Contents

Task 0: DbService Script

Summary

The data used for testing can be found HERE.

For testing the databases I decided to use Typescript as it the language I am most comfortable with. There are four key aspects to the script. - CLI Interface - This is responsible for defining the CLI interface for the user to use the script - DataFile - This is responsible for connecting to the file and loading the file with a stream - DbServices - This is responsible for taking the loaded entities and saving them to the database - Processor - This is responsible for batching the limiting the stream before forwarding the entities on to the DbService

I created three DBServices one for each of the database types and a mock DbService for testing the rest of the script.

Usage:

Running with docker

Prerequisites:

- docker
- docker-compose

Run Containers:

```
docker-compose up -d
```

This command will build and run the derby and dbrunner containers and pull and run the MongoDB and mongo client containers. All the containers are deployed to the same network for connectivity between the containers.

```
docker ps
CONTAINER ID   IMAGE                           PORTS                              NAMES
26d042700aa6   mongoclient/mongoclient:latest  3300->3000/tcp                     mongoclient
4721de6f59a2   dbrunner                                                           dbrunner
f72b7e7068fb   mongo:4.2                       27017-27019->27017-27019/tcp       mongo_db
2ca836065aa6   apache-derby                    1527->1527/tcp, 5000->5000/tcp     derby_db
```

Web-Clients are deployed at the following endpoints. - http://localhost:5000 for the apache-derby client - http://localhost:3300 for the mongo client

Run commands:

```
docker exec -it dbrunner bash dbrunner --help
```

Running on system

Prerequisites:

- node ˆ12.0.0
- npm or yarn

You will need to make sure you update the .env file to include your references to the apache-derby client drivers.

1) Install Packages: `yarn install` or `npm install`
2) Build Project: `yarn build` or `npm run build`
3) Running Script: `node ./bin/dbrunner --help`

DBRunner Script CLI

This tool was built for easy interaction with the different DBs. The script includes timings of different stages and options to change to the processing pattern of the source data file. eg. batch size and limit.

```
root@4721de6f59a2:/app# dbrunner --help
dbtester <cmd> [args]

Commands:
  dbtester write  runs loads a CSV file into the database
  dbtester clean  cleans all data from the database
  dbtester query  queries the database for an id

Options:
      --version    Show version number                                 [boolean]
  -h, --help       Show help                                           [boolean]
  -s, --service    service to use for loading
                           [choices: "mock", "mongo", "derby"] [default: "mock"]
  -f, --file       csv file to load the data from        [default: "./data.csv"]
  -l, --limit      The total number of row you want to run 0 == all [default: 0]
  -b, --batchSize  The number of items you want to run in each batch 0 == all
                                                                  [default: 0]
  -q, --query      The id to query in the database              [default: 23413]
  -o, --optimize   Weather or not to optimize the Db Service
                                                [boolean] [default: false]
```

Task 1: Derby ./src/services/derby.ts

To load the data into derby using my script I had to find an npm library that supported JBCD this script will load the select JDBC driver into a tmp JVM and will then use that to interface to. Once I had this setup I just needed to create a DBService for derby. This initialized the connection and created a client for the Processor to use.

By extending the base entity I was able to create Derby specific Entity function that the DerbyDbService needs. These include generating an SQL string for bulk write, write and query using the Entities Id.

```sql
CREATE TABLE TESTING (
    id int,
    dateTime TIMESTAMP,
    year int,
    mDate int,
    month VARCHAR(9),
    day VARCHAR(9),
    sensorId int,
    sensorName VARCHAR(39),
    hourlyCount int
);
```

Test Results:

Write Logs: ./logs/derby.1000.txt

Query Logs: ./logs/derby.query.txt

Write	Read
1123009ms	1505ms

Optimizations

In order to optimize the queries I decided to create a unique index for the id column this would allow people to search for a single record faster. I also added an index to the dateTime column which is standard practice when dealing with time series data. This will allow for faster range searches and filters, for example, how many people traveled in January.

```sql
CREATE TABLE TESTING (
    id int,
    dateTime TIMESTAMP,
    year int,
    mDate int,
    month VARCHAR(9),
    day VARCHAR(9),
    sensorId int,
    sensorName VARCHAR(39),
    hourlyCount int
);
CREATE UNIQUE INDEX index_testing_id on TESTING(id);
CREATE INDEX index_testing_datetime on TESTING(dateTime);
```

NOTE: Unique index could only write about 60'000 before slowing down to an unusable level tried with batch size 100 and 1000 estimated time 7 hours.

