Overview

This semester was spent working on testing whether using Binary Blobs to write records on an SQLite Database (DB) would both save time and space compared to writing data using FileOutputStream for PAMGuard. I also was responsible for designing and coordinating the Student Poster for the Coding Group. I have been in coding consultation calls with other students, the instances of which might have been mentioned in the reflective logs.

The current data shows that FileOutputStream is significantly faster and smaller in size than the SQLite DB with blobs. This is more likely than not a result of wrong coding for populating the *.dat* files with OutputStream.

This report will discuss specific expectations for this experiment, code design, data analysis and results, challenges faced, and what is left to be improved.

Find code on **GitHub**: dolphin-acoustics-vip/DBAnalytics_ver02

Tasks

- ✓ Maven Java Project on Eclipse (SQLite jar libraries available on GitHub rep).
- ✓ Create table “NormalThings” with column entries: “Index_ID , time_stamp , write_time , random_int , blob”.
- ✓ Fill records into the database and run experiments on datavases of 100, 1000, and 10000 records of different blob sizes, e.g. 100 byte, 1kByte, 10kByte, for binary blobs of both zeros and random numbers.
- ✓ Analyze database for: DB Size in Bytes, Total Write Time (WT), Average WT for each record, WT Approximate Standard Deviation, Minimum WT, Maximum WT, Most Common WT, and WT Increase.
- ✓ Compare the above analysis results with results from updating the same data to a normal FileOutputStream.

How to Run

Open project on Eclipse.

Run in order *InitializeDB* > *PopulateDB* > *DataAnlysis* > *FileStream*. Each process will print out a message indicating completion as shown in *Figures 1, 2, 3*.

```
Database Initialized: Rows-1000_BlobSize-100_BlobType-Random_0.db
```

Figure 1: Example of what InitializeDB prints out at completion.

```
Database Populated: Run 1
                    151,552 Bytes
                    1000 Bytes / Zeros Binary Blob
                    100 Lines
          Write time: 00:00:0.543  (Average 0.005 s)
                                   (Minimum 0.003 s)
                                   (Maximum 0.009 s)
                                   (Most Common 0.005 s -> 56/100 writes)
```

Figure2: Example of what PopulateDB prints out at completion.

```
1 , 1000 , 1000 , Random , 2756608 , 00:00:34.415 , 0.034 , 0.00405 , 0.0 , 0.129 , 0.0 , -2.29E-6
Done with Rows-1000_BlobSize-1000_BlobType-Random.csv!
```

Figure3: Example of what DataAnalysis prints out at completion.

CAUTION: Please do not run *PopulateDB* twice. You must first initialize the database through *InitializeDB* then repopulate it through *PopulatedDB.* Not doing so will produce an unclean database which will obscure data analysis.

NOTE: You will find that some databases (large ones ex: 10,000 records with 10 kByte Blobs) take a few minutes (~3 min) to populate, but the message indicates that it took less (ex: 00:01:20.875). This will be discussed below. The total Write Time (WT) is accurate, but the program goes back to update the table with calculations of specific WTs, which results in twice the time of computing.

This method (1 run automatically generates 900 files from 18 types of files with the use of for-loops) consistently shows WTs which are 3 times as large as generating a single DB file. I am yet unclear as to why this is to get an accurate reading on the total WT please see page 3. You will only be able to do 1 database at a time.

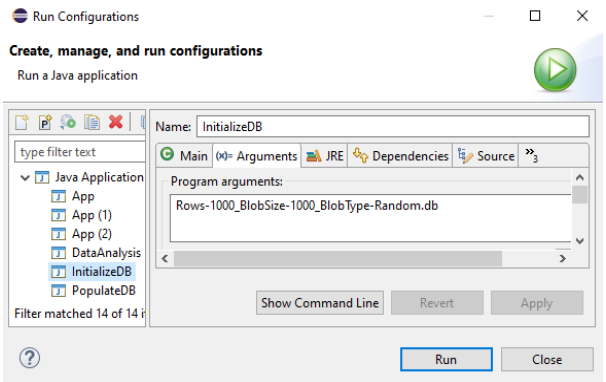

Figure 4: Example of an Arguments Configuration for InitializeDB.

