Section A

Question 1

Part a: Neo4J

1. MATCH(o: Order) - [: EMPLOYEE] -> (e: Employee {name: ‘Joe Ghosh’})  
   RETURN count(o)
2. MATCH(o: Order {ID: ‘123’}) - [:EMPLOYEE] - () - [:EMPLOYEE\_TERRITORY] - () - [:REGION] - (r: Region)  
   RETURN collect(r.name)  
   Would this work? MATCH (:Order {ID:123})-[\*3]->(r:Region) RETURN DISTINCT r
3. MATCH(s: Shipper) - [:SHIPPER] - (o: Order {ID: ‘123’}) - [:CUSTOMER] - (c: Customer)  
   RETURN s.name, c.name | MATCH(o: Order {ID: ‘123’})- [:CUSTOMER|SHIPPER] ->(p)  
   RETURN p.name

MATCH(o: Order {ID: 123}) - [:CUSTOMER|SHIPPER] -> (p) RETURN distinct p.name

1. MATCH(s: Shipper) - [:SHIPPER] - () - [:ORDER\_DETAIL] - (p: Product) - [:SUPPLIER] - (t: Supplier)  
   RETURN s.name, count(p), collect(t.name)
2. MATCH(c: Customer) - [:CUSTOMER] - () - [:EMPLOYEE] - (e: Employee {name: ‘Mark’})  
   WHERE NOT EXISTS {  
    MATCH(c: Customer) - [:CUSTOMER] - () - [:ORDER\_DETAIL] - () - [:SUPPLIER] - (s: Supplier {name: ‘Jeff’})  
   }  
   RETURN c.name

// I think for older syntax, we can use the following code https://stackoverflow.com/questions/10952332/return-node-if-relationship-is-not-present

MATCH (c:customer)

WHERE NOT (c:customer)<-[:CUSTOMER]-(o:order)-[:ORDER-DETAIL]-()-[:SUPPLIER]-(s:supplier {name: 'Jeff'})

AND (c:customer)<-[:CUSTOMER]-(o:order)-[:EMPLOYEE]-(e:employee {name: 'Mark'})

RETURN c.name

What about this:

MATCH (c:Customer)<-[:CUSTOMER]-()-[:EMPLOYEE]->(e:Employee{name: 'Mark'})

WITH c

MATCH (c:Customer)<-[:CUSTOMER]-()-[:ORDER\_DETAIL]->()-[:SUPPLIER]->(s:Supplier)

WHERE s.name <> 'Jeff'

RETURN DISTINCT c.name

Thoughts? <- Think its okay, but maybe just write (c ) for the second match? <- You are right, thanks!

How about?:

MATCH (c:Customer) - [\*2] – (e:Employee{name: “Mark”}), (c:Customer) –[\*3] – (s:Supplier)

WHERE s.name <> “Jeff”

RETURN DISTINCT c.name

Part b: MongoDB

1. db.author.find({}, {author: 1})
2. db.author.find({rating: 2.0}, {author.name: 1})  
   R
3. db.author.aggregate([  
    {  
    $geoNear: {  
    near: { type: “Point”, coordinates: [45, 47] },  
    distanceField: “distance”,  
    maxDistance: 100,  
    spherical: true  
    }  
    },  
   ])

Alternative:

db.author.find({“location”: {“$near”: [45, 47], “$maxDistance”: 100}})

1. db.author.find({comments: {$elemMatch: {upVotes: {$gt: 50}, downVotes: {$lt: 20}}}})  
     
   **Alternative**  
     
   (**Comment on that**: I think the one below is different from the above. The one above will return the author if it finds (at least) one *single* comment that contains both more than 50 upvotes *and* less than 20 downvotes.The one below will return the author if it finds 1) a comment with more than 50 upvotes and 2) a comment with less than 20 downvotes, but not necessarily the same comment. Though, I’m not sure which of the 2 cases we are requested to do.)

|  |
| --- |
| db.author.find({  $and: [  {  "comments.upvotes": { $gt: 50 },  },  {  "comments.downvotes": { $lt: 20 }  },  ],  }) |

1. db.author.find({ “comments.text”: /point/, tags: {“History” } })

