Pick & Mix Isolation Levels: Mixed Serialization Graph Testing

Jack Waudby[1], Paul Ezhilchelvan[1], and Jim Webber[2]

[1] Newcastle University, School of Computing,
{j.waudby2,paul.ezhilchelvan}@newcastle.ac.uk
[2] Neo4j, London,
jim.webber@neo4j.com

Abstract. Concurrency control is an integral component in achieving high performance in many-core databases. Implementing serializable transaction processing efficiently is challenging. One approach, *serialization graph testing* (SGT) faithfully implements the *conflict graph theorem* by aborting only those transactions that would actually violate serializability (introduce a cycle), thus maintaining the required acyclic invariant. Alternative approaches, such as *two-phase locking*, disallow certain valid schedules to increase throughput, whereas SGT has the theoretically optimal property of accepting all and only conflict serializable schedules. Historically, SGT was deemed unviable in practice due to the high computational costs of maintaining an acyclic graph. Research has however overturned this historical view by utilising the increased computational power available due to modern hardware. Furthermore, a survey of 24 databases suggests that not all transactions demand conflict serializability but different transactions can perfectly settle for different, weaker isolation levels which typically require relatively lower overheads. Thus, in such a mixed environment, providing only the isolation level required of each transaction should, in theory, increase throughput and reduce aborts. The aim of this paper is to extend SGT for mixed environments subject to Adya's *mixing-correct theorem* and demonstrate the resulting performance improvement. We augment the YCSB benchmark to generate transactions with different isolation requirements. For certain workloads, *mixed serialization graph testing* can achieve up to a 28% increase in throughput and a 19% decrease in aborts over SGT.

Keywords: Databases · Concurrency Control · Weak Isolation · Serialization Graph Testing · Mixing-Correct Theorem · YCSB

1 Introduction

In a database management system (DBMS) concurrency control is responsible for ensuring the effects of concurrently executing transactions are isolated from each other, providing each with the illusion of running alone in the DBMS. This is captured by the correctness criteria *serializability*: if transactions are assumed to be individually correct, then an execution of transactions equivalent to a serial execution of the same transactions guarantees a correct DBMS state [1].

Practically, DBMSs provide *conflict serializability* which is sufficient for guaranteeing serializability. Conflict serializability is based on *conflicts* between transactions. Two transactions conflict if both operate on the same data item and at least one operation is a write. A conflict imposes an order between transactions. If a serial execution can be found that respects the order imposed by conflicts then the execution is conflict serializable [1]. The goal of concurrency control protocols can be reformulated as allowing as many theoretically possible conflict serializable executions without reducing DBMS performance.

Implementing serializable transaction processing efficiently is difficult. The classical approach is *two-phase locking* (2PL) [2]. Such pessimistic approaches perform well under high contention, but suffer when workloads are dominated by read-only transactions. Optimistic approaches such as *timestamp ordering* and *optimistic concurrency control* [3] perform better in low contention workloads, but exhibit many unnecessary aborts when contention is high. However, each approach sacrifices a degree of concurrency to achieve higher throughput, approximating the complete space of conflict serializable executions. To illustrate this, consider concurrent transactions T_W and T_R both attempt to access a single data item, x. Assume T_W holds a write lock on x, and T_R wishes to read x. Under 2PL, depending on the deadlock detection strategy used, one of T_W or T_R will be aborted. Thus, discounting a perfectly legal conflict serializable schedule.

An alternative approach is *serialization graph testing* (SGT) [1,4]. In SGT, the scheduler maintains an acyclic *conflict graph* of the execution it controls. For each operation in a transaction, each conflict is determined and represented by an edge in the conflict graph; a cycle check is then performed before executing the operation, if a cycle is detected then the transaction is aborted. SGT has the theoretically optimal property of avoiding unnecessary aborts, accepting all conflict serializable executions. Despite its advantageous theoretical properties, SGT has seldom been utilized in practical systems owing to the computational costs of maintaining an acyclic graph, notably the cost of cycle checking. Recent work has refuted this perceived wisdom. In [4] it was demonstrated how SGT can be implemented efficiently in a many-core database, offering comparable, and often higher, performance when compared to traditional and contemporary concurrency control protocols.

Despite recent advances, serializable transaction processing performance often remains unsatisfactory for application demands. Another tool at DBMSs disposal to increase performance is to execute transactions at *weak isolation* levels, e.g., Read Committed. Here, the number of permissible schedules is increased at the expense of potentially allowing non-serializable behavior, e.g., *Fuzzy Reads*. Weak isolation is pervasive in real world systems, with most systems offering a range of isolation levels; a comprehensive survey of the isolation levels supported by commercial and open source databases is given in Table 1. A database that allows concurrent transactions to be executed at different isolation levels is said to be *mixed* [5]. For example, transaction T_{RC} can run at Read Committed and transaction T_S at Serializable.

