

Bachelor Practical Course

Implementation: Multiversion Concurrency Control Mechanism of Hekaton Database Engine

Tuan Anh Ma

*Student of Business Information System
Department of Informatics
Technical University of Munich
Email: anh.ma@tum.de*

Advisor: Dr. Viktor Leis

*Chair III Datenbank Systems
Department of Informatics
Technical University of Munich
Email: leis@in.tum.de*

Abstract—Multiversion Concurrency Control (MVCC) is a lock-free concurrency control mechanism which is implemented in database management systems (DBMS) to provide high transaction rates to scale well in demanding conditions and high contention. Taking advantage of this useful property, many in-memory DBMS have applied this mechanism to optimize their transactional processing performance. In order to understand more about MVCC, this paper presents a simple implementation of MVCC mechanism simulating the Microsoft's new database engine 'Hekaton' as well as some experimental results to show its overheads compared to a normal DBMS that is implemented without concurrency control.

1. Introduction

In order to simulate the MVCC mechanism employed in Hekaton as in the paper of Cristian Diaconu, et al. [2], we write a simple DBMS in C++ programming language to implement the TPC-C schema. This system has its tables accessed only via primary key (PK) index resembling Hekaton tables. Furthermore, the transactions of this system are applied all the rules, mechanisms and ideas of MVCC from the paper of Per-Åke Larson et al. [1]. At the end of this report, we will show some experimental results which indicate the overheads of MVCC transaction processing.

2. Interfaces, Classes and Program Files

2.1. Interfaces

- Interface '*Attribute*': This interface defines the meta-data of each column in a table such as name, type, ... Each column in a table is considered as an *Attribute* object.
- Interface '*Table*': This interface stores the meta-data of each table such as table name, number of PK, ... Each table is a *Table* object.
- Interface '*Version*': This interface defines the *begin timestamp*, *end timestamp*, *garbage indicator*, ... for each version within a specific table.

2.2. Classes (Structures)

- Class '*{table-name}*': Each table in this TPC-C schema is also an object with its own name. These objects inherit from interface '*Table*' and declares the table indexes, which are implemented as C++ *unordered_multimap*. It also has a function *import()* to import the data from *.tbl files into the table.
- Class '*Tuple*': In the MVCC mechanism, each tuple is also considered as a version, therefore, class '*Tuple*' inherits from the interface '*Version*'. Every class '*{table name}*' will have its own class '*Tuple*' as its member class and the member variables of class '*Tuple*' correspond to the columns of the table.
- Class '*Transaction*': This is the most important class of the program. This class represents a transaction object and has many member variables and functions to support the MVCC mechanism. We discuss this class in more detail in section 5.
- Class '*{SQL-types}*': These classes consist of class '*Integer*', '*Numeric*', '*Timestamp*', '*Char*' and '*Varchar*' to convert between C++ primitive types and SQL variable types.
- Other supporting classes: There are other classes in program such as class '*Predicate*' which serve for the comprehensible purpose and the conciseness of the program.

2.3. Program Files

- The interfaces '*Attribute*', '*Table*' and class '*Transaction*' are written in files *.hpp, *.cpp of the same name.
- The files Schema.hpp and Schema.cpp declare and define class '*Version*', '*Tuple*' as well as the '*{table-name}*' classes.
- The remaining '*{SQL-type}*' classes reside in files Types.hpp and Types.cpp.
- The transaction objects and the threads will be created in file *main.cpp*

3. Data Storing and Indexing

3.1. Data Storage

As mentioned in section 1, our program employs the TPC-C schema for testing the MVCC mechanism. The TPC-C schema consists of 9 tables with each of them has their own class and inherits from the interface 'Table'. Inside every table classes, we define its own 'Tuple' class. The member variables of these 'Tuple' classes correspond to the columns of the table. Moreover, all these classes have their own constructors and destructors. In the constructors, the member functions *import()* will be called to store the data from the *.tbl files *directly* to the PK indexes, which are implemented as C++ 'unordered_multimap'. Therefore, scanning through a table will be a random access to the records.

