DELFT UNIVERSITY OF TECHNOLOGY

MASTERS THESIS

Minimizing aborts in an epoch based transaction protocol for deterministic databases

Author:
Marcus SCHUTTE

Supervisor:
Dr. Asterios KATSIFODIMOS
Co-supervisor:
Dr. Burcu OZKAN
Daily supervisor:
Kyriakos PSARAKIS MSc

A thesis submitted in fulfillment of the requirements for the degree of Master of Science

in the

Web Information Systems Group
Software Technology

July 16, 2023

Declaration of Authorship

I, Marcus SCHUTTE, declare that this thesis titled, "Minimizing aborts in an epoch based transaction protocol for deterministic databases" and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed: _____

Date: _____

DELFT UNIVERSITY OF TECHNOLOGY

Abstract

Electrical Engineering, Mathematics and Computer Science
Software Technology

Master of Science

Minimizing aborts in an epoch based transaction protocol for deterministic databases

by Marcus SCHUTTE

Today's need for highly available systems leads to data partitioning and replication across multiple nodes. Providing strong transactional consistency in a distributed database requires extensive communication. For this, algorithms such as two phase commit are used. These communication algorithms add extra network latency's. For application developers and database systems, this is the reason for lowering the isolation level of a database. Deterministic databases run transactions effectively without communication between replicas. Most deterministic databases need the read write sets of a transaction prior to execution to calculate a deterministic execution schedule. Aria does not need the read write sets a priory but uses an epoch based commit protocol. The commit protocol is an optimistic concurrency control algorithm that executes all transactions against a snapshot in the execution phase and determines which transactions can commit in the commit phase. For most workloads Aria outperforms state of the art deterministic databases. However, for high contention workloads Aria suffers performance because of high abort rates. To overcome this problem this thesis proposes two solutions: 1) Lowering the isolation level to snapshot isolation. 2) Reordering the input sequence of transactions on transaction degree. We have found that lowering the isolation level to snapshot isolation allows for 3% less aborts per epoch and reduces latency from 210 ms to 170ms. Reordering the transaction sequence allows for 5 percent less aborts per epoch for snapshot isolation and serializable isolation level. Reordering transactions on degree for serializable isolation level reduces the average latency from 210 ms to 150 ms. Snapshot isolation reordering transactions on degree reduces the average latency from 170 ms to 120 ms.

Acknowledgements

I would like to express my gratitude towards my thesis supervisors. They helped me forming my research such that it became an exciting journey. I would like to thank Kyriakos for providing me with Aria: the system that this thesis is dedicated to improve. But more importantly for believing in me and my algorithms. I would like to thank Burcu for answering my questions about isolation levels and introducing me to interesting papers about the topic. Some complicated topics where hard to grasp for me and sometimes it felt like I would ask the same question twice. However, Burcu always made time for me and answered my questions patiently. Last but not least, I want to thank Asterios for his enthusiasm. For me this worked contagious, after our meetings I felt refreshed and motivated to work on my thesis some more.

Contents

List of Figures

List of Tables

List of Abbreviations

RAW	**R**ead **A**fter **Write** dependency
WAR	**W**rite **A**fter **R**ead dependency
WW	**W**rite **W**rite conflict
WR	**W**rite **R**ead conflict
RW	**R**ead **W**rite conflict
SSG	**S**tart Ordered **S**erialization **Graph**

Chapter 1

Introduction

Large web applications, such as Twitter and Facebook, use distributed and replicated systems to manage data at scale. These systems execute transactions in parallel to enhance processing time. A *transaction* is a sequence of operations performed on the application's state Gray and Reuter, 1992. Transactions provide application developers with a small set of guarantees which simplifies reasoning about concurrent programming in a distributed system Kaki et al., 2017. Most importantly, transactions provide atomicity, consistency, isolation and durability. The isolation level specifies whether a transaction can read the effects of another transaction. This serves as a contract between the database system and the application developer. In an ideal world concurrent executions of transactions are serializable Papadimitriou, 1979. The effect of the transactions on the database is similar as if one transaction would be executed before the other. In other words, there exists a serial order.

However, applications generally use database systems configured with lower isolation guarantees such as read commit and snapshot isolation, because these isolation levels offer better performance Crooks et al., 2017. Lowering the isolation level allows the application developer to make a trade off between throughput and correctness Berenson et al., 2007. Without serializability transactional anomalies appear Gan et al., 2020; Tang et al., 2022. Examples of transactional anomalies are lost update and write skew. These anomalies can manifest in an application as double spending and overselling. Nonetheless, little was known about the severity of concurrency bugs in practise. Until a recent study Cheng et al., 2023 showed that concurrency bugs lead to all kind of unwanted issues in real world applications such as double spending and overselling.

As an example, consider a bank account x with \$100 on the account. Suppose that transactions T_1 wants to subtract \$40 and transaction T_2 wants to subtract \$60. Now suppose that the transactions execute their reads and writes in the following order; Transaction T_1 reads a value of \$100. Transaction T_2 is executed concurrently. T_2 performs a read and also reads $x = 100$. T_1 computes the new value $x = 100 - 60 = 40$. T_2 compute the new value, subtracts $x = 100 - 40 = 60$. T_1 first writes 40 to storage. T_2 overwrites this value and writes 60 to storage. This is an example of Lost Update allowed under read-commit isolation. In distributed systems often referred to as a transaction isolation problem Berenson et al., 2007.

Most commercial databases Microsoft, n.d.; Postgres, n.d.; Oracle, n.d. apply read commit as there default isolation level. To make it more attractive to use strong isolation levels, latency for isolation levels such as serializability and snapshot isolation must be reduced as much as possible. In this work, we look at reducing latency in a different kind of database: a deterministic database. Deterministic databases reduce latency by removing coordination between replicas.

1.1 Deterministic databases

To achieve high availability, distributed databases replicate and partition data over multiple nodes. Transactions that access data on multiple nodes are called distributed transactions. Supporting distributed transactions requires coordination between the replicas, often implemented with two phase commit (2PC) Mohan, Lindsay, and Obermarck, 1986. The drawback of 2PC is that it takes multiple network round trips and adds extra latency to the system.

Deterministic databases Thomson et al., 2012; Faleiro and Abadi, 2014; Faleiro, Abadi, and Hellerstein, 2017; Lu et al., 2020 take a different approach. Asynchronous communication and network latency's introduce randomness in a distributed system. In non-deterministic databases this randomness could produce different output when given the same input twice. Instead of requiring communication between replicas, deterministic databases take out stochastic processes so that replicas always produce the same output for the same input. All replicas receive the same input, instead of a replicating the output of a transaction.

Most deterministic databases Thomson et al., 2012; Faleiro, Abadi, and Hellerstein, 2017; Faleiro and Abadi, 2014 need the read/write set of the transactions before execution to calculate a deterministic schedule. Often the read write sets are obtained by executing the transactions in a test run. In a test run transactions run against a snapshot without committing writes to storage. This leads to extra overhead. Since every transaction is executed at least twice.

Aria Lu et al., 2020 does not need the read/write set a priory. Aria executes transactions in batches called epochs. The batches are processed in two phases: the execution phase and the commit phase. During the execution phase the transactions read from the same snapshot of data. At the end of the epoch a conflict detection algorithm determines which transactions are allowed to commit and which need to abort. The transactions that abort due to concurrency conflicts are rescheduled in the next epoch. One of the downsides of the aforementioned process is that when one key is accessed frequently within one epoch a lot of transactions will have stale reads and need to abort.

Although Aria shows promising performance on most benchmarks, in high content workloads Aria suffers performance because of high abort rates Lu et al., 2021. Therefore, this work aims at improving Aria for high contention workloads. With as goal to minimize the aborting transactions per epoch. There are two potential optimizations: 1) Committing transactions with a weaker isolation level could allow more transactions to commit and therefore increase performance. 2) Reorder transactions according to a given consistency model to minimize number of aborts.

Weak isolation levels trade off consistency for better performance. Lowering the consistency level to a weaker isolation levels remain useful Adya, 1999. Application developers need to be aware of the anomalies. Some workloads doe not show any anomalies under snapshot isolation. This is called robustness against snapshot isolation Beillahi, Bouajjani, and Enea, 2019. Optimizing the commit order for a serializable consistency model means removing as less nodes as possible such that the start order serialization graph is acyclic. This problem is also known as the feedback vertex cover and is known to be NP-complete Karp, 2010. The optimal solution is expected not to scale well, a fast and close to optimal heuristic is sufficient.

The related works, chapter 2 discusses the different system design decisions that exist for distributed databases. Chapter 3 explains how Aria works. In transaction protocols (chapter 4) the different conflict detection methods are developed. First the isolation level is lowered to snapshot isolation in chapter 4.2. This is without

changing the order of the input sequence. Section 4.2.5 provides a statistical analysis of the potential impact that snapshot isolation has on the number of aborts. Section 4.3 proceeds with finding an optimal reordering for serializable consistency model. The approach uses start order serialization graphs. Section 4.4 optimizes the commit order for snapshot isolation on a different transaction model called BC-graphs Zhang et al., 2023. To evaluate how close the methods are to an optimal reordering a ILP formulation is developed that minimizes the number of aborts. The reordering methods we propose in this thesis need a total view of the read write sets before resolving conflicts. Chapter 5 go's in more depth about what kind of communication between the workers is needed. Chapter 6 shows the results of the newly proposed methods on a offline workload that simulates an epoch based commit scheduler. This chapter also shows the proposed methods integrated in the system, configured with fallback on and off. Chapter 7 summarizes the most important results and findings and answers the research questions. The discussion in chapter 8 addresses the way start order serialization graphs are constructed and if the proposed algorithms are properly evaluated. The future works in section 8.6 proposes a reordering optimization algorithms with better run time. The chapter explores the use of a different batch based commit scheduler for optimizing the commit order. Lastly, this chapter proposes to extend Aria to support optimizations for transactions that consist of commutative operations.

1.2 Research questions

1. Does performance of deterministic database increase when lowering the isolation level from serializability to snapshot isolation?
2. Can the number of aborted transactions per epoch be minimized for serializable isolation?
3. Similarly, can the number of aborts be minimized under snapshot isolation?
4. How close are the reordering heuristics to an optimal commit order?
5. Does Aria perform better broadcasting read write sets or in a master worker setup?
6. For which workloads does lowering the isolation guaranty to snapshot isolation result in better performance?

1.3 Contributions

1. Snapshot isolation for Aria is developed in chapter 4.2.
2. A fast heuristic that reorders transactions while maintaining serializability, based on transaction degree is developed in chapter 4.3. This method uses Start Order Serialization graphs.
3. A fast heuristic for reordering transactions under snapshot isolation is developed in chapter 4.4. This method uses BC-graph structure.
4. An ILP that finds an optimal commit order under serializability is formulated in section 4.3.4. An ILP that finds the optimal commit order of snapshot isolation is formulated in section 4.4.2.

5. Chapter 5 discusses the trade offs between coordinator worker or broadcasting communication configuration.

6. A theory of when relaxing constraints to snapshot isolation can reduce the number of aborts for an epoch based commit scheduler is given in chapter 4.1.

Chapter 2

Related work

SQL interface databases such as MySQL Microsoft, n.d. and PostgreSQL Postgres, n.d. are often compared to Swiss pocket knives. They try to do everything but they fail at being proficient in one thing. Today's needs for systems to be highly available has lead to another type of database: NoSQL. Databases such as Cassandra Lakshman and Malik, 2009, Amazon Dynamo DeCandia et al., 2007 and MongoDB prefer availability over consistency and provide little to no transactional guarantees. However, there are also systems Shute et al., 2013; Rao, Shekita, and Tata, 2011 that provide higher consistency. Distributed system often need to make trade offs between certain design choices. Such as consistency levels, isolation guaranties and optimistic concurrency control or two phase locking. This chapter explores some of the state of the art systems and there system design.

2.1 Consistency

To obtain high availability databases replicate data. In case of a node failing a replica can step in. In distributed systems there are different contexts in which consistency is used. Section 2.2 discusses consistency in terms of ACID transactions. This section discusses consistency in terms of the CAP theorem. The consistency level specifies if two concurrent processes are able to see different versions of the data at the same time. Even tough they are connected to different nodes.

Replication can either be *synchronous* or *asynchronous*. Synchronous replication waits for updates to be persistent at the replications which can result in high latency. Asynchronous replication does not wait for updates to be persistent at replicas, reducing latency. However when failures occur this gives less consistency then synchronous replication as some data might be lost. This is essentially what the CAP theorem provides: A system can have only two of the three: consistency, availability and partition tolerance [17]. For many distributed systems such as Dynamo, Cassandra and PNUTS this is the reason to lower consistency guarantee [11, 24, 7]. They provide eventual consistency. Databases such as F1 Shute et al., 2013 and IBM Spinnaker Rao, Shekita, and Tata, 2011 use synchronous consensus algorithms such as Paxos Lamport, 2001 to keep replicas consistent. This is on the very strong spectrum of consistency.

2.2 Distributed transactions

Besides replication, data is often partitioned over nodes to scale out horizontally. Distributed transactions access data over multiple partitions. In 1980 system R implemented the first relational database Astrahan et al., 1976. This was a high level relational database manager based on Codds Codd, 1970 relational model for data.

Not much later the term ACID transactions was introduced by Gray and Reuter Gray and Reuter, 1992. Transaction processing systems provide ACID conditions in a distributed system settings. These conditions are atomicity, consistency, isolation and durability. **Atomicity** provides that either all or non of the changes of a transaction take place. To obtain atomicity in distributed systems use coordination algorithms such as two phase commit (2PC) Mohan, Lindsay, and Obermarck, 1986. **Consistency** provides that a transaction needs to transform the database from one consistent state to another. Regarding the database constraints. **Isolation** provides that regardless of some concurrent execution of transactions, the effect on the database is as if they where executed serial. **Durability**: Once a transaction commits (completes successfully) the changes cannot suffer data loss because of a node failure.

Throughout the years many different type of isolation levels have been developed. Ranging form weaker guaranties, atomic read (ar) and casual consistency (cc) to stronger guaranties such snapshot isolation and serializability Cerone, Bernardi, and Gotsman, 2015. An isolation level works as a contract between the database and the application developer. They describe what a developer can expect if two transactions run concurrent. Weaker isolation levels (ar, cc) allow anomalies such lost update. Snapshot isolation and serializability disallow lost update. However transactions under snapshot isolation can put the database in inconsistent state since it allows for write skew.

2.3 Locking or optimistic concurrency control

Concurrency control (CC) algorithms coordinate data access for transactions in a distributed system setting. Pessimistic concurrency control algorithms use locks. Transactions lock the data that they are reading/writing during there processing time. An example is two phase locking (2PL) Gray, Lorie, and Putzolu, 1975. Optimistic concurrency control (OCC) Kung and Robinson, 1981 algorithms execute transactions in parallel and determine which writes to make durable in the commit phase. In high content workloads optimistic concurrency control algorithms suffer high abort rates Wang and Kimura, 2016; Agrawal, Carey, and Livny, 1987. Wasting resources on transactions that abort. Hybrid approaches apply heuristics to determine when to use OCC or locking concurrency control Thomasian, 1998. Another hybrid is mostly optimistic concurrency control Wang and Kimura, 2016. In this approach only read locks are acquired and released before commit. At commit time conflicts are resolved. There even exist learning approaches that try to learn from the workload which concurrency control algorithm is optimal Wang et al., 2021.

