

DILITH: Serializable Commit at the Speed of Consensus, Proved for Clusters of Any Size

Theo Bulut¹

¹ScramDB

theo@scramdb.com

ScramDB

DILITH is the commit protocol of ScramDB: the layer that makes a transaction spanning several replicated shards atomic, durable and serializable while the same nodes serve analytical queries. It answers the client after one round of durable consensus, the floor for a replicated system; it consults no clock and no central service on any correctness path; and any surviving node can finish any transaction a crashed node left behind. Held to one standard in ScramDB's cluster harness beside our implementations of three published protocols and of textbook 2PC with parallel phases, on identical scenarios and seeds, its weighted cost is **88.0% lower than Parallel Commits [1]**, **88.3% lower than textbook 2PC with parallel phases**, **41.7% lower than Sundial [2]** and **83.0% lower than sequential 2PC [3]**. It is **the only one of the five that passes every correctness, liveness and bounded-state gate** both across 3 regions and under crashes, leader kills and partitions, **the only one with a correct history on every run of every suite**, and **the only design, beside 11 published ones, with every property of the comparison satisfied**. Its properties are model checked with those faults as actions the checker takes, and **proved for instances of any size**.



Figure 1 | DILITH's weighted cost against each baseline, pooled over the 5 scored scenarios and their hidden seeds. Lower is better; the cost weights transaction latency, analytical freshness and bytes per commit.

Table 1 | DILITH against the four, at a glance. The cost row is each baseline's weighted cost as a multiple of DILITH's (lower is better, and DILITH is the unit). Green marks a full score or a scenario passed and red a failed gate, with the gate named; each row gets a section of its own further on.

	DILITH	Sundial	Parallel Commits	Textbook 2PC with parallel phases	Sequential 2PC
weighted cost, relative to DILITH's	1.0x	1.7x	8.3x	8.5x	5.9x
passes every gate, seven scenarios side by side	19 of 19	15 of 19	11 of 19	5 of 19	14 of 19
passes every gate, 25 deployments	70 of 75	59 of 75	34 of 75	7 of 75	35 of 75
correct history, 25 deployments	75 of 75	68 of 75	47 of 75	43 of 75	51 of 75
correct history, restore stress	24 of 24	14 of 24	3 of 24	2 of 24	2 of 24
across 3 regions, 27 nodes, partitions	passes every gate	fails: unbounded state	fails: abandoned work	fails: abandoned work, unbounded state	fails: abandoned work, unbounded state
crashes, leader kills, partitions, 9 nodes	passes every gate	fails: abandoned work, correctness, unbounded state	fails: abandoned work	fails: abandoned work, unbounded state	fails: abandoned work, unbounded state

1. What DILITH guarantees, and where the others stop

- **Serializable transactions across shards.** Every multi-shard transaction is atomic and durable, and all the histories the system produces are serializable: equivalent to some serial order [4], on all nodes and at all times.

The guarantee is serializability, not strict serializability: the real-time order of two transactions that touch disjoint shards and never observe each other is not promised. The checker that judges each run replays all committed transactions and compares the state of each node to the replay. **It found no violation in any of DILITH's 126 runs: a correct history in all runs of all suites**, where the best baseline reaches 68 of 75 campaign runs and 14 of 24 restore-stress runs.

- **A correct answer from the leader or from any replica.** A transactional read or an analytical query can be served by any node that holds the data, and the answer is consistent with that serial order: a replica may be behind the leader, never on a different history. Each run closes by reading all keys on each live node and running a transaction from each of them; all of the results have to fit the one serial order.
- **One consensus round, then the answer.** A transaction cannot be acknowledged before its outcome is durable on a majority of every shard it touched, so one parallel round of durable consensus is the floor; DILITH answers the client as soon as that round completes. Under contention the median transaction takes **36 ms with DILITH** against 40 ms, 1,578 ms, 1,580 ms and 1,097 ms for the four baselines in the order of the table above; across three regions, **304 ms** against 320 ms, 15,237 ms, 16,506 ms and 30,000 ms.
- **No clock and no central service on the correctness path.** No timestamp oracle, sequencer, central lock manager or shared storage service sits between a transaction and its commit. Clock skew cannot produce a wrong answer, and there is no service whose outage stops commits. Of the 11 published designs in the matrix at the end of this paper, **none combines this with one durable round, serializability and interactive transactions over replicated shards; DILITH does.**
- **Crash recovery from whichever node survives.** If the node driving a transaction dies, any other node can finish it, and every node reaches the same outcome. No deadline ever turns a committed transaction into an aborted one. Under crashes, leader kills and partitions on 9 nodes, **DILITH abandons 0 transactions and all four baselines fail the abandoned-work gate.**
- **Bounded retention.** The state a shard keeps for the protocol stays under a constant bound, and that bound is a theorem over the specification. The gate that measures memory growth over a run is passed by DILITH across 3 regions under partitions, where Sundial, textbook 2PC with parallel phases and sequential 2PC fail it, and under the nine-node faults; **DILITH is the only one of the five to pass it in both scenarios.** Under the restore stress, where a node crashes every few seconds, no implementation passes it on most runs (Figure 8).
- **Analytics under concurrent commits.** A query that touches many shards reads a single consistent snapshot, and transactions go on committing while it runs. On the analytical scenario with extra read replicas, the query median is **12 ms with DILITH** against 21 ms, 76 ms and 67 ms for the baselines that produced a result, at a freshness lag of 0.0 ms.

