

Chapter 6: Weak Isolation and Distribution

Introduced by Peter Bailis

Selected Readings:

Atul Adya, Barbara Liskov, and Patrick O’Neil. Generalized Isolation Level Definitions. *ICDE*, 2000.

Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon’s Highly Available Key-Value Store. *SOSP*, 2007.

Eric Brewer. CAP Twelve Years Later: How the ”Rules” Have Changed. *IEEE Computer*, 45, 2 (2012).

Conventional database wisdom dictates that serializable transactions are the canonical solution to the problem of concurrent programming, but this is seldom the case in real-world databases. In practice, database systems instead overwhelmingly implement concurrency control that is non-serializable, exposing users to the possibility that their transactions will not appear to have executed in some serial order. In this chapter, we discuss why the use of this so-called “weak isolation” is so widespread, what these non-serializable isolation modes actually do, and why reasoning about them is so difficult.

Overview and Prevalence

Even in the earliest days of database systems (see Chapter 3), systems builders realized that implementing serializability is expensive. The requirement that transactions appear to execute sequentially has profound consequences for the degree of concurrency a database can achieve. If transactions access disjoint sets of data items in the database, serializability is effectively “free”: under these disjoint access patterns, a serializable schedule admits data parallelism. However, if transactions contend on the same data items, in the worst case, the system cannot process them with any parallelism whatsoever. This property is fundamental to serializability and independent of the actual implementation: because transactions cannot safely make progress independently under all workloads (i.e., they must *coordinate*), any implementation of serializability may, in effect, require serial execution. In practice, this means that transactions may need to wait, decreasing throughput while increasing latency. Transaction processing expert Phil Bernstein suggests that serializability typically incurs a three-fold performance penalty on a single-node database compared to one of the most common weak isolation levels called Read Committed [13]. Depending on the implementation, serializability may also lead to more aborts, restarted transactions, and/or deadlocks. In distributed databases, these costs increase because networked communication is expensive, increasing the

time required to execute serial critical sections (e.g., holding locks); we have observed multiple order-of-magnitude performance penalties under adverse conditions [7].

As a result, instead of implementing serializability, database system designers instead often implemented weaker models. Under weak isolation, transactions are not guaranteed to observe serializable behavior. Instead, transactions will observe a range of anomalies (or “phenomena”): behaviors that could not have arisen in a serial execution. The exact anomalies depend on which model is provided, but example anomalies include reading intermediate data that another transaction produced, reading aborted data, reading two or more different values for the same item during execution of the same transaction, and “losing” some effects of transactions due to concurrent writes to the same item.

These weak isolation modes are surprisingly prevalent. In a recent survey of eighteen SQL and “NewSQL” databases [5], we found that only three of eighteen provided serializability by default and eight (including Oracle and SAP’s flagship offerings) did not offer serializability at all! This state of affairs is further complicated by often inaccurate use of terminology: for example, Oracle’s “serializable” isolation guarantee actually provides Snapshot Isolation, a weak isolation mode [15]. There is also a race to the bottom among vendors. Anecdotally, when vendor A, a major player in the transaction processing market, switched its default isolation mode from serializability to Read Committed, vendor B, who still defaulted to serializability, began to lose sales contracts during bake-offs with vendor A. Vendor B’s database was clearly slower, so why would the customer choose B instead of A? Unsurprisingly, vendor B now provides Read Committed isolation by default, too.

The Key Challenge: Reasoning about Anomalies

The primary reason why weak isolation is problematic is that, depending on the application, weak isolation anomalies can result in application-level inconsistency: the invari-

ants that each transaction preserves in a serializable execution may no longer hold under weak isolation. For example, if two users attempt to withdraw from a bank account at the same time and their transactions run under a weak isolation mode allowing concurrent writes to the same data item (e.g., the common Read Committed model), the users may successfully withdraw more money than the account contained (i.e., each reads the current amount, each calculates the amount following their withdrawal, then each writes the “new” total to the database). This is not a hypothetical scenario. In a recent, colorful example, an attacker systematically exploited weak isolation behavior in the Flexcoin Bitcoin exchange; by repeatedly and programmatically triggering non-transactional read-modify-write behavior in the Flexcoin application (an vulnerability under Read Committed isolation and, under a more sophisticated access pattern, Snapshot Isolation), the attacker was able to withdraw more Bitcoins than she should have, thus bankrupting the exchange [1].

