the reduce workers have to restart.

(b) All the map workers have to restart to generate data for the reduce phase.

(c) Reads the same input data as the old one, with no change to any other workers.

(d) The MapReduce job restarts from the last checkpoint.

---

(a) No — reduce workers have no dependency on each other
(b) No — their output is sitting in intermediate files
(c) Yes — it contacts each map worker and requests data, just like before
(d) No — there is no checkpointing

# Question 8

When can a *reduce* worker start executing its first *reduce* function?

(a) When every single map worker has completed its work.

(b) As soon as at least one map worker has completed all of its work.

(c) When at least one map worker generates its first (key, value) pair.

(d) It can start concurrently with the map workers.

---

(a) Yes — the worker needs to be sure that *all* the data for the keys that will be needed by that worker are generated
(b) No — all the needed *(key, value)* sets may not have been generated yet
(c) No — all the needed *(key, value)* sets may not have been generated yet
(d) No — *map* processing must be finished

# Question 9

As a *Bigtable* table gets more columns added to it:

(a) Columns are split into column families and may all be distributed among multiple servers.

(b) All columns within a tablet are still managed by one server.

(c) The operation will fail since all columns must be set up at the start.

(d) New columns may be placed on other servers while the old ones remain in place.

---

(a) No — Tables are only partitioned among rows; columns are never split
(b) Yes — Columns are never split
(c) No — Column families must be configured but columns can be freely added
E.g., anchor columns to identify pages that link to a URL
(d) No — Columns are never split

# Question 10

Bigtable's *memtable* is:

(a) A cache of frequently accessed table rows stored at the client.

(b) The set of recently committed changes to Bigtable stored on a tablet server.

(c) A cache of frequently accessed table rows stored on the Bigtable master.

(d) Queued client writes to Bigtable that didn't yet get sent to a tablet server.

---

Write operations are logged and then written to a memory-based table of recent changes, called a memtable.
"As write operations execute, the size of the memtable increases. When the memtable size reaches a threshold, the memtable is frozen, a new memtable is created, and the frozen memtable is converted to an SSTable and written to GFS.

*(a, d) – No, it is a server-based structure & not present on clients*
*(c) It's not a cache of frequently-accessed content; it's recent updates*

# Question 11

Metadata tablets:

(a) Allow clients to find the tablet server responsible for a key.

(b) Store access control permissions for rows and columns of the table.

(c) Track disk and memory usage of the table.

(d) Balance the distribution of tablets among servers.

---

(a) Yes — they form a balance tree to locate a tablets responsible for a range of keys
(b) No — Access is managed by the master but is not per-row/column.
(c) No — Each tablet server tracks tablet size and the master tracks overall resource use.
(d) No — the master deals with determining tablet migration

# Question 12

*Bigtable* does *not* allow a client to:

(a) Access old versions of data in a table cell.

(b) Iterate over all keys in a range.

(c) Add columns to a table.

(d) Perform atomic changes on multiple rows.

---

(a) No — It DOES allow you to specify the latest version you want
(b) No — Content is sorted by key, making it easy to iterate through rows
(c) No — You CAN add new tables
(d) Bigtable does not provide ACID semantics, does not have transactions, and does not allow a client to lock multiple rows to make changes.

# Question 13

*Spanner* appears to violate Brewer's CAP theorem. In reality, it:

(a) Provides a CA model (consistency + availability) since partitions cannot occur.

(b) Provides an AP model (availability + partition tolerance) but sometimes gives up on consistency.

(c) Provides a CP model (consistency + partition tolerance) but sometimes loses availability.

(d) Uses the TrueTime API to provide all three: consistency, availability, and partition tolerance.

---

Spanner uses two-phase locking to lock resources and two-phase commit for committing transactions.

This means a transaction will have to wait if servers cannot be reached.

Note: because partitions are *extremely* unlikely (but can occur), you have CAP

TrueTime helps with transaction timestamps – external consistency – but has nothing to do with CAP

# Question 14

Spanner's *TrueTime* provides:

(a) A globally unique timestamp that reflects the real time of day.

(b) A way to synchronize clocks across multiple data centers.

(c) A range of time that includes the current time of day.

(d) A way to convert between time zones at different data centers so remote commit timestamps make sense.

---

(a) No — each timestamp is not guaranteed to be unique
(b) No — Each data center keeps its own time – they don't synchronize
(c) Yes — TrueTime provides a range of time, from the earliest possible time to the latest
(d) No — Time zones aren't a factor. You never want to use timezones or year-month-date-hour-minute-second times; count time since an epoch

# Question 15

*Commit wait* in Spanner is used to:

(a) Delay a commit until other concurrent transactions have completed.