Update id index to be non UNIQUE:

```sql
CREATE INDEX index_testing_id on TESTING(id);
```

Test Results:

NOTE: Ran with batch size 100.

Write Logs: ./logs/derby.100.indexed.txt

Query Logs: ./logs/derby.query.indexed.txt | Write | Read | | :——-: | :—: | | 1572915ms | 53ms |

Comparing Indexed VS Non-Indexed

Type	Write	Read	RW 1:1 Ratio	RW 500:1 Ratio	RW 1000:1 Ratio
Indexed	1572915ms	53ms	1572968ms	3743ms	1624ms
Non Indexed	1123009ms	1505ms	1124514ms	3192ms	2625ms
Indexed Diff	+449906ms	-1452ms	+448454ms	+551ms	-1001ms

The RW Ratios the influence the index has depending on how many read vs write requests you are planning to make. From this, we can see that if you plan to make 500 read requests to every write request then it is still not worth including an index. As the average request would be 551ms slower with an index.

Adding Secondary Index

TODO: Writing up of secondary Index

Task 2: MongoDB ./src/services/mongo.ts

MongoDB has better support for typescript than Derby using the package `mongodb` I was able to connect directly to the mongo DB. After creating the `MongoDBService` I was able to run my script using the new service.

As mongoDB is a document store there is no reason to define a structure for the object we only need to set which collection we want to save the objects to.

Test Results:

Write Logs: ./logs/dynamo.1000.txt

Query Logs: ./logs/dynamo.query.txt

Write	Read
75223ms	178ms

Optimizations

To optimize mongo I attempted to create a similar structure to the derby optimizations. Which includes creating an index on the id and an index on the dateTime field. Unlike in derby where the DB could not handle the unique index mongo had no problems. The indexes were setup as follows.

```
// Creates ID Index
collection.createIndex({
        "id": 1 // creates an ascending ordered index on column id
        // NOTE: cannot use hashed index when using unique options  https://docs.mongodb.com/manual/reference/m
    }, {
        unique: true,
        name: `index_testing_id`,
    });

// Creates DateTime Index
collection.createIndex({
        "dateTime": 1 // creates an ascending ordered index on column dateTime
    }, {
        unique: false,
        name: `index_${this.config.tableName}_dateTime`,
    });
```

NOTE: Mongo could not handle a batch size of 1000 when running with the indexes. I had to restart the DB and re-run with a batch size of 100

Test Results:

Write Logs: ./logs/dynamo.100.indexed.txt

Query Logs: ./logs/dynamo.query.indexed.txt

Write	Read
228669ms	29ms

Comparing Indexed VS Non-Indexed

Type	Write	Read	RW 1:1 Ratio	RW 500:1 Ratio	RW 1000:1 Ratio
Indexed	228669ms	29ms	114349ms	485ms	257ms
Non Indexed	75223ms	178ms	37701ms	328ms	253ms
Indexed Diff	+153446ms	-149ms	+76649ms	+158ms	+4ms

In this comparison, we can see that using mongo DB indexes only start to become beneficial after the RW ratio of 1000:1.

Task 3: Java Heap File

Summary

For task 3 we were required to implement a heap file using java. Converting the provided CSV file into binary broken up into records on pages.

Design

In designing the heap I decided to go with fixed lengths for all fields to create a simpler workflow for reading and writing. After writing a script that could scan over all the rows in the CSV file. I was able to reduce the file to find the max byte lengths required for each field. With these values, I was able to create an entity class that defined each column and the required byte length. I then created a serialize and deserialize function in the class. These functions convert the row into binary and convert binary back into the row. Once I had these methods working I could then start on the paging. I need to add the page breaker to the end of each entity so when scanning the algorithm can check if this is the last entity in the page and continue onto the next page.

