Assignment 2 - DB2

Daniel Brand

Andreas Kostecka

1. RELATIONS AND TRANSACTIONS

1.1 Implementing relations

Points to discuss (not limited to):

- data structures chosen to store the relations
  - **flights**

    ```
    HashSet<Integer>
    ```

    The *HashSet* guarantees the uniqueness of flight IDs while providing a fast access due to hashing.
  - **seats**

    ```
    TreeMap<Integer, HashSet<Integer>>
    ```

    Since in this relation we have a mapping from a seat ID to a flight ID, the *TreeMap* will be used, whose access times are fast. Furthermore, each seat ID can occur on many flights, whose ID is unique, so a seat ID will be mapped to a *HashSet* of flight IDs.
  - **passengers**

    ```
    HashSet<Integer>
    ```

    For the passengers a *HashSet* will be used for the same reasons as for flights above.
  - **reservations**

    ```
    TreeMap<Integer,
         HashSet<TreeMap<Integer, Integer>>>
    ```

    First of all a unique passenger may book a seat on several flights, so once again a *TreeMap* is used for the mappings while a *HashSet* stores the booked seats in flights. Another *TreeMap* is used for the simple mapping of seat to flight ID. The reasons for this design decision is the same as stated above.
- cardinality of the relations
  In our simulated database there is a one-to-many relationship between passengers and reservations but only a one-to-one relationship between passengers and flights. So this means a passenger can have more than one reservation but only on different flights. It is not possible that a passenger can book multiple seats for one flight.
  Furthermore we have a one-to-many relationship between one flight and seats.

1.2 Implementing transactions

Points to discuss (not limited to):

- **access methods of the transaction to the relations (read and write operations):**
  In our design a transaction has no direct access to the database items. We implement already a simple class *ConcurrencyControlManager.java* which provides access to our simulated database. For this assignment the class provides only simple access for getting and inserting data. In the further assignments we are going to implement also the complete locking in this class with the whole data structures needed.
  We provide the following read/write operations via the ConcurrencyManager:
  - **getAllFlights**

    ```
    Return: HashSet<Integer>
    ```

    Returns all the unique flight IDs.
  - **getAllPassengers**

    ```
    Return: HashSet<Integer>
    ```

    Returns all the unique passenger IDs.
  - **getAllReservations**

    ```
    Return: TreeMap<Integer,
         HashSet<TreeMap<Integer, Integer>>>
    ```

    Returns all flights booked by all passengers.
  - **getSeatsPerFlightId**

    ```
    Parameter values: Integer flightId

    Return: HashSet<Integer>
    ```

    Returns all seat IDs to the corresponding flight ID.
  - **deleteReservationByFlightAndPassengerId**

    ```
    Parameter values: TreeMap<Integer, Integer>
    reservation, Integer passengerId
    ```

    Deletes a reservation from a passenger.
  - **insertReservationByPassengerId**

    ```
    Parameter values: Integer passengerId,
    TreeMap<Integer, Integer> entry

    Return: boolean
    ```

    Inserts a reservation for a passenger.
  - **getReservationsByPassengerId**

    ```
    Parameter values: Integer passengerId

    Return: HashSet<TreeMap<Integer, Integer>>
    ```

Returns all flight reservations from a passenger.

Note: The operations may change in further assignments!

- **properties of each transaction (operations to make, relations to access, interference with other transactions):**
  - **book-Transaction**
    This transaction books a reservation for a given seat on a flight. It invokes another transaction which checks first if the passenger has already booked a seat on the corresponding flight.
  - **cancel-Transaction**
    This transaction cancels a reservation from a passenger. First this transaction calls another for checking if the reservation to delete exists. Then the transaction invokes another transaction for gaining all reservations from the passenger. Last the transaction calls the ConcurrencyControlMechanism-class to delete the reservation.
  - **myFlights-Transaction**
    This transaction returns all the flight IDs where the passenger holds a reservation. Invokes a transaction from the ConcurrencyControlMechanism-class for getting all flights from a passenger.
  - **totalReservations-Transaction**
    This transaction returns the sum of all reservations on all flights. It calls our implemented "SUM"-function in the ConcurrencyControlMechanism-class.

  Note: The properties may change in further assignments!

- **distribution of the transactions which you consider for experimenting:**
  We want to do 2 variations of tests, one being deterministic and the other one will be randomized. The deterministic tests, i.e. tests whose results we know and can expect, will be used for immediate debugging while the randomized tests will do some pre defined amount of transactions with a random choice of transaction type (one of the 4 possible transactions). The deterministic ones will have a smaller amount of transactions whose types will also be pre defined, so we are talking about an uniform distribution for them.

2. SERIAL SCHEDULES

2.1 Changelog

Our previous implementation was messy and made scheduling hard. So we refactored our program in its entirety. A transaction is now an object with several properties. An agent creates a new *DBTransaction* object and calls the now single public method *exec* in *ConcurrencyControlManager.java*. In the manager based on the transaction type the actual transactions will be done. By moving everything to here the locking should be become easier. The methods are still the same as documented in part 1.