(b) Delay the start of a transaction until other transactions that access the same data have committed.

(c) Wait until all resources are released before committing.

(d) Have the commit order of transactions match time of day order.

---

(a) No — other transactions don't necessarily interfere. A transaction will only be delayed under certain circumstances
(b) No — a transaction can start; it may be delayed if it needs an exclusive lock
(c) No — releasing locks is part of commit – and nothing to do with commit wait
(d) This is the whole point of *commit wait*

A transaction will wait until the transaction's *commit timestamp* is definitely in the past prior to committing and releasing locks.

This ensures that any future transaction that accesses any of that data will get a timestamp that is greater than the previous transaction.

# Question 16

Spanner can provide *lock-free reads* by:

(a) Allowing clients to read data up to a specific time in the past.

(b) Employing optimistic concurrency control techniques.

(c) Using commit wait operations in a transaction to give the illusion of concurrency.

(d) Using strong strict two-phase locking (SS2PL).

---

(a) Yes. If a transaction does not need to read the *latest* data, it can read data that is the latest up to a specified time. This provides a consistent view of the database (**snapshot**) even while other transactions are modifying data
(b) No — optimistic concurrency control techniques are not used.
(c) No — This has nothing to do with lock-free reads
(d) No — This is used for normal read/write operations; not lock-free reads

# Question 17

A *superstep* in BSP (bulk synchronous parallel) is:

(a) A sequence of computation steps that terminate in writing a checkpoint file.

(b) A set of concurrent operations that terminate in a barrier.

(c) The set of operations performed on behalf of one vertex.

(d) The period of time between computations during which messages are sent and received.

---

(a) No — a checkpoint file is not written at the end of each superstep; that would be really time consuming
(b) Yes — a a superstep ends in a barrier, where all messages must be sent to their destinations and ready for delivery to the next iteration
(c) No — BSP isn't specifically built for graphs. A superstep also is not the set of operations on one node
(d) No — this describes what happens inside a superstep

# Question 18

When a worker dies in Pregel, another worker takes over and:

(a) Computation restarts on the vertices that the dead worker was responsible for since the last superstep.

(b) All workers except those that already voted to halt must restart their computation from the last checkpoint.

(c) All workers must restart their computation from the last checkpoint.

(d) Computation restarts on the vertices that the dead worker was responsible for since the last checkpoint.

---

Vertices communicate with each other at each superstep.

You cannot restart just one worker from the last checkpoint because it will need to receive the same set of messages from vertices on other workers as it did in the past

ALL workers need to restart from the same point in time – the last checkpoint

# Question 19

Combiners in Pregel are used to:

(a) Combine multiple vertices into one vertex.

(b) Combine multiple edges into one edge.

(c) Form a disjoint union of two or more graphs, creating one graph.

(d) Optimize message delivery by combining multiple messages sent to a vertex.

---

Combiners give you the opportunities to combine multiple messages into one.

For example, suppose one worker processes 10,000 vertices and 1,000 of them all send a message to some vertex on another worker.

Instead of sending 1,000 messages to another worker, a *combiner* can pre-process them on that worker and send a single message instead.

# Question 20

A Pregel job *terminates* when:

(a) Each vertex votes to halt and no vertex receives a message.

(b) All vertices vote to halt.

(c) No vertices sent any messages to any other vertices.

(d) At least one vertex requests to halt.

---

A vertex states it is done by *voting to halt.*

However, if it receives a message in the next superstep, its state is changed to **active**

When all vertices *vote to halt* and none are made *active* again, the job is done.

# Question 21

A difference between Spark *transformations* and *actions* is that a *transformation*:

(a) Modifies an existing RDD while an action creates a new one.

(b) Only rearranges an RDD while an action performs computation on the data.

(c) Produces a new RDD while an action reads but does not generate an RDD.

(d) Is used only on original data while actions can be pipelined together on those results.

---

(a) No — RDDs are immutable
(b) No — this doesn't even make sense
(c) Yes — RDDs are original data or the result of transformations. Actions create final output
(d) No — actions do not produce results that can be processed by other actions.

# Question 22

*Multihoming* refers to:

(a) Connecting a computer to multiple networks.

(b) Replicating content across multiple computers.

(c) Sharding content across multiple computers, so each has only a part of the content.

(d) Caching frequently-used content at caching servers or proxies.

---

A *home* refers to a network address. A multihomed system is connected to multiple networks. A multihomed data center is the same: multiple ISP connections to the same data center.

# Question 23

An *Akamai CDN* load balances incoming requests by using:

(a) A hardware load balancer at each data center.

(b) Its own DNS (Domain Name System) servers that may return different addresses for the same domain name.

(c) HTTP redirection to allow a server to redirect a request to a more suitable server.

