About

The 2PLDDB project is a centralized two phase locking database system. It has the following features:

- Distributed servers' environment to simulate centralized two phase locking.
- Deadlock detector and resolver.
- SQLite as the database engine.

Design

Our design of 2PLDDB distributed database system is illustrated in the next figure. The centralized site is a special server with a centralized lock manager of the whole system. This site maintains queues for requesting/releasing locks. The system is capable to support multiple other sites. Other site is a normal server with a local database. Each other site has a transaction manager and data processor. Processing takes place where it was submitted. Other site is assumed to make sure local concurrency control.

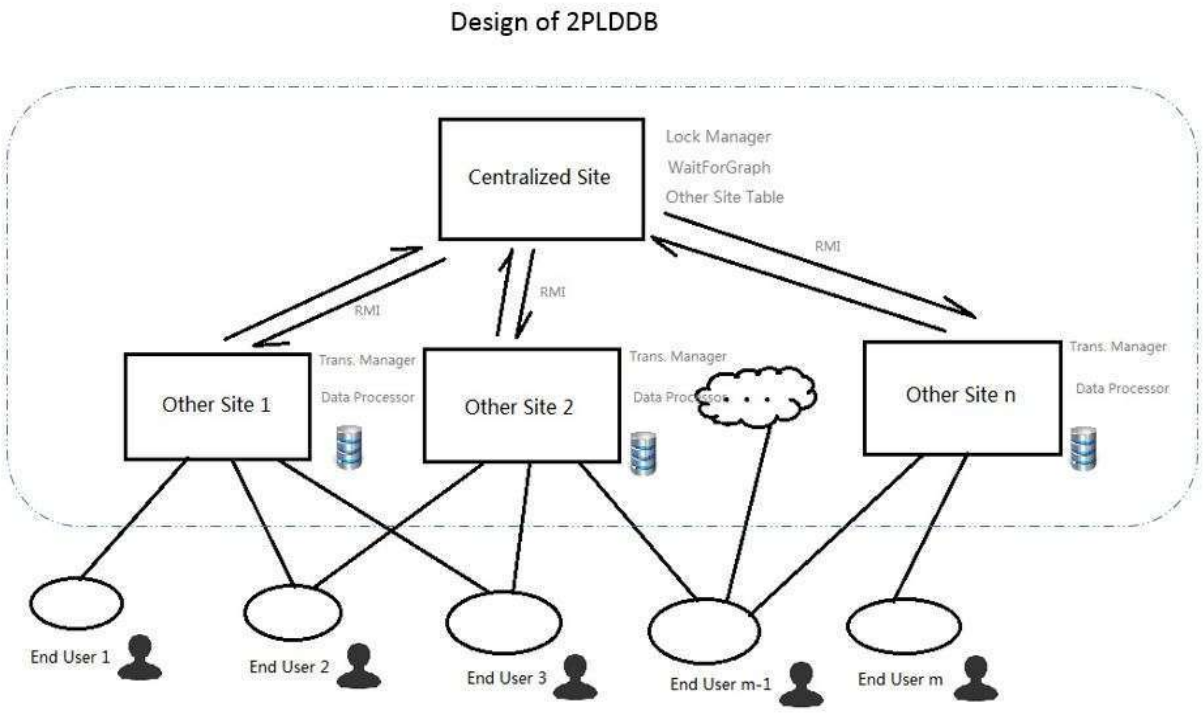

Figure 1: Design of 2PLDDB

Since the purpose of this project is to focus on centralized two phase locking, the system is implemented with the following two assumptions:

- Distributed end user environment is assumed. Only the part in the dashed box of above figure is implemented.
- Database commit protocol is assumed. Only one copy of database is maintained. So there assumes no data consistency issues.

The following are some technologies used in this project:

- Communication between central site and other sites is achieved with Java RMI.
- The database is powered by SQLite.
- Communication between other sites and database is achieved with JDBC.
- The Java system is thread-safe.

2PLDDB is designed to be extensible so that parameters can be configured easily. Parameters such as operation processing time, deadlock checking frequency can be configured in a class called “Constants”.

Algorithms

Centralized Two Phase Locking

The two phase locking in 2PLDDB is a centralized version where a central site is maintaining a global lock manager. In the meanwhile, it is also a strict two phase locking in which a transaction’s locks are released together.

