Anh Quang Chu 015332629
Brian Tran 010647299

Experiment 1

Transcript for Part 1 of Experiment 1

```
sqlite> CREATE TABLE foo (col1 int);
sqlite> INSERT INTO foo (col1) VALUES (1), (2), (3), (4), (5), (6), (7), (8), (9), (10), (11), (12),
(13), (14), (15), (16), (17), (18), (19), (20), (21), (22), (23), (24), (25), (26), (27), (28), (29), (30),
(31), (32), (33), (34), (35), (36), (37), (38), (39), (40), (41), (42), (43), (44), (45), (46), (47), (48),
(49), (50);
sqlite> PRAGMA journal_mode=delete;
delete
```

Made a copy called DB1.sqlite

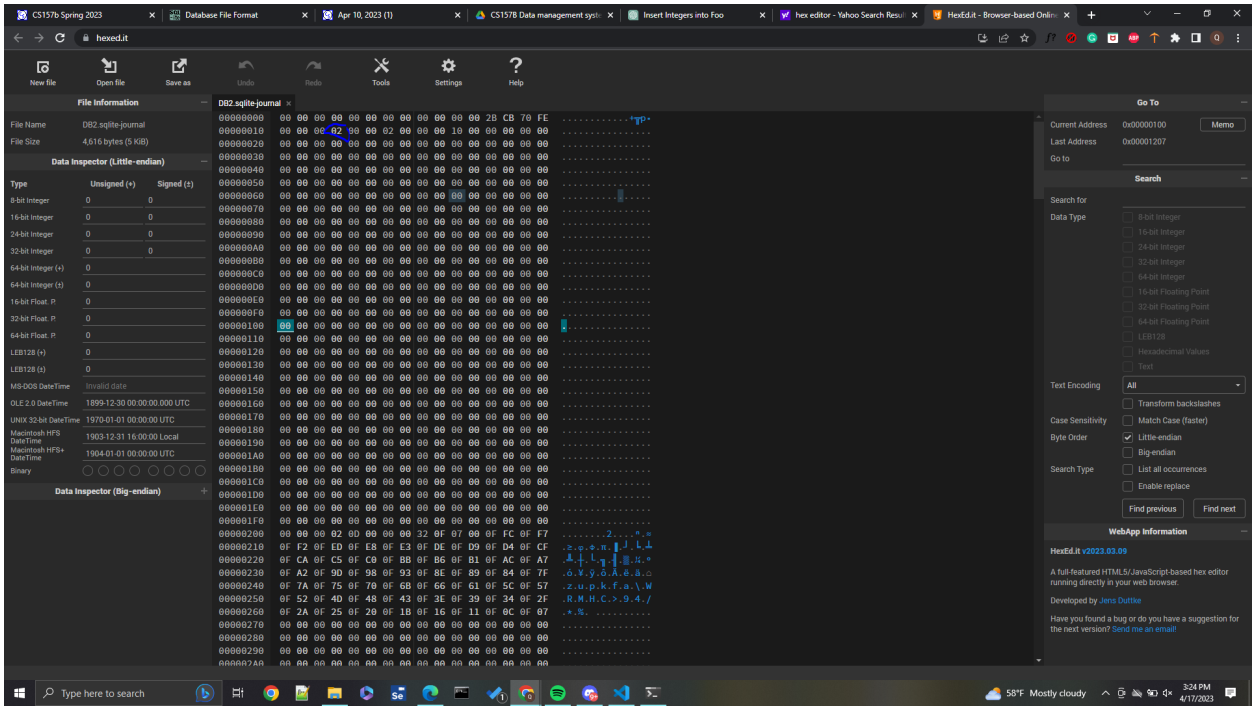
```
sqlite> BEGIN TRANSACTION;
sqlite> DELETE FROM foo;
```

Made copy called DB2.sqlite

```
sqlite> INSERT INTO foo (col1) VALUES (51), (52), (53), (54), (55);
sqlite> END TRANSACTION;
```