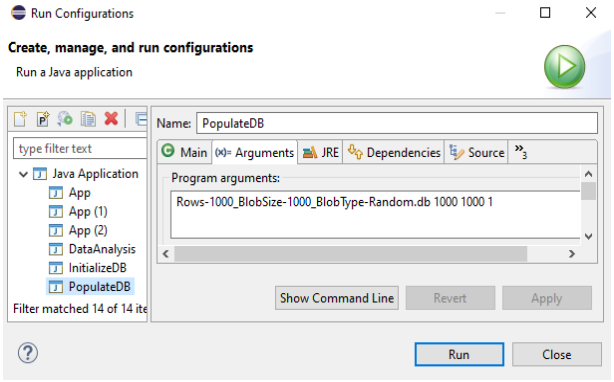

Figure 5: Example of an Arguments Configuration for PopulateDB.

There is an optional directory on **GitHub**: dolphin-acoustics-vip/DBAnalytics_ver01 which allows user input as opposed to automatically generating hundreds of DB files for analysis. This will generate one *.db* file at a time.

To run:
Configure arguments for *InitializeDB.java* to indicate

"<FileName.db>"

, as shown in the example of *Figure 4*, and for *PopulateDB.java*

"<FileName.db> <# of Records> <Binary Blob Size in Bytes> <Binary Blob Type (either 0 (Zeros) or 1 (Random))>"

, as shown in the example of *Figure 5.*

NOTE: This method will produce the fastest WTs. The report will discuss why the method outlined in page 2 might be slowing the program down.

Design

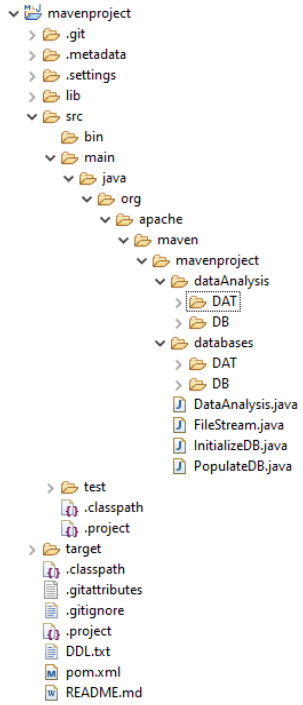

Figure 1: Eclipse Navigator for Maven project located in dolphin-acoustics-vip/DBAnalytics

Beside, in *Figure 1*, is depicted what the maven project directory should look like in eclipse. I only began working with Eclipse for this project, so I was not sure how to clean the Directories well while maintaining the Maven Project Structure.

There are four main java files: *InitializeDB, PopulateDB, FileStream, ec*and *DataAnlysis.* These should be run consecutively, and preferably once as instructed in the previous page. Because of the maven dependencies and directory organization, I could not manage to run the files using the console, instead I made configurations for the arguments as shown in *Figures 1* and *2.* The code has also been commented, so only major design decisions will be discussed in this report.

Note that there are subdirectories *databases* and *dataAnalysis.* One stores all the populated databases with sample names (900 *.db* files):

Rows-100_BlobSize-100_BlobType-Zeros_0.db
Rows-100_BlobSize-1000_BlobType-Random_0.db
Rows-100_BlobSize-10000_BlobType-Zeros_0.db
Rows-1000_BlobSize-100_BlobType-Random_0.db
Rows-1000_BlobSize-1000_BlobType-Zeros_0.db
Rows-1000_BlobSize-10000_BlobType-Random_0.db
Rows-10000_BlobSize-100_BlobType-Zeros_0.db
Rows-10000_BlobSize-1000_BlobType-Random_0.db
Rows-10000_BlobSize-10000_BlobType-Zeros_0.db

The *.db* files were read using DBeaver (https://dbeaver.io/download/) SQLite connection. *Figure 7* and *8* depict what a snippet of the database (with Binary Blobs of Zeros and Random Numbers (1 Kbytes)) looks like. Initially, blobs seemed to be compressed (from DB Size outputs), but the size calculations had been wrong. Upon fixing, it seems that zero blobs are not compressed.