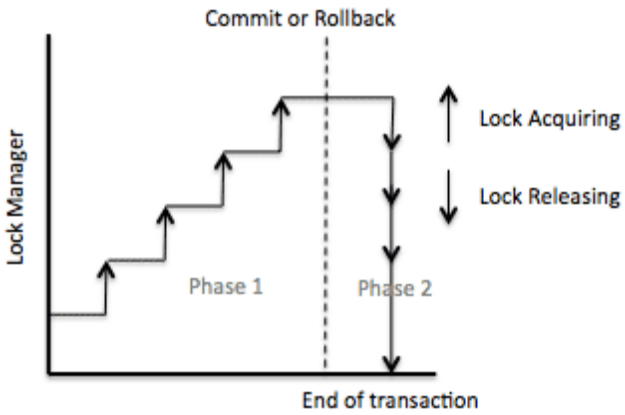

Figure 2: Strict Two Phase Locking

In 2PLDDB, transactions are processed locally at other sites. Once an operation obtains its locks, it will start processing before all of transaction’s locks are acquired. This approach will distribute computation workload for other sites.

The detailed algorithm implemented in 2PLDDB is according to the one in chapter 11 of the textbook (Oszu & Valduriez, 2012).

The lock types used in the 2PLDDB are “Read lock” and “Write lock”. The lock compatibility for different transactions is as following:

	Read Lock	Write Lock
Read Lock	Compatible	NOT Compatible
Write Lock	NOT Compatible	NOT Compatible

Table 1: The lock compatibility for different transactions

Deadlocks Detection and Resolution

Deadlocks are detected by the lock manager. The general idea is to check deadlocks with a certain period of time (by default, 3 seconds). Since lock manager already maintains a lock table and queues, a wait-for-graph can be constructed during deadlock checking. There might be more efficient algorithms to detect deadlocks (e.g. maintain a wait-for-graph for server life time). We will discuss this issue in “experience” section.

CheckDeadlocks:

*1. Construct a wait-for-graph **wfg**.*

*2. For each variable **var** in lockTable*

*3. For each lock unit **lu** for **var***

*4. For each operation **op** in the queue list for **var***

*5. If **lu**'s transaction **t1** != **op**'s transaction **t2***

*6. add an edge **t1** to **t2** in **wfg***

*7. Check whether **wfg** has a cycle*

Figure 3: CheckDeadLocks Algorithm

To resolve deadlock issue, one transaction on a circle is aborted. All locks and queueing operations of that transaction are released. A message is sent back to the other site of the aborted transaction. After receiving this message from the centralized site, the corresponding other site aborts the whole transaction and proceeds for the next transaction.

Note that only one transaction is aborted each time. If there are several transactions needed to be aborted to resolve deadlocks, the system has to go through several checking periods and abort one transaction at each time. It might take longer to resolve deadlocks in this circumstance.

Input Format

```
TRANSACTION:
r(X);
mX=1;
w(X);
TRANSACTION:
r(A);
mA=A+1;
w(A);
```

Figure 4: Simple Input

For each other site, its input file is a serialized transaction file. For demo purpose, each other site reads its transactions from this transaction file. Please make sure the format of transaction file is correct. Taking the following transaction file as an example, it is a list of transaction. Each transaction starts with a "TRANSACTION:" keyword. There are three different kinds of operations in 2PLDDB. Read operation starts with an 'r' letter. Write operation starts with a 'w' letter. Assignment operation starts with an 'm' letter ('m' for math). Both binary and unary assignments are possible. But you have to make sure any variables on the RHS of the assignment has been read while any variable of write operation has been assigned. Otherwise, there are transaction processing errors.

```
50386  w(Y);
50387  TRANSACTIO
50388  r(A);
50389  mA=A+1;
50390  w(A);
50391  r(B);
50392  mB=B+14;
50393  w(B);
50394  TRANSACTIO
50395  r(B);
50396  mB=B+15;
50397  w(B);
50398  r(X);
50399  mX=X+1;
50400  w(X);
```

Figure 5: Complicated Input

The above are a complicated history input for other site. It contains tens of thousands of transactions on only a few variables. If you bomb two other sites with the above input file, the sites run for very long time. You can observe a lot of dead locks and the procedure how dead locks are resolved.