2.2 Points to discuss (not limited to)

- **chosen mechanism to ensure serial schedules, especially in case of multiple threads:**
  All locking and scheduling happens in *ConcurrencyControlManager.java*. To ensure a serial schedule, we created the 2 methods
  - private static synchronized void lockDatabase(int myId)
  - private static void unlockDatabase(int myId)

  which operate on the global lock *databaseLock* of type *Lock*. A description of *Lock* is given in the next section. By adding the *synchronized* keyword we ensure no second thread enters that method, so at any time only one thread even could lock the database. The function *unlockDatabase* should not be synchronized, since no thread will ever move to that code until the first thread who locked the database unlocks it. Another mechanism is used in those aforementioned 2 methods. In every function a synchronized block surrounded by an infinite loop is located in the following form

```
while(true) {
  synchronized(databaseLock) {
    if(!databaseLock.open) continue;
    ...
    break;
  }
}
```

  The unlock method does not have the if - statement, so that this thread will actually unlock the database. By further synchronizing on the *databaseLock* we avoid race conditions between the first and second thread, the first thread trying to unlock while the second tries to lock. The infinite loop has to be there to ensure that the second thread, i.e. the second transaction, stays the second thread. The two above methods are called (by wrapping in the two methods stated below) at the beginning and the end of a transaction respectively.

- **implementation details of locking**
  For the locking part we introduced the 2 methods
  - lockManagementGrowing(DBTransaction transaction)
  - lockManagementShrinking(DBTransaction transaction)

  and the static subclass *Lock* of our manager. The subclass *Lock* has the following fields
  - LockType type (either LOCK-S or LOCK-X)
  - boolean open (open == true means open lock)
  - HashSet<Integer> agentIDs (will be important later on)

  The type of a lock was not important for this part but will surely be later on for 2PL. The *open* field is the important part. The usage of agentIDs is already simulated in database locking but was not essential. A global variable called *serial* defines which scheduling scheme will be used.

3. 2PL AND CONCURRENCY

3.1 Changelog

- **Data-Structures**
  We simplified the data-structure of our reservations from

  ```
  TreeMap<Integer,
       HashSet<TreeMap<Integer, Integer>>>
  ```

  to

  ```
  TreeMap<Integer, TreeMap<Integer, Integer>>
  TreeMap<PASSENGER-ID,
       TreeMap<FLIGHT-ID, SEAT-ID>>
  ```

- **Choosing a free seat**
  We return now the list of all possible free seats of a flight to the passenger. The passenger chooses then randomly a free seat from the list for booking.

- **Test-Cases**
  Our test-cases are now more realistic. The transaction occurrences have the following order *book* >*cancel* >*myFlights* >*totalReservations*. Furthermore we split the manager class into 3 sub classes. In the class *ConcurrencyControlManager* we kept all about locking and scheduling, the class *DBAccess* has all direct database access methods with a corresponding SQL statement as a comment above them to indicate, what we actually do in those methods. The third class is called *DBConnection* and is now our interface to the outside, i.e. the method **exec** is now in there, together with the transactions and calls to DBAccess and ConcurrencyControlManager.

3.2 Points to discuss (not limited to)

- **types of implemented locks** The locks we implemented are **LOCK-S** and **LOCK-X**. We thought we had to strictly keep to the lecture notes for *Datenbanken Vertiefung* where those two are the only discussed lock types for the **2PL** chapter. In our current implementation sometimes the concurrent modification exception is thrown, because for the following possible and not so unprobable schedule:
  - Transaction X acquires an exclusive lock on data A. Before acquisition any conflicting lock was accounted for, i.e. a transaction waits until every conflicting lock gets released
  - While X waits, another transaction Y acquires some other lock on either data A or on the whole table and starts its works. Transaction X however never re-checks for conflicting locks.
  - Now transaction X can finally start and starts removing tuples or does any kind of write(A).
  - Since Y wants to read that data A and has acquired a shared lock on data A, this transaction runs into an exception.

  So we have an existing problem with exclusive locks which can be easily solved by using an intentional exclusive lock on the table. We will fix this problem until the next assignment. As briefly discussed above, we support table and row or tuple locks.

