INSTITUTE OF TECHNOLOGY TRALEE

WINTER EXAMINATIONS AY 2014-2015

# Advanced Database Programming

**DBMS 81001**
**CRN 48064**

**External Examiner**: Mr. Sean McHugh

**Internal Examiner**: Mr Peter Given

**Duration**: 2 Hours

**Instructions to Candidates:**

i) Answer any **three** questions.

ii) All questions carry equal marks. Submit all your rough-work, marks may be lost otherwise.

---

Question 1 :

i) Explain Master-Master Replication in CouchDB and explain using an example how conflicts are dealt with. (16 marks) ... notes Week 1

…...

- According to CouchDB, the safest place to **store your data is everywhere**, and it gives you the tools to do it. It supports called multi-master or **master-master** replication.
- Each CouchDB server is equally able to **receive updates, respond to requests, and delete data, regardless** of whether it's able to connect to any other server.
- In this model, changes are selectively replicated in one direction, and all data is subject to replication in the same way.
- In other words, there is **no sharding**.
- Servers participating in replication will all have all of the data.

...

- It prevents **conflicts** by ensuring only the most recent document revisions are modified.
- There are **no transactions or locking** in CouchDB.
- To modify an existing record, you first read it out, taking note of the _id and _rev.
- Then, you request an update by providing the full document, including the _id and _rev.
- All operations are first come, first served.

- By requiring a matching _rev, CouchDB ensures that the document you think you're modifying hasn't been altered behind your back while you weren't looking.

**Additional Info**

- CouchDB also keeps old values around to deal with conflicts.

  Example
  Lets say you were maintaining a calendar of events one is on your mobile database and the other is on your laptop database. Lets say you get a text about the date and venue and update the phone database. Then you get an email and update the laptop database and replicate between them. Couch will not know which value is correct so this is a good reason to keep old values around as it improves consistency.

//**********

ii) CouchDB is made "of the web, for the web". Discuss. (9 marks)
.....

- CouchDB was designed with the **Web** in mind and all the innumerable **flaws, faults**, failures, and glitches.
- CouchDB offers a **robustness** unmatched by most other databases.
- Other systems tolerate occasional network drops, CouchDB **thrives even when connectivity** is only rarely available.
- CouchDB aims to support a **variety of deployment** scenarios from the datacenter down to the smartphone.
- You can run CouchDB on your Android phone, on your MacBook, and in your datacenter.
- **Straightforward to layer** in web technologies—such as load balancers and caching layers—and still end up with something that's true to CouchDB's APIs.
- CouchDB's attention to robustness in the face of uncertainty makes it a great choice if your system must stand up to the **harsh realities of the Internet**.
- By leveraging standard webisms like **HTTP/REST and JSON**, CouchDB fits in easily wherever web technologies are prevalent, which is increasingly everywhere.
- A system for developing and **deploying web apps directly** through CouchDB with no other middleware.

//**********

iii)
Discuss the strengths and weaknesses of CouchDB (8 marks)

**CouchDB's Strengths**

- CouchDB is a **robust and stable**.
- Built on the philosophy that **networks are unreliable and hardware failure** is imminent, CouchDB offers a heartily decentralized approach to data storage.

- **Small enough** to live in your smartphone and **big enough** to support the enterprise, CouchDB affords a variety of deployment situations.
- CouchDB is as much an **API** as a database.
- There are an increasing number of **alternative** implementations and CouchDB service providers built on hybrid back ends.
- Because CouchDB is made "of the Web, for the Web," it's fairly **straightforward to layer** in web technologies—such as load balancers and caching layers—and still end up with something that's true to CouchDB's APIs.

**CouchDB's Weaknesses**

- CouchDB's mapreduce-based views, while novel, can't perform all the fancy data slicing you'd expect from a relational database.
- In fact, you shouldn't be running ad **hoc queries** at all in production.
- Also, CouchDB's **replication strategy** isn't always the right choice.
- CouchDB replication is all or nothing, meaning all replicated servers will have the same contents.
- There is **no sharding** to distribute content around the datacenter.
- The principal reason for adding more CouchDB nodes is not to spread the data around so much as to increase throughput for read and write operations.