Index_ID	time_stamp	write_time	random_int	blob
1	2022-04-24 17:10:58.095	0.007	7845	
2	2022-04-24 17:10:58.102	0.006	1404	
3	2022-04-24 17:10:58.108	0.005	5875	
4	2022-04-24 17:10:58.113	0.005	9732	
5	2022-04-24 17:10:58.118	0.005	4256	
6	2022-04-24 17:10:58.123	0.006	3340	
7	2022-04-24 17:10:58.129	0.005	5830	
8	2022-04-24 17:10:58.134	0.006	3207	
9	2022-04-24 17:10:58.140	0.005	9213	
10	2022-04-24 17:10:58.145	0.006	1226	

Figure 2: DBeaver view of a DB example for Zeros Binary Blob (1 KBytes).

Index_ID	time_stamp	write_time	random_int	blob
1	2022-04-24 19:20:40.019	0.005	410	û} < ï§Ä1 Á Óäoj ©ÎÃ _
2	2022-04-24 19:20:40.024	0.006	629	¤JR>Øá4N ÕOÑ ×Ü *©ñÙfmÙ
3	2022-04-24 19:20:40.030	0.006	420	z Éð 4U>$= ¦èI ÜI,Ç>
4	2022-04-24 19:20:40.036	0.006	97	âoO d Ê4 Èù Ávæ
5	2022-04-24 19:20:40.042	0.005	920	ì ñ M¹d£c·<±á E× ùÁ *`ðJ
6	2022-04-24 19:20:40.047	0.005	613	Ç/Êbw;ë?H 6 ~Ð§Èò a LSWè
7	2022-04-24 19:20:40.052	0.006	692	Yw VÞ8 ÄÅá< j ²/~¾©.Ê_
8	2022-04-24 19:20:40.058	0.005	592	&Ä2ÎOw8¶A© n ´H7%Ø *¼ qø¦
9	2022-04-24 19:20:40.063	0.007	146	Ø^ön X^þ £yFÄÉ ôjß©ì z
10	2022-04-24 19:20:40.070	0.005	408	¥õã°¶G»xLöC[# yÆ. +l C¥Î

Figure 3: DBeaver view of a DB example for Random Numbers Binary Blob (1 KBytes).

The *dataAnalysis* directory contains *.csv* files of the same names as the *.db* files, which store information on: *DB Size in Bytes, Total WT, Average WT for each record, WT Approximate Standard Deviation, Minimum WT, Maximum WT, Most Common WT, and WT Increase*.

I was able to run the programs using the *databases* directory, however the program is currently not recognizing the directory path. This is very inconvenient as now, when you run the program yourself, you will notice the *../src/* directory fill up with *.db* and *.csv* files. I was not able to fix this in the given time – I still face trouble navigating the maven project directory configurations on eclipse. Because of this inconvenience, I have already provided files in the mentioned subdirectories. When you run the program, for the time being, you will have to wait for all 3 classes to run before collecting the files and separating them manually into the neat directories. I apologize for this inconvenience.

How *PopulateDB* works

The program fills the DBs as shown in *Figures 7 & 8.* Default local timestamp has been used, random integers are selected to occupy the *random_int* column, and binary blobs (either with zeros or random numbers) are generated.

I tried to use a cryptographically secure way to generate the random numbers for the Binary Blob, so I used *SecureRandom().nextBytes(byte[BlobSize])*. Please be ware that despite the name this turned out to be not crypto-safe. A more secure way would have been perhaps *SecureRandom.getInstanceStrong().nextBytes(bytes[BlobSize])*, but I could not get it to work. The SQLite version I am using made it difficult to forcefully compress blobs before inserting them onto the DB.

After the DB has been populated with records in the specified number of records, blob types, and blob sizes, the program reiterates through the DB to calculate each record's WT. There was no automatic way to do this in SQLite, so I used the default timestamps to mark the beginning of the record writing and used the timestamp on the next record to calculate the time between records which is approximately the WT of the previous record. I could not find a way to record the true WT, but this estimate should be very close, if not indistinguishable, to the true WT. This process would always leave the last row of the DB with a WT of zero (without a next line to get a timestamp from), so I programmed it to write an additional line and then delete it at the end of the WT updating process.