2.4 Transaction chopping

Recent works involve breaking transactions down in smaller parts. Executing transactions as atomic pieces in order to allow more concurrency in distributed systems Mu et al., 2014. In locking based systems the idea behind this is that shorter transactions reduces the time that keys are locked. Using system knowledge a transaction can be chopped into smaller pieces that obtain the same result Shasha et al., 1995. PWV Faleiro, Abadi, and Hellerstein, 2017 uses data-flow analysis to determine when to expose a transactions write before the transaction is finished executing. Some approaches try to rearrange data over partitions to minimize the number of distributed transactions Curino et al., 2010.

2.5 Deterministic databases

Deterministic databases take a different approach. Instead of replicating the output of transactions, the input is replicated. Distributed databases sacrifice latency to ensure serializability and atomicity using synchronous distributed transaction algorithms such as two phase commit (2PC) Mohan, Lindsay, and Obermarck, 1986 or Paxos Lamport, 2001. Deterministic databases produce the same output on the same input. Les coordination is needed resulting to reducing synchronisation overhead Thomson and Abadi, 2010. Deterministic databases BOHM Faleiro and Abadi, 2014, Calvin Thomson et al., 2012 and PWV Faleiro, Abadi, and Hellerstein, 2017 need the write set a priory to build a dependency graph and decide which transactions can commit. Aria Lu et al., 2020 does not need to now the write set a priory but suffers performance in high contention workloads Lu et al., 2021.

2.6 Epoch based commit protocol

Systems can buffer operation and then process them as a group. This method is called *batching*. It can be applied in multiple parts in a Database. In the communication layer messages are batched to reduce message complexity and latency Friedman and Van Renesse, 1997. Requested transactions can also be batched and processed in a group such as in a group commit. Or slightly different: epoch based commit such as in Coco Lu et al., 2021. Group commit focuses on reducing disc latency, epoch based commit reduces commit latency in a distributed system by batching the transactions in the two phase commit (2PC) protocol. In the evaluation phase of optimistic concurrency control algorithm batching gives the opportunity to reorder transactions and optimize for minimal aborts. Ding et al. Ding, Kot, and Gehrke, 2018 have developed a greedy algorithm for reordering transactions to minimize aborts. They reorder on different cost functions and abort transactions with the highest cost until the serialization graph is acyclic. Dring et al Ding, Kot, and Gehrke, 2018 use different policies as cost function. They reorder on in degree of nodes to minimize number of aborts. They reorder on number of retries to minimize tail latency's. Finding an minimal commit order is equivalent to the feedback vertex set problem. Choosing nodes based on degree gives a good approximation for the feedback vertex set (FVS) problem Cutello and Pappalardo, 2015. Other methods assign lazy time stamps. During commit time the real time stamps are assigned and the ordering of transactions order is calculated Yu et al., 2016. This is similar to reordering the transactions in the validation state of a batched approach.

Chapter 3

System overview

Aria is a deterministic database that implements a key-value store. One instance of Aria contains one replica of the data. Aria partitions data over different worker nodes. To scale out horizontally. Within one instance of Aria data is not replicated. In the context of this work Aria is adapted to work as a statefull-function as a service platform (SFAAS). The key components are a sequencer, workers, a Kafka input stream processor called: Ingress and a Kafka output stream processor called Egress. The sequencer makes sure that all Aria replicas receive the same order of input transactions. The Workers handle the transaction logic. An high level overview of the data flow in Aria is demonstrated in Figure 3.1. Here one replica of an Aria instance is shown. The keys are round robin partitioned over the workers. The requests come in to the Ingress stream. The workers consume the request and receive transactions with keys that are located at there partition. A transaction request always invokes at least one key. The first key a transaction invokes determines which worker will receive the transaction. If a transaction wants to access a key located on a other worker, it can do so by making an asynchronous call. All the workers know which keys are at the other workers and they can make request to each other.

The workers operate in two phases. First the workers process the functions scheduled for that epoch. All functions are executed against the same snapshot. The workers keep track of the reads and writes of the functions in read and write sets. All writes are buffered and only made persistent in the commit phase. When a transaction makes a read, the worker checks if the transaction did not make a previous write to that key. If this is the case the worker returns the buffered write of the transaction. Allowing for a transaction to read its own writes. After all functions are done, the workers resolve conflicts and determine which transactions can commit and which should be scheduled for the next epoch. To resolve conflicts a network round trip is needed.

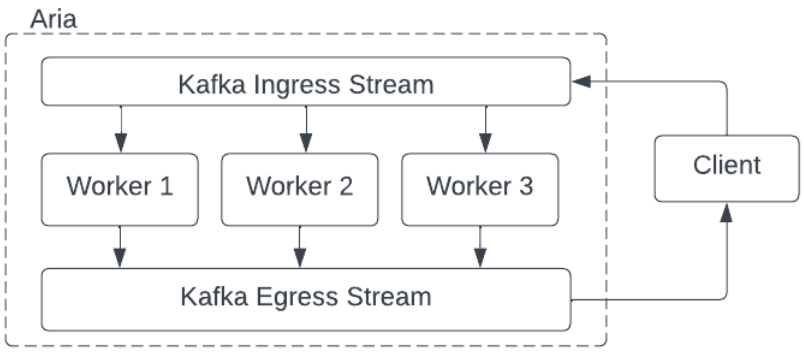


FIGURE 3.1: One replica of Aria high level system overview

```
1  def transactionT1():
2      write(x,1)
3      a = read(y)
4      if a > 0:
5          b = read(x)
6          write(x, b+1)
7
8  def transactionT2():
9      a = read(y)
10     b = read(x)
11     write(x, a+b)
```

FIGURE 3.2: Two transactions represented as sequence of atomic operations on storage (b).

If the abort rate of an epoch is high, Aria can be configured to use the fallback mechanism. This is a locking concurrency algorithm based on Calvin Thomson et al., 2012. After one epoch each transaction is executed once and the read write sets are known. Calvin uses these read write sets to calculate a deterministic locking schedule. With the fallback mechanism Aria ensures that it always performce as least as good as Calvin.

3.1 Data flow example in Aria

For this example consider the system configured with 2 workers. Key x is located on worker 1 and has an initial value of $x = 3$. Key y is located at worker 2 and has an initial value of $y = 5$. As input the system is given two transactions:

```
def transactionT1():
    x=1
    if (y>0):
        x = x + 1

def transactionT2():
    x = x + y
```

Transaction T_1 writes to x and updates x after checking a condition on y. Transaction T_2 updates x by adding y to it. Suppose that these two transaction result atomic operations on the database as implemented in Figure 3.2. This might not be the most optimal sequence of atomic operations to decode these transactions. However for the sake of the example this gives an interesting flow through the system.

Figure 3.3 gives an overview on a timeline how the transactions are executed. At the start of the epoch the read and write sets are empty. The first key that T_1 accesses is x therefore T_1 is routed to start at Worker 1. Similarly transaction T_2 is routed to start on Worker 2. T_1 first writes 1 to x. Worker 1 updates the write set to contain the write of T_1. Then T_1 makes an asynchronous call to Worker 2 with the rest of the instructions for the transaction. At the same time T_2 reads the value of y on Worker 2 and makes an asynchronous call to Worker 1. Worker 2 updates the read set to keep track of T_2's read of y.

Transaction T_2 continuous on worker 1 (the middle transaction on the timeline of Worker 1 in Figure 3.3). Transaction T_2 reads the value of x. Worker 1 returns the initial value of $x = 3$. Although T_1 already wrote to x before. This is because the workers buffer writes. This write is not committed and lives only in the buffer. After T_2 received the value from storage it makes an asynchronous call to Worker 2 with

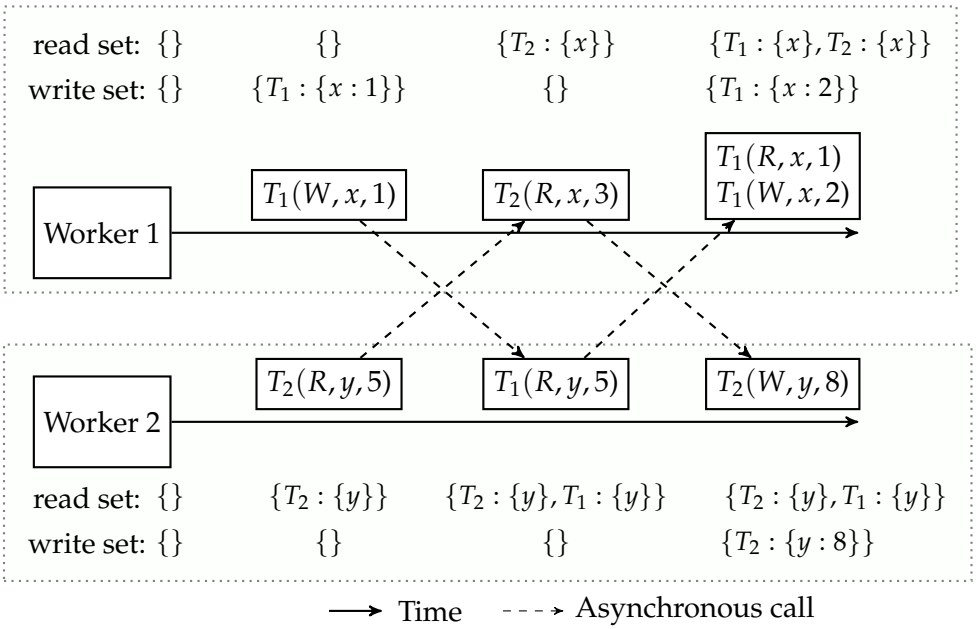


FIGURE 3.3: Data flow example Arias execution phase. The workers keep track of the transactions reads and writes in the read write sets. A transaction is represented as a triple: $T_i(\text{operation}, \text{key}, \text{value})$. Where R is the read operation and W a write.

the last of its instructions. Concurrently transaction T_1 read the value of y on Worker 2. Then Transaction T_1 makes a last asynchronous call to Worker 1. Both workers update the read write sets accordingly.

Worker 1 now executes the last instructions from T_1. T_1 first reads x. This read returns a value of $x = 1$. T_1's previous write was buffered and the workers always check if a worker already wrote to a value. Allowing a transaction to read its own writes. Then transaction T_1 calculates $x + 1 = 2$ and writes $x = 2$ to the buffer. At the same time, Worker 2 executes the last instruction of T_2: a write to the buffer with value $y = 8$.

After the execution phase is finished the workers need to resolve conflicts. See section 4.2.4 to see how Aria resolves conflicts. Then the workers communicate which transactions can commit and which transactions need to abort. The aborted transactions are rescheduled for the next epoch. When the workers know which transactions to commit. The buffered writes of the transactions that can commit are made persistent to storage. The read write sets are cleared and a new epoch can begin.

Chapter 4

Transaction protocols

In this chapter we propose three new transaction protocols. First, in Section 4.1 a bit of background knowledge about start order serialization graphs is established. Section 4.2 introduces snapshot isolation without any optimal reordering. This section also includes a statistical effectiveness analysis (Section 4.2.5) of default serializability, deterministic reordering and snapshot isolation. Section 4.3 proposes reordering method for serializable Isolation model based on the degree of a transaction. Section 4.4 proposes the reordering methods based on degree for snapshot isolation.

4.1 Building start order serialization graphs

This chapter introduces how to build start order serialization graphs. Moreover, it gives a brief analyis of the different patterns that occur in the start order serialization graph for an epoch based commit scheduler. A start-ordered serialization graph (SSG) is a directed graphs with transactions as nodes and dependencies as edges Adya, 1999. From a commit history a Start-ordered serialization graph can be formed. The patterns in the SSG will help reasoning what performance optimizations are possible. For instance, if two transactions try to update the same key only one can commit. They are directly conflicting. Section 4.1.2 shows what this pattern looks in the start order serialization graph. Furthermore, in section 4.1.3 we will see that on workload YCSB-B1 snapshot isolation is not expected to outperform serializability.

4.1.1 Conflict types in epoch based transaction scheduler

Because all transactions read from the same snapshot, all reads are concurrent. They happen at the beginning of the epoch. If a transaction T_i writes to a value that transaction T_j reads, this resolves in a read write (RW) anti-dependency. Since all transactions read from the snapshot, a transaction T_i will never read from another transaction T_j's write within an epoch.. Therefore there are no write read (WR) dependencies. If a transactions T_i writes to the same value as T_j there is a Write Write (WW) conflict. However, the transactions writes are all buffered and still unordered. The conflict detection algorithm can choose a direction by ordering the writes. Figure 4.1 shows a timeline of two transactions reading from and writing to the same key. Resulting in two RW and one WW dependency. The direction of the WW dependency depends on which transaction commits first. If there is a undirected write write edge we choose the direction to point from the transaction with largest transaction id to smaller transaction id.

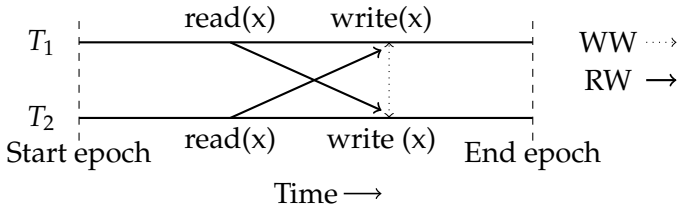


FIGURE 4.1: Timeline of two transactions reading from the same snapshot. Both transactions update x.

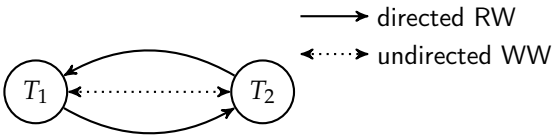


FIGURE 4.2: Start order Serialization Graph (SSG) of two conflicting updates.

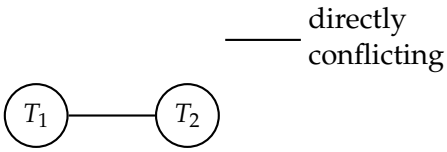


FIGURE 4.3: Directly conflicting transactions

4.1.2 Directly conflicting transactions

Transactions often update a value. An update is a transaction consisting of a read and then a write to the same key. For instance:

$$a = \text{Read}(x)$$
$$\text{Write}(x, a + 1)$$

The transaction above increments the value of x by 1. Although simple, this transaction forms a building block for many transactional workloads. Since it is one of the transactions of the YCSB Gray et al., 1994 workload it is worth studying.

Consider the following example:

Example 4.1.1 *Transactions T_1 and T_2 both update the value of key x.*

$$T_1 : R(x)W(x)$$
$$T_2 : R(x)W(x)$$

As explained in Section 4.1.1 the reads are concurrent since they both read from the same snapshot. However the writes always happen later in time and this results to a RW dependency from transaction T_1 to T_2. The same happens the other way around from T_2 to T_1. The direction of the WW dependency is yet to be determined. This is essentially the same as Figure 4.1 tries to clarify. The start order serialization

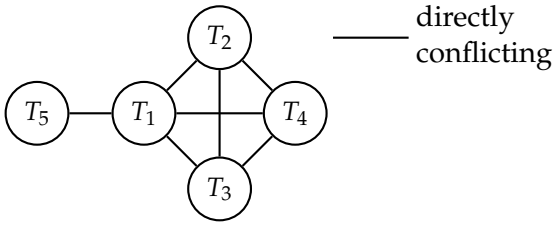


FIGURE 4.4: Directly conflicting transactions

graph is given if Figure 4.2. The order of committing T_1 and T_2 decides the direction of the WW edge. The important observation here is that no matter which order we commit T_1 and T_2 there will always be a cycle in the dependency graph. With a RW edge, followed by a WW edge. Only one of the two transactions can commit. From now on this work will refer to two transactions that have this pattern, as two directly conflicting transactions.

