Coordination Avoidance in Database Systems

Peter Bailis, Alan Fekete[†], Michael J. Franklin, Ali Ghodsi, Joseph M. Hellerstein, Ion Stoica
UC Berkeley and [†]University of Sydney

ABSTRACT

Minimizing coordination, or blocking communication between concurrently executing operations, is key to maximizing scalability, availability, and high performance in database systems. However, uninhibited coordination-free execution can compromise application correctness, or consistency. When is coordination necessary for correctness? The classic use of serializable transactions is sufficient to maintain correctness but is not necessary for all applications, sacrificing potential scalability. In this paper, we develop a formal framework, invariant confluence, that determines whether an application requires coordination for correct execution. By operating on application-level invariants over database states (e.g., integrity constraints), invariant confluence analysis provides a necessary and sufficient condition for safe, coordination-free execution. When programmers specify their application invariants, this analysis allows databases to coordinate only when anomalies that might violate invariants are possible. We analyze the invariant confluence of common invariants and operations from real-world database systems (i.e., integrity constraints) and applications and show that many are invariant confluent and therefore achievable without coordination. We apply these results to a proof-of-concept coordination-avoiding database prototype and demonstrate sizable performance gains compared to serializable execution, notably a 25-fold improvement over prior TPC-C New-Order performance on a 200 server cluster.

1. INTRODUCTION

Minimizing coordination is key in high-performance, scalable database design. Coordination—informally, the requirement that concurrently executing operations synchronously communicate or otherwise stall in order to complete—is expensive: it limits concurrency between operations and undermines the effectiveness of scale-out across servers. In the presence of partial system failures, coordinating operations may be forced to stall indefinitely, and, in the failure-free case, communication delays can increase latency [9, 28]. In contrast, coordination-free operations allow aggressive scale-out, availability [28], and low latency execution [1]. If operations are coordination-free, then adding more capacity (e.g., servers, processors) will result in additional throughput; operations can execute on the new resources without affecting the old set of resources. Partial failures will not affect non-failed operations, and latency between any database replicas can be hidden from end-users.

Unfortunately, coordination-free execution is not always safe. Uninhibited coordination-free execution can compromise application-level correctness, or consistency.[1] In canonical banking application examples, concurrent, coordination-free withdrawal operations can result in undesirable and "inconsistent" outcomes like negative account balances—application-level anomalies that the database should prevent. To ensure correct behavior, a database system must coordinate the execution of these operations that, if otherwise executed concurrently, could result in inconsistent application state.

This tension between coordination and correctness is evidenced by the range of database concurrency control policies. In traditional database systems, serializable isolation provides concurrent operations (transactions) with the illusion of executing in some serial order [15]. As long as individual transactions maintain correct application state, serializability guarantees correctness [30]. However, each pair of concurrent operations (at least one of which is a write) can potentially compromise serializability and therefore will require coordination to execute [9, 21]. By isolating users at the level of reads and writes, serializability can be overly conservative and may in turn coordinate more than is strictly necessary for consistency [29, 39, 53, 58]. For example, hundreds of users can safely and simultaneously retweet Barack Obama on Twitter without observing a serial ordering of updates to the retweet counter. In contrast, a range of widely-deployed weaker models require less coordination to execute but surface read and write behavior that may in turn compromise consistency [2, 9, 22, 48]. With these alternative models, it is up to users to decide when weakened guarantees are acceptable for their applications [6], leading to confusion regarding (and substantial interest in) the relationship between consistency, scalability, and availability [1, 9, 12, 18, 21, 22, 28, 40].

In this paper, we address the central question inherent in this tradeoff: when is coordination strictly necessary to maintain application-level consistency? To do so, we enlist the aid of application programmers to specify their correctness criteria in the form of *invariants*. For example, our banking application writer would specify that account balances should be positive (e.g., by schema annotations), similar to constraints in modern databases today. Using these invariants, we formalize a *necessary* and sufficient condition for invariant-preserving and coordination-free execution of an application's operations—the first such condition we have encountered. This property—invariant confluence ($\mathcal{I}$-confluence)—captures the potential scalability and availability of an application, independent of any particular database implementation: if an application's operations are $\mathcal{I}$-confluent, a database can correctly execute them without coordination. If operations are not $\mathcal{I}$-confluent, coordination is required to guarantee correctness. This provides a basis for *coordination avoidance*: the use of coordination only when necessary.

While coordination-free execution is powerful, are any *useful* operations safely executable without coordination? $\mathcal{I}$-confluence analysis determines when concurrent execution of specific operations can be "merged" into valid database state; we accordingly

 Articles from this volume were invited to present their results at the 41st International Conference on Very Large Data Bases, Aug. 31st - Sept. 4th, 2015, Kohala Coast, Hawaii.
Proceedings of the VLDB Endowment, Vol. 8, No. 3

[1] Our use of the term "consistency" in this paper refers to *application-level* correctness, as is traditional in the database literature [15, 21, 25, 30, 56]. As we discuss in Section 5, replicated data consistency (and isolation [2, 9]) models like linearizability [28] can be cast as application criteria if desired.

analyze invariants and operations from several real-world databases and applications. Many production databases today already support invariants in the form of primary key, uniqueness, foreign key, and row-level check constraints [9, 42]. We analyze these and show many are $\mathcal{I}$-confluent, including forms of foreign key constraints, unique value generation, and check constraints, while others, like primary key constraints are, in general, not. We also consider entire *applications* and apply our analysis to the workloads of the OLTP-Benchmark suite [23]. Many of the operations and invariants are $\mathcal{I}$-confluent. As an extended case study, we examine the TPC-C benchmark [55], the preferred standard for evaluating new concurrency control algorithms [23, 35, 46, 52, 54]. We show that ten of twelve of TPC-C's invariants are $\mathcal{I}$-confluent under the workload transactions and, more importantly, compliant TPC-C can be implemented without any synchronous coordination across servers. We subsequently scale a coordination-avoiding database prototype linearly, to over 12.7M TPC-C New-Order transactions per second on 200 servers, a 25-fold improvement over prior results.

Overall, $\mathcal{I}$-confluence offers a concrete grasp on the challenge of minimizing coordination while ensuring application-level correctness. In seeking a necessary and sufficient (i.e., "tight") condition for safe, coordination-free execution, we require the programmer to specify her correctness criteria. If either these criteria or application operations are unavailable for inspection, users must fall back to using serializable transactions or, alternatively, perform the same ad-hoc analyses they use today [12]. Moreover, it is already well known that coordination is required to prevent several read/write isolation anomalies like non-linearizable operations [9, 28]. However, when users *can* correctly specify their application correctness criteria and operations, they can maximize scalability without requiring expertise in the milieu of weak read/write isolation models [2, 9]. We have also found that $\mathcal{I}$-confluence to be a useful design tool: studying specific combinations of invariants and operations can indicate the existence of more scalable algorithms [18].

In summary, this paper offers the following high-level takeaways:

1. Serializable transactions preserve application correctness at the cost of always coordinating between conflicting reads and writes.
2. Given knowledge of application transactions and correctness criteria (e.g., invariants), it is often possible to avoid this coordination (by executing some transactions without coordination, thus providing availability, low latency, and excellent scalability) while still preserving those correctness criteria.
3. Invariant confluence offers a necessary and sufficient condition for this correctness-preserving, coordination-free execution.
4. Many common integrity constraints found in SQL and standardized benchmarks are invariant confluent, allowing order-of-magnitude performance gains over coordinated execution.

While coordination cannot always be avoided, this work evidences the power of application invariants in scalable and correct execution of modern applications on modern hardware. Application correctness does not always require coordination, and $\mathcal{I}$-confluence analysis can explain both when and why this is the case.

Overview. The remainder of this paper proceeds as follows: Section 2 describes and quantifies the costs of coordination. Section 3 introduces our system model and Section 4 contains our primary theoretical result. Readers may skip to Section 5 for practical applications of $\mathcal{I}$-confluence to real-world invariant-operation combinations. Section 6 subsequently applies these combinations to real applications and presents an experimental case study of TPC-C. Section 7 describes related work, and Section 8 concludes.

2. CONFLICTS AND COORDINATION

As repositories for application state, databases are traditionally tasked with maintaining correct data on behalf of users. During concurrent access to data, a database ensuring correctness must therefore decide which user operations can execute simultaneously and which, if any, must coordinate, or block. In this section, we explore the relationship between the correctness criteria that a database attempts to maintain and the coordination costs of doing so.

By example. As a running example, we consider a database-backed payroll application that maintains information about employees and departments within a small business. In the application, *a.*) each employee is assigned a unique ID number and *b.*) each employee belongs to exactly one department. A database ensuring correctness must maintain these application-level properties, or *invariants* on behalf of the application (i.e., without application-level intervention). In our payroll application, this is non-trivial: for example, if the application attempts to simultaneously create two employees, then the database must ensure the employees are assigned distinct IDs.

Serializability and conflicts. The classic answer to maintaining application-level invariants is to use serializable isolation: execute each user's ordered sequence of operations, or *transactions*, such that the end result is equivalent to some sequential execution [15, 30, 53]. If each transaction preserves correctness in isolation, composition via serializable execution ensures correctness. In our payroll example, the database would execute the two employee creation transactions such that one transaction appears to execute after the other, avoiding duplicate ID assignment.

While serializability is a powerful abstraction, it comes with a cost: for arbitrary transactions (and for all implementations of serializability's more conservative variant—conflict serializability), any two operations to the same item—at least one of which is a write—will result in a *read/write conflict*. Under serializability, these conflicts require coordination or, informally, blocking communication between concurrent transactions: to provide a serial ordering, conflicts must be totally ordered across transactions [15]. For example, given database state $\{x = \bot, y = \bot\}$, if transaction T_1 writes $x = 1$ and reads from y and T_2 writes $y = 1$ and reads from x, a database cannot both execute T_1 and T_2 entirely concurrently and maintain serializability [9, 21].

