How to make Serializable (almost) as fast as Read Committed

Jose M. Faleiro
Yale University

Database systems are one of the most important pieces of the modern application technology stack. In theory, database systems simplify the management of state and concurrency by guaranteeing the ACID properties. These properties facilitate the development of modular applications: applications interact with the database via transactions, and as long as each individual transaction is correct, the database system ensures that arbitrary compositions of transactions are correct, even in the presence of concurrency and failures. By abstracting away concurrency and failures via ACID, database systems ostensibly permit application developers to focus solely on what matters most, application logic.

Unfortunately, this picture is not as rosy in practice. In order to guarantee correctness in the presence of concurrency, database systems must *isolate* concurrent transactions from each other. To this end, database systems provide a variety of *isolation levels*, varying from strong to weak, which allow applications to make tradeoffs between correctness (a.k.a consistency) and performance. Strong isolation levels, such as *serializable*, permit fewer interleavings among conflicting transactions, which guarantees strong consistency at the expense of performance. In contrast, weak isolation levels, such as *read committed*, permit more interleavings among conflicting transactions, allowing transactions to observe inconsistent database states in order to improve performance. Only the strongest isolation level, serializable, guarantees that arbitrary compositions of concurrent transactions preserve correctness. But since this convenience comes at the cost of performance, applications often eschew serializable isolation in favor of weaker isolation levels. Indeed, database systems actively *encourage* the use of weak isolation levels; SQL Server and PostgreSQL provide read committed isolation by default, while Oracle 12c and SAP HANA do not even provide serializable isolation. Needless to say, this widespread adoption of weak isolation has been a pain point for application developers and system administrators, who at best, have to reason about complex transaction interleavings, and at worst, recover from silent data corruption, leading to a loss of revenue, sleep, and sanity.

Each isolation level is a *specification* of legal sequences of conflicting database operations that are permitted to execute concurrently. These specifications fundamentally limit the performance of strong isolation as compared to weak isolation levels because every implementation is constrained by the corresponding specification. Counter-intuitively, however, the differences in performance between various isolation levels in modern database systems, is *not* due to their specifications, but instead a facet of their implementations.

This talk will discuss research which explains the differences between strong and weak isolation levels in modern database systems. I will show that the differences arise due to the *recoverability* mechanisms used in modern systems. While there has been no dearth of research on mechanisms for isolation (a.k.a concurrency control), database systems employ mechanisms for recoverability that are decades old. Indeed, the most popular recoverability mechanism employed by modern main-memory database systems, *group commit*, was invented by Dieter Gawlick and David Kinkade in IMS/VS FastPath in 1975.

Modern recoverability mechanisms only permit a transaction's writes from being read when it commits or at least finishes executing. Thus, there is typically a delay between the time a write is performed, and the time the write can be read. This delay adversely impacts strong isolation levels (such as serializable). This is because (as an approximation) serializable requires that transactions always read the latest value of a record; any delay in satisfying a read will delay the corresponding reading transaction.

Recoverability mechanisms delay making transactions' writes visible because database systems can arbitrarily abort a transaction prior to the point that its commit record is made durable; a database system may abort a transaction due to deadlock handling logic, failures, optimistic validation errors, or simply because the transaction consumes resources that are in short supply. Database systems' ability to arbitrarily abort transactions forces recoverability mechanisms to make extremely pessimistic assumptions about when a transaction's writes are safe from being rolled back. This talk will make the case for curtailing database systems' ability to arbitrarily abort transactions. I will present the design of a research prototype which only aborts transactions under a restricted set of conditions, and thus avoids overly conservative recoverability mechanisms. This prototype can perform comparably to and often outperform an optimized state-of-the-art implementation of read committed isolation under a variety of transaction processing workloads.