Suppose a workload consists of a lot of directly conflicting transactions. The question is: which transactions do we commit to find an optimal order? This is where reordering the sequence on degree can be quite powerful. Consider the following example:

Example 4.1.2 *Transactions T_1, T_2, T_3 and T_4 all try to update x_1. Transaction T_1 and T_5 both update x_2. Figure 4.4 shows a conflict graph with edges between directly conflicting transactions.*

If transaction T_1 would commit first, this would force T_2, T_3, T_4 and T_5 to abort. Resulting in 1 commit and 4 aborts. However if transaction T_5 commits first, T_1 must abort. But one of T_2, T_3 and T_4 can still commit. Resulting in 2 transactions that can commit. The same result can be obtained by reordering the transactions based on there degree. For the this example only count the directly conflicting edges. The degrees are: degree(T_1)=4, degree(T_2)=3, degree(T_4)= 3, degree(T_4)= 3 and degree(T_5)=1. Reordered on degree from small to large results in the sequence: T_5, T_2, T_3, T_4, T_1. A transaction protocol that commits transactions in this order chooses T_5 to commit first. After committing T_5 one can already see that T_1 needs to abort. Since it conflicts with T_5. The protocol commits T_2 second. This means that T_3 and T_4 would abort. The example gives a sketch of what can be gained by reordering. More precise description algorithms for reordering under snapshot isolation and Serializability are provided later in this chapter.

4.1.3 Snapshot optimizable

Some workloads are able to benefit from relaxing isolation constraints and others are not. In this chapter we explore when it is expected that lowering the Isolation level to snapshot isolation results in a performance gain. To illustrate this consider the following example:

Example 4.1.3 *Transactions T_1 reads x_2 and updates x_1. Transactions T_2 reads x_3 and updates x_2. Transactions T_3 reads x_1 and updates x_3. This results in a read write edge between transactions T_1 and T_2, T_2 and T_3 and T_3 and T_1. As shown in Figure 4.5a.*

Serializable histories do not allow for cycles in the start order serialization graph Adya, Liskov, and O'Neil, 2000. Snapshot isolation does allow for cycles, however there need to be two adjacent RW edges in the cycle Adya, 1999. The history of 4.5a is not serializable since there exist a cycle. However, this history is allowed under snapshot isolation since all the edges are RW edges. For this example snapshot isolation would allow more transactions to commit than serializability. This property will be further referred to as snapshot optimizable.

However workloads that make a lot of updates have a lot of directly conflicting transactions. For these workloads lowering the isolation guaranty to snapshot isolation would not reduce the number of aborts. To see this consisdere the following example:

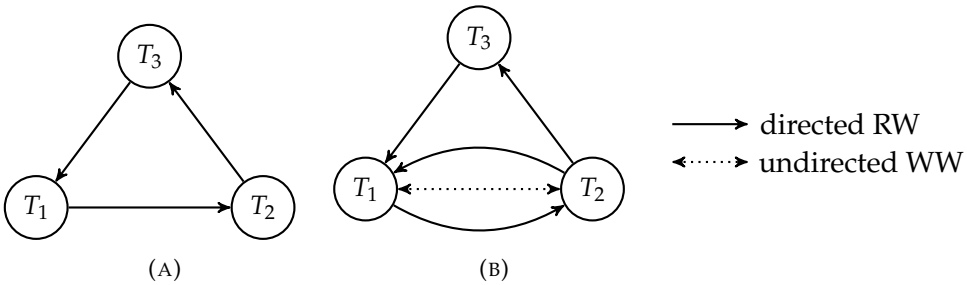


FIGURE 4.5: Example of serialization graph patterns in YCSB-B1. The history in (a) is allowed under snapshot isolation but not serializable. (b) shows two directly conflicting transactions as part of a cycle.

Example 4.1.4 *Transactions T_1* ***updates*** *x_2 and* ***updates*** *x_1. Transactions T_2 reads x_3 and* ***updates*** *x_2. Transactions T_3 reads x_1 and* ***updates*** *x_3. The Start Order Serialization graph is given in Figure 4.5b.*

Here T_1 and T_2 both try to update x_2 resulting in a RW edge from T_1 to T_2, a RW edge from T_2 to T_1 and a undirected/undecided WW edge between T_1 and T_2. In other words T_1 and T_2 are directly conflicting. It does not matter which isolation level is configured: Snapshot isolation or serializability. T_1 or T_2 needs to abort. The take away from this example is that workloads that are update heavy are not able to benefit from lowering the isolation level from serializability to snapshot isolation. This is somewhat similar to robustness against snapshot isolation Beillahi, Bouajjani, and Enea, 2019. In which a workload does not show any anomalies when the isolation level is relaxed from serializability to snapshot isolation.

4.2 Snapshot isolation

In this section performance of snapshot isolation is compared with Aria's default serializable conflict detection algorithm and deterministic reordering (also serializable). The variant of snapshot isolation implemented here does not make use of any reordering optimization. This section starts off with introducing an axiomatic theory of consistency models based on Cerone, Bernardi, and Gotsman, 2015. Then, a small recap of how Aria's already existing methods work and ensure serializability. Finally, in section 4.2.3 Cerone et al's definition of snapshot isolation is used to develop a snapshot isolation conflict detection algorithm for Aria.

4.2.1 Isolation levels

Cerone et al Cerone, Bernardi, and Gotsman, 2015 have developed an axiomatic framework that declare different consistency models. This framework is used because the definitions are free of implementation details. This section briefly explains how Atomic Read Consistency, snapshot isolation and serializability are defined by Cerone et al. Although Atomic read consistency itself is not used in this work, it forms a basis for the other consistency levels and is therefore relevant to discuss.

Definition 4.2.1 *Atomic read consistency satisfies the* INT *and* EXT *axioms.*

Atomic read consistency is the baseline consistency model in Cerone, Bernardi, and Gotsman, 2015 and is defined to satisfy INT and EXT axioms. The *internal consistency axiom* INT ensures repeatable read within a transaction. That is, a transaction always reads its own writes. The *external consistency axiom* EXT provides that a transaction always reads from the latest write of a transaction. This would be violated if for instance transaction T_2 writes to a key x two times and transaction T_1 reads the first value of x. Than T_1 does not have the final value of x. Moreover EXT provides *atomic visibility*. That is all external reads come from transactions that are committed. In other words, this makes *dirty reads* impossible. If a transaction reads a value form another transaction that ends up aborting, this is called a *dirty read*.

By default Aria provides internal reads INT and external reads EXT. Aria keeps track of write sets of transactions. Before returning a value for a read, Aria checks if a transaction already wrote to that key. If this is the case than Aria returns the transactions own write. This makes sure that a transaction always reads its own writes: INT. Aria also provides EXT. If a transaction did not write to a key, the reads come from a snapshot of committed transacting. Aria buffers writes to the end of the epoch. This ensures that a transaction always reads from a committed transaction and transactions always only expose there latest writes.

Snapshot isolation is stronger than read consistency. Where atomic read consistency suffers from lost update. Snapshot isolation does not allow for a lost update to happen.

Definition 4.2.2 *Snapshot isolation requires* INT, EXT, PREFIX *and* NOCONFLICT.

PREFIX states that if a transaction T_i observes a write of transaction T_j it can also observe all transactions that precede T_j. NOCONFLICT means that no two transactions can write to the same key concurrent. In Aria PREFIX is always provided since if T_i can observe writes of transaction T_j this means that transaction T_i is scheduled in an epoch after T_j. Transactions always read from the committed snapshot. Never from concurrent transactions that are executed in the same epoch. Transactions preceding T_j happen in the same or in a previous epoch as T_j. Therefore T_i observes all transactions preceding T_j.

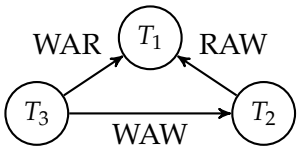


FIGURE 4.6: Dependency graph, example of a WAR, RAW and WAW.

Snapshot isolation is one of the strongest consistency models, however it suffers from anomalies such as write skew. Serializability is stronger than snapshot isolation and does not allow for any such anomalies. The definition of serializability from Cerone et al. Cerone, Bernardi, and Gotsman, 2015 will not be used in this work. But for completeness we give it here below.

Definition 4.2.3 *Serializability is defined to satisfy* INT, EXT *and* TOTALVIS.

By definition TOTALVIS means that the order in which the writes of transactions should be visible to each other must be a total order. A total order is transitive, irreflexive and every two elements in the set are related by this order. Informally this means that every two transactions T_i and T_j can be ordered either with T_j happening before T_i or T_i happening before T_j.

4.2.2 Dependency graph

In Aria Lu et al., 2020 Lu et al. define there own type of dependency graph. Different from the start order serialization graphs introduced in section 4.1, the dependencies of Aria are based on the order of the given input sequence. Each epoch Aria Lu et al., 2020 gets as input a sequence of transactions ordered by the sequencing layer. Three different conflict types are defined between transactions. The conflict types depend on the transaction order produced by the sequencer. Transaction T_i and T_j have a write after read conflict (WAR) if $i < j$ and transaction T_i reads a value from a key that T_j writes to. Transactions T_i and T_j have a read after write conflict (RAW) if $i < j$ and T_i writes to a value that T_j reads. Finally, if transaction T_i and T_j write to the same key, again $i < j$ there is a write after write conflict (WAW) from T_j to T_i. The direction of the dependencies are always from transaction with large id to smaller id.

Consider the following history:

Example 4.2.1 *An epoch consisting of transactions T_1, T_2 and T_3 with the following operations:*

$$T_1 : R(x)R(z)W(x)$$
$$T_2 : R(x)W(y)$$
$$T_3 : W(z)W(y)$$

Transactions T_2 reads a value that transactions T_1 writes to. Note that transaction T_2 does not read the value that T_1 writes. This is not possible since all transactions read from the same snapshot and T_1's writes are not committed yet. However they do conflict on the same key. This results in a RAW dependency from T_2 to T_1. T_3 writes to a key after T_2 also writes to this key, resulting in a WAW from T_3 to T_2. There is a WAR from T_3 to T_1 because T_3 writes to z and T_1 reads z. A dependency graph of this example is given in Figure 4.6. Note here that as stated above, the edges always point in the direction of the smaller transaction id.

4.2.3 Consistency in Aria

Section 4.2.1 explains that Aria provides INT and EXT by default. Now we will see how Aria provides serializability. For this we need to show that Aria satisfies the TOTALVIS axiom. Aria enforces a total visibility order by tracking conflicting operations such as WAR and RAW dependencies.

To produce a serializable output and remain deterministic Aria's default conflict resolution algorithm commits a transactions if:

Rule 1 (Default serializability) *The transactions do not have WAW or RAW dependencies on previous transactions. Defined by Lu et al., 2020.*

If there is a RAW dependency a transaction has a dirty read since it reads an updated value. If there is a WAW two transactions try to write to the same key and only one is allowed to. In theory a WAW or RAW dependency do not necessary produce a non serializable schedule. However the converse is true: If there are no RAW and no WAW the history is serializable. For a proof of why Aria committing transactions according to Rule 1 is serializable see the proof of Lu et al., 2020.

Aria Lu et al., 2020 developed another conflict detection algorithm called deterministic reordering. All transactions in the epoch, although ordered by the sequencer, are executed concurrent. Therefore the transactions can be reordered turning RAW conflict into WAR. As result more transactions can commit. In practice transactions are not reordered. However, the constraints for when a transaction can commit are lowered.

Rule 2 (Deterministic reordering) *A transaction can commit if adheres two following rules: 1) it has no WAW conflicts and 2) it has not both a RAW dependency and a WAR dependency. Defined by Lu et al., 2020.*

The transactions that have no WAW and no RAW dependencies where already able to commit. However if a transaction has a RAW and no WAR this means the transactions can be reordered such that the RAW becomes a WAR. For a proof of why aria committing transactions according to Rule 2 is Serializable see the proof of Lu et al. Lu et al., 2020. Section 4.2.4 provides a more detailed example how deterministic reordering turns a RAW is turned into a WAR.

In the rule below Aria's a conflict detection algorithm is proposed to commit transaction with snapshot isolation. For this the definition of snapshot isolation from Cerone, Bernardi, and Gotsman, 2015, introduced in section 4.2.1, is used.

Rule 3 (Snapshot isolation) *A transaction T_i can commit if it has no WAW dependencies on previous transactions.*

In comparison with Rule 1 and Rule 2 The constraints for a transaction to commit are less strict. A transaction can commit if it has no WAW on preceding transactions.

Theorem 1 *Aria following rule 3 enforces snapshot isolation.*

Proof Section 4.2.1 showed that Aria provides INT, EXT and PREFIX. For snapshot isolation the only thing left to prove is *NoConflict*. Since Rule 3 only commits transactions if there are no WAW conflicts on preceding transactions NOCONFLICT is satisfied. Hence, Aria committing transactions with Rule 3 provides snapshot isolation. □

```
def serializable(
        T, reads, writes
    ):
    if WAW(T, reads, writes) or
    RAW(T, reads, writes):
        return True
    return False

def snapshot_isolation(
        T, reads, writes
    ):
    if WAW(T, reads, writes):
        return True
    return False
```

```
def deterministic_reordering(
    T, reads, writes
    ):
    if WAW(T, reads, writes):
        return True
    if RAW(T, reads, writes) =
    False or WAR(T, reads, writes)
     = False:
        return False
    return True
.
```

FIGURE 4.7: Pseudo code for conflict detection methods: default serializable, deterministic reordering and snapshot isolation. Conflict detection algorithm returns True if a transaction has conflict and False if there are no conflicts.

$T_1 : R(x)W(y)$
$T_2 : R(y)W(z)R(q)$
$T_3 : R(y)R(z)W(p)$
$T_4 : W(q)R(p)$

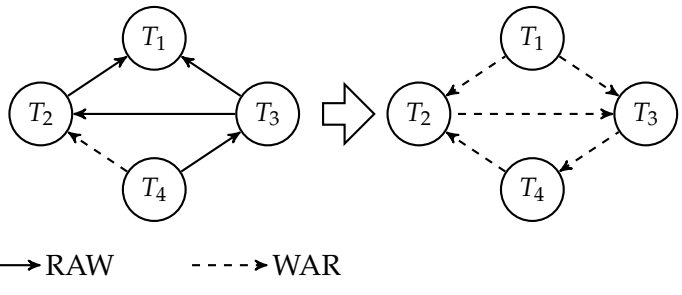


FIGURE 4.8: left from the arrow: A dependency graph of the history of transaction T_1, T_2, T_3 and T_4. Right from the arrow: The RAW's turned into WAR's according to deterministic reordering.

4.2.4 Example: Different conflict detection methods in practise.

Consider the history given in Figure 4.8. The left hand side of the arrow shows the dependency graph before changing the WAR's into RAW's. The only transaction with no WAW or RAW dependencies is transaction T_1. Therefore, according to Rule 1, using the default serializability conflict detection method of Aria, only transaction T_1 can commit. Remember that Rule 2 a transaction can commit if it has no WAW, and not both a WAR and RAW dependency. The transactions of Figure 4.8 that comply with this rule are T_1, T_2 and T_3. T_1 has no dependencies. T_2 and T_3 only have RAW's. Therefore T_1, T_2 and T_3 can commit. However transaction T_4 not. Since T_4 has a WAR and a RAW. The effect of deterministic reordering is shown in Figure 4.8 on the right hand side of the arrow. With reordering, all the RAW dependencies are changed into WAR dependencies. However the resulting graph contains one cycle. To remain an acylcic history T_4 needs to abort. For this case, reordering the transaction to T_3, T_2, T_1 would result in a Serial order. Under snapshot isolation, as implemented according to Rule 3, all transactions are allowed to commit. Because there are no WAW dependencies.

