TRANSACTION EXPERMENT

Linh Tran and Rui Zhang
Department of Computer Science
University of Arizona
ruizhang@cs.arizona.edu and linhtran@cs.arizona.edu

Abstract

As a scientific field, computer science has long been viewed as the hybrid of engineering and mathematics, whereas the science component is often less paid attention to. The lack of scientific methodology raised the questions of credibility, validity, and reliability against computer science research, especially on the issue of experiments. Normally, the experiment results are limited within the bound of performance comparison of algorithms and implementations, which are considered engineering and mathematical efforts, rather than searching for significant and interesting phenomena, and the explanation of such phenomena via theories and the verification or falsification of relevant theories and hypothesis, like other scientific fields do. In this paper, we will be discussing the efforts and approaches we tried to perform scientific experiment on DBMS and the hypothesis we raised on some phenomena we found during the experiments. Also, we will emphasize on the design of our experimental platform ExpTran, on which we conducted our experiments.

1. Introduction

Science, engineering and mathematics have long been considered as the three main components of Computer Science. Although called science, for decades researchers have been focusing on the engineering and mathematical aspects of the field, not too much effort has been invested in the science component. Engineering is the design, analysis, and construction of work for practical purposes. One example of engineering in Computer Science is the process of either inventing new module or reusing some existing module to speed up the software development process. Researchers in Computer Science also spend a lot of time studying existing algorithms and trying to make improvement in terms of time complexity, space complexity, in which case is again considered as engineering activities. Mathematics, on the other hand, is the body of knowledge centered on concepts such as quantity, structure, space and also the academic discipline that studies them [4][5]. Mathematics has been highly involved in computer science. Mathematics in Computer Science involves the development of theories and approaches for computer and information sciences, and the design, implementation, and analysis of algorithms and software tools for mathematical computation and reasoning, and the integration of mathematics and computer science for scientific and engineering application [4]. The analysis of recursive algorithm that utilizes the recurrence equation is one straightforward example, which could be viewed as basic application of mathematical in Computer Science. In a broader view, mathematical theories such as graph theory, or discrete mathematics, set theories are applied everywhere. Science may be defined as developing general, predictive theories that describe and explain observed phenomena, and evaluating these theories [5], say via hypothesis testing. While the database field has some very strong mathematical and engineering work, the scientific perspective has been much less prominent [3]. The purpose of this project is to discover some valuable phenomenon in doing scientific research in computer science, to be specific, in Database area. We will develop necessary experimental tool and conduct experiments scientifically and hope to raise the attention of the science aspect in computer science.

Unlike traditional fields such as physics, chemistry, and biology, which all study existing objects in nature, Computer Science research focus on human made objects, thus it introduces some conceptual difficulties in understanding the scientific component or aspect in Computer Science. The objective of natural science is to

understand the underlying principle as how nature object behaves. One may argue that there is no such motivation in Computer Science because it was human who design the computer. However, because it is the human made object, computer scientists understand the behavior of computers at the first place. When doing research, they try to come up with the correlation between the input and output by treating the studied object as a black-box. And then study the behavior of the black-box to generalize hypothesis or theory, make observations to make prediction in the future with new input, and finally try to test the validity of the hypothesis. As the matter of facts, there are a lot of phenomena in CS that is worthwhile to study. One example in DBMS, the transaction processing module as a functional object could be viewed as a black-box by which we could look into the phenomenon appeared in scheduling, caused by deadlock, convoy phenomenon, and *hotspot*. Through the studying of these phenomena, it can help us to understand the behavior of the transaction manager in a scientific way. The common way of studying a scientific matter is through observation. Therefore, in order to study the transaction manager, we need to develop a tool to assist us to make observation. Thereafter, we developed this `ExpTran` (Experiments of Transactions) experimental platform that will help us to specifically study the transaction manager's behavior of Oracle DBMS.

2. Background

What is transaction? A database transaction is a unit of interaction with a database management system that is treated in a coherent and reliable way independent of other transactions that must be either entirely completed or aborted. Ideally, a database system will guarantee all of the ACID properties for each transaction. Databases usually store information that describes the current state of an enterprise. A single transaction might require several queries, each reading and/or writing information in the database. When this happens it is usually important to be sure that the database is not left with only some of the queries carried out and transactions should not interfere with each other. Each transaction must be designed so that it maintains the correctness of the relationship between the database state and the real world enterprise it is modeling [6].

What is a transaction manager? A system that manages transactions and controls their access to a DBMS is called a transaction manager. It is usually consists of a monitor, one or more DBMSs, and a set of application programs containing transactions. The database is the heart of the transaction manager because it persists beyond the lifetime of any particular transaction.
What is scheduling? A schedule is a list of actions ("reading", "writing", "aborting", and "committing") from a set of transactions. The order in which two actions of a transaction appears in the schedule must be the same as the order in which they appear in the transaction. Schedule represents an actual or potential execution sequence. A schedule that either contains an abort or commit for each transaction whose actions are listed in it is called the complete schedule. If the action of different transactions are not interleaved, that is transactions are executed from start to finish one by one, it is called a serial schedule. A serializable schedule over a set of committed transactions is a schedule whose effect of any consistent database instance is guaranteed to be identical to a data of some complete serial schedule over this set of committed transactions [5].

What are ACID properties? The ACID properties of a DBMS allow safe sharing of data. Without these ACID properties, everyday occurrences such using computer systems to buy products would be difficult and the potential for inaccuracy would be huge. The acronym stands for *Atomicity*, *Consistency*, *Isolation*, and *Durability*.