The costs of coordination. The coordination overheads above incur three primary penalties: increased latency (due to stalled execution), decreased throughput, and, in the event of partial failures, unavailability. If a transaction takes d seconds to execute, the maximum throughput of conflicting transactions operating on the same items under a general-purpose (i.e., interactive, non-batched) transaction model is limited by $\frac{1}{d}$, while coordinating operations will also have to wait. On a single system, delays can be small, permitting tens to hundreds of thousands of conflicting transactions per item per second. In a partitioned database system, where different items are located on different servers, or in a replicated database system, where the same item is located (and is available for operations) on multiple servers, the cost increases: delay is lower-bounded by network latency. On a local area network, delay may vary from several microseconds (e.g., via Infiniband or RDMA) to several milliseconds on today's cloud infrastructure, permitting anywhere from a few hundred transactions to a few hundred thousand transactions per second. On a wide-area network, delay is lower-bounded by the speed of light (worst-case on Earth, around 75ms, or about 13 operations per second [9]). Under network partitions [13], as delay tends towards infinity, these penalties lead to unavailability [9, 28]. In contrast, operations executing without coordination can proceed concurrently and will not incur these penalties.

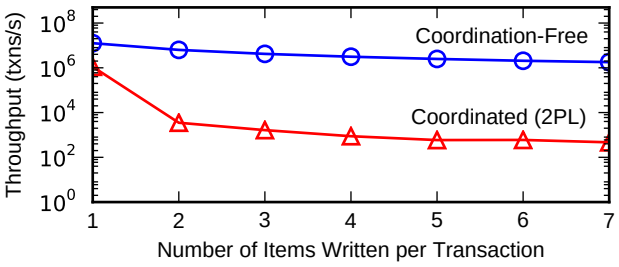


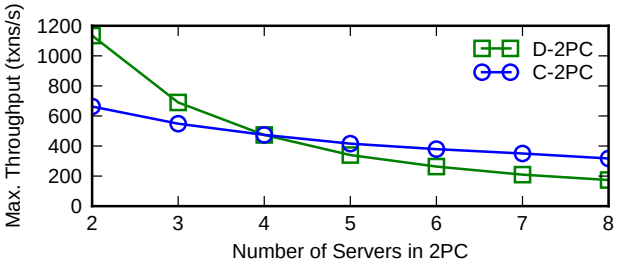
Figure 1: Microbenchmark performance of coordinated and coordination-free execution of transactions of varying size writing to eight items located on eight separate multi-core servers.

Quantifying coordination overheads. To further understand the costs of coordination, we performed two sets of measurements—one using a database prototype and one using traces from prior studies.

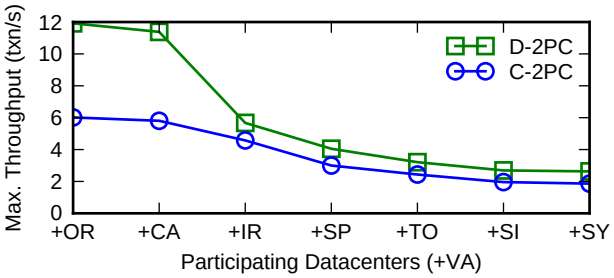
We first compared the throughput of a set of coordinated and coordination-free transaction execution. We partitioned a set of eight data items across eight servers and ran one set of transactions with an optimized variant of two-phase locking (providing serializability) [15] and ran another set of transactions without coordination (Figure 1; see [10, Appendix A] for more details). With single-item, non-distributed transactions, the coordination-free implementation achieves, in aggregate, over 12M transactions per second and bottlenecks on *physical resources*—namely, CPU cycles. In contrast, the lock-based implementation achieves approximately 1.1M transactions per second: it is unable to fully utilize all multi-core processor contexts due to lock contention. For distributed transactions, coordination-free throughput decreases linearly (as an N-item transaction performs N writes), while the throughput of coordinating transactions drops by over three orders of magnitude.

While the above microbenchmark demonstrates the costs of a particular *implementation* of coordination, we also studied the effect of more fundamental, implementation-independent overheads (i.e., also applicable to optimistic and scheduling-based concurrency control mechanisms). We determined the maximum attainable throughput for coordinated execution within a single datacenter (based on data from [60]) and across multiple datacenters (based on data from [9]) due to blocking coordination during atomic commitment [15]. For an N-server transaction, classic two-phase commit (`C-2PC`) requires N (parallel) coordinator to server RTTs, while decentralized two-phase commit (`D-2PC`) requires N (parallel) server to server broadcasts, or N^2 messages. Figure 2 shows that, in the local area, with only two servers (e.g., two replicas or two coordinating operations on items residing on different servers), throughput is bounded by 1125 transactions/s (via `D-2PC`; 668/s via `C-2PC`). Across eight servers, `D-2PC` throughput drops to 173 transactions/s (resp. 321 for `C-2PC`) due to long-tailed latency distributions. In the wide area, the effects are more stark: if coordinating from Virginia to Oregon, `D-2PC` message delays are 83 ms per commit, allowing 12 operations per second. If coordinating between all eight EC2 availability zones, throughput drops to slightly over 2 transactions/s in both algorithms. ([10, Appendix A] provides more details.)

These results should be unsurprising: coordinating—especially over the network—can incur serious performance penalties. In contrast, coordination-free operations can execute without incurring these costs. The costs of actual workloads can vary: if coordinating operations are rare, concurrency control will not be a bottleneck. For example, a serializable database executing transactions with disjoint read and write sets can perform as well as a non-serializable database without compromising correctness [34]. However, as these



a.) Maximum transaction throughput over local-area network in [60]



b.) Maximum throughput over wide-area network in [9] with transactions originating from a coordinator in Virginia (VA; OR: Oregon, CA: California, IR: Ireland, SP: São Paulo, TO: Tokyo, SI: Singapore, SY: Sydney)

Figure 2: Atomic commitment latency as an upper bound on throughput over LAN and WAN networks.

results demonstrate, minimizing the amount of coordination and its degree of distribution can therefore have a tangible impact on performance, latency, and availability [1, 9, 28]. While we study real applications in Section 6, these measurements highlight the worst of coordination costs on modern hardware.

Our goal: Minimize coordination. In this paper, we seek to minimize the amount of coordination required to correctly execute an application's transactions. As discussed in Section 1, serializability is *sufficient* to maintain correctness but is not always *necessary*; that is, many—but not all—transactions can be executed concurrently without necessarily compromising application correctness. In the remainder of this paper, we identify when safe, coordination-free execution is possible. If serializability requires coordinating between each possible pair of conflicting reads and writes, we will only coordinate between pairs of operations that might compromise *application-level* correctness. To do so, we must both raise the specification of correctness beyond the level of reads and writes and directly account for the process of reconciling the effects of concurrent transaction execution at the application level.

3. SYSTEM MODEL

To characterize coordination avoidance, we first present a system model. We begin with an informal overview. In our model, transactions operate over independent (logical) "snapshots" of database state. Transaction writes are applied at one or more snapshots initially when the transaction commits and then are integrated into other snapshots asynchronously via a "merge" operator that incorporates those changes into the snapshot's state. Given a set of invariants describing valid database states, as Table 1 outlines, we seek to understand when it is possible to ensure invariants are always satisfied (global validity) while guaranteeing a response (transactional availability) and the existence of a common state (convergence), all without communication during transaction execution (coordination-freedom). This model need not directly correspond to

Property	Effect
Global validity	Invariants hold over committed states
Transactional availability	Non-trivial response guaranteed
Convergence	Updates are reflected in shared state
Coordination-freedom	No synchronous coordination

Table 1: Key properties of the system model and their effects.

a given implementation (e.g., see the database architecture in Section 6)—rather, it serves as a useful abstraction. The remainder of this section further defines these concepts; readers more interested in their application should proceed to Section 4. We provide greater detail and additional discussion in [10].

Databases. We represent a state of the shared database as a set D of unique *versions* of data items located on an arbitrary set of database servers, and each version is located on at least one server. We use $\mathcal{D}$ to denote the set of possible database states—that is, the set of sets of versions. The database is initially populated by an initial state D_0 (typically but not necessarily empty).

Transactions, Replicas, and Merging. Application clients submit requests to the database in the form of transactions, or ordered groups of operations on data items that should be executed together. Each transaction operates on a logical *replica*, or set of versions of the items mentioned in the transaction. At the beginning of the transaction, the replica contains a subset of the database state and is formed from all of the versions of the relevant items that can be found at one or more physical servers that are contacted during transaction execution. As the transaction executes, it may add versions (of items in its writeset) to its replica. Thus, we define a transaction T as a transformation on a replica: $T : \mathcal{D} \to \mathcal{D}$. We treat transactions as opaque transformations that can contain writes (which add new versions to the replica's set of versions) or reads (which return a specific set of versions from the replica). (Later, we will discuss transactions operating on data types such as counters.)

Upon completion, each transaction can *commit*, signaling success, or *abort*, signaling failure. Upon commit, the replica state is subsequently *merged* ($\sqcup$:$\mathcal{D} \times \mathcal{D} \to \mathcal{D}$) into the set of versions at least one server. We require that the merged effects of a committed transaction will eventually become visible to other transactions that later begin execution on the same server.[2] Over time, effects propagate to other servers, again through the use of the merge operator. Though not strictly necessary, we assume this merge operator is commutative, associative, and idempotent [5, 50]. In our initial model, we define merge as set union of the versions contained at different servers. (Section 5 discusses additional implementations.) For example, if server $R_x = \{v\}$ and $R_y = \{w\}$, then $R_x \sqcup R_y = \{v, w\}$.