Each of these is a gate the evidence in the next sections is held to, a property the model checker checks under injected faults, or a theorem proved in Lean, and usually all three.

2. Results

Five implementations run in ScramDB's cluster harness on identical scenarios and seeds: DILITH and our own implementations of four baselines. Sequential 2PC, in the shape Percolator popularised [5], is the engine's baseline and the reference the scores are normalised to; Parallel Commits follows the published design [1], [6]; Sundial follows the logical-lease design [2]; textbook 2PC with parallel phases is our own implementation of classical two-phase commit [3] over the replicated shards, with both phases fanned out in parallel: prepare at all shards at once, then commit at all of them at once. The score is a weighted cost, lower is better, over 10 terms with weights of 0.45 on transaction latency, 0.35 on analytical latency and freshness and 0.20 on bytes per commit, pooled over seeds held out from development, and one checker judges every run of all five against the same standard.

2.1. Scenario by scenario

Against the engine's own sequential 2PC, **DILITH's cost is below a fifth on four of the 5 scored scenarios and below half on the fifth.** The gap is widest under contention and narrowest on the key-value micro workload, where a transaction only touches one or two shards.

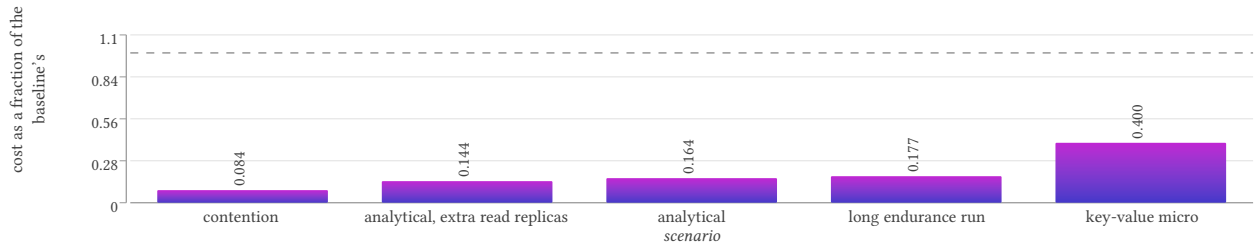


Figure 2 | DILITH's weighted cost as a fraction of sequential 2PC's, per scored scenario. Lower is better, and the dashed rule marks a tie. Pooled over the five, DILITH's cost is 0.170 of the baseline's.

2.2. Against the strongest published baseline

Parallel Commits is the published design closest to DILITH in intent, and the research paper's term-by-term comparison is against it. **Even against Sundial, the closest of the four on the pooled score, DILITH's cost is 41.7% lower.**