3. Overview

ExpTran stands for Experiment of Transactions and is developed in Java as a specialized experimental platform for studying transaction processing. ExpTran is a software environment that carries out experiments; stores experiment results, retrieves results, and visualizes them.

The purpose and focus of this project is to examine the parallel execution of transactions as to study some interesting issues in concurrency control in transaction management. Eventually, we hope to develop certain generalized hypotheses and theories through the observation of the phenomena appear in the experiments. The interesting phenomena we target at may appear in *deadlocks*, *convoy phenomenon*, and *hotspot* problems. By studying these phenomena and then addressing our hypotheses, we hope to understand the limitations of DBMS in terms of transaction processing. For example, under what settings of the parameters, will the system generate a "good" schedule? Will the system be falsifiable under some conditions? In order to pursuit these goals, ExpTran will assist us to carry out experiments.

The ExpTran system consists of three components, as shown in figure 1 below. The first component is the interface that communicates with Oracle database via JDBC and handles the execution of transactions. The second component is for data preparation, which consists of TransactionGenerator and ScheduleGenerator. Its task is to randomly generating multiple transactions to feed to the execution module. The last component is the result storage and retrieval module, which stores the transaction execution result, including the execution orders and values read and written by each operation, into a result table, and also retrieves the results at time of performing analysis. Besides these main components, a GUI is also developed in assist to visualize the transactions and the corresponding execution result both textually and graphically.

A more detailed description and discussion of the design of ExpTran will be presented later in section 4 & 5 & 6. In section 7, we will address the hypothesis we stated at the beginning, and discuss the procedure of the experiments, which we designed, to support and falsify the hypothesis. In section 8, we will draw conclusions based on the experiments. Section 9 will bring up some issues regarding to extend this project to a broader range, such as to integrate more complex transaction models. Moreover, we will discuss how to integrate this system into AZDBLAB, which is a more general DBMS experimental tool used to study query optimizers. By integrating ExpTran into AZDBLAB, we hope to make it more powerful that it can help researchers perform their studies among many issues in different DBMS.

4. System Design

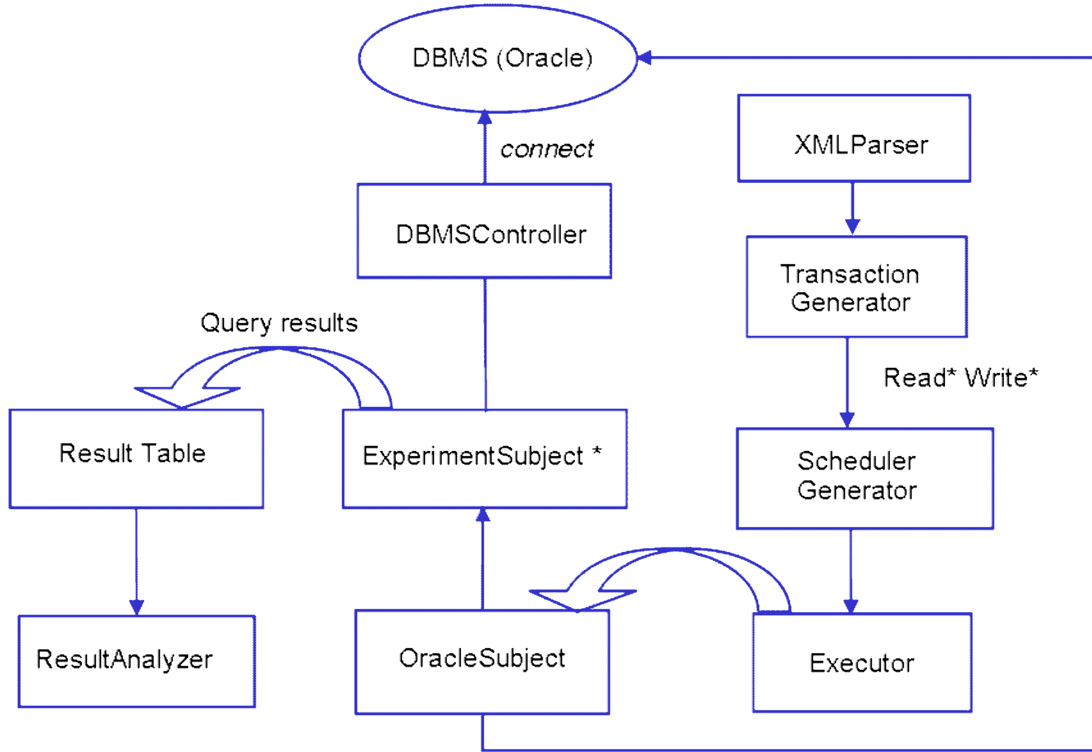

Figure 1. ExpTran Design overview

4.1. Design of ExpTran

In ExpTran, there are five main components: experiment specification, generator, and experiment subject, executor and result store. The experiment specification consists of two parts, the experiment data and the XML parser. The experiment data consists two categories. One is the specification for the experiment table and the other is the specification for each transaction in our experiment. The XML parser handles these experiment specifications and passes it into the corresponding components. Transaction generator will generate multiple transactions, and then the schedule generator will create a serialized order of these transactions. Afterward, a sequence of transactions will be fed into the executor. The executor will be responsible for creating a single thread for each transaction, and execute them according to the schedule (specified order). During the execution, each operation will be assigned a unique order number as to reflect the actual execution order according to the DBMS. In our design, each transaction is embedded into an experiment subject, which extends the general DBMS class, which is the interface of the actual DBMS that we run our experiment on. After each operation within a transaction is being executed, the result will be stored into the result table through result store component. The detail of class diagram is shown in the figure below. We will describe each of the modules in detail.

