CSE 535: Distributed Systems
Project 4
Due: December 15, 2024 (11:59 pm)

Abstract

The objective of this project is to implement a Byzantine fault-tolerant distributed transaction processing system (a.k.a a permissioned blockchain system) that supports a simple banking application. To this end, similar to the third project, we partition servers into multiple clusters where each cluster maintains a data shard. Each data shard is replicated on all servers of a cluster to provide fault tolerance. The system supports two types of transactions: intra-shard and cross-shard. To process intra-shard transactions the linear PBFT protocol implemented in the second project should be used while the two-phase commit protocol is used to process cross-shard transactions.

1 Project Description

We discuss the project in three steps. We first, explain the architecture of the system and the supported application followed by a brief overview of the linear PBFT protocol (from the second project). Finally, the two-phase commit protocol is discussed.

1.1 Architecture and Application

In this project, similar to all previous projects, you are supposed to deploy a simple banking application where clients send their transfer transactions in the form of (x, y, amt) to the servers where x is the sender, y is the receiver, and amt is the amount of money to transfer. This time, we shift our focus towards real-world applications deployed in untrustworthy environments by examining a large-scale key-value store that is partitioned across multiple Byzantine clusters, with each cluster managing a distinct shard of the application's data. The system architecture is illustrated in Figure 1.

As shown, the data is divided into three shards: D_1, D_2, and D_3. The system comprises a total of 12 servers, labeled S_1 through S_{12}, organized into three clusters: C_1, C_2, and C_3. Each data shard D_i is replicated across all servers within its respective cluster C_i to ensure fault tolerance, operating under the assumption that servers adhere to a Byzantine failure model, where at most one server in each cluster may be faulty at any given time.

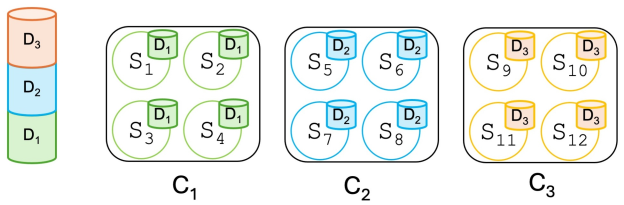

Figure 1: System Architecture

Each client sends its transactions to the primary server of the cluster that holds its record. We assume that clients have access to the shard mapping, which informs them about the

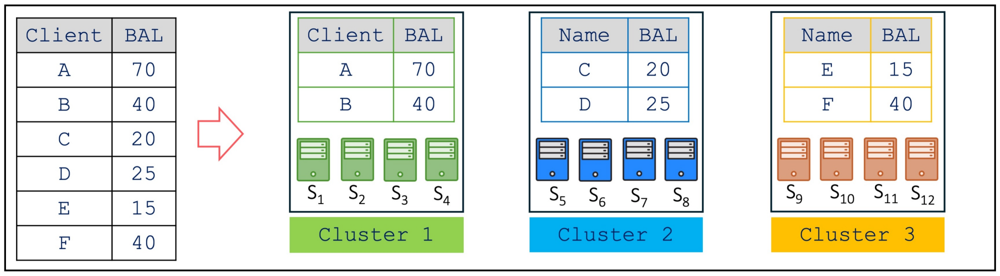

Figure 2: System Architecture

data items stored in each cluster of servers. Clients can initiate both intra-shard and cross-shard transactions. A single client process can send all client requests to the servers and you should not implement clients as closed-loop. The reason behind this design choice is that in the specific application of this project, all transactions that transfer money from account x to any other account are received from the same client x. Hence, if clients are closed-loop, there is no need to lock any records because no two concurrent transactions will reduce the balance of the same client (i.e., double-spending cannot happen even without implementing a locking mechanism). Note that servers still need to maintain the timestamp of the last request received from each client in order to prevent replay attacks.

For an intra-shard transaction (x, y, amt), client x sends its request to the primary server of the cluster that holds data item x (and also y). The servers in that cluster then execute linear PBFT protocol to reach consensus on the order of the request. Similalry, in the case of a cross-shard transaction (x, y, amt), client x sends its request to the primary server of the cluster that maintains data item x. In this scenario, in contrast to the third project, the cluster receiving the client request acts as the coordinator for the two-phase commit protocol (not the client itself).