3.2. Indexes

Apart from the table 'History', which does not have any index, each table has either a single-columned PK or a composite PK. In summary, the schema has four types of PK: <Integer>, <Integer, Integer>, <Integer, Integer, Integer> and <Integer, Integer, Integer, Integer>. To build the PK index, we take advantage of the C++ container *unordered_multimap* with its key is a C++ *tuple* and its value stores the real data of a record. We cannot employ the container *unordered_map* for PK because a record in a table may have many versions which certainly have the same PK. Figure 1 from the paper of Per-Åke Larson et al. [1] depicts the PK index of a table. An example of the table 'NewOrder' in TPC-C schema, which has its composite PK consisting of 3 Integers is illustrated in Listing 1:

```
struct NewOrder : public Table{
    struct Tuple : public Version {
        Integer no_o_id, no_d_id, no_w_id; //Pkey
        Tuple(Integer o_id, Integer d_id, Integer w_id):
            no_o_id(o_id), no_d_id(d_id), no_w_id(w_id){ }
    };
    unordered_multimap<tup_3Int, NewOrder::Tuple>
        pk_index;
    NewOrder():name("neworder"){
        tables.push_back(*this); import();
    }
    ~NewOrder(){};
    void import();
};
```

Listing 1: Declaration of table 'NewOrder'

4. Versioning and Version Visibility

4.1. Versioning

As indicated in Listing 1, a tuple in the MVCC mechanism is also a version. The interface 'Version' provides each class 'Tuple' with the 'begin' timestamp, the 'end' timestamp and a flag variable 'isGarbage'

to indicate whether or not the version is still a valid version (see "Figure 1: Example account table with one hash index." from the paper of Per-Åke Larson et al. [1] for the illustration of a record format in MVCC). Timestamps are implemented as *uint64_t* global variable and is monotonically increased. Because the 'begin' and 'end' fields of a version can also contain transaction-ID, we take advantage of the most significant bit (the 64th bit) to distinguish between timestamps and transaction-IDs. In this manner, we implement the following components:

- A monotonically increasing counter 'GMI_cnt': This counter is a global variable and increasingly generates timestamps in the range from 2^0 to $(2^{63} - 1)$ for the versions as well as for the Transaction objects.
- A constant global variable 'INF': This variable represents the infinity timestamp for the versions: *const uint64_t INF = (1ull<<63);*
- A constant global variable 'NOT_SET': This variable defines the null (or the not-set) value of the 'begin' and 'end' fields in a version. It has the value 2^{63} .
- A monotonically increasing counter 'GMI_tid': This is a global variable and increasingly generates transaction-IDs in the range from $(2^{63} + 1)$ to $(2^{64} - 1)$ for the Transaction objects.

At the beginning when a version is created, its 'begin' and 'end' timestamps will be set to 'INF'. After it received its payload (by the function *import()*), its 'begin' field will be assigned to a proper timestamp retrieved from the 'GMI_cnt'. Listing 2 represents an example for the class 'Version'.

```
struct Version {
    uint64_t begin = 0;
    uint64_t end = 0;
    bool isGarbage = false;
    void setTime(uint64_t begin, uint64_t end){
        this->begin = begin;
        this->end = end;
    }
    Version(){
        begin = INF;
        end = INF;
    }
    virtual ~Version(){};
};
```

Listing 2: The interface 'Version'

4.2. Version visibility

Our implementation strictly follows the concept of version visibility and the process of checking the visibility of a version in section 2.5 from the paper of Per-Åke Larson et al. [1]. All the rules for version visibility are implemented in class 'Transaction', which we will discuss in more detail in section 5.