Server Logs

In 2PLDDB, logs for each site are printed on screen. You can redirect logs into some log files in both Linux and Windows. The following figure shows logs for central site. The central site prints lock table and quotable with timestamp. Whenever there is deadlock, central site detects it and abort it. The aborting information is also printed.

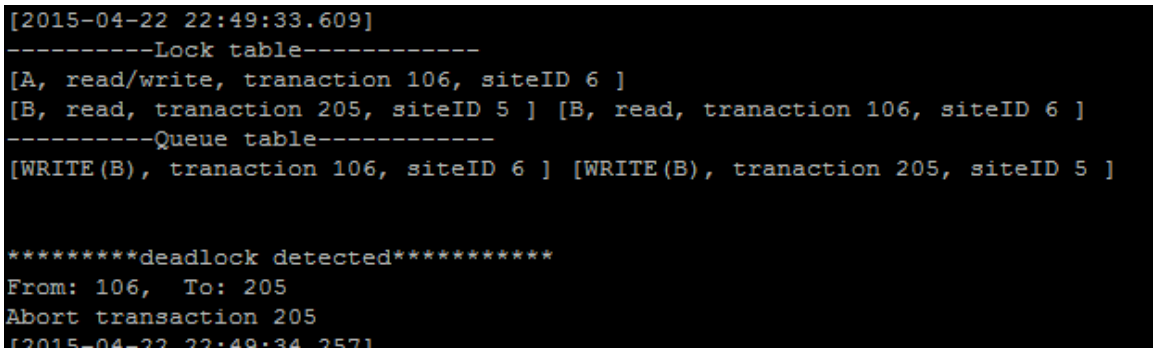

Figure 6: Central Site Logs

Similarly, logs are recorded for each other site with timestamps.

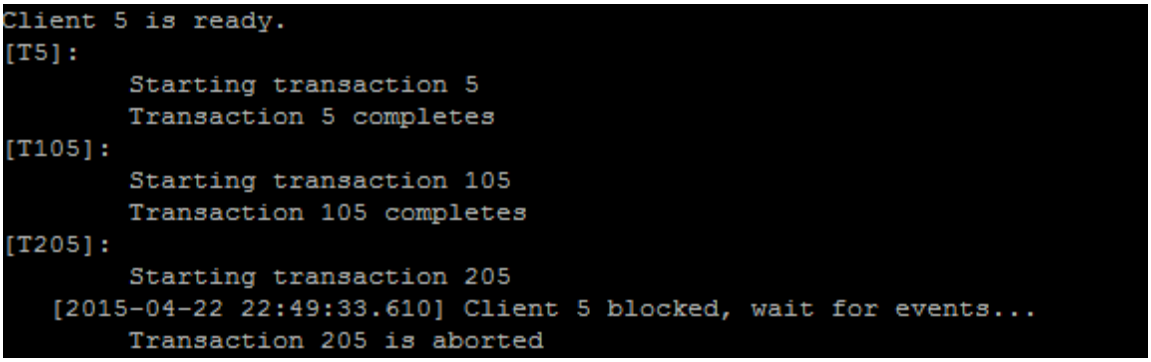

Figure 7: Other Site Logs

Install SQLite

Today, almost all the flavors of Linux OS are being shipped with SQLite.

To install SQLite in Windows, do the following:

(1). Go to SQLite download page, and download precompiled binaries from Windows section: http://www.sqlite.org/download.html

(2). you will need to download sqlite-shell-win32-*.zip and sqlite-dll-win32-*.zip zipped files.

(3). Create a folder C:\>sqlite and unzip above two zipped files in this folder which will give you sqlite3.def, sqlite3.dll and sqlite3.exe files.

(4). Add C:\>sqlite in your PATH environment variable and finally go to the command prompt and issue sqlite3 command, which should display a result something as below.