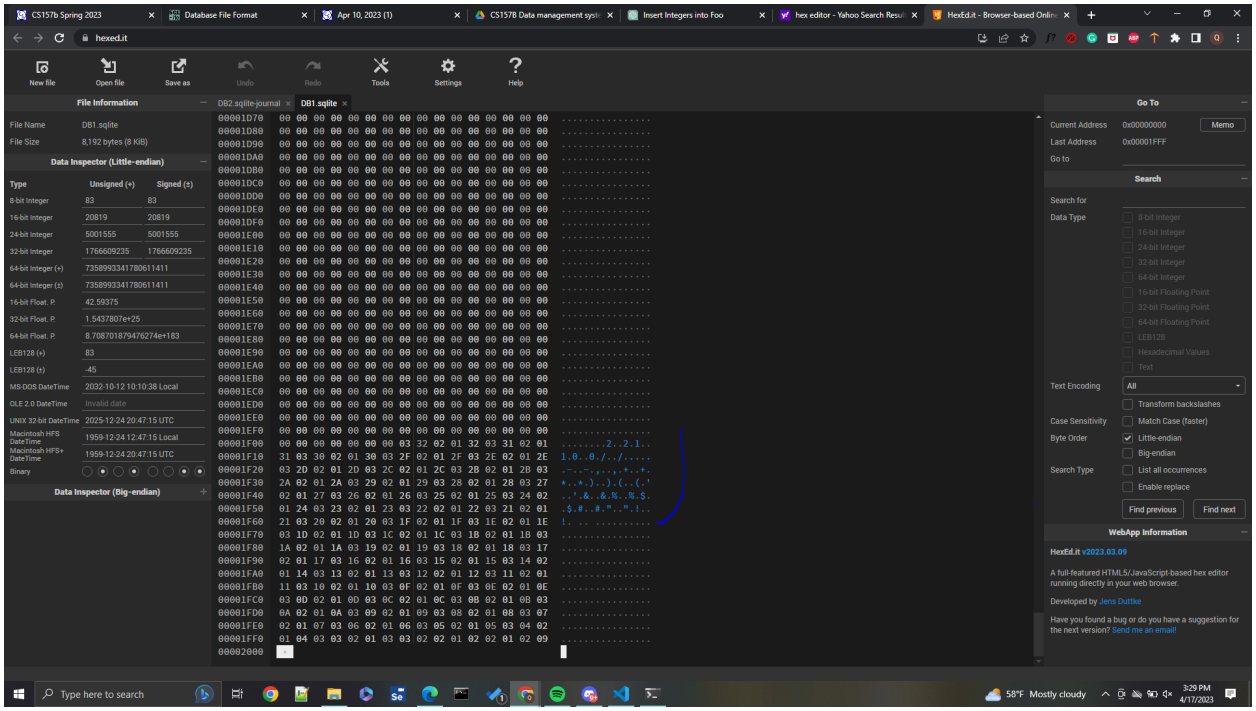
Redoing experiment again but with ROLLBACK

```
sqlite> CREATE TABLE foo (col1 int);
sqlite> INSERT INTO foo (col1) VALUES (1), (2), (3), (4), (5), (6), (7), (8), (9), (10), (11), (12),
(13), (14), (15), (16), (17), (18), (19), (20), (21), (22), (23), (24), (25), (26), (27), (28), (29), (30),
(31), (32), (33), (34), (35), (36), (37), (38), (39), (40), (41), (42), (43), (44), (45), (46), (47), (48),
(49), (50);
sqlite> PRAGMA journal_mode=delete;
delete
sqlite> BEGIN TRANSACTION;
sqlite> DELETE FROM foo;
sqlite> INSERT INTO foo (col1) VALUES (51), (52), (53), (54), (55);
sqlite> ROLLBACK;
sqlite> END TRANSACTION;
Runtime error: cannot commit - no transaction is active
sqlite>
```



Based on the content in the hex editor, the page number is found after the first 8 bytes. Then the page count can be find within the 4 bytes after that. So we found the page number at the 12th byte from the beginning of the journal file.

Compared to DB1.sqlite in hex editor, we found the similar section in the DB1 sqilite journal as follow:



After inserting 5 more entries, the journal file did not change

```
sqlite> INSERT INTO foo (col1) VALUES (51), (52), (53), (54), (55);
```

This is because the transaction is still going on and not ended yet so there is no change to the journal file.

```
sqlite> END TRANSACTION;
```

After ending the transaction, the journal is deleted

When we redid the experiment with ROLLBACK, the journal file was deleted when ROLLBACK was called since the changes done during the transaction were undone.

