FIT: A Distributed Database Performance Tradeoff

Jose M. Faleiro
Yale University
jose.faleiro@yale.edu

Daniel J. Abadi
Yale University
dna@cs.yale.edu

Abstract

Designing distributed database systems is an inherently complex process, involving many choices and tradeoffs that must be considered. It is impossible to build a distributed database that has all the features that users intuitively desire. In this article, we discuss the three-way relationship between three such desirable features — fairness, isolation, and throughput (FIT) — and argue that only two out of the three of them can be achieved simultaneously.

1 Introduction

A transaction that involves data which resides on more than one node in a distributed system is called a distributed transaction. At a minimum, distributed transactions must guarantee atomicity; the property that either all or none of a transaction's updates must be reflected in the database state. Atomicity prohibits a scenario where updates are selectively applied on only a subset of nodes. In order to guarantee atomicity, distributed transactions require some form of coordination among nodes; at the very least, each node must have a way of informing other nodes of its willingness to apply a transaction's updates.

As a consequence of the unavoidable coordination involved in distributed transactions, database lore dictates that there exists a tradeoff between *strong isolation* and *throughput*. The intuition for this tradeoff is that in a system which guarantees isolation among conflicting transactions, distributed coordination severely impacts the amount of concurrency achievable by the system by extending the time for which conflicting transactions are unable to make meaningful progress. The only way to get around this limitation is to provide weak isolation, which allows transactions to execute concurrently *despite the presence of conflicts*. The intuition is clear: a system which implements distributed transactions can provide either one of strong isolation or good throughput, but not both.

In this article, we put this intuition under the microscope, and find that there exists another variable, *fairness*, that influences the interplay between strong isolation and throughput. We make the case that weakening isolation is not the only way to "defeat" distributed coordination; if a system is given the license to selectively prioritize or delay transactions, it can find a way to pay the cost of distributed coordination without severely impacting throughput. Based on our intuition and an examination of existing distributed database systems, we propose that instead of an isolation-throughput tradeoff, there exists a *three-way* tradeoff between fairness, isolation, and throughput (FIT); a system that foregoes one of these properties can guarantee the other two. In this way,

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

designers who are comfortable reasoning about the three-way tradeoff behind the CAP theorem [10] can use the same type of reasoning to comprehend the FIT tradeoff for distributed database systems.

The remainder of this article proceeds as follows: Section 2 outlines our assumptions and definitions. Section 3 describes the general principles underlying the FIT tradeoff. Section 4 describes FIT in action using examples of real systems. Section 5 discusses how the tradeoff may be more broadly applicable beyond distributed transactions. Section 6 concludes the paper.

2 Assumptions and Definitions

We assume that data is *sharded* across a set of nodes in a distributed system. Each node is referred to as a partition. A distributed transaction refers to a type of transaction whose read- and write-sets involve records on more than one partition. A transaction must be guaranteed to have one of two outcomes; commit or abort. We classify aborts as being either logic-induced or system-induced. A logic-induced abort occurs when a transaction encounters an explicit abort statement in its logic. A system-induced abort occurs when the database forcibly aborts a transaction due to events beyond the transaction's control such as deadlocks and node failures.

When processing distributed transactions, we require that a database satisfy *safety*, *liveness*, and *atomicity*; defined as follows:

- **Safety.** The partitions involved in a distributed transaction must agree to either commit or abort a transaction. A transaction is allowed to commit if *all* partitions can commit, otherwise, it must abort.
- **Liveness.** If a distributed transaction is always re-submitted whenever it experiences a system-induced abort, then it is guaranteed to eventually commit. [1]
- **Atomicity.** If a transaction commits, then *all* of its updates must be reflected in database state. Otherwise, none of its updates must be reflected in the database's state.

2.1 Fairness

We do not attempt to formally define fairness. Instead, we provide intuition for what it means for a system to be fair and then discuss our intuition in the context of two examples: group commit [8] and lazy transaction evaluation [9].