5. Transaction Object

5.1. Member Variables

- `"const uint64_t Tid"`: This is the transaction-ID, which is retrieved in the constructor from the global increasing counter `'GMI_tid'`.
- `"const uint64_t begin"`: This variable stores the 'begin' timestamp of a transaction, we also apply it as the logical read time when the transaction object reads a version. It will be initiated in the constructor by calling the global increasing counter `'GMI_cnt'`.
- `"uint64_t end"`: This variable stores the 'end' timestamp of a transaction. It will be initiated in the constructor with the constant variable `'NOT_SET'`. When the transaction successfully commits, this variable will be assigned to a timestamp retrieved from the global increasing counter `'GMI_cnt'`.
- `"int CommitDepCounter"`: This variable counts how many unresolved commit dependencies this transaction is waiting for. This counter is assigned to the value 0 by default and will be increased if this transaction *speculatively* reads, *speculatively* ignores or *speculatively* updates a version. A transaction cannot commit until its `CommitDepCounter` equals 0.
- `"bool AbortNow"`: In the context of commit dependencies, other transactions can set this variable to *true* to order this transaction to abort.
- `"int CODE"`: The purpose of this variable is to support the testing process as the exit code for a transaction. 'CODE' can have the following values to indicate how the transaction ended, where values from -9 to -12 are to indicate the aborts due to errors of the query statements:
 - 1: The transaction commits successfully.
 - -1: The transaction has aborted because the version-visibility validation failed
 - -2: The transaction has aborted because the at least one phantom was detected.
 - -19: The transaction has aborted due to a cascaded abort by commit dependencies.
 - -9: Reading an invalid (invisible or a non-existing) version.
 - -10: Updating an invalid version.
 - -11: Deleting an invalid version.
 - -12: Inserting a version that has the duplicate PK with a visible version in the table.
- `"State state"`: The type `'State'` of this variable is defined as an *enum class*, which has the following values *Active*, *Preparing*, *Committed*, *Aborted*. Therefore, the variable `'state'` represents the current state of a transaction. A transaction is *'Active'* if it is processing its content (the query statements). A transaction is in *'Preparing'* state if it has precommitted (it finished processing its content and has retrieved an 'end' timestamp). States *'Committed'* and *'Aborted'* are self-explained.

Furthermore, the class `'Transaction'` has four types of set. A Transaction object stores the `'Tid'` of the transactions

that have a commit dependency on it (transactions that are waiting for this transaction to commit to be allowed to commit.) into the `vector<uint64_t> CommitDepSet`. If a transaction must abort, it refers to this `CommitDepSet` and sets the `AbortNow` variable of all transactions in this set to *true*. In addition, as specified by Per-Åke Larson et al. [1], a Transaction object also has three other sets (`ReadSet`, `WriteSet` and `ScanSet`) to store the necessary information for the validation process. The `ReadSet` stores only the pointers of versions queried by `SELECT`-statements, while the `WriteSet` stores the pointers of versions required by `INSERT`-, `UPDATE`- and `DELETE`-statements. Every element in the `WriteSet` has its first component points to the old version and its second component points to the new version. For insert operations, the first component is always *'nullptr'* and for delete operations, the second component is always *'nullptr'*. Because it will be complicated to generalize the `ScanSet`, we create a `ScanSet` for every table scanned and resort to `C++vectors` to store the information. Listing 3 shows the `ReadSet`, the `WriteSet` and two `ScanSets` for the tables `'Item'` and `'Stock'`.

```
vector<Version*> ReadSet;
vector<pair<Version*, Version*>> WriteSet;

vector<pair<Item_PK*, Integer*>> ScanSet_Item;
vector<pair<Stock_PK*, tup_2Int*>> ScanSet_Stock;
```

Listing 3: `ReadSet`, `WriteSet` and `ScanSets`

5.2. Member Functions

Because our program currently supports only the PK index, the predicate parameters of the member functions in an Transaction object only indicate the PK.

- `bool checkVisibility(Version&)`: This function receives a version as its parameter and apply the case analyses described in section 2.5 of the paper of Per-Åke Larson et al. [1] to determine if the received version is visible to the transaction. This function returns *true* if the version is visible, otherwise it returns *false*.
- `bool checkUpdatibility(Version&)`: This function receives a version as its parameter and apply the procedure described in section 2.6 of the paper of Per-Åke Larson et al. [1] to determine if the transaction is allowed to update the received version. This function returns *true* if the version is updatable, otherwise it returns *false*.
- `void execute(int)`: The content (query statements) of a transaction is implemented in this function. The constructor of a transaction object will call this function to begin the normal processing phase with the parameter of this function is the ID of the thread executing the transaction.
- `Version* read(string, Predicate, bool)`: This function is responsible for looking up a specific version for the transaction such as for the `SELECT`-statements. There are three parameters that are passed to this functions. The first parameter is the name of the table that the required