Transcript for Part 2 of Experiment 1

sqlite> CREATE TABLE foo (col1 int);
sqlite> INSERT INTO foo (col1) VALUES (1), (2), (3), (4), (5), (6), (7), (8), (9), (10), (11), (12), (13), (14), (15), (16), (17), (18), (19), (20), (21), (22), (23), (24), (25), (26), (27), (28), (29), (30), (31), (32), (33), (34), (35), (36), (37), (38), (39), (40), (41), (42), (43), (44), (45), (46), (47), (48), (49), (50);
sqlite> PRAGMA journal_mode=WAL;
wal

Make a copy called DB3.sqlite

sqlite> BEGIN TRANSACTION;
sqlite> DELETE FROM foo;

Make a copy called DB4.sqlite

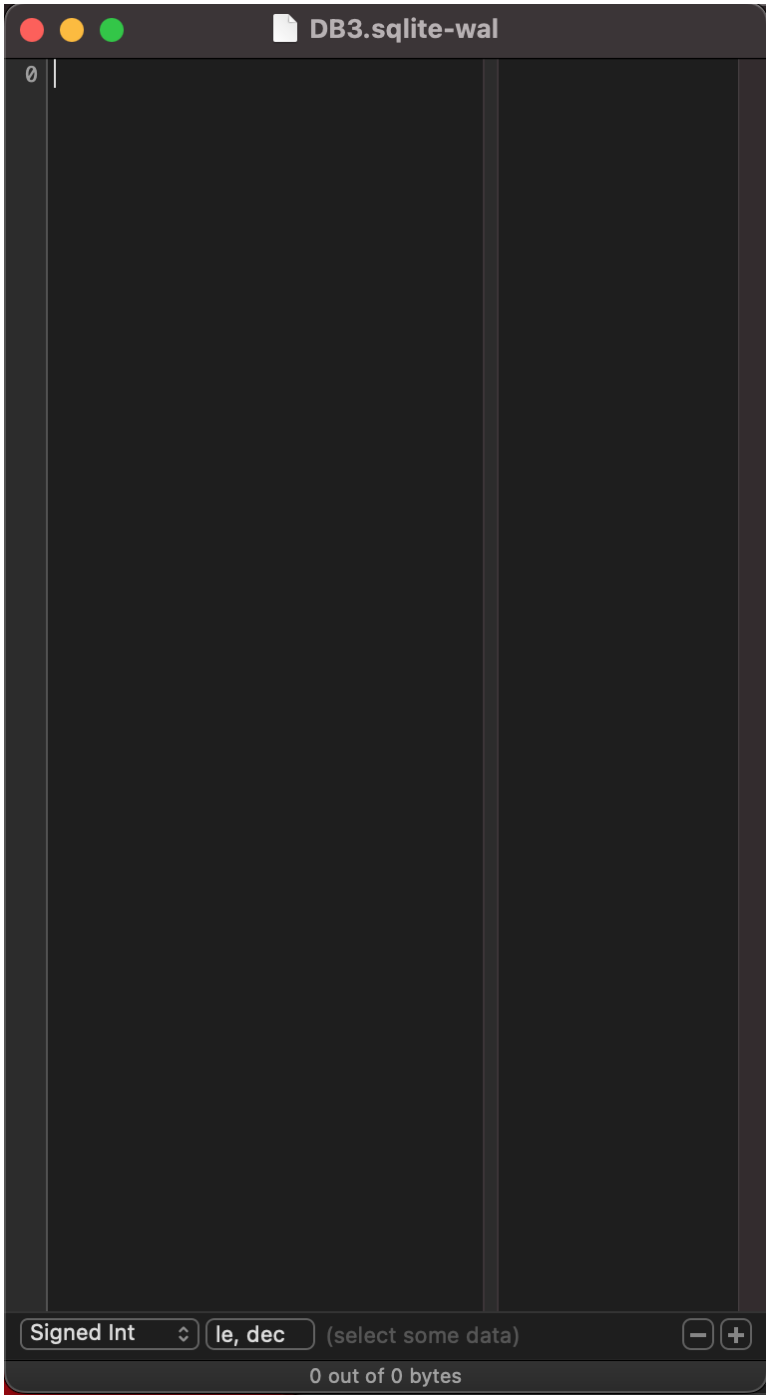
sqlite> INSERT INTO foo (col1) VALUES (51), (52), (53), (54), (55);
sqlite> END TRANSACTION;