Figure 2 shows an example of the system where a dataset consisting of 6 data items is partitioned into 3 different data shards. Each data shard is then maintained by (replicated on) a cluster of four servers.

1.2 Intra-Shard Transactions: Linear PBFT Protocol

To process intra-shard transactions, we utilize the linear PBFT protocol from the second project without any modification to the protocol logic other than adding the locking mechanism. In the linear PBFT protocol, a client sends an intra-shard request (x, y, amt) to the primary server of the cluster that holds both data items x and y. Once the primary receives the request, it first checks two conditions: (1) there are no locks on data items x and y, and (2) the balance of x is at least equal to amt. If both conditions are satisfied, the leader acquires locks on both data items x and y and initiates the consensus process by sending a pre-prepare message to all servers in the cluster. Otherwise, the leader simply ignores the transaction.

It's important to note that such behavior may lead to the omission of valid transactions, especially if the leader is slow and has not yet executed a prior transaction that could increase the balance of data item x or release the lock on x or y. This situation could ultimately result in the replacement of a slow leader through view-change (e.g., when the client's timer times out, it resubmits the transaction to all servers, as in PBFT). As we discussed in Project 3, this kind of indeterministic behavior is permissible within our straightforward design. A more sophisticated approach would involve the leader tracking pending transactions to assess the potential for future execution, such as if there are transactions that could increase the balance of x. However, you are not required to implement this in your project.

Upon receiving a valid pre-prepare message, each backup server must also secure locks on both data items x and y. To achieve this, the backup server must have received all transactions with smaller sequence numbers, enabling it to verify the validity of the transaction. For example, consider a scenario where a backup server receives a pre-prepare message for sequence number n that includes a transaction accessing data items x and y. The backup server needs to ensure that both data items x and y are not locked or in use by any previous transactions that have yet to be executed. This necessitates that the backup server is aware of all preceding transactions—specifically, those with sequence numbers less than n.

Consequently, the backup server first verifies whether it has received pre-prepare messages for all smaller sequence numbers. Once this check is complete, it assesses the transaction's validity before sending a prepare message. Similar to a slow leader, a slow backup may ignore a valid transaction simply because it hasn't executed a prior transaction, resulting in active locks. One straightforward solution for the backup is to wait a brief period to see if the lock will be released.

Another potential solution, applicable to both the leader and backups, is to acquire locks only during the execution phase. While this approach could reduce the number of skipped transactions, it may lead to significant delays, especially when cross-shard transactions are involved, causing an intra-shard transaction to wait a considerable amount of time before execution. This represents a trade-off between ignoring transactions and experiencing high latency. Although the first solution is recommended, you are not required to implement either of these solutions.

The remaining phases of the protocol will proceed as usual, with nodes sending reply messages back to the client once the request has been executed. After executing the transactions, the nodes will release the locks on the data items.

1.3 Cross-Shard Transactions: Two-Phase Commit Protocol

To process cross-shard transactions, we need to implement the two-phase commit (2PC) protocol across the clusters involved in each transaction. Since our focus is on transfer transactions, similar to the third project, each cross-shard transaction involves two distinct shards. In this configuration, one of the clusters (the cluster that maintains the data item of the sender) acts as the coordinator for the 2PC protocol.

The client initiates a cross-shard request (x, y, amt) by contacting the primary node from the cluster that holds data item x (the coordinator cluster).

Prepare phase in the coordinator cluster. The coordinator cluster then initiates linear PBFT protocol to achieve consensus on the order of the transaction. Similar to intra-shard transactions, the primary node and all backups need to make sure that first, there are no locks on data items x and second, the balance of x is at least equal to amt. If another transaction has already secured a lock on item x or if there is no sufficient balance, similar to intra-shard transactions, the leader of the coordinator server simply ignores the transaction. To prevent concurrent updates to the data items, all servers acquire locks on data item x (the primary server before sending the pre-prepare message and all other servers after receiving a valid pre-prepare message). Before obtaining locks, nodes need to ensure that they have received all transactions with smaller sequence numbers.

If nodes were able to achieve consensus on committing the transaction, the primary node sends a prepare message including the client request and $2f + 1$ matching commit messages to *every* node within the cluster that maintains record y (i.e., participant cluster). Moreover, a record will be added to the datastore of each node in the coordinator cluster upon reaching consensus. Servers within the coordinator cluster also execute the transaction and update their write-ahead logs (WAL), allowing them to undo changes in the case of an abort. However, they do not send reply messages back to clients in this phase (as the transaction has not been committed yet).

