A Performance Study of Isolation Levels in SQL Server 2008

Project Report

Adrian Kwok
adriank@sfu.ca
SFU ID #200136359

CMPT 740
Professor Luk
April 13th, 2011

1. INTRODUCTION

The primary concern of this project is to empirically show the performance repercussions of choosing an isolation level in SQL Server 2008 R2 that constricts the level of concurrency more-so than necessary. For example, in an application where no unrepeatable reads are possible in the first place, what are the penalties of choosing an isolation level of *REPEATABLE_READ* or higher? While many sources stress that a transaction's isolation level should be chosen with care due to possible performance degradation with concurrent transactions, it is not immediately clear whether this would result in a system that is marginally slower, or orders of magnitude slower in real world scenarios.

In this vein, this project utilizes a simple database of tuples in SQL Server and a highly concurrent benchmarking application written in Java in an attempt to demonstrate the consequences of setting "incorrect" isolation levels to eliminate the read anomalies specified in SQL-92**[1]**. As a side-effect of the results presented in this paper, we also show that snapshot isolation (i.e. multi-version concurrency control) performs admirably under many different scenarios – sometimes as well as the lowest isolation level – while still being able to eliminate most anomalies.

2. METHODOLOGY

Since the goal of the project was to demonstrate the performance of the isolation levels offered in SQL Server 2008 R2 under varying real world scenarios, there were five important aspects that were crucial during the creation of the experiment environment:

1.) The benchmark application (i.e., the client) should utilize multithreading to emulate a large number of transactions being sent simultaneously to the server.
2.) To ensure the integrity of the study's results, the benchmarking application should be carefully engineered in such a way that a minimal amount of processing be done client-side during the benchmark.
3.) The study should encompass all three major read-anomalies that SQL Server's five isolation levels seek to address – specifically, *Dirty Reads*, *Unrepeatable Reads*, and *Phantoms* should be shown to be impossible under certain `READ_UNCOMMITTED`, `READ_COMMITTED`, `REPEATABLE_READ`, `SERIALIZABLE`, and `SNAPSHOT` isolation levels. This will require after-benchmark analysis.
4.) The study should show the server's performance under a vast number of concurrent transactions varying from a simple single-threaded architecture (i.e. 1 transaction at a time) to as many concurrent transactions as the server will allow.
5.) The study should utilize transactions that are somewhat realistic and should, at the very least, provide a range of different possible workloads for each transaction.

After much deliberation, four parameters which would likely heavily influence the overall running time of a benchmark were defined as follows:

1.) The number of concurrent transactions.
2.) The distribution of "reader" transactions versus "writer" transactions.
3.) The complexity of each transaction – more specifically, the complexity of each *write* transaction.
4.) The total number of transactions performed per benchmark.

How all of these aspects were covered during the process of performing this study will now be discussed in the following two subsections.

2.1 METHODOLOGY: CLIENT

The benchmarking client is a Java (J2SE) application utilizing Microsoft's JDBC driver**[8]** to connect to the provided SQL Server 2008 R2 instance. The client relies on Java's `ExecutorService`**[11]** API to create and maintain a thread pool consisting of X threads: each thread is a `runnable TransactionTask` object, which just sends a predefined SQL query to the server. To satisfy the first requirement discussed previously, every `TransactionTask` is initialized and set up prior to the start of the thread pool, and absolutely no additional work other than executing the already initialized query and storing the query's results (if applicable) is done while the thread pool is running.

The client first initializes a predetermined number (1000 in benchmarks discussed in this paper) of `TransactionTask` objects. Since the benchmark relies on a mixture of "reader" transactions and "writers" to show the anomalies in question, a weighted coin is flipped during the initialization of each `TransactionTask`, specifying whether it will eventually be sending a *SELECT* query or an UPDATE query to the server. The client then adds all of these `TransactionTask` objects to the `ExecutorService`'s thread pool, and starts the pool. Each `TransactionTask` that is currently in the thread pool is then executed, which results in queries being sent to the SQL Server database – when a `TransactionTask` in the pool is completed, another `TransactionTask` enters the thread pool in its place. The client then waits until all `TransactionTask` objects are done executing, and afterwards analyses the results if requested by the user.

High-level pseudocode descriptions of the client and each `TransactionTask` are provided below:

```
0   Client
1   {  Initialize a thread pool of size X threads – X is the number of concurrent xacts.
2      Create an array of TransactionTask objects of size Y called Tasks.  This is the total #
               of xacts performed in the benchmark.
3      Initialize all objects in Tasks by flipping a weighted coin and assigning it
               either a "select" query or "update" query.
4      Initialize the thread pool.
5      START the benchmark timer.
6      Add all TransactionTasks in Tasks into the thread pool, executing them automatically.
7      SLEEP until all TransactionTasks are completed.
8      STOP the benchmark timer.
9      Analyze results of each TransacionTask (if specified to do so by the user).
10  }
```

```
0   TransactionTask (on thread execution)
1   {  Create a new connection to the DB.
2      Execute SQL query(or queries) defined by client during initialization.
3      Store query results into self, if the query is a SELECT query.
4      Close connection to the DB.
5   }
6   TransactionTask (on results analysis)
7   {  Analyse results depending on read anomaly requested.
8   }
```

Of course, there is additional code written in the client to automate the collection of benchmark results (e.g. automating the number of iterations performed, the number of concurrent transactions, etc.). Furthermore, the definition of each SQL query in a `TransactionTask` depends on the type of anomaly being tested. The SQL queries are described in further detail in the following database methodology section.

2.2 METHODOLOGY: DATABASE

The experiment's SQL Server database consists of a simple table "Experiment" consisting of 1000 tuples. Each tuple contains fields "ID" and "Value", where "ID" is an indexed unique key. Prior to each benchmark, the table is prepared such that there are exactly 1000 rows, with each "ID" ranging from 1 to 1000, and every "Value" field set to 0.

Since the actual workload per benchmark needs to vary from instance to instance for result gathering purposes, this is accomplished by requesting writer transactions to update or insert a prespecified number of random rows. Prior to the transactions' execution, these query statements are precompiled by the client; this is to offload processing done during the actual benchmark. As such, the random rows that each `TransactionTask` writer is meant to update are decided prior to the benchmark. The number of random rows that are updated is equivalent to the parameter $xact\_complexity$ ($0 \leq xact\_complexity \leq 1$) multiplied by the number of rows in the table.

Dirty Read

Reader TransactionTask	Writer TransactionTask
1 BEGIN TRANSACTION 2 SET ISOLATION LEVEL 3 Results = *SELECT* Value FROM Experiment 4 END TRANSACTION	1 BEGIN TRANSACTION 2 SET ISOLATION LEVEL 3 UPDATE Experiment SET Value=1 WHERE ID=RANDOM_ID1, ID=RANDOM_ID2, ID=RANDOM_ID3, … 4 ABORT 5 END TRANSACTION

***Table 1**: SQL Queries used to demonstrate "Dirty Read" anomalies.*

As all "Value" fields in the Experiment table are set to 0 initially, determining the number of dirty reads is trivial: since all updates are aborted, each TransactionTask only needs to go through "Results" and tabulate values that are equal to 1, as any value not equal to 0 infers a dirty read.

Unrepeatable Read

Reader TransactionTask	Writer TransactionTask
1 BEGIN TRANSACTION 2 SET ISOLATION LEVEL 3 Results1 = *SELECT* Value FROM Experiment 4 Results2 = *SELECT* Value FROM Experiment 5 END TRANSACTION	1 BEGIN TRANSACTION 2 SET ISOLATION LEVEL 3 UPDATE Experiment SET Value=1 WHERE ID=RANDOM_ID1, ID=RANDOM_ID2, ID=RANDOM_ID3, … 4 COMMIT 5 END TRANSACTION

***Table 2**: SQL Queries used to demonstrate "Unrepeatable Read" anomalies.*

Again, determining whether a reader TransactionTask has experienced an unrepeatable read is trivial: it only needs to check if any values for a corresponding row in Result1 are different than that in Results2.

Phantoms

When trying to demonstrate phantom anomalies, each writer needs to prune the database back to the initial 1000 tuples before inserting a random number of rows to the table; otherwise, if the table isn't pruned, it may grow exceptionally large over time and skew results gathered from transactions that are executed last.

The number of rows inserted into the table during each TransactionTask is determined, as before, by the $xact\_complexity$ user parameter.