Testing

I tested reading and writing on several page sizes listed below.

PageSize	512	1024	2048	4096	8192	16384	32768	65536	131072
Pages count	893649	397178	198589	96611	48306	24153	12036	6008	3002
Write ms	75223	18738	17065	17688	***! 15709 !***	16418	16386	18478	18771
Read ms	6914	3223	1745	1198	749	601	440	485	***! 292 !***
RW 1:1 Ratio ms	41069	10981	9405	9443	***! 8229 !***	8510	8413	9482	9532
RW 500:1 Ratio ms	7050	3254	1776	1231	779	633	***! 472 !***	521	639
RW 1000:1 Ratio ms	6982	3238	1760	1214	764	617	456	503	***! 310 !***

From the table above we can see that larger page sizes are better for performance however because of the shallow exploration in the heap implementation there are some advantages for smaller page sizes when it comes to indexing. Having smaller page sizes would allow the index to find the record faster however there would be overhead maintaining the index on write operations. I would expect there to a large difference between the read and write time if an index was implemented. From the test, we have run on the mongo and derby databases.

Adding a B+ Tree Index

To create the B+ Tree I first created some helper class that could let me search through the heap using a `Random Access File` This allowed me to parse a dbLookUp key of `record Id` and `page Id` and it would load that record out of the file rather than scanning though the whole file. Once I had this I needed to start building the tree. I decided to have the keys as some value in from the csv and the values as a class containing the respective `record Id` and `page Id` to the heap file location. Which I could then use to lookup the records from the heap. To create the b tree I have three main classes `bTreeRoot.java`, `dbIndexNode.java`, `dbLeafNode.java` and `dbInnerNode.java`.

New Files

All the new files for this implementation are located under the `./Index`

- Index
  - bTree

        * bTreeDB.java
            - This class extends the dbIndexNodeLoader and contains the functionality for scanning over the whole tree and saving each node to the file. It also is used for some debugging when scanning of the tree.
        * bTreeRoot.java
            - This class is responsible for initializing the bTree and handing actions that are preformed on the tree.
        * bTreeStats.java
            - This class was used for debugging the tree as its hard to tell what exactly is going on sometimes. This object get parsing recursively down through the nodes and each of the nodes then adds itself to the bTreeStats class. Onces all the nodes have been added it is much easer to loop over all the nodes and find ones that are not doing what they are supposed to be doing.
        * dbIndexNode.java
            - Is an abstract class that both the Leaf and Inner Node classes extends. This contains all the common functionality between the two node types.
        * dbIndexNodeLoader.java
            - This abstracts out the fact that there are two heap files exposing a save and read function for each of the node types and sending those requests to the corresponding `dbEntityLoader.java` class
        * dbInnerNode.java
            - These nodes contain children node which link to other dbInnerNods or dbLeafNodes.
        * dbLeafNode.java
            - These nodes contain values which are the lookup keys for the source heap file.
        * TreeNodeType.java
            - This file is an enum that defines the different types of nodes that are included in the b+-tree it also includes function for converting to and from an int for serialization/de-serialization.
    - dbstore
        * dbBytePage.java
            - This class is responsable for storing an array of Idbentities which it can serialize and deserialize. This page's byte size is defined by the number of bytes provided when constructed.
        * dbQntPage.java
            - This class extends `dbBytePage` but is provided an quantity of items you want to store on a page rather than the number of bytes.
        * Idbentity.java
            - This is an interface the defines serialize, deserialize and getSize;
        * Idbkey.java
            - This is an interface the defines getIndex which is used by the `rafdb` to seek that index in the file.
        * IdbStoreable.java
            - This is an interface that extends both `Idbentity` and `Idbkey` which is everything that is required to store an item in the `rafdb`.
        * dbStoreable.java
            - This is an abstract implementation of the IdbStorable interface providing common functions that will normally be common across implementation of the interface.
        * rafdb.java
            - This is a wrapper around a Random Access File that tries to imitate a db client. Having functions like connect and close.
            - The reason I chose to Random Access File was because I new onces I had the index I would be able to jump to the correct byte and start reading the record. Which was perfect for a raf.
    - entity
        * dbEntity.java
            - This is the base class for the dbEntityLoader new entities will be able to extend this class and implement the abstract methods to use functionality of the dbEntityLoader
        * dbEntityKey.java
            - This class contains the `pageId` and `recordId` as well as methods for serializing and deserializing the key.
        * dbEntityLoader.java
            - This class extends the rafdb and abstracts away the bytes[] that you need to use the raf this allows you to use the above class to interface with the rafdb. And includes functions like findEntity using a dbEntityKey and saveEntity using a dbEntity. This class also implements the Iteratable interface returning an iterator that lets you loop over all the dbEntities in the file.
    - index
        *