- **implementation details of the concurrency mechanism (requesting/granting locks)** The locking magic happens in the class *ConcurrentControlManager* within the two methods **lockManagementGrowing** and **lockManagementShrinking**. The names should imply which method does what. In our implementation the shrinking phase happens exactly at the end of each transaction, so the growing phase lasts right until the end of a transaction. As requested for the **2PL** scheme no lock gets released/removed until the shrinking phase begins and whilst shrinking no lock can be acquired. The following variables are essential for our locking mechanism:
  - HashSet<Lock> flightLocks
  - HashSet<Lock> seatLocks
  - HashSet<Lock> passengersLocks
  - HashSet<Lock> reservationLocks
  - TreeMap<Integer, HashSet<Lock>> locksPerTransaction
  - Lock flightsTableLock
  - Lock seatsTableLock
  - static Lock passengersTableLock
  - static Lock reservationsTableLock

  The sets of locks are used for row/tuple locking for their respective table as indicated by the names of the sets. The locks of type **Lock** are used as table locks. The data structure of *Lock* is discussed below. The variable *locksPerTransaction* is essential for our shrinking phase. In the method *lockManagementShrinking* a transaction simply opens and/or removes any lock held by that transaction. Those lock references are kept in a *HashSet* and are mapped to the unique *agentId* of a thread. Since any thread can only do one transaction at a time, this mapping is unique in turn. Further information is saved in the **Lock** data structure with the following fields:
  - LockType type;
  - LockStatus status;
  - HashSet<Integer> agentIDs;
  - DBStructure.Table table;
  - Integer row;
  - Integer flightId;

  As discussed above the lock type can either be **LOCK-S**, **LOCK-X** or for convenience while debugging **UNSET**. The status of a lock is now an *enum*. A lock can have status **OPEN**, **LOCKED** or **REMOVED**. The status *REMOVED* is only used for tuple locking while a table lock is either *OPEN* or *LOCKED*. The *agentIDs* are for table locks. Tuple locks are created for each transaction, i.e. multiple *LOCK-S* locks on one row exist multiple times. The last 3 fields of the

data structure *Lock* are important for the shrinking and growing phase alike. So we save for which table that lock was granted and on which data. The field *flightId* is or can be used for the tables *seats* and *reservations*.

growing phase. This phase is built up as follows:

1. Check for a conflicting table lock and remember it
2. Check for conflicting row/tuple locks and remember them
3. Wait for each remembered lock until status either becomes *OPEN* or *REMOVED*
4. Always check for timeout, in case of timeout, abort transaction
5. If no timeout occurred any conflict could be solved so either create a new lock (tuple locking) or set a table lock to *LOCKED*

The method *lockManagementGrowing* is called in DBConnection right before a call to a function in DBAccess. At the end of the growth function either a lock reference or null is returned. If DBConnection receives a null pointer the transaction aborts.

4. EXPERIMENTS

4.1 Changelog

Once again we had to rearrange or modificate a few things here and there. Notable this time is the implementation of *multiple granularity locking*. So now we have the lock types IS, S, IX and X. The lock type SIX is defined but unused. As per definition in the lecture notes we first lock our table either in IS or IX and then the wanted tuple, in case of reading or writing a whole table S and X are set on the table. Furthermore we now support granularity escalations. Should a compatible lock be granted on an already locked table that older luck gets a higher granularity. The new method

```
ConcurrencyControlManager.
 isCompatible(LockType type1, LockType type2)
```

represents the multiple granularity locking matrix by comparing two given lock types. Another new method is called

```
ConcurrencyControlManager.
 determineWantedTableLockType(
   LockType wantedType, boolean tableLock)
```

where we map a wanted lock to a table lock type. A thread may request only LOCK-S or LOCK-X on a tuple but the table then requires an intentional lock. The returned type of this method is then compared to an existing lock on a table by using above function.

4.2 Points to discuss (not limited to)

- **experiments setup (programming language, transaction distributions, measures)** Our implementation is written in Java. For multithreading we created a subclass of *Thread*, those threads are then started with *.start()* and by calling *.join()* in the main process on those threads we wait for their termination.
- **each experiment (method, results, findings)** We got one interesting result. Our implementation of serial and 2PL seem to scale at the same pace with a growing agent or thread count as seen in 1. No matter how much time we invested to find an error within our implementation we could not locate any. This leads to our gues that maybe Java, OS or the hardware do something behind the scenes. For each test run with a different amount of threads we recreated the data structures, but did that in the same way. So maybe something gets cached and thus becomes real fast while accessing. We could not determine the real cause though. The expected result for serial would have been for the throughput to stay constant independent to the thread count, since in serial mode at any time only a single process can execute the query.

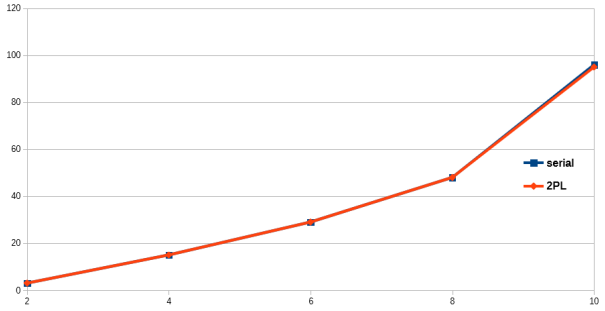

Figure 1: Throughput in 1/s with varying agent count