Reader TransactionTask	Writer TransactionTask
1 BEGIN TRANSACTION 2 SET ISOLATION LEVEL 3 Results1 = *SELECT* Value FROM Experiment 4 Results2 = *SELECT* Value FROM Experiment 5 END TRANSACTION	1 BEGIN TRANSACTION 2 SET ISOLATION LEVEL 3 DELETE FROM Experiment WHERE ID>1000 4 INSERT INTO Experiment (Value) VALUES (1), (1), … , (1) 5 COMMIT 6 END TRANSACTION

***Table 3**: SQL Queries used to demonstrate "Phantom" anomalies.*

Determining whether a phantom anomaly has occurred during a reader TransactionTask's execution is a bit trickier to do efficiently: note that if the number of returned entries in Results1 and Results2 differ, then surely a phantom has occurred. However, if the number of entries in Results1 and Results2 are the same, this doesn't necessarily mean that a phantom hasn't occurred: observe the scenario where 2 rows are deleted and 2 new rows are inserted – although the number of rows are unchanged, the rows themselves have. Thus, if the size of Results1 and Results2 is the same, it is necessary to scan through all items in Results1 to ensure that every row that exists in Results1 also exists in Results2.

3. CHALLENGES

Although theoretically the study should be rather simple to implement and execute, in reality there were many challenges that were encountered during the process of conducting this study. Unfortunately, since the SQL Server instance is provided by CSIL, it is not entirely clear whether the issues faced are due to the server's configuration or due to a deeper underlying problem with SQL Server – although the latter seems unlikely as SQL Server is a commercial, widely adopted enterprise application.

The most notable problem faced was that the provided SQL Server instance has severe issues in dealing with a large number of concurrent transactions; while the client (which was run on a crippled virtual machine, again provided by CSIL) was able to handle up to 2000 threads with ease, any benchmark that tried to send more than 500 concurrent transactions to the database would result in a cryptic SQL exception being thrown:

> *"Transaction X (Process ID X) was deadlocked on lock resources with another process and has been chosen as the deadlock victim. Rerun the transaction"*

When this exception is thrown, all active transactions immediately throw the same exception repeatedly. Similarly, the database grinds to a halt: the activity monitor in SQL Server Management Studio becomes unresponsive and self terminates, and reconnection attempts fail. Initially, my interpretation of this error was that a deadlock was occurring due to the semantics of the transaction itself; that is, a transaction with a lock on row X may have requested, say, a row lock on row Y, while another transaction holding a row lock on row Y may be requesting a lock on row X, resulting in a deadlock. However, after much experimentation –

eventually even disabling the use of locks with the NOLOCK SQL query hint – the exception still surfaced even with the simplest of transactions.

After much research and troubleshooting, the documentation for the exception was examined more carefully**[9]** – it seemed that the exception was a "catch-all" exception on **all** resources that can deadlock. Most notably, the same exception is thrown regardless of whether the system is deadlocked due to locks (which is resolved rather quickly by SQL Server) or if the deadlock is due to more complex resources such as a lack of available memory or process worker threads (which, for obvious reasons, takes much longer to resolve). Surely enough, the activity monitor in SQL Server Management Studio, when filtered properly, showed that the deadlock resource in question was related to the number of available threads **[Figure 1]**.

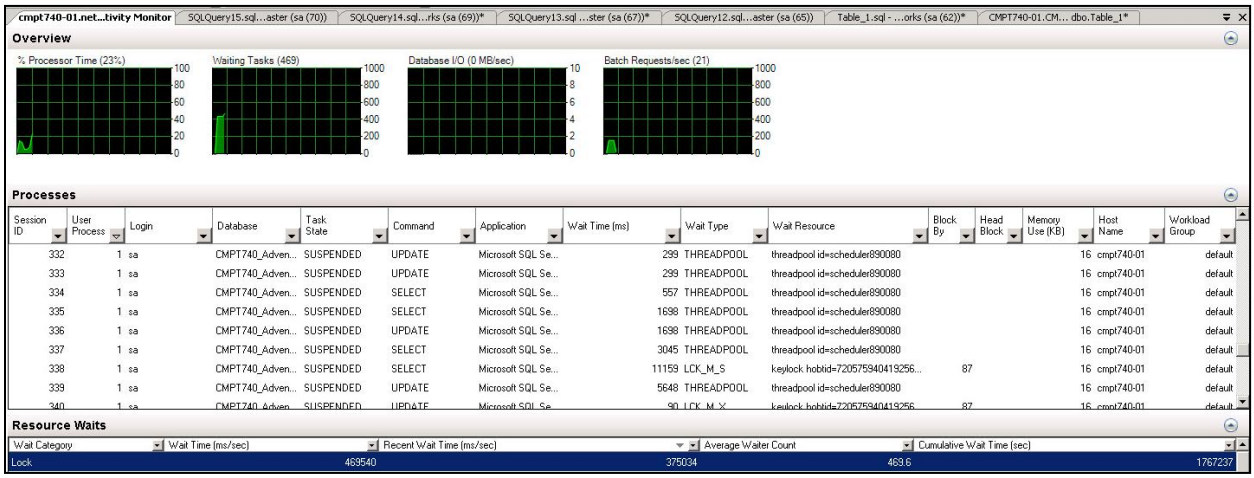

Figure 1: *Server deadlock due to exhausted resources shown via SQL Server Management Studio.*

To confirm this hypothesis, a deadlock trace was conducted via SQL Server Profiler, which resulted in the deadlock graph shown in **[Figure 2]**. The graph shows a singular victim process being deadlocked due to a thread pool, attached to hundreds of other processes.

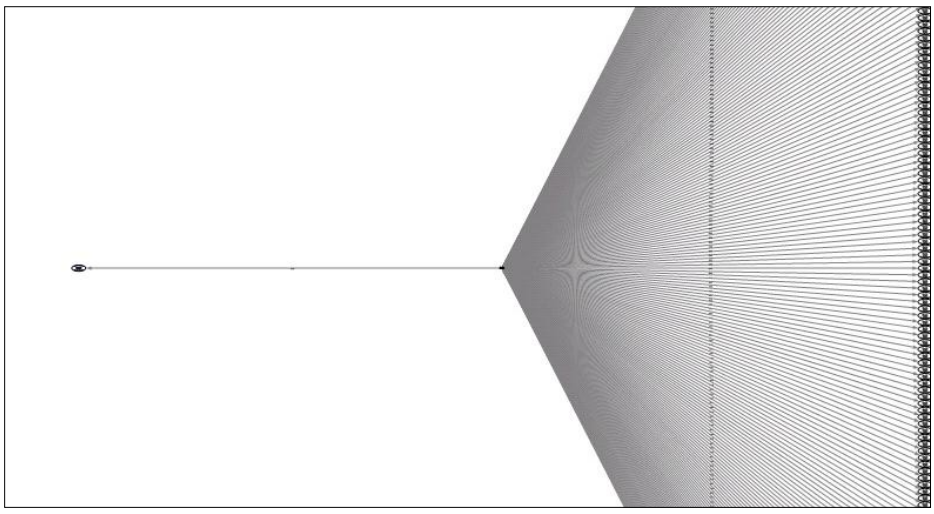

Figure 2*: Server deadlock due to exhausted resources, shown via deadlock graph from SQL Server Profiler.*

Another problem arose while trying to gather results from the benchmarking client: as the client was run on a virtual machine provided by CSIL, the results generated were prone to severe fluctuations. An attempt to mitigate these fluctuations is shown in the Results section of this paper – each singular benchmark would be

run numerous times, and an average value would be used instead of a singular result. However, this is only a quick fix to address a more serious underlying issue with the experiment environment.

The cause of these fluctuations seems to be caused by the lack of available system resources on the virtual machine. When the client is running a benchmark with a moderate number of concurrent transactions (e.g. around 300), the CPU usage on the system is fully saturated – however, Java.exe only utilizes around 60% of the CPU, and the remaining cycles are consumed by *sqlservr.exe*, which implies that both the benchmarking client and SQL Server instance (or a large component thereof) are running on the same virtual machine. As well, due to the low system resources available to the client, each benchmark would take many hours to complete (the 'heavy workload' benchmarks took upwards of 12 hours to complete on average), and any other application running on the same machine would easily influence the test results. Ideally, a dedicated box should be used for the client, but unfortunately this was not available, even after explicitly requesting so.

While these configurations lead to less-than-ideal experimental conditions, the results gathered do seem to indicate a trend that is consistent among several different benchmarks. Fortunately, the Java code written for this experiment (and subsequent graphs) can be easily be conducted on a more stable system if the need arises to confirm the findings in this paper.