The classical method to implement a mixed DBMS is to opt for a 2PL-variant in which transactions to vary the duration they hold locks [6]. For example, T_{RC} would release read locks on data items immediately after performing the read operation, whereas T_S holds all locks until commit time. This mechanism and others used in mixed DBMSs suffer from the same problem as their serializable equivalents: some valid executions are prevented leading to unnecessary aborts. This begs the question, how can SGT be extended to support transactions executed at weak isolation levels, whilst accepting all and only valid executions? Such an approach would permit higher concurrency and performance. In [5] Adya provides a graph-based system model for defining weak isolation and describes the *mixing-correct theorem*, a correctness criteria for a mixed schedule. This paper presents *mixed serialization graph testing* (MSGT), which accept all valid schedules under the mixing-correct theorem, thus maintaining SGT's property of minimizing aborts. We evaluate MSGT's performance using the YCSB benchmark [7] that has been adapted to generate transactions with different isolation requirements.

The rest of the paper is structured as follows: Section 2 provides an overview of serialization graph testing and discusses the adaptations made in [4] to optimize SGT for a many-core DBMS. Section 3 provides a survey of isolation levels supported by 24 DBMSs highlighting the prevalence of mixed DBMSs and demonstrating the utility of MSGT. Section 4 describes Adya's mixing-correct theorem. Section 5 presents mixed serialization graph testing. Section 6 evaluates MSGT using the YCSB benchmark, before Section 7 concludes.

2 Serialization Graph Testing

This section presents serialization graph testing. Section 2.1 describes the conflict graph theorem and how it is used by SGT. In Section 2.2, the algorithmic adjustments made in [4] are given, before Section 2.3 describes the many-core optimized graph data structure used in [4], which serves as the basis for mixed serialization graph testing.

2.1 Protocol Description

An execution of transactions can be represented by a *schedule*, a time ordered sequence of their operations. For example, consider transactions T_1, T_2, and T_3 shown in schedule s below.

$$s = w_1[x]\ r_2[x]\ r_2[y]\ w_1[y]\ w_2[z]\ w_3[z]\ r_3[x]\ r_3[a]\ w_4[a]\ c_1\ c_3\ c_2\ c_4$$

This schedule can be represented by a *conflict graph* $CG(s)$, shown in Figure 1. Nodes represent transactions and conflicting operations a_i of T_i and b_j of T_j such that $a_i[x] < b_j[x]$, where $T_i \neq T_j$, are represented by an edge $T_i \rightarrow T_j$; possible conflict pairs are $(a, b) \in [(r, w), (w, r), (w, w)]$. For example, in s, T_2 reads x after T_1 writes to x, thus there exists an edge from T_1 to T_2 in Figure 1.

Changing the order of conflicting operations *could* alter the behavior of at least one transaction. Therefore, an execution of transactions is conflict serializable if a serial ordering of transactions that satisfies all conflict edges can be found. Such a serial ordering exists iff the conflict graph is acyclic. This is known as the *conflict graph theorem* [1]. Note, s is not conflict serializable because $CG(s)$ in Figure 1 contains a cycle.

Theorem 1 (Conflict Graph Theorem). *A schedule s is conflict serializable iff its corresponding conflict graph $CG(s)$ is acyclic.*

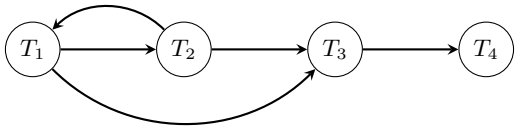


Fig. 1: Conflict graph representation of s.

SGT directly utilizes the conflict graph theorem by maintaining an acyclic conflict graph. For each operation, conflicts are determined and edges inserted into the graph. An important point to note is the orientation of edge insertions: a transaction only inserts edges incoming to *itself*. After edge insertion, a cycle check is performed *before* executing the operation. If executing the operation would introduce a cycle the offending transaction is aborted and its edges removed. At commit time, a final cycle check is executed, if no cycle is found the transaction commits and removes its edges. In short, SGT provides serializability by ensuring the acyclic invariant.

2.2 Algorithmic Adjustments

Due to space constraints nodes must be pruned from the conflict graph. The SGT algorithm sketched in Section 2.1 allows transactions to commit with incoming edges if they pass a cycle check. Under this scheme simply deleting nodes of committed transactions can lead to subtle serialization violations. Assume in Figure 1, T_3 has committed and T_2 is active. If T_3 is removed upon commitment, T_2 may subsequently perform an operation that introduces a cycle with T_3, but which goes undetected due to T_3's removal, introducing a serialization error. To solve this, in [4] a transaction can commit iff it has no incoming edges, else delaying until this condition is met. As transactions only insert edges incoming to themselves, after issuing a commit request, no more incoming edges are added. Thus, once all incoming edges are removed from the committing node, via the parent node aborting or committing, it will not be in a cycle.

This rule also achieves two desirable properties: *recoverability* and *order preservation*. Recoverability ensures failures do not leave the database in an

inconsistent state. In s, T_1 writes x, then T_2 reads x. If T_2 commits before T_1, T_1 may subsequently abort, at which point T_2 has read from a value that never existed. Delaying T_2's commit until it has no incoming edges prevents this issue; if T_1 aborts then T_2 is also aborted. *Order preservation* ensures the real-time commit order matches the serialization order. In s, T_4 overwrites the value a read by T_3, thus in the serial order: $T_3 \rightarrow T_4$. If T_4 commits before T_3 no recoverability issues are introduced, but the real-time commit order does not match the serialization order. Delaying T_4's commit until it has no incoming edges avoids this, providing users with an improved, more intuitive, experience.