Access SQLite

To view values in SQLite, you can do the following commands:

```
Use SQLite3
---------------------------------
sqlite3 g2pl.db // connect to database called g2pl.db
select * from g2pl_database; // print the table
.quit // DO NOT forget to quit after viewing data values, OTHERWISE sites cannot write because the database is holden by sqlite3.exe
```

Figure 8: How to Access SQLite

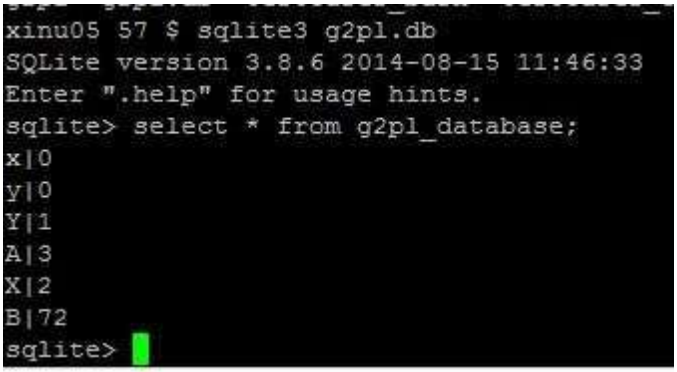

Figure 9: Database Values

Please remember to remove the database after you run a test case. A new database will be created for you when there is no available database. Otherwise the next test cases will access variable values in the old database.

Source Code

In this section, I am describing the structure of source code. The following figure shows the structure of source code.

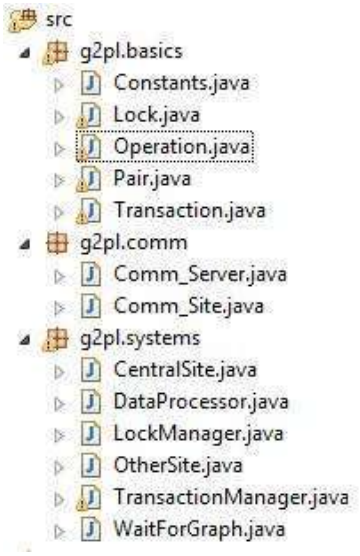

Figure 10: Source Code Structure

The system has a global package called "g2pl". There are three sub-packages under "g2pl": basics, comm, systems.

- Basics sub-package contains basic data structure and utilities.
- Comm sub-package contains Java RMI interfaces for central server and other sites.
- System sub-packages contains system infrastructures of 2PLDDB.

Then each file will be briefly described:

- Constants.java contains global parameters such as delays, operation types and etc.
- Lock.java contains lock class definition.
- Operation.java contains operations defined for 2PLDDB.
- Transaction.java contains transaction class.
- Comm_Server.java defines Java RMI interfaces for central site.
- Comm_Site.java defined Java RMI interfaces for other sites.
- CentralSite.java contains process for central site. A central site will have a lock manager and a site table storing site ids.
- DataProcessor.java contains data processor that can be called by each other site for reading and writing.
- LockManager.java is an implementation of lock manager. Lock manager contains a lock table and a queue table. This is maintained by central site.
- OtherSite.java contains process for other sites. Each other site will have a transaction manager.
- TransactionManager.java maintains pending transaction history for each other site.
- WriteForGraph.java contains a wait-for-graph structure which central site will use to detect deadlocks.

Data

Correctness

The following test cases are designed to test correctness of the system:

(Note that "transactions_few" refers test cases with small number of transactions, e.g. 3 transactions. "transactions_many" refers test cases with large number of transactions, e.g. 10000 transactions. "transactions_long" means there are many operations in each transaction, e.g. 30 operations in a single transaction.)

- Single other site /few transactions: verify data values in MySQL (transactions_few_1.txt)
- Single other site /many transactions: the other site should not block before processing all transactions; the values in MySQL should keep changing (transactions_many_1.txt)
- Two other sites /few transactions: consider aborted transactions and verify data values in MySQL
- Two other sites /one with few transactions and one with many transactions: the few transactions site should block and wait for new transactions, while the many transactions site should keep processing
- Two other sites /one with many transactions and one with long transactions: observe which site is taking control of locks for longer time. (transactions_few_1.txt)
- More than two sites: the system should keep running. The purpose is to test deadlock resolver is working fine.

Time to Resolve Deadlocks

Since the efforts to run complete deadlock experiments are huge, the performance is evaluated in a certain scenario: Windows 8, CPU i5-3230, Java RMI for process communication and transactions with four operations and SQLite.