The process of iterating through the DB for a second time in order to update the WT for each record takes approximately the same amount of time as the first iteration. That is why, when the process takes ~1 min to complete, the total WT of the DB would have been only ~0.5 min.

PopulateDB contains a lot of private methods which are duplicated in *DataAnalysis*. These methods are not required in the record filling process; however, they are useful to construct the informative output message which is shown in *Figure 2.* Without this message, there would be no duplication of methods in the two java files.

DBAnalytics_ver02 uses 4 for-loops to populate 900 *.db* files. As a result, the first ~5 DB writings happen within the expected time (usually 0.5 s for DB of 50 records), at which point the total WT starts to double and triple. I tried investigating why, and this problem persists even when using 1 for-loop. When using no automation in DBAnalytics_ver01 (initializing and populating one DB at a time) this problem disappears. Nonetheless, I took ~9 hours to make sure to collect data with non-delays in order to perform an analysis.

How *DataAnalysis* works

The program creates *.csv* files for each *.db* file available. The intent of these files is to contain however many runs (ex: 50) of DBs with the same configurations (say 100 records and 1Kbyte random binary blobs) by enabling Compilation and Run commands for *InitializeDB* and *PopulateDB* from directly from *DataAnalysis*. The generated *.csv* files would be then analyzed for consistencies or discrepancies. Once again, I faced problems with path navigation in the eclipse maven project (I maintain this view because I have successfully implemented such a program in a previous project with similar code from what can be seen in https://www.journaldev.com/937/compile-run-java-program-another-java-program). I am willing to continue working on this problem post-examination period, if I am allowed, otherwise it will be a problem for a future student.

The class currently reads and analyzes 50 files of the DB with the same length, blob type, and blob size, and updates 50 entries to one of the corresponding 18 *.csv* files.

The program then queries the DB for information such as total WT, average WT, etc… I will focus on the logic used to calculate DBSize (Bytes), WT Increase (s), and WT Standard Deviation (s), since these were not readily available functions in SQLite.

DB Size (Bytes) had no built-in function in SQLite. I took advantage of a StackOverflow question (Podehl) which multiplies *pragma_page_count()* and *pragma_page_count()* to determine the DB Size in Bytes. I would have preferred to get a free-space reading as well, but I could not find a more accurate version than this. I would urge the reader to please verify that this method produces a reliable reading on the DB Size.

```
SELECT SUM((x - x_bar) * (y - y_bar)) / SUM((x - x_bar) * (x - x_bar)) as Slope
    FROM (
        SELECT [ID] AS x,
        AVG([ID]) AS x_bar,
        [MovingAverage] AS y,
        AVG([MovingAverage]) AS y_bar
        FROM (
            SELECT [Index_ID] AS ID,
            [time_stamp],
            [write_time],
            AVG([write_time]) OVER (
                ORDER BY [time_stamp] ROWS BETWEEN 2 PRECEDING AND 2 FOLLOWING
                ) MovingAverage FROM [NormalThings]
            ) MovingAverageTable
        )
```

Figure 4: SQLite Query to estimate the slope of the moving average of WT, to determine Linear Regression in WT as number of writes increases.

WT Increase was a complicated calculation to make (see *Figure 9*). After calculating the WT Moving Average, the query takes *ID* and *MovingAverage* as x and y axis of an imaginary scatter plot. The slope (see *Figure 10*) of the fitting line of that plot is approximated as usual (see top of *Figure 9*).

```
Slope              |
-------------------|
0.000001330363697|
```

Figure 5: WT Linear Regression (Increase) Example.

WT Standard Deviation was approximated by taking the square root of the sample variance, which can be calculated using: $s^2 = \frac{\Sigma (X - \bar{x})^2}{n - 1}$ (Bhandari). The standard deviation is the square root of the sum of an exact value minus the mean of the sample squared, divided by one less than the sample size. In SQLite this command is translated to:

```
SELECT ( SUM([write_time]*[write_time])-
         (SUM([write_time])*SUM([write_time]))/(COUNT(Index_ID)-
1) )
       /((COUNT(Index_ID)-1)-1)
FROM NormalThings
```

I did not use `COUNT(Index_ID)` to get the number of records on the DB because that takes a long time once the DB Length becomes 1000, 10000, 100000, etc... Instead, I kept track of the length of the DB within the method. In other cases I would order the DB by ID in descending order, and took the first ID entry to determine how many records were on the DB. `COUNT()` is of linear time complexity, while the two alternatives mentioned are in constant time.

