Scalability Crash Course

Scaling the database:

***Importance of Database Setup:***

Directions: run the SQL script *sqlCreateModifyScale1*, *insertScript1.js* on Node and test the following queries, then drop the database and recreate by running SQL createModifyScale1, and rerun the queries.

|  |  |  |  |
| --- | --- | --- | --- |
| Query | Duration/fetch | *modified*  Duration/fetch | Difference/  efficiency |
| SELECT \* FROM businesses WHERE owner\_first\_name = 'else' LIMIT 15000 |  |  |  |
| SELECT \* FROM businesses WHERE business\_name = 'Klein Inc' LIMIT 15000 |  |  |  |

What were the differences seen? What is the difference between setting something as TEXT or VARCHAR? Do you expect to see differences? Why or why not?

***Importance of Indexing:***

Directions:

1. Run the SQL script *sqlCreateScale2* and insert 10000 businesses using *insertRowBusinesses*. Record the query time.
2. Figure out how to add an index in MySQL and add a **Hash Index**(what is the difference between HASH and BTREE?)on all of the *businesses* table’s column as well as the country column in *Addresses* table.
3. Run the same query to insert 10000 businesses. Record the query time.
4. Run insertScript2.js on Node and test the following queries.
5. Add a HASH index to the city (go to edit and preferences to change the timeout time to at least 30 minutes). While this script is running(remember, this has to index 10 million records), move on to **JOINS**)

|  |  |  |  |
| --- | --- | --- | --- |
| Query | Query time scale2 *unindexed*  Duration | Query time scale2 *indexed*  Duration | Difference/  efficiency |
| *insertRowBusiness* (paste onto MySQL WB) |  |  |  |
| *SELECT \* FROM businesses WHERE owner\_first\_name = “else” (indexed)* |  |  |  |
| *SELECT \* FROM addresses WHERE city = 'gussie bury'* |  |  |  |

***Importance of joining on foreign keys:***

Foreign keys automatically use primary keys which are indexed to join tables. See the performance difference when you do not use a foreign key.

|  |  |  |
| --- | --- | --- |
| Query | Query time scale2  Duration | Difference/  efficiency |
| *SELECT \* FROM scale2.businesses AS b*  *JOIN addresses AS a ON a.country = b.country*  *WHERE b.owner\_first\_name = 'else'*  *LIMIT 10000* |  |  |
| *SELECT \* FROM scale2.businesses AS b*  *JOIN addresses AS a ON a.business\_id = b.id*  *WHERE b.owner\_first\_name = 'else'*  *LIMIT 10000* | 6.985 sec / 0.031 sec  10000 row(s) returned |  |
| *SELECT \* FROM scale2.businesses*  *JOIN business\_infos ON department = ‘electronics’*  *WHERE owner\_first\_name = 'else'*  *LIMIT 10000* |  |  |

***Bad Queries:***

Bad queries are those that are looking through more data than they should to get the job done. A good tool to use when queries are taking too long is to use EXPLAIN in front of your query. MySQL will show you what it is currently looking through to find the desired data.

Direction:

Run these queries and figure out what data they are retrieving, then fix them.

|  |  |  |
| --- | --- | --- |
| Query | Query time scale2  Duration | Difference/  efficiency |
| *SELECT country, business\_name FROM businesses*  *JOIN addresses ON addresses.business\_id = businesses.id*  *JOIN business\_infos ON business\_infos.business\_id = businesses.id*  *WHERE business\_name IN*  *(SELECT business\_name FROM businesses*  *JOIN business\_infos ON business\_id = businesses.id*  *WHERE number\_of\_like > 800000)* |  |  |
| *SELECT count(country), country FROM businesses*  *JOIN addresses ON addresses.business\_id = businesses.id*  *JOIN business\_infos ON business\_infos.business\_id = businesses.id*  *WHERE business\_name IN*  *(SELECT business\_name FROM businesses*  *JOIN business\_infos ON business\_id = businesses.id*  *WHERE number\_of\_like > 800000)*  *GROUP BY country* |  |  |
| *Fixed Query:* |  |  |
| *Fixed Query:* |  |  |

***Sharding/Partitioning Data:***

Your database may get to big one day (RARE) and it may be better to separate it into larger pieces. This is usually done to reduce the size you are querying against or to purge old data. Partitioning is a method of splitting up your database across multiple physical storage. Due to the nature of innoDB tables(what we are use to using), we cannot partition

Direction:

1. Run this query:
   1. alter table business\_infos

partition by HASH(id)

partitions 200.

This will divide your business\_infos table into 200 separate tables based on their ids. What kind of queries can you run here? Note: this cannot be done with innoDb engines. What are the differences between these 2?

How can we divide a database vertically? What is the method for doing this? Why?