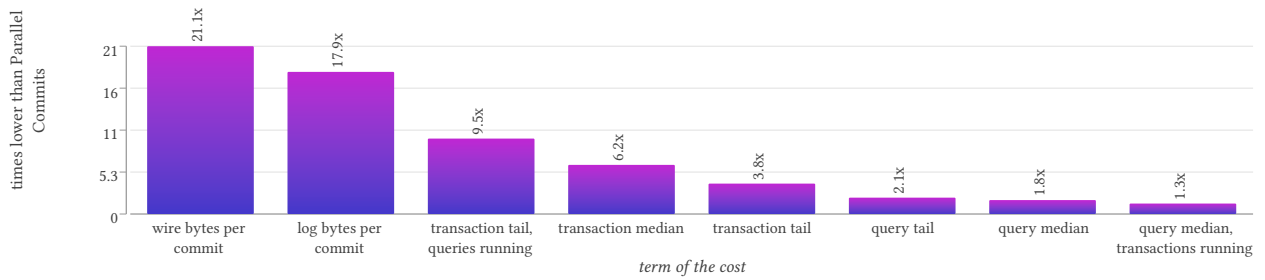


Figure 3 | How much lower each term of the cost is, DILITH against Parallel Commits, on the 8 terms where the two differ; the remaining 2 (freshness lag and bytes per query) sit at their floor for both. Per commit DILITH sends 21.1 times fewer bytes over the wire, and its median transaction latency is 6.2 times lower.

2.3. All five, side by side

Seven scenarios, from a contended rack to three regions and a rack under faults, with the five implementations on the same seeds. **In each panel of the two figures below and on each scenario, DILITH's value is the best of the five or level with it within the harness's tie margin of 2%.**

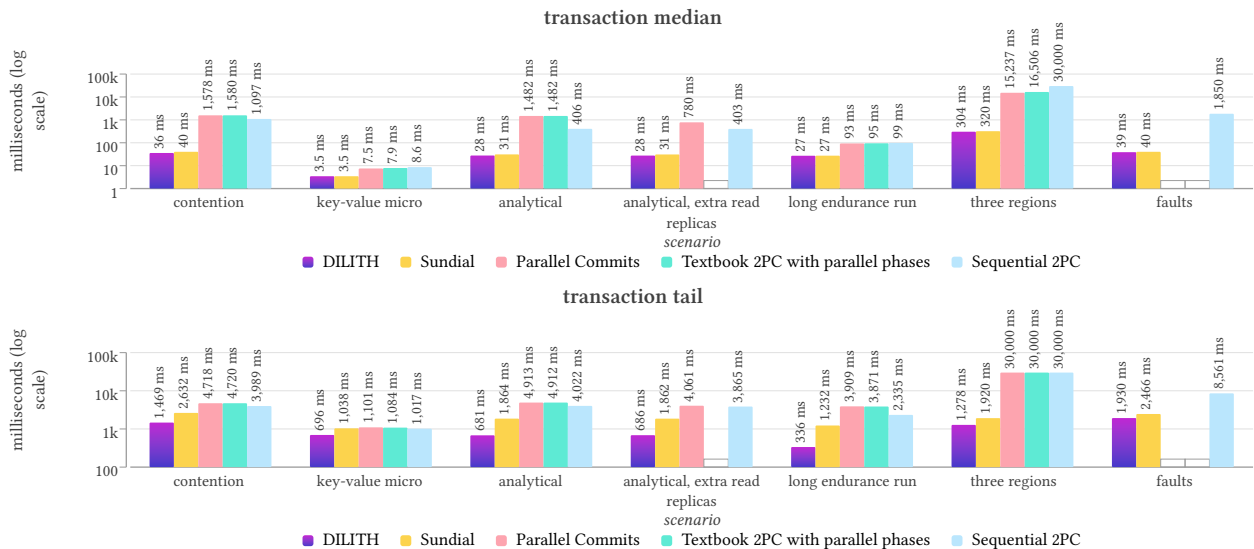


Figure 4 | Transaction latency of the five implementations on the seven side-by-side scenarios, means over the seeds held out from development. Lower is better. A hollow stub marks a protocol that produced no result on that scenario, because its implementation raised or ran past the suite's CPU budget of 2400 s per run.