In effect, each transaction can modify its replica state without modifying any other concurrently executing transactions' replica state. Replicas therefore provide transactions with partial "snapshot" views of global state (that we will use to simulate concurrent executions, similar to revision diagrams [17]). Importantly, two transactions' replicas do not necessarily correspond to two physically separate servers; rather, a replica is simply a partial "view" over the global state of the database system. For now, we assume transactions are known in advance (see also [10, Section 8]).

Invariants. To determine whether a database state is valid according to application correctness criteria, we use *invariants*, or predicates over replica state: $I : \mathcal{D} \to \{true, false\}$ [25]. In our payroll example, we could specify an invariant that only one user in a database has a given ID. This invariant—as well as almost all invariants we consider—is naturally expressed as a part of the database schema (e.g., via DDL); however, our approach allows us to reason about invariants even if they are known to the developer but not declared to the system. Invariants directly capture the notion of ACID Consistency [15, 30], and we say that a database state is *valid* under an invariant I (or I-valid) if it satisfies the predicate:

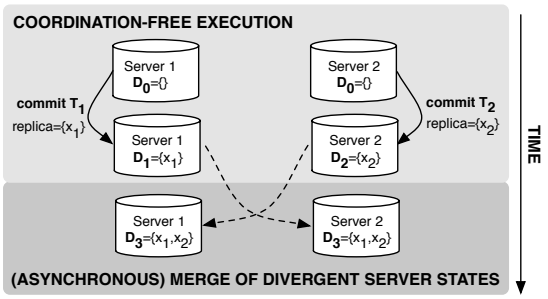


Figure 3: An example coordination-free execution of two transactions, T_1 and T_2, on two servers. Each transaction writes to its local replica, then, after commit, the servers asynchronously exchange state and converge to a common state (D_3).

Definition 1. A replica state $R \in \mathcal{D}$ is *I-valid* iff $I(R) = true$.

We require that D_0 be valid under invariants. Section 4.3 provides additional discussion regarding our use of invariants.

Availability. To ensure each transaction receives a non-trivial response, we adopt the following definition of *availability* [9]:

Definition 2. A system provides *transactionally available* execution iff, whenever a client executing a transaction T can access servers containing one or more versions of each item in T, then T eventually commits or aborts itself either due to an *abort* operation in T or if committing the transaction would violate a declared invariant over T's replica state. T will commit in all other cases.

Under the above definition, a transaction can only abort if it explicitly chooses to abort itself or if committing would violate invariants over the transaction's replica state.[3]

Convergence. Transactional availability allows replicas to maintain valid state *independently*, but it is vacuously possible to maintain "consistent" database states by letting replicas diverge (contain different state) forever. This guarantees *safety* (nothing bad happens) but not *liveness* (something good happens) [49]. To enforce state sharing, we adopt the following definition:

Definition 3. A system is *convergent* iff, for each pair of servers, in the absence of new writes to the servers and in the absence of indefinite communication delays between the servers, the servers eventually contain the same versions for any item they both store.

To capture the process of reconciling divergent states, we use the previously introduced merge operator: given two divergent server states, we apply the merge operator to produce convergent state. We assume the effects of merge are atomically visible: either all effects of a merge are visible or none are. This assumption is not always

[2] This implicitly disallows servers from always returning the initial database state when they have newer writes on hand. This is a relatively pragmatic assumption but also simplifies our later reasoning about admissible executions. This assumption could possibly be relaxed by adapting Newman's lemma [24], but we do not consider the possibility here.

[3] This basic definition precludes fault tolerance (i.e., durability) guarantees beyond a single server failure [9]. We can relax this requirement and allow communication with a fixed number of servers (e.g., $F+1$ servers for F-fault tolerance; F is often small [22]) without affecting our results. This does not affect scalability because, as more replicas are added, the communication overhead required for durability remains constant.

necessary but it simplifies our discussion and, as we later discuss, is maintainable without coordination [9, 11].

Maintaining validity. To make sure that both divergent and convergent database states are valid and, therefore, that transactions never observe invalid states, we introduce the following property:

Definition 4. A system is *globally I-valid* iff all replicas always contain I-valid state.

Coordination. Our system model is missing one final constraint on coordination between concurrent transaction execution:

Definition 5. A system provides coordination-free execution for a set of transactions T iff the progress of executing each $t \in T$ is only dependent on the versions of the items t reads (i.e., t's replica state).

That is, in a coordination-free execution, each transaction's progress towards commit/abort is independent of other operations (e.g., writes, locking, validations) being performed on behalf of other transactions. This precludes blocking synchronization or communication across concurrently executing transactions.

By example. Figure 3 illustrates a coordination-free execution of two transactions T_1 and T_2 on two separate, fully-replicated physical servers. Each transaction commits on its local replica, and the result of each transaction is reflected in the transaction's local server state. After the transactions have completed, the servers exchange state and, after applying the merge operator, converge to the same state. Any transactions executing later on either server will obtain a replica that includes the effects of both transactions.

4. CONSISTENCY SANS COORDINATION

With a system model and goals in hand, we now address the question: when do applications require coordination for correctness? The answer depends not just on an application's transactions or on an application's invariants. Rather, the answer depends on the *combination* of the two under consideration. Our contribution in this section is to formulate a criterion that will answer this question for specific combinations in an implementation-agnostic manner.

In this section, we focus almost exclusively on providing a formal answer to this question. The remaining sections of this paper are devoted to practical interpretation and application of these results.

4.1 $\mathcal{I}$-confluence: Criteria Defined

To begin, we introduce the central property (adapted from the constraint programming literature [24]) in our main result: invariant confluence (hereafter, $\mathcal{I}$-confluence). Applied in a transactional context, the $\mathcal{I}$-confluence property informally ensures that divergent but I-valid database states can be merged into a valid database state—that is, the set of valid states reachable by executing transactions and merging their results is closed (w.r.t. validity) under merge. In the next sub-section, we show that $\mathcal{I}$-confluence analysis directly determines the potential for safe, coordination-free execution.

We say that S_i is a *I-T-reachable state* if, given an invariant I and set of transactions T (with merge function $\sqcup$), there exists a (partially ordered) sequence of transaction and merge function invocations that yields S_i, and each intermediate state produced by transaction execution or merge invocation is also I-valid. We call these previous states *ancestor states*. Note that each ancestor state is either I-T-reachable or is instead the initial state (D_0).

We can now formalize the $\mathcal{I}$-confluence property:

Definition 6 ($\mathcal{I}$-confluence)**.** A set of transactions T is $\mathcal{I}$-confluent with respect to invariant I if, for all I-T-reachable states D_i, D_j with a common ancestor state, $D_i \sqcup D_j$ is I-valid.

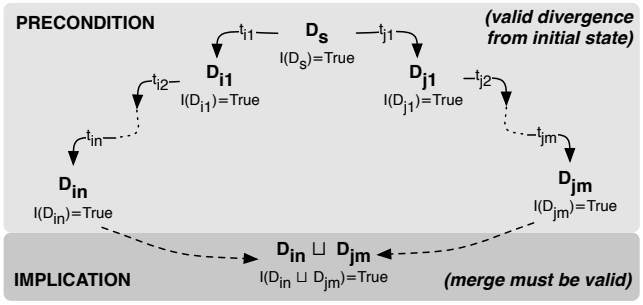


Figure 4: An $\mathcal{I}$-confluent execution illustrated via a diamond diagram. If a set of transactions T is $\mathcal{I}$-confluent, then all database states reachable by executing and merging transactions in T starting with a common ancestor (D_s) must be mergeable ($\sqcup$) into an I-valid database state.

Figure 4 depicts an $\mathcal{I}$-confluent merge of two I-T-reachable states, each starting from a shared, I-T-reachable state D_s. Two sequences of transactions $t_{in} \ldots t_{i1}$ and $t_{jm} \ldots t_{j1}$ each independently modify D_s. Under $\mathcal{I}$-confluence, the states produced by these sequences (D_{in} and D_{jm}) must be valid under merge.[4]

$\mathcal{I}$-confluence holds for specific combinations of invariants and transactions. In our payroll database example from Section 2, removing a user from the database is $\mathcal{I}$-confluent with respect to the invariant that user IDs are unique. However, two transactions that remove two different users from the database are not $\mathcal{I}$-confluent with respect to the invariant that there exists at least one user in the database at all times. Section 5 discusses additional combinations of invariants (with greater precision).

4.2 $\mathcal{I}$-confluence and Coordination

We can now apply $\mathcal{I}$-confluence to our goals from Section 3:

Theorem 1. A globally I-valid system can execute a set of transactions T with coordination-freedom, transactional availability, convergence if and only if T is $\mathcal{I}$-confluent with respect to I.

We provide a full proof of Theorem 1 in [10, Appendix B] (which is straightforward) but provide a sketch here. The backwards direction is by construction: if $\mathcal{I}$-confluence holds, each replica can check each transaction's modifications locally and replicas can merge independent modifications to guarantee convergence to a valid state. The forwards direction uses a partitioning argument [28] to derive a contradiction: we construct a scenario under which a system cannot determine whether a non-$\mathcal{I}$-confluent transaction should commit without violating one of our desired properties (either compromising validity or availability, diverging forever, or coordinating).

Theorem 1 establishes $\mathcal{I}$-confluence as a necessary and sufficient condition for invariant-preserving, coordination-free execution. If $\mathcal{I}$-confluence holds, there exists a correct, coordination-free execution strategy for the transactions; if not, no possible implementation can guarantee these properties for the provided invariants and transactions. That is, if $\mathcal{I}$-confluence does not hold, there exists at least one execution of transactions on separate replicas that will violate the given invariants when servers converge. To prevent invalid states from occurring, at least one of the transaction sequences will have to forego availability or coordination-freedom, or the system will have to forego convergence. $\mathcal{I}$-confluence analysis is independent of any given implementation, and effectively "lifts" prior discussions

[4] We require these states to have a common ancestor to rule out the possibility of merging states that could not have arisen from transaction execution (e.g., even if no transaction assigns IDs, merging two states that each have unique but overlapping sets of IDs could be invalid).

of scalability, availability, and low latency [1, 9, 28] to the level of application (i.e., not "I/O" [6]) correctness. This provides a useful handle on the implications of coordination-free execution without requiring reasoning about low-level properties such as physical data location and the number of servers.