Redoing the experiment but with ROLLBACK

sqlite> CREATE TABLE foo (col1 int);
sqlite> INSERT INTO foo (col1) VALUES (1), (2), (3), (4), (5), (6), (7), (8), (9), (10), (11), (12), (13), (14), (15), (16), (17), (18), (19), (20), (21), (22), (23), (24), (25), (26), (27), (28), (29), (30), (31), (32), (33), (34), (35), (36), (37), (38), (39), (40), (41), (42), (43), (44), (45), (46), (47), (48), (49), (50);
sqlite> PRAGMA journal_mode=WAL;
wal
sqlite> BEGIN TRANSACTION;
sqlite> DELETE FROM foo;
sqlite> INSERT INTO foo (col1) VALUES (51), (52), (53), (54), (55);
sqlite> ROLLBACK;

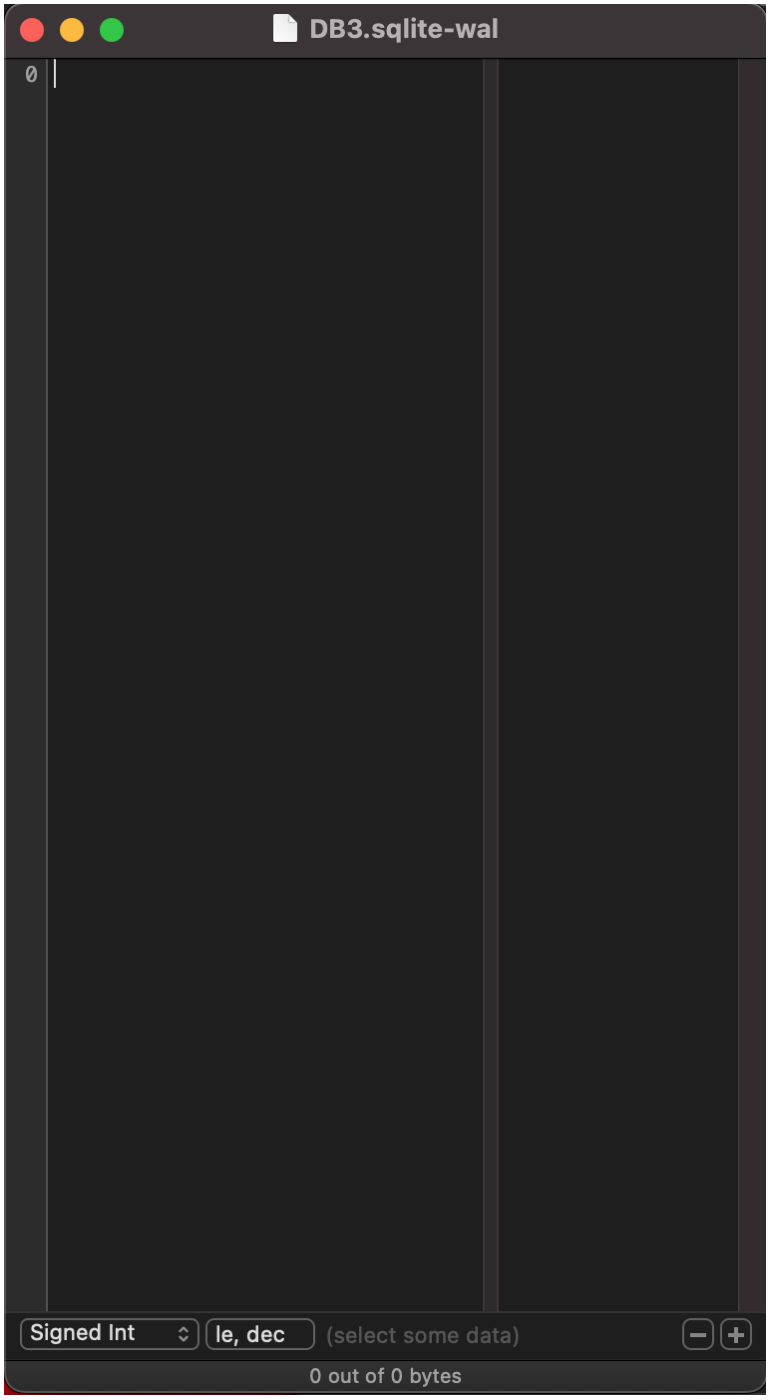
```
sqlite> END TRANSACTION;
Runtime error: cannot commit - no transaction is active
sqlite>
```