2.3 Many-Core Optimizations

Many-core DBMSs use optimistic approaches as their pessimistic counterparts suffer from poor performance as the core count increases. A common bottleneck in optimistic approaches is an exclusive single-threaded verification phase. Regards SGT, in [4] to avoid a global lock for graph operations, a data structure with a node-local locking protocol is used. Nodes in the graph each store a transaction status (committed, active, aborted) and two sets of pointers representing incoming and outgoing edges. Nodes can then be locked in two modes:

- ***Shared mode:*** transactions can concurrently access the node for edge insertions, edge deletions, and cycle checking. Edge sets guarantee thread-safe concurrent access for scans, insertions, and deletes under the shared lock.
- ***Exclusive mode:*** used for the commit-critical check for incoming edges.

The protocol works as follows: when a transaction T_x identifies a conflict with transaction T_y it first checks if an edge already exists from T_y to T_x, if so, no additional work is needed. Else, T_x acquires a shared lock on T_y's node. If T_y is active, then an edge pointer to T_x is inserted into T_y's outgoing edge set and an edge pointer from T_y is inserted into T_x's incoming edge set. T_x then checks for a cycle using a *reduced depth-first search* (DFS) algorithm. Reduced DFS begins at the validating node (T_x) and traverses only the portion of the graph that is needed; each step holds nodes in shared lock mode. At commit time, T_x acquires an exclusive lock on its node and checks for incoming edges. If there are none, T_x commits and shared locks are acquired on each node in T_x's outgoing edge set and the edge from T_x removed. Else, there exists at least one incoming edge and the exclusive lock is released and the check is repeated. Another common bottleneck is the reliance on a global timestamp allocator, this is avoided in [4] by letting conflict graph nodes double up as transaction ids.

SGT requires a mechanism to derive conflicts. In [4] rows store a sequential history of accesses. Each access stores the operation type, read or write, and the transaction id. Decoupling the access information from the graph data structure decreases contention. Note, sequentially ordered access is ensured by per-row spin-locks which are released immediately after the operation completes, i.e., the lock is not held until commit time, an improvement over lock-based approaches.

In summary, the SGT implementation in [4] uses a commit rule to simplify node removal, decouples conflict detection, and employs a highly parallel graph

structure that allows concurrent cycle checking. In particular, this protocol scales well as the commit critical check only shortly blocks other threads from accessing a node and only the part of the conflict graph needed for validation is traversed. Lastly, it minimizes unnecessary aborts, accepting all conflict serializable schedules and provides an ideal baseline for the development of MSGT.

3 Mixing in the Wild

This section motivates the development of a mixed graph-based scheduler that minimizes unneccessary aborts by surveying the isolation levels supported by commercial and open source DBMSs.

It is rare for practical DBMSs to offer applications only a singular isolation level, instead permitting transactions to be run at different isolation levels. In order to assess this claim we surveyed the isolation levels offered by 24 DBMSs in Table 1[3]. Classification was performed based on each database's public documentation. We found 7 isolation levels represented: Read Uncommitted, Read Committed, Cursor Stability, Snapshot Isolation, Consistent Read, Repeatable Read, and Serializable. Note, the exact behavior of each isolation level is highly system-dependent. Interestingly, we found 18 databases supported multiple isolation levels. Of systems offering a singular isolation level Serializable was the most common; these systems were typically NewSQL [8] systems, e.g., CockroachDB [9]. This may suggest a trend away from mixed DBMSs, however, TiDB recently added support for Consistent Read isolation [10] indicating the utility of weaker isolation in practical systems remains.

4 Mixing Theory

This section presents the correctness criteria utilized by MSGT. Section 4.1 reproduces the system model from [5], which is used to define weak isolation levels in Section 4.2, before the mixing-correct theorem is defined in Section 4.3.

4.1 System Model

In Adya's system model, transactions consist of an ordered sequence of read and write operations to an arbitrary set of data items, book-ended by a `BEGIN` operation and a `COMMIT` or an `ABORT` operation. The set of items a transaction reads from and writes to is termed its *item read set* and *item write set*. Each write creates a *version* of an item, which is assigned a unique timestamp taken from a totally ordered set (e.g., natural numbers) version i of item x is denoted x_i; hence, a multiversioned system is assumed. All data items have an initial *unborn* version $\perp$ produced by an initial transaction $T_\perp$. The unborn version is

[3] * Indicates the default setting, [a] Referred to as Read Stability, [b] Behaves like Read Committed due to MVCC implementation, [c] Implemented as Snapshot Isolation, [d] Requires manual lock management, [e] Behaves like Consistent Read.

Table 1: Isolation Levels Supported by Open Source & Commercial DBMSs.

