High Performance Mixed Graph-Based Concurrency Control

Jack Waudby[1]

[1]Supervised by Paul Ezhilchelvan, Newcastle University, Newcastle upon Tyne, Tyne and Wear, NE1 7RU, United Kingdom

Abstract

Modern applications are often built on top of many-core OLTP databases. In such systems, concurrency control is an essential component in achieving high performance. Graph-based concurrency control was historically deemed nonviable due to concerns over the computational costs of maintaining an acyclic conflict graph. This conventional wisdom has been refuted by recent research. The work conducted in this PhD has sought to further investigate the usefulness of graph-based concurrency control. Specifically, we propose *mixed serialization graph testing* (MSGT), a concurrency control protocol that allows transactions to concurrently execute at different isolation levels whilst minimizing unnecessary aborts. MSGT combines a recently proposed concurrent graph data structure with Adya's *mixing-correct theorem*. The practical utility of MSGT is illustrated by a survey of isolation levels supported by 24 ACID databases. MSGT has been implemented in a prototype many-core database and a preliminary evaluation using an augmented YCSB workload performed. Initial results indicate the efficacy of MSGT at leveraging transactions specified with weaker isolation requirements. In such workloads, MSGT is able to outperform serializable graph-based concurrency control by up to 23%. We cast further doubt on the notion that graph-based concurrency control in many-core OLTP databases is impractical.

Keywords

Databases, Concurrency Control, Weak Isolation, Serialization Graph Testing, Mixing-Correct Theorem

1. Introduction

Database concurrency control is responsible for ensuring the effects of concurrently executing transactions are isolated from each other. This is captured by the correctness criteria *serializability* [1]. Implementing serializable transaction processing efficiently is a challenging task and many strategies have been proposed [2]. Until recently, graph-based concurrency control was discounted as a viable strategy, despite possessing the theoretically optimal property of accepting all conflict serializable schedules [1]. Graph-based concurrency control directly uses the *conflict graph theorem* by maintaining an acyclic conflict graph. The computational costs of this were perceived to be intolerable. However, this was refuted in [3] who, using a concurrent data structure to represent the conflict graph, demonstrated graph-based concurrency control can achieve comparable, and often higher, performance in a many-core database when compared to alternative strategies.

Despite advances in serializable transaction processing performance, it often remains unsuitable for application demands. Another tool at databases' disposal to achieve improved performance is to execute transactions at weaker isolation levels [4, 5]. Central to this PhD project is addressing the following question: **can graph-based concurrency control support transactions executed at weak isolation levels, whilst still accepting all valid executions?** Such a *mixed* approach would minimize aborts, permitting higher concurrency and performance. This paper describes our initial attempts to achieve this goal and describes *mixed serialization graph testing* (MSGT) which blends the data structure from [3] with Adya's *mixing-correct theorem* [6].

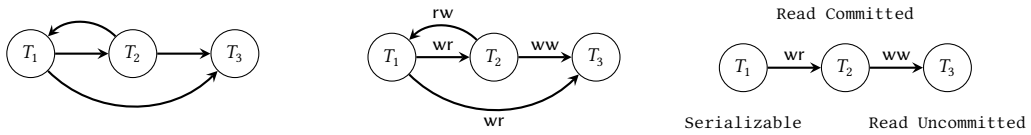
The remainder of this paper is organized as follows. Section 2 motivates the need for a high performance mixed concurrency control protocol. Section 3 describes the necessary background introducing serializable graph-based concurrency control and weak isolation theory, before Section 4 presents mixed serialization graph testing. Section 5 gives our initial evaluation of MSGT's performance. Section 6 presents the PhD work plan and concludes this paper.

2. Motivation

To motivate the development of a high performance mixed graph-based concurrency control we surveyed the isolation levels offered by 24 ACID databases. 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.

Proceedings of the VLDB 2022 PhD Workshop, September 5, 2022. Sydney, Australia

j.waudby2@newcastle.ac.uk (J. Waudby)
https://jackwaudby.github.io/ (J. Waudby)
0000-0002-6649-2744 (J. Waudby)



(a) Conflict graph, ***CG(s)***. (b) Direct serialization graph, ***DSG(s)***. (c) Mixed serialization graph, ***MSG(s)***.

Figure 1: Various representations of an execution of transactions.

Of systems offering a singular isolation level Serializable was the most common; these were typically NewSQL [7] systems, e.g., CockroachDB [8]. This may suggest a trend away from mixed databases, however, TiDB [9] recently added support for Consistent Read isolation indicating the utility of weaker isolation remains.