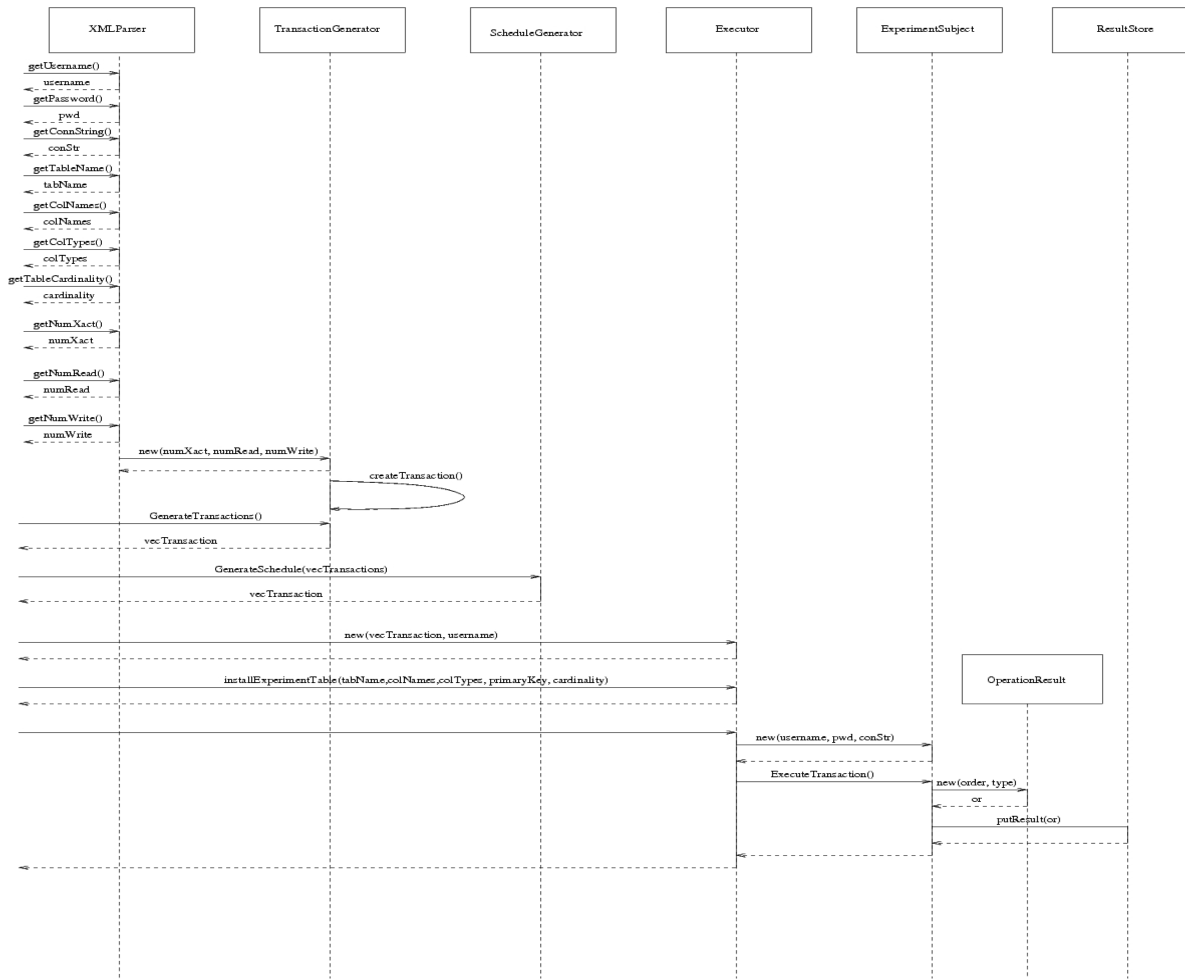

Figure 2. Sequence diagram

4.1.1. Module I: Experiment Specification

This Module contains the specification for the experiments. The specification has two parts, first part is the experiment data, and the second part is the `XMLParser`.

`Experiment Data`

The experiment data includes the configuration information with username, password, and connection string to the DBMS.

```
<configure
   username = "rui"
   password = "rui"
   connectstring = "jdbc:oracle:thin:@aloe.cs.arizona.edu:1521:oracle"
/>
```

The experiment data also specifies the structure of the data table. The experiment table structure contains the name of the table in which we will be running our experiment, the number of attributes in the table along with the attributes' name and type, the cardinality of the table.

```
<tableExperiment tableName = "dumtrans" numCol = "2" cardinality = "50">
<col name = "ID" type = "INT"/>
<col name = "num" type = "INT"/>
</tableExperiment>
```

Organizing this experiment data information in a separate, specific file helps to track and modify the data more efficiently as well as limit errors.

The experiment data also includes the overall information about all transactions and the actual information about each the transaction that we will be running. The overall information contains the total number of transactions, and the seeds we use to generate random number for the generator component. For example:

```
<xactProperty numXact = "2" seedSG = "1000" seedTG = "9"/>
```

Information about each transaction includes the transaction ID, the number of read operations and the number of write operations belong to that specific transaction. Moreover, the indication of whether the operations will be applied to single tuple or multiple tuples as well as the ID of the tuple to be accessed by the transaction are also included.

```
<transaction xactID = "0" numRead = "100" numWrite = "100" singleRow = "1"
tuple = “1”/>
```

`XML Parser`

As mentioned before, we store our experiment data in XML files format. In order to utilize this data, we map the data into specific data structure, so that other components can access the structure for specific piece of information. We developed the `XMLParser` for this purpose. The XML Parser will read in from the XML files the configuration information, the data table information, and the transactions information; store the information in its private data structure. When other components need to retrieve any of the information, they can request the XML Parser for the specific information through its public methods.