4. Results and Discussion

As discussed in the Section 2 previously, there were four parameters defined that may influence the time it takes for each benchmark to complete: the number of concurrent transactions (num_xact), the distribution of reader and writer transactions (reader_dist), the complexity of each update transaction (num_complexity), and the number of overall transactions performed (num_tasks).

For each anomaly being tested, five isolation levels are benchmarked with varying numbers of concurrent transactions. The SQL queries used during each anomaly test is defined in the previous "Methodology" section. To gather meaningful results from each isolation level benchmark and to eliminate outliers, each benchmark is iterated 20 times, and the results are denoted as singular points on a graph. A 6^{th} degree polynomial curve is used as the best-fit for these points to facilitate readability.

Of note, however, is that the overall number of transactions performed in these benchmarks is fixed to be 1000, as increasing this number serves no benefit other than increasing the accuracy of the results – something that is already ensured by running the benchmarks multiple times in succession.

4.1 Dirty Read Scenario Benchmark

A dirty read is defined as a transaction that reads changes made by another transaction that has not yet been committed. First, to confirm that dirty reads cannot occur in isolation levels higher than *READ_UNCOMMITTED* as stated in **[7]**, a simple benchmark utilizing queries from **[Table 1]** which varies the number of concurrent transactions from 1 to 401 in steps of 10 is performed, The results are shown in **[Figure 3].**

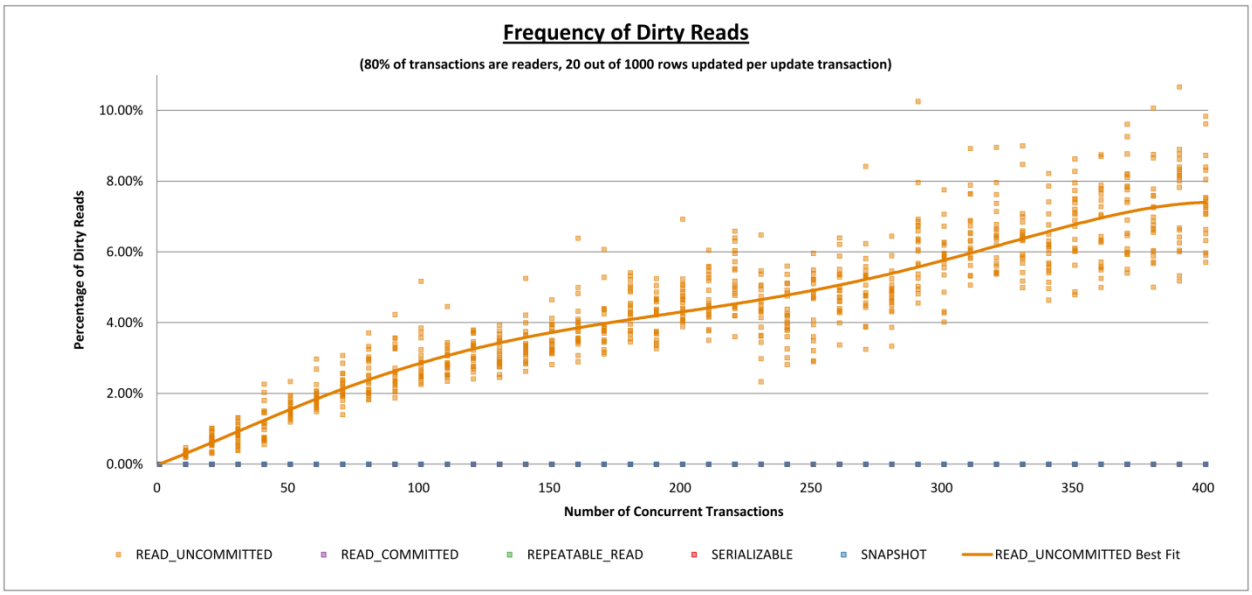

Figure 3: *Frequency of the "Dirty Read" anomalies occurring under various isolation levels.*

As clearly shown, dirty reads occur more frequently as the number of concurrent transactions increase, and only under *READ_UNCOMMITTED*; that is, choosing an isolation level higher than *READ_COMMITTED* serves no additional benefit in eliminating dirty reads. To illustrate the performance degradation from choosing an isolation level higher than *READ_COMMITTED*, a low writer complexity benchmark is performed and its results are shown in **[Figure 4]**.

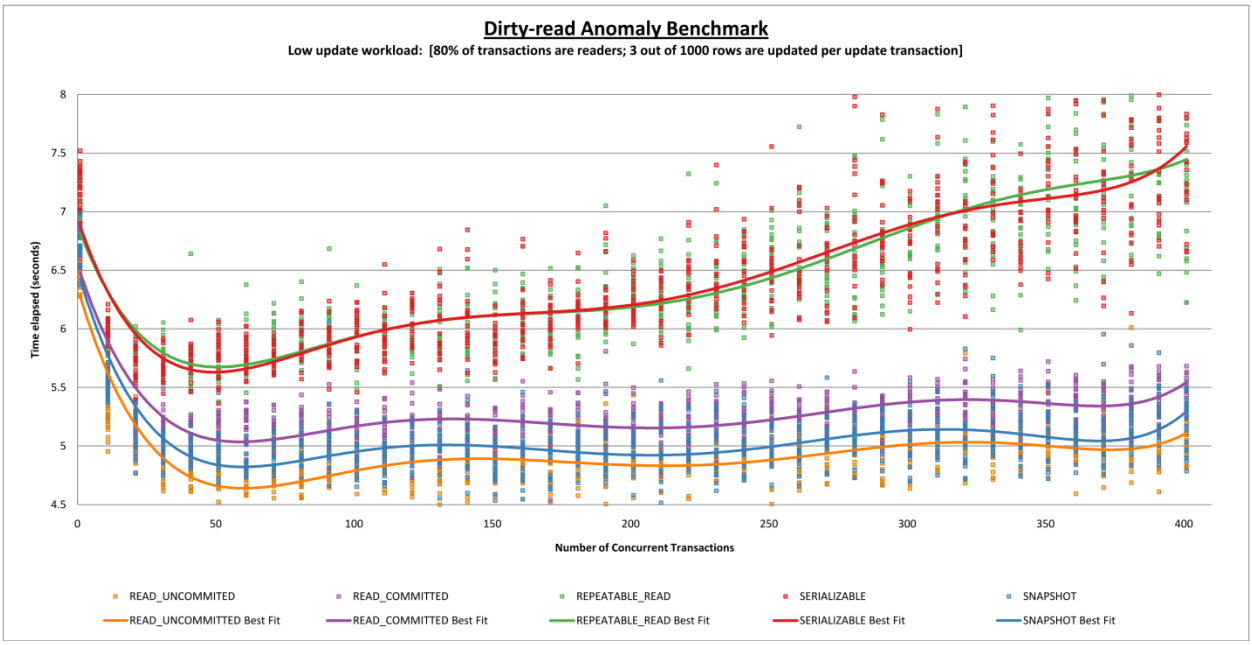

Figure 4: *Performance of different isolation levels with "Dirty Read" scenario transactions – low workload.*

[Figure 4] shows a significant performance penalty (≈ 36% slower than *READ_COMMITTED* with 400 concurrent transactions) when using either *REPEATABLE_READ* or *SERIALIZABLE* to eliminate the anomaly. This is likely due to the way locks are handled under these isolation levels: in *REPEATABLE_READ* and

SERIALIZABLE, all S- and X-locks are held until the end of a transaction, whereas in *READ_COMMITTED* these locks can be released at any time during a transaction, allowing for increased concurrency.

For example, the "*SELECT* Value" reader query will be able to relinquish locks after analyzing each row in *READ_COMMITTED*, allowing concurrent write transactions to write to the row even when the transaction is not completely finished. However, in *REPEATABLE_READ*/*SERIALIZABLE*, the reader transaction must hold the lock for all rows until it is finished with the transaction, meaning that all concurrent write transactions are blocked until the entire reader transaction is completed. Do note that this clearly means *REPEATABLE_READ*s are very possible under *READ_COMMITTED* – if a writer updates a row and commits after the row's lock is released by a reader transaction, the reader will read a different value for that row if the same query is executed again.

What is surprising, however, is that *SNAPSHOT_ISOLATION* actually results in better performance than *READ_COMMITTED* while still being able to prevent dirty reads from occurring – and as shown in later sections, is also capable of preventing many other anomalies efficiently. This will turn out to be a continuing trend with multi-version concurrency control.