//**********

Question 2:

i) Explain the programming model used in the code below. Explain what the code below achieves and show some sample output, noting that there is a collection of 100,000 phone numbers (with "country", "area", "prefix" and "number" fields) between 1-800-555-0000 and 8-800-565-9999 in the '***phones'*** collection and that ***distinctDigits*** is a function that extracts an array of all distinct numbers

**(16 marks)**

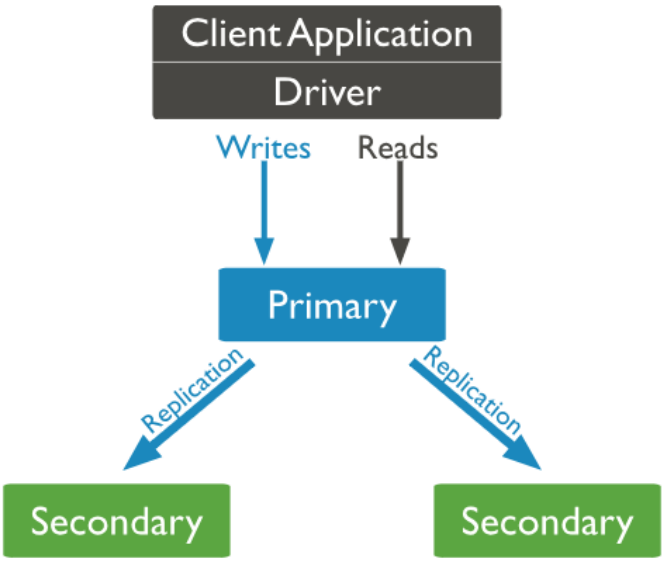
```
m = function() {
var digits = distinctDigits(this);
emit({digits : digits, country : this.components.country},
{count : 1});
}
r = function(key, values) { var
total = 0;
for(var i=0; i<values.length; i++) {
total += values[i].count;
  }
return { count : total }; }

results = db.runCommand({ mapReduce: 'phones',
map: m, reduce: r,
out: 'phones.report' })
```

//**********

ii) Using a diagram, discuss the use of replica sets and explain why Mongo prefers an odd number of nodes in a replica set (**9 marks)** Week 2 & 3

……




**Description of Diagram above,**

- A replica set contains seven bearing nodes and one arbiter nodes. Out of the seven bearing nodes a primary is chosen.
- Primary node receives all write operations. Primary records all changes to its data set in operation log
- The secondary datasets replicate the primary log and apply operations to there datasets to reflect primary.

**Replica Sets**

- Mongo was built to scale out, not to run stand-alone.
- A replica set in MongoDB is a group of mongod processes that provide redundancy and high availability.
- It was built for data consistency and partition tolerance.
- Can't write to a secondary node or can't directly read from it.
- There is only one master per replica set, and you must interact with it.
- It is the gatekeeper to the set.
- Replicating data has its own issues not found in single-source databases.
- In the Mongo setup, one problem is deciding who gets promoted when a master node goes down.
- Mongo deals with this by giving each mongod service a vote, and the one with the freshest data is elected the new master.
- A write in a Mongo replica set isn't considered successful until most nodes have a copy of the data.