With the new mode WAL, the journal in the WAL file can be viewed as followed



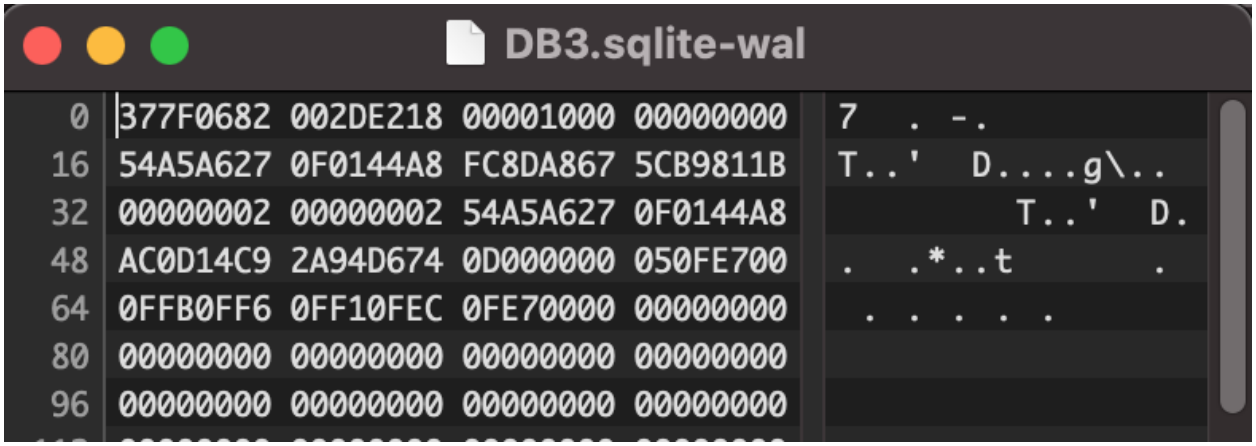
As observation, the wal file is empty (ask prof about this weird behavior for confirmation)

After the insertion of 5 more entries the wal file is still empty

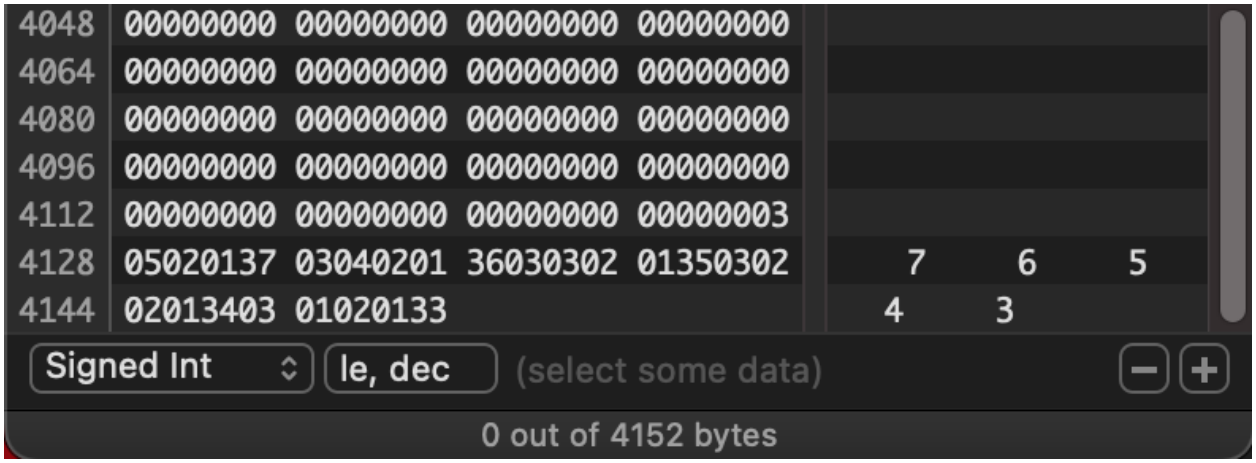


After we end the transaction, the wal file has some changes:

