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 | 0.281 sec / 23.188 sec(5015 row(s) returned) | 0.390 sec / 21.922 sec 5909 row(s) returned |  |
| SELECT \* FROM businesses WHERE business\_name = 'Klein Inc' LIMIT 15000 | 1.954 sec / 23.859 sec  1.172 sec / 21.609 sec 1003 row(s) returned | 0.484 sec / 21.094 sec  1.359 sec / 20.984 sec  959 row(s) returned |  |

What were the differences seen? What is the difference between setting something as TEXT or VARCHAR?

**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) | 0.172 sec | 0.703 sec |  |
| *SELECT \* FROM businesses WHERE owner\_first\_name = “else” (indexed)* |  | 0.266 sec / 1.562 sec  1000 row(s) returned |  |
| *SELECT \* FROM addresses WHERE city = 'gussie bury'* | 6.985 sec / 0.000 sec | 0.016 sec / 0.000 sec |  |

**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)* | 26.828 sec / 0.281 sec |  |
| *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* | 242.484 sec / 0.000 sec |  |
| *Fixed Query:* |  |  |
| *Fixed Query:* |  |  |

**Sharding/Partitioning Data:**

Your database may get to big one day and it may be better to separate it into larger pieces. Partitioning is a method of splitting up your database across multiple physical storage. This allows smaller subsets of data.

Direction:

1. Run the query. Divide your database1 horizontally(divide the number of rows) and retest the query.

|  |  |  |  |
| --- | --- | --- | --- |
| SELECT \* FROM businesses WHERE owner\_first\_name = 'else' LIMIT 15000 | 0.281 sec / 23.188 sec(5015 row(s) returned) | 0.390 sec / 21.922 sec 5909 row(s) returned |  |
| SELECT \* FROM businesses WHERE business\_name = 'Klein Inc' LIMIT 15000 | 1.954 sec / 23.859 sec  1.172 sec / 21.609 sec 1003 row(s) returned | 0.484 sec / 21.094 sec  1.359 sec / 20.984 sec  959 row(s) returned |  |

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

***Connecting the front end:***

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* |  |  |
|  |  |  |

To modularize of code.

select \* from businesses

JOIN addresses ON business\_id = businesses.id

GROUP BY city

LIMIT 5000