4.3 Discussion and Limitations

$\mathcal{I}$-confluence captures a simple (informal) rule: ***coordination can only be avoided if all local commit decisions are globally valid***. (Alternatively, commit decisions are composable.) If two independent decisions to commit can result in invalid converged state, then replicas must coordinate in order to ensure that only one of the decisions is to commit. Given the existence of an unsafe execution and the inability to distinguish between safe and invalid executions using only local information, a globally valid system *must* coordinate in order to prevent the invalid execution from arising.

Use of invariants. Our use of invariants in $\mathcal{I}$-confluence is key to achieving a *necessary* and not simply sufficient condition. By directly capturing application-level correctness criteria via invariants, $\mathcal{I}$-confluence analysis only identifies "true" conflicts. This allows $\mathcal{I}$-confluence analysis to perform a more accurate assessment of whether coordination is needed compared to related conditions such as commutativity (Section 7).

However, the reliance on invariants also has drawbacks. $\mathcal{I}$-confluence analysis only guards against violations of any provided invariants. If invariants are incorrectly or incompletely specified, an $\mathcal{I}$-confluent database system may violate application-level correctness. If users cannot guarantee the correctness and completeness of their invariants and operations, they should opt for a more conservative analysis or mechanism such as employing serializable transactions. Accordingly, our development of $\mathcal{I}$-confluence analysis provides developers with a powerful option—but only if used correctly. If used incorrectly, $\mathcal{I}$-confluence allows incorrect results, or, if not used at all, developers must resort to existing alternatives.

This final point raises several questions: can we specify invariants in real-world use cases? Classic database concurrency control models assume that "the [set of application invariants] is generally not known to the system but is embodied in the structure of the transaction" [25, 56]. Nevertheless, since 1976, databases have introduced support for a finite set of invariants [14, 26, 29, 32, 37] in the form of primary key, foreign key, uniqueness, and row-level "check" constraints [42]. We can (and, in this paper, do) analyze these invariants, which can—like many program analyses [18]—lead to new insights about execution strategies. We have found the process of invariant specification to be non-trivial but feasible in practice; Section 6 describes some of our experiences.

(Non-)determinism. $\mathcal{I}$-confluence analysis effectively captures points of *unsafe non-determinism* [6] in transaction execution. As we have seen in many of our examples thus far, total non-determinism under concurrent execution can compromise application-level consistency [5, 36]. But not all non-determinism is bad: many desirable properties (e.g., classical distributed consensus among processes) involve forms of acceptable non-determinism (e.g., any proposed outcome is acceptable as long as all processes agree) [31]. In many cases, maximizing safe concurrency requires non-determinism.

$\mathcal{I}$-confluence analysis allows this non-deterministic divergence of database states but makes two useful guarantees about those states. First, the requirement for global validity ensures safety (in the form of invariants). Second, the requirement for convergence ensures liveness (in the form of convergence). Accordingly, via its use of invariants, $\mathcal{I}$-confluence allows users to scope non-determinism while permitting only those states that are acceptable.

Invariant	Operation	$\mathcal{I}$-C?	Proof #
Attribute Equality	Any	Yes	1
Attribute Inequality	Any	Yes	2
Uniqueness	Choose specific value	No	3
Uniqueness	Choose some value	Yes	4
`AUTO_INCREMENT`	Insert	No	5
Foreign Key	Insert	Yes	6
Foreign Key	Delete	No	7
Foreign Key	Cascading Delete	Yes	8
Secondary Indexing	Update	Yes	9
Materialized Views	Update	Yes	10
`>`	Increment [Counter]	Yes	11
`<`	Increment [Counter]	No	12
`>`	Decrement [Counter]	No	13
`<`	Decrement [Counter]	Yes	14
`[NOT] CONTAINS`	Any [Set, List, Map]	Yes	15, 16
`SIZE=`	Mutation [Set, List, Map]	No	17

Table 2: Example SQL (top) and ADT invariant $\mathcal{I}$-confluence along with references to formal proofs in [10, Appendix C].

5. APPLYING INVARIANT CONFLUENCE

As a test for coordination requirements, $\mathcal{I}$-confluence exposes a trade-off between the operations a user wishes to perform and the properties she wishes to guarantee. At one extreme, if a user's transactions do not modify database state, she can guarantee any satisfiable invariant. At the other extreme, with no invariants, a user can safely perform any operations she likes. The space in-between contains a spectrum of interesting and useful combinations.

Until now, we have been largely concerned with formalizing $\mathcal{I}$-confluence for abstract operations; in this section, we begin to leverage this property. We examine a series of practical invariants by considering several features of SQL, ending with abstract data types and revisiting our payroll example along the way. We will apply these results to full applications in Section 6.

In this section, we focus on providing intuition and informal explanations of our $\mathcal{I}$-confluence analysis. Interested readers can find a more formal analysis in [10, Appendix C], including discussion of invariants not presented here. For convenience, we reference specific proofs from [10, Appendix C] inline.

5.1 $\mathcal{I}$-confluence for Relations

We begin by considering several constraints found in SQL.

Equality. As a warm-up, what if an application wants to prevent a particular value from appearing in a database? For example, our payroll application from Section 2 might require that every user have a last name, marking the `LNAME` column with a `NOT NULL` constraint. While not particularly exciting, we can apply $\mathcal{I}$-confluence analysis to insertions and updates of databases with (in-)equality constraints (Claims 1, 2 in [10, Appendix C]). Per-record inequality invariants are $\mathcal{I}$-confluent, which we can show by contradiction: assume two database states S_1 and S_2 are each I-T-reachable under per-record in-equality invariant I_e but that $I_e(S_1 \sqcup S_2)$ is false. Then there must be a $r \in S_1 \sqcup S_2$ that violates I_e (i.e., r has the forbidden value). r must appear in S_1, S_2, or both. But, that would imply that one of S_1 or S_2 is not I-valid under I_e, a contradiction.

Uniqueness. We can also consider common uniqueness invariants (e.g., `PRIMARY KEY` and `UNIQUE` constraints). For example, in our payroll example, we wanted user IDs to be unique. In fact, our earlier discussion in Section 2 already provided a counterexample showing that arbitrary insertion of users is not $\mathcal{I}$-confluent under these invariants: {Stan:5} and {Mary:5} are both I-T-reachable states that can be created by a sequence of insertions (starting at $S_0 = \{\}$), but their merge—{Stan:5, Mary:5}—is not I-valid. Therefore,

uniqueness is not $\mathcal{I}$-confluent for inserts of unique values (Claim 3). However, reads and deletions are both $\mathcal{I}$-confluent under uniqueness invariants: reading and removing items cannot introduce duplicates.

Can the database safely *choose* unique values on behalf of users (e.g., assign a new user an ID)? In this case, we can achieve uniqueness without coordination—as long as we have a notion of replica membership (e.g., server or replica IDs). The difference is subtle ("grant this record this specific, unique ID" versus "grant this record some unique ID"), but, in a system model with membership (as is practical in many contexts), is powerful. If replicas assign unique IDs within their respective portion of the ID namespace, then merging locally valid states will also be globally valid (Claim 4).

Foreign Keys. We can consider more complex invariants, such as foreign key constraints. In our payroll example, each employee belongs to a department, so the application could specify a constraint via a schema declaration to capture this relationship (e.g., `EMP.D_ID FOREIGN KEY REFERENCES DEPT.ID`).

Are foreign key constraints maintainable without coordination? Again, the answer depends on the actions of transactions modifying the data governed by the invariant. Insertions under foreign key constraints *are* $\mathcal{I}$-confluent (Claim 6). To show this, we again attempt to find two I-T-reachable states that, when merged, result in invalid state. Under foreign key constraints, an invalid state will contain a record with a "dangling pointer"—a record missing a corresponding record on the opposite side of the association. If we assume there exists some invalid state $S_1 \sqcup S_2$ containing a record r with an invalid foreign key to record f, but S_1 and S_2 are both valid, then r must appear in S_1, S_2, or both. But, since S_1 and S_2 are both valid, r must have a corresponding foreign key record (f) that "disappeared" during merge. Merge (in the current model) does not remove versions, so this is impossible.

From the perspective of $\mathcal{I}$-confluence analysis, foreign key constraints concern the *visibility* of related updates: if individual database states maintain referential integrity, a non-destructive merge function such as set union cannot cause tuples to "disappear" and compromise the constraint. This also explains why models such as read committed [2] and read atomic [2] isolation as well as causal consistency [9] are also achievable without coordination: simply restricting the visibility of updates in a given transaction's read set does not require coordination between concurrent operations.

Deletions and modifications under foreign key constraints are more challenging. Arbitrary deletion of records is unsafe: a user might be added to a department that was concurrently deleted (Claim 7). However, performing cascading deletions (e.g., SQL `DELETE CASCADE`), where the deletion of a record also deletes *all* matching records on the opposite end of the association, is $\mathcal{I}$-confluent under foreign key constraints (Claim 8). We can generalize this discussion to updates (and cascading updates).