Prepare phase in the participant cluster. Upon receiving a valid prepare message, the primary server of the participant cluster performs a lock checking and then initiates consensus on the transaction. If consensus is achieved, the primary node sends a prepared message including $2f + 1$ matching commit messages to every node within the coordinator cluster (the cluster that maintains record x). Otherwise, an abort message with $2f + 1$ matching commit messages (that prove the abort decision) will be sent to every node within the coordinator cluster.

There is an important difference between the coordinator cluster and the participant cluster in the prepare phase of two-phase commit that should be noted: while the leader of the coordinator cluster simply ignores a transaction with insufficient balance or unavailable locks (i.e., there is no abort decision), the leader of the participant phase needs to run consensus even on an abort decision.

Upon reaching consensus in the participant cluster a record will be added to the datastore of each node. If the decision is commit, servers within the participant cluster execute the transaction, and update their write-ahead logs (WAL), allowing them to undo changes in the case of an abort. Similar to the coordinator cluster, nodes should not send reply messages back to clients.

Commit phase in the coordinator cluster. Upon receiving prepared messages from the participant cluster with either commit or abort decision, the leader of the coordinator cluster initiates consensus within its cluster. Upon achieving consensus, the primary node sends the outcome (either commit or abort) to every node within the participant cluster. If the outcome is a commit, each server within the coordinator cluster appends an entry to its datastore, releases its locks, removes entries from the WAL, and sends a PBFT reply message back to the client. If the outcome is an abort or if a timeout occurs, each server appends an entry to its datastore and use the WAL to undo the executed operations before releasing the locks. The server then sends a PBFT reply message with an "abort" as the result of executing the requested transaction.

Commit phase in the participant cluster. Upon receiving the commit message from the leader of the coordinator cluster, the leader of the participant cluster runs consensus within its cluster on the outcome. If the outcome is commit, each server appends an entry to its datastore, releases its lock, removes entries from the WAL and sends an Ack message back to the leader of the coordinating cluster. Note that the servers in the participant cluster do

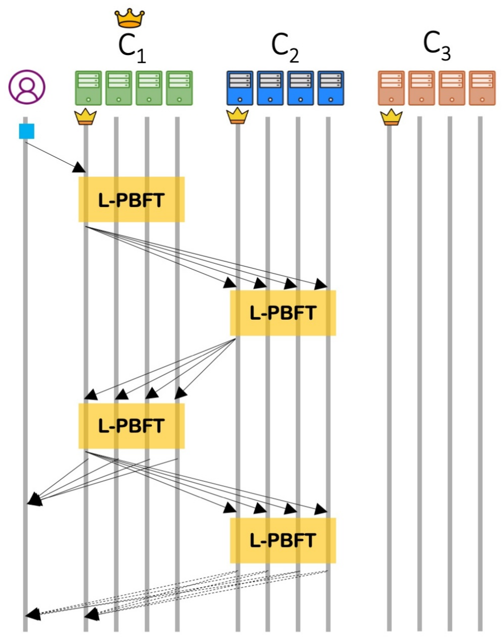

Figure 3: Cross-shard Transactions

not need to send a reply message back to the client.

If the participant cluster has committed the request in its prepare phase and the final outcome is also a commit, the leader of the coordinator cluster keeps forwarding the commit message to the unresponsive servers in the participant cluster (using a timer) until it receives $f+1$ Ack messages from different servers of the participant cluster.

As an optimization, if the participant cluster has already aborted the request in its prepare phase, it does not need to run consensus on the abort decision again (as the transaction has already been aborted in the participant cluster).

Figure 3 demonstrates the flow of a cross-shard transaction between clusters C_1 and C_2. As mentioned before, when a cross-shard transaction is performed, each server (within one of the involved shards) needs to append two entries to its datastore: one after the prepare phase and one after the commit phase.