4.2.5 Effectiveness analysis

Pseudo code for the different conflict detection algorithms are listed in Figure 4.7. Compare the algorithms and note that each algorithm relaxes the constraints for a transaction to commit. Serializability requires no WAR or RAW. Deterministic reordering relaxes this by requiring no WAW and no RAW or WAR. Snapshot isolation relaxes the constraint even further and only checks for WAW. From a theoretical point of view snapshot isolation allows for more transactions to commit since it requires less constraints. However, what is the probability that a transaction can commit under snapshot isolation? How does this compare to serializability and deterministic reordering? This chapter build a theory for a statistical effectiveness analysis that gives insights of the theoretical probability that a transaction T_i can commit.

Let us proceed with calculating the probability that a transaction T_i can commit for different isolation models. Suppose the keys $k_1, ..., k_N$ of the database are distributed following some distribution D. A transaction has r reads and w writes from or to keys K_i sampled from this distribution. So every transaction is a set of

$$T_i = \{W(K_{i_1}), ..., W(K_{i_w}), R(K_{j_1}), ..., K(R_{j_r})\}$$

writes and reads. The order is not important. The keys are all independent identically distributed. That is, the keys that transaction T_i accesses do not depend on another transaction T_j for $i \neq j$. The probability that two transactions access the same key $K_i, K_j \sim D$ is calculated as follows:

$$p_{conflict} = P(K_i = K_j) = P(\bigcup_{n=1}^{N} K_i = n \wedge K_j = n) = \sum_{n=1}^{N} P(K_i = n)P(K_j = n).$$

Since K_i and K_j are independent. Later we will evaluate the probability of a single $p_{conflict}$ for different distributions. First lets calculate the probability that a transaction has no WAW, RAW and WAR. The probability that a write operation of transaction T_i has no conflicts is given by:

$$P(\text{operation has no conflicts}) = (p_{no\_conflict})^{iw}$$

Since T_i makes w writes and there are i transactions preceding. Moreover:

$$P(\neg\text{WAW}) = (p_{no\_conflict})^{2iw}$$
$$P(\neg\text{RAW} = (p_{no\_conflict})^{irw}$$
$$P(\neg\text{WAR}) = (p_{no\_conflcit})^{irw}$$

Serializability requires no WAW and no RAW hence:

$$P(T_i \text{ commits serializable}) = P(\neg\text{WAW})P(\neg\text{RAW}).$$

Deterministic reordering can commit if there are no WAW or there are no RAW and WAR.

$$P(T_i \text{ commit deterministic reordering}) = P(\neg\text{WAW})\,(1-(1-P(\neg\text{RAW}))(1-P(\neg\text{WAR})))$$

Snapshot isolation is the most relaxed and requires only no WAW:

$$P(T_i \text{ commits snapshot Isolation}) = P(\neg\text{WAW})$$

To evaluate these probabilities the probability that two keys are equal $p_{conflict}$ still needs to be calculated. However, $p_{conflict}$ is dependent of the distribution of the keys. Below we will give a calculation for $p_{conflict}$ when D is uniform and when D is zipfen generated. The latter because YCSB Gray et al., 1994 uses this one as it gives a good model for high content workloads. In the uniform case all keys have the same probability of getting accessed. the probability of a single key is $P(K = k) = 1/N$ where N is the total number of keys:

$$p_{no\_conflict} = 1 - \sum_{n=1}^{N} P(K_i = n)P(K_j = n) = 1 - \sum_{n=1}^{N} 1/N^2 = 1 - \frac{1}{N}$$

Zipfs law states that the most frequent key is accessed twice as often as the second most frequent and so on. The zipfh distribution has the parameters N for total number of keys, s the skew factor and k is the rank of a key.

$$P(K = k) = \frac{\frac{1}{k^s}}{H_{N,s}}$$

where

$$H_n, s = \sum_{n=1}^{N} \frac{1}{n^s}$$

is the harmonic function that normalizes the probability. Therefore

$$p_{no\_conflict} = 1 - 1/H_{N,s}^2 \sum_{n-1}^{N} \frac{1}{n^{2s}}.$$

The random variable for transactions T_i introduced in the beginning is a binary random variable. $T_i = 1$ if the transaction commit and $T_i = 0$ else. Model the batch B as the sum of T_i

$$B = T_1 + ... + T_b$$

	uniform	zipf
Serializability	70.61	12.96
Deterministic reordering	83.79	18.50
Snapshot isolation	88.59	50.20
$P(no\ conflict)$	0.9999	0.98408

TABLE 4.1: Expected value and $p_{conflict}$ for uniform distributed key value store with 10000 keys, an epoch of $N = 100$ and each transactions making 5 writes and 10 reads. The Zipf distribution has 1000 keys, f z=0.99, but here the transactions make 4 reads and 1 write.

With this, it it possible to calculate the number of expected transactions that can commit in an epoch. For this calculate the expected value of the batch B_i.

$$E(B) = \sum_{i=1}^{B} P(T_i = 1)$$

The probability density of B in Figure 4.9 gives good understanding of different isolation levels and there statistical performance.

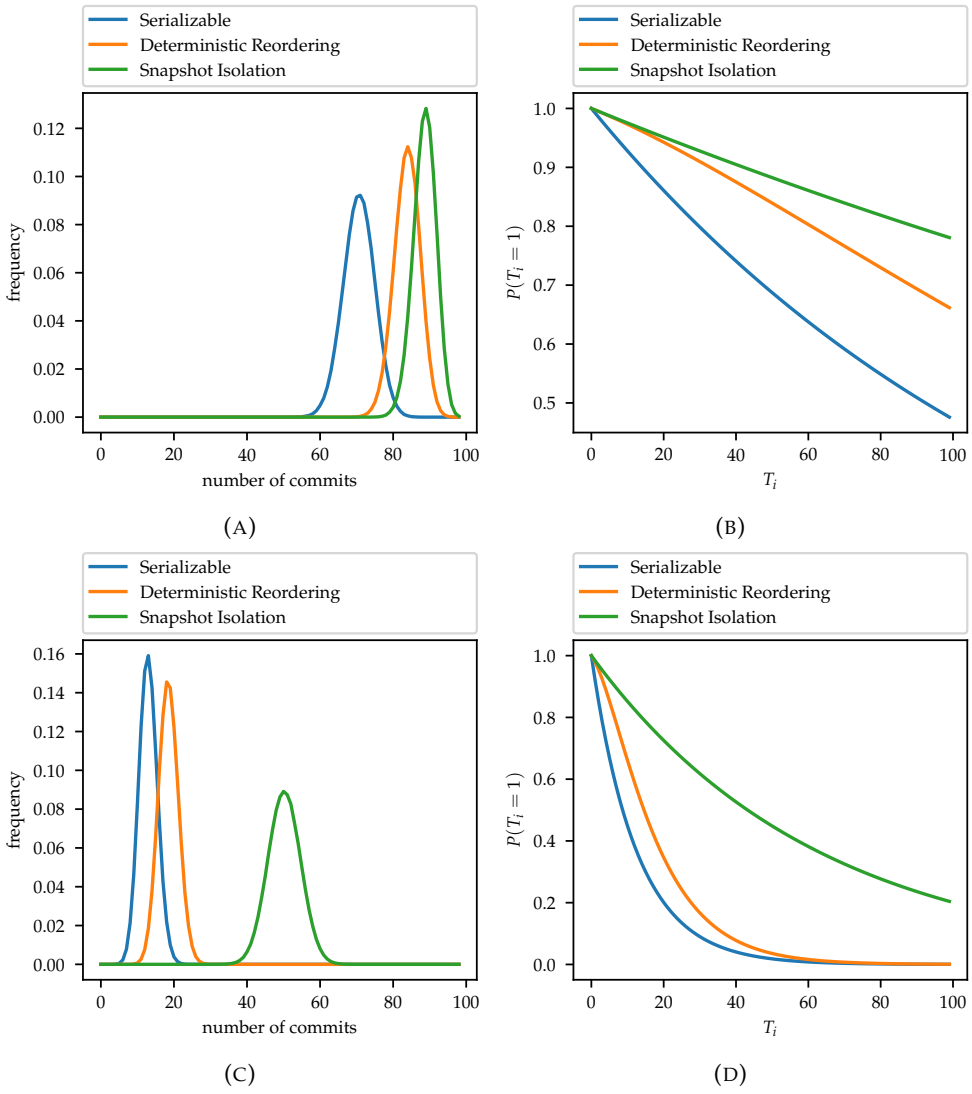


FIGURE 4.9: (a) Probability density functions for different Isolation levels for 10000 uniform distributed keys with batch size 100, 10 reads and 5 writes. (b) corresponding probability of $P(T_i = 1)$ for different isolation levels. (c) probability of $P(T_i = 1)$ for 10000 Zipf generated keys for transaction with 1 write and 4 reads. (d) is the corresponding probability function of $P(T_i = 1)$

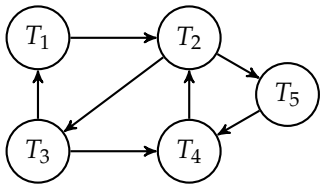


FIGURE 4.10: Start order serialization graph with cycles.

4.3 Serializable reordering on degree

Aria is an epoch based commit scheduler which commits transactions according to a given input order. The sequencing layer determines the order of the transactions before they are processed by Aria. It is possible to reorder the transactions, provided it is done so in a deterministic way. This section explores optimizing the commit order of transactions. We propose an algorithm that commits transactions based on degree. To compare this algorithm to an optimal ordering we formulate an ILP that finds an optimal reordering.

4.3.1 Reordering on degree

The problem of finding an optimal ordering is known as the minimum feedback vertex set problem. This problem is NP-complete Karp, 2010. The feedback vertex set (FVS) of a directed graph is a set of nodes which removal makes the graph acyclic. The minimum feedback vertex set is a FVS of minimal size. When a commit history is represented as a start order serialization graph (SSG), the problem of finding an optimal commit order reduces to minimal FVS. In the SSG transactions are the nodes and edges represent dependencies between the transactions. The degree of a node is the total number of edges connected to that node. The intuition behind the proposed algorithm is that nodes with a higher degree have a larger probability of being in a cycle. Nodes that have a larger degree represent transactions that access more high contention data. Therefore, commit the transactions with the lowest degree first.

The proposed algorithm takes a start order serialization graph $G = (V, E)$ as input. The nodes V of the graph represent the transactions. The edges E are the dependencies between transactions. The nodes are reordered on in degree. The algorithm keeps track of a set W of nodes that will commit. W starts out empty. Add the nodes with the lowest degree to W first. Only add a node to W if non of its neighbors are in W. This condition ensures that the result is acyclic. All the transactions in the result set form a sub graph $G' = (W, E') \subseteq G$ from the original SSG G. The sub graph G' needs to remain acyclic since this will be the actual start order serialization graph when the transactions are committed. For tie breakers the transactions with the smallest transaction id has priority. See Algorithm 1 for a more detailed description of the heuristic.

Some optimizations are used to speed up the algorithm. Al transactions that have no incoming and/or no outgoing edges are removed and can commit. These transactions cannot form a cycle. Tarjan's strongly connected components algorithm Tarjan, 1972 is used to split large start order serialization graphs into smaller connected components.

Algorithm 1 Reordering based on degree.

Input: $G = (V, E)$ a directed start order serialization graph.
Output: $W \subseteq V$ subset of transactions to commit

1: $W = \emptyset$
2: V = SORTONDEGREE(V)
3: **for** $v \in V$ **do**
4: **if** NEIGHBORS(v) $\cap W = \emptyset$ **then**
5: $V = V \cup \{v\}$
6: **end if**
7: **end for**
8: **return** W

4.3.2 Reordering on degree example

Consider the start order serialization graph given in Figure 4.10. First lets count the degree of the transactions. Transaction T_1 has one outgoing edge and one incoming edge. The total degree of T_1 is 2. Transaction T_2 has 2 outgoing and 2 incoming edges. Therefore, degree$(T_2) = 4$. Below the degree of every edge is given:

$$\begin{aligned}\text{degree}(T_1) &= 2\\ \text{degree}(T_2) &= 4\\ \text{degree}(T_3) &= 3\\ \text{degree}(T_4) &= 3\\ \text{degree}(T_5) &= 2\end{aligned}$$

Reordering the transactions on degree from small to large results in the sequence:

$$T_1, T_5, T_3, T_4, T_2.$$

Lets iterate trough the algorithm where we start of with an empty result set: $W = \{\}$. First, add T_1 to the result set, $W = \{T_1\}$. T_5 is next in the sequence. We can add T_5 without creating a cycle in the graph. See Figure 4.10. Resulting to $W = \{T_1, T_5\}$. By the same reasoning it is possible to add T_3 and T_4 to W. Now we have $W = \{T_1, T_3, T_4, T_5\}$. However, it is not possible to add T_2 without creating a cycle. So T_2 is excluded from the result set W. Since there are no more transactions in the sequence to consider, the algorithm terminates. The transactions in the result set W can commit.

Note that Algorithm 1 actually does not check if the adding a remaining node creates a cycle. But only adds a transaction T if the neighbors of T are not in the result set. This ensures the result to be acyclic and is less cost worthy than checking for cycles. This might result in a few cases where transactions are aborted while committing them would not necessarily lead to isolation level violations. As we will see in figure 6.5b in chapter 6.

4.3.3 Determinism

In Aria determinism needs to be maintained to ensure replicas commit the same data. To see that this reordering is deterministic consider the following: When replaying an epoch this results in the same start order serialization graph structure.

Each transactions has the same number of out going and in going edges as before. Tie breakers are solved with transactions id's. Hence replaying an epoch would result to the same transactions to commit. The transactions are all executed against the same snapshot. Thus this method reorders transactions deterministic.

4.3.4 Optimal reordering

The method of Algorithm 1 developed in the previous sections raises the natural question: how far from optimal is this approach. To answer that question we develop a integer linear model (ILP) in this section to optimize the maximal number of transactions that can commit in an epoch. First we develop an ILP that solves for histories when all transactions are directly conflicting. This will come in handy as we will see later in section 6.1 one of the workloads YCSB-B1 has this property. The other ILP solves the optimal commit order for general workloads.

Omitting the read only transactions, given a serialization graph: G. Each edge $e = \{T_1, T_2\} \in E$ in $G = (V, E)$ denotes two transactions that are directly conflicting. Only one can commit. Take $x_i \in \{0, 1\}$ as decision variable to decide if T_i should commit. This leads to the following ILP:

ILP 1 (Only directly conflicting transactions)

$$\begin{aligned} \text{maximize:} \quad & \sum_{j=1}^{m} x_j \\ \text{subject to:} \quad & \forall \{T_i, T_j\} \in E \quad x_i + x_j \leq 1 \\ & \forall i \qquad\qquad\quad x_i \in \{0, 1\} \end{aligned}$$

The first constraint above says: if T_i and T_j are connected by an edge, only one of them is allowed to commit ($x_i = 1$).

When there is not such as nice structure that all transactions are directly conflicting and it is still desirable to obtain an optimal commit order. One can look at all the simple cycles present in the Start order serialization graph and add constraints for each cycle. Given a graph $G = (V, E)$. Suppose the set C contains all simple cycles. Write down a cycle $c \in C$ as a vertex sequence $c = (T_{i_1}, T_{i_2}, \ldots, T_{i_n})$ with $T_i \in V$. Again choosing $x_i \in \{0, 1\}$ as decision variable to decide if T_i should commit.