Table 1
Isolation Levels Supported by ACID Databases.

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

* Indicates the default setting.

Our survey's findings are corroborated by a 2017 survey of database administrators on how applications use databases [4], the survey found the majority of transactions execute at Read Committed. In short, this evidence illustrates the ubiquity of mixed databases and motivates the work to be conducted in this PhD.

3. Background

This section describes serializable graph-based concurrency control, the many-core optimizations made in [3], and introduces a correctness criteria for mixed databases.

3.1. Serialization Graph Testing

Graph-based concurrency control, also known as *serialization graph testing* (SGT), directly utilizes the conflict graph theorem [1] by maintaining an acyclic conflict graph. An execution of transactions can be represented by a *schedule*. Consider transactions T_1, T_2, and T_3 shown in schedule s below; a write on item x by transaction T_i is denoted by $w_i[x]$, a read by $r_i[x]$, and a commit operation by c_i.

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

This schedule can be represented by a *conflict graph* $CG(s)$, shown in Figure 1a. 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 1a. 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 1a contains a cycle.

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

In SGT, for each operation within a transaction, conflicts are determined and edges inserted into the graph. After edge insertion, a cycle check is performed *before* executing the operation; in [3] a *reduced depth-first search* (DFS) is used for cycle checking, which starts from the validating node, searching only the necessary portion of the graph. If executing the operation would introduce a cycle the offending transaction is aborted and its edges removed. At commit time, a transaction delays until it has no incoming edges, at which point it cannot be involved in a cycle. When the transaction terminates it removes its outgoing edges. In short, SGT provides serializability by ensuring the acyclic invariant and thus accepting all valid conflict serializable schedules.

In a many-core database, common sources of performance degradation are, (i) reliance on a global timestamp allocator for transaction ids [2], (ii) use of a single-thread validation phase [10], and (iii) when conflicts are common, optimistic protocols exhibit a high number of aborts [11]. In the SGT implementation in [3], a concurrent graph data structure is developed which uses a node-level locking protocol to avoid using a global lock for graph operations. To avoid a global counter bottleneck, graph nodes double up as transaction ids. Lastly, owing to SGT's acceptance of all valid conflict serializable schedules, aborts are naturally minimized.

3.2. Mixing-Correct Theorem

To define weak isolation levels Adya [6] uses a *direct serialization graph*, *DSG*, which annotates a conflict graph with the ways transactions have *directly* conflicted: *write-depends* (ww), *read-depends* (wr), and *anti-depends* (rw). The corresponding *DSG* of *s* is given in Figure 1b. Non-serializable behaviour (anomalies) are defined by stating properties about the *DSG*, and isolation levels by which anomalies they prevent. For brevity, in this paper we consider 3 isolation levels, for a full enumeration see [6].

- **Read Uncommitted:** proscribes anomaly *Dirty Write (G0)*, the *DSG* cannot contain cycles consisting entirely of ww 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 ww and wr edges.
- **Serializable:** proscribes anomalies *G0*, *G1*, and *G2*, the *DSG* cannot contain any cycles.

Then to define a correctness criteria for a mixed database, Adya uses a *DSG* variant referred to as a *mixed serialization graph*, *MSG*. A *MSG* includes a transaction's declared isolation level and only includes *relevant* and *obligatory* conflicts. A relevant conflict is a conflict that is pertinent to a given isolation level, e.g., read-depends (wr) 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 anti-depends (rw) 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 (ww) are relevant to all transactions regardless of isolation level thus always included.
2. Read-depends edges (wr) are relevant for edges incoming to Read Committed and Serializable transactions.
3. Anti-depends edges (rw) are included for outgoing edges from Serializable transactions.

Now in a mixed database, a schedule is correct if each transaction is provided the isolation guarantees that pertain to its level, leading to the *mixing-correct theorem* [6, p.54–56]. Figure 1c illustrates the differences between *DSG* and *MSG* representations of a schedule with the non-relevant and non-obligatory edges removed.