version resides in. The second parameter is the predicate (the PK) of the version. Because other member functions such as *update()* and *remove()* also resort to this function to read a particular version, the last parameter indicates if this function is called directly from function *execute()* (by a SELECT-statement) or from another function (by an UPDATE- or a DELETE-statement). Function *read()* will scan the table specified by the first parameter with the predicate specified by the second parameter to search for the required version. The index and the predicate are then stored in the appropriate ScanSet. Every version matching the predicate will be passed to function *checkVisibility()* to determine its visibility. Once the required version is found, this function will apply the third parameter to decide whether or not this version will be stored into the ReadSet and then returns the pointer to the version. If the required version is not found, this function returns the *'nullptr'*.

- *Version* update(string, Predicate)*: This function is employed to perform the UPDATE-statements. Its parameters store the name of the queried table and a predicate respectively to search for the required version. A version is updatable if and only if it is visible. Therefore, this function resorts to function *read()* to search for the version and to determine the visibility of that version. If the version is not visible, this function returns *'nullptr'*, otherwise it receives a pointer to that version and calls the function *checkUpdatibility()* to inspect if the version can be updated. If the version is not updatable, this function returns *'nullptr'*, otherwise it changes the *'end'* field of the required version to the transaction-ID of the performing transaction to make this version invisible to other transactions. This function then creates a copy of the version found as a new version and set the *'begin'* and *'end'* fields of this copy to the transaction-ID and the value *'INF'* respectively. Finally, the addresses of the old and the new versions will be stored into the WriteSet and the pointer to the new version will be returned for further change in the payload by the calling function.
- *Version* insert(string, Version*, Predicate)*: This function is employed to perform the INSERT-statements. It has the first two parameters resembling the function *updated()* but the third parameter describes the version that will be inserted. First, this function calls the function *read()* to inspect if there is any visible version with the same predicate in the table. If there is such a version, the transaction is not allowed to insert the new version and therefore, this function returns the *'nullptr'*. If it receives a *'nullptr'* from the function *read()*, this function stores the new version into the table and indexes it. Finally, the *'begin'* and *'end'* field of the new inserted version will be set to the transaction-ID and the value *'INF'* respectively and an appropriate new entry will be added into the WriteSet.
- *Version* remove(string, Predicate)*: This function is responsible for the DELETE-statements. It performs in the same manner as the function *update()* except it does not create a new version and the second component of the entry added into the WriteSet is a *'nullptr'*.

- *void precommit()*: After a transaction finished executing all its query statements, it calls this function to change the state to *'Preparing'* as well as to retrieve a valid end timestamp from the counter *'GMT_cnt'*.
- *int validate()*: The purpose of this function is to ensure *"read stability"* and *"phantom avoidance"* for the transaction before it is allowed to commit. This function strictly follow the concept of "Figure 3: Possible validation outcomes" in the paper of Per-Åke Larson et al. [1]. In this manner, the validation consists of two steps: verifying if the versions in the ReadSet still visible and checking for phantom versions. In order to verify the version visibility, a transaction compares the end timestamps of every version in its ReadSet to its *'end'* field. If the end timestamp of a version in the ReadSet is less than the end timestamp of the transaction, then that version becomes invisible and this function returns the value -1 and the transaction must abort with exit code -1. Once all versions in the ReadSet are qualified, the phantom avoidance step begins by scanning every ScanSet of the transaction. If there is any phantom version appears, the transaction will receive the exit code -2 from this function and must abort with this exit code. This function returns the value 1 if all the tests pass.
- *void abort(int)*: A transaction calls this function with the exit code as the function parameter if it must abort. In this function, the state of a transaction is changed to *'Aborted'*, the variable *'CODE'* is set to the value of the parameter and the WriteSet is scanned to perform the rollback.
- *void commit()*: This function is only called at the end of a transaction to change its state to *'Committed'* if that transaction successfully performs its content and passed all the validation steps.