Perhaps surprisingly, few developers I talk with regarding their use of transactions are even aware that they are running under non-serializable isolation. In fact, in our research, we have found that many open-source ORM-backed applications assume serializable isolation, leading to a range of possible application integrity violations when deployed on commodity database engines [6]. The developers who are aware of weak isolation tend to employ a range of alternative techniques at the application level, including explicitly acquiring locks (e.g., SQL “SELECT FOR UPDATE”) and introducing false conflicts (e.g., writing to a dummy key under Snapshot Isolation). This is highly error-prone and negates many of the benefits of the transaction concept.

Unfortunately, specifications for weak isolation are often incomplete, ambiguous, and even inaccurate. These specifications have a long history dating to the 1970s. While they have improved over time, they remain problematic.

The earliest weak isolation modes were specified operationally: as we saw in Chapter 3, popular models like Read Committed were originally invented by modifying the duration for which read locks were held [17]. The definition of Read Committed was: “hold read locks for a short duration, and hold write locks for a long duration.”

The ANSI SQL Standard later attempted to provide an implementation-independent description of several weak isolation modes that would apply not only to lock-based mechanisms but also to multi-versioned and optimistic methods as well. However, as Gray et al. describe in [11], the SQL Standard is both ambiguous and under-specified: there are multiple possible interpretations of the English-language description, and the formalization does not capture all behaviors of the lock-based implementations. Additionally,

the ANSI SQL Standard does not cover all isolation modes: for example, vendors had already begun shipping production databases providing Snapshot Isolation (and labeling it as serializable!) before Gray et al. defined it in their 1995 paper. (Sadly, as of 2015, the ANSI SQL Standard remains unchanged.)

To complicate matters further, Gray et al.’s 1995 revised formalism is also problematic: it focuses on lock-related semantics and rules out behavior that might be considered safe in a multi-versioned concurrency control system. Accordingly, for his 1999 Ph.D. thesis [2], Atul Adya introduced the best formalism for weak isolation that we have to date. Adya’s thesis adapts the formalism of multi-version serialization graphs [12] to the domain of weak isolation and describes anomalies according to restrictions on those graphs. We include Adya’s corresponding ICDE 2000 paper, but isolation aficionados should consult the full thesis. Unfortunately, Adya’s model is still underspecified in some cases (e.g., what exactly does G0 mean if no reads are involved?), and implementations of these guarantees differ across databases.

Even with a perfect specification, weak isolation is still a real challenge to reason about. To decide whether weak isolation is “safe,” programmers must mentally translate their application-level consistency concerns down to low-level read and write behavior [3]. This is ridiculously difficult, even for seasoned concurrency control experts. In fact, one might wonder what benefits of transactions remain if serializability is compromised? Why is it easier to reason about Read Committed isolation than no isolation at all? Given how many database engines like Oracle run under weak isolation, how does modern society function at all – whether users are booking airline flights, administering hospitals, or performing stock trades? The literature lends few clues, casting serious questions about the success of the transaction concept as deployed in practice today.

The most compelling argument I have encountered for why weak isolation seems to be “okay” in practice is that few applications today experience high degrees of concurrency. Without concurrency, most implementations of weak isolation deliver serializable results. This in turn has led to a fruitful set of research results. Even in a distributed setting, weakly isolated databases deliver “consistent” results: for example, at Facebook, only 0.0004% of results returned from their eventually consistent store were “stale” [19], and others have found similar results [10, 25]. However, while for many applications weak isolation is apparently not problematic, it can be: as our Flexcoin example illustrates, given the possibility of errors, application writers must be vigilant in accounting for (or otherwise explicitly ignoring) concurrency-related anomalies.

Weak Isolation, Distribution, and “NoSQL”

With the rise of Internet-scale services and cloud computing, weak isolation has become even more prevalent. As I mentioned earlier, distribution exacerbates overheads of serializability, and, in the event of partial system failures (e.g., servers crashing), transactions may stall indefinitely. As more and more programmers began to write distributed applications and used distributed databases, these concerns became mainstream.

The past decade saw the introduction of a range of new data stores optimized for the distributed environment, collectively called “NoSQL.” The “NoSQL” label is unfortunately overloaded and refers to many aspects of these stores, from lack of literal SQL support to simpler data models (e.g., key-value pairs) and little to no transactional support. Today, as in MapReduce-like systems (Chapter 5), NoSQL stores are adding many these features. However, a notable, fundamental difference is that these NoSQL stores frequently focus on providing better availability of operations via weaker models, with an explicit focus on fault tolerance. (It is somewhat ironic that, while NoSQL stores are commonly associated with the use of non-serializable guarantees, classic RDBMSs do not provide serializability by default either.)

