Approaches of Achieving Serializability in Distributed Database

Peter Yu (zy264)

I. Introduction

Distributed database systems (DDS) are becoming increasingly popular due to the scalability, availability, and fault tolerance they offer. However, ensuring the correctness of transactions in a DDS is a challenging problem, and one important, if not the most important, correctness criterion for a DDS is serializability, which ensures that the execution of concurrent transactions produces the same results as if the transactions were executed serially. In this literature review, we will discuss the different approaches and techniques people have adopted to achieve serializability in distributed database systems.

II. Fundamental Theory of Serializability

The concept of serializability became a common correctness criterion for distributed database in the 1970s, and techniques such as two-phase locking appeared before a comprehensive theory was developed. Two-phase locking (2PL) was one of the earliest techniques used to achieve serializability in distributed systems. In 2PL, transactions acquire locks on database objects before accessing them and release them after they are done. This ensures that transactions do not interfere with each other and that their execution is serialized. Although 2PL was widely used, it had some limitations, such as high lock contention, which can lead to poor performance in highly concurrent systems. Hence, researchers sought to describe the problem of serializability more accurately. The first work that formalized the complexity of the problem and categorize the different classes of serializability appeared in 1979 by Papadimitriou [1]. One fundamental result in the paper was, testing whether a history h is serializable is NP-complete, even if h has no dead transactions. This motivates us to divide the set of all serializable transactions (SR) into finer subsets, including DSR (interchangeable SR), Q (SR with serializability points) and SSR (strict SR, no reversed transaction pairs in serialized history). Moreover, through analysis of dependency graphs, the paper shows that a history h is in 2PL if and only if h^*, a transformed version of h, is in Q. Similarly, serializability criteria such as the P1 and P3 protocol are also subsets of SR. The general result is included in Fig. 1.

While serialization is usually discussed on the level of transactions, researchers explore finer granularities of serialization to improve efficiency. For example, read and write of the same row in a database could have no causal dependence and no interference, hence no synchronization is needed to prevent inconsistency. In this regard, Bernstein *et al.*[2] proposed

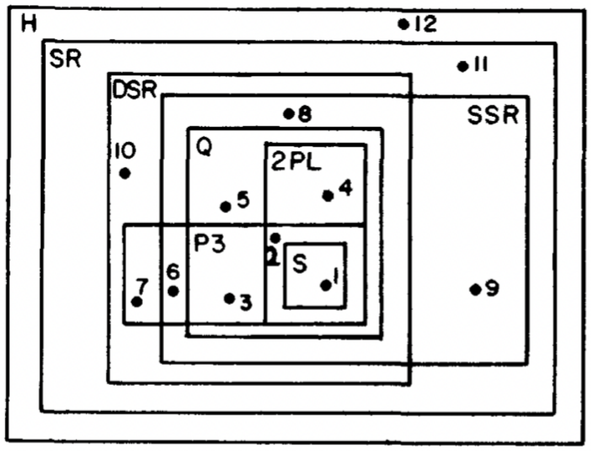

Fig. 1. Categories of serializable history

protocols including P1, P3 as different levels of consistency. P1 ensures the order of read messages from one transaction that conflict with write messages from another transaction, and P3 prevents two transactions from reading each other's output from both reading before either writes (classical race condition). If we have enough insight into the different kinds of transactions, we can choose a protocol that best fits the user scenario.

III. Graph Representation of Serialization

Researchers have always relied on graphs to represent the serialization property of execution histories. In general, a dependency graph is serializable if it is acyclic. The early papers [1][2] adopt different types of dependency graphs, and in more recent papers, we observe that the directed serialization graph (DSG) proposed by Adya in 1999 [3] has become a relatively common representation. In a DSG, each node represents a transaction, and edges between nodes indicate a dependency relationship between transactions. A dependency between two transactions exists if one transaction performs a write operation on a data item that another transaction reads or writes later. The edge direction is from the earlier transaction to the later transaction.

Despite the differences, these graph theory approaches share two major difficulties: (1) it is much harder to prove the nonexistence of a cycle in a graph than to identify a cycle in a graph, and (2) constructing a graph can be inefficient. Therefore, applications that use dependency graphs often make reasonable assumptions or simplifications based on the use case. For example, in a traditional relational database, we may assume every write is preceded by a read of the variable [1]. If there is a clearly defined set of transactions, we can divide transactions into classes, so that under some class pipelining rules, there is no interference within each class [2].

Graphs are an important tool for both theoretical derivation discussed in Section II and preanalysis of transactions in real world applications, as we shall see in the next section.

IV. Improved Distributed Database Systems

In 1980, Bernstein proposed the SSD-1 protocol [2]. It builds upon the Snapshot Isolation (SI) protocol, which allows transactions to read from a consistent snapshot of the database without blocking other transactions. The SSD-1 protocol extends SI by introducing a mechanism for detecting conflicts between transactions and aborting conflicting transactions to ensure serializability. SSD-1 chooses between P1, P2, P3 protocols based on requirement and efficiency.

Bernstein and Reed also proposed MVCC (Multi-Version Concurrency Control). In MVCC, each transaction reads a consistent snapshot of the database that is represented by a set of versions of each data item. Each version has a timestamp that indicates the time at which it was created or updated that are used for reading and writing. Moreover, when the transaction is ready to commit, there is a validation phase that checks whether the versions that a transaction has read are still valid when it tries to commit. If any of the versions have been updated by other transactions, the validation fails, and the transaction must be aborted and restarted with a new snapshot. This validation phase is what ensures serializability.