Database System	**Isolation Level**						
	RU	***RC***	***CS***	***SI***	***CR***	***RR***	***S***
Actian Ingres 11.0	✓	✓	✓	✗	✗	✓	✓*
Clustrix 5.2	✗	✓[e]	✗	✗	✗	✓*[c]	✓
CockroachDB 20.1.5	✗	✗	✗	✗	✗	✗	✓*
Google Spanner	✗	✗	✗	✗	✗	✗	✓*
Greenplum 6.8	✓[b]	✓*	✗	✗	✗	✓	✗
Dgraph 20.07	✗	✗	✗	✓*	✗	✗	✗
FaunaDB 2.12	✗	✗	✗	✓	✗	✗	✓*
Hyper	✗	✗	✗	✗	✗	✗	✓
IBM Db2 for z/OS 12.0	✓	✓[a]	✓*	✗	✗	✓	✗
MySQL 8.0	✓	✓	✗	✗	✗	✓*	✓
MemGraph 1.0	✗	✗	✗	✓*	✗	✗	✗
MemSQL 7.1	✗	✓*[e]	✗	✗	✗	✗	✗
MS SQL Server 2019	✓	✓*	✗	✓	✗	✓	✓
Neo4j 4.1	✗	✓*	✗	✗	✗	✗	✓
NuoDB 4.1	✗	✓	✗	✗	✓*	✗	✗
Oracle 11g 11.2	✗	✓*	✗	✓	✗	✗	✗
Oracle BerkeleyDB	✓	✓	✓	✓	✗	✗	✓
Oracle BerkeleyDB JE	✓	✓	✗	✗	✗	✓*	✓
Postgres 12.4	✓[b]	✓*	✗	✗	✗	✓[c]	✓
SAP HANA	✗	✓*	✗	✓	✗	✗	✗
SQLite 3.33	✓	✗	✗	✗	✗	✗	✓*
TiDB 4.0	✗	✗	✗	✓*	✓	✗	✗
VoltDB 10.0	✗	✗	✗	✗	✗	✗	✓*
YugaByteDB 2.2.2	✗	✗	✗	✓*	✗	✗	✓

located at the start of each item's version order. An execution of transactions on a database is represented by a *history*, H. This consists of a *partial order of events*, which reflects (i) each transaction's read and write operations, (ii) data item versions read and written and (iii) commit or abort operations, and a *version order*, which imposes a total order on committed data item versions.

There are three types of dependencies between transactions, which capture the ways in which transactions can *directly* conflict. *Read dependencies* capture the scenario where a transaction reads another transaction's write. *Antidependencies* capture the scenario where a transaction overwrites the version another transaction reads. *Write dependencies* capture the scenario where a transaction overwrites the version another transaction writes. Their definitions are as follows:

Read-Depends Transaction T_j *directly read-depends* (wr) on T_i if T_i writes some version x_k and T_j reads x_k.

Anti-Depends Transaction T_j *directly anti-depends* (rw) on T_i if T_i reads some version x_k and T_j writes x's next version after x_k in the version order.

Write-Depends Transaction T_j *directly write-depends* (ww) on T_i if T_i writes some version x_k and T_j writes x's next version after x_k in the version order.

Using these definitions, a history can be represented by a *direct serialization graph*, $DSG(H)$. Nodes correspond to committed transactions and edges mark *direct dependencies*, or conflicts, between transactions. Again the direction of these dependencies indicate the apparent order of transactions in a serial execution. Anomalies are defined by stating properties about the DSG.

To illustrate the difference between an Adya history and a schedule, s from Section 2 is given again with versions accessed by each operation (version order: $[x_0 \ll x_1, y_0 \ll y_1, z_2 \ll z_3, a_0 \ll a_4]$). Figure 2 shows $DSG(H)$.

$$H = w_1[x_1]\ r_2[x_1]\ r_2[y_0]\ w_1[y_1]\ w_2[z_2]\ w_3[z_3]\ r_3[x_1]\ r_3[a_0]\ w_4[a_4]\ c_1\ c_3\ c_2\ c_4$$

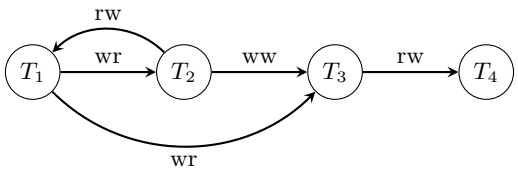


Fig. 2: Direct serialization graph representation of H.

The above *item-based* model can be extended to handle *predicate-based* operations [5]. Database operations are frequently performed on set of items provided a certain condition called the *predicate*, P holds. When a transaction executes a read or write based on a predicate P, the database selects a version for each item to which P applies, this is called the version set of the predicate-based denoted as $Vset(P)$. A transaction T_j changes the matches of a predicate-based read $r_i(P_i)$ if T_i overwrites a version in $Vset(P_i)$.