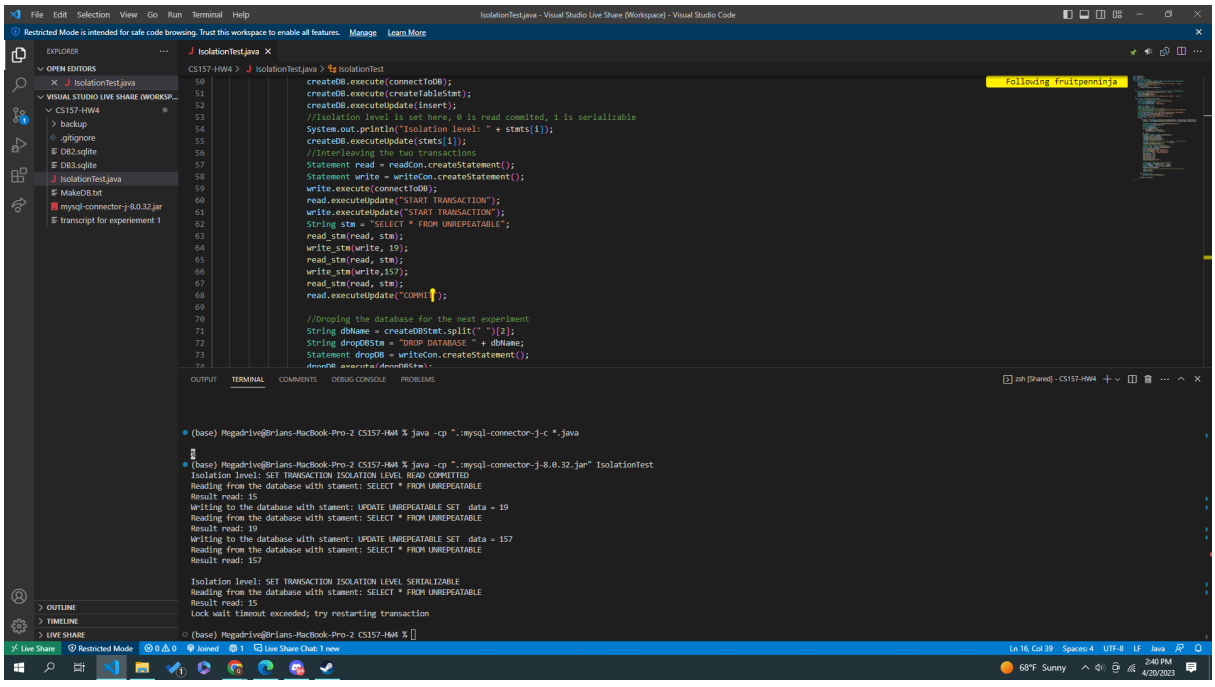
1. First change is in the beginning of the file



2. Second change is at the very end of the file



Experiment 2



For reading committed transaction type which is the default setting of the database. The transaction can see update of the snapshot state of database (affected by other transaction) after it runs. In this case the select query on reading transaction can only see the state of the database the moment it started running.

As for serializable isolation level, these are a type of transaction isolation level used in database management systems to ensure that concurrent transactions do not interfere with each other and maintain consistency of data. When multiple transactions are running concurrently and accessing the same data, the database system uses locks to prevent conflicts and ensure the proper execution order of transactions.

In a serializable isolation level, the database system uses strict locking mechanisms to prevent concurrent transactions from accessing the same data simultaneously. This means that when a transaction acquires a lock on a certain piece of data, no other transaction can acquire a conflicting lock on the same data until the first transaction has completed. This is done to ensure that transactions are executed one at a time, sequentially, as if they were executed in a serial order, hence the name "serializable."

However, this strict locking mechanism can lead to a phenomenon known as "serialization anomalies" or "serialization failures." If multiple transactions are waiting for locks on conflicting data items, and the database system cannot resolve the conflicts, it may result in a deadlock situation where none of the transactions can proceed. This can lead to timeouts, where a transaction is unable to complete within a certain time limit set by the database system or the application, resulting in an error or rollback of the transaction.