**Odd number of nodes in a replica set**

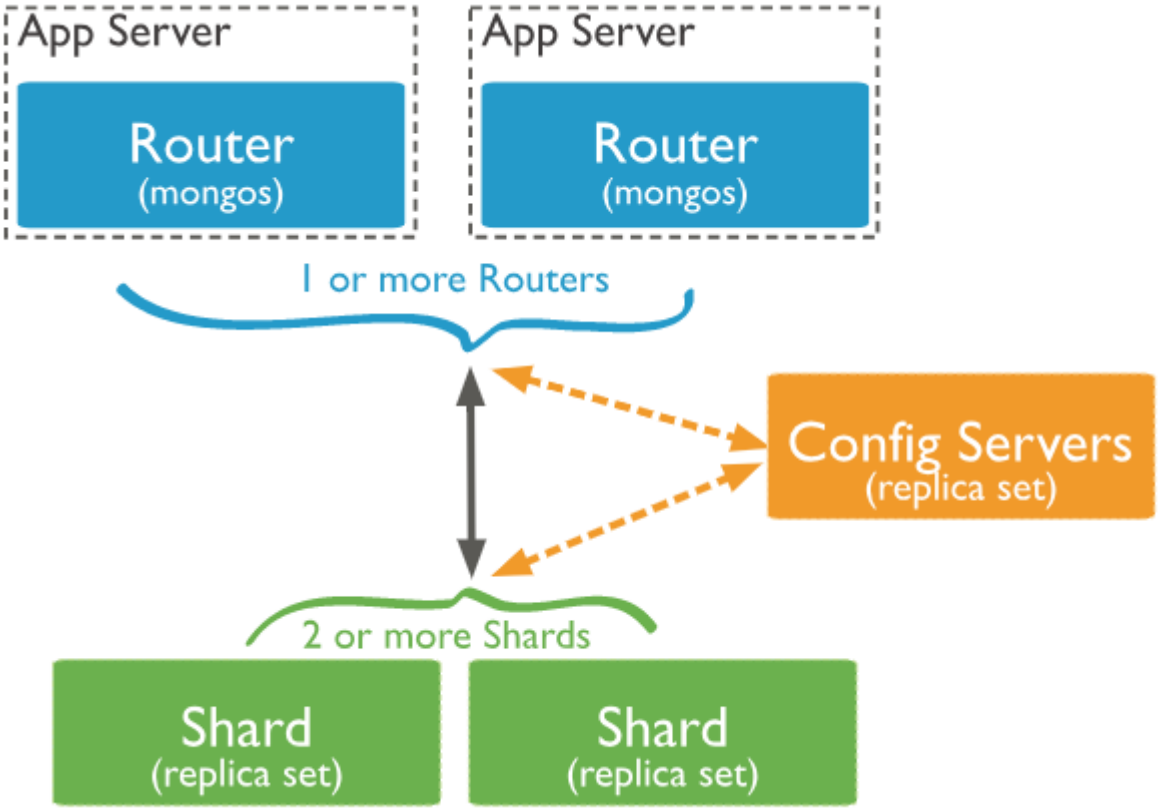
- MongoDB expects an odd number of total nodes in the replica set.
- Consider a five-node network, for example.
- If connection issues split it into a three- node fragment and a two-node fragment, the larger fragment has a clear majority and can elect a master and continue servicing requests.
- With no clear majority, a quorum couldn't be reached.
- To see why an odd number of nodes is preferred, consider what might happen to a four-node replica set.
- Say a network partition causes two of the servers to lose connectivity from the other two.
- One set will have the original master, but since it can't see a clear majority of the network, the master steps down.
- The other set will similarly be unable to elect a master because it too can't communicate with a clear majority of nodes.
- Both sets are now unable to process requests and the system is effectively down.
- Having an odd number of total nodes would have made this particular scenario—a fragmented network where each fragment has less than a clear majority—less likely to occur.
- Some databases (e.g., CouchDB) are built to allow multiple masters, but Mongo is not, and so it isn't prepared to resolve data updates between them
- MongoDB deals with conflicts between multiple masters by simply not allowing them.
- Mongo always knows the most recent value; the client needn't decide.
- Mongo's concern is strong consistency on writes, and preventing a multimaster scenario is not a bad method for achieving it.
- You may not always want to have an odd number of servers replicating data.
- In that case, you can either launch an arbiter (generally recommended) or increase voting rights on your servers (generally not recommended).

//**********

iii) Using a diagram, explain how Mongo handles very large data sets **(8 marks)**

……..

The following graphic describes the interaction of components within a sharded cluster:




MongoDB shards data at the collection level, distributing the collection data across the shards in the cluster.