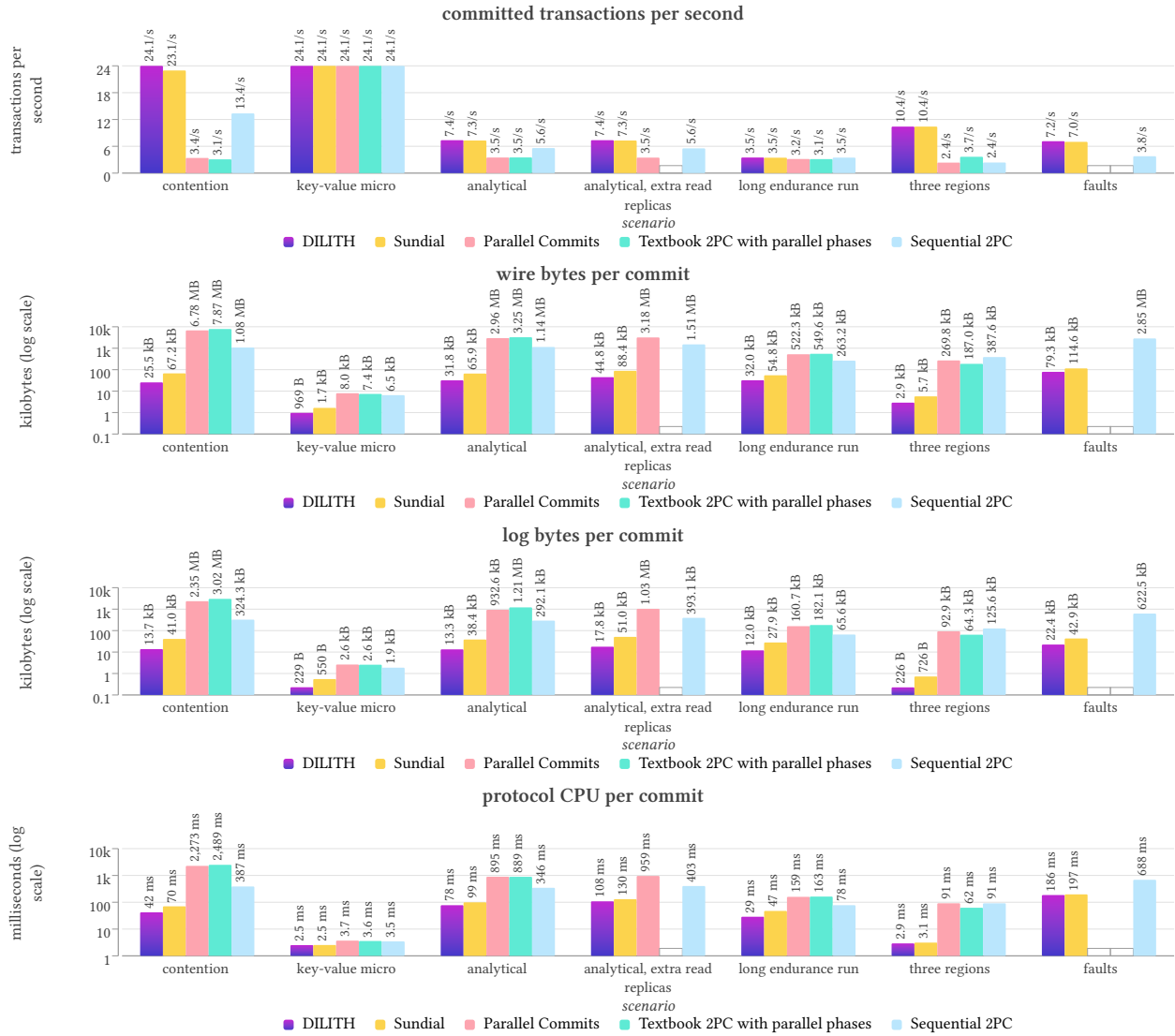


Figure 5 | Throughput first, then what each commit costs in wire bytes, log bytes and CPU, on the same scenarios and seeds. Higher is better on the first panel, lower on the other three; a hollow stub is a protocol with no result there. The one side-by-side measure not shown is the protocol’s own state per node, where Sundial keeps less on each of the 4 scenarios on which it passes every gate (68 B against DILITH’s 1.1 kB on the analytical scenario); the memory growth of the whole state is the bounded-state gate of [Section 2.7](#).

2.4. In absolute terms

Table 2 | The raw figures for DILITH, and in grey those of sequential 2PC on the same engine. Latencies are milliseconds from a transaction’s arrival to its answer, every statement and retry included; bytes include replication. No transaction was abandoned and none was left in doubt in any run of the five scored, fault-free scenarios.