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

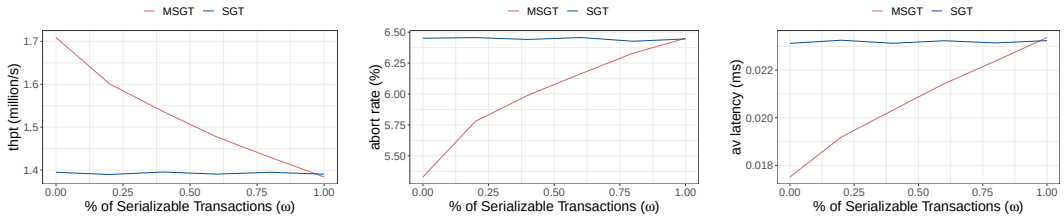
4. Mixed Serialization Graph Testing

In this section, we describe mixed serialization graph testing, focusing on the adjustments to the SGT algorithm (sketched in 3.1) and the concurrent graph data structure; we direct the reader to [3] for their original in-depth description.

For MSGT we use the graph data structure in [3] to represent an *MSG*, which requires one alteration: nodes include transactions' required isolation levels. MSGT proceeds in the same manner as SGT, with one key exception: for each operation, edge insertion of a detected conflict is subject to MSG's edge inclusion rules enumerated in Section 3.2. MSGT's edge insertion algorithm is given in Algorithm 1. First, if an edge already exists for this conflict type no further action is needed, else a cascading abort check if performed. If the parent node has aborted and the inserting node is Serializable or Read Committed it must also abort to avoid *G1a* anomalies. Then, the edge is inserted iff it satisfies MSG's inclusion rules, before a cycle check is executed. If a cycle is found the transaction must abort. At commit time the transaction delays until it has no incoming edges.

5. Preliminary Evaluation

We implemented MSGT and SGT in our prototype in-memory database. Experiments were performed using a Azure Standard D48v3 instance with 48 virtualized CPU cores and 192GB of memory.



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

Figure 2: YCSB - SGT and MSGT with 40 cores when varying the proportion of Serializable transactions from 0% to 100% with medium contention $\theta = 0.8$ and 50% update rate.

Algorithm 1: MSGT Edge Insertion

```
1  Input: Node& this, Node& from, Conflict cType
2  if (from.id,cType) ∉ this.inSet then
3      if cType != RW ∧ this.iso == (RC ∨ S) ∧
         from.state == aborted then
4          return false // cascading abort
5      if cType == ww then
6          this.inSet.add(from.id)
7          from.outSet.add(this.id)
8      else if cType == wr ∧ this.iso != RU then
9          this.inSet.add(from.id)
10         from.outSet.add(this.id)
11     else if cType == rw ∧ from.iso == S then
12         this.inSet.add(from.id)
13         from.outSet.add(this.id)
14     else
15         return true // not relevant/obligatory
16     cycle = cycleCheck(thisNode)
17     return !cycle
18 else
19     return true // edge already exists abort
```

In our preliminary experiments we use the Yahoo! Cloud Serving Benchmark (YCSB) [12]. YCSB has 1 table with a primary key and 10 additional columns each with 100B of random characters; we use a table with 100K rows. There are 2 transaction types: 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 consists of solely read operations. The proportion of update transactions is controlled by the parameter, U. 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; increasing the probability of conflicts between transactions. 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%).

Due to space constraints we report only the results of the experiment which measures 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, U is fixed to 50%, medium contention is used ($\theta = 0.8$) and the framework is configured to use 40 cores. Prior to experiments, tables are loaded, followed by a warm-up period, before a measurement period; both are configurable, we use 60 seconds and 5 minutes respectively. We measure the following metrics: (i) *throughput:* committed transactions per second, (ii) *abort rate:* proportion of transactions aborted, and (iii) *average latency:* latency time of committed transactions (in *ms*) averaged across the measurement period.

In Figure 2a, SGT's throughput is invariant to ω, 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 23% increases in throughput. At $\omega = 0.4$, this drops to a 10% increase and at $\omega = 0.8$ a 2.5% gain. When $\omega = 1.0$, all transactions are executed at Serializable which allows us to ascertain the overheads of MSGT compared to SGT. SGT outperforms MSGT however the difference in throughput is less than 1%.

6. Conclusion & Work Plan

In this paper, we presented MSGT, 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 SGT by up to 23%. Like SGT, MSGT minimizes the number

of aborted transactions, accepting all useful schedules under the mixing-correct theorem. In summary, this paper strengthens recent work refuting the assumption that graph-based concurrency control is impractical. The next steps of this PhD projects are:

1. **MSGT Optimizations:** we have identified two optimizations to improve MSGT's performance: *relevant reduced DFS* and *early commit*. The first is based on the observation that under the current scheme transactions can unnecessarily abort from detecting a non-relevant cycle. The second allows weak isolation transactions to commit with incoming edges, provided none are relevant to their isolation level, reducing latency.