5.3. Constructor

Each transaction possesses only one constructor. The constructor of a transaction acquires the *'Tid'* for that transaction from the counter *'GMI_tid'* and initiates the *'begin'* field with a timestamp retrieved from the counter *'GMI_cnt'* as well as assigns the *'end'* field of that transaction to the value *'NOT_SET'*. Furthermore, the *'state'* of a transaction will be set to *'Active'* and the current transaction object is added to a global vector *'TransactionManager'*, which manages all the running transactions in the system. Finally, the constructor calls the function *execute()* to begin the transactional processing.

6. Correctness Testing

We conducted three simple tests for *visibility validation*, *phantom detection* and *"cascaded aborts"* to verify the correctness of the program. The tests for visibility validation and phantom detection follows the rules depicted in the "Figure 3: Possible validation outcomes" in the paper of Per-Åke Larson et al. [1]. Two threads are required for each test and the tests are run on table *'WareHouse'* due to its small size in order to easily observe the changes occurring in the table. Note that truly concurrency is very difficult to

achieve, so that our tests will let the threads sleep and run in turn. We call the first created thread T1 and the second one T2. As soon as T2 is created, it sleeps for one second to let T1 execute first.

For the test of visibility validation, T1 read a specific tuple and then sleeps for 2 seconds. T2 then deletes this version and commits. T1 wakes up and has to abort with the code -1 (the program from the Github branch 'test-visibility-validation' performs this test). In order to test the phantom detection, we let T1 read a tuple and precommit. Before T1 performs the validation steps, it sleeps for 2 seconds. T2 then performs an update on the tuple that was read by T1 and commits. T1 wakes up and has to abort with the code -2 because of the new version created by the update of T2 (the program from the Github branch 'test-phantom-avoidance' performs this test). In the test for cascaded abort in the context of commit dependency, T1 first updates a tuple, precommits and then sleeps for 2 seconds before performing the validation steps. T2 then read a tuple with the same predicate as the tuple updated by T1. Thus, T2 will *speculatively* read the tuple which is newly create by T1 because T1 is in the 'Preparing' state and has received an end timestamp. In this context, T2 has to wait for T1 to commit in a loop and checks for the commit from T1 every 0.5 seconds. Then, T1 wakes up and aborts itself as well as orders T2 to abort because the *Tid* of T2 is in T1's *CommitDepSet* since the speculative read. Finally, T2 has to abort with the exit code -19 (the program from the Github branch 'test-cascaded-aborts' performs this test).

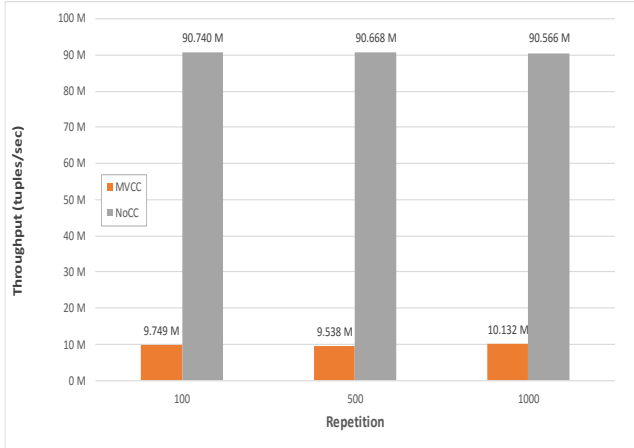


Figure 1: Scan performance of MVCC and NoCC

7. Experimental Results

The experiments were run on an Intel Core i5-2520M CPU @ 2.50 GHz machine that has a single CPU with two cores (four threads). The system has 8 GB of memory, 3 MB L3 cache , 256 KB L2 cache per core and runs on operating system Xubuntu 16.04 LTS. In order to estimate the overheads of the MVCC mechanism, we implement another simple database system which does

not apply any concurrency control mechanism and also bases on the TPC-C schema. We call this database system as NoCC (no currency control) for short. Each table in NoCC stores its data on an array and indexes them by an '*unordered_map*'. Because the storage structure is different between MVCC and NoCC, we first show the throughput of scanning the biggest table in the TPC-C schema, the table '*OrderLine*' with 1425564 tuples, 100 times, 500 times and 1000 times with a single thread. Figure 1 shows the scanning speed of these two mechanisms. It is very obvious that NoCC with sequentially access has the overwhelming throughput compared to the ramdomly access of MVCC. The throughputs between the repetitions are almost constant.