Scenario	median (ms)	slowest 5% (ms)	wire bytes / commit	log bytes / commit
contention	36 (1097)	1469 (3989)	25.5 kB (1.08 MB)	13.7 kB (324.3 kB)
key-value micro	3.5 (8.6)	696 (1017)	969 B (6.5 kB)	229 B (1.9 kB)
analytical	28 (406)	681 (4022)	31.8 kB (1.14 MB)	13.3 kB (292.1 kB)
analytical, extra read replicas	28 (403)	686 (3865)	44.8 kB (1.51 MB)	17.8 kB (393.1 kB)
long endurance run	27 (99)	336 (2335)	32.0 kB (263.2 kB)	12.0 kB (65.6 kB)

2.5. Across regions

Once the cluster spans regions, every consensus round costs a continental round trip, and a commit protocol’s round count shows. Of the campaign’s 25 deployments, the 5 shown below spread the cluster over 2, 3, 4 or 5 regions and run up to 45 nodes. **On each of them DILITH has the lowest median transaction latency of the five.** Sundial keeps pace on latency here, and under region partitions it fails: unbounded state ([Section 2.7](#)); **the other three are at least 8 times slower on each cross-region deployment.**

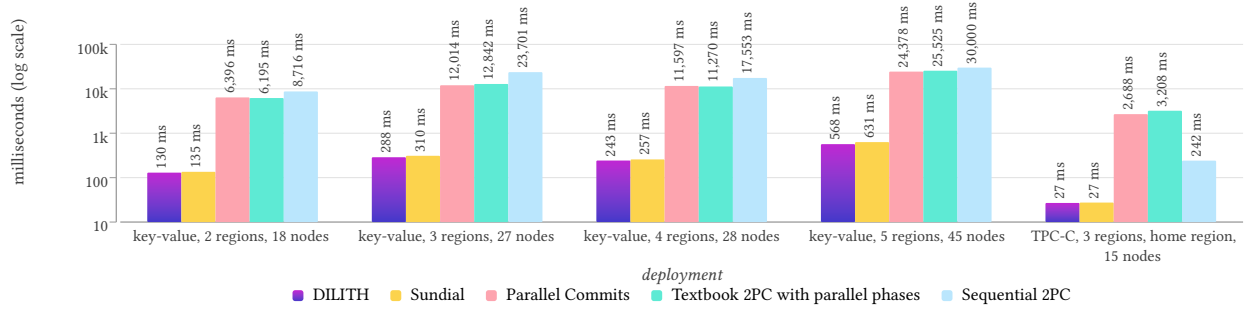


Figure 6 | Median transaction latency on the cross-region deployments of the campaign, means over 3 seeds on a logarithmic axis, with the same offered load for every protocol. Higher latency is worse. A median at the top of the axis is the client’s deadline: most of that protocol’s transactions were abandoned there.

2.6. Under load

The saturated deployments offer more than the cluster can commit, so throughput measures what a protocol does at the ceiling. **In all of them DILITH commits the most or is level with Sundial within the tie margin, and on TPC-C, 24 nodes, saturated it is the only implementation with a result.**

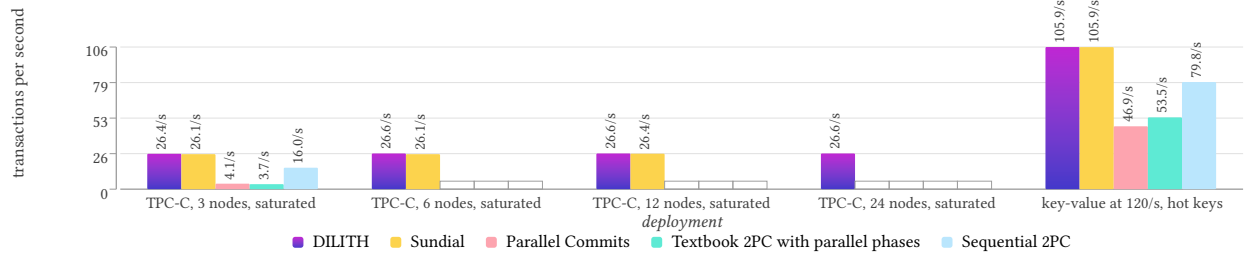


Figure 7 | Committed transactions per second on the saturated deployments, means over 3 seeds; the offered load is 30 and 120 per second. Higher is better. A hollow stub marks a protocol that produced no result there, because its implementation raised or ran past the campaign’s CPU budget of 2400 s per run.

Where a baseline beats DILITH by more than the tie margin anywhere in the campaign, it is Sundial, on 3 measurements: transaction median on *key-value at 60/s, hot keys*, 3.4 ms against DILITH’s 3.5 ms; transaction median on *key-value at 120/s, hot keys*, 3.5 ms against DILITH’s 3.7 ms; and transaction tail on *endurance, key-value, 6 nodes*, 1,222 ms against DILITH’s 1,407 ms.