2. **Additional Experiments:** we will quantify MSGT's performance under various application-level benchmarks, e.g., TPC-C [13], and analyze their isolation requirements to help understand where various transaction types occur in practice.

3. **Supporting Additional Isolation Levels:** this PhD project will explore how MSGT can support additional isolation levels, e.g., Snapshot Isolation [14], and quantify the overheads.

4. **Distributed MSGT:** this thesis will explore how MSGT can be integrated into a distributed shared-nothing database. Specifically, how isolation levels, e.g., Read Committed, can be highly available [5] in the presence of concurrent transactions executing at isolation levels that are provably unavailable, e.g., Serializable.

Acknowledgments

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]. We would like to thank Paul Ezhilchelvan and Jim Webber for their valuable discussions.

References

[1] P. A. Bernstein, V. Hadzilacos, N. Goodman, Concurrency Control and Recovery in Database Systems, Addison-Wesley, 1987.

[2] X. Yu, G. Bezerra, A. Pavlo, S. Devadas, M. Stonebraker, Staring into the abyss: An evaluation of concurrency control with one thousand cores, Proc. VLDB Endow. 8 (2014) 209–220.

[3] D. Durner, T. Neumann, No false negatives: Accepting all useful schedules in a fast serializable many-core system, in: 35th IEEE International Conference on Data Engineering, ICDE 2019, Macao, China, April 8-11, 2019, IEEE, 2019, pp. 734–745.

[4] A. Pavlo, What are we doing with our lives?: Nobody cares about our concurrency control research, in: Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017, ACM, 2017, p. 3.

[5] P. Bailis, A. Davidson, A. D. Fekete, A. Ghodsi, J. M. Hellerstein, I. Stoica, Highly available transactions: Virtues and limitations, Proc. VLDB Endow. 7 (2013) 181–192.

[6] A. Adya, Weak Consistency: A Generalized Theory and Optimistic Implementations for Distributed Transactions, PhD Thesis (1999).

[7] A. Pavlo, M. Aslett, What's really new with NewSQL?, SIGMOD Rec. (2016).

[8] R. Taft, I. Sharif, A. Matei, N. VanBenschoten, J. Lewis, T. Grieger, K. Niemi, A. Woods, A. Birzin, R. Poss, P. Bardea, A. Ranade, B. Darnell, B. Gruneir, J. Jaffray, L. Zhang, P. Mattis, Cockroachdb: The resilient geo-distributed SQL database, in: Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, June 14-19, 2020, ACM, 2020, pp. 1493–1509.

[9] D. Huang, Q. Liu, Q. Cui, Z. Fang, X. Ma, F. Xu, L. Shen, L. Tang, Y. Zhou, M. Huang, W. Wei, C. Liu, J. Zhang, J. Li, X. Wu, L. Song, R. Sun, S. Yu, L. Zhao, N. Cameron, L. Pei, X. Tang, Tidb: A raft-based HTAP database, Proc. VLDB Endow. 13 (2020) 3072–3084.

[10] H. T. Kung, J. T. Robinson, On optimistic methods for concurrency control, ACM Trans. Database Syst. 6 (1981) 213–226.

[11] Y. Yuan, K. Wang, R. Lee, X. Ding, J. Xing, S. Blanas, X. Zhang, BCC: reducing false aborts in optimistic concurrency control with low cost for in-memory databases, Proc. VLDB Endow. 9 (2016) 504–515.

[12] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, R. Sears, Benchmarking cloud serving systems with YCSB, in: Proceedings of the 1st ACM Symposium on Cloud Computing, SoCC 2010, Indianapolis, Indiana, USA, June 10-11, 2010, ACM, 2010, pp. 143–154.

[13] TPC, Benchmark C, revision 5.11, Technical Report, TPC, 2010. http://www.tpc.org/tpc_documents_current_versions/pdf/tpc-c_v5.11.0.pdf.

[14] H. Berenson, P. A. Bernstein, J. Gray, J. Melton, E. J. O'Neil, P. E. O'Neil, A Critique of ANSI SQL Isolation Levels, in: M. J. Carey, D. A. Schneider (Eds.), Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data, San Jose, California, USA, May 22-25, 1995, ACM Press, 1995, pp. 1–10.