Figure 4 shows how a sequence of transactions updates different data structures of a single server. We consider a simple database with only 4 clients. During the processing of intra-shard transaction (A, B, 20), locks on records A and B are acquired and released by every node. As shown, the transaction is executed on the database resulting in updating the balance of A and B. The second transaction is a cross-shard transaction between clients A and E. Here nodes acquire the lock on record A, run PBFT, append an entry to the datastore showing that the prepare phase was successful, execute the transaction (update the balance of A) and add records to the WAL. Before receiving the commit message for the second transaction, another intra-shard transaction (C, D, 5) takes place and updates the database and the lock table. Finally, the server receives a commit message from the client for the second transaction. The server appends an entry to the datastore, releases the lock on record A, and deletes the corresponding records from the WAL.

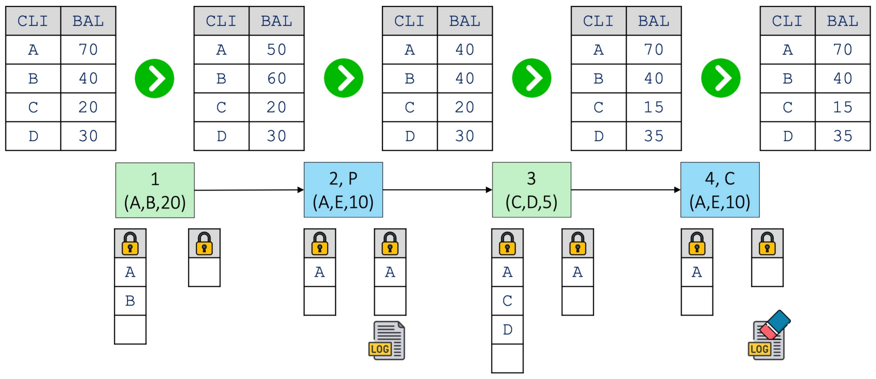

Figure 4: Processing Transactions

2 Implementation Details

Your implementation, similar to Project 3, should support a dataset of 3000 data items with $id = 1$ to $id = 3000$. As shown in Figure 1, there are 12 servers organized into three clusters of size 4. The data needs to be maintained in a key-value database (use some lightweight DBMS, using files is not accepted). Your program should first read a given input file. This file will consist of a set of triplets (x,y,amt). Assume all data items (i.e., client records) start with 10 units. A client process sends requests to the primary server of the shard that maintains the sender data item (x). Relying on a single client process to initiate all requests would be okay. Each request is a transfer (x,y,amt) which is either intra-shard (if x and y belong to the same shard) or cross-shard (if x and y belong to different shards).

NOTE: For this assignment, a fixed shard mapping has been defined, and it is **mandatory** to use this provided shard mapping in your implementation. This ensures consistency in testing your solution and establishes a common understanding of intra-shard and cross-shard transactions for everyone.

Cluster	Data Items
C1	[1,2,3,...,1000]
C2	[1001,1002,...,2000]
C3	[2001,2002,...,3000]

Table 1: Shard Mapping

IMPORTANT: A transaction should be aborted not only in cases of insufficient balance or lock unavailability but also when the system fails to reach consensus, such as in the absence of a majority agreement. Additionally, any other scenario where consensus cannot be achieved will also result in the transaction being aborted.

- Your program should be able to process a given input file containing a set of transactions, with each set representing an individual *test case* (see section 6.4 for more details).

- Your program should have a PrintBalance function which reads the balance of a given client (data item) from the database and prints the balance on all servers. This function accepts a client ID (or data item ID) as input and retrieves the balance for the specified client ID across all servers within the corresponding cluster.

- Your program should have a PrintDatastore function that prints the set of committed transactions on each server (i.e., datastore in your first project). Note that this data structure maintains all committed transactions and is different from the key-value store that maintains the balance of all clients. Refer to the test case guide for the required log format in the datastore.

- Your program should have a Performance function which prints throughput and latency. The throughput and latency should be measured from the time the client process initiates a transaction to the time the client process receives a reply message.

- While you may use as many log statements during debugging please ensure such extra messages are not logged in your final submission.

- We do not want any front-end UI for this project. Your project will be run on the terminal.

3 Bonus!

We briefly discuss two possible optimizations that you can implement and earn extra credit.

1. **Cross-shard transactions with more than two participant shards.** In both Projects 3 and 4, we covered cross-shard transactions with only two involved shards. However, this is not necessarily the case for all distributed systems. Your task is to implement cross-shard transactions that access all three shards. To do so, we need to change the input workload. For this part, client x can send a request (x, y, z, amt1, amt2) to transfer amt1 to client y and amt2 to client z. Note that this is still an atomic transaction and can be committed if both transfers happen.