- Mongo's primary strength lies in its ability to handle huge amounts of data (and huge amounts of requests) by replication and horizontal scaling.
- MongoDB uses sharding to support deployments with very large data sets and high throughput operations.
- Sharding is a method for distributing data across multiple machines.
- One of the central reasons for Mongo to exist is to safely and quickly handle very large datasets.
- Rather than a single server hosting all values in a collection, some range of values are split (or in other words, sharded) onto other servers.
- For example, in our phone numbers collection, we may put all phone numbers less than 1-500-000-0000 onto Mongo server A and put numbers greater than or equal to 1-500-000-0001 onto a server B.
- Mongo makes this easier by autosharding, managing this division for you.
- Like replica sets, there's a special parameter necessary to be considered a shard server (which just means this server is capable of sharding).
- Because Mongo is focused on large datasets, it works best in large clusters, which can require some effort to design and manage.

**Horizontal Scaling**

- Horizontal Scaling involves dividing the system dataset over multiple servers adding additional servers to increase capacity.

**Vertical Scaling**

- Vertical scaling involves spreading system dataset over a single server, adding more GPU/RAM to increase performance.

//**********

Question 3:

**i)** Appendix 1 shows a graph database. Explain how the following Gremlin queries arrive at a result (13 marks)

**a.** g.V.filter{it.name=='Wine Expert Monthly'}.outE.inV.name

**b.** ***alice***.bothE('friends').bothV.name (Note ***alice*** is a reference to the Vertex named "Alice")

**c.** ***alice***.bothE('friends').bothV.except([alice]).loop(3){it.loops <= 2}.name (Note ***alice*** is a reference to the Vertex named "Alice")

**d.**
```
wines_count = [:]
g.V.outE('likes').outV.name.groupCount( wines_count)

wines_count
```

…..
Week 5

//**********

**ii)** Using a diagram, discuss Neo4J's distribution models and compare it to MongoDB's distribution model. **(12 marks)** Week 5

Week 5

A relational **model** may create a category table and a many-to-many relationship between a single winery's wine and some combination of categories and other data.

But this isn't quite how humans mentally **model** data.
…...

The Neo4j **distributions** provide several tools for fast lookups with Lucene and easy-to-use (if sometimes cryptic) language extensions like Gremlin and the REST interface.
…….
If you choose to **distribute**, the name "high availability" cluster should give away their strategy.
…..
**Neo4j via Gremlin**
There are several languages that interoperate with Neo4j: Java code, REST, Cypher, Ruby console, and others.

The one we'll use is called Gremlin, which is a graph traversal language written in the Groovy programming language.

//**********

**iii)** Discuss the strengths and weaknesses of Neo4J

**(8 marks)**

**Neo4j's Strengths**

- Neo4j is one of the finest examples of **open source graph databases**.
- Graph databases are perfect for **unstructured data**, in many ways even more so than document datastores.
- Not only is Neo4j typeless and schemaless, but it puts no constraints on how data is related.
- It is, in the best sense, a free-for-all. Currently, Neo4j can support 34.4 billion nodes and 34.4 billion relationships, which is more than enough for most uses (Neo4j could hold more than 42 nodes for each of Facebook's 800 million users in a single graph).
- The Neo4j distributions provide several tools for fast lookups with Lucene and easy-to-use (if sometimes cryptic) language extensions like Gremlin and the REST interface.
- Beyond ease of use, Neo4j is fast. Unlike join operations in relational databases or map-reduce operations in other databases, graph traversals are constant time.
- Like data is only a node step away, rather than joining values in bulk and filtering the desired results—as most of the databases we've seen operate.
- It doesn't matter how large the graph becomes; moving from node A to node B is always one step if they share a relationship. Finally, the Enterprise edition provides for highly available and high read- traffic sites by way of Neo4j HA.

**Neo4j's Weaknesses**

- Neo4j does have a few shortcomings.
- Edges in Neo4j cannot direct a vertex back on itself.
- We also found its choice of nomenclature (node rather than vertex, and relationship rather than edge) to add complexity when communicating
- Although HA is excellent at replication, it can only replicate a full graph to other servers.
- It cannot currently shard subgraphs, which still places a limit on graph size (though, to be fair, that limit measures in the tens of billions).
- Finally, if you are looking for a business-friendly open source license, Neo4j may not be for you.
- Where the Community edition is GPL, if you want to run a production environment using the Enterprise tools (which includes HA and backups), you'll probably need to purchase a license.

//**********

Question 4:

i) "***It can be a bit difficult to classify exactly what Redis is***." Discuss this statement using examples where appropriate **(16 marks)** Week 4

….