As an example of these NoSQL stores, we include a paper on the Dynamo system, from Amazon, presented at SOSP 2007. Dynamo was introduced to provide highly available and low latency operation for Amazon’s shopping cart. The paper is technically interesting as it combines several techniques, including quorum replication, Merkle tree anti-entropy, consistent hashing, and version vectors. The system is entirely non-transactional, does not provide any kind of atomic operation (e.g., compare and swap), and relies on the application writer to reconcile divergent updates. In the limit, any node can update any item (under hinted hand-off).

By using a merge function, Dynamo adopts an “optimistic replication” policy: accept writes first, reconcile divergent versions later [21, 16]. On the one hand, presenting a set of divergent versions to the user is more friendly than simply discarding some concurrent updates, as in Read Committed isolation. On the other hand, programmers must reason about merge functions. This raises many questions: what is a suitable merge for an application? How do we avoid throwing away committed data? What if an operation should not have been performed concurrently in the first place? Some open source Dynamo clones, like Apache Cassandra, do not provide merge operators and simply choose a “winning” write based on a numerical timestamp. Others, like Basho Riak, have adopted “libraries” of automatically mergeable datatypes like counters, called Commutative Replicated Data Types [22].

Dynamo also does not make promises about recency of reads. Instead, it guarantees that, if writes stop, eventually all replicas of a data item will contain the same set of writes. This eventual consistency is a remarkably weak guarantee: technically, an eventually consistent datastore can return stale (or even garbage) data for an indefinite amount of time [9]. In practice, data store deployments frequently return recent data [25, 10], but, nevertheless, users must reason about non-serializable behavior. Moreover, in practice, many stores offer intermediate forms of isolation called “session guarantees” that ensure that users read their own writes (but not the writes of other users); interestingly, these techniques were developed in the early 1990s as part of the Bayou project on mobile computing and have recently come to prominence again [24, 23].

Trade-offs and the CAP Theorem

We have also included Brewer’s 12 year retrospective on the CAP Theorem. Originally formulated following Brewer’s time building Inktomi, one of the first scalable search engines, Brewer’s CAP Theorem pithily describes trade-offs between the requirement for coordination (or “availability”) and strong guarantees like serializability. While earlier results described this trade-off [18, 14], CAP became a rallying cry for mid-2000s developers and has considerable impact. Brewer’s article briefly discusses performance implications of CAP, as well as the possibility of maintaining some consistency criteria without relying on coordination.

Programmability and Practice

As we have seen, weak isolation is a real challenge: its performance and availability benefits mean it is extremely popular in deployments despite the fact that we have little understanding of its behavior. Even with a perfect specification, existing formulations of weak isolation would still be a extremely difficult to reason about. To decide whether weak isolation is “safe,” programmers must mentally translate their application-level consistency concerns down to low-level read and write behavior [3]. This is ridiculously difficult, even for seasoned concurrency control experts.

As a result, I believe there is a serious opportunity to investigate semantics that are not subject to the performance and availability overheads of serializability but are more intuitive, usable, and programmable than existing guarantees. Weak isolation has historically been highly challenging to reason about, but this need not be the case. We and others have found that several high-value use cases, including index and view maintenance, constraint maintenance, and distributed aggregation, frequently do not actually require coordination for “correct” behavior; thus, for these use cases,

serializability is overkill [4, 8, 20, 22]. That is, by providing databases with additional knowledge about their applications, database users can have their cake and eat it too. Further identifying and exploiting these use cases is an area ripe for research.

Conclusions

In summary, weak isolation is prevalent due to its many benefits: less coordination, higher performance, and greater availability. However, its semantics, risks, and usage are poorly understood, even in an academic context. This is particularly baffling given the amount of research devoted to serializable transaction processing, which is considered by many to be a “solved problem.” Weak isolation is arguably even more deserving of such a thorough treatment. As I have

highlighted, many challenges remain: how do modern systems even work, and how should users program weak isolation? For now, I offer the following take-aways:

- Non-serializable isolation is prevalent in practice (in both classical RDBMSs and recent NoSQL upstarts) due to its concurrency-related benefits.
- Despite this prevalence, many existing formulations of non-serializable isolation are poorly specified and difficult to use.
- Research into new forms of weak isolation show how to preserve meaningful semantics and improve programmability without the expense of serializability.