2. **Small Bank Benchmark.** Testing the performance and functionality of distributed systems is challenging. To be able to evaluate distributed systems and compare their performance against each other, we need to use standard benchmarks. Such benchmarks are specifications and program suites for evaluating systems. For distributed databases, different benchmarks have been proposed such as Yahoo! Cloud Serving Benchmark (YCSB) or TPC-C (short for Transaction Processing Performance Council Benchmark C). Another benchmark that is close to what we have implemented is SmallBanck which simulates a banking application. This workload models a banking application where transactions perform simple read and update operations on their checking and savings accounts. All of the transactions involve a small number of tuples. The transactions' access patterns are skewed such that a small number of accounts receive most of the requests. It contains three tables and six types of transactions. The user table contains users' personal information, the savings table contains the balances,

and the checking table contains the checking balances. If you are interested in seeing the performance of your system in a real deployment, you can implement the SmallBank benchmark and evaluate your system under that. The details of the SmallBank benchmark can be found in [1].

4 Submission Instructions

4.1 Lab Repository Setup Instructions

Our lab assignments are distributed and submitted through GitHub. If you don't already have a GitHub account, please create one before proceeding. To get started with the lab assignments, please follow these steps:

1. **Join the Lab Assignment Repository**: Click on the provided link to join the lab assignment system.

   Important: If you are not able to accept the assignment please contact one of us immediately for assistance.

To set up the lab environment on your laptop, follow the steps below. If you are new to Git, we recommend reviewing the introductory resources linked here to familiarize yourself with the version control system.

Once you accept the assignment, you will receive a link to your personal repository. Clone your repository by executing the following commands in your terminal:

```
$ git clone git@github.com:F24-CSE535/2pcbyz-<YourGithubUsername>.git
$ cd 2pcbyz-<YourGithubUsername>
```

This will create a directory named `2pcbyz-<YourGithubUsername>` under your home directory, which will serve as the Git repository for this lab assignment. These steps are crucial for properly setting up your lab repository and ensuring smooth submission of your assignments.

4.2 Lab Submission Guidelines

For lab submissions, please push your work to your private repository on GitHub. You may push as many times as needed before the deadline. We will retrieve your lab submissions directly from GitHub for grading after the deadline.

To make your final submission for Lab 4, please include an explicit commit with the message **submit lab4** from your main branch. Afterward, visit the provided link to verify your submission. Follow all instructions carefully, as no waivers will be granted for submission errors.

5 Deadline, Demo, and Deployment

This project will be due on December 15. You need to prepare a recording and a report for the project (the details will be announced later).

6 Tips and Policies

6.1 General Tips

- Start early!
- Make sure you understand the PBFT protocol, the two-phase locking and the two-phase commit lecture notes before you start.
- Make sure your PBFT implementation (From project 2) works correctly.

6.2 Implementation

- You need to continue with the programming language used in your first three projects.
- Your implementation should demonstrate reasonable performance in terms of throughput and latency, measured by the number of transactions committed per second and the average processing time for each transaction. We will assess the performance of all submissions against each other and projects with unreasonably poor performance may face point deductions.

6.3 Possible Test Cases

Below is a list of some of the possible scenarios (test cases) that your system should be able to handle.

- All test cases of project 2
- Concurrent independent intra-shard transactions in different clusters
- Concurrent independent cross-shard transactions
- Concurrent intra-shard and cross-shard transactions
- Concurrent (intra-shard or cross-shard) transactions accessing the same data item(s)
- All possible failures and timeout scenarios discussed in the two-phase commit protocol
- No consensus if too many servers fail
- No commitment if any cluster aborts

6.4 Example Test Format

The testing process involves a csv file (.csv) as the test input containing a set of transactions along with the corresponding live servers involved in each set. A set represents an individual test case and your implementation should be able to process each set sequentially i.e. in the given order.

Your implementation should prompt the user before processing the next set of transactions. That is, the next set of transactions should only be picked up for processing when the user requests it. Until then, depending on your implementation, all servers should remain idle until prompted to process the next set. You are not allowed to terminate your servers after executing a set of transactions, as consecutive test cases may be interdependent.

After executing one set of transactions, when all the servers are idle and waiting to process the next set, your implementation should allow the use of functions from section 2.1.1 (such as *PrintBalance*, *printDatastore*, etc.). The implementation will be evaluated based on the output of these functions after each set of transactions is processed.