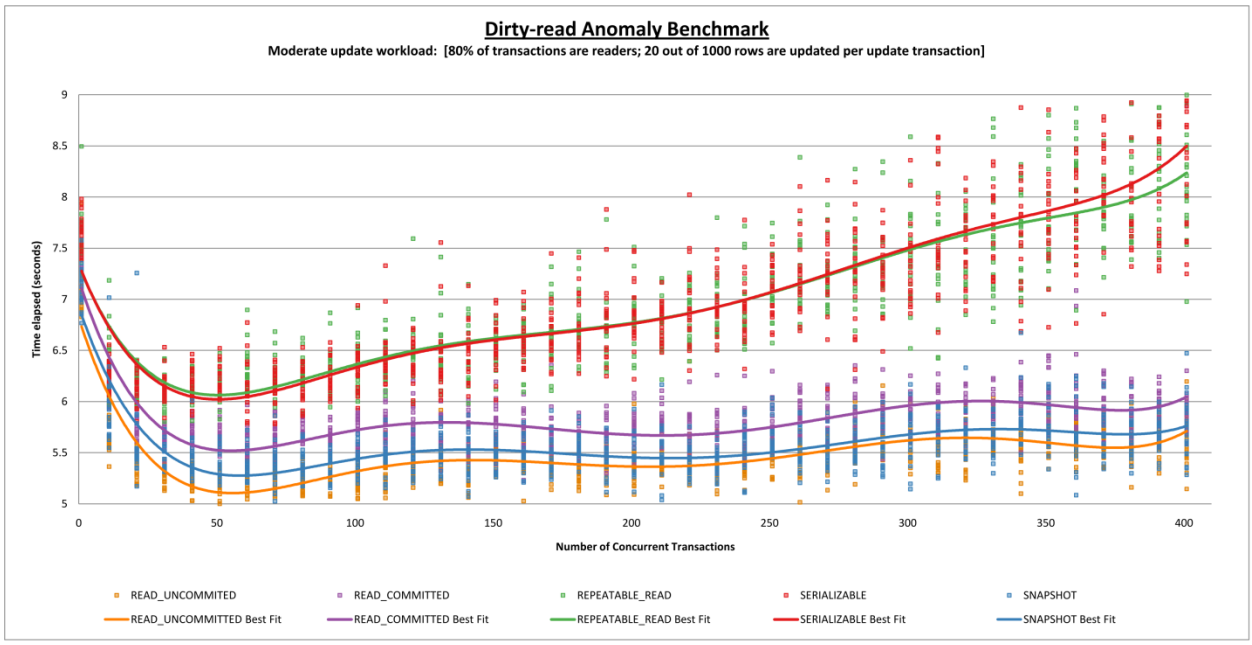

Figure 5: *Performance of isolation levels with "Dirty Read" scenario transactions – medium workload.*

For completeness, **[Figure 5]** shows the same benchmark, but with a higher workload per writer transaction (each transaction updates 20 rows as opposed to 3 rows previously); both *REPEATABLE_READ* and *SERIALIZABLE* take almost 44% longer to complete the benchmark than *SNAPSHOT_ISOLATION*. These results are similar to those from **[Figure 4]**. However, if the workload is increased even further – that is, each writer updates to 20% of all available entries in the table) – the difference between isolation levels diminishes, as shown in **[Figure 6]**. This is likely because as the number of row updates increases per transaction, so do the number of blocked transactions due to X-locks being held (i.e. write/write conflicts) – thus, the performance penalty from these blocked transactions start dominating the performance improvements given by simply improving read/write concurrency in *READ_COMMITTED* and *REPEATABLE_READ*.

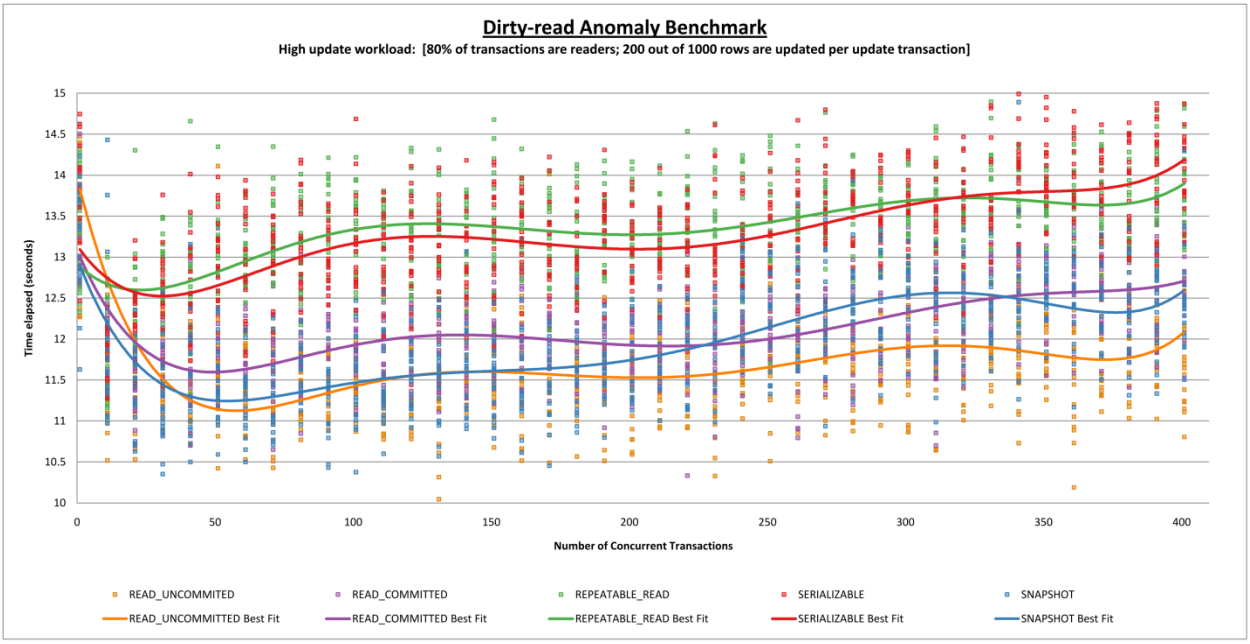

Figure 6: *Performance of different isolation levels with "Dirty Read" scenario transactions – high workload.*

These figures also highlight an important point: while initially increasing the number of concurrent transactions (e.g. from 1 transaction to 40 concurrent transactions) does speed up the time it takes to complete the benchmark, increasing it any further beyond a specific point actually results in gradually worse performance.

4.2 UNREPEATABLE READ SCENARIO BENCHMARK

Similar to the benchmarks for the dirty read scenario, this section shows that unrepeatable read anomalies can occur in isolation levels lower than *REPEATABLE_READ*, and the overall performance of each isolation level under a variety of workloads. As the queries are very similar to those in the previous section – that is, the only difference is that each reader now does two consecutive *SELECT*s as opposed to one, and each writer commits its transaction instead of aborting – and the performance differences in isolation levels are due to the way the different isolation levels handle locks as explained in Section 4.1, these results will only be briefly discussed.

As shown in **[Figure 7]**, unrepeatable reads are only possible under the *READ_UNCOMMITTED* and *READ_COMMITTED* isolation levels. At first glance, the results may seem a bit surprising – although both isolation levels do not prevent the anomaly from occurring, clearly *READ_COMMITTED* results in fewer anomalies. However, note that this is likely because in *READ_UNCOMMITTED*, even writes from transactions that have not committed yet (e.g. still in progress) will be read by reader transactions, as opposed to in *READ_COMMITTED* where reader transactions will never do a read while another write transaction is acting upon it, and will be blocked until a write is fully complete. This correlates to a higher likelihood that a reader transaction will read *some* modified data under *READ_UNCOMMITTED*.