ILP 2 (General ILP)

$$\begin{aligned} \text{maximize:} \quad & \sum_{j=1}^{m} x_j \\ \text{subject to:} \quad & \forall c = (T_{i_1}, T_{i_2}, \ldots, T_{i_n}) \in C \quad x_{i_1} + x_{i_2} + \ldots + x_{i_n} \leq \|c\| - 1 \\ & \forall i \qquad\qquad\qquad\qquad\qquad\quad x_i \in \{0, 1\} \end{aligned}$$

where $\|c\|$ is the length of the cycle. For every simple cycles in G a constraint is added to the ILP. The sum of the transactions in the cycle must be strict smaller then the length of the cycle. This constraint makes sure that not all transactions of a cycle end up in the result set together.

Example 4.3.1 *ILP for the Start Order Serialization graph of Figure 4.10.*

The ILP needs all simple cycles of the Start Serialization Graph as input. A simple cycle is path that does not contain a node twice except for the start and end node. In this case there are 3 simple cycles. One from T_1 to T_2 to T_3. Denote this by $(T_1T_2T_3)$.

The other cycles are $(T_2T_3T_4)$ and $(T_2T_5T_4)$. The length of the cycle is defined as the number of nodes in the cycle. Here, all the cycles have length 3.

To formulate the ILP, introduce the decision variables x_1, x_2, x_3, x_4 and x_5. These are binary values: $x_i \in \{0,1\}$. The variable $x_i = 1$ if T_i ends up in the result set. Conversely, $x_i = 0$ if T_i is excluded from the result set. The objective is to maximize the number of transaction that can commit. Therefore this ILP maximizes over the sum of the decision variables. For every simple cycle add a constraint to ensure that the result set is acyclic. For instance for the cycle (T_1, T_2, T_3) with length 3 add the constraint:

$$x_1 + x_2 + x_3 \leq 2.$$

This means that x_1, x_2 and x_3 cannot be 1 all together. In that case $x_1 + x_2 + x_3 = 3 \nleq 2$. In other words, this constraint makes sure that T_1, T_2 and T_3 are not in the result set altogether. At least one needs to be excluded from the result set. Therefore the cycle $(T_1T_2T_3)$ would not appear in the result set. The ILP is formulated as follows:

$$\begin{array}{lll} \text{maximize:} & x_1 + x_2 + x_3 + x_4 + x_5 & \\ \text{subject to:} & x_1 + x_2 + x_3 & \leq 2 \\ & x_2 + x_3 + x_4 & \leq 2 \\ & x_2 + x_4 + x_5 & \leq 2 \\ \forall i & x_i \in \{0,1\} & \end{array}$$

If an ILP solver solves this ILP the optimal solution would be: $x_1 = 1, x_2 = 0, x_3 = 1, x_4 = 1$ and $x_5 = 1$. This means that committing transactions $W = \{T_1, T_3, T_4, T_5\}$ is the optimal commit order of the Start Order Serialization graph of Figure 4.10.

One of the downsides of this methods is that for high content workloads the number of simple cycles does not scale well. According to Jhonsen a fully connected graph of size n contains

$$\sum_{i=1}^{n-1} \binom{n}{n-i+1} (n-i)!$$

simple cycles Johnson, 1975. In high content workloads the start order serialization graph has a high connectivity degree and contains many cliques. A clique is a fully connected sub graph. The number of cycles is in high contention workloads is large. This results to a non-polynomial number of constraints in the ILP.

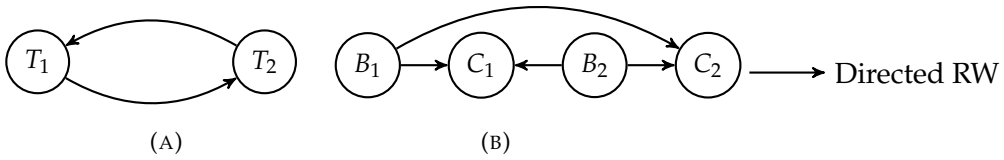


FIGURE 4.11: Read write anti dependency cycle in SSG in (a) results in an acyclic BC-graph in (b).

4.4 Snapshot isolation reordering on degree

This chapter optimizes the reordering of transactions under snapshot isolation consistency model. A new graph structure: BC-graph is introduced and used to optimize snapshot isolation. The approach of the previous chapter is extended to work on the BC-graph structure in algorithm 2. We develop ILP 3 and compare this optimal solution with Algorithm 2.

4.4.1 Constructing BC-graphs

Adapting Algorithm 1 to work for snapshot isolation is not straight forward. On a start order serialization graph, the serializable isolation model is equivalent to obtaining an acyclic graph. However snapshot isolation allows cycles with at least two adjacent read write anti dependencies Fekete et al., 2005. This makes finding an optimal commit order even more complicated than finding the maximal feedback vertex set. An algorithm that checks if a history is consistent with snapshot isolation on start order serialization graphs would need to iterate through all cycles and check if there are at least two adjacent read write edges. Luckily there is another data structure called BC-graphs on which snapshot isolation is proven to be acyclic Zhang et al., 2023. A BC-graph is formed from a start order serialization graph (SSG). For every transaction T_i in the SSG create two nodes B_i, C_i and an edge between these nodes. For every read write anti-dependency between T_i to T_j create an edge from B_i to C_j. For each write-read or write-write edge between T_i and T_j. create an edge from C_i to B_j. See Figure 4.11 for an example of a snapshot isolation history that creates a cycle in the SSG but is acyclic in the BC-graph.

Algorithm 2 adjust Algorithm 1 to work for snapshot isolation on BC-polygraphs. The Algorithm keeps the same approach by reordering the nodes on degree. However, when a node B_i is a candidate to add to the results set. Besides from checking its own neighbors, it needs to check if its partners C_i neighbors are not already in the result set.

This algorithm might not be the most optimal. To see this apply algorithm 2 to the example in Figure 4.11b. B_1, B_2 both have the smallest in-degree: 0, lets choose B_1 to add to the result set W first. Then B_1's partner: C_1 also needs to be added to the result set W. The algorithm cannot add nodes B_2 and C_2 from transaction T_2 since C_2 is $B_2's$ neighbor and is already in the result set. However T_2 is allowed to commit under snapshot isolation consistency model. This might lead to transactions being unnecessary aborted. Algorithm 3 solves this problem by checking whether adding a node would add a cycle to the graph. However this algorithm is more costly. Checking for cycles in a directed graph is often implemented with a breath first search. Doing a breath first search every time a node is added is quite costly. There are some different approaches that keep track of each nodes ancestors. Search complexity will be $O(1)$ however when adding a node all ancestors need to be updated.

Algorithm 2 Reordering based on degree for BC-graphs.

Input: $G = (V, E)$ a directed BC-graph.
Output: $W \subseteq V$ subset of transactions to commit

```
1: W = ∅
2: V = SortOnInDegree(V)
3: for v ∈ V do
4:     if neighbors(v) ∩ W = ∅ then
5:         p = partner(v)
6:         if neighbors(p) ∩ W = ∅ then
7:             W = W ∪ {v, p}
8:         end if
9:     end if
10: end for
11: return W
```

Algorithm 3 Check if adding a node creates a cycle

Input: $G = (V, E)$ a directed BC-graph.
Output: $W \subseteq V$ subset of transactions to commit

```
1: W = ∅
2: V = SortOnInDegree(V)
3: for v ∈ V do
4:     E' = {(a, b) ∈ E : a, b ∈ W ∪ {v}}
5:     H = (W ∪ {v}, E')          ▷ H is a sub graph of G with all nodes of W ∪ {v}
6:     if acyclic(H) then
7:         W = W ∪ {v}
8:     end if
9: end for
10: return W
```

4.4.2 ILP formulation for snapshot isolation

Formulating an ILP for Snapshot isolation is almost similar to ILP 2 of section 4.3.4. Using the BC-graph structure an optimal commit order needs to require the BC-graph to be acyclic. Take $B_i, C_i \in \{0,1\}$ as decision variables whether or not to include a node into the result set. The objective is to maximize the total number of nodes include. A constraint needs to be added to ensure that if B_i is chooses (i.e. $B_i = 1$) then C_i is chosen to. Similarly as in ILP 2 for every simple cycle in the graph a add a constraint to the ILP. Suppose again C is the set of all simple cycles. A cycle $c = \{v_1, ..., v_n\} \in C$ is represented as a sequence of nodes. The nodes in the cycle can be C-nodes or B-nodes.

ILP 3 (Snapshot Isolation)

$$
\begin{array}{lll}
\text{maximize:} & \sum_{j=1}^{m} B_j + C_j & \\
\text{subject to:} & \forall c = (v_1, \ldots, v_n) \in C & v_1 + ... + v_n \leq \|c\| - 1 \\
& \forall i & B_i = C_i \\
& \forall i & B_i, C_i \in \{0,1\}
\end{array}
$$

With $\|c\|$ the length of the cycle. Note that the condition $B_i = C_i$ requires that if one of the nodes belonging to transaction T_i is included in the result set also its partner is included in the result set.

As practical optimization for finding the optimal solution in the ILP above, first find all cycles consisting of 2 vertices. Add constraints for these cycles to the ILP. Then, it is possible to remove the edges between these cycles, reducing the total amount of simple cycles in the remaining graph.

Chapter 5

Communication

Serializability Papadimitriou, 1979 is the golden standard when it comes to distributed transactions. To achieve serializability extensive communication is required between partitions Cerone, Bernardi, and Gotsman, 2015. Only weak isolation levels such as read atomic and casual consistency do not need coordination between partitions Bailis et al., 2016. Isolation levels such as parallel-snapshot isolation and stronger require coordination Sovran et al., 2011. This chapter explores the different design choices that are possible in Aria in terms of communication and coordination. More specifically the chapter explains which conflict detection algorithms need the local read write sets or the global read write sets. Table 5.1 shows an overview of which methods needs which read write set.

In a distributed system like Aria it is desirable that the algorithms that are used are as distributed as possible. In a sense this means that the algorithms used are non-repetitive. The calculations that are executed on worker 1 are not executed the exact same way on worker 2. The work is divided over the workers. However, within Aria there is a trade off between message complexity and distribution of the algorithm. Remember from chapter 3 that aria processes transactions in two phases. 1) The execution phase and 2) the commit phase. After phase 1 every worker has partial knowledge about the read and write sets. They only know the reads and writes that happened on there partition of the data. The next chapters describe in more depth when in the commit phase communication is needed and how conflict detection can be distributed as much as possible.

5.1 Communication in Aria's conflict detection

During the commit phase there are two possibilities for the conflict detection algorithm to communicate. Depending on the conflict detection algorithm 1) the workers need the global read-write sets at the beginning of the commit phase or 2) the workers only need the local read write sets but need to communicate which transactions need to abort. Consider the following example:

Method	local read set	local write set	global read write sets
Default serializable	✓	✓	
Deterministic reordering			✓
Snapshot isolation		✓	
Reorder on degree			✓

TABLE 5.1: Sets needed to resolve conflicts per conflict detection methods.

Example 5.1.1 *Suppose Aria is configured to have two workers. Key x is located on worker 1 and key y is located on worker 2. There are two transactions T_1 and T_2:*

$$T1: \quad R(x)R(y)W(x)$$
$$T2: \quad R(x)R(y)W(y)$$

After execution phase the workers have the following read write sets. Worker 1 has read set: $\{x : \{T_1, T_2\}\}$ and write set: $\{x : \{T_1\}\}$. While worker 2 has read set: $\{y : \{T_1, T_2\}\}$ and write set: $\{y : \{T_2\}\}$. The workers do not have total knowledge over the reads and writes in the system. If the workers where to resolve conflicts at this point, without sharing the read write sets, the following would happen. Transaction T_1 has no dependencies on previous transactions, since it is the first transaction in the sequence. Therefore, T_1 can commit In worker 1 transactions T_2 reads a value that T_1 writes to. So transaction T_2 has a RAW dependency on T_1. In worker 2 transaction T_2 writes to y after x reads y. This results in a WAR dependency. What happens next depends on the configured conflict detection algorithm.

5.1.1 Default serializability

The default serializability algorithm is able to detect which transactions need to abort without communicating the read write sets up front. According to Rule 1 of Chapter 4.2 transactions can commit if there is no WAW or no RAW. Following this rule worker 1 would abort T_2 since it has a RAW dependency on T_1. However worker 2 does not find a reason to abort T_2. Thus a network round trip is needed for the workers to inform there peers which transactions need to abort.

5.1.2 Deterministic reordering

Suppose deterministic reordering is configured. As described in Rule 2 of Chapter 4.2. Recall a transaction can commit if there is no WAW and not both a RAW and a WAR. Worker 1 would commit T_2 since it only has a RAW. Similarly, Worker 2 would commit T_2 since it has only a WAR dependency. While if the system would have total knowledge about the dependencies it would notice that there is a WAR and a RAW and the system needs to abort transaction T_2.

The difference with deterministic reordering is that in Rule 2 the condition is composed of an AND clause. Just as in the example, if one RAW is located in one worker and a WAR is located on a different worker, the workers would fail to detect this anomaly. Therefore, deterministic reordering needs to communicate read write sets before resolving conflicts.

5.1.3 Snapshot isolation

Snapshot isolation only checks for WAW. Similar as with Default Serializability this can be done locally on the worker.After resolving conflicts a network round trip is needed to inform there peers which transactions need to abort.

5.1.4 Communicating read write sets or aborts

It is possible to configure snapshot isolation and default serializable to communicate read write sets before resolving conflicts, or communicating aborts after. The communication complexity of broadcasting read write sets or broadcasting aborts is the same. If the workers know the read write set before conflict detection they

can calculate all the aborts and do not need to communicate after resolving conflicts. On the other hand, if all the workers calculate aborts without sharing the read write sets first, then they need to inform the other workers which transactions need to abort. Sharing aborts only has the advantage that the work is more distributed. No two workers do the same calculation. However the workers do not have a total overview of the complete read write set of in the system. Therefore it could happen that Worker 1 aborts transaction T_1 and worker 2 aborts T_2. Afterwards they share there aborts and they conclude that both T_1 and T_2 need to abort. While if they had a total overview of the read write set they could choose between aborting T_1 or T_2 Another advantages of broadcasting before resolving conflicts is that each worker gets a total view of the logic aborts. The workers do not commit transactions that are already aborted by logic on another worker. Resulting in less conflicting transactions to begin with.

5.2 Graph optimization algorithms

The conflict detection algorithms that use the start order serialization graphs need a total view of the read/write set of all transactions in order to resolve conflicts. The workers need to make one network round trip to share read write sets. After that, each worker can build a start order serialization graph and start the algorithm. A small optimization is possible. That makes the algorithm a bit more distributed. Consider the following start order serialization graph $G = (V, E)$ with transactions V and dependencies between transactions described by E. A typical start order serialization graph consists of multiple connected components.

$$G = C_1 \cup ... \cup C_n,$$

where $C_i = (V_i, E_i)$ such that $V_i \cap V_j = \emptyset \quad \forall i, j$ and $\bigcup_i E_i = E$. That is, there are no edges between connected components and the connected components do not share nodes. Each worker can use Tarjans algorithm to find connected components Nuutila and Soisalon-Soininen, 1994. A worker only needs to solve for the connected components C_i that contains a transactions that writes to one of keys located at there own partition.

$$\{C_i : \exists T_j \text{ s.t. write\_set}(T_j) \cap C_i \neq \emptyset\}$$

In this way workers do not have to do work for transactions that have no writes at that workers partition. Moreover after resolving conflicts no extra network round trip is needed to inform peers about aborts. Because all the workers find the same aborts.