(d) Multiple computers configured to share the same IP address.

---

Akamai's DNS servers return an address representing an Akamai edge caching server that is up, near the query location (low latency), and may have the content.

# Question 24

*Quorum* in a cluster means:

(a) Having a collection of servers vote on which node runs which services.

(b) Finding out if there are enough live nodes to continue running the cluster.

(c) Propagating the state of the cluster configuration to all nodes in the system.

(d) Determining how failover of a service takes place if the node running it dies.

---

Quorum defines the minimum set of nodes that need to be functioning to keep the cluster running

# Question 25

A *cluster file system* differs from a network file system in that:

(a) It is designed to be fault tolerant by replicating content across multiple servers.

(b) It is designed to by dynamically scalable by adding more storage servers.

(c) Systems access storage at a block level instead of a file system level.

(d) It allows multiple systems to access files concurrently.

---

Multiple systems share a connection to the same storages but each system runs its own file system code. A distributed lock manager (DLM) provides mutual exclusion for critical sections.

# Question 26

The technique of fencing in a cluster refers to:

(a) Disabling or isolating a faulty component in the cluster.

(b) Establishing a perimeter of protection so outside systems cannot communicate with members of the cluster.

(c) Separating storage nodes from computing nodes.

(d) Tracking which nodes in the cluster are alive.

---

Fencing is the act of taking a component out of service

# Question 27

For Bob to send Alice a message that only Alice can read, he would encrypt it with:

(a) Bob's private key.

(b) Bob's public key.

(c) Alice's private key.

(d) Alice's public key.

---

Bob needs to encrypt the message so that only Alice can decrypt it.
Alice is the only one who knows her private key.
For Alice to decrypt a message with her private key, it must have been encrypted with her public key.

# Question 28

If it takes one day to try all combinations of a 60-bit symmetric key, how long would it take to crack a 65-bit key?

(a) 5 days.

(b) Approximately one month.

(c) Approximately one year.

(d) Approximately 32 years.

---

Each additional bit of a key doubles the search space
5 more bits → 25 = $2^5$ time bigger search space
1 day for a 60-bit key → 32 days for a 65-bit key

# Question 29

The *Diffie-Hellman* algorithm is designed to:

(a) Use a trusted third party that will distribute a session key securely to both parties.

(b) Use public key cryptography to communicate among two parties.

(c) Create a session key that can be shared with any number of participants.

(d) Allow two parties to come up with a shared key that nobody else can derive.

---

(a) No — it does not rely on a trusted third party
(b) No — it does not use public key cryptography
(c) No — it creates a common key but only among two participants
(d) Yes — it allows two parties to generate a common key using a local private 'key" and a remote public "key"

# Question 30

A *digital signature* differs from a message authentication code because:

(a) It is encrypted.

(b) It does not rely on hash functions or checksums.

(c) It requires the use of public key cryptography.

(d) It identifies the user who generated the signature.

---

(a) No — Both digital signatures and MACs are encrypted
(b) No — Both rely on hash functions
(c) Yes — that is the key difference. It also ensures non-repudiation since only one sender – the holder of the private key – has the private key
(d) No — You'd like both to identify the sender but the actual signature or MAC does not have to contain the identity of the user. For digital signatures, you will generally send a digital certificate to the receiver so they have your identity and public key: the certificate contains the identity but a certificate ≠ signature

# Question 31

The *Challenge Handshake Authentication Protocol* has this advantage over the Password Authentication Protocol:

(a) An encrypted communication channel is used to disallow sniffing the network.

(b) The password is never sent across the network.

(c) It allows the client to authenticate the server.

(d) It results in sending fewer messages over the network.

---

(a) No.
(b) Yes — CHAP was designed so that the authentication credentials sent by the client are different each session (since they are a function of the challenge)
(c) PAP can just as easily authenticate the server
(d) It actually requires one more message since the service needs to send the challenge

# Question 32

A *digital certificate* stores:

(a) Message authentication codes for a web site.

(b) Digital signatures of web pages for a site.

(c) A user's public key.

(d) A user's private key.

---

A digital certificate is a way of binding an identity to a public key.

It basically stores { name, public-key, issuer }
Where *issuer* identifies the organization that certifies the binding
The structure is signed with the issuer's private key

# Question 33

SSL uses a *hybrid cryptosystem*, which means:

(a) Public key cryptography is used to pass a symmetric session key.

(b) A different key is used to encrypt data flowing in each direction.

(c) Multiple layers of encryption are used to provide increased security.

(d) Every message has a Message Authentication Code attached to it.

---

The definition of a hybrid cryptosystem is that public key cryptography is used for key exchange and symmetric cryptography for bulk encryption

# The end