4.1.2. Module II: Transaction & Schedule Generator

`TransactionGenerator`: The transaction generator will generate a list of transactions as specified in the experiment data file. Individual transaction details can be retrieved by requesting `XMLParser` and then stored in the transaction object. Each transaction instance contains its transaction ID, the numbers of read and write operations, and a list of actual operations, which belong to this transaction. There are three types of operations, read, write and commit. The read operation simply retrieves information existing in the data table in database. The write operation updates a single tuple or multiple tuples in the database, depending on the specification in XML files. The last operation of every transaction is the commit operation. These operations have the format of an SQL statement of ‘SELECT’, ‘UPDATE’ and ‘COMMIT’, respectively.

For each transaction, we have the number of read operations and the number of write operations. Based on these, we first randomly generate the order of the read operations. And since the length of the operation list is fixed, the rest empty slots in the list will then be filled with write operations. Thus we will have the read operations interleave with the write operations randomly. Finally a commit will be put at the end of the transaction.

ScheduleGenerator: Transaction generator is responsible for generating a list of transactions. Schedule generator is responsible for specifying the order in which these transactions will be executed. Thus schedule generator will first initialize a set of transactions, and then randomly assign the order of execution to each transaction. For example, T_1, T_2, and T_3 are three generated transactions. After scheduling by the schedule generator, the order of execution of these transactions may become T_2, T_1, and T_3.

4.1.3. Module III: Executor

Executor is a very important part of the ExpTran system. It handles the execution of the transactions. Moreover, it provides the mechanism of assigning each executed operation a unique order number. This order number helps us to study the global order of operations in execution series. The executor consists two main methods, ExecutionTransaction() and InstallExperimentTable(). ExecutionTransaction() method handles the transaction execution. It iterates through the list of transactions available and executes a commit before each transaction to ensure these transactions will be started correctly, since the duration of a transaction is from the last commit to the next commit. Then the isolation level for each transaction is set. The valid isolation levels vary according to different DBMS. For instance, in Oracle, "SERIALIZABLE" and "READ_COMMITTED" are the two available levels. And by default, the "READ_COMMITTED" is used. [5] After the configuration steps, each transaction, carried by an individual ExperimentSubject instance, is executed as a Java thread. Therefore, executor simulates the parallel execution of multi-transactions by multi-threads in Java. InstallExperimentTable() method creates the data table on which all the transaction operations are performed. It uses RepeatableRandom() number generator to produce random numbers to populate the table. With the same seed passed into the number generator every time, the Executor ensures the data table is exactly identical for each experiment, if repeatability is required. The Executor also provides the functionality to keep track of the global execution order of all the operations.

4.1.4. Module IV: Experiment Subject

ExperimentSubject is a generic class, which provides the basic functionalities of performing read, write and commit operations. It takes a single transaction as input and acts as the carrier of the transaction. ExperimentSubject extends DBMSController class by making use of the connection and statement instances created by DBMSController to the experiment DBMS. ExecuteTransaction() is an abstract method to be implemented by the sub-classes of ExperimentSubject. In our system, OracleSubject implements this method. Within ExecuteTransaction(), the list of operations in a transaction is iterated through. At each time, the corresponding operation method is called, such as executeRead(), executeWrite() or executeCommit(), in comply with the types of operations. At the same time, a unique order number, produced by Executor, is assigned to each operation. After this, the results, including the order, the operation name, the returned data, if it is read operation, are stored into the result table through ResultStore. We will lay down the detail for ResultStore in section module V.

4.1.5. Module V: Result Store

`ResultStore` is another interface between DBMS and `ExpTran` system. But `ResultStore` is only responsible for storing and retrieving the result from result table, which could be located in a different DBMS than the one used in the experiment. But for simplicity reason, we installed both experiment table and the result table in the same DBMS. As stated before, the results will be put into the table according to their types. In our design, we implemented `ReadResult` and `WriteResult` classes, which both extended the `OperationResult` class. The difference between these two is that `ReadResult` stores the value read from the data table, while the `WriteResult` remembers the new value associated with the write operation.

5. Graphical User Interfaces

In this project, we want to design experiments to study about some interesting issues in Oracle DBMS's transaction processing system. Graphical User Interface will be an effective representation for our study. Thus, we designed our tool to present the textual and graphical visualization of transaction processing.

Our GUI consists of two components, one represents the input data, and one represents the experiment result. The GUI allows us to carry out experiments from XML specification file easily and also allow us to view the result both textually and graphically, thus we have the two components. The first component allows the selection of the XML data input and display the parameter values such as the transactions details with total number of transactions, the seed number, and the number of read and write operations in each transaction in detail as shown in figure 3. The actual configuration of individual transaction is also displayed graphically.