Materialized Views. Applications often pre-compute results to speed query performance via a materialized view [53] (e.g., `UNREAD_CNT` as `SELECT COUNT(*) FROM emails WHERE read_date = NULL`). We can consider a class of invariants that specify that materialized views reflect primary data; when a transaction (or merge invocation) modifies data, any relevant materialized views should be updated as well. This requires installing updates at the same time as the changes to the primary data are installed (a problem related to maintaining foreign key constraints). However, given that a view only reflects primary data, there are no "conflicts." Thus, materialized view maintenance updates are $\mathcal{I}$-confluent (Claim 10).

5.2 $\mathcal{I}$-confluence for Data Types

So far, we have considered databases that store growing sets of immutable versions. We have used this model to analyze several useful constraints, but, in practice, databases do not (often) provide these semantics, leading to a variety of interesting anomalies. For example, if we implement a user's account balance using a "last writer wins" merge policy [50], then performing two concurrent withdrawal transactions might result in a database state reflecting only one transaction (a classic example of the Lost Update anomaly) [2,9]. To avoid variants of these anomalies, many optimistic, coordination-free database designs have proposed the use of *abstract data types* (ADTs), providing merge functions for a variety of uses such as counters, sets, and maps [19,44,50,58] that ensure that all updates are reflected in final database state. For example, a database can represent a simple counter ADT by recording the number of times each transaction performs an `increment` operation on the counter [50].

$\mathcal{I}$-confluence analysis is also applicable to these ADTs and their associated invariants. For example, a row-level "greater-than" (>) threshold invariant is $\mathcal{I}$-confluent for counter `increment` and `assign` ($\leftarrow$) but not `decrement` (Claims 11, 13), while a row-level "less-than" (<) threshold invariant is $\mathcal{I}$-confluent for counter `decrement` and `assign` but not `increment` (Claims 12, 14). This means that, in our payroll example, we can provide coordination-free support for concurrent salary increments but not concurrent salary decrements. ADTs (including lists, sets, and maps) can be combined with standard relational constraints like materialized view maintenance (e.g., the "total salary" row should contain the sum of employee salaries in the `employee` table). This analysis presumes user program explicitly use ADTs, and, as with our generic set-union merge, $\mathcal{I}$-confluence ADT analysis requires a specification of the ADT merge behavior ([10, Appendix C] provides several examples).

5.3 Discussion and Limitations

We have analyzed a number of combinations of invariants and operations (shown in Table 2). These results are by no means comprehensive, but they are expressive for many applications (Section 6). In this section, we discuss lessons from this classification process.

Analysis mechanisms. Here (and in [10, Appendix C]), we manually analyzed particular invariant and operation combinations, demonstrating each to be $\mathcal{I}$-confluent or not. To study actual applications, we can apply these labels via simple static analysis. Specifically, given invariants (e.g., captured via SQL DDL) and transactions (e.g., expressed as stored procedures), we can examine each invariant and each operation within each transaction and identify pairs that we have labeled as $\mathcal{I}$-confluent or non-$\mathcal{I}$-confluent. Any pairs labeled as $\mathcal{I}$-confluent can be marked as safe, while, for soundness (but not completeness), any unrecognized operations or invariants can be flagged as potentially non-$\mathcal{I}$-confluent. Despite its simplicity (both conceptually and in terms of implementation), this technique—coupled with the results of Table 2—is sufficiently powerful to automatically characterize the I-confluence of the applications we consider in Section 6 when expressed in SQL (with support for multi-row aggregates like Invariant 8 in Table 3).

By growing our recognized list of $\mathcal{I}$-confluent pairs on an as-needed basis (via manual analysis of the pair), the above technique has proven useful—due in large part to the common re-use of invariants like foreign key constraints. However, one could use more complex forms of program analysis. For example, one might analyze the $\mathcal{I}$-confluence of *arbitrary* invariants, leaving the task of proving or disproving $\mathcal{I}$-confluence to an automated model checker or SMT solver. While I-confluence—like monotonicity and commutativity (Section 7)—is undecidable for arbitrary programs, others have recently shown this alternative approach (e.g., in commutativity analysis [18,40] and in invariant generation for view serializable transactions [47]) to be fruitful for restricted languages. We view

language design and more automated analysis as an interesting area for more speculative research.

Recency and session support. Our proposed invariants are declarative, but a class of useful semantics—recency, or real-time guarantees on reads and writes—are operational (i.e., they pertain to transaction execution rather than the state(s) of the database). For example, users often wish to read data that is up-to-date as of a given point in time (e.g., "read latest" [20] or linearizable [28] semantics). While traditional isolation models do not directly address these recency guarantees [2], they are often important to programmers. Are these models $\mathcal{I}$-confluent? We can attempt to simulate recency guarantees in $\mathcal{I}$-confluence analysis by logging the result of all reads and any writes with a timestamp and requiring that all logged timestamps respect their recency guarantees (thus treating recency guarantees as invariants over recorded read/write execution traces). However, this is a somewhat pointless exercise: it is well known that recency guarantees are unachievable with transactional availability [9, 21, 28]. Thus, if application reads face these requirements, coordination is required. Indeed, when application "consistency" means "recency," systems cannot circumvent speed-of-light delays.

If users wish to "read their writes" or desire stronger "session" guarantees [45] (e.g., maintaining recency on a per-user or per-session basis), they must maintain affinity or "stickiness" [9] with a given (set of) replicas. These guarantees are also expressible in the $\mathcal{I}$-confluence model and do not require coordination between different users' or sessions' transactions.

Physical and logical replication. We have used the concept of replicas to reason about concurrent transaction execution. However, as previously noted, our use of replicas is simply a formal device and is independent of the actual concurrency control mechanisms at work. Specifically, reasoning about replicas allows us to separate the *analysis* of transactions from their *implementation*: just because a transaction is executed with (or without) coordination does not mean that all query plans or implementations require (or do not require) coordination [9]. However, in deciding on an implementation, there is a range of design decisions yielding a variety of performance trade-offs. Simply because an application is $\mathcal{I}$-confluent does not mean that all implementations will perform equally well. Rather, $\mathcal{I}$-confluence ensures that a coordination-free implementation exists.

Requirements and restrictions. Our techniques are predicated on the ability to correctly and completely specify invariants and inspect user transactions; without such a correctness specification, for arbitrary transaction schedules, serializability is—in a sense—the "optimal" strategy [38]. By casting correctness in terms of admissible application states rather than as a property of read-write schedules, we achieve a more precise statement of coordination overheads. However, as we have noted, this does not obviate the need for coordination in all cases. Finally, when full application invariants are unavailable, individual, high-value transactions may be amenable to optimization via $\mathcal{I}$-confluence coordination analysis.

6. EXPERIENCES WITH COORDINATION

When achievable, coordination-free execution enables scalability limited to that of available hardware. This is powerful: an $\mathcal{I}$-confluent application can scale out without sacrificing correctness, latency, or availability. In Section 5, we saw combinations of invariants and transactions that were $\mathcal{I}$-confluent and others that were not. In this section, we apply these combinations to the workloads of the OLTP-Bench suite [23], with a focus on the TPC-C benchmark. Our focus is on the coordinaton required in order to correctly execute each and the resulting, coordination-related performance costs.

#	Informal Invariant Description	Type	Txns	$\mathcal{I}$-C
1	YTD wh sales = sum(YTD district sales)	MV	P	Yes
2	Per-district order IDs are sequential	S_{ID}+FK	N, D	No
3	New order IDs are sequentially assigned	S_{ID}	N, D	No
4	Per-district, item order count = roll-up	MV	N	Yes
5	Order carrier is set iff order is pending	FK	N, D	Yes
6	Per-order item count = line item roll-up	MV	N	Yes
7	Delivery date set iff carrier ID set	FK	D	Yes
8	YTD wh = sum(historical wh)	MV	D	Yes
9	YTD district = sum(historical district)	MV	P	Yes
10	Customer balance matches expenditures	MV	P, D	Yes
11	Orders reference New-Orders table	FK	N	Yes
12	Per-customer balance = cust. expenditures	MV	P, D	Yes

Table 3: TPC-C Declared "Consistency Conditions" (3.3.2.x) and $\mathcal{I}$-confluence analysis results (Invariant type: MV: materialized view, S_{ID}: sequential ID assignment, FK: foreign key; Transactions: N: New-Order, P: Payment, D: Delivery).

6.1 TPC-C Invariants and Execution

The TPC-C benchmark is the gold standard for database concurrency control [23] both in research and in industry [55], and in recent years has been used as a yardstick for distributed database concurrency control performance [52, 54, 57]. How much coordination does TPC-C actually require a compliant execution?

The TPC-C workload is designed to be representative of a wholesale supplier's transaction processing requirements. The workload has a number of application-level correctness criteria that represent basic business needs (e.g., order IDs must be unique) as formulated by the TPC-C Council and which must be maintained in a compliant run. We can interpret these well-defined "consistency criteria" as invariants and subsequently use $\mathcal{I}$-confluence analysis to determine which transactions require coordination and which do not.

Table 3 summarizes the twelve invariants found in TPC-C as well as their $\mathcal{I}$-confluence analysis results as determined by Table 2. We classify the invariants into three broad categories: materialized view maintenance, foreign key constraint maintenance, and unique ID assignment. As we discussed in Section 5, the first two categories are $\mathcal{I}$-confluent (and therefore maintainable without coordination) because they only regulate the *visibility* of updates to multiple records. Because these (10 of 12) invariants are $\mathcal{I}$-confluent under the workload transactions, there exists some execution strategy that does not use coordination. However, simply because these invariants are $\mathcal{I}$-confluent does not mean that *all* execution strategies will scale well: for example, using locking would *not* be coordination-free.

As one coordination-free execution strategy (which we implement in Section 6.2) that respects the foreign key and materialized view invariants, we can use RAMP transactions, which provide atomically visible transactional updates across servers without relying on coordination for correctness [11]. In brief, RAMP transactions employ limited multi-versioning and metadata to ensure that readers and writers can always proceed concurrently: any client whose reads overlap with another client's writes to the same item(s) can use metadata stored in the items to fetch any "missing" writes from the respective servers. A standard RAMP transaction over data items suffices to enforce foreign key constraints, while a RAMP transaction over commutative counters as described in [11] is sufficient to enforce the TPC-C materialized view constraints.