Fairness corresponds to the idea that a database system does not deliberately prioritize nor delay certain transactions. When a transaction is received by the database, the system immediately attempts to process the transaction, and never artificially adds latency to a transaction (i.e. blocking the transaction for reasons other than waiting for data or for conflicting transactions to finish) for the purpose of facilitating the execution of other transactions.

Database systems have a long history of trading off fairness in order to obtain good performance. One of the prominent examples, "group commit", was proposed by DeWitt et al. in the context of database systems whose records could reside entirely in main-memory [8]. In such a setting, the time to execute a transaction's logic is dramatically shortened, but the cost of writing log records to disk – for durability purposes – limits overall throughput. Accordingly, DeWitt et al. proposed that the database accumulate log records from several transactions into a single disk write. Writing log records in batches circumvented the disk write bottleneck at the expense of fairness; despite having executed their logic, certain transactions are unable to commit until the set of buffered log records exceeds an appropriate threshold.

In prior work, we proposed a mechanism for lazy transaction evaluation, one of whose goals was to exploit spatial locality in on-line transaction processing systems [9]. Lazy transaction evaluation exploits spatial locality

[1] Liveness guards against partitions trivially achieving safety by choosing to abort transactions via system-induced aborts.

by amortizing the cost of bringing database records into the processor cache and main-memory buffer pool over several transactions. In order to achieve this, we proposed that a lazy database system defer the execution of transactions in order to obtain batches of unexecuted transactions whose read- and write-sets overlapped. Leaving transactions unexecuted meant that obtaining the most up-to-date value of a record would involve executing deferred transactions. Hence, when a client issued a query to read the value of a record, the query would need to initiate and wait for the execution of the appropriate set of deferred transactions. Lazy evaluation ensured that data-dependent transactions were executed together, so that the cost of bringing records into the processor cache and main-memory buffer pool was amortized across those transactions. This led to throughput benefits but came at the expense of fairness to client issued queries.

2.2 Isolation

Synchronization independence is the property that one transaction cannot cause another transaction to block or abort, even in the presence of conflicting data accesses [1]. Synchronization independence effectively decouples the execution of concurrently executing transactions. As a consequence, systems which allow transactions to run with synchronization independence are susceptible to race conditions due to conflicts.

Intuitively, synchronization independence implies that a system is unable to guarantee any form of isolation among conflicting transactions. We thus classify a system as providing weak isolation if it allows transactions to run with synchronization independence. If a system does not allow transactions to run with synchronization independence, it is classified as providing strong isolation.

3 The FIT Tradeoff

The safety and liveness requirements of Section 2 *necessitate* some form of coordination in processing a distributed transaction. The partitions involved in a transaction need to agree on whether or not to commit a transaction (safety), and cannot cheat by deliberately aborting transactions (liveness). In order to reach agreement, partitions require some form of coordination. The coordination involved in distributed transaction processing is at the heart of the FIT tradeoff.

The coordination associated with distributed transactions entails a decrease in concurrency in systems which guarantee strong isolation. If a transaction is in the process of executing, then, in order to guarantee strong isolation, conflicting transactions must not be allowed to make any meaningful progress. Distributed coordination extends the duration for which conflicting transactions are unable to make meaningful progress, which limits the throughput achievable under strong isolation.

This suggests that one way to reduce the impact of coordination is to allow transactions to run with synchronization independence (Section 2.2). Synchronization independence implies that conflicting transactions do not interfere with each other's execution. The implication of synchronization independence is that the inherent coordination required to execute a distributed transaction has no impact on the execution of other transactions. In other words, coordination has no impact on concurrency. As a consequence, systems which run transactions with synchronization independence are able to achieve good throughput. However, this comes at the expense of strong isolation. Since synchronization independence implies that transactions do not block due to conflicts, a weak isolation system does not stand to gain any benefits from sacrificing fairness; there is no point delaying or prioritizing transactions because there exist no inter-transaction dependencies in the first place.