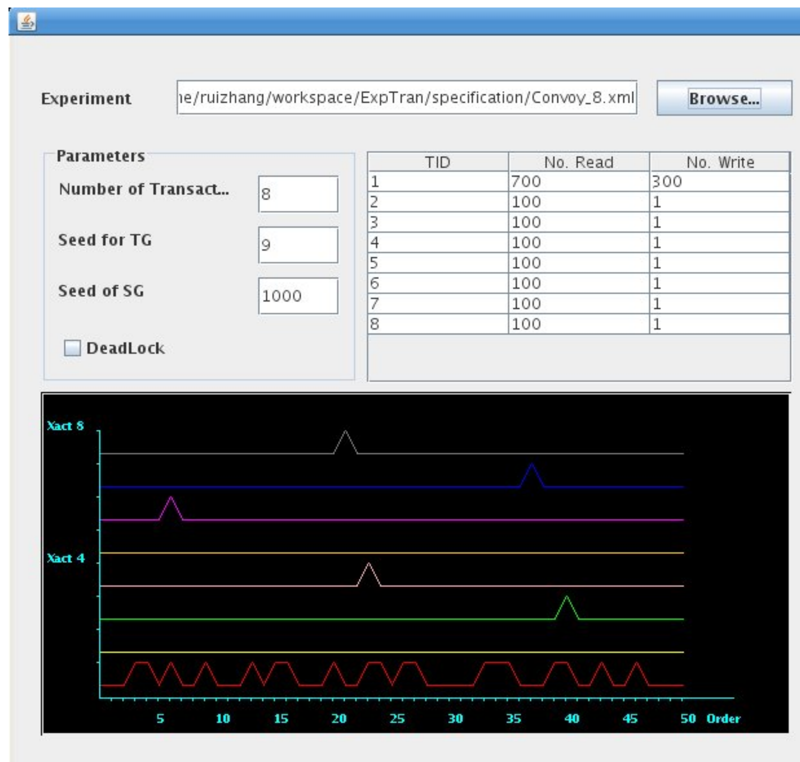

Figure 3. Experiment data display

The “Start Experiment” invokes the execution of the experiment, and the result will then be shown on the right hand-side textually. Results can also be viewed graphically by clicking the “Graphic” button. The result here shows the execution order, transaction ID, operation type, and the operation result. All the writes are highlighted. In the graphical representation, each transaction is represented by a different colored curve, so that we can easily identify their interleaving execution.

265 270 275 280 285 290 295 300 305 310 Order

Figure 4. Graphical result visualization

Result

Order	Transaction	Operation	Result
634	1	READ	475
635	1	READ	475
636	1	READ	475
637	1	READ	475
638	1	READ	475
639	1	READ	475
640	1	WRITE	482
641	1	WRITE	483
642	1	READ	483
643	1	READ	483
644	1	WRITE	486
645	1	READ	486
646	1	READ	486
647	1	READ	486
648	1	READ	486
649	1	WRITE	491
650	1	READ	491
651	1	WRITE	493
652	1	WRITE	494
653	1	READ	494
654	1	READ	494
655	1	READ	494
656	1	READ	494
657	1	READ	494
658	1	READ	494
659	1	READ	494

Start Experiment   Graphical ...

Figure 5. Textual result visualization

6. Special Issues in System Design

6.1. Operation Order

As mentioned in previous sections, one of the most critical issues in this design is to keep track of the actual execution order of each operation as executed by the DBMS. This is critical because the number reflects the alignment of the operations and thus helps us to reconstruct the real order so as to study some of the phenomena in the transaction execution. Thus, we create a static variable in `Executor` as to be incremented by one at a time and assigned to each operation after its execution. However, the difficulty is that we have to make sure that incrementing the variable and executing the operation have to be atomic, in another word, whenever a thread is updating the variable and executing the operation, no other threads are able to change the value of the variable, and thus, preserves the real order of the operations. To achieve this goal, we set the methods of executing operations with synchronized keyword in Java, and also include the updating of the order variable within the methods. The synchronized keyword enforces at each time only one thread is accessing the method, and thus, guarantees updating the variable and executing the operation together is atomic.

6.2. Replicability

There are mainly two reasons for us to address the replicability issue. First, we want to allow other users to take the exact same experiment and verify our observation and conclusions from the results. And the most effective way is to make sure that the other users can actually conduct the same experiments with strictly identical configurations, such as the raw data, the parameters and so forth. Second, on the other hand, we may encounter unpredictable factors in the results, which may be expressed by the inconsistency of the results from the same experiment. Such situation may actually help us to study the undeterministic part of the transaction processing. The study is valid only if the raw data and the settings of the experiment is replicable.

To achieve the goal of replicability, we developed `RepeatableRandom` class. It is basically an random number generator, which allows the user to specify the seed for the number generator explicitly. If so, the generator can produce the exact same sequence of integer numbers. This class is utilized in several places, including `TransactionGenerator`, `ScheduleGenerator` and `Executor`. `TransactionGenerator` randomly arranges the order of operations in each transaction it produces. `ScheduleGenerator` creates a random order of all the transactions generated by the previous class. Executor populates the data table with randomly numbers. Thus, by using `RepeatableRandom`, we can guarantee that these factors are totally replicable.

6.3. Random read write and "hotspot" read write

The write lock of Oracle DBMS is at the tuple level, rather than table. Thus, the effect is different between operating on a single tuple and on multiple tuples in the data table, which, obviously in the former case, if multiple transactions all have write operations, there will be conflicts for sure. But in the second case, it may not necessarily have conflicts. Therefore, in order to study both cases, we create a parameter in the experiment specification file, which tells whether a particular transaction should only access one single tuple, which is often referred as the "hotspot," or it could actually work on multiple tuples.

7. Experiments

7.1. Deadlock

Deadlock is a common issue in many computer science areas such as operating system, architecture, and etc. In database, it happens quite often in transaction processing, in which transactions are waiting on the resources held by each other. Generally, the way to resolve deadlock problem is to kill one of the transaction that is currently holding a resource and dispatch the resource to other transactions. The selection of transaction to kill is an interesting issue to be studied.

Hypothesis: In the situation of deadlock of two transactions, Oracle DBMS will randomly select one transaction to kill if the length of the transactions is small otherwise, Oracle DBMS will pick the shorter one.