4.2 Weak Isolation Levels

Using the system model in Section 4.1, definitions of isolation levels are given via a combination of constraints and the prevention of types of cycles in the DSG. In total 11 isolation levels are presented in [5]. To simplify discussions we consider a subset:

- Read Uncommitted: proscribes anomaly *Dirty Write (G0)*, the DSG cannot contain cycles consisting entirely of write-depends edges.
- Read Committed: proscribes *G0* and anomalies, (i) *Aborted Read (G1a)*, transactions cannot read data item versions created by aborted transactions, (ii) *Intermediate Reads (G1b)*, transactions cannot read intermediate data item versions, and (iii) *Circular Information Flow (G1c)*, the DSG cannot contain cycles consisting of write-depends and read-depends edges.
- Repeatable Read: proscribes *G0*, *G1*, and *Write Skew (G2-item)*, the DSG cannot contain cycles containing one or more item-anti-depends edges.
- Serializable: proscribes anomalies *G0*, *G1*, and *G2*, the DSG cannot contain any cycles; this extends the coverage to predicate-anti-depends edges.

4.3 Mixing of Isolation Levels

To define a correctness criteria for a mixed DBMS, a DSG variant is used to represent a mixed history, referred to as a *mixed serialization graph*, $MSG(H)$.

A MSG only includes *relevant* and *obligatory* conflicts. A relevant conflict is a conflict that is pertinent to a given isolation level, e.g., read-depends edges are relevant to Read Committed transactions but not Read Uncommitted transactions. An obligatory conflict is a conflict that is relevant to one transaction but not the other, e.g., an item-anti-depends edge between a Read Committed transaction and a Serializable transaction is relevant to the Serializable transaction and not the Read Committed transaction but still must be included in the MSG. Adya defines the edge inclusion rules for an MSG as follows:

1. Write-depends edges are relevant to all transactions regardless of isolation level thus always included.
2. Read-depends edges are relevant for edges incoming to Read Committed, Repeatable Read, or Serializable transactions.
3. Item-anti-depends edges are included for outgoing edges from Repeatable Read and Serializable transactions.
4. Predicate-anti-depends edges are included for outgoing edges from Serializable transactions.

Now in a mixed DBMS, a history is correct if each transaction is provided the isolation guarantees that pertain to its level leading to the *mixing-correct theorem* [5]. Figure 3 illustrates the differences between DSG and MSG representations of a history with the non-relevant and non-obligatory edges removed.

Theorem 2 (Mixing-Correct Theorem). *A history H is mixing-correct if $MSG(H)$ is acyclic and phenomena G1a and G1b do not occur for Read Committed, Repeatable Read, and Serializable transactions.*

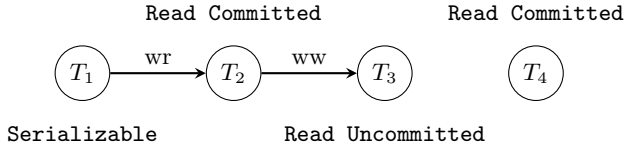


Fig. 3: Mixed serialization graph representation of H.

5 Mixed Serialization Graph Testing

This section presents mixed serialization graph testing. Section 5.1 describes the protocol design. Section 5.2 outlines MSGT's advantages and disadvantages. Lastly, Section 5.3 gives implementation details.

5.1 Protocol Design

MSGT blends together the SGT implementation from Section 2 with the mixing-correct theorem from Section 4. Rather than using the graph data structure in Section 2.3 to represent a conflict graph, in MSGT it is used to represent a mixed serialization graph with one addition: nodes include transaction's desired isolation level. For each operation, conflicts are determined using the transaction's isolation level and the edge inclusion rules for a mixed serialization graph enumerated in Section 4.3: when a transaction T_i detects a conflict with T_j it is inserted into the mixed graph if the conflict is relevant to T_i or obligatory for T_j. No changes are necessary to the per-row meta-data as all direct dependencies can be derived from the access history. After edge insertion, a cycle check is performed *before* executing the operation. If executing the operation would introduce a cycle the offending transaction is aborted. At commit time, as in [4] a transaction can commit if and only if it has no incoming edges. This simplifies node deletion and provides order preservation. Note, recoverability in no longer ensured as under the mixing-correct theorem it is permissible for a Read Uncommitted transaction to read from a transaction that subsequently aborts.

5.2 Discussion

As seen in Section 3 many DBMSs offer weak isolation levels and thus a considerable portion of applications are built atop such guarantees. Typically, mixed isolation is an after thought as DBMSs chase performance, in MSGT mixed isolation is a catered for as a *first-class citizen*. Such an approach provides a higher degree of concurrency and hence performance, whilst also providing the optimal property of no unnecessary aborts.

It is worth noting the utility of weak isolation is limited to applications that can tolerate potentially non-serializable behavior. Additionally, if a workload exhibits low contention or is designed in a manner such that anomalies provably do not occur [11], then the additional overhead of managing the MSG has little benefit.

5.3 Implementation Details

MSGT was implemented in our prototype in-memory DBMS which has a pluggable concurrency control module[4]. The DBMS has a pool of worker threads and each transaction is pinned to a specific worker thread for its duration. Each worker thread has an independent workload generator. Thus when a transaction is committing we repeatedly execute the commit routine (check for incoming edges).

To ensure high operation throughput under a concurrent workload data structures use atomic operations were appropriate. For safe memory reclamation in a concurrent environment epoch-based garbage collection is used [12] and nodes' edge sets are recycled after a transaction is committed and deleted.