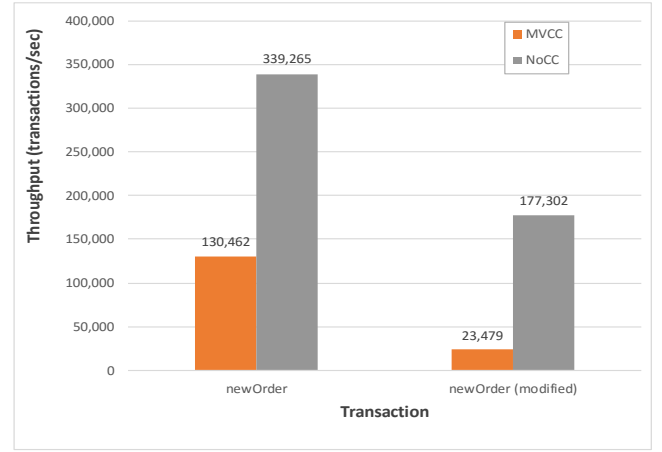


Figure 2: Throughput by performing the transaction 'newOrder'

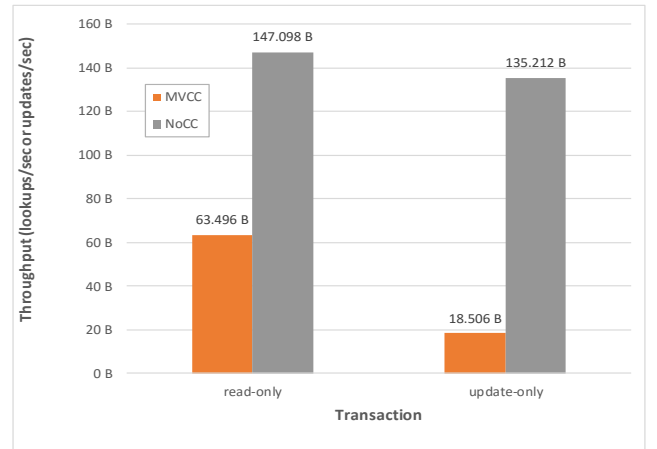


Figure 3: Throughput by performing of read-only and update-only transactions

Next, we perform two other performance tests to see how fast the MVCC and NoCC approaches execute the read-only and update-only transactions. We write a read-only transaction with only one random lookup on the table '*OrderLine*' and a update-only transaction with only one random update also on the table '*OrderLine*'.

These two transactions will be repeated 1,000,000 times with a single thread. Figure 3 depicts the throughputs in billions (B) of transactions per second. The reason for the performance difference between MVCC and NoCC in this test is that the MVCC mechanism has to carry the overhead for all the rules and the validations to ensure the proper concurrency control. This is also the case for the next experiment by running the transaction *'newOrder'* of the TPC-C benchmark 1,000,000 times with a single thread. Because a transaction must abort if it inserts a record with duplicate PK into a table, we noticed that most transactions abort due to the INSERT-statement into the table *'Order'* and *'NewOrder'* before the second for-loop is executed. Therefore, we slightly modified the transaction *'newOrder'* by placing these two INSERT-statements after the second for-loop, at the end of the transaction. Figure 2 illustrates the throughput of both the original and the modified transaction *'newOrder'*.

8. Conclusion

The MVCC mechanism must carry some overhead compared to the NoCC but it still provides an acceptable performance for the transactions. Our implementation still has a limitation that it cannot perform the transactions truly concurrently but it is able to perform the versioning and the validation properly to guarantee the correct concurrency control. This leaves a further improvement for this implementation.

Acknowledgments

The author would like to thank Dr. Viktor Leis for his advices and recommendations during the implementation.

References

- [1] Larson, Per-Åke, et al. *"High-performance concurrency control mechanisms for main-memory databases."* Proceedings of the VLDB Endowment 5.4 (2011): 298-309.
- [2] Diaconu, Cristian, et al. *"Hekaton: SQL server's memory-optimized OLTP engine."* Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data. ACM, 2013.