We designed a simple case of deadlock with two transactions T1 and T2, each with only two write operations and an arbitrary number of read operations. At the beginning, transaction T1 writes tuple A that is to acquire write lock on tuple A, and T2 writes tuple B that is to acquire write lock on tuple B. At later time, T1 writes tuple B and T2 writes tuple A, in which case, the lock on tuple B and A are already held by T2 and T1 respectively. Therefore, neither T1 nor T2 can acquire the lock at this later time. This is the deadlock situation. The Oracle DBMS kills one of the two transactions in case of deadlock. Since the longer transaction intuitively is more informative in that it will return more information than shorter one, we assume the DBMS will always kill the shorter one then. In the experiment we varied the length of the two transactions. The results are shown in figure 6. We observed that when two transactions have the same length, or if the length difference is less than 4, the determination of which transaction to kill is random. Otherwise, the DBMS will pick the shorter transaction to kill, allowing the other to continue execution.

Order	Transaction	Operation	Result
1	2	WRITE	200
2	2	WRITE	201
3	1	WRITE	100
4	2	COMMIT	
5	1	READ	100
6	1	READ	100
7	1	READ	100
8	1	READ	100
9	1	WRITE	105
10	1	COMMIT	

(a)

196	2	READ	200
197	1	READ	100
198	2	READ	200
199	1	READ	100
200	2	READ	200
201	1	READ	100
202	2	READ	200
203	2	WRITE	301
204	1	COMMIT	
205	2	COMMIT	

(b)

Figure 6. Deadlock scenario (a) DBMS killed transaction 2 (b) DBMS killed transaction 1

7.2. Convoy phenomenon

The word convoy is originated from military. It is used to describe a line of armed vehicles. An unusual phenomenon associated with the convoy is that the speed of the first vehicle affects the speed of the whole line of vehicles. For example, if the first truck is attacked or runs into some problem, the rest line would have to stop or move slower than before. Similarly, in transaction processing, such situation also exists. For example, if we have multiple transactions running in parallel and among them, one transaction requires a write lock, in which it will block out all other transactions. This is where the

convoy phenomenon appears and especially when this transaction is long, the convoy phenomenon becomes significant since the long transaction may delay the overall execution time to a great extend.

Hypothesis: Once a convoy is formed, the parallel execution of transactions becomes serialized and stable; besides, the CPU load is increased significantly.

In order to investigate into the convoy phenomenon, the key issue is to make sure that most of the transactions will eventually executed in a serialized fashion. In our experiment, we specified that all the transactions to acquire the same write lock on a particular tuple in the data table. Once the first transaction acquires the write lock, all other will have to wait and so on and so forth. We specified one of the transactions to be a long one, in another word, to contain excessive number of read and write operations, so that it can simulate the slow truck in the convoy phenomenon described at the beginning.

In particular, this is the detail of our experiment:

```
<xactProperty numXact = "8" seedSG = "1000" seedTG = "9"/>

    <transaction xactID = "1" numRead = "700" numWrite = "300" singleRow = "1" tuple = "1"/>
    <transaction xactID = "2" numRead = "100" numWrite = "1" singleRow = "1" tuple = "1"/>
    <transaction xactID = "3" numRead = "100" numWrite = "1" singleRow = "1" tuple = "1"/>
    <transaction xactID = "4" numRead = "100" numWrite = "1" singleRow = "1" tuple = "1"/>
    <transaction xactID = "5" numRead = "100" numWrite = "1" singleRow = "1" tuple = "1"/>
    <transaction xactID = "6" numRead = "100" numWrite = "1" singleRow = "1" tuple = "1"/>
    <transaction xactID = "7" numRead = "100" numWrite = "1" singleRow = "1" tuple = "1"/>
    <transaction xactID = "8" numRead = "100" numWrite = "1" singleRow = "1" tuple = "1"/>
```

After running this experiment, we found that at the beginning, all the transaction IDs are interleaved. But after a while, they are not interleaving anymore, but totally serialized such that each transaction is only executed after the previous transaction is committed.

Order	Transaction	Operation	Result
1	7	READ	149
2	3	READ	149
3	6	READ	149
4	5	READ	149
5	4	READ	149
6	1	READ	149
7	8	READ	149
8	2	READ	149
9	7	READ	149
10	8	READ	149
11	1	READ	149
12	4	READ	149
13	5	READ	149
14	6	READ	149
15	3	READ	149
16	5	READ	149
17	4	READ	149
18	1	READ	149
19	8	READ	149
20	7	READ	149
21	2	READ	149
22	7	READ	149
23	8	READ	149
24	1	WRITE	103
25	4	READ	149
26	5	READ	149

(a)

Order	Transaction	Operation	Result
1501	4	READ	423
1502	4	READ	423
1503	4	READ	423
1504	4	READ	423
1505	4	READ	423
1506	4	READ	423
1507	4	READ	423
1508	4	READ	423
1509	4	READ	423
1510	4	READ	423
1511	4	READ	423
1512	4	READ	423
1513	4	READ	423
1514	7	WRITE	737
1515	4	COMMIT	
1516	7	READ	737
1517	7	READ	737
1518	7	READ	737
1519	7	READ	737
1520	7	READ	737
1521	7	READ	737
1522	7	READ	737
1523	7	READ	737
1524	7	READ	737
1525	7	READ	737
1526	7	READ	737

(b)

Figure 7. Textual result representation of convoy phenomenon (a) Execution is highly interleaved at first. (b) Execution is serialized later

We performed the similar experiments multiple times, and all the results showed that the convoy phenomenon exists and the execution of the transaction becomes stable after the phenomenon is formed, especially when we have a very long transaction. Due to the scope of this project, we didn't extend our observation to the CPU load. We will examine this in future work.