[4] https://github.com/jackwaudby/spaghetti

Adya's system model is defined in terms of a multiversion model, but as the MSGT scheduler allows transactions to optimistically read dirty records, the possibility of cascading aborts is introduced. Unwinding writes due to cascading aborts lead to unnecessary system load and which is not useful for both user and system. Therefore, only one uncommitted transaction is allowed to modify a data item. Also, the prototype DBMS does not currently support predicate-based operations, thus an item-based read/write model is assumed. Hence, some isolation levels, e.g., Repeatable Read can not be captured.

6 Evaluation

In this section, we experimentally compare MSGT with SGT using our prototype in-memory DBMS. The evaluation focuses on two implementations: a graph-based scheduler (SGT) and a mixed graph-based scheduler (MSGT).

Experiments were performed using an Azure Standard D48v3 instance with 48 virtualized CPU cores and 192GB of memory. Prior to each experiment, tables are loaded, followed by a warm-up period, before a measurement period; both are of configurable length, we use 60 seconds and 5 minutes respectively. We measure the following metrics:

- ***Throughput:*** number of transactions committed per second.
- ***Abort rate:*** rate at which transactions are being aborted.
- ***Average latency:*** the latency time of committed transactions (in ms) averaged across the measurement period.

Our experiments use the Yahoo! Cloud Serving Benchmark (YCSB) [7]. YCSB was originally designed to evaluate large-scale Internet applications, it is re-purposed here as a microbenchmark to allow various aspects of an OLTP workload to be altered. Specifically, we tweak the proportion of serializable transactions, differ the contention level, and increase the core count to measure scalability. YCSB has a one table with a primary key and 10 additional columns each with 100B of random characters. For all our experiments, we use a YCSB table of 100K rows. There are two types of transaction: read or update, each contains 10 independent operations accessing 10 distinct items. Update transactions consist of 5 reads and 5 writes that occur in random order. Read transactions consist solely of read operations. The proportion of update transactions is controlled by the parameter, U, it is fixed to 50% for our experiments. Data access follows a Zipfian distribution, where the frequency of access to hot records is tuned using a skew parameter, θ. When $\theta = 0$, data is accessed with uniform frequency, and when $\theta = 0.9$ it is extremely skewed. In order to measure the impact of transactions running at weaker isolation we introduce an additional parameter, ω, which controls the proportion of transactions running at Serializable isolation. The remainder are split between Read Committed (90%) and Read Uncommitted (10%).

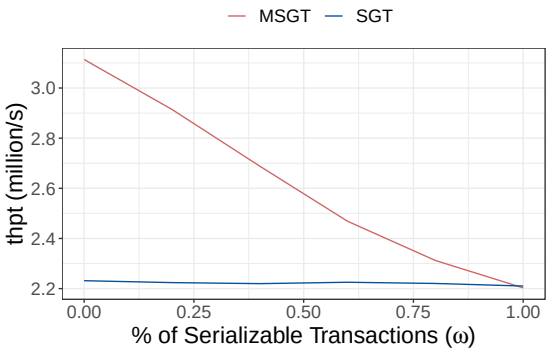


Fig. 4: Throughput as serializable transactions (ω) varied from 0% to 100%.

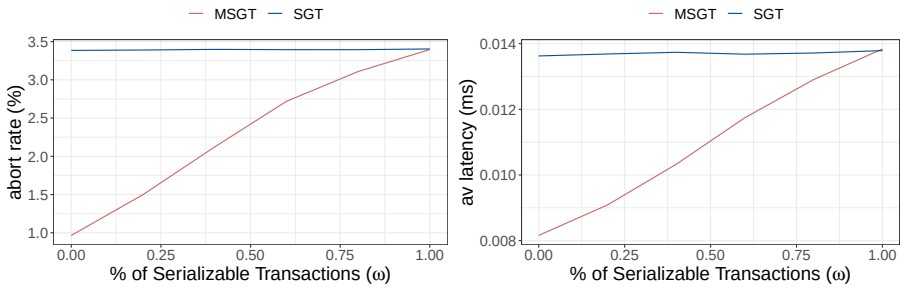
6.1 Isolation

We begin with measuring the impact of increasing the proportion of transactions executing at Serializable isolation from 0% to 100%. This aims to test MSGT's ability to leverage its theoretical properties to offer increased performance when transactions are run at weaker isolation levels. For this experiment, we opt for a medium contention level, $\theta = 0.8$, and the framework is configured to run with 40 cores.

In Figure 4, SGT's throughput is invariant to the proportion of Serializable transactions, this is anticipated as it is unable to take advantage of transactions' declared isolation levels, in effect, executing all transactions at Serializable. Meanwhile, the throughput of MSGT decreases as ω is increased, converging towards SGT's throughput. When there are no Serializable transactions ($\omega = 0.0$), MSGT achieves a 39% increase in throughput. At $\omega = 0.4$, this drops to a 21% increase and at $\omega = 0.8$ a 4% gain is exhibited. When $\omega = 1.0$, SGT marginally outperforms MSGT, this can be attributed to the additional overhead of managing the MSG. This relationship is reflected in the abort rate displayed in Figure 5a, across the range of ω, SGT's abort rate varies from a 3x increase over MSGT's abort rate to an equivalent abort rate when all transactions are executed at Serializable. A higher abort rate degrades the user experience, reduces throughput and, as can be seen in Figure 5b, harms latency.