Data Analysis

The *Table 1* below shows analysis data taken for 18 types of *.db* files, averaged over 50 runs each**.** If DB size has been calculated correctly, the results show that DB sizes remain consistent for

DB Rows	Blob Size (Bytes)	Blob Type	DB Size (Bytes)	Tot WT	Avrg WT (s)	WT S. Dev (s)	Min WT (s)	Max WT (s)	Most Common WT (s)	WT Increase
100	100	Random	32,768	00:00:0.629	0.006	0.001	0.004	0.012	0.006	1.38E-05
100	100	Zeros	32,768	00:00:0.617	0.006	0.002	0.004	0.015	0.006	1.13E-05
100	1,000	Random	151,552	00:00:0.509	0.005	0.001	0.004	0.013	0.005	-1.09E-05
100	1,000	Zeros	151,552	00:00:0.642	0.006	0.002	0.004	0.014	0.006	1.61E-05
100	10,000	Random	1,056,768	00:00:0.542	0.005	0.001	0.003	0.011	0.006	-1.77E-05
100	10,000	Zeros	1,056,768	00:00:0.536	0.005	0.001	0.003	0.009	0.005	1.60E-06
1,000	100	Random	172,032	00:00:6.325	0.006	0.007	0.003	0.166	0.005	1.33E-06
1,000	100	Zeros	172,032	00:00:6.296	0.006	0.005	0.003	0.161	0.005	1.27E-06
1,000	1,000	Random	1,380,352	00:00:6.447	0.006	0.010	0.003	0.324	0.005	-4.31E-07
1,000	1,000	Zeros	1,380,352	00:00:6.412	0.006	0.011	0.003	0.335	0.005	-4.94E-07
1,000	10,000	Random	10,272,768	00:00:6.238	0.006	0.007	0.004	0.181	0.005	-1.81E-07
1,000	10,000	Zeros	10,272,768	00:00:6.187	0.006	0.002	0.003	0.017	0.005	1.72E-06
10,000	100	Random	1,601,536	00:1:5.433	0.007	0.011	0.003	0.447	0.006	1.09E-07
10,000	100	Zeros	1,601,536	00:1:5.127	0.007	0.012	0.003	0.432	0.005	3.69E-07
10,000	1,000	Random	13,701,120	00:1:5.638	0.007	0.011	0.003	0.354	0.005	4.63E-08
10,000	1,000	Zeros	13,701,120	00:1:3.253	0.006	0.011	0.003	0.373	0.005	2.65E-07
10,000	10,000	Random	102,477,824	00:1:1.934	0.006	0.011	0.003	0.380	0.005	-9.46E-08
10,000	10,000	Zeros	102,477,824	00:1:8.65	0.007	0.011	0.003	0.434	0.006	1.73E-07

Table 1: Average (for 50 runs) of details about each type of DB produced in this model: Records 100-10,000, Blob Sizes 100-10 Kbytes, Blob types of zeros and randoms (uncompressed).

either blobs of zeros or random numbers. The zero blobs are not automatically compressed (although at earlier stages, because of an inconsistency with the DB Size calculations it seemed they were. The DB writings (excluding blobs) seem to contribute 2.5-7 MB to the DB. I was not able to find a way to check for free space size.

Average Write Times for individual records, regardless of DB or Blob Size or type, range from 5-7 ms. Notice that Minimal WTs also are consistently between 3-4 ms. The total WTs also remain consistent regardless of blob size or type. It is only as number of records increase that total WT is affected. For larger number of records, the chances of WTs that increase from 10 to 400 ms occurring increases from 0% to nearly 0.5%. Although these max WTs seem to be more concentrated towards the end of the table, they are spread out and don't occur at regular intervals.

The standard deviation from the mean WT value increases from 1ms to 12 ms as number of records increases. There is virtually no increase in the WTs as more records are added, or if the blob sizes are larger (see last column of *Table 1*).