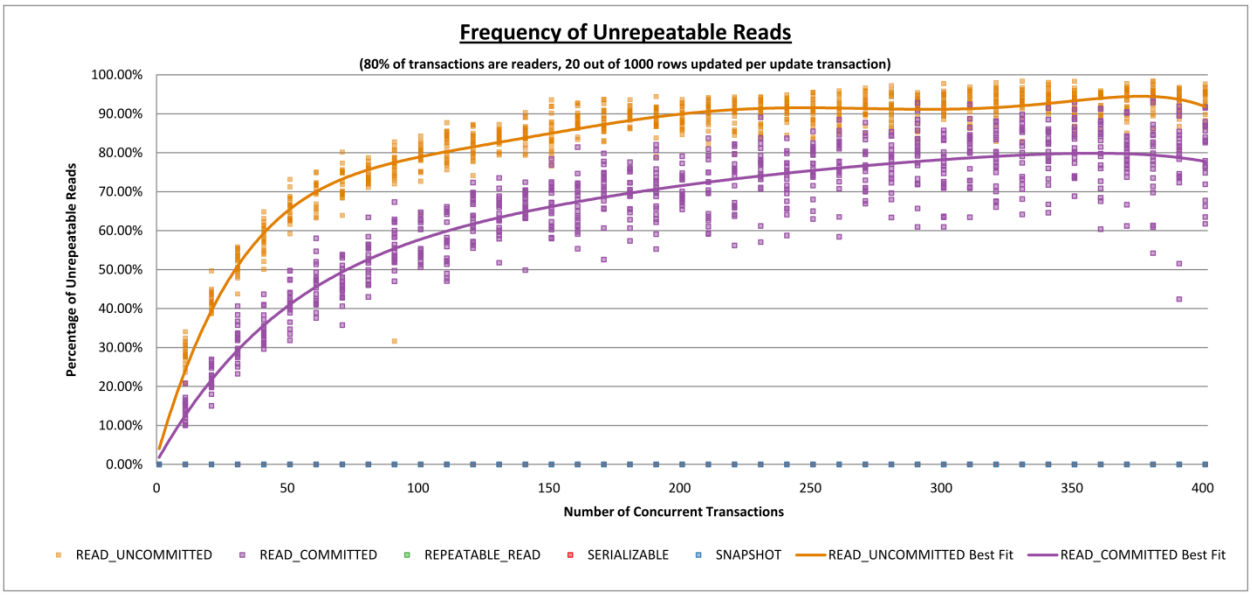

Figure 7: *Frequency of "Unrepeatable Read" anomalies occurring under various isolation levels.*

As with before, **[Figures 8, 9, and 10]** show the performance differences between isolation levels under varying writer workloads.

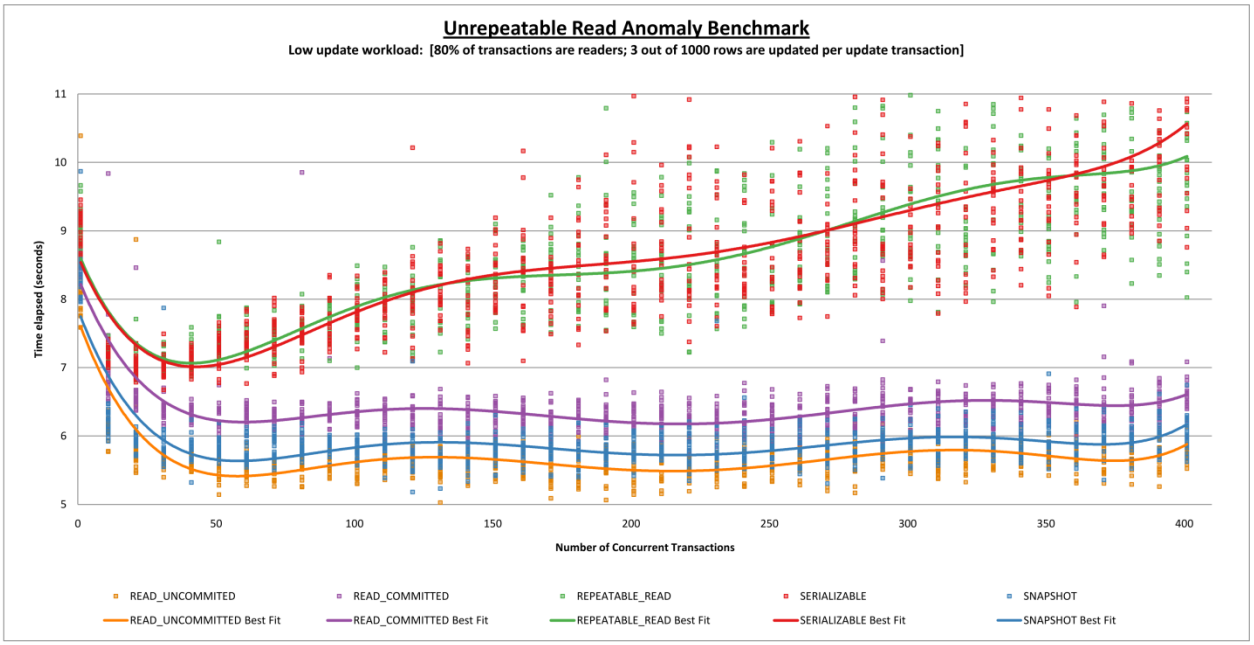

Figure 8: *Performance of isolation levels with Unrepeatable Read scenario transactions –low workload.*

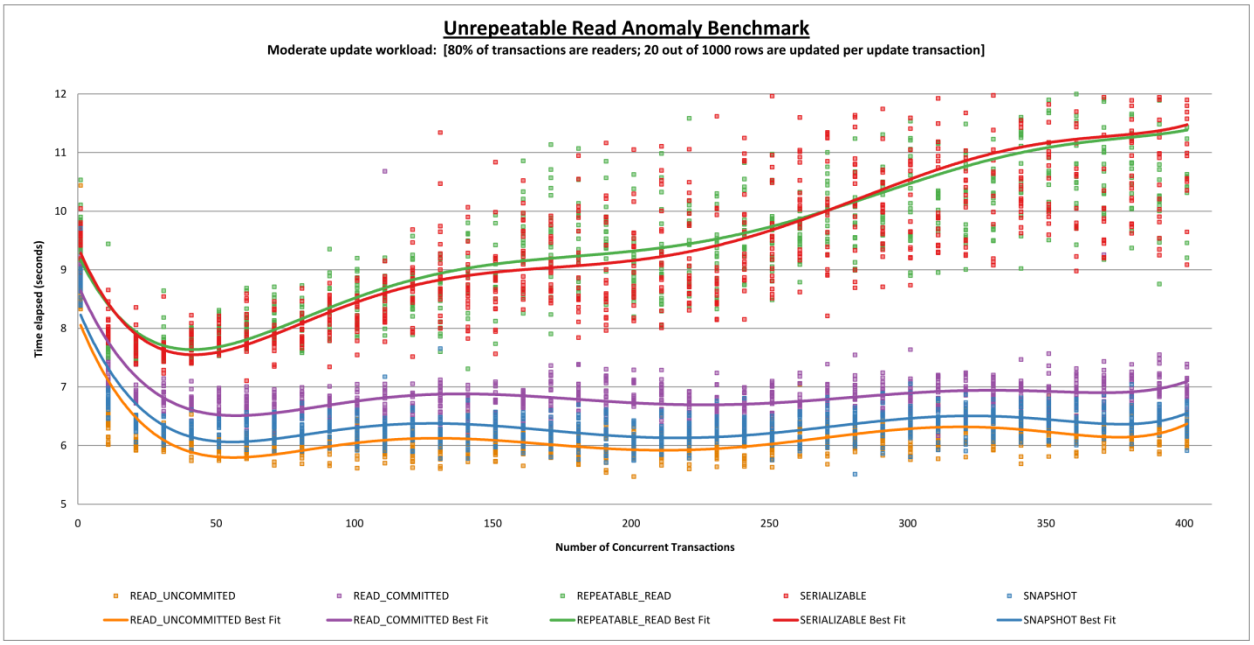

Figure 9: *Performance of isolation levels with Unrepeatable Read scenario transactions –medium workload.*

Of particular note is that in **[Figure 9]**, choosing *REPEATABLE_READ* or *SERIALIZABLE* to eliminate unrepeatable reads takes almost 74% longer to complete the benchmark (as opposed to 44% in Section 4.1 under the same workload) over *SNAPSHOT_ISOLATION* for 400 concurrent transactions. This is likely due to the inclusion of two full table scans in the reader transaction as opposed to only one in the previous section.