5.3 Distributed or coordinated

In the distributed setup each worker needs to broadcast there read write sets to there peers. In terms of message complexity, for n workers this results to $n(n-1)$ messages. This does not scale efficiently when the number of workers grows. Another solution is to configure the system to use a worker coordinator setup. The workers share there read write sets with the coordinator. The coordinator then has the total view of the global start order serialization graph. The coordinator then resolves the conflicts and informs the workers which transactions need to abort.

Chapter 6

Experimental results

This chapter evaluates the methods proposed in this thesis. The different benchmarks used for evaluation are introduced in section 6.1. Section 6.2.1 shows the performance of lowering the isolation level to snapshot isolation. This is without any order optimization. The experiment in section 6.2.2 presents the results of serializable ordering on degree. Section 6.2.3 shows the results of reordering on degree for snapshot isolation. The results of different communication strategies from chapter 5.1 can be seen in section 6.3. Section 6.4 shows the results of all the methods integrated in the system with the fallback mechanism turned on. In section 6.4 an experiment is conducted to see the influence of the skew factor on the performance of the methods. To compare the conflict detection algorithms ability to scale out, chapter 6.4.2 caries out an experiment that shows the effect of adding multiple workers to the system. In some of the figures reordering on degree is abbreviated to ROD.

6.1 Benchmarks

The benchmarks used in this work are adopted from the Yahoo! Cloud Serving Benchmark (YCSB) Cooper et al., 2010. There are two types of variants of YCSB workload: 1) YCSB-A where all write operations are independent of all the read operations. 2) YCSB-B where all writes dependent on all the reads. For this work the YCSB-B benchmark is adopted into two variants. One variant YCSB-B1 where a transaction consists of one operation of YCSB-B. The other variant is called YCSB-B5 and consists of five operations of the YCSB-B workload, combined into one transaction to create a transactional workload Wang and Kimura, 2016. Table 6.1 gives an overview of the two workloads and there characteristics.

6.1.1 Default workload parameters

If not otherwise stated the following default system specifications apply: The workloads configured to have an 80 to 20 ratio reads and writes. The system is configured to have two worker nodes. Every workload consists of 10K transactions.

Workload	Transaction operations	Snapshot optimizable
YCSB-B1	Update, read, transfer	✗
YCSB-B5	5 sequenced update/reads	✓

TABLE 6.1: Workload overview.

```
# Get the value from the key value store.
def read(key_1):
    return storage.get(key_1)

# Read the value from key 1 and write the update value to storage.
def update(key_1):
    x = storage.get(key_1)
    x = x + 1
    storage.put(key_1, x)
    return x
```

```
# Transfer mony from accont with key 1 to key 2.
def transfer(key_1, key_2):
    x = storage.get(key_1)
    # Check the constraint violation.
    if x - 1 > 0:
        update(key_2)
        x = x - 1
        storage.put(key_2, x)
    else:
        raise NotEnoughCreditException()
```

FIGURE 6.1: Read, update and transfer operations form YCSB.

```
# Create a transaction from a sequence of YCSB operations
def sequenced_operations(key_operation_set: Map[Key, str]):
    for key, operation in key_operation_set.items()
        if operation == 'update':
            update(key)
        if operation == 'read':
            read(key)
        if operation == 'transafer':
            key2 = generate_key()
            transfer(key, key2)
```

FIGURE 6.2: Create a transaction from a sequence of YCSB operations. Note that `generate_key` is the zipfian key generation function of YCSB.

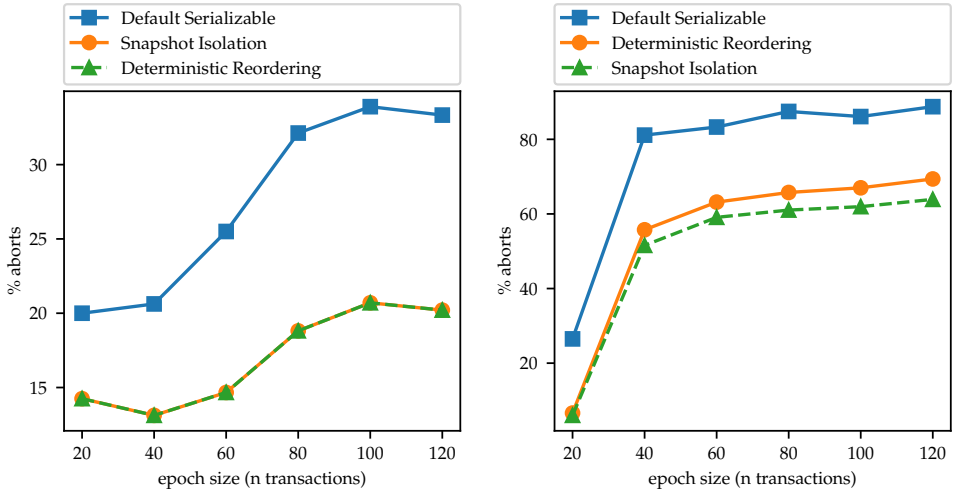
6.1.2 Workload YCSB-B1

This workload consist of small transactions. All writes depend on a read to the same key. It is a common workload in online transaction processing. It is the default workload of the Yaho! Cloud Serving Benchmark (YCSB) Cooper et al., 2010. The workload consists of three different operations. A read operation to a single key, an update and a transfer. An update is essentially a read to a key followed by a write to the same value. A transaction takes two keys. In this case it is a positive update to the first key followed by an negative update to the second key. There is a constraint on key one and the transaction fails if the value is negative after subtracting. Pseudo code of the implementation of these operations can be found in Figure 6.1.

6.1.3 Workload YCSB-B5

Workload YCSB-B5 consists of larger transactions. This workload contains 5 operations of the YCSB-B1 workload batched together to create a transactional workload Wang and Kimura, 2016. Figure 6.2 shows how the YCSB operations can be transformed to a transaction of sequenced operations. The function takes as input a map of keys and operations and delegates the operation with the corresponding key.

Two important properties of this workload are: 1) A write can depend on multiple reads to different keys. 2) A write is still always part of an update and therefore also dependent on a read to the same key.



(A) YCSB-B1 (B) YCSB-B5

FIGURE 6.3: Result of snapshot isolation, deterministic reordering and default Serializability on offline benchmarks YCSB-B1 (a) and YCSB-B5 (b) for different epoch size.

6.1.4 Offline workload

To compare the performance of different conflict detection algorithms an offline transactional database is created. The read write sets from a run of YCSB-B1 and YCSB-B5 are saved to this database. Configured for different epochs sizes. With this offline database, conflict detection algorithms are quickly compared without the need of spinning up the system.

6.2 Offline experiments

This section shows the outcome of the experiments on the offline benchmark. Section 6.2.1 shows that lowering the isolation level to snapshot isolation allows for more transactions to commit per epoch. Section 6.2.2 shows the effects of reordering transactions on degree for serializable isolation level compared with deterministic reordering. Section 6.2.3 shows the effect of reordering on degree for snapshot isolation. Compared with snapshot isolation without reordering.

6.2.1 Snapshot isolation

This shows the results of lowering the isolation level to snapshot isolation. The conflict detection algorithm for snapshot isolation is developed in section 3. Snapshot isolation is compared with default serializability and deterministic reordering. Figure 6.3a shows the performance of the methods on YCSB-B1. Deterministic reordering and snapshot isolation outperform serializability. However snapshot isolation and deterministic reordering perform similarly. This can be explained by the fact that YCSB-B1 consists of short transactions. In the workload there are only reads, updates and transfers. Compare the constraints in Rule 2 of deterministic reordering and Rule 3 of snapshot isolation. Because workload YCSB-B1 consists of small transactions, all writes are part of updates. If a transactions has a RAW or WAR this means it has also a WAW. Therefore there are no transactions that are allowed

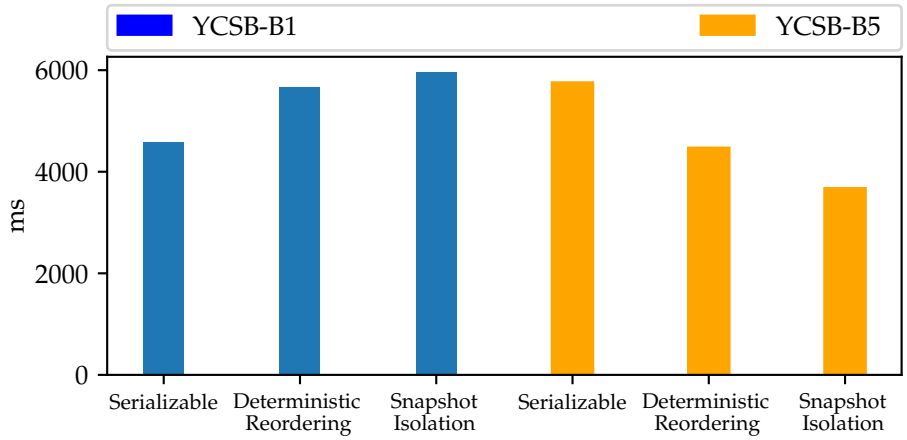


FIGURE 6.4: Average latency from 10 runs in Aria for different conflict detection methods: serializable, deterministic reordering and snapshot isolation on the YCSB-B1 and YCSB-B5 workload.

by snapshot isolation but not by deterministic reordering. The workload is not optimizable by lowering the isolation level to snapshot isolation, see section 4.1 for more details. On YCSB-B5 however, as figure 6.3b shows, snapshot isolation outperforms deterministic reordering and serializability. This is consistent with the effectiveness analysis of section 4.2.5.

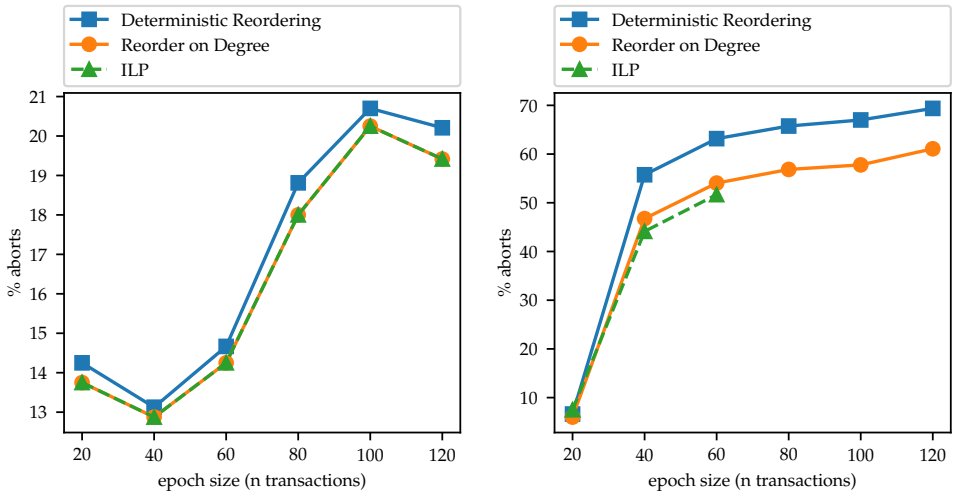
The results in Figure 6.4 show the average latency of Aria on YCSB-B1 and YCSB-B5 in the online system configured with two workers with the fallback off. This figure shows that snapshot isolation outperforms Serializability and deterministic reordering on the workload YCSB-B5 but it shows worse performance on YCSB-B1. This corresponds with the results form Figure 6.3. Snapshot isolation and deterministic reordering performing worse on YCSB-B1 could be a consequence of serializable conflict detection algorithm terminating faster.

6.2.2 Serializable reordering on degree

In Figure 6.5 serializable reordering on degree (Algorithm 1), deterministic reordering and the optimal ordering of ILP 2 are compared on workload YCSB-B1 and YCSB-B5. Configured with different epoch sizes. The plot shows that serializable reordering on degree outperforms deterministic reordering and is very close to the optimal solution of ILP 2. Note here that the solution of the ILP is only calculated up till epoch size 60. For larger epoch sizes it is not possible to construct the ILP. There are to many simple cycles in the graph.

6.2.3 Snapshot isolation reordering on degree

In Figure 6.6 Algorithm 2 (snapshot isolation reorder on degree) and Algorithm 3 (sapshot isolation reorder on degree check acyclic), as presented in section 4.4, are compared with snapshot isolation of section 4.2 (without reordering) as baseline. The left plot shows the number of aborts. The right plot shows the run time of the algorithms. Both Algorithm 2 and Algorithm 3 outperform the baseline and are very close to the optimal ordering calculated by ILP 3. From the left plot of Figure 6.6 follows that Algorithm 3 outperforms Algorithm 2. They are both very close to the optimal solution. However the right plot of Figure 6.6 shows that the run time of



(A) YCSB-B1 (B) YCSB-B5

FIGURE 6.5: Comparison of deterministic reordering, serializable reordering on degree (Algorithm 1) and optimal solution (ILP 2 on YCSB-B1 (a) and YCSB-B5 (b).

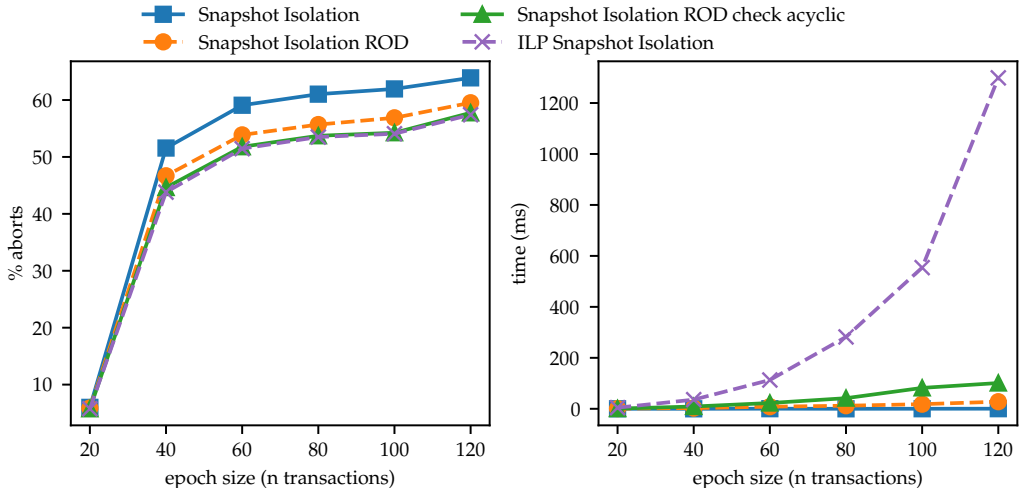


FIGURE 6.6: Comparison of snapshot isolation, snapshot isolation reordering on degree, snapshot isolation reordering on degree check acyclic and ILP 3 on YCSB-B5.