7.3. Hotspot

The phrase hotspot refers to the location that is extremely popular. In database, it is used to describe the tuples in a table that is accessed at a high frequency. Based on the previous experiment, we also see a correlation between convoy and hotspot phenomenon, since most of the transactions need to acquire the lock on the same tuple, this specific tuple could be viewed as a hotspot. It will, to some extend, cause the serialized execution of the transactions, thus, it may delay the total execution time.

Hypothesis: *Hotspot* lock generally causes the delay of the execution of transactions.

The purpose of this experiment is to compare the execution time of two set of transactions, one with hotspot and one without. In this experiment, we designed two treatments, both of them have the same configuration in terms of all parameters, except that the first set of transactions all write to the same tuple (hotspot), whereas the transactions in the other treatment write to different tuples (non-hotspot).

Hotspot:

```
<xactProperty numXact = "3" seedSG = "1000" seedTG = "9"/>

    <transaction xactID = "0" numRead = "200" numWrite = "100" singleRow = "1" tuple = "1"/>
    <transaction xactID = "1" numRead = "200" numWrite = "100" singleRow = "1" tuple = "1"/>
    <transaction xactID = "2" numRead = "200" numWrite = "100" singleRow = "1" tuple = "1"/>
```

Non-Hotspot:

```
<xactProperty numXact = "3" seedSG = "1000" seedTG = "9"/>

    <transaction xactID = "0" numRead = "200" numWrite = "100" singleRow = "1" tuple = "1"/>
    <transaction xactID = "1" numRead = "200" numWrite = "100" singleRow = "1" tuple = "2"/>
    <transaction xactID = "2" numRead = "200" numWrite = "100" singleRow = "1" tuple = "3"/>
```

We performed the same experiments 20 times to collect the result data. In general, we found that the execution time of the hotspot treatment is longer compared to the non-hotspot one, as indicated by the graph below — the average execution time of hotspot treatment was 4.2 sec and non-hotspot treatment was 2.85 sec.

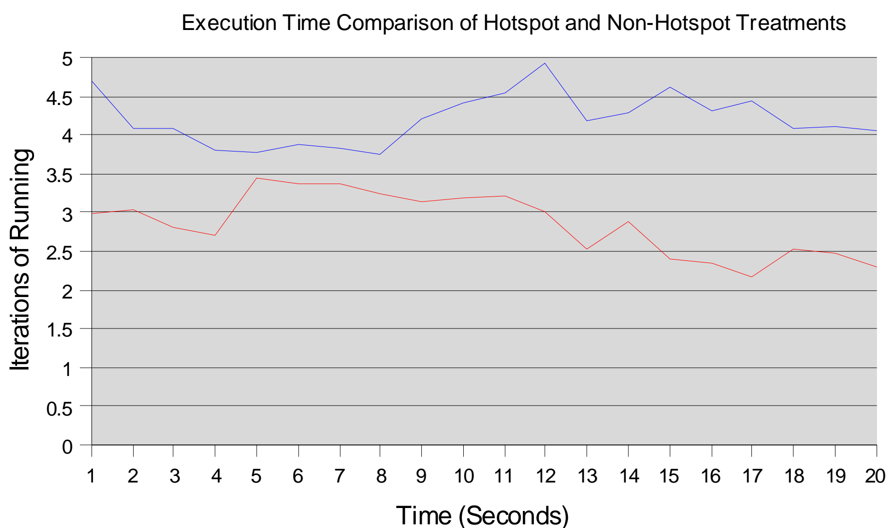

Figure 8. Execution time comparison of hotspot (blue curve) vs. non-hotspot (red curve) treatments

8. Conclusion

In this paper, we spent some effort in studying Transaction processing scientifically through observations of well-designed experiments. In order to conduct experiments properly, we developed an experimental platform--`ExpTran`. The platform consists of five major components. They are Experiment Specification, Transaction & Schedule Generator, `Executor`, `ExperimentSubject`, and `ResultStore`. Experiment Specification provides both the experiment data and configurations in the form of XML files and the parsing mechanism as the interface between the data file and the platform. Transaction Generator randomly creates a set of read and write operations for each transaction. And the Schedule Generator helps to reorder the generated transactions to form a serialized schedule. The serialized transaction schedule is then fed into the `Executor`. The `Executor` creates an `ExperimentSubject` for each individual transaction, such that each transaction is embedded in an `ExperimentSubject`, which handles the execution of the operations. The Executor then runs the transactions as Java threads according to the order in the schedule, but in a paralleled fashion. As the transactions are being executed, the result of each operation is stored into the result table via `ResultStore` component immediately. The result of each operation includes the global order of the operation, and for read operation, the value returned by the read, while for write operation, the value to be written into the experiment table. And finally, we designed a GUI that helps us to easily perform experiments as well as visualize the experiment settings and results.

With the above experiment tool, we conducted several experiment falling into three categories. These experiments were designed to study the transaction-processing module of DBMS as a black-box. The issues studied including deadlock, convoy phenomenon, and hotspot. For each category, we drew a hypothesis as follows:

- *Deadlock*: In the situation of deadlock of two transactions, Oracle DBMS will randomly select one transaction to kill if the length of the transactions differs less than 4; otherwise, Oracle DBMS will pick the shorter one.
- *Convoy phenomenon*: Once the problem is formed, it becomes stable, and also the CPU load is increased significantly.

- *Hotspot*: Generally, if no deadlock is involved, transactions on non-hotspot are executed faster than on hotspot.

9. Future work

As an experimental platform, `ExpTran` has a great potential to be enhanced. One way to improve it would be integrating it into AZDBLAB, which currently is a dedicated experimental system for studying query optimizers. The visualization design of the result display could be improved as well such that it could be more accurate and informative in terms of displaying important issues, such as deadlocks, convoys and so forth.