In the years following, many works have been done to improve upon these ideas. Neumann *et al.* presents a novel MVCC implementation [8]. While traditional serializable MVCC models detect conflicts at the granularity of records, the new implementation logs the comparison summaries for restricted attributes. It is a scalable lock-free system where readers never block writers.

Meanwhile, the Lynx protocol is designed to work in geo-distributed environments with high latency and low bandwidth links between data centers [4]. The idea behind Lynx is to use transaction chains to reduce the communication overhead and latency of maintaining global orderings of transactions. Transaction chains are sequences of transactions that are ordered according to their dependencies. During static analysis, a transaction chain transforms a transaction into a SC-graph, in which serializability is implied if the SC-graph has no SC-cycle. During execution, a chain is executed deterministically in its respective data center.

Somewhat similar to Lynx, SLOG (Serializable, Low-latency, Geo-replicated Transactions) [5] also focuses on the workloads which contain physical region locality in data access. In contrast to Lynx, SLOG takes into consideration the lack of information about all transactions running in the system during the planning process, and avoids static analysis of transaction. It is more similar to MVCC, but has three phases: the read phase, the validation phase, and the commit phase. Moreover, transactions that involve multiple data centers are all handled by the same region, then the set order is distributed to all local logs. This approach requires each data center to keep a local log and a global log, and log distribution requires careful locking. However, in comparison to Google Spanner, SLOG does not use Paxos or 2 phase commit, hence it significantly outperforms Spanner in write-heavy use cases.

V. Formal Methods and Automated Reasoning

There has also been a noticeable increase in the use of formal methods and automated reasoning, These approaches have the potential to provide strong guarantees about the correctness of distributed systems, while also reducing the burden on developers to manually verify the correctness of their programs.

Lucas Brutschy *et al.* [6] proposes a new technique for verifying the serializability of transactions in systems that provide causal consistency. The technique uses a form of static analysis to analyze the system's communication graph and identify transactions that may violate serializability. Noticeably, it uses DSG as an important intermediate tool, but continues to translate the problem to first-order logic to input into an SMT solver.

Similarly, Kartik Nagar *et al.* [7] presents an automated approach for detecting serializability violations in weakly consistent distributed systems. The approach combines the DSG-based characterization of serializability and the framework of abstract execution, and aims to generate a set of constraints that must be satisfied in order for the system to be serializable.

Finally, Rahmani *et al.* [9] aims to solve a slightly different problem using formal methods. Instead of performing static analysis, this approach seeks to automatically repair serializability bugs in execution histories. The model performs schema refactoring, which involves modifying the database schema in order to add constraints and eliminate serializability violations. While this approach is quite novel, we note that this method creates schemas that are convoluted with internal logical relations, which is sometimes against the principle of database systems.

VI. Conclusion

Serializability is a fundamental correctness property of distributed database systems. Over the years, researchers have proposed several techniques and protocols to achieve serializability, such as two-phase locking, timestamp ordering, and multi-version concurrency control. Theoretical works have established an understanding of the complexity of the problem and the tradeoffs of the techniques and protocols above.

In recent years, several new protocols such as Fast MVCC, Lynx, and SLOG have been proposed that aim to achieve better performance while maintaining serializability guarantees. There is also a growing trend towards using formal methods and automated reasoning. These approaches provide stronger guarantees of the correctness of distributed systems.

Overall, serializability remains an active area of research, and we expect to see continued progress in this field as distributed systems become more complex and widely used.

References

[1] C. H. Papadimitriou, "The serializability of concurrent database updates," Journal of the ACM, vol. 26, no. 4, pp. 631–653, 1979.

[2] P. A. Bernstein, D. W. Shipman, and J. B. Rothnie, "Concurrency control in a system for distributed databases (SDD-1)," ACM Transactions on Database Systems, vol. 5, no. 1, pp. 18–51, 1980.

[3] Atul Adya. 1999. Weak Consistency: A Generalized Theory and Optimistic Implementations for Distributed Transactions. PhD Thesis. Massachusetts Institute of Technology, Dept. of Electrical Engineering and Computer Science.

[4] Y. Zhang, R. Power, S. Zhou, Y. Sovran, M. K. Aguilera, and J. Li, "Transaction chains: achieving serializability with low latency in geo-distributed storage systems," Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles, 2013.

[5] K. Ren, D. Li, and D. J. Abadi, "SLOG: Serializable, Low-latency, Geo-replicated Transactions," Proceedings of the VLDB Endowment, vol. 12, no. 11, pp. 1747–1761, 2019.

[6] L. Brutschy, D. Dimitrov, P. Müller, and M. Vechev, "Static serializability analysis for causal consistency," Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, 2018.

[7] K. Nagar and S. Jagannathan, "Automated detection of serializability violations under weak consistency," arXiv.org, 21-Jun-2018. [Online]. Available: https://arxiv.org/abs/1806.08416. [Accessed: 07-May-2023].

[8] T. Neumann, T. Mühlbauer, and A. Kemper, "Fast serializable multi-version concurrency control for main-memory database systems," Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, 2015.

[9] K. Rahmani, K. Nagar, B. Delaware, and S. Jagannathan, "Repairing serializability bugs in distributed database programs via Automated Schema Refactoring," Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, 2021.

[10] H. Garcia-Molina, "Using semantic knowledge for transaction processing in a distributed database," ACM Transactions on Database Systems, vol. 8, no. 2, pp. 186–213, 1983.

[11] N. Crooks, Y. Pu, L. Alvisi, and A. Clement, "Seeing is believing," Proceedings of the ACM Symposium on Principles of Distributed Computing, 2017.