Unlike systems that offer weak isolation, strong isolation systems must ensure that only a single transaction among many concurrent conflicting transactions is allowed to make meaningful progress. Any kind of latency due to coordination in the middle of a transaction thus reduces concurrency. In order to achieve good throughput, a strong isolation system must find a way to circumvent the concurrency penalty due to coordination. This can be done by giving up fairness. In giving up fairness, the database gains the flexibility to pick the most opportune

moment to pay the cost of coordination. *Which* moment is "most opportune" depends on the specific technique a system uses to circumvent the cost of coordination. For example, in some cases, it is possible to perform coordination outside of transaction boundaries so that the additional time required to do the coordination does not increase the time that conflicted transactions cannot run. In general, when the system does not need to guarantee fairness, it can deliberately prioritize or delay specific transactions in order to benefit overall throughput. A system which chooses to eschew fairness can guarantee isolation and good throughput.

A system which chooses fairness must make its best effort to quickly execute individual transactions. This constrains the system's ability to avoid the reduction in concurrency due to coordination. Therefore, systems which guarantee fairness and strong isolation cannot obtain good throughput.

In summary, the choice of how a system chooses to pay the cost of coordination entailed in supporting distributed transactions divides systems into three categories: 1) Systems that guarantee strong isolation and fairness at the expense of good throughput, 2) Systems that guarantee strong isolation and good throughput at the expense of fairness, and 3) Systems that guarantee good throughput and fairness at the expense of strong isolation.

4 FIT in Action

In this section, we examine several distributed database systems, and classify them according to which two properties of the FIT tradeoff they achieve.

4.1 Isolation-Throughput Systems

4.1.1 G-Store

G-Store is a system which extends a (non-transactional) distributed key-value store with support for multi-key transactions [5]. G-Store's design is tailored for applications that exhibit temporal locality, such as online multi-player games. During the course of a game instance, players interact with each other and modify each other's state. Transactions pertaining to a particular instance of a game will operate on only the keys corresponding to the game's participants.

In order to allow applications to exploit temporal locality, G-Store allows applications to specify a *KeyGroup*, which is a group of keys whose values are updated transactionally. G-Store requires that the scope of every transaction is restricted to the keys in a single KeyGroup. When an application creates a KeyGroup, G-Store moves the group's constituent keys from their respective partitions onto a single *leader* node. Transactions on a KeyGroup always execute on the group's leader node, which precludes the need for a distributed commit protocol.

By avoiding a distributed commit protocol, the period for which conflicting transactions on the same KeyGroup are blocked from making meaningful progress is significantly reduced. G-Store thus pays the cost of distributed coordination prior to transaction execution by moving all the keys in a KeyGroup to the leader node. KeyGroup creation involves an expensive grouping protocol, the cost of which is amortized across the transactions which execute on the KeyGroup.

The requirement that transactions restrict their scope to a single KeyGroup favors transactions that execute on keys which have already been grouped. On the flip side, this requirement is unfair to transactions that need to execute on a set of as yet ungrouped keys. Before such transactions can begin executing, G-Store must first disband existing KeyGroups to which some keys may belong, and then create the appropriate KeyGroup.

4.1.2 Calvin

Calvin is a database system designed to reduce the impact of coordination required while processing distributed transactions [14, 15]. Prior to their execution, Calvin runs transactions through a preprocessing layer, which imposes a total order on the transactions. The database's partitions then process transactions such that their serialization order is consistent with the total order, which implies that Calvin guarantees serializable isolation (satisfying our criteria for strong isolation).

In imposing a total order on transactions, the preprocessing layer solves two important problems:

- It eliminates deadlocks by pre-defining the order in which conflicting transactions must execute.
- It guarantees that a transaction's outcome is *deterministic*, despite the occurrence of failures. Each database partition is required to process transactions in a manner consistent with the total order, which implies that the total order of transactions is effectively a *redo log*; the state of the database is completely dependent on the total order of transactions.