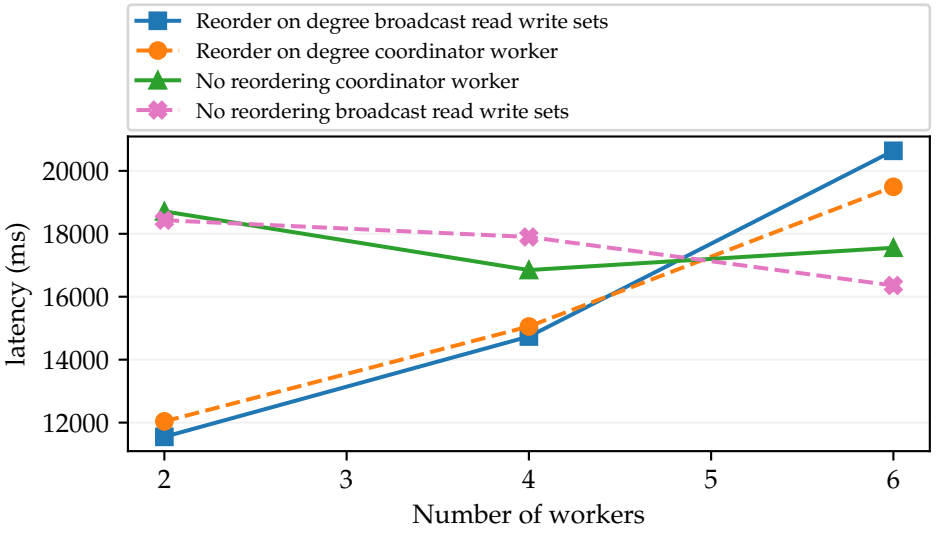


FIGURE 6.7: Different communication configurations for snapshot isolation on YCSB-B10. Configured with 8 reads and 2 writes and a zipf constant of 0.7.

Algorithm 3 is twice as long as 2. Therefore Algorithm 2 is favored over Algorithm 3.

6.3 Coordinated or distributed

Figure 6.7 shows the results of how different snapshot isolation conflict detection methods scale out. Snapshot isolation reordering on degree is compared with snapshot isolation without reordering. The methods are configured with different communication strategies from chapter 5. When the system is configured with 2 and 4 workers, snapshot isolation reordering on degree outperforms snapshot isolation with no reordering. However configured with 6 workers no reordering outperforms the reordering methods. Whether the system is configured in a coordinator worker setup or broadcasting read write sets does not seem to influence the performance.

6.4 Fallback on

This section shows the results of the proposed conflict detection algorithms integrated in Aria configured with the fallback mechanism turned on. Table 6.2 shows the 50 and 90 percentile latency's of default serializable, deterministic reordering, snapshot isolation, serializable reordering on degree and snapshot isolation reordering on degree. The table shows that lowering the isolation level to snapshot isolation reduces the latency with about 8 milliseconds for the 50 percentile compared with deterministic reordering. For the 99 percentile lowering the isolation level reduces the latency with about 40 milliseconds. The reordering methods are twice as fast on the 50 percentile. For the 99 percentile serializable reordering on degree is very close to snapshot isolation. However snapshot isolation reordering on degree is almost twice as fast as serializable reordering on degree.

The column that shows the average percentage of aborts of table 6.2 also has an interesting outcome. Lowering the isolation level to snapshot isolation allows for 0.73% more aborts average. This is somehow similar to the results on the offline

Method	Latency(ms)		average abort % per epoch
	50p	99p	
Default serializable	29	145	3.37
Deterministic reordering	30	117	2.73
Snapshot isolation	22	104	2.02
Serializability reordering on degree	17	97	0.09
Snapshot isolation reordering on degree	16	49	0.08

TABLE 6.2: Final results developed methods integrated in system. Configured with as input 300 transactions/second on the zipf distribution with skew parameter $s = 0.7$. Each transaction makes 2 writes and 8 reads. The number of workers is 4 total. Fallback mechanism is used at an abort rate of 0.1.

workload on YCSB-B5 that figure 6.3b shows. However the reordering methods have an average abort rate of 0.9% and 0.8% for deterministic respectively snapshot isolation reordering. This is way more than the increase of the reordering methods on YCSB-B5 that figure 6.5b and figure 6.6 show. This could partially be explained by the fact that the results of table 6.2 are from benchmark YCSB-B10 and these are even longer transactions.

6.4.1 Skew factor

The skew factor plays important role on performance of distributed databases. This work sets out to minimize the number of aborts in high contention workloads. The experiment in figure 6.8 is conducted to find out what the influence of the skew factor is on the in this thesis developed methods. Figure 6.8 shows that for low contention workloads, when the skew factor is 0.0 or 0.4, the reordering methods perform similar as the methods without reordering. For more high contention workloads, with skew factor 0.8, 0.9 and 0.99, the reordering methods clearly outperform the other methods. Moreover, figure 6.8 shows that lowering the isolation level to snapshot isolation is especially effective for high contention workloads.

6.4.2 Scale out

Aria is designed to partition data across multiple nodes and scale out horizontal;y. The experiment of figure 6.9 aims at investigating how the conflict detection methods scale out when multiple workers are added to the system. To keep the workload balanced, the experiment is configured to have as input throughput: number of workers $\cdot$ $50 + 200$ transactions per second. For two workers this is 300 transactions/seconds, four workers get as input 400 transactions/seconds and so on. Figure 6.9 shows that the reordering methods, for snapshot isolation and serializable isolation level, are able to scale out and outperform non-reordering methods for up to 8 workers.

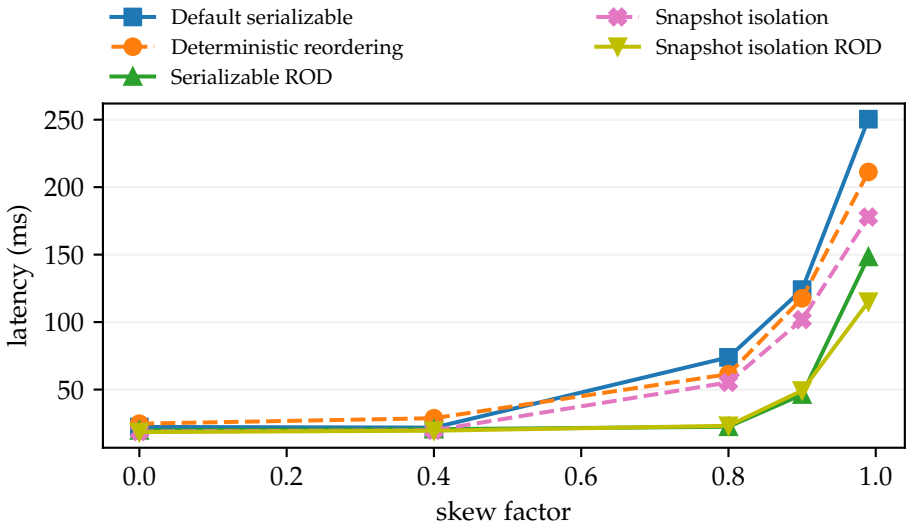


FIGURE 6.8: System performance with fallback mechanism on for skew factor ranging from 0 (uniform) til 0.99. Benchmark YCSB-B10 is used with 300 transactions/second as throughput. Configured with 4 workers total.

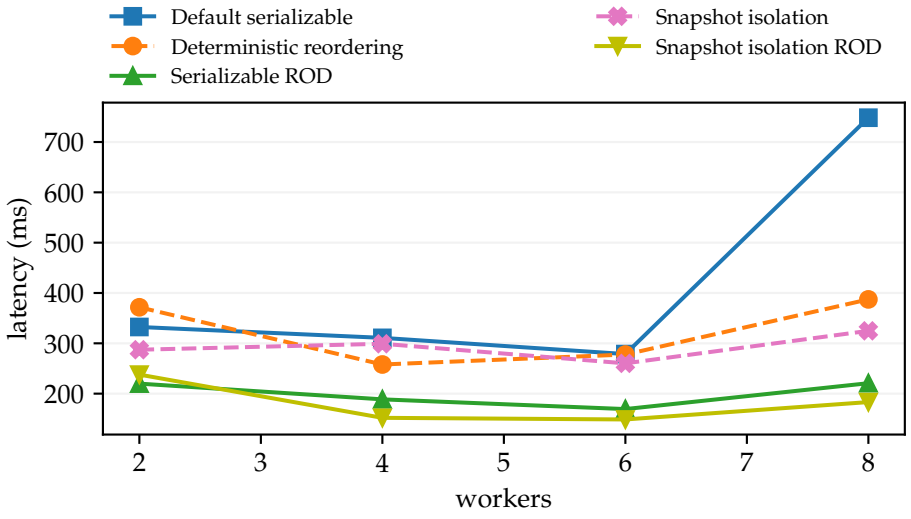


FIGURE 6.9: Scaling out horizontally with fallback mechanism on. Aria performance configured with fallback mechanism on, skew factor 0.9 and as input throughput: number of workers · 50 + 200 (transactions/second).

Chapter 7

Conclusion

First, this chapter summarizes the most important results and findings. Then, this chapter answers the research questions as posed in the introduction. The goal of this thesis is to minimize the number of aborts in epoch based commit scheduler like Aria for high contention workloads. To this end 2 solutions where proposed: 1) lowering the isolation level to snapshot isolation and 2) reordering the order of the transactions. Figure 4.9c of chapter 4.2.5 shows the probability density function of snapshot isolation compared with deterministic reordered and default serializability. The figure shows that snapshot isolation is expected to commit more transactions compared with deterministic reordering. The theory of chapter 4.1.3 explains why workloads consisting of short transactions are not able to commit more transactions when lowering the isolation level to snapshot isolation. Figure 6.3a and figure 6.4 of section 6.2.1 confirm this theory. Figure 6.3a shows that lowering the isolation level to snapshot isolation does not improve the number of aborts on YCSB-B1. Figure 6.4 shows that on YCSB-B1 in a live system setting with the fallback mechanism turned off, the average latency is not improved by lowering the isolation level to snapshot isolation.

On the other hand, lowering the isolation level to snapshot isolation on YCSB-B5 reduces the total number of aborts by 3% (figure 6.3b) and improves the latency with 1000 ms when the fallback mechanism is turned off (figure 6.4). Moreover, figure 6.5 shows that reordering the transactions on degree allows for more transactions to commit within one epoch than deterministic reordering. Similarly, reordering the sequence for snapshot isolation reduces the number of aborts (figure 6.6). Both methods are close to the optimal solution calculated by the ILP's.

Integrated in the system, with the fallback mechanism turned on the reordering methods are able to improve latency (table 6.2). On uniform workloads, when the skew factor is lower, the reordering methods perform similar as the other methods (figure 6.8). The higher the skew factor, the more the effect lowering the isolation level has on the average latency (figure 6.8). The methods that reorder transactions on degree are able to scale out and perform better than the non reordering methods for Aria configured with up to 8 workers.

Below we answer the research questions.

1. Does performance of deterministic database increase when lowering the isolation level from serializability to snapshot isolation?

On workloads that consist of short transactions, such as YCSB-B1, performance does not increase when lowering the isolation level to snapshot isolation. This is shown in figure 6.3a. For longer transactions, such as YCSB-B5 as show in figure 6.3b, snapshot isolation allows to commit about 3% more transactions per epoch. Figure 6.4 shows Aria configured with the fallback mechanism off in a live system setting. The same results apply. Lowering the isolation level to snapshot isolation results to an

improved latency of 1000 ms on YCSB-B5. For smaller transactions lowering the isolation level does not improve the latency. Such as on YCSB-B1.

When Aria is configured with the fallback mechanism on similar result are observed. Snapshot isolation reduces the average latency. Figure 6.8 shows that the higher the skew factor the more the latency is reduced. Without reordering, for a skew factor of 0.99 snapshot isolation improves the latency with about 30 ms. Moreover, for the same skew factor, when comparing snapshot isolation reordering on degree with serializable reordering on degree, the snapshot isolation reordering on degree outperforms serializable reordering on degree with about 30 ms.

2. Can the number of aborted transactions per epoch be minimized for serializable isolation level?

For small transaction, on YCSB-B1, reordering on degree minimizes the number of commits about 1% (Figure 6.5a). For larger transactions, on YCSB-B5 serializable reordering on degree is able to abort 5% more transactions per epoch compared with deterministic reordering (Figure 6.5b).

3. Can the number of aborts be minimized under snapshot isolation?

Figure 6.6 shows for snapshot isolation that reordering on degree allows for around 3% more transactions to commit per epoch on YCSB-B5.

4. How close are the reordering heuristics to an optimal commit order?

Snapshot isolation and serializable reordering on degree are both close to the optimal solutions calculated by the ILP's. Figure 6.5a shows that serializable reordering on degree is optimal. In figure 6.5b one can see that serializable reordering on degree is within a 1 percent distance close to the optimal solution. Figure 6.6 shows that snapshot isolation reordering on degree is within 1 percent close to the optimal solution calculated by the ILP. The reordering method that uses the more precise but more cost-worthy algorithm to check for cycles performs optimal. However figure 6.6 also shows that the time performance of snapshot isolation reordering on degree check acyclic is worse than snapshot isolation reordering on degree.

5. Does Aria perform better broadcasting read write sets or in a master worker setup?

Figure 6.7 shows that for performance coordinator worker or broadcasting configuration perform similar.

6. When does lowering the isolation guarantee to snapshot isolation result in better performance?

The results in figure 6.3a show that for short transactions lowering the isolation guarantee to snapshot isolation does not allow for more transactions to commit. Chapter 4.1 explains this by observing the patterns in the start order serialization graph.

Chapter 8

Discussion

In this chapter we discuss the evaluation of the algorithms that this work proposes. Section 8.1 discusses if it is possible to improve the reordering techniques and if they are optimal. Section 8.2 discusses the run time of the conflict detection algorithms. Section 8.3 discusses the impact of the fallback mechanism on the results. Section 8.4 discusses how the methods are expected to perform for different benchmarks and workload parameters. Finally, the future works, in section 8.6, proposes three ways to extend this work: 1) A reordering method that has a linear run time (section 8.6.2), 2) supporting optimizations for commutative operations (section 8.6.1) and 3) applying reordering in other batch commit schedulers (section 8.6.3).

8.1 Building dependency graphs

Recall from chapter 4.1 that whenever two transactions write to the same key this forms a write write dependency between the two. The write write dependencies are initially undirected, but then chosen to point from transactions with larger transaction id to transactions with smaller transaction ids. Although the proposed reordering methods are very close to the optimal solutions calculated by the ILP's, it is good to keep in mind that the ILP's find the optimal solution for a given start order serialization graph. If we choose different directions for the write write, this results in different start order serialization graphs. It is possible that different start order serialization graphs result in better solutions. However on workloads such as YCSB-B, all the writes depend on reads. Whenever two transactions write to the same key they also read from the same key. Resulting in two directly conflicting transactions. Therefore the direction of the writes is not expected to be of a large influence on the solution. However, for other workloads when the writes are independent of the reads, the direction of the write write edges could be of influence.

8.2 Run time conflict detection algorithms

Aria does not need the most optimal commit scheduler, but a fast and effective heuristic. The methods that reorder transactions on degree need the start order serialization graph. To build the graph, the read write sets of each transaction need to be compared with the read write sets of all the other transactions. To detect the dependencies between the transactions in the graph. Therefore, building a graph runs in $O(n^2)$. With n the number of transactions. Arguably reordering on degree might not scale well when more workers are added to the system. The workers are configured to process a fixed amount of transactions per epoch. The number of transactions in

one epoch in total in the system is

$$\text{number of transactions} = \text{transactions per worker} \cdot \text{number of workers}$$

In high content workloads, transactions access multiple partitions this results in large connected graphs. The reordering on degree algorithm suffers performance when scaling out horizontally. Although figure 6.9 of section 6.1 shows that the reordering methods outperform the other methods up to 8 workers. It is expected that when more workers are added to the system performance degrades. Therefore section 8.6.2 proposes a reordering mechanism that has a linear run time.