OR

db.author.find( comments: {$elemMatch: {text: $regex: {“.\*point.\*”}, tags: "History”}})

Question 2

1. For Disk
   * On disk locality is important, nearby elements in a data structure should be placed close together to prevent access latency.
   * Data structures should make up a whole number of pages

For memory:

* + Data structures should be designed around the cache, I.e. so a node in a tree structure fits exactly on a cache lines
  + Can use specific cache conscious indexes
* Data could be kept denormalised, thus not requiring any join operations.
* The $lookup operator can be used in the aggregation pipeline stage.
* Could be done in the application instead of in the db

Alternative answer if you thought of embedding vs linking

* + Embedding replicates the data into the document as a sort of pre-join
  + Means more data storage
  + Means faster access which is good if the child join data is often used along with the parent
  + Linking means having a relationship/pointer to another document which contains the data you want
  + Useful for flexibility as the data may change and you don’t need to go and update a replica
  + Uses less space

1. Compared to other storage mediums

* Durable long-term storage, can last for potentially hundreds of years
* Powerful for combinatorial computations
* Very high density of information
* Silicon is a limited resource

Compared to glass

* + Glass is very breakable; DNA can be stored in a liquid medium resistant to being bumped around
  + Can’t exploit chemical parallelization in glass
  + Making a copy requires making a new piece of glass
  + Glass storage contributes to Silicon scarcity, DNA does not
  + Flash pages have limited write cycles, we want to balance writes across every page to improve lifetime
  + Because an erase is an unnecessary operation which contributed to wear we need to maintain invalid pages
  + This is managed by the FTL

Section B

Question 3

Part a: Definitions probably won’t be asked but this was helpful for my revision

1. The CAP theorem is the idea that a scalable system can’t have consistency, availability and partition tolerance at a high level. One must be compromised.
2. In datacenters racks are now considered the unit of deployment not individual servers. In future we would like to divorce resources from their physical layout and just think of allocating resources to an application regardless of how those resources are distributed.
3. Used to deal with temporary failures in the Dynamo system. If a node is unresponsive a replica will be sent to the next node after the first N in the preference list with a hint in the metadata of where the replica was meant to go. The node will store this data in a separate database and attempt to send it when the original node is healthy again.
4. A public cloud is where a company provided cloud infrastructure over the internet to the public, typically anyone with a credit card. Alternatively a private cloud is where the cloud infrastructure is just used internally by the company who manages it. A hybrid cloud is where both types of infrastructure are used an channels are setup so data and even applications can be shared.

Part b: Spark

1. Clients define the way a computation has to be performed i.e the business logic through Transformations and are then able to perform the computation using Actions.
2. Transformations: filter, flatMap, map, groupByKey, reduceByKey  
   Actions: collect, count, reduce, save  
     
   Transformations result in another RDD, actions result in a returned value to the program or write data to external storage
3. val words = [...];

val lines = spark.textFile(...);

val splitAndFilterStage = lines.flatMap(line => line.split(" ")).filter(word => words.contains(word));

val mapStage = splitAndFilterStage.map(word => (word, 1));

val counts = mapStage.reduceByKey(\_ + \_); // reduce by key is not actions

counts.save(...) // call actions to do the computation

1. (This is a wild guess) Partition the key-value store based on the key, so that when a value at a certain key is updated, only any RDDs with data in that partition in their lineage will need to be recomputed. This can be done by keeping a global overview of which partitions have been updated since an RDD was last computed – if a partition has been updated, then when an action is run, any partitions which have been updated should be recomputed. This doesn’t limit scalability – the key-value store can be as big as it wants, it’s fault tolerant since if a node fails, then only that partition needs to be recomputed, and it’s consistent since if a partition is updated, then the results are recomputed, so they’re always consistent with the state of the dataset at that time.