Two of TPC-C's invariants are not $\mathcal{I}$-confluent with respect to the workload transactions and therefore *do* require coordination. On a per-district basis, order IDs should be assigned sequentially (both uniquely and sequentially, in the New-Order transaction) and orders should be processed sequentially (in the Delivery transaction). If the database is partitioned by warehouse (as is standard [52, 54, 57]), the former is a distributed transaction (by default, 10% of New-Order

transactions span multiple warehouses). The benchmark specification allows the latter to be run asynchronously and in batch mode on a per-warehouse (non-distributed) basis, so we, like others [54, 57], focus on New-Order. Including additional transactions like the read-only Order-Status in the workload mix would increase performance due to the transactions' lack of distributed coordination and (often considerably) smaller read/write footprints.

Avoiding New-Order Coordination. New-Order is not $\mathcal{I}$-confluent with respect to the TPC-C invariants, so we can always fall back to using serializable isolation. However, the per-district ID assignment records (10 per warehouse) would become a point of contention, limiting our throughput to effectively $\frac{100W}{RTT}$ for a W-warehouse TPC-C benchmark with the expected 10% distributed transactions. Others [57] (including us, in prior work [9]) have suggested disregarding consistency criteria 3.3.2.3 and 3.3.2.4, instead opting for unique but non-sequential ID assignment: this allows inconsistency and violates the benchmark compliance criteria.

During a compliant run, New-Order transactions must coordinate. However, as discussed above, only the ID assignment operation is non-I-confluent; the remainder of the operations in the transaction can execute coordination-free. With some effort, we can avoid distributed coordination. A naïve implementation might grab a lock on the appropriate district's "next ID" record, perform (possibly remote) remaining reads and writes, then release the lock at commit time. Instead, as a more efficient solution, New-Order can defer ID assignment until commit time by introducing a layer of indirection. New-Order transactions can generate a temporary, unique, but non-sequential ID (`tmpID`) and perform updates using this ID using a RAMP transaction (which, in turn, handles the foreign key constraints) [11]. Immediately prior to transaction commit, the New-Order transaction can assign a "real" ID by atomically incrementing the current district's"next ID" record (yielding `realID`) and recording the `[tmpID, realID]` mapping in a special ID lookup table. Any read requests for the `ID` column of the Order, New-Order, or Order-Line tables can be safely satisfied (transparently to the end user) by joining with the ID lookup table on `tmpID`. In effect, the New-Order ID assignment can use a nested atomic transaction [44] upon commit, and all coordination between any two transactions is confined to a single server.

6.2 Evaluating TPC-C New-Order

We subsequently implemented the above execution strategy in a distributed database prototype to quantify the overheads associated with coordination in TPC-C New-Order. In brief, the coordination-avoiding query plan scales linearly to over 12.7M transactions per second on 200 servers while substantially outperforming distributed two-phase locking. Our goal here is to demonstrate—beyond the microbenchmarks of Section 2—that safe but judicious use of coordination can have meaningful positive effect on performance.

Implementation and Deployment. We employ a multi-versioned storage manager, with RAMP-Fast transactions for snapshot reads and atomically visible writes/"merge" (providing a variant of regular register semantics, with writes visible to later transactions after commit) [11] and implement the nested atomic transaction for ID assignment as a sub-procedure inside RAMP-Fast's server-side commit procedure (using spinlocks). We implement transactions as stored procedures and fulfill the TPC-C "Isolation Requirements" by using read and write buffering as proposed in [9]. As is common [35, 46, 52, 54], we disregard per-warehouse client limits and "think time" to increase load per warehouse. In all, our base prototype architecture is similar to that of [11]: a JVM-based partitioned, main-memory, mastered database.

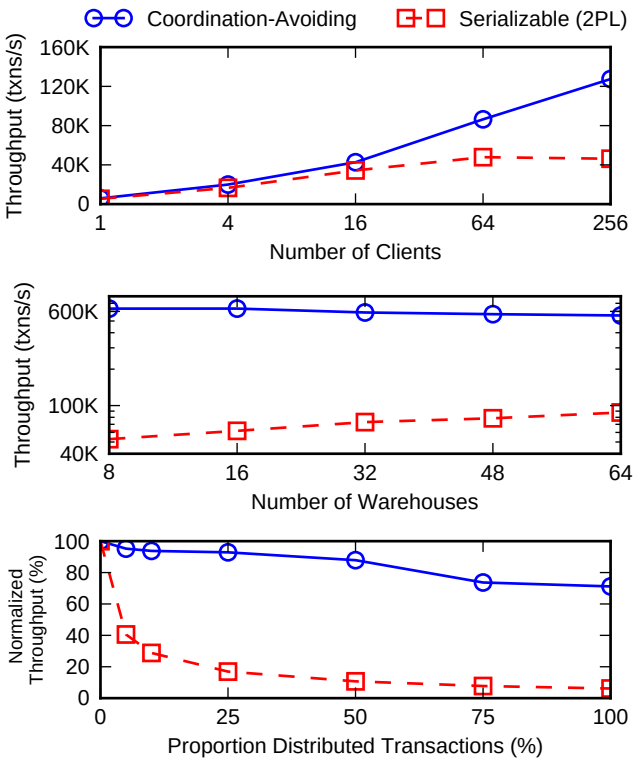


Figure 5: TPC-C New-Order throughput across eight servers.

For an apples-to-apples comparison with a coordination-intensive technique within the same system, we also implemented textbook two-phase locking (2PL) [15], which provides serializability but also requires distributed coordination. We totally order lock requests across servers to avoid deadlock, batching lock requests to each server and piggybacking read and write requests on lock request RPC. As a validation of our implementation, our 2PL prototype achieves per-warehouse (and sometimes aggregate) throughput similar to (and often in excess of) several recent serializable database implementations (of both 2PL and other approaches) [35, 46, 52, 54].

By default, we deploy our prototype on eight EC2 `cr1.8xlarge` instances in the Amazon EC2 `us-west-2` region (with non-co-located clients) with one warehouse per server (recall there are 10 "hot" district ID records per warehouse) and report the average of three 120 second runs.

Basic behavior. Figure 5 shows performance across a variety of configurations, which we detail below. Overall, the coordination-avoiding query plan far outperforms the serializable execution. The coordination-avoiding query plan performs some coordination, but, because coordination points are not distributed (unlike 2PL), physical resources (and not coordination) are the bottleneck.

Varying load. As we increase the number of clients, the coordination-avoiding query plan throughput increases linearly, while 2PL throughput increases to 40K transactions per second, then levels off. As in our microbenchmarks in Section 2, the former utilizes available hardware resources (bottlenecking on CPU cycles at 640K transactions per second), while the latter bottlenecks on logical contention.

Physical resource consumption. To understand the overheads of each component in the coordination-avoiding query plan, we used JVM profiling tools to sample thread execution while running at peak throughput, attributing time spent in functions to relevant modules within the database implementation (where possible):

Code Path	Cycles
Storage Manager (Insert, Update, Read)	45.3%
Stored Procedure Execution	14.4%
RPC and Networking	13.2%
Serialization	12.6%
ID Assignment Synchronization (spinlock contention)	0.19%
Other	14.3%

The coordination-avoiding prototype spends a large portion of execution in the storage manager, performing B-tree modifications and lookups and result set creation, and in RPC/serialization. In contrast to 2PL, the prototype spends less than 0.2% of time coordinating, in the form of waiting for locks in the New-Order ID assignment; the (single-site) assignment is fast (a linearizable integer increment and store, followed by a write and fence instruction on the spinlock), so this should not be surprising. We observed large throughput penalties due to garbage collection (GC) overheads (up to 40%)—an unfortunate cost of our highly compact (several thousand lines of Scala), JVM-based implementation. However, even in this current prototype, physical resources are the bottleneck—not coordination.

Varying contention. We subsequently varied the number of "hot," or contended items by increasing the number of warehouses on each server. Unsurprisingly, 2PL benefits from a decreased contention, rising to over 87K transactions per second with 64 warehouses. In contrast, our coordination-avoiding implementation is largely unaffected (and, at 64 warehouses, is even negatively impacted by increased GC pressure). The coordination-avoiding query plan is effectively agnostic to read/write contention.

Varying distribution. We also varied the percentage of distributed transactions. The coordination-avoiding query plan incurred a 29% overhead moving from no distributed transactions to all distributed transactions due to increased serialization overheads and less efficient batching of RPCs. However, the 2PL implementation decreased in throughput by over 90% (in line with prior results [46,54], albeit exaggerated here due to higher contention) as more requests stalled due to coordination with remote servers.

Scaling out. Finally, we examined our prototype's scalability, again deploying one warehouse per server. As Figure 6 demonstrates, our prototype scales linearly, to over 12.74 million transactions per second on 200 servers (in light of our earlier results, and, for economic reasons, we do not run 2PL at this scale). Per-server throughput is largely constant after 100 servers, at which point our deployment spanned all three `us-west-2` datacenters and experienced slightly degraded per-server performance. While we make use of application semantics, we are unaware of any other compliant multi-server TPC-C implementation that has achieved greater than 500K New-Order transactions per second [35, 46, 52, 54].

Summary. We present these quantitative results as a proof of concept that executing even challenging workloads like TPC-C that contain complex integrity constraints are not necessarily at odds with scalability if implemented in a coordination-avoiding manner. Distributed coordination need not be a bottleneck for all applications, even if conflict serializable execution indicates otherwise. Coordination avoidance ensures that physical resources—and not logical contention—are the system bottleneck whenever possible.

