The Case for Invariant-Based Concurrency Control (Abstract)

Peter Bailis
UC Berkeley

By the metrics of wide adoption and industrial deployment, serializable transactions have failed. Despite the convenience and power of serializability, in today's RDBMSs, weak isolation guarantees like Read Committed isolation are overwhelmingly the default option and are sometimes the strongest (particularly among "NewSQL" stores) [3]. Even within the database community, many of us have acquiesced by settling for models like Snapshot Isolation that are apparently "good enough" despite exposing highly nuanced Write Skew anomalies (i.e., races) [1]. Despite the hype surrounding the return of serializability in systems like HStore [16], VoltDB, and Spanner [8], serializable transactions remain prohibitively expensive in a distributed (and, especially, geo-replicated) environment. (Read-only and single-partition transactions are an exception, but, as Stonebraker noted in 1985, these workloads have long been considered "delightful" from a concurrency control standpoint [15].) These costs are no surprise. The algorithmic forebears of these new systems [10, 14, 17] pre-date the era of large-scale Internet services, and, more fundamentally, the performance, availability, and latency limitations (i.e., the *coordination* overheads) are fundamental to these strong semantics rather than the faults of any given implementation [4]. Any claims of unilateral performance or availability parity between serializable systems and weaker alternatives should be examined with suspicion.

Unfortunately, in the words of one anonymous five-star wizard of data management, "once you give up serializability, you fall off a cliff." How do we program these weak isolation levels? Overwhelmingly, and, in stark contrast to the beautiful abstraction of serializability, the alternatives offered by weak isolation have been driven by studying *mechanisms* rather than application requirements. For example, in 1975, faced with the observation that strict two-phase locking can be expensive, Jim Gray et al. asked [11]: what happens if we hold read locks for shorter? A simple tweak to the locking mechanism became a new policy (Read Committed isolation) that has haunted database users for the last 38 years. When is Read Committed isolation safe for a given application? The literature lends few clues [6], and I challenge any self-respecting concurrency control researcher—or, better, the end-users we serve as a community—to provide a good answer. In retrospect, a better question might have been: what alternatives to serializability can we provide that make programming easier for developers? The former pattern of discourse dominates the exhausting literature of alternative isolation levels [3], some of which this author is complicit in inventing. I am hardly the first to pose the latter question, but, today, most of these alternatives lie deep in the bowels of 1980s-era concurrency control literature, largely forgotten in this new era of cloud-enabled, Big Data, web-scale data management systems.

7th Biennial Conference on Innovative Data Systems Research (CIDR '15) January 4-7, 2015, Asilomar, California, USA.

It is high time that concurrency control systems—as deployed in practice and not simply in our undergraduate textbooks—actually serve the requirements of end-user applications. High performance non-serializable concurrency control must integrate application-level criteria as a basis for correct execution. At SIGMOD 1979, Kung and Papadimitriou taught us the (seldom heeded) lesson that, without any knowledge of application semantics, we cannot do better than serializability while ensuring correctness [13]. Yet the demand for more scalable, less coordination-intensive and therefore non-serializable concurrency control necessitates a shift beyond the read-write interface and towards increased application semantics.

Invariants, or declarative constraints on acceptable database states, are a promising means of capturing correctness criteria. First, invariants allow users to reason about their applications instead of low-level read/write behavior. This eliminates the error-prone process of manually translating between low-level read/write traces (i.e., prohibited phenomena that define weak isolation levels) and application correctness criteria [2]. Second, from an implementation perspective, instead of specifying *how* an application's correctness should be guaranteed, an invariant leaves considerable leeway in terms of implementation and optimization. In recent work, we have demonstrated how invariants directly determine the potential for coordination-free execution, illustrated via a 25-fold improvement in compliant TPC-C New-Order throughput and order-of-magnitude improvements over serializable isolation due to decreased coordination between concurrent transactions [4]. Third, invariants have already crept into data management solutions in various forms, including primary key, foreign key, and check constraints. In an ongoing survey of open-source ORM-backed web applications, we have found widespread adoption of user-level invariant-based concurrency control mechanisms (e.g., Ruby on Rails Validations), which are largely undocumented in the database systems community yet, by usage, are over an order of magnitude more prevalent than transactions. The basic concept of (and arguments for) invariant-based concurrency control date to at least the early 1970s [9]. However, the recent rise (and re-discovery) of its similarly vintage cousins weak isolation [11], eventual consistency [12], and distributed transactions [7] heralds the possibility of a profitable rebirth.

Invariant-based concurrency control presents several opportunities for the CIDR community. My collaborators and I have already begun classifying common invariants as requiring coordination or not (and therefore achievable in a scalable system), yielding results as above [4]. However, there are a range of existing challenges: how should an invariant requiring coordination actually be maintained? What is the space of programs that pass the necessary invariant confluence condition [4], and how should we analyze full programs beyond SQL? Which practical invariants—beyond those found in RDBMSs today—are common cases ripe for optimization [5]? The answers to these questions, coupled with the further development of invariant-based concurrency control systems offers great promise. We can do better, and our users deserve more humane and more usable high performance database concurrency control abstractions.

References

[1] A. Adya. *Weak consistency: a generalized theory and optimistic implementations for distributed transactions*. PhD thesis, MIT, 1999.

[2] P. Alvaro, P. Bailis, N. Conway, and J. M. Hellerstein. Consistency without borders. In *SoCC*, 2013.

[3] P. Bailis, A. Davidson, A. Fekete, A. Ghodsi, J. M. Hellerstein, and I. Stoica. Highly available transactions: Virtues and limitations. In *VLDB*, 2014.

[4] P. Bailis, A. Fekete, M. J. Franklin, A. Ghodsi, J. M. Hellerstein, and I. Stoica. Coordination avoidance in database systems. In *VLDB*, 2015.

[5] P. Bailis, A. Fekete, A. Ghodsi, J. M. Hellerstein, and I. Stoica. Scalable atomic visibility with RAMP transactions. In *SIGMOD*, 2014.

[6] A. J. Bernstein, P. M. Lewis, and S. Lu. Semantic conditions for correctness at different isolation levels. In *ICDE*, 2000.

[7] W. Chu and G. Ohlmacher. Avoiding deadlock in distributed data bases. In *ACM Annual Conference*, 1974.

[8] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, et al. Spanner: Google's globally-distributed database. In *OSDI*, 2012.

[9] J. J. Florentin. Consistency auditing of databases. *The Computer Journal*, 17(1):52–58, 1974.

[10] H. Garcia-Molina and K. Salem. Main memory database systems: An overview. *TKDE*, 4(6):509–516, 1992.

[11] J. Gray, R. Lorie, G. Putzolu, and I. Traiger. Granularity of locks and degrees of consistency in a shared data base. Technical report, IBM, 1975.

[12] P. R. Johnson and R. H. Thomas. Network working group RFC 677: Maintenance of duplicate databases, 1975.

[13] H.-T. Kung and C. H. Papadimitriou. An optimality theory of concurrency control for databases. In *SIGMOD*, 1979.

[14] B. Liskov. Practical uses of synchronized clocks in distributed systems. *Distributed Computing*, 6(4):211–219, 1993.

[15] M. Stonebraker. The case for shared nothing. In *HPTS*, 1985.

[16] M. Stonebraker, S. Madden, D. J. Abadi, S. Harizopoulos, et al. The end of an architectural era: (it's time for a complete rewrite). In *VLDB*, 2007.

[17] A. Whitney, D. Shasha, and S. Apter. High volume transaction processing without concurrency control, two phase commit, SQL or C++. In *HPTS*, 1997.