The test input file will contain four columns:

1. **Set Number**: Set number corresponding to a set of transactions.
2. **Transactions**: A list of individual transactions, each on a separate row, in the format `(Sender, Receiver, Amount)`.
3. **Live Servers**: A list of servers that are active and available for all transactions in the corresponding set (including Byzantine servers).
4. **Byzantine Servers**: A list of servers that exhibit Byzantine behavior for all transactions in the corresponding set.

An example of the test input file is shown below:

Set Number	Transactions	Live Servers	Byzantine Servers
1	(21, 700, 2)	[S1, S2, S4, S5, S6, S8, S9, S10, S11, S12]	[S9]
	(100, 501, 8)		
	(1001, 1650, 2)		
	(2800, 2150, 7)		
	(1003, 1001, 5)		
2	(702, 1301, 2)	[S1, S2, S3, S5, S6, S8, S9]	[S3, S6, S8]
	(1301, 1302, 3)		
	(600, 1502, 6)		

Table 2: Example Test Input File

This example test scenario demonstrates the basic structure and approach that will be used to assess your implementation.

6.5 Grading Policies

- Your projects will be graded based on multiple parameters:
  1. The code successfully compiles and the system runs as intended.
  2. The system passes all tests.
  3. The system demonstrates reasonable performance.

4. You are able to explain the flow and different components of the code and what is the purpose of each class, function, etc.
5. The implementation is efficient and all functions have been implemented correctly.
6. The number of implemented and correctly operating additional bonus optimizations (extra credit).

- **Late Submission Penalty**: For every day that you submit your project late, there will be a 10% deduction from the total grade, up to a maximum of 30% deduction within the first 3 days of the original deadline. The final deadline for submitting Project 4 and still receiving 70% (assuming the project works perfectly) is December 18.

6.6 Academic Integrity Policies

We are serious about enforcing the integrity policy. Specifically:

- The work that you turn in must be yours. The code that you turn in must be code that you wrote and debugged. Do not discuss code, in any form, with your classmates or others outside the class (for example, discussing code on a whiteboard is not okay). As a corollary, it's not okay to show others your code, look at anyone else's, or help others debug. It is okay to discuss code with the instructor and TAs.

- You must acknowledge your influences. This means, first, writing down the names of people with whom you discussed the assignment, and what you discussed with them. If student A gets an idea from student B, both students are obligated to write down that fact and also what the idea was. Second, you are obligated to acknowledge other contributions (for example, ideas from Websites, AI assistant tools, Chat GPT, existing GitHub repositories, or other sources). The only exception is that material presented in class or the textbook does not require citation.

- You must not seek assistance from the Internet. For example, do not post questions from our lab assignments on the Web. Ask the course staff, via email or Piazza, if you have questions about this.

- You must take reasonable steps to protect your work. You must not publish your solutions (for example on GitHub or Stack Overflow). You are obligated to protect your files and printouts from access.

- Your project submissions will be compared to each other and existing implementations on the internet using plagiarism detection tools. If any substantial similarity is found, a penalty will be imposed.

- We will enforce the policy strictly. Penalties include failing the course (and you won't be permitted to take the same class in the future), referral to the university's disciplinary body, and possible expulsion. Do not expect a light punishment such as voiding your assignment; violating the policy will definitely lead to failing the course.

- If there are inexplicable discrepancies between exam and lab performance, we will overweight the exam, and possibly interview you. Our exams will cover the labs. If, in light of your exam performance, your lab performance is implausible, we may discount or even discard your lab grade (if this happens, we will notify you). We may also conduct an interview or oral exam.
- You are welcome to use existing public libraries in your programming assignments (such as public classes for queues, trees, etc.) You may also look at code for public domain software such as GitHub. Consistent with the policies and normal academic practice, you are obligated to cite any source that gave you code or an idea.
- The above guidelines are necessarily generalizations and cannot account for all circumstances. Intellectual dishonesty can end your career, and it is your responsibility to stay on the right side of the line. If you are not sure about something, ask.

References

[1] Michael J Cahill, Uwe Röhm, and Alan D Fekete. Serializable isolation for snapshot databases. *Transactions on Database Systems (TODS)*, 34(4):1–42, 2009.