6.3 Analyzing Additional Applications

These results begin to quantify the effects of coordination-avoiding concurrency control. If considering *application-level* invariants, databases only have to pay the price of coordination when necessary. We were surprised that the "current industry standard for evaluating the performance of OLTP systems" [23] was so amenable to

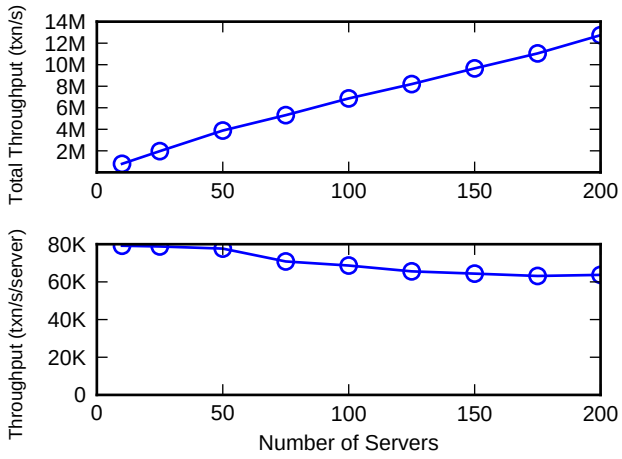


Figure 6: Coordination-avoiding New-Order scalability.

coordination-avoiding execution—at least for compliant execution as defined by the official TPC-C specification.

For greater variety, we also studied the workloads of the recently assembled OLTP-Bench suite [23], performing a similar analysis to that of Section 6.1. We found (and confirmed with an author of [23]) that for nine of fourteen remaining (non-TPC-C) OLTP-Bench applications, the workload transactions did not involve integrity constraints (e.g., did not modify primary key columns), one (`CH-benCHmark`) matched TPC-C, and two specifications implied (but did not explicitly state) a requirement for unique ID assignment (`AuctionMark`'s `new-purchase` order completion, `SEATS`'s `NewReservation` seat booking; achievable like TPC-C order IDs). The remaining two benchmarks, `sibench` and `smallbank` were specifically designed (by an author of this paper) as research benchmarks for serializable isolation. Finally, the three "consistency conditions" required by the newer TPC-E benchmark are a proper subset of the twelve conditions from TPC-C considered here (and are all materialized counters). It is possible (even likely) that these benchmarks are underspecified, but according to official specifications, TPC-C contains the most coordination-intensive invariants among all but two of the OLTP-Bench workloads.

Anecdotally, our conversations and experiences with real-world application programmers and database developers have not identified invariants that are radically different than those we have studied here. A simple thought experiment identifying the invariants required for a social networking site yields a number of invariants but none that are particularly exotic (e.g., username uniqueness, foreign key constraints between updates, privacy settings [11, 20]). Nonetheless, we view the further study of real-world invariants to be a necessary area for future investigation. In the interim, these preliminary results hint at what is possible with coordination-avoidance as well as the costs of coordination if applications are not $\mathcal{I}$-confluent.

7. RELATED WORK

Database system designers have long sought to manage the trade-off between consistency and coordination. As we have discussed, serializability and its many implementations (including lock-based, optimistic, and pre-scheduling mechanisms) [15, 16, 25, 30, 52–54, 57] are sufficient for maintaining application correctness. However, serializability is not always necessary: as discussed in Section 1, serializable databases do not allow certain executions that are correct according to application semantics. This has led to a large class of application-level—or semantic—concurrency control models and

mechanisms that admit greater concurrency. There are several surveys on this topic, such as [29,53], and, in our solution, we integrate many concepts from this literature.

Commutativity. One of the most popular alternatives to serializability is to exploit *commutativity*: if transaction return values (e.g., of reads) and/or final database states are equivalent despite reordering, they can be executed simultaneously [18,41,58]. Commutativity is often sufficient for correctness but is not necessary. For example, if an analyst at a wholesaler creates a report on daily cash flows, any concurrent sale transactions will *not* commute with the report (the results will change depending on whether the sale completes before or after the analyst runs her queries). However, the report creation is $\mathcal{I}$-confluent with respect to, say, the invariant that every sale in the report references a customer from the customers table. [18, 39] provide additional examples of safe non-commutativity.

Monotonicity and Convergence. The CALM Theorem [7] shows that monotone programs exhibit deterministic outcomes despite reordering. CRDT objects [50] similarly ensure convergent outcomes that reflect all updates made to each object. These outcome determinism and convergence guarantees are useful *liveness* properties [49] (e.g., a converged CRDT OR-Set reflects all concurrent additions and removals) but do not prevent users from observing inconsistent data [40], or *safety* (e.g., the CRDT OR-Set does not—by itself—enforce invariants, such as ensuring that no employee belongs to two departments), and are therefore not sufficient to guarantee correctness for all applications. Further understanding the relationship between $\mathcal{I}$-confluence and CALM is an interesting area for further exploration (e.g., as $\mathcal{I}$-confluence adds safety to confluence, is there a natural extension of monotone logic that incorporates $\mathcal{I}$-confluent invariants—say, via an "invariant-scoped" form of monotonicity?).

Use of Invariants. A large number of database designs—including, in restricted forms, many commercial databases today—use various forms of application-supplied invariants, constraints, or other semantic descriptions of valid database states as a specification for application correctness (e.g., [14, 21, 26, 29, 32, 33, 37, 40–42, 47]). We draw inspiration and, in particular, our use of invariants from this prior work. However, we are not aware of related work that discusses when coordination is strictly *required* to enforce a given set of invariants. Moreover, our practical focus here is primarily oriented towards invariants found in SQL and from modern applications.

In this work, we provide a necessary and sufficient condition for safe, coordination-free execution. In contrast with many of the conditions above (esp. commutativity and monotonicity), we explicitly require more information from the application in the form of invariants (Kung and Papadimitriou [38] suggest this is information is *required* for general-purpose non-serializable yet safe execution.) When invariants are unavailable, many of these more conservative approaches may still be applicable. Our use of analysis-as-design-tool is inspired by this literature—in particular, [18].

Coordination costs. In this work, we determine when transactions can run entirely concurrently and without coordination. In contrast, a large number of alternative models (e.g., [4, 8, 26, 33, 37, 42, 43]) assume serializable or linearizable (and therefore coordinated) updates to shared state. These assumptions are standard (but not universal [17]) in the concurrent programming literature [8,49]. (Additionally, unlike much of this literature, we only consider a single set of invariants per database rather than per-operation invariants.) For example, transaction chopping [51] and later application-aware extensions [3, 14] decompose transactions into a set of smaller transactions, providing increased concurrency, but in turn require that individual transactions execute in a serializable (or strict serializable) manner. This reliance on coordinated updates is at odds with our goal of coordination-free execution. However, these alternative techniques are useful in reducing the duration and distribution of coordination once it is established that coordination is required.

Term rewriting. In term rewriting systems, $\mathcal{I}$-confluence guarantees that arbitrary rule application will not violate a given invariant [24], generalizing Church-Rosser confluence [36]. We adapt this concept and effectively treat transactions as rewrite rules, database states as constraint states, and the database merge operator as a special *join* operator (in the term-rewriting sense) defined for all states. Rewriting system concepts—including confluence [4]—have previously been integrated into active database systems [59] (e.g., in triggers, rule processing), but we are not familiar with a concept analogous to $\mathcal{I}$-confluence in the existing database literature.

Coordination-free algorithms and semantics. Our work is influenced by the distributed systems literature, where coordination-free execution across replicas of a given data item has been captured as "availability" [12, 28]. A large class of systems provides availability via "optimistic replication" (i.e., perform operations locally, then replicate) [48]. We—like others [17]—adopt the use of the merge operator to reconcile divergent database states [45] from this literature. Both traditional database systems [2] and more recent proposals [40,41] allow the simultaneous use of "weak" and "strong" isolation; we seek to understand *when* strong mechanisms are needed rather than an optimal implementation of either. Unlike "tentative update" models [27], we do not require programmers to specify compensatory actions (beyond merge, which we expect to typically be generic and/or system-supplied) and do not reverse transaction commit decisions. Compensatory actions could be captured under $\mathcal{I}$-confluence as a specialized merge procedure.

The CAP Theorem [1, 28] recently popularized the tension between strong semantics and coordination and pertains to a specific model (linearizability). The relationship between serializability and coordination requirements has also been well documented in the database literature [21]. We recently classified a range of weaker isolation models by availability, labeling semantics achievable without coordination as "Highly Available Transactions" [9]. Our research here addresses when particular *applications* require coordination.

In our evaluation, we make use of our recent RAMP transaction algorithms [11], which guarantee coordination-free, atomically visible updates. RAMP transactions are an *implementation* of $\mathcal{I}$-confluent semantics (i.e., Read Atomic isolation, used in our implementation for foreign key constraint maintenance). Our focus in this paper is *when* RAMP transactions (and any other coordination-free or $\mathcal{I}$-confluent semantics) are appropriate for applications.

Summary. The $\mathcal{I}$-confluence property is a necessary and sufficient condition for safe, coordination-free execution. Sufficient conditions such as commutativity and monotonicity are useful in reducing coordination overheads but are not always necessary. Here, we explore the fundamental limits of coordination-free execution. To do so, we explicitly consider a model without synchronous communication. This is key to scalability: if, by default, operations must contact a centralized validation service, perform atomic updates to shared state, or otherwise communicate, then scalability will be compromised. Finally, we only consider a single set of invariants for the entire application, reducing programmer overhead without affecting our $\mathcal{I}$-confluence results.