The preprocessing layer is able to avoid deadlocks and ensure determinism because it is a logically (but not necessarily physically) *centralized* component[2] The use of a centralized component to disseminate information in a distributed system is a form of coordination. By funneling transactions through the preprocessing layer, Calvin pays some of the cost of distributed coordination up front, prior to the actual execution of transactions.

The guarantees of determinism and deadlock-freedom, obtained via coordination *prior* to transaction execution, keep the amount of coordination required *during* transaction execution down to a bare minimum. Deadlock-freedom eliminates any form of coordination required to deal with deadlocks. Determinism implies that, in the event of a failure, a partition can correctly re-construct its state from the total order of transactions. Therefore, partitions do not need to plan for failures using expensive operations such as forced log writes and synchronous replication. Minimizing the impact of coordination implies that the amount of time that conflicting transactions are blocked from making progress is also minimized, which helps Calvin achieve good throughput.

The preprocessing layer is replicated for fault-tolerance, and uses Paxos [11] to decide on a total ordering for transactions. Since it is logically centralized, the preprocessing layer must be able to totally order transactions at a rate which allows partitions to be fully utilized. If this were not the case, then Calvin's throughput would be bottlenecked by preprocessing layer. Accordingly, the preprocessing layer runs each Paxos instance over a large batch of transactions, amortizing the cost of consensus across the batch. In batching the input to each Paxos instance, the preprocessing layer sacrifices fairness; the transaction at the beginning of the batch must wait for enough transactions to accumulate before it can be processed. The *fair* strategy would be to run a Paxos instance on every transaction, however, its throughput would not be enough to fully utilize Calvin's partitions.

4.2 Fairness-Isolation Systems

4.2.1 Spanner

Spanner is Google's geo-scale distributed database system [4]. Spanner's data is partitioned for scale and each partition is replicated for fault-tolerance. Spanner uses multiversioning to support non-blocking read-only transactions, and a combination of two-phase locking (2PL) and two-phase commit (2PC) for transactions which perform updates. Spanner's concurrency control mechanisms guarantee that update transactions execute under strict serializability, which satisfies our criterion for strong isolation. Spanner processes transactions in a fair manner, a transaction begins executing as soon as it is submitted; Spanner does not deliberately delay transactions in order to benefit others.

[2]The predecessor of Calvin used a physically centralized preprocessor [13], while the more well-known versions of Calvin use a distributed preprocessor [14, 15].

Spanner commits transactions using 2PC. In a non-replicated implementation of 2PC, the coordinator must persist its commit/abort decision on disk, and each partition must similarly persist its prepare vote and the final commit decision. For correctness, some of these disk writes must be *forced*, which means that a node cannot take any further actions (related to the current instance of 2PC) until it is sure that the write is stable on disk. The coordinator's commit/abort decision and each partition's prepare vote fall into this category.

In a scenario where multiple copies of partitions exist, successful replication is analogous to a disk write. In order to correctly implement 2PC in a replicated setting, therefore, state must be "persisted" through replication. In keeping with the analogy, the equivalent of a forced write is *synchronous* replication. This means that before a distributed transaction can commit, it must pay the cost of replicating every partition's prepare vote (although per-partition replication occurs in parallel), and the cost of replicating the coordinator's final commit decision.

Spanner's 2PC protocol negatively impacts concurrency. In order to avoid cascading rollbacks and to guarantee strict serializable isolation, a transaction cannot release its locks until after it has obtained a commit decision, which implies that a transaction whose execution is blocked due to lock conflicts must wait for at least the duration of two synchronous replications, for *every* transaction that precedes it in the lock queue, for *every* lock it has not yet acquired. The end result is that Spanner's overall throughput is severely limited.

4.3 Fairness-Throughput Systems

4.3.1 Eventually Consistent Systems

Most eventually consistent systems do not support multi-record atomic transactions and therefore fall out of the scope of this article, which explores the fairness-isolation-throughput tradeoff involved in implementing distributed atomic transactions. For example, the original version of Dynamo supported only single-key updates [7][3]. As of the time of this article, Riak is in the same category [2].