- At a basic level, it's a key-value store, of course, but that simple label doesn't really do it justice
- Redis supports advanced data structures, though not to the degree that a document-oriented database would.
- It supports set-based query operations but not with the granularity or type support you'd find in a relational database.
- And, of course, it's fast, trading durability for raw speed.
- In addition to being an advanced data structure server, Redis is a blocking queue (or stack) and a publish-subscribe system.
- All of this makes Redis more of a toolkit of useful data structure algorithms and processes than a member of any specific database genre.
- Redis' expansive list of client libraries makes it a drop-in option for many programming languages.

//**********

ii) Discuss the durability options in Redis and say when they might be used (9 marks)

**Durability**

- Redis has a few persistence options.
- First is no persistence at all, which will simply keep all values in main memory.
- If you're running a basic caching server, this is a reasonable choice since **durability** always increases latency.
- One of the things that sets Redis apart from other fast-access caches like memcached is its built-in support for storing values to disk.
- By default, key- value pairs are only occasionally saved.
- You can run the LASTSAVE command to get a Unix timestamp of the last time a Redis disk write succeeded, or you can read the last_save_time field from the server INFO output.
- You can force **durability** by executing the SAVE command (or BGSAVE, to asynchronously save in the background).
- Another **durability** method is to alter the snapshotting settings in the configuration file.

**Snapshotting**

- We can alter the rate of storage to disk by adding, removing, or altering one of the save fields.
- By default there are three, prefixed by the save keyword followed by a time in seconds and a minimum number of keys that must change before a write to disk occurs.
- For example, to trigger a save every 5 minutes (300 seconds) if any keys change at all, you would write the following:

save 300 1

- The configuration has a good set of defaults.

- The set means if 10,000 keys change, save in 60 seconds; if 10 keys change, save in 300 seconds, and any key changes will be saved in at least 900 seconds (15 minutes).

save 900 1
save 300 10
save 60 10000

You can add as many or few save lines as necessary to specify precise thresholds.

**Append-Only File**

- Redis is eventually durable by default, in that it asynchronously writes values to disk in intervals defined by our save settings, or it is forced to write by client-initiated commands.
- This is acceptable for a second-level cache or session server but is insufficient for storing data you need to be durable, like financial data. (2nd Level Cache – on disk, 1st Level Cache in memory, which reduces the calls made to the DB server, by conserving data already loaded from the database. Database access is therefore, necessary only when the retrieving data is currently not available in the cache.)
- If a Redis server crashes, our users might not appreciate having lost money.
- Redis provides an append-only file (*appendonly.aof*) that keeps a record of all write commands.
- If the server crashes before a value is saved, it executes the commands on startup, restoring its state; *appendonly* must be enabled by setting it to yes in the *redis.conf* file.

*appendonly yes*

- Then we must decide how often a command is appended to the file.
- Setting always is the more durable, since every command is saved.
- It's also slow, which often negates the reason people have for using Redis.
- By default *everysec* is enabled, which saves up and writes commands only once a second.
- This is a decent trade-off, since it's fast enough, and worst case you'll lose only the last one second of data.
- Finally, no is an option, which just lets the OS handle flushing.
- It can be fairly infrequent, and you're often better off skipping the append-only file altogether rather than choosing it.

*# appendfsync always*
*appendfsync everysec*
*# appendfsync no*

- Append-only has more detailed parameters, which may be worth reading about in the config file when you need to respond to specific production issues.
- Many **durability** and replication settings that conform to whatever your needs may be.

**Strengths**

- Beyond even a data structure store, however, Redis's **durability** options allow you to trade speed for data safety up to a fairly fine point.
- Built-in master-slave replication is another nice way of ensuring better **durability** without requiring the slowness of syncing an append-only file to disk on every operation.

//**********

iii) Discuss four features which Redis provides that Memcachd doesn't (8 marks)

…..

- One of the things that sets Redis apart from other fast-access caches like memcached is its built-in support for storing values to disk.
- Selectively deleting/expiring items in the cache.
- Ability to query keys of a particular type.
- Persistence - essential for objects that seldom change