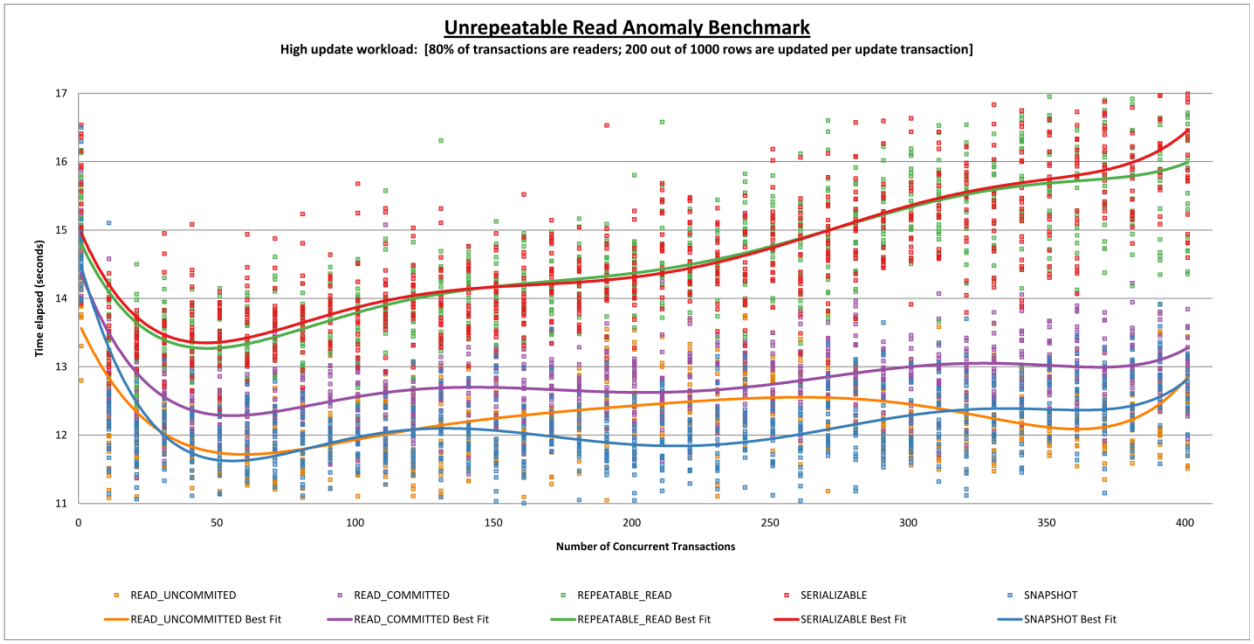

Figure 10: *Performance of isolation levels with Unrepeatable Read scenario transactions –high workload.*

As in Section 4.1, as the workload for each writer transaction increases, the blocking due to X-locks (or in the case of *SNAPSHOT_ISOLATION*, transaction aborts) dominates the improvements in concurrency between readers and writers, and as such *REPEATABLE_READ/SERIALIZABLE* only takes roughly 28% longer to complete the benchmark over *SNAPSHOT_ISOLATION*.

4.3 PHANTOM SCENARIO BENCHMARK

Phantom anomalies are induced via the SQL queries shown previously in Section 2.2; that is, if an *INSERT* operation by any writer transaction happens in-between any two *SELECT* statements in a reader transaction, a phantom anomaly will occur. Unlike in the previous section, the benchmark results for these phantom scenarios are much more interesting, and show a greater diversity in both occurrence rates of anomalies and the performance differences between varying isolation levels.

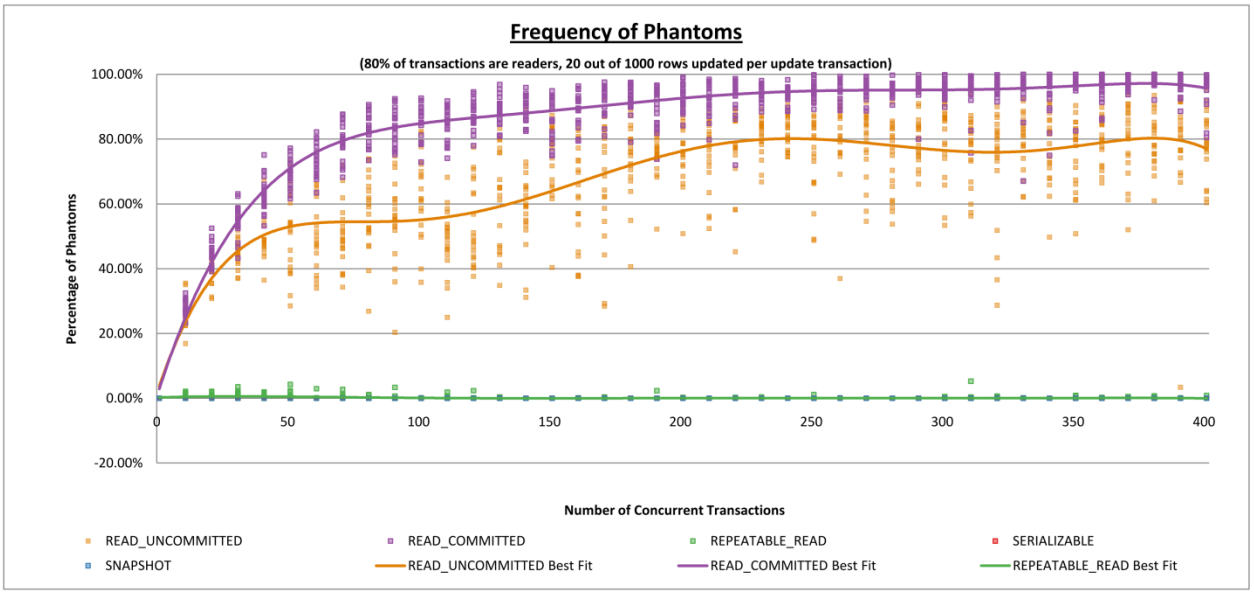

Figure 11: *Frequency of "Phantom" anomalies occurring under various isolation levels.*

As shown in **[Figure 11]**, phantom anomalies occur more often in *READ_COMMITTED* than in *READ_UNCOMMITTED* – this is likely since reader *SELECT*s are never blocked in *READ_UNCOMMITTED* (i.e. both *SELECTs* can be completed very quickly*)* thus meaning that the likelihood a row would be inserted between the two *SELECT*s will be less than in *READ_COMMITTED*, where a *SELECT* may be blocked due to a writer transaction.

However, a more surprising result is that while phantoms *are* shown to be possible under *REPEATABLE_READ*, they are nowhere near as prevalent as in *READ_UNCOMMITTED* and *READ_COMMITTED* – the reasoning for this is that it's quite rare that a row could be inserted between the first and second *SELECT*s in the reader transaction in *REPEATABLE_READ*, since a shared lock is held until the end of a transaction for the first *SELECT* statement. If the first *SELECT* statement is on a table with more than 1000 entries, this implies that **all** other concurrent writer transactions that are attempting to execute the first query (a *DELETE* transaction) will be blocked. Thus, this implies that for a row to be inserted between a reader transaction's first and second *SELECT* queries, a writer's *DELETE* must be executed first, *then* a reader's first *SELECT*, and *then* the writer's *INSERT*. However, a similar blocking scenario happens here: since there are many more readers than writers, and the readers can all share the same lock, the chances of this happening are slim as *DELETES* with ID > 1000 will be continuously blocked if the table has more than 1000 entries – and by the time it's unblocked, it's likely a reader transaction's second query (*SELECT*) will have been executed already as it shares the same S-lock as the first *SELECT* query.