	2 Other Sites	3 Other Sites
Average	16.14	17.92
Standard Deviation	4.468	6.942

Figure 11: Time to detect and resolve deadlocks (milliseconds)

I tested the same system with MySQL database. Surprisingly, the time to resolve deadlocks with the above configuration is about 2 ~ 3 milliseconds. So the database connection takes a lot of time in the above experiment. Therefore, the time to resolve deadlocks are insignificant compared to other factors such as communication delays in a light-traffic transaction system.

Starvation

In the meanwhile, there is a “starvation” experiment of deadlocks. “transactions_many_2.txt” is a test case with many frequent shared variables. If three or more other sites run with this test cases, there will be more than one cycles in wait-for-graph. The speed of cycle formation is faster than cycle removing. We observe the following results:

Other site 1: abort, abort, abort …

Other site 2: abort, abort, abort …

Other site 3: abort, abort, abort …

In this scenario, the system behaves really badly. More discussion for this issue is discussed in “Experience” section.

Experience

During the development of 2PLDDB, I learned a lot of new knowledge such as system concurrency. I would like to share some difficulties and experience in this project in this section.

There are many other concurrency issues in a multi-threaded systems besides transaction concurrency issues. Transaction concurrency is just one of concurrency problems. I have a good experience to implement a complete concurrent system in Java. Our system deals with communication synchronization, task synchronization. An interesting example is how to block or unblock other sites. After other site sends a lock request, the lock might be put in waiting queue. The other site must be blocked at this time. In my implementation, I change the other site's status to blocked and call wait() function. After some time, the central site grants the lock and sends back an unblocked message. Then I call notifyAll() function to unblock the other site.

For the deadlock detecting problem, I use an on-fly method to construct wait-for-graph. I construct a new wait-for-graph during deadlock detection and depose it after a checking. Obviously, it is simple to implement. However, I think it might not be the most efficient method in a real complex distributed database systems. Dynamically maintaining a wait-for-graph can immediately detect a deadlock. But you have to add/remove edges and transactions very carefully during server runtime. It is more difficult to implement and brings some other issues such as failure cases.

Another issue is about starvation. In 2PLDDB, deadlocks are checked every predefined time interval. Once cycles are detected, one cycle is removed. If there are more than one cycles, the system has to wait for another time interval to remove the second edge. The reason of this algorithm is simpler to implement and works in most normal scenarios. However, this algorithm is working badly in some heavy-traffic scenarios. It results in repeated abortions which is called starvation. In this case, a faster deadlock remover with less overhead might be a little helpful.

Conclusion

Concurrency control aims to ensure the database is consistent while maximizing degree of concurrency. There are generally two types of concurrency controls: optimistic and pessimistic. Two phase locking is a pessimistic concurrency control methods. It prevents undesired or incorrect operations on shared objects by other concurrent transactions.

Although two phase locking is a sufficient condition of serializability, a weakness is it may be subject to deadlocks. Deadlocks occur when two or more competing transactions are waiting for each other to release locks. Deadlocks can be monitored by detecting cycles in wait-for-graph. They can be resolved by aborting transactions which form cycles. However, the system might perform badly when the speed of cycle formations is faster than cycle removing.

The two phase locking implemented in 2PLDDB is a centralized version where a central site is maintaining a global lock manager. In the project, two phase locking is implemented in a distributed servers' environment. The locks' acquiring and releasing procedures are visible. Moreover, deadlocks are detected and resolved in a predefined time interval in 2PLDDB. The project has an extensible framework. In current release, it is focus on concurrency control.

From the experiments, we discover the time to resolve deadlocks are insignificant compared to other factors such as communication delays in a light-traffic transaction system.

However, there is much future work in order to develop a complete distributed database based on 2PLDDB. The design of storage can be extended to distributed storage. This requires future work on commit protocols such as two phase commit or three phase commit. In the meanwhile, the system assumes no failures. In real distributed database, site failure and recovery are very important aspects of a reliable distributed system. Moreover, more complicated concurrency controls should be implemented as well. Then an adaptive distributed database can be developed.

References

Oszu, T., & Valduriez, P. (2012). *Principles of Distributed Database Systems.* Prentice Hall.

Two Phase Lcoking. (n.d.). Retrieved from Wikipedia: http://en.wikipedia.org/wiki/Two-phase_locking