Alternatively, see: <https://spark.apache.org/docs/latest/streaming-programming-guide.html>

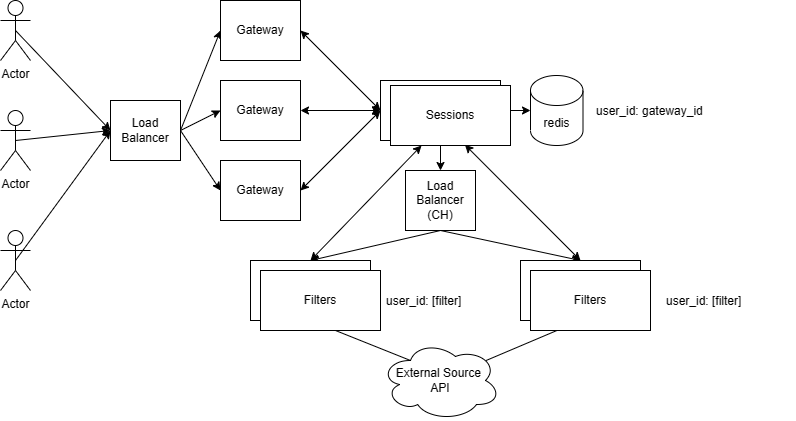
Question 4

Do we know if there are any previous examples on what sort of thing they’re looking at for this question?

I made a simple design here, any comments/critics are welcomed:

We assume the performance of the External Source API is not affected by the number of connections we make, and it never fails (since this is not part of our design and we can’t control this).

(Ignore the “Actor” labels, I could not find a way to hide it in draw.io :(



Load Balancer (left): A load balancer which forwards the clients’ connection requests to one of the Gateway servers via some load balancing algorithm (RR, random, consistent hashing, etc.)

Gateway: Responsible for hosting websocket connection with multiple clients. When a wss connection is established (or when client changed its filters), it forwards (client\_id, gateway\_id, client\_filters) to Sessions server.

Sessions: When it received the said triple from any gateway, it adds user\_id to gateway\_id to its redis. It also forwards (user\_id, client\_filters) to the load balancer CH. It also receives (alert\_message, [user\_id]) from Filters servers and forward the alert to the corresponding gateway that connects to these users. Replica Sessions continuously pull data from master and carry on with the task when master fails.

Load Balancer (CH): Use consistent hashing (upon user\_id) to determine which Filters server should the said tuple go to.

Filters: Stores user\_id to its corresponding filters in memory (or maybe filter to [user\_id] if many will subscribe to same filter? Depend on which is more efficient), Filters replica pulls updates and store in memory as well. It also listens to the External Source API for new alerts. Once it gets a new alert, it calculates which users will receive the alert and sent the alert with the user list back to sessions. (This is computationally expensive, so we have multiple Filters).

Monitor: Coordinates configurations, detects server failures, etc. (zookeeper, so distributed, no SPoF) (Not shown in the diagram).

Scalability:

When more clients want to connect, we simply add more Gateways. This will not affect the rest of the system. If the LB is overloaded, we can add more LB in parallel, and use DNS load balancing technique to randomly assign clients to one of the load balancers (also improve fault-tolerance).

The total storage capacity of Sessions can also be easily scaled since it is based on redis. As Sessions itself does not involve much computation, we might not need to scale this component (is this argument convincing?).

When more clients are connected, we simply add more Filters in parallel. The demarcation between Filters will be automatically decided by the Load Balancer CH. (Method I use to make LB CH fault-tolerant can also improve its scalability when it is overloaded).

Fault-tolerance:

If one of the Gateways is down, the rest can subsume its part of the work, thanks to the load balancer (The client will indeed disconnect but they will be notified by the failure and can reconnect, so I don’t think this is a big issue, different opinions are welcomed).

Sessions and Filters both uses master-replica approach. When the master machine is down, the replica can carry on with the job. (Although I do admit master-replica is expensive and we don’t have the read speedup it brings, so not optimal. Any suggestions?)

To avoid Load Balancer (CH) to become a SPoF, we can have multiple LB CH in parallel (with the same hash function) and let Sessions to randomly (or in round-robin order, anything) send messages to these LBs. (Or maybe we can use a DMQ like Kafka, with exactly-once-processing guarantee for messages. LBs are consumers and Sessions is the producer. Too heavyweight?)

Monitor uses zookeeper, which can be distributed across multiple machines, hence no SPoF there.

Preferably, the said servers are distributed across multiple datacenters, so the datacenter itself does not become a SPoF.