dbIndexValue.java

    * dbIntIndexKey.java
    * dbStringIndexKey.java
- utils
    * Args.java
    * Cli.java
    * Deserialize.java
    * Serialize.java
-

dbEntityRow.java

-

dbindex.java

-

dbload.java

-

dbquery.java

bTreeRoot This class is responsible for initializing the bTree and handing actions that are preformed on the tree.

dbIndexNode Is an abstract class that both the Leaf and Inner Node classes extends. This contains all the common functionality between the two node types.

dbInnerNode These nodes contain children node which link to other dbInnerNods or dbLeafNodes.

dbLeafNode These nodes contain values which are the lookup keys for the source heap file.

Other Files

TreeNodeType This file is an enum that defines the different types of nodes that are included in the b+-tree it also includes function for converting to and from an int for serialization/de-serialization.

bTreeStats This class was used for debugging the tree as its hard to tell what exactly is going on sometimes. This object get parsing recursively down through the nodes and each of the nodes then adds itself to the bTreeStats class. Onces all the nodes have been added it is much easer to loop over all the nodes and find ones that are not doing what they are supposed to be doing.

db

Serialization/De-Serialization

Once the b+-tree had been filled with the records from the Heap we can then try to save the b tree to its heap file. This will mean we can build the index once and continue to use it in the future. To serialize the b+-Tree I created two Heap files, one for the inner nodes and the other for the leaf nodes. The reason I used two files, was because the Inner nodes and Leaf nodes have different sizes, and having them in separate files means I don't have to pad the smaller records. Another feature that I wanted to implement was to make it so a user of the index would not have to load the whole index into memory to use it. To do this, I changed how the serialization work, rather than each reference to another node being serialized within that node, I would only serialize the heap file reference `record Id` and `page Id`. I would then only need to load nodes from the index when they were required.

Implementing this extra feature proved to be much more difficult than I first anticipated. I ran into many issues when trying to deserialize the heap index. The first issue what the de-serialization itself, it appeared that the offsets were off as I was getting byte-looking characters in some of the strings once they were deserialized. I narrowed down the bug to the de-serialization of the node references the offsets there were not correct after this the index heap could be loaded correctly, at least for the root node. When I tried to run a search on the root node however I received a java heap size exception. I realized the reference nodes that I was creating were all unique objects, so I started to implement a cache in the `dbIndexNodeLoader.java` that would create a new instance if it had not been loaded yet and return the instance if it had already been loaded. I was not able to see this through to completion though as I started to run out of time and I had not made a start on the other parts of the project. So I decided to leave the Index implementation as an in-memory index.

All in Comparison

Now that we have results for all the different DBs let's see how they stack up. We will use each database's best performance per RW Ratio.

X	Java Heap	PageSize	MongoDB	Indexed	Derby	Indexed
RW 1:1 Ratio ms	***! 8229 !***	8192	37701	NO	1124514	NO
RW 500:1 Ratio ms	472	32768	***! 328 !***	NO	3192	NO
RW 1000:1 Ratio ms	310	131072	***! 253 !***	NO	1624	YES

Now that we have all the databases together we can see the winner is *MongoDB*. I did think initial MongoDB was going to win on all fronts but clearly, the JavaHeap has the fastest write times. Though we have to keep in mind that the Java Heap did not have to go over the network will uploading, unlike Mongo and Derby. Further testing would be required for the Java Heap to be on an even playing field.