As far as the experiment is concerned, right now we are studying the simplest form of transaction, the flat transaction. In the future it would be interesting if we extend the experiment to a broader scope so that more models of transactions could be studied, such as chained transaction and nested transaction. Such transactions will contain extra operations like savepoint, rollback and so on. We also want to measure the CPU load as mentioned in the discussion of the convoy phenomenon. Moreover, most of the experiments we did in this paper were relatively easy to be carried out and all the results were expected without surprise. Nevertheless, what we are really looking forward to see is more exciting issues and maybe surprising results. Thus, we will work on increasing the complexity of the experiment as to dig deeper in the searching of more complex hypothesis and results.

References

1. J. Gray, and A. Reuter, **Transaction Processing: Concepts and Techniques**, (1993) Morgan Kaufmann Publishers, San Mateo, California

2. P. Lewis, A. Bernstein, and M. Kifer, **Databases and Transaction Processing: An application-Oriented Approach**, (2001), Addison Wesley

3. R. Snodgrass, *The Science of Databases*, (Feb 23rd, 2007), http://www.cs.arizona.edu/projects/soc/sodb/

4. Wikipedia, Mathematics, (Feb 23rd, 2007), http://en.wikipedia.org/wiki/Mathematics

5. Mathematics in Computer Science, (Feb 23rd, 2007), http://www.springer.com/west/home/birkhauser?SGWID=4-40290-70-173671506-0

6. Database Transaction, (Feb 23rd, 2007), http://www.answers.com/topic/database-transaction

APPENDIX

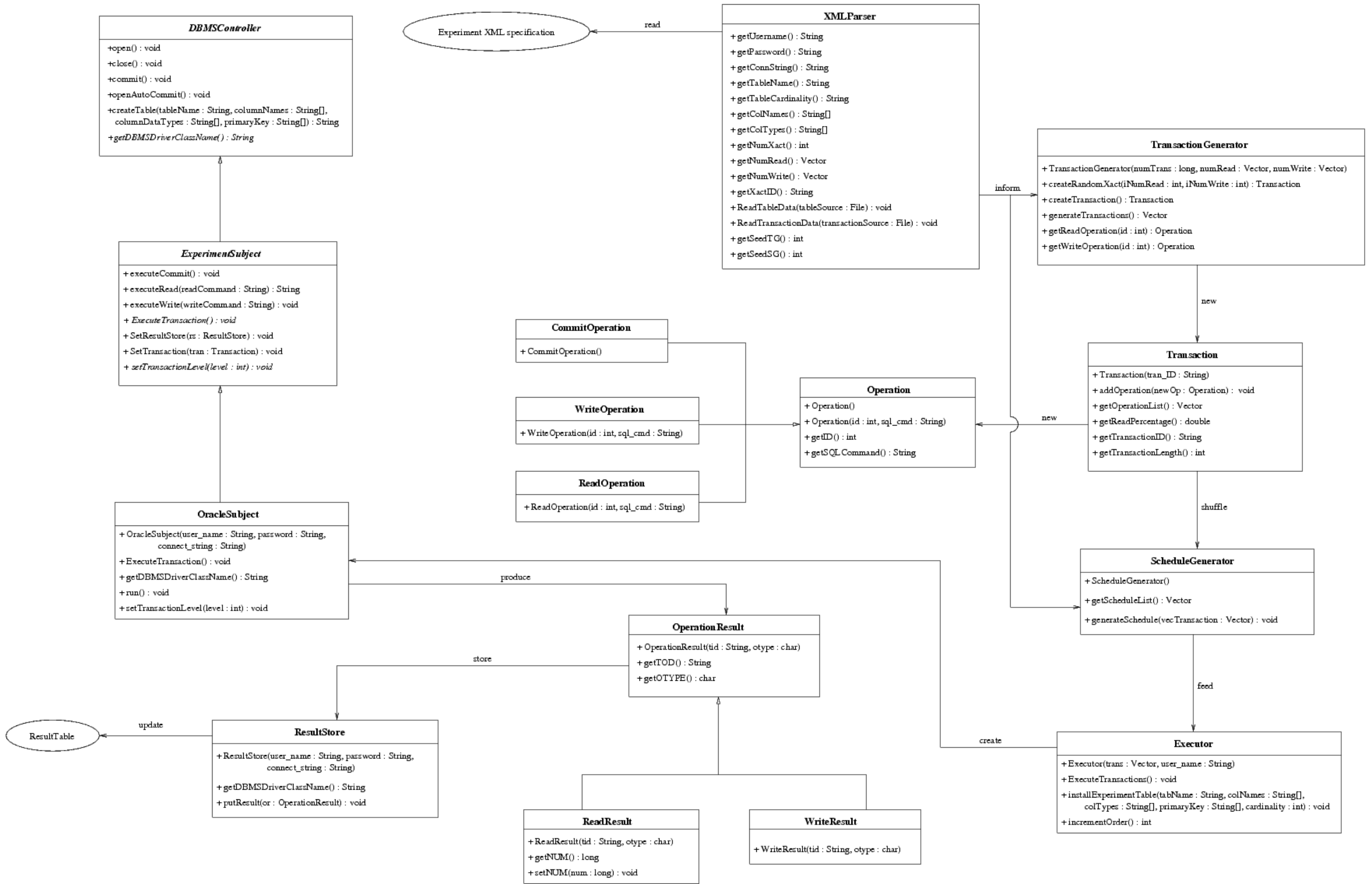

Figure 9. Class diagram of `ExpTran`