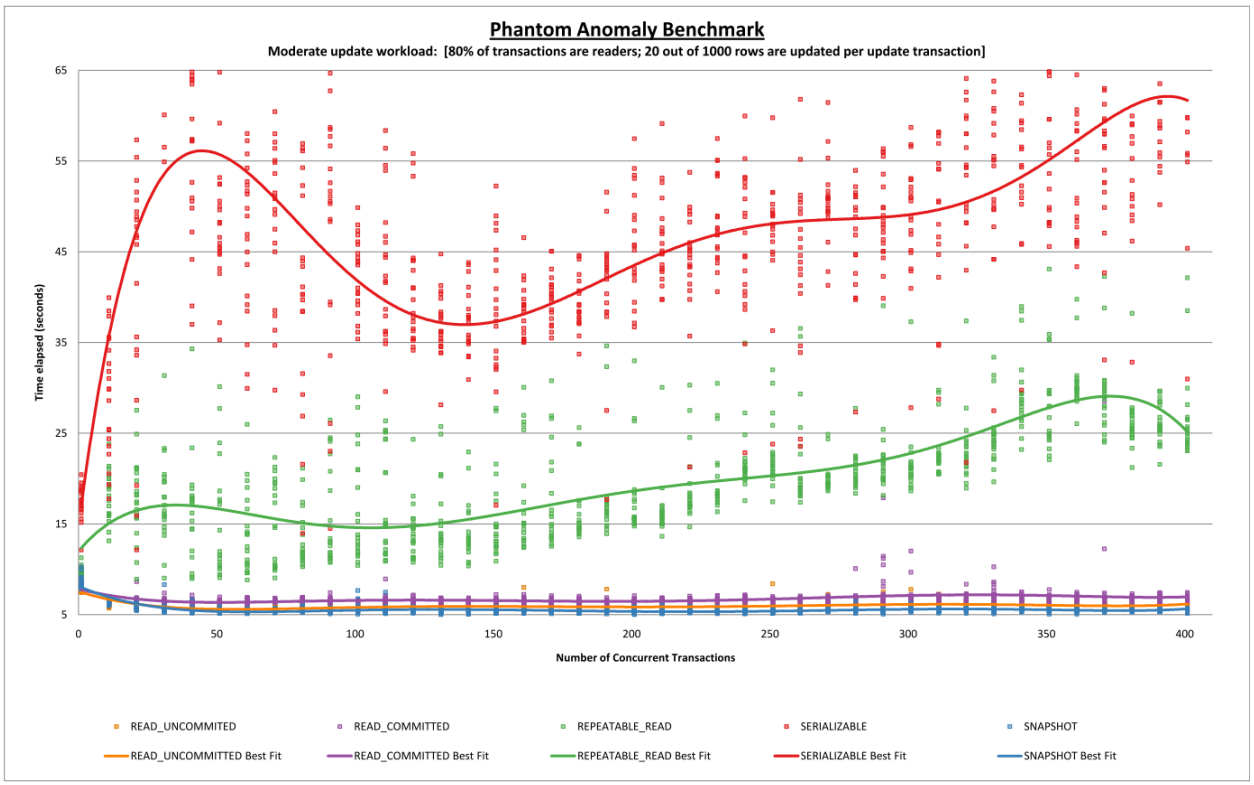

Figure 12: *Performance of isolation levels with Phantom scenario transactions –moderate workload.*

[Figure 12] shows the vast difference in benchmark completion times between *SERIALIZABLE, REPEATABLE_READ* and the other isolation levels. One thing of particular interest is that while *REPEATABLE_READ* does not completely prevent phantom anomalies from occurring, it still takes a considerably longer time to complete the benchmark when compared to *READ_COMMITTED* and *READ_UNCOMMITTED*. This is likely due to two reasons: by requiring S- and X-locks to be held for the full duration of a transaction, there is considerably less concurrency for similar reasons shown via the discussion in Section 4.1. However, another interesting finding is that *REPEATABLE_READ* was the only isolation level which resulted in frequent deadlocks as the number of concurrent transactions increased: although these deadlocks were resolved rather quickly (as they are actual deadlocks due to lock acquisition and not due to deadlocked resources), they still noticeably influenced the overall results, as can be shown by the many green outlier points in **[Figure 12]**. These deadlocks did not occur under *SERIALIZABLE*.

To determine the cause of these deadlocks under *REPEATABLE_READ*, a trace was performed via SQL Profiler and the resulting deadlock graphs are shown in [**Figures 13 and 14].**

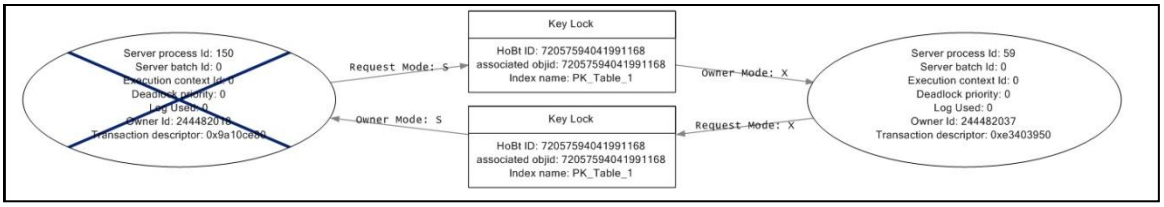

Figure 13: *Simple deadlock occurring under REPEATABLE_READ*

The first deadlock graph likely corresponds to the following scenario: suppose a reader transaction X has an S-lock on some row A during its *SELECT* query. It wishes to acquire an S-lock on some subsequent row B – but row B was recently added by a writer transaction Y and thus Y has an X-lock on row B (keep in mind that in *REPEATABLE_READ* locks are only released at the *end* of a transaction). Now consider if writer transaction Y wishes to *DELETE* row A, and thus requests an X-lock – a simple deadlock as shown in **[Figure 13]** then occurs. The same issue is compounded further with several different transactions in a deadlock as shown in **[Figure 14]**.

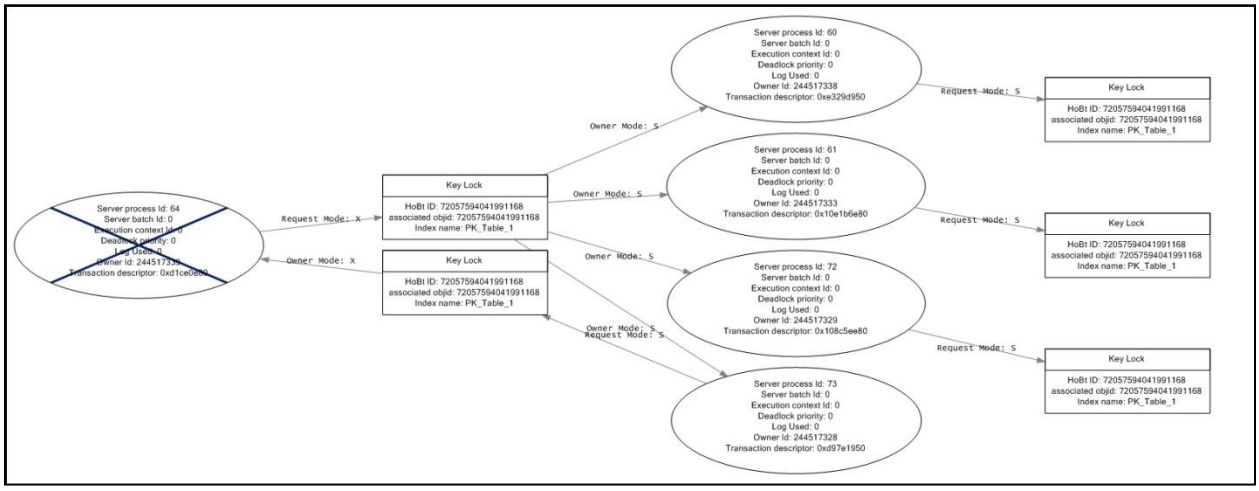

Figure 14: *Compounded deadlock occurring under REPEATABLE_READ*

In any case, **[Figure 11]** clearly shows that only *SNAPSHOT_ISOLATION* and *SERIALIZABLE* are able to eliminate phantom anomalies – and from **[Figure 12]**, the performance difference between the two isolation levels is orders of magnitude apart; in the worst case, *SERIALIZABLE* takes over 11-times longer to complete the benchmark than *SNAPSHOT_ISOLATION*, which is consistently fast regardless of the number of concurrent transactions. This is likely because *SERIALIZABLE* assigns range locks on "ID", causing almost every reader transaction (which needs to access the whole possible range of "ID") to be blocked whenever a writer transaction is running – as the number of concurrent transactions increases, the time consumed coordinating these blocked transactions dominates. Furthermore, as *SERIALIZABLE* relies on an underlying B-Tree structure to assign and enforce locks, whenever an *INSERT* or *DELETE* is issued, the structure needs to be updated; that is, for an INSERT, the B-Tree's internal nodes may need to be split potentially multiple times, and in a DELETE, the tree may need to be rebalanced, which are both costly operations. This is a big issue in our set of SQL queries as there are numerous *INSERT* and *DELETE* operations.