In conclusion, as calculated from the averages of 50 runs for each type of the 18 DB outlined above, the WT for a single record remains 5-7 ms regardless of number of records on the table, size of the blob, or type. The longer the table stays connected it seems WTs of up to ~400 ms are more likely to occur (0.5%), which affects the standard deviation by 10 ms as the number of records increases. Overall, there is no upwards or downwards trend when it comes to WTs of individual records, and zero Blobs are not compressed.

When comparing the DB blob results to the simple FileOutputStream, the results for the output stream seem much faster and less space consuming, as seen in *Table 2*. This is likely the result of writing the wrong type of data to the Output stream. I was not able to write binary blobs, so I tried to mimic the effect with multiple bytes. Still the results for each normal File output stream seem instantaneous compared to the DBs. I believe this should not be the case, so after this submission I will try to fix the Output Stream code.

Run	DB Rows	Blob Size (Bytes)	Blob Type	DB Size (Bytes)	Total Write Time (millis)
1	100	100	Zeros	29,112	0
2	100	100	Random	29,112	0
3	100	1,000	Zeros	209,112	0
4	100	1,000	Random	209,112	0
5	100	10,000	Zeros	2,009,112	2
6	100	10,000	Random	2,009,112	1
7	1,000	100	Zeros	290,112	2
8	1,000	100	Random	290,112	0
9	1,000	1,000	Zeros	2,090,112	0
10	1,000	1,000	Random	2,090,112	3
11	1,000	10,000	Zeros	20,090,112	0
12	1,000	10,000	Random	20,090,112	0
13	10,000	100	Zeros	2,900,112	1
14	10,000	100	Random	2,900,112	0
15	10,000	1,000	Zeros	20,900,112	2
16	10,000	1,000	Random	20,900,112	0
17	10,000	10,000	Zeros	200,900,112	1
18	10,000	10,000	Random	200,900,112	1

Table 2: Sample of results form FileOutputStream.

A sample of the print out from the file Output stream *.dat* files is shown below in *Figure 11.*

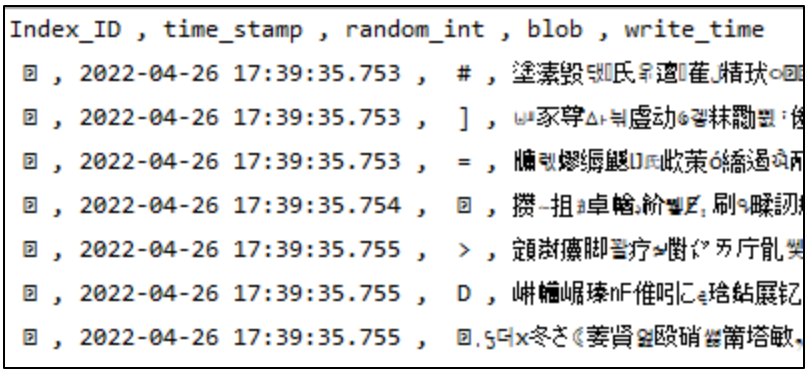

Figure 6: Output from FileOutputStream

TODO

1. Fix FileOutputStream code to show true equivalence to SQLite DB tests.
2. Fix directory paths so that all created files are deposited into respective comprehensible directories and not simply dumped into the main directory.
3. Consider the free space left in all of the DB for better analysis of the space used.
4. Improve implementation of automating DB populating for 1000+ files so that for-loops do not interfere in the writing time of the DBs.

References

Bhandari, Pritha. “Understanding and Calculating Variance.” *Scribbr*, 12 Oct. 2020, https://www.scribbr.com/statistics/variance/.

“How to Compile and Run Java Program from Another Java Program.” *JournalDev*, 17 Mar. 2019, https://www.journaldev.com/937/compile-run-java-program-another-java-program.

Podehl, Axel. “How to Get the Current Sqlite Datbase Size of an Attached Database.” Stack Overflow, 22 Jan. 2020, https://stackoverflow.com/questions/59857556/how-to-get-the-current-sqlite-datbase-size-of-an-attached-database.