*FINALLY,* ***CACHE***

Supposedly, 90% of facebook data never hits the database because they cache so many of their queries. Let’s cache some queries for ourselves.

Using these commands:

SHOW VARIABLES LIKE 'query\_cache%' //shows cache variables

SHOW VARIABLES LIKE 'have\_query\_cache' //shows if cacheing is available

SET GLOBAL <variable\_name> //sets variable

***Cache is set by bit size. So to have a cache of 1 kilobyte, its 8000(8 bit per byte, 8 byte per kilobyte)***

See the difference of some queries you ran earlier.

***Connecting the front end:***

**Load Test**

Only after you have completely optimized one server should you utilize another server. This way, every server is optimized for its performance. This is simple math in that you should have 2 100% efficient servers rather than 4-50% efficient servers. Initially, the load test could not handle 10 request in 10 seconds, we will see how much better the code can get!

***Code problems:***

The easiest way to make sure your code scales to handle more user is locate areas of bad code. This can be database errors or how you handle data. Check for chunks of code that can be fixed or be optimized.

Directions:

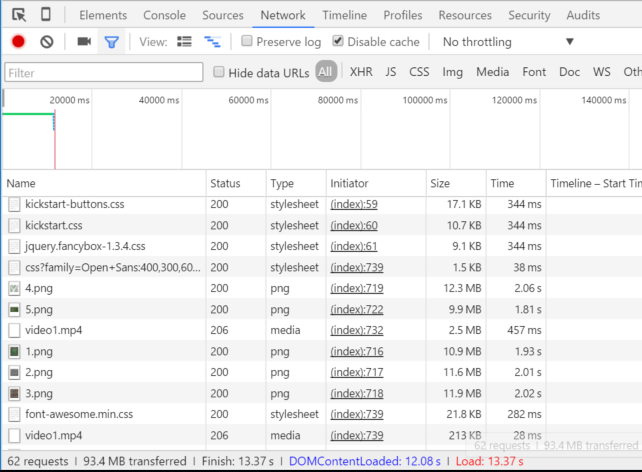
Fix the chunks of code in server.js that are not optimized. Record the differences.

|  |  |  |
| --- | --- | --- |
| Fixes | Response Time Before | Response Time After |
| *Runtime algorithm(N^2 vs N^3)* |  |  |
| *Inefficient db write usage* |  |  |
| *Inefficient db read usage* |  |  |

**Compression:**

Compression allows you to send large files at the speed of those over 4 times smaller. Compression is like deflating a large medicine ball and delivering that for the ***browser*** to inflate. Of course, the ***browser*** has to have a pump in order to inflate it and of course, the ***browser*** should TELL you if it has a pump or not before you should ship him a useless ball. This is done by checking the request headers:





*What files are we sending through to the browser that can/should be compressed?*

*Here funny video about how important a good compression algorithm is from the hit HBO show Silicon Valley:*

[*https://www.youtube.com/watch?v=JlwwVuSUUfc*](https://www.youtube.com/watch?v=JlwwVuSUUfc)

***Direction:*** Use the module gzippo to compress static files in your static folder.

**Minifying and Compiling**

Every static file is its own request to the server to serve its static file. If you reduce this down to one big request, then minify it, it can really save on the number of request your server takes. Use gulp, an automated task performer that can both minify and compile your css into one file that can be served over the request!

Direction:

Npm install gulp

Look at gulpfile.js and add a minify image gulp task.

Run both task by initiating gulp <taskname>

**Cache:**

The less your server has to calculate, the better. Cache can occur in multiple places in your application. If you can cache it, DO IT! CACHE everything!

Direction: Implement cache on your server using node-cache. CACHE everything you possibly can!

***Load Test part 2***

Change the load test to send a 20 request every second. Change it further to have multiple connections (multiple browsers). What is your load test like now?

**Load Balancing**

Just like the analogy of the receptionist and the worker, you can have a server as a receptionist as well whose only job is to dish out request to the workers. These are called load balancers. We will build a simple one now.

Directions:

Deploy the scaling project to three aws servers.

Use two to run servers, and one to run the balancer.

Put the 2 addresses into the load balancer.

It is as easy as that. You now have a load balancer that pushes request out to 2 different servers evenly.

***AutoScaling***

Many webservers/cloud services now allow you to autoscale depending on how heavy your load is. We can implement that ourselves. Whenever our request is going higher than 1000 request/min(60000 request/min), we should spin up a new server in case that happens.

Directins:

Install the amazon command line

Study these 3 API documentation: node.childProcesses, amazon SDK for ec2(runInstance method, waitFor and describeInstances)

Write a shell script with the given commands in the project folder

Implement an auto launching server when request reaches over 1000.