2.7. Under faults

Two scenarios inject faults while transactions run: a 27-node cluster across 3 regions with region partitions, and a 9-node cluster where nodes crash, leaders are killed and the network is partitioned. A protocol passes a scenario only if all its histories are correct, no client gave up on a transaction the cluster could not finish, and the protocol’s memory stayed bounded over the run.

Table 3 | Which gates each implementation passed under faults, over 2 seeds per scenario. A correctness failure means the checker found a read, a range or a final state that no serial order explains; abandoned work means clients gave up on transactions the cluster could not finish; unbounded state means the protocol’s memory grew over the run.

Implementation	Across 3 regions, 27 nodes	Crashes, leader kills and partitions, 9 nodes
DILITH	passes every gate	passes every gate
Sundial	fails: unbounded state	fails: abandoned work, correctness, unbounded state
Textbook 2PC with parallel phases	fails: abandoned work, unbounded state	fails: abandoned work, unbounded state
Parallel Commits	fails: abandoned work	fails: abandoned work
Sequential 2PC	fails: abandoned work, unbounded state	fails: abandoned work, unbounded state

2.8. Every suite, every gate

The same standard applies to all suites: 19 runs over seven scenarios side by side, 12 runs per workload of a restore stress that compacts logs aggressively and crashes nodes continually, and 75 runs over the 25 deployments of the

campaign, from 3-node racks to 45 nodes across 5 regions, with TPC-C, key-value, bank and saturation workloads, endurance runs, and mixed-fault racks with skewed clocks.

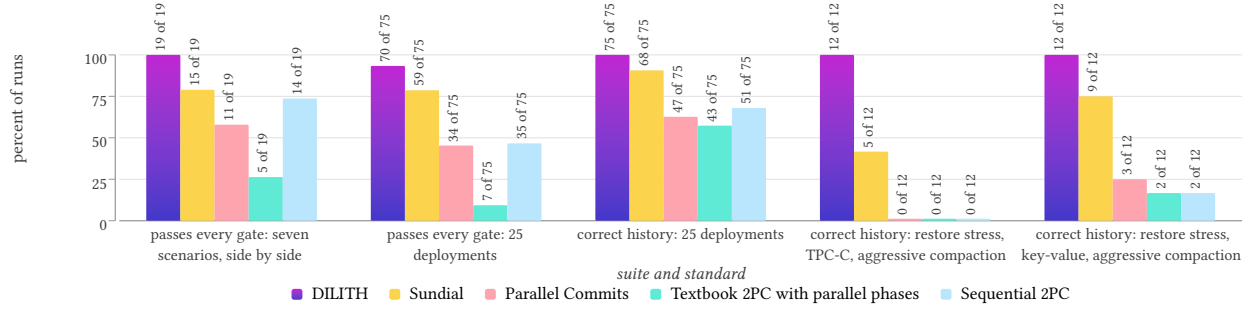


Figure 8 | How many runs met the standard in each suite, as a share on the axis and as a count on each column. DILITH passes every gate on 19 of 19 side-by-side runs and 70 of 75 campaign runs, and keeps a correct history on each run of each suite. The 5 campaign runs it misses are all on *every fault, mixed rack, 9 nodes* and *every fault, mixed rack, 18 nodes*, where it failed the bounded-state gate (5 runs) and the abandoned-work gate (2 runs). The restore stress is beyond every implementation’s liveness and bounded-state gates on most of its 24 runs; what separates them is the history, which **DILITH keeps correct on all 24**, against 14, 3, 2 and 2 for the baselines, three of which produced no result on most runs.

3. Against the published designs

Table 4 | How DILITH and 11 published designs compare on 6 properties. Green is a property the design has, red one it lacks, amber a partial one, grey one that does not apply or is not verifiable from the published description. **Of the 12 rows, only DILITH’s is green from end to end.** The six properties are the ones this paper compares; designs that replace the replication layer, TAPIR [7] and Janus [8], are outside the comparison, since DILITH’s constraint is to leave that layer unchanged.