However, Cassandra supports the notion of a "batch" — a "transaction" containing multiple INSERT, UPDATE, or DELETE statements that are performed as single logical operation — either the entire batch succeeds or none of it will [6]. Thus, a Cassandra batch supports the minimum level of atomicity to be within the scope of this article. However, there is no batch isolation. Other clients are able to read the partial results of the batch, even though these partial results may end up being undone if the batch were to abort. Thus, Cassandra's batch functionality clearly gives up isolation according to our definition in this article.

Since Cassandra makes no attempt to guarantee isolation of batches, the process of deciding whether a batch should succeed or abort can occur entirely independently of other batches or Cassandra operations. In general, throughput is unaffected by logical contention, and there is no need to delay or reorder transactions in order to improve throughput. Thus, by giving up isolation, the execution of Cassandra batches is both fair and occurs at high throughput.

4.3.2 RAMP Transactions

Bailis et al. propose a new isolation model, Read Atomic (RA) isolation, which ensures that either all or none of a transaction's updates are visible to others [1]. RA isolation's atomic visibility guarantee provides useful semantics on multi-partition operations where most weak isolation systems provide none.

RA isolation is implemented via Read Atomic Multi-Partition (RAMP) transactions. RAMP transactions guarantee *synchronization independence*, which means that a transaction *never* blocks others, even in the presence of read-write or write-write conflicts. Because conflicting transactions are allowed to concurrently update the same record(s), RAMP transactions cannot enforce value constraints (e.g., number of on call doctors must be greater than zero). We thus classify RAMP transactions as guaranteeing weak isolation.

[3] DynamoDB, a relatively recent Amazon Web Services product that is based partly on the design of Dynamo, supports multi-record atomic transactions. This OCC-based implementation gives up throughput to achieve isolation and fairness.

In order to provide atomicity (*not* the same as atomic visibility — see Section 2), RAMP transactions execute a commit protocol which resembles two-phase commit. Synchronization independence ensures that a transactions's commit protocol does not block the execution of conflicting transactions. As a consequence, the coordination entailed by RAMP transactions's commit protocol does not impact concurrency, which implies that RAMP transactions can achieve good throughput. Furthermore, because coordination does not impact concurrency, the system does not need to rely on unfairness to mask the cost of coordination. For this reason, RAMP transactions can guarantee good throughput *and* fairness simultaneously.

5 Applicability Beyond Distributed Transactions

One of the fundamental realities of modern multicore architectures is that contention on shared memory words can lead to severe scalability problems due to cache coherence overhead [3]. The impact of the cost of contention in multicore systems is analogous to the impact of distributed coordination in multi-partition transactions, it is an unavoidable cost that the database must mitigate in order to attain good performance. This suggests that the FIT tradeoff can also help guide the design of database systems on multicore machines. We now examine two recent serializable databases specifically designed to address the contention bottleneck on multicore machines by trading off fairness for overall system throughput.

Silo is a main-memory database system explicitly designed to reduce contention on shared memory words [16]. Like the main-memory system of DeWitt et al. [8], Silo buffers transactions's log records in memory prior to flushing them to disk. However, Silo's group commit technique is fundamentally different from that of DeWitt et al. because it is primarily designed to minimize synchronization across a machine's cores. Whenever a processing core appends log records to the in-memory buffer, it must synchronize with the other cores in the system (because the in-memory buffer is shared among all cores). In order to avoid this synchronization cost at the end of every transaction, each core tracks a transaction's log records in a *core-local* buffer. Cores periodically move the contents of their local buffers to the global buffer at the granularity of a large batch of transactions, amortizing the cost of synchronization over the batch. This reduction in synchronization comes at the cost of fairness; a transaction is unable to commit until its log records are moved to the global buffer and flushed to disk, even though it may have finished executing its logic much earlier.