8. CONCLUSION

ACID transactions and associated strong isolation levels dominated the field of database concurrency control for decades, due in

large part to their ease of use and ability to automatically guarantee application correctness criteria. However, this powerful abstraction comes with a hefty cost: concurrent transactions must coordinate in order to prevent read/write conflicts that could compromise equivalence to a serial execution. At large scale and, increasingly, in geo-replicated system deployments, the coordination costs necessarily associated with these implementations produce significant overheads in the form of penalties to throughput, latency, and availability. In light of these trends, we developed a formal framework, called invariant confluence, in which application invariants are used as a basis for determining if and when coordination is strictly necessary to maintain correctness. With this framework, we demonstrated that, in fact, many—but not all—common database invariants and integrity constraints are actually achievable without coordination. By applying these results to a range of actual transactional workloads, we demonstrated an opportunity to avoid coordination in many cases that traditional serializable mechanisms would otherwise coordinate. The order-of-magnitude performance improvements we demonstrated via coordination-avoiding concurrency control strategies provide compelling evidence that invariant-based coordination avoidance is a promising approach to meaningfully scaling future data management systems.

Acknowledgments. The authors would like to thank Peter Alvaro, Neil Conway, Shel Finkelstein, and Josh Rosen for helpful feedback on earlier versions of this work, Dan Crankshaw, Joey Gonzalez, Nick Lanham, and Gene Pang for various engineering contributions, and Yunjing Yu for sharing the Bobtail dataset. This research is supported in part by NSF CISE Expeditions Award CCF-1139158, LBNL Award 7076018, DARPA XData Award FA8750-12-2-0331, the NSF Graduate Research Fellowship (grant DGE-1106400), and gifts from Amazon Web Services, Google, SAP, the Thomas and Stacey Siebel Foundation, Adobe, Apple, Inc., Bosch, C3Energy, Cisco, Cloudera, EMC, Ericsson, Facebook, GameOnTalis, Guavus, HP, Huawei, Intel, Microsoft, NetApp, Pivotal, Splunk, Virdata, VMware, and Yahoo!.

9. REFERENCES

[1] D. J. Abadi. Consistency tradeoffs in modern distributed database system design: CAP is only part of the story. *IEEE Computer*, 45(2):37–42, 2012.

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

[3] D. Agrawal et al. Consistency and orderability: semantics-based correctness criteria for databases. *ACM TODS*, 18(3):460–486, Sept. 1993.

[4] A. Aiken, J. Widom, and J. M. Hellerstein. Behavior of database production rules: Termination, confluence, and observable determinism. In *SIGMOD 1992*.

[5] P. Alvaro, N. Conway, J. M. Hellerstein, and W. Marczak. Consistency analysis in Bloom: a CALM and collected approach. In *CIDR 2011*.

[6] P. Alvaro et al. Consistency without borders. In *SoCC 2013*.

[7] T. J. Ameloot, F. Neven, and J. Van Den Bussche. Relational transducers for declarative networking. *J. ACM*, 60(2):15:1–15:38, May 2013.

[8] H. Attiya, R. Guerraoui, D. Hendler, et al. Laws of order: Expensive synchronization in concurrent algorithms cannot be eliminated. In *POPL 2011*.

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

[10] P. Bailis, A. Fekete, M. J. Franklin, A. Ghodsi, et al. Coordination avoidance in database systems (Extended version). 2014. `arXiv:1402.2237`.

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

[12] P. Bailis and A. Ghodsi. Eventual Consistency today: Limitations, extensions, and beyond. *ACM Queue*, 11(3), 2013.

[13] P. Bailis and K. Kingsbury. The network is reliable: An informal survey of real-world communications failures. *ACM Queue*, 12(7):20, 2014.

[14] A. J. Bernstein and P. M. Lewis. Transaction decomposition using transaction semantics. *Distributed and Parallel Databases*, 4(1):25–47, 1996.

[15] P. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency control and recovery in database systems*. Addison-wesley New York, 1987.

[16] P. A. Bernstein, D. W. Shipman, and J. B. Rothnie, Jr. Concurrency control in a system for distributed databases (SDD-1). *ACM TODS*, 5(1):18–51, Mar. 1980.

[17] S. Burckhardt, D. Leijen, M. Fähndrich, and M. Sagiv. Eventually consistent transactions. In *ESOP*. 2012.

[18] A. T. Clements et al. The scalable commutativity rule: designing scalable software for multicore processors. In *SOSP 2013*.

[19] N. Conway et al. Logic and lattices for distributed programming. In *SoCC 2012*.

[20] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, et al. PNUTS: Yahoo!'s hosted data serving platform. In *VLDB 2008*.

[21] S. Davidson, H. Garcia-Molina, and D. Skeen. Consistency in partitioned networks. *ACM Computing Surveys*, 17(3):341–370, 1985.

[22] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, et al. Dynamo: Amazon's highly available key-value store. In *SOSP 2007*.

[23] D. E. Difallah, A. Pavlo, C. Curino, and P. Cudre-Mauroux. OLTP-Bench: An extensible testbed for benchmarking relational databases. In *VLDB 2014*.

[24] G. Duck, P. Stuckey, and M. Sulzmann. Observable confluence for constraint handling rules. In *ICLP 2007*.

[25] K. P. Eswaran et al. The notions of consistency and predicate locks in a database system. *Commun. ACM*, 19(11):624–633, 1976.

[26] H. Garcia-Molina. Using semantic knowledge for transaction processing in a distributed database. *ACM TODS*, 8(2):186–213, June 1983.

[27] H. Garcia-Molina and K. Salem. Sagas. In *SIGMOD 1987*.

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

[29] P. Godfrey et al. *Logics for databases and information systems*, chapter Integrity constraints: Semantics and applications, pages 265–306. Springer, 1998.

[30] J. Gray. The transaction concept: Virtues and limitations. In *VLDB 1981*.

[31] J. Gray and L. Lamport. Consensus on transaction commit. *ACM TODS*, 31(1):133–160, Mar. 2006.

[32] P. W. Grefen and P. M. Apers. Integrity control in relational database systems–an overview. *Data & Knowledge Engineering*, 10(2):187–223, 1993.

[33] A. Gupta and J. Widom. Local verification of global integrity constraints in distributed databases. In *SIGMOD 1993*, pages 49–58.

[34] R. Johnson, I. Pandis, and A. Ailamaki. Eliminating unscalable communication in transaction processing. *The VLDB Journal*, pages 1–23, 2013.

[35] E. P. Jones, D. J. Abadi, and S. Madden. Low overhead concurrency control for partitioned main memory databases. In *SIGMOD 2010*.

[36] J. W. Klop. *Term rewriting systems*. Stichting Mathematisch Centrum Amsterdam, 1990.

[37] H. K. Korth and G. Speegle. Formal model of correctness without serializabilty. In *SIGMOD 1988*.

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

[39] L. Lamport. Towards a theory of correctness for multi-user database systems. Technical report, CCA, 1976. Described in [3, 49].

[40] C. Li, J. Leitao, A. Clement, N. Preguiça, R. Rodrigues, et al. Automating the choice of consistency levels in replicated systems. In *USENIX ATC 2014*.

[41] C. Li, D. Porto, A. Clement, J. Gehrke, et al. Making geo-replicated systems fast as possible, consistent when necessary. In *OSDI 2012*.

[42] Y. Lin, B. Kemme, R. Jiménez-Peris, et al. Snapshot isolation and integrity constraints in replicated databases. *ACM TODS*, 34(2), July 2009.

[43] S. Lu, A. Bernstein, and P. Lewis. Correct execution of transactions at different isolation levels. *IEEE TKDE*, 16(9), 2004.

[44] N. A. Lynch, M. Merritt, W. Weihl, and A. Fekete. *Atomic Transactions: In Concurrent and Distributed Systems*. Morgan Kaufmann Publishers Inc., 1993.

[45] K. Petersen, M. J. Spreitzer, D. B. Terry, M. M. Theimer, and A. J. Demers. Flexible update propagation for weakly consistent replication. In *SOSP 1997*.

[46] K. Ren, A. Thomson, and D. J. Abadi. Lightweight locking for main memory database systems. VLDB 2013.

[47] S. Roy, L. Kot, et al. Writes that fall in the forest and make no sound: Semantics-based adaptive data consistency, 2014. `arXiv:1403.2307`.

[48] Y. Saito and M. Shapiro. Optimistic replication. *ACM CSUR*, 37(1), Mar. 2005.

[49] F. B. Schneider. *On concurrent programming*. Springer, 1997.

[50] M. Shapiro et al. A comprehensive study of convergent and commutative replicated data types. Technical Report 7506, INRIA, 2011.

[51] D. Shasha, F. Llirbat, E. Simon, and P. Valduriez. Transaction chopping: algorithms and performance studies. *ACM TODS*, 20(3):325–363, Sept. 1995.

[52] 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*.

[53] M. Tamer Özsu and P. Valduriez. *Principles of distributed database systems*. Springer, 2011.

[54] A. Thomson, T. Diamond, S. Weng, K. Ren, P. Shao, and D. Abadi. Calvin: Fast distributed transactions for partitioned database systems. In *SIGMOD 2012*.

[55] TPC Council. TPC Benchmark C revision 5.11, 2010.

[56] I. L. Traiger, J. Gray, C. A. Galtieri, and B. G. Lindsay. Transactions and consistency in distributed database systems. *ACM TODS*, 7(3):323–342, 1982.

[57] S. Tu, W. Zheng, E. Kohler, B. Liskov, and S. Madden. Speedy transactions in multicore in-memory databases. In *SOSP 2013*.

[58] W. Weihl. *Specification and implementation of atomic data types*. PhD thesis, Massachusetts Institute of Technology, 1984.

[59] J. Widom and S. Ceri. *Active database systems: Triggers and rules for advanced database processing*. Morgan Kaufmann, 1996.

[60] Y. Xu et al. Bobtail: avoiding long tails in the cloud. In *NSDI 2013*.