Design	year	one durable round	no clock on a correctness path	no timestamp service	serializable	interactive transactions	replicated shards
Percolator-style 2PC [5]	2010	no	partial	no	no	yes	partial
Spanner [9]	2012	no	no	yes	yes	yes	yes
Paxos Commit [10]	2006	no	yes	yes	n/a	n/a	no
Granola [11]	2012	yes	no	yes	yes	no	yes
Clock-SI [12]	2013	no	no	yes	no	yes	no
MDCC [13]	2013	yes	yes	yes	no	yes	yes
FaRM [14]	2015	no	no	yes	yes	yes	yes
Sundial [2]	2018	no	yes	yes	yes	yes	no
Parallel Commits [1]	2019	yes	no	yes	yes	yes	yes
TiDB async commit, TiFlash [15], [16]	2020	yes	no	no	no	yes	yes
Cornus [17]	2022	yes	yes	yes	n/a	n/a	partial
DILITH	2026	yes	yes	yes	yes	yes	yes

4. Assurance

The safety properties are stated and checked independently of the experiments.

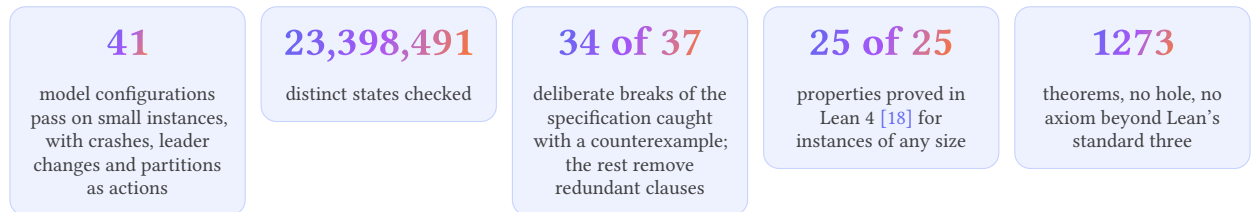


Figure 9 | The formal record behind the guarantees. The specification is transcribed from the implementation and model checked; the theorems end in serializability of every reachable state.

The protocol’s specification is written in TLA+ [19], transcribed from the implementation, and model checked on small instances with node crashes and restarts, leader changes and region partitions as actions the checker takes. The checker was then handed deliberately broken variants of the specification and rejected 34 of 37 with a counterexample; each of the rest removed a clause the remaining rules already imply. Proofs in Lean 4 [18] over that specification cover 25 of the 25 properties, and they hold for instances of any size. The schedules that break

commit protocols in practice, a lost outcome, a late copy of a message, a client left in doubt, a replica that lags, a coordinator that dies mid-flight, are each covered by a regression test, a model configuration, a broken variant and a theorem.

5. How the evidence is produced

Everything in this paper comes from our own experiments, and every figure and every number is generated from their recorded results; the paper is rebuilt from those records whenever they change, and results produced by any other version of the harness are rejected. The harness is ScramDB’s own: it drives every node, disk flush, consensus round and network hop of a cluster, calibrated against the engine, and its checker replays all committed transactions and compares each node to the replay. The four baselines are our own implementations of the published designs, run in the same harness on the same seeds and held to the same standard as DILITH. The experiments behind this paper comprise 630 runs over four suites (side by side, seven scenarios; restore stress, a crash every 4 seconds; whole-state endurance runs; campaign, twenty-five deployments), 10.4 hours of cluster time and 158.8 hours of CPU. The whole-state endurance suite has no figure here: on its long runs no implementation’s memory is flat over the last quarter, DILITH included; the research paper gives those ratios, the full method and every result.

6. Availability

DILITH runs on ScramDB nodes alone, needs no timestamp service, sequencer or change to the replication layer, and serves transactional, analytical and vector workloads from the same nodes. Questions about DILITH go to ScramDB.

