Introduction to Database System - Assignment 5 Report

Team 22 - 108020017 黃珮綺, 108020021 賴柏翰

Phase 1 Report

1. Implementation Details

PrimaryKey

We use a primary key to uniquely identify records in a table. Each `PrimaryKey` instance contains the table name that the key belongs to and a map that maps the field name to the corresponding key value. We define the hash code of the primary key to be the additives of the hash codes of the table name and the key values (multiplied by 31). It will be used by the lock table to tell if the two `PrimaryKeys` indicate the same record.

ConservativeConcurrencyMgr

We have moved the `ConservativeConcurrencyMgr` to the same directory as `ConcurrencyMgr` to use its feature directly. we define a new method `getLocks()` that acquires locks on the read/write keys. Nonetheless, we leave the inherited method, except `onTxCommit()` and `onTxRollback()`, empty since we think that the record should be protected by locks on its primary key. Hence, there is no need for additional locks on the blocks and the files. At last, the concurrency manager will hold all the locks until the transaction commits or rolled back.

TransactionMgr

Given the isolation level, the transaction manager will create a new transaction with the corresponding concurrency manager. There are already 4 types of concurrency managers, say serial concurrency manager (isolation level = 8), repeatable-read concurrency manager (isolation level = 4), read-committed concurrency manager (isolation level = 2), and read uncommitted concurrency manager (isolation level = 1). We add an alternative that will run the conservative concurrency manager (isolation level = 16).

StoredProcedure

In `StoredProcedure`, we modified the function `prepare()` to use different isolation levels. We define the default value of the isolation level in `vanilladb.properties`.

After the connection is established, the stored procedure will prepare the parameters and ask the transaction manager to create a new transaction with the default isolation level. The stored procedure begins to execute queries and will then return the result set if the transaction finishes normally.

MircoTxnProc

We override the function `prepare()` of `StoredProcedure` to accommodate conservative locking. We collect the read/write objects in advance, then create the transaction, and get locks of these objects. The new transaction will be locked by a global lock until it gets the locks of its read/write objects. On the other hand, if the isolation level is not at the conservative locking level, the transaction manager will create a new transaction just as the `StoredProcedure` does.

Here, the procedure reads/writes the records by their id. Also, it is clear that the ids serve as the primary key of the item table. Thus, in methods `getReadKeys()` and `getWriteKeys()`, we construct each primary key with the key entry map that only maps one field `i_id` to the id. If there are two or more features in a primary key, the key entry map should map more fields.

The way we implement conservative locking can make sure that the transactions with lower `txNum`s acquire locks before transactions with higher `txNum`s as a result of the global lock. However, it is fairly slow because the transactions are able to be created only after the previous transaction gets the locks of its read/write objects, which reduces concurrencies. We have come up with a solution to this, that is, implementing a queue-version lock table in the concurrency manager and letting the transaction request the lock right after it was created (in the global lock) and get the lock later (outside the global lock). Yet, this is not included in this assignment.

2. Challenges of Implementation for TPC-C Benchmark

In contrast to the microbenchmark, the TPC-C benchmark includes not only `SELECT` and `UPDATE` SQL commands but also `INSERT` and `DELETE` SQL commands. To implement conservative concurrency on the TPC-C benchmark, we need to obtain the lock in advance for these two additional commands. For `INSERT`, we need to locate the empty slot for the insert record, and for `DELETE`, we must know which record was deleted previously. Any of these are complicated to be implemented because of the difficulty to target the read/write sets of a given transaction.

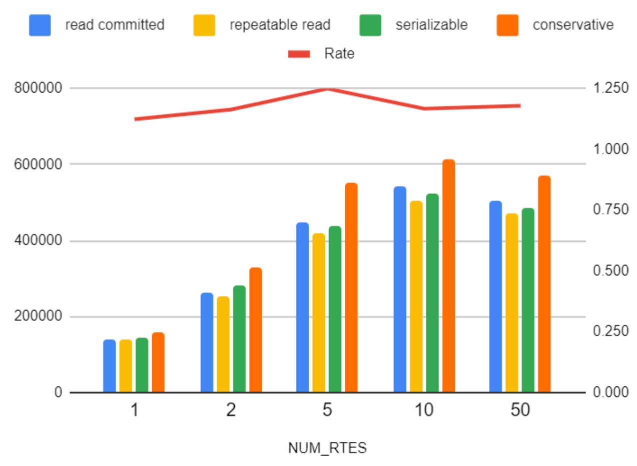
3. Experiments

We perform some experiments on the performance of our implementations. The following will show the plot of the result. The raw data will be provided in the Appendix.

Environment

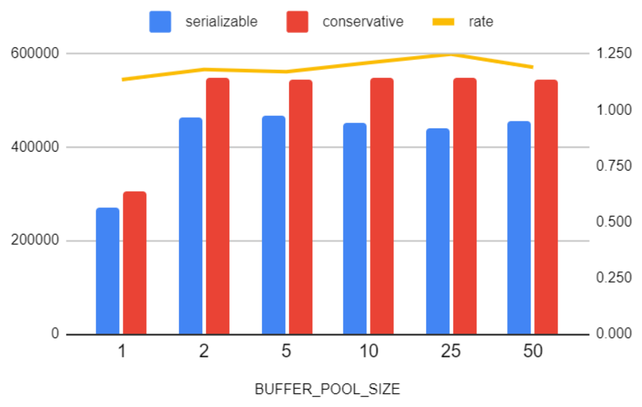
`Intel Core i5-8300H CPU @2.30GHz, Windows 11`

Improvement



> In general, the throughput increases as the number of RTEs increases, and there is a slight decrease in throughput when the number of RTEs is larger than 50. We take a closer look at the throughput of each isolation level. Conservative concurrency has the most throughput, followed by read-committed and serializable concurrency; repeatable-read concurrency has the least throughput. Also, we compare the throughput of serializable and conservative concurrency. The ratio increases as the number of RTEs increases, and reaches its maximum of 1.25 at RTEs of 5. When the number of RTEs is larger than 10, the ratio falls back to around 1.6. We will perform the experiment with RTEs of 5 in the next section.
>
> It is worth mentioning that deadlocks often occur in serial concurrency, repeatable read-concurrency, and read-committed concurrency. Especially when the number of RTE increases, the number of aborted transactions increases significantly. However, we didn’t detect deadlocks during the experiments of conservative concurrency. This is because that conservative locking ensures the former transactions acquire locks before executing later transactions, there should not be deadlocks.

Impact of Buffer Pool Size



> When the size of the buffer pool is very limited, say as large as the block size (=4,096), the throughput is greatly affected. It is almost half of the throughput of the buffer pool of size 102,400. This possibly results from the limited blocks for each transaction to pin. Yet, we didn't find obvious changes in throughput by the buffer pool size when it is larger than 4,096.
>
> For the improvement of conservative concurrency, the rate reaches its maximum at a buffer pool size of 102,400, and its minimum at a buffer pool size of 4,096.

4. References

Source Code

Appendix