References

- [1] Flexcoin: The Bitcoin Bank, 2014. <http://www.flexcoin.com/>; originally via Emin Gün Sirer.
- [2] A. Adya. *Weak consistency: a generalized theory and optimistic implementations for distributed transactions*. PhD thesis, MIT, 1999.
- [3] P. Alvaro, P. Bailis, N. Conway, and J. M. Hellerstein. Consistency without borders. In *SoCC*, 2013.
- [4] P. Bailis. *Coordination avoidance in distributed databases*. PhD thesis, University of California at Berkeley, 2015.
- [5] P. Bailis, A. Davidson, A. Fekete, A. Ghodsi, J. M. Hellerstein, and I. Stoica. Highly Available Transactions: Virtues and limitations. In *VLDB*, 2014.
- [6] P. Bailis, A. Fekete, M. J. Franklin, A. Ghodsi, J. M. Hellerstein, and I. Stoica. Feral Concurrency Control: An empirical investigation of modern application integrity. In *SIGMOD*, 2015.
- [7] P. Bailis, A. Fekete, M. J. Franklin, J. M. Hellerstein, A. Ghodsi, and I. Stoica. Coordination avoidance in database systems. In *VLDB*, 2015.
- [8] P. Bailis, A. Fekete, A. Ghodsi, J. M. Hellerstein, and I. Stoica. Scalable atomic visibility with RAMP transactions. In *SIGMOD*, 2014.
- [9] P. Bailis and A. Ghodsi. Eventual consistency today: Limitations, extensions, and beyond. *ACM Queue*, 11(3), 2013.
- [10] P. Bailis, S. Venkataraman, M. J. Franklin, J. M. Hellerstein, and I. Stoica. Probabilistically Bounded Staleness for practical partial quorums. In *VLDB*, 2012.
- [11] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O’Neil, and P. O’Neil. A critique of ANSI SQL isolation levels. In *SIGMOD*, 1995.
- [12] P. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency control and recovery in database systems*, volume 370. Addison-Wesley New York, 1987.
- [13] P. A. Bernstein and S. Das. Rethinking eventual consistency. In *SIGMOD*, 2013.
- [14] S. Davidson, H. Garcia-Molina, and D. Skeen. Consistency in partitioned networks. *ACM CSUR*, 17(3):341–370, 1985.
- [15] A. Fekete, D. Liarakapis, E. O’Neil, P. O’Neil, and D. Shasha. Making snapshot isolation serializable. *ACM TODS*, 30(2):492–528, June 2005.

- [16] J. Gray, P. Helland, P. O'Neil, and D. Shasha. The dangers of replication and a solution. In *SIGMOD*, 1996.
- [17] J. Gray, R. Lorie, G. Putzolu, and I. Traiger. Granularity of locks and degrees of consistency in a shared data base. Technical report, IBM, 1976.
- [18] P. R. Johnson and R. H. Thomas. Rfc 667: The maintenance of duplicate databases. Technical report, 1 1975.
- [19] H. Lu, K. Veeraraghavan, P. Ajoux, J. Hunt, Y. J. Song, W. Tobagus, S. Kumar, and W. Lloyd. Existential consistency: measuring and understanding consistency at Facebook. In *SOSP*, 2015.
- [20] S. Roy, L. Kot, G. Bender, B. Ding, H. Hojjat, C. Koch, N. Foster, and J. Gehrke. The homeostasis protocol: Avoiding transaction coordination through program analysis. In *SIGMOD*, 2015.
- [21] Y. Saito and M. Shapiro. Optimistic replication. *ACM Comput. Surv.*, 37(1), Mar. 2005.
- [22] M. Shapiro, N. Preguica, C. Baquero, and M. Zawirski. A comprehensive study of convergent and commutative replicated data types. INRIA TR 7506, 2011.
- [23] D. Terry. Replicated data consistency explained through baseball. *Communications of the ACM*, 56(12):82–89, 2013.
- [24] D. B. Terry, A. J. Demers, K. Petersen, M. J. Spreitzer, M. M. Theimer, et al. Session guarantees for weakly consistent replicated data. In *PDIS*, 1994.
- [25] H. Wada, A. Fekete, L. Zhao, K. Lee, and A. Liu. Data consistency properties and the trade-offs in commercial cloud storage: the consumers' perspective. In *CIDR*, 2011.