References

- [1] Cockroach Labs, “Parallel Commits: An Atomic Commit Protocol for Globally Distributed Transactions.” [Online]. Available: <https://www.cockroachlabs.com/blog/parallel-commits/>
- [2] X. Yu, Y. Xia, A. Pavlo, D. Sanchez, L. Rudolph, and S. Devadas, “Sundial: Harmonizing Concurrency Control and Caching in a Distributed OLTP Database Management System,” *Proceedings of the VLDB Endowment*, vol. 11, no. 10, pp. 1289–1302, 2018, doi: [10.14778/3231751.3231763](https://doi.org/10.14778/3231751.3231763).
- [3] J. Gray, “Notes on Data Base Operating Systems,” in *Operating Systems: An Advanced Course*, vol. 60, in Lecture Notes in Computer Science, vol. 60., Springer, 1978, pp. 393–481. doi: [10.1007/3-540-08755-9_9](https://doi.org/10.1007/3-540-08755-9_9).
- [4] C. H. Papadimitriou, “The Serializability of Concurrent Database Updates,” *Journal of the ACM*, vol. 26, no. 4, pp. 631–653, 1979, doi: [10.1145/322154.322158](https://doi.org/10.1145/322154.322158).
- [5] D. Peng and F. Dabek, “Large-scale Incremental Processing Using Distributed Transactions and Notifications,” in *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2010, pp. 251–264.
- [6] R. Taft et al., “CockroachDB: The Resilient Geo-Distributed SQL Database,” in *ACM SIGMOD International Conference on Management of Data*, 2020, pp. 1493–1509. doi: [10.1145/3318464.3386134](https://doi.org/10.1145/3318464.3386134).
- [7] I. Zhang, N. K. Sharma, A. Szekeres, A. Krishnamurthy, and D. R. K. Ports, “Building Consistent Transactions with Inconsistent Replication,” in *25th ACM Symposium on Operating Systems Principles (SOSP)*, 2015, pp. 263–278. doi: [10.1145/2815400.2815404](https://doi.org/10.1145/2815400.2815404).
- [8] S. Mu, L. Nelson, W. Lloyd, and J. Li, “Consolidating Concurrency Control and Consensus for Commits under Conflicts,” in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2016, pp. 517–532.
- [9] J. C. Corbett et al., “Spanner: Google’s Globally-Distributed Database,” in *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2012, pp. 251–264.
- [10] J. Gray and L. Lamport, “Consensus on Transaction Commit,” *ACM Transactions on Database Systems*, vol. 31, no. 1, pp. 133–160, 2006, doi: [10.1145/1132863.1132867](https://doi.org/10.1145/1132863.1132867).
- [11] J. Cowling and B. Liskov, “Granola: Low-Overhead Distributed Transaction Coordination,” in *USENIX Annual Technical Conference (ATC)*, 2012, pp. 223–235.
- [12] J. Du, S. Elnikety, and W. Zwaenepoel, “Clock-SI: Snapshot Isolation for Partitioned Data Stores Using Loosely Synchronized Clocks,” in *IEEE 32nd International Symposium on Reliable Distributed Systems (SRDS)*, 2013, pp. 173–184. doi: [10.1109/SRDS.2013.26](https://doi.org/10.1109/SRDS.2013.26).
- [13] T. Kraska, G. Pang, M. J. Franklin, S. Madden, and A. Fekete, “MDCC: Multi-Data Center Consistency,” in *8th ACM European Conference on Computer Systems (EuroSys)*, 2013, pp. 113–126. doi: [10.1145/2465351.2465363](https://doi.org/10.1145/2465351.2465363).
- [14] A. Dragojević et al., “No Compromises: Distributed Transactions with Consistency, Availability, and Performance,” in *25th ACM Symposium on Operating Systems Principles (SOSP)*, 2015, pp. 54–70. doi: [10.1145/2815400.2815425](https://doi.org/10.1145/2815400.2815425).
- [15] TiKV Transaction SIG, “Async Commit Design Spec.” [Online]. Available: <https://github.com/tikv/sig-transaction/blob/master/design/async-commit/spec.md>
- [16] D. Huang et al., “TiDB: A Raft-based HTAP Database,” *Proceedings of the VLDB Endowment*, vol. 13, no. 12, pp. 3072–3084, 2020, doi: [10.14778/3415478.3415535](https://doi.org/10.14778/3415478.3415535).
- [17] Z. Guo et al., “Cornus: Atomic Commit for a Cloud DBMS with Storage Disaggregation,” *Proceedings of the VLDB Endowment*, vol. 16, no. 2, pp. 379–392, 2022, doi: [10.14778/3565816.3565837](https://doi.org/10.14778/3565816.3565837).
- [18] L. de Moura and S. Ullrich, “The Lean 4 Theorem Prover and Programming Language,” in *28th International Conference on Automated Deduction (CADE)*, in Lecture Notes in Artificial Intelligence, vol. 12699. 2021, pp. 625–635. doi: [10.1007/978-3-030-79876-5_37](https://doi.org/10.1007/978-3-030-79876-5_37).
- [19] L. Lamport, *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, 2002.