8.3 Fallback mechanism

In some of the experiments the fallback mechanism is turned off. This is done to see the effect of the methods uninfluenced by the fallback mechanism. When the fallback mechanism is turned off, the effects of a better conflict detection algorithm become clearly visible. Less aborts results in less transactions that need to be processed in the next epochs. The case could be made that Aria configured with the fallback mechanism turned off is broken for high contention workloads. Suppose that N transactions are all conflicting with each other because they update the same key. Only one transactions can commit per epoch and it would take N epochs to commit these transactions. With the fallback mechanism turned off the effect of aborts propagate in the next epochs. The fallback mechanism is an important part of Aria's performance. With the fallback mechanism turned on, it is harder to reason about the systems behavior. It is complicated to reason about the interactions of Arias optimistic concurrency control algorithm and the fallback mechanism. Reordering transactions on degree means that transactions with a high degree are forwarded to the fallback mechanism. This means that the fallback mechanism needs to process transactions that have a high degree in the start order serialization graph.

8.4 Evaluation

To get a good understanding of how a distributed database performs it is important to evaluate the system on different workloads and for different workload parameters. Parameters that could influence the performance of a distributed system are: the number of partitions a transaction accesses, the skew factor, the number of workers in the system and the length of the transactions. Often when databases set out to improve for specific workload parameters it happens that they loose performance on other workloads. However, figure 6.8 of section 6.4 shows that for uniform workloads the methods that use reordering on degree perform similar as the other methods. This means that even for uniform workloads the proposed methods do not waste to much resources on calculating optimal orderings.

To get a better understanding of the reordering methods the methods need to be tested against the other parameters. For this, more offline workloads could be created that vary the transactions read write percentage, transactions length and skew factor. This could also give more insight in when lowering the isolation guaranty to snapshot isolation could reduce latency. Informally, snapshot isolation only checks for write conflicts on keys instead of read after write conflict. Therefore it is expected that increasing the write read ratio this decreases snapshot isolation performance. However, when transactions access more keys, say with a read write ratio of 80%

reads and 20% writes, this is expected to improve snapshot isolations performances, compared with serializability.

Testing the effect of the number of partitions a transaction accesses can only be evaluated in an online system setting. In our test setup on YCSB-B10, all the workers make 10 asynchronous calls to different keys. The number of workers varies from 2 til a total of 8. Therefore in our test setup the transactions almost always access all the workers. More experiments need to be done to see how the reordering algorithms compare to the other methods when the transactions access less workers, for instance only 2. However, the less partitions a transaction accesses the easier a workload is to paralelize. Usually performance increases when transactions access less data.

The proposed methods used in this work are evaluated offline on the YCSB-B1 and YCSB-B5 workload. The proposed are also evaluated in online system setting on YCSB-B10. These are widely accepted benchmarks and often used for comparing performance of distributed databases. However to give a better overview of the performance of system other benchmarks such as TPC-C would also need to be evaluated.

8.5 Broadcast or coordinated

Performance is similar for broadcast or coordinator worker configuration. However, when the system is configured with a reordering on degree algorithms, the workers do the same work in broadcasting setup. Therefore, total resources used in coordinator worker setup during conflict detection is the number of workers times larger than in coordinator worker setup. For instance, TPC-C also measures price/performance *TPC Benchmark C* 2010. Distributed systems often stay way from coordinator worker configurations to avoid single point of failure. However, when in broadcast configuration every worker of Aria is a single point of failure. Every worker needs to respond in order to continue to the next epoch. For a coordinator configuration does not mean that another cloud instance needs to be deployed. The workers can also use a consensus algorithm leader election protocol appoint one of the workers to do the job of the coordinator.

In the current testing setup the program is evaluated on one machine. The system is designed to run in the cloud. When running on multiple machines network latency would increase. This could impact the communication strategy. To validate the value of the system in the cloud the system would need to be tested in a cloud environment.

8.6 Future work

In this chapter we present some ideas of how to improve Aria. section 8.6.1 explains how to extend Aria to support optimizations for commutative operations. section 8.6.2 proposes a different reorder method that has a linear run time. Moreover, since Aria is a deterministic databases some possibilities for reordering are limited. section 8.6.3 addresses this by suggesting to implement reordering techniques in other databases that use batch commit.

8.6.1 Supporting optimizations for commutative operations

Transactions protocol often consider transactions as a black box. They do not make assumptions about the data types a transaction accesses, or the operations that are executed. Some examples of data types in databases are: strings, date times, arrays and numbers. Operations could be string concatenation, array append or compare and set. Making assumptions about the type of operations of transactions gives room for certain optimizations. For instance Doppel Narula et al., 2014 assumes operations of transactions are commutative. An example of commutative operator is addition. Most transactions involving money transfers are commutative. An epoch based commit scheduler such as Aria would be a great candidate to support commutative transactions. Transactions could be chopped into pieces. Because the operations are commutative the order of the pieces is invariant. It is possible to distribute the pieces over the workers and execute them consecutively. The workers do not need to read from the snapshot but could expose writes to each other at any time. Resulting in more concurrency within one epoch. However some care needs to be taken when handling transactions that have abort logic. For instance transactions that have non-negative constraints. Exposing there writes early could result in cascading aborts.

8.6.2 Reorder on hot keys

Reordering transactions on degree depends on building a start order serialization graph. This has a run time of $O(n^2)$. As argued in the discussion this does not scale well when multiple workers are added to the system. To reorder the sequence every transaction needs a weight. It is possible to consider other cost functions that assign a weight to each transaction. Below we propose a different cost function that could be considered for future research and runs in $O(n)$. With n the number of transactions. The function works as follows. First, count per key how many transactions write to that key. Call this the weight of a key. Keys that are accessed often, so called hotkeys, will have a larger weight. For this, one iteration over all transactions is needed. Within the loop the function needs to iterate through the write set. This is bound by $O(n \cdot W)$, where W is the size of the write set. Then iterate again over the transactions and calculates the weight of a transaction to be the sum of the weights of the keys in the transactions write set. In this way transactions that access a lot of hotkeys will end up having a larger weight. The cost of the second iteration is again $O(n \cdot W)$.

8.6.3 Other epoch based commit schedulers

One of the main problems in Aria is that only one transaction can update the value of a key per epoch. To allow for more writes within an epoch, a transaction needs

to expose its writes to other transactions. However, doing this, determinism will be lost. Coco Lu et al., 2021 is a non-deterministic epoch based commit scheduler. It uses two phase commit as concurrency control algorithm. Coco applies two phase commit in epochs to reduce the overhead added by 2PC. This system could benefit from reordering the transactions. In Coco transactions are allowed to read other writes within an epoch. This allows read after write dependencies. This creates chains with transactions that depend on other transactions to commit. Other optimizations are possible that commits the largest chain and reduces cascading aborts as much as possible.

Bibliography

Adya, Atul (1999). "Weak consistency: a generalized theory and optimistic implementations for distributed transactions". PhD thesis. Massachusetts Institute of Technology, Dept. of Electrical Engineering and ...

Adya, Atul, Barbara Liskov, and Patrick O'Neil (2000). "Generalized isolation level definitions". In: *Proceedings of 16th International Conference on Data Engineering (Cat. No. 00CB37073)*. IEEE, pp. 67–78.

Agrawal, Rakesh, Michael J Carey, and Miron Livny (1987). "Concurrency control performance modeling: Alternatives and implications". In: *ACM Transactions on Database Systems (TODS)* 12.4, pp. 609–654.

Astrahan, Morton M. et al. (1976). "System R: Relational approach to database management". In: *ACM Transactions on Database Systems (TODS)* 1.2, pp. 97–137.

Bailis, Peter et al. (2016). "Scalable atomic visibility with RAMP transactions". In: *ACM Transactions on Database Systems (TODS)* 41.3, pp. 1–45.

Beillahi, Sidi Mohamed, Ahmed Bouajjani, and Constantin Enea (2019). "Checking robustness against snapshot isolation". In: *Computer Aided Verification: 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part II 31*. Springer, pp. 286–304.

Berenson, Hal et al. (2007). "A critique of ANSI SQL isolation levels". In: *arXiv preprint cs/0701157*.

Cerone, Andrea, Giovanni Bernardi, and Alexey Gotsman (2015). "A framework for transactional consistency models with atomic visibility". In: *26th International Conference on Concurrency Theory (CONCUR 2015)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

Cheng, Chaoyi et al. (2023). "Developer's Responsibility or Database's Responsibility? Rethinking Concurrency Control in Databases". In: *13th Annual Conference on Innovative Data Systems Research (CIDR'23). January 8-11, 2023, Amsterdam, The Netherlands.*

Codd, Edgar F (1970). "A relational model of data for large shared data banks". In: *Communications of the ACM* 13.6, pp. 377–387.

Cooper, Brian F et al. (2010). "Benchmarking cloud serving systems with YCSB". In: *Proceedings of the 1st ACM symposium on Cloud computing*, pp. 143–154.

Crooks, Natacha et al. (2017). "Seeing is believing: A client-centric specification of database isolation". In: *Proceedings of the ACM Symposium on Principles of Distributed Computing*, pp. 73–82.

Curino, Carlo et al. (2010). "Schism: a workload-driven approach to database replication and partitioning". In.

Cutello, Vincenzo and Francesco Pappalardo (2015). "Targeting the minimum vertex set problem with an enhanced genetic algorithm improved with local search strategies". In: *Intelligent Computing Theories and Methodologies: 11th International Conference, ICIC 2015, Fuzhou, China, August 20-23, 2015, Proceedings, Part I 11*. Springer, pp. 177–188.

DeCandia, Giuseppe et al. (2007). *Dynamo: Amazon's Highly Available Key-Value Store. SOSP, 2007*.

Ding, Bailu, Lucja Kot, and Johannes Gehrke (2018). "Improving optimistic concurrency control through transaction batching and operation reordering". In: *Proceedings of the VLDB Endowment* 12.2, pp. 169–182.

Faleiro, Jose M and Daniel J Abadi (2014). "Rethinking serializable multiversion concurrency control". In: *arXiv preprint arXiv:1412.2324*.

Faleiro, Jose M, Daniel J Abadi, and Joseph M Hellerstein (2017). "High performance transactions via early write visibility". In: *Proceedings of the VLDB Endowment* 10.5.

Fekete, Alan et al. (2005). "Making snapshot isolation serializable". In: *ACM Transactions on Database Systems (TODS)* 30.2, pp. 492–528.

Friedman, Roy and Robbert Van Renesse (1997). "Packing messages as a tool for boosting the performance of total ordering protocols". In: *Proceedings. The Sixth IEEE International Symposium on High Performance Distributed Computing (Cat. No. 97TB100183)*. IEEE, pp. 233–242.

Gan, Yifan et al. (2020). "IsoDiff: debugging anomalies caused by weak isolation". In: *Proceedings of the VLDB Endowment* 13.12.

Gray, Jim and Andreas Reuter (1992). *Transaction processing: concepts and techniques*. Elsevier, pp. 5–6.

Gray, Jim et al. (1994). "Quickly generating billion-record synthetic databases". In: *Proceedings of the 1994 ACM SIGMOD international conference on Management of data*, pp. 243–252.

Gray, Jim N, Raymond A Lorie, and Gianfranco R Putzolu (1975). "Granularity of locks in a shared data base". In: *Proceedings of the 1st International Conference on very large data bases*, pp. 428–451.

Johnson, Donald B (1975). "Finding all the elementary circuits of a directed graph". In: *SIAM Journal on Computing* 4.1, pp. 77–84.

Kaki, Gowtham et al. (2017). "Alone together: compositional reasoning and inference for weak isolation". In: *Proceedings of the ACM on Programming Languages* 2.POPL, pp. 1–34.

Karp, Richard M (2010). *Reducibility among combinatorial problems*. Springer.

Kung, Hsiang-Tsung and John T Robinson (1981). "On optimistic methods for concurrency control". In: *ACM Transactions on Database Systems (TODS)* 6.2, pp. 213–226.

Lakshman, Avinash and Prashant Malik (2009). "Cassandra: structured storage system on a p2p network". In: *Proceedings of the 28th ACM symposium on Principles of distributed computing*, pp. 5–5.

Lamport, Leslie (2001). "Paxos made simple". In: *ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001)*, pp. 51–58.

Lu, Yi et al. (2020). "Aria: a fast and practical deterministic OLTP database". In.

— (2021). "Epoch-based commit and replication in distributed OLTP databases". In.

Microsoft (n.d.). *SQL Server*. https://www.microsoft.com/en-us/sql-server.

Mohan, C, Bruce Lindsay, and Ron Obermarck (1986). "Transaction management in the R* distributed database management system". In: *ACM Transactions on Database Systems (TODS)* 11.4, pp. 378–396.

Mu, Shuai et al. (2014). "Extracting more concurrency from distributed transactions". In: *11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14)*, pp. 479–494.

Narula, Neha et al. (2014). "Phase reconciliation for contended in-memory transactions". In: *11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14)*, pp. 511–524.

Nuutila, Esko and Eljas Soisalon-Soininen (1994). "On finding the strongly connected components in a directed graph". In: *Information processing letters* 49.1, pp. 9–14.

Oracle (n.d.). *MySQL Cluster*. https://www.mysql.com/products/cluster/.

Papadimitriou, Christos H (1979). "The serializability of concurrent database updates". In: *Journal of the ACM (JACM)* 26.4, pp. 631–653.

Postgres (n.d.). *PostgreSQL*. https://www.postgresql.org/.

Rao, Jun, Eugene J Shekita, and Sandeep Tata (2011). "Using paxos to build a scalable, consistent, and highly available datastore". In: *arXiv preprint arXiv:1103.2408*.

Shasha, Dennis et al. (1995). "Transaction chopping: Algorithms and performance studies". In: *ACM Transactions on Database Systems (TODS)* 20.3, pp. 325–363.

Shute, Jeff et al. (2013). "F1: A distributed SQL database that scales". In.

Sovran, Yair et al. (2011). "Transactional storage for geo-replicated systems". In: *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pp. 385–400.

Tang, Chuzhe et al. (2022). "Ad hoc transactions in web applications: the good, the bad, and the ugly". In: *Proceedings of the 2022 International Conference on Management of Data*, pp. 4–18.

Tarjan, Robert (1972). "Depth-first search and linear graph algorithms". In: *SIAM journal on computing* 1.2, pp. 146–160.

Thomasian, Alexander (1998). "Distributed optimistic concurrency control methods for high-performance transaction processing". In: *IEEE Transactions on Knowledge and Data Engineering* 10.1, pp. 173–189.

Thomson, Alexander and Daniel J Abadi (2010). "The case for determinism in database systems". In: *Proceedings of the VLDB Endowment* 3.1-2, pp. 70–80.

Thomson, Alexander et al. (2012). "Calvin: fast distributed transactions for partitioned database systems". In: *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pp. 1–12.

TPC Benchmark C (2010). Last accessed 30 January 2023. URL: tpc.org/tpcc/.

Wang, Jia-Chen et al. (2021). "Polyjuice: High-Performance Transactions via Learned Concurrency Control." In: *OSDI*, pp. 198–216.

Wang, Tianzheng and Hideaki Kimura (2016). "Mostly-optimistic concurrency control for highly contended dynamic workloads on a thousand cores". In: *Proceedings of the VLDB Endowment* 10.2, pp. 49–60.

Yu, Xiangyao et al. (2016). "Tictoc: Time traveling optimistic concurrency control". In: *Proceedings of the 2016 International Conference on Management of Data*, pp. 1629–1642.

Zhang, Jian et al. (2023). "Viper: A Fast Snapshot Isolation Checker". In.