6.2 Contention

In the next experiment we measure the effect of increasing contention in the system by varying θ from 0.6 to 0.9. Contention happens when multiple transactions try to read or write the same database item. In theory, contention increases the chance of conflicts between transactions. This should translate into an increase in the number of edges inserted into the conflict graph. Under SGT all edges are inserted, whereas, MSGT utilizes isolation levels to be more selective over edge insertions (only adding relevant or obligatory edges) hence it inserts less



(a) Abort rate *vs.* ω (b) Average latency *vs.* ω.

Fig. 5: Serializable transactions (ω) varied from 0% to 100%.

edges into the conflict graph, and should find less cycles (aborts) compared to SGT. We set the proportion of Serializable transactions to $\omega = 0.2$. Again the experiment was run with 40 cores.

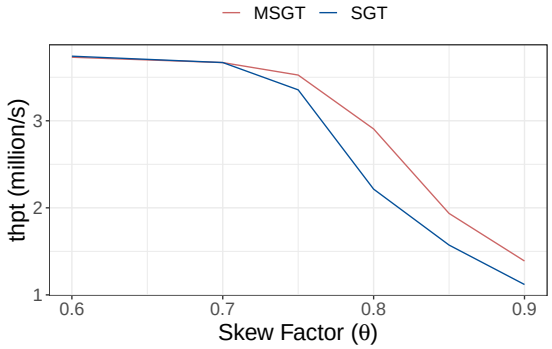
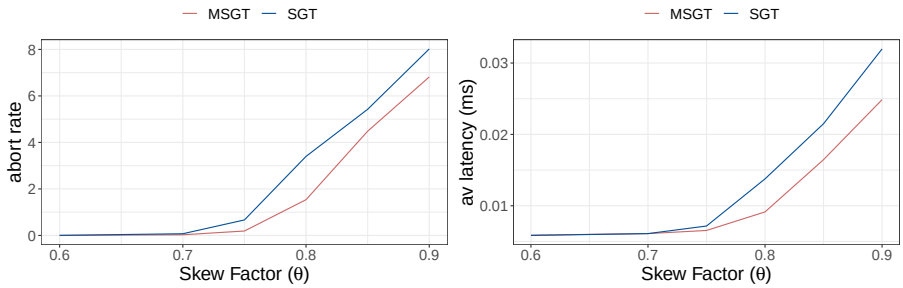


Fig. 6: Throughput as contention factor (θ) varied from 0.6 to 0.9.

Figure 6 displays the throughput of SGT and MSGT as the contention is increased. As θ increases the throughput decreases for both protocols. For low levels of contention SGT performs marginally better than MSGT (<1% difference), but under high contention this reverses and MSGT offers a 24% increase in throughput. Figure 7a shows that after $\theta = 0.7$, the abort rate begins increasing for both protocols. At the highest level of contention ($\theta = 0.9$), 0.1% of the data is accessed by 35% of the queries, and SGT aborts 17% more transactions than MSGT. Lastly, in Figure 7b, above $\theta = 0.7$, MSGT achieves between a 9% and 28% reduction in the average latency.



(a) Abort rate *vs.* θ (b) Average latency *vs.* θ.

Fig. 7: Contention factor (θ) varied from 0.6 to 0.9.

6.3 Update Rate

For the next experiment, we explore the effect of varying the proportion of update operations (U) within each transaction. For this experiment, we opt for a medium contention level, $\theta = 0.8$, set the proportion of Serializable transactions to $\omega = 0.2$, with the framework configured to run with 40 cores.

From Figure 8 it can be seen that at both extremes $U = 0.0$ and $U = 1.0$ MSGT displays no benefit over SGT. When $U = 0.0$, the workload is read-only thus no (ww, wr, rw) conflicts are generated. Conversely, with $U = 1.0$ all transactions are write-only, hence only ww conflicts can occur, which are always inserted into the graph under SGT and MSGT. In both cases MSGT is unable leverage its selective conflict detection rules. However, between the extremities MSGT is able to produce higher throughput compared to SGT (up to 28% when $U = 0.2$).

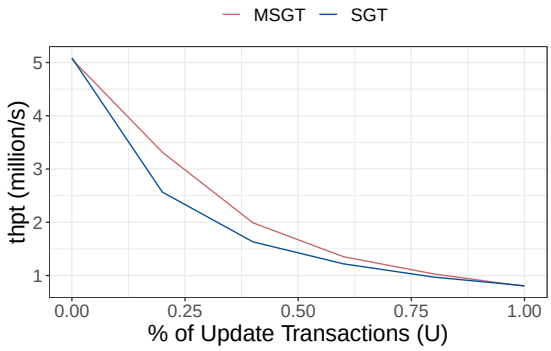
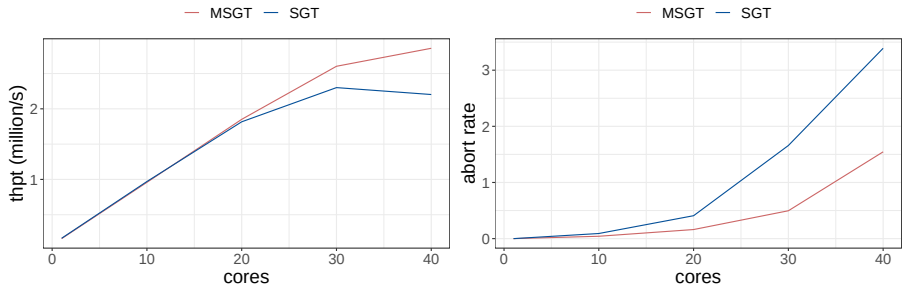


Fig. 8: Throughput as update rate transactions (U) varied from 0.0 to 1.0.