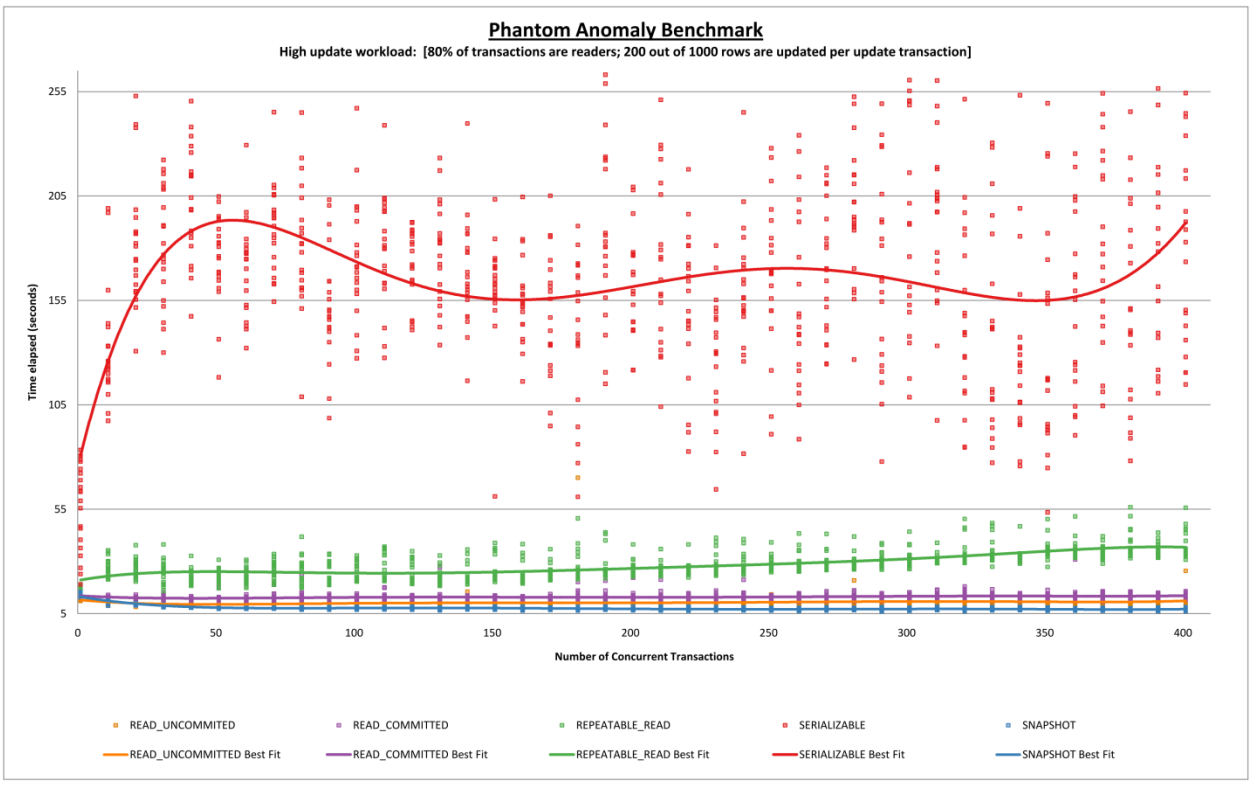

Figure 15: *Performance of isolation levels with Phantom scenario transactions –high workload.*

To further demonstrate the significant performance differences between *SERIALIZABLE* and the other isolation levels for eliminating phantom anomalies, a high-workload benchmark were conducted and its results are shown in **[Figure 15]** – this benchmark took 3091 minutes to complete. In these results, *SERIALIZABLE*'s time to complete the benchmark far exceeds even that of *REPEATABLE_READ* – with an average benchmark taking over seven times as long to complete. Similarly, the performance gap between *SERIALIZABLE* and *SNAPSHOT_ISOLATION* – the only isolation levels which eliminate phantom anomalies – exceeds a factor of twenty.

On the other hand, SNAPSHOT_ISOLATION is only 'slow' when there are write/write conflicts (therefore causing one transaction to rollback and retry) – and this can only happen if a writer transaction manages to choose the same rows to *DELETE* as another transaction, and in this case, first-committer-wins is a very quick solution to this problem (no blocking). All other queries are quickly completed via *SNAPSHOT_ISOLATION*.

Of particular note is that *SNAPSHOT_ISOLATION* gives the most consistent performance out of all isolation levels tested in all of Section 4 – it is often indistinguishable from *READ_UNCOMMITTED* in speed while being able to eliminate all three popular read anomalies.

5. CONCLUDING REMARKS

Although the transactions used in the benchmarks in this paper may not fully reflect real-world scenarios, they still do manage to give a general idea of the performance differences between the various isolation levels offered in SQL Server 2008 R2. Most importantly, they clearly show the performance repercussions of choosing 'incorrect' isolation levels to mitigate anomalies common in concurrent transactions – specifically, as the number of concurrent transactions increase, *SERIALIZABLE* and *REPEATABLE_READ* are significantly slower than *READ_COMMITTED* in eliminating dirty reads, the widely accepted 'ideal' isolation level to do so. However, the differences between choosing *SERIALIZABLE* and *REPEATABLE_READ* to eliminate unrepeatable reads, on the other hand, are negligible; furthermore, with only the isolation levels defined in SQL-92, the SERIALIZABLE isolation level does eliminate phantoms, but at a tremendous cost in doing so.

The most notable and unexpected result from this experiment is that *SNAPSHOT_ISOLATION* eliminates all of the anomalies outlined in SQL-92 while performing consistently quickly in all of the scenarios tested – in all cases, it's comparable to or even faster than *READ_UNCOMMITTED*. Thus, if a user only needs to eliminate these common anomalies and is unsure of what isolation level to choose, it may be ideal to default to SNAPSHOT_ISOLATION for consistent concurrent performance.

However, it is important to note that SNAPSHOT_ISOLATION, unlike the lock-based *SERIALIZABLE* isolation level, is not always serializable. While it does eliminate all of the anomalies listed in SQL-92, there many are other anomalies that exist in multi-version concurrency control protocols**[2]** – the most notable of which is write skew. In most cases, the responsibility to eliminate these anomalies is placed on the programmer to account for it in their application design; indeed, as noted by Fekete et al. in their paper "Making snapshot isolation serializable"**[3]**, the application itself needs to be modified – usually slightly – to eliminate these anomalies. In this vein, work has been done in attempting to automate the discovery of such anomalies as shown by Jorwekar et al.**[5]**.

In the end, while *SNAPSHOT_ISOLATION* works well in eliminating *most* anomalies, there is no 'perfect solution' in eliminating *all* anomalies efficiently and easily, and it is up to the user's discretion to choose an isolation level most suitable for their needs; the experiments presented in this paper have empirically shown the performance implications of these choices.

6. REFERENCES

1.) "Information Technology – Database Language SQL". ISO/IEC 9075:1992, July 30, 1992. Retrieved on April 4, 2011 from http://www.contrib.andrew.cmu.edu/~shadow/sql/sql1992.txt.

2.) A. Fekete, E. O'Neil, and P. O'Neil, "A read-only transaction anomaly under snapshot isolation," ACM *SIGMOD Record, vol. 33, issue 3, pp. 12-14,* September 2004.

3.) A. Fekete, D. Liarokapis, E. O'Neil, P. O'Neil, and D. Shasha, "Making snapshot isolation serializable," *ACM Trans. Database Syst.* vol. 30, issue 2, pp. 492-528, June 2005.

4.) H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil, "A critique of ANSI SQL isolation levels," *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*, pp. 1-10, 1995.

5.) S. Jorwekar, A. Fekete, K. Ramamritham, and S. Sudarshan, "Automating the detection of snapshot isolation anomalies*," in VLDB '07: Proceedings of the 33rd international conference on very large data bases*, pp. 1263–1274, 2007.

6.) "Today's Annoyingly-Unwieldy Term: Intra-Query Parallel Thread Deadlocks", B. Duncan. September 24, 2008. Retrieved on April 4, 2011 from http://blogs.msdn.com/b/bartd/archive/2008/09/24/today-s-annoyingly-unwieldy-term-intra-query-parallel-thread-deadlocks.aspx

7.) "SET TRANSACTION ISOLATION LEVEL (Transact-SQL)", Microsoft Corp., *MSDN Documentation on SQL Server 2008 R2*. Retrieved on April 4, 2011 from http://msdn.microsoft.com/en-us/library/ms173763.aspx

8.) "Overview of the JDBC Driver", Microsoft Corp., *MSDN Documentation on SQL Server 2008 R2*. Retrieved on April 4, 2011 from http://msdn.microsoft.com/en-us/library/ms378749.aspx

9.) "Detecting and Ending Deadlocks", Microsoft Corp., *MSDN Documentation on SQL Server 2008 R2*. Retrieved on April 4, 2011 from http://msdn.microsoft.com/en-us/library/ms178104.aspx

10.) "INF: Analyzing and Avoiding Deadlocks in SQL Server", Microsoft. Article ID: 169960, Revision 3.0. October 13, 2003. Retrieved on April 4, 2011 from http://support.microsoft.com/kb/q169960/

11.) "ExecutorService (Java 2 Platform SE 5.0)", Oracle Corp., Java 2 JDK SE Documentation. Retrieved on April 4, 2011 from http://download.oracle.com/javase/1.5.0/docs/api/java/util/concurrent/ExecutorService.html