Doppel is a main-memory database system which can exploit commutativity to increase the amount of concurrency in transaction processing [12]. Doppel exploits commutativity by periodically replicating high contention records across all cores in the system; when a record is replicated, commuting operations are allowed to execute on any given replica. In order to satisfy non-commuting operations, Doppel periodically aggregates the value at each replica into a single record. Doppel effectively cycles between two phases of operation: 1) a "joined-phase", in which only a single replica of a record exists, with no restrictions on which transactions are allowed to execute, 2) a "split-phase", in which high contention records are replicated across all cores, and only transactions with commuting operations on replicated records are allowed to execute (there exist no restrictions on operations on non-replicated records). Doppel is able to obtain concurrency in the split-phase by allowing commuting operations on replicated records. However, replicating records blocks the execution of transactions which include non-commuting operations on replicated records, which implies that Doppel is unable to guarantee fairness.

6 Conclusions

This article made the case for the existence of the fairness-isolation-throughput (FIT) tradeoff in distributed database systems. The FIT tradeoff dictates that a distributed database system can pick only two of these three properties; a system which foregoes one of these three properties can guarantee the other two. We believe that the FIT tradeoff is also applicable to the design of multicore database systems.

Although system implementers have long treated transaction isolation and throughput as "first-class citizens" of database design, we hope that explicitly thinking about the concept of "fairness" will help database architects more completely understand the tradeoffs involved in building distributed database systems.

7 Acknowledgments

We thank Peter Bailis and Alexander Thomson for providing insightful feedback on this article.

References

[1] P. Bailis, A. Fekete, J. M. Hellerstein, A. Ghodsi, and I. Stoica. Scalable atomic visibility with ramp transactions. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, pages 27–38. ACM, 2014.

[2] Basho. Riak. `http://basho.com/riak`.

[3] S. Boyd-Wickizer, M. F. Kaashoek, R. Morris, and N. Zeldovich. Non-scalable locks are dangerous. In *Proceedings of the Linux Symposium*, pages 119–130, 2012.

[4] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, et al. Spanner: Google's globally distributed database. *ACM Transactions on Computer Systems (TOCS)*, 31(3):8, 2013.

[5] S. Das, D. Agrawal, and A. El Abbadi. G-store: a scalable data store for transactional multi key access in the cloud. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 163–174. ACM, 2010.

[6] Datastax. Cql for cassandra 1.2. `http://www.datastax.com/documentation/cql/3.0/cql/cql_reference/batch_r.html`.

[7] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, pages 205–220. ACM, 2007.

[8] D. J. DeWitt, R. H. Katz, F. Olken, L. D. Shapiro, M. R. Stonebraker, and D. A. Wood. Implementation techniques for main memory database systems. In *Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data*, pages 1–8. ACM, 1984.

[9] J. M. Faleiro, A. Thomson, and D. J. Abadi. Lazy evaluation of transactions in database systems. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, pages 15–26. ACM, 2014.

[10] S. Gilbert and N. Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *ACM SIGACT News*, 33(2):51–59, 2002.

[11] L. Lamport. The part-time parliament. *ACM Transactions on Computer Systems (TOCS)*, 16(2):133–169, 1998.

[12] N. Narula, C. Cutler, E. Kohler, and R. Morris. Phase reconciliation for contended in-memory transactions. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, pages 511–524. USENIX Association, 2014.

[13] A. Thomson and D. J. Abadi. The case for determinism in database systems. *Proceedings of the VLDB Endowment*, 3(1-2):70–80, 2010.

[14] A. Thomson, T. Diamond, S.-C. Weng, K. Ren, P. Shao, and D. J. Abadi. Calvin: Fast distributed transactions for partitioned database systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 1–12. ACM, 2012.

[15] A. Thomson, T. Diamond, S.-C. Weng, K. Ren, P. Shao, and D. J. Abadi. Fast distributed transactions and strongly consistent replication for oltp database systems. *ACM Transactions on Database Systems (TODS)*, 39(2):11, 2014.

[16] S. Tu, W. Zheng, E. Kohler, B. Liskov, and S. Madden. Speedy transactions in multicore in-memory databases. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 18–32. ACM, 2013.