6.4 Scalability

In this experiment we fix the workload factors and vary the core count (1 to 40) to evaluate MSGT's scalability compared to SGT. We anticipate that MSGT scales better than SGT as its scheduler generally performs less work (edge insertions and cycle checking). From Figure 9a it can be seen that until 20 cores the throughput of both protocols is indistinguishable; in fact, up to 10 cores SGT exhibits between a 1.2% and 3.1% increase over MSGT. After this point, a gap appears, at 30 cores MSGT has 13.1% higher throughput and at 40 cores this difference increases to 27.9%.

In Figure 9b it can be seen the abort rate of the protocols starts to diverge after 10 cores: SGT has an abort rate of 1.65% and 3.39% at 30 and 40 cores respectively, whereas, MSGT's is 0.50% and 1.54%.



(a) Throughput *vs.* cores. (b) Abort rate *vs.* cores.

Fig. 9: Core count varied 1 to 40.

7 Conclusion

In this paper we presented mixed serialization graph testing, a graph-based scheduler that leverages Adya's mixing-correct theorem to permit transactions to execute at different isolation levels. When workloads contain transactions running at weaker isolation levels, MSGT is able to outperform serializable graph-based concurrency control by up to 28%. Additionally, MSGT scales as the number of cores is increased, an important property given modern hardware. Like SGT, MSGT minimizes the number of aborted transactions, accepting all useful schedules under the mixing-correct theorem, which greatly improves user experience. As part of future work we wish to extend our performance evaluation to include industry standard benchmarks such as TPCx-IoT [13] and TPC-C [14]. In summary, this paper strengthens recent work refuting the assumption that graph-based concurrency control is impractical.

Acknowledgements

J. Waudby was supported by the Engineering and Physical Sciences Research Council, Centre for Doctoral Training in Cloud Computing for Big Data [grant number EP/L015358/1].

References

References

1. Bernstein, P., Hadzilacos, V., Goodman, N.: Concurrency Control and Recovery in Database Systems. (1987)
2. Eswaran, K., Gray, J., Lorie, R., Traiger, I.: The Notions of Consistency and Predicate Locks in a Database System. Commun. ACM, vol. 19(11), pp. 624–63 (1976)
3. Kung, H., Robinson, J.: On Optimistic Methods for Concurrency Control. ACM Trans. Database Syst., vol. 6(2), pp. 213–226 (1981)
4. Durner, D., Neumann, T.: No False Negatives: Accepting All Useful Schedules in a Fast Serializable Many-Core System. 35th IEEE International Conference on Data Engineering, 734–745 (2019)
5. Adya, A.: Weak Consistency: A Generalized Theory and Optimistic Implementations for Distributed Transactions. PhD Thesis, Massachusetts Institute of Technology (1999)
6. Gray, J., Lorie, R., Putzolu, G.,Traiger, I.: Granularity of Locks and Degrees of Consistency in a Shared Data Base. Modelling in Data Base Management Systems, 365–394 (1976)
7. Cooper, B., Silberstein, A., Tam E., Ramakrishnan, R., Sears, R.: Benchmarking cloud serving systems with YCSB. Proceedings of the 1st ACM Symposium on Cloud Computing, pp. 143–154 (2010)
8. Pavlo, A., Aslett, M.: What's Really New with NewSQL?. SIGMOD Rec., vol. 45(2), pp. 45–55 (2016)
9. Taft, R. et al.: CockroachDB: The Resilient Geo-Distributed SQL Database. Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data., pp. 1493–1509 (2020)
10. TiDB Transaction Isolation Levels, https://docs.pingcap.com/tidb/dev/transaction-isolation-levels. Last accessed 9 May 2022
11. Fekete, A., Liarokapis, D., O'Neil, E., O'Neil, P., Shasha, D.: Making snapshot isolation serializable. ACM Trans. Database Syst., vol. 30(2), pp. 492–528 (2005)
12. Fraser, K.: Practical lock-freedom. PhD Thesis, University of Cambridge (2004)
13. Poess, M., Nambiar, R., Kulkarni, K., Narasimhadevara, C., Rabl, T., Jacobsen, H.A.: Analysis of TPCx-IoT: The First Industry Standard Benchmark for IoT Gateway Systems. 34th IEEE International Conference on Data Engineering, 1519–1530 (2018)
14. TPC Benchmark C, revision 5.11, http://www.tpc.org/tpc_documents_current_versions/pdf/tpc-c_v5.11.0.pdf. Last accessed 19 July 2022