



WHITE PAPER · DATA STORAGE ENGINES

# Distributed SQL Storage

Consensus, sharding and PostgreSQL compatibility

---

White Paper 6 · Data Storage Engines series · Version 1.0 · 26 October 2026

Bassirah Technology — Innovation & R&D

# Distributed SQL Storage: Consensus, Sharding and PostgreSQL Compatibility

White Paper 6 · Data Storage Engines series · Bassirah Technology — Innovation & R&D · 26 October 2026

## Document control

| Field   | Value                                                                                                                                             |
|---------|---------------------------------------------------------------------------------------------------------------------------------------------------|
| Series  | Data Storage Engines, paper 6 of 6                                                                                                                |
| Author  | Bassirah Technology — Innovation & R&D                                                                                                            |
| Version | 1.0 · 26 October 2026                                                                                                                             |
| Scope   | A technology survey. It describes published research and publicly documented systems; it does not report Bassirah benchmarks or customer results. |

## Contents

|                                                                           |   |
|---------------------------------------------------------------------------|---|
| 1. Abstract.....                                                          | 4 |
| 2. Introduction .....                                                     | 4 |
| 2.1 What distributed SQL adds to a single PostgreSQL primary .....        | 4 |
| 2.2 Scope.....                                                            | 4 |
| 2.3 How this paper builds on papers 1 to 5.....                           | 4 |
| 3. Architecture building blocks.....                                      | 5 |
| 3.1 Three layers .....                                                    | 5 |
| 3.2 Shared-nothing distributed SQL.....                                   | 5 |
| 3.3 Disaggregated compute and storage .....                               | 5 |
| 3.4 Unbundled, sharded and serverless designs .....                       | 5 |
| 4. Replication and consensus.....                                         | 6 |
| 4.1 A consensus group per shard .....                                     | 6 |
| 4.2 Leaders, leases and consistent reads.....                             | 6 |
| 4.3 Follower reads and read replicas .....                                | 6 |
| 4.4 Asynchronous replication between clusters.....                        | 7 |
| 5. Sharding and data placement.....                                       | 7 |
| 5.1 Hash and range sharding.....                                          | 7 |
| 5.2 Splitting, rebalancing and hot spots.....                             | 7 |
| 5.3 Multi-region placement and geo-partitioning .....                     | 7 |
| 6. Distributed transactions and time.....                                 | 8 |
| 6.1 Commit across shards.....                                             | 8 |
| 6.2 Clocks: TrueTime, hybrid logical clocks and bounded uncertainty ..... | 8 |
| 6.3 Isolation levels .....                                                | 8 |
| 6.4 Deterministic and geo-optimized alternatives .....                    | 9 |
| 7. Storage engines underneath .....                                       | 9 |

|                                                                      |    |
|----------------------------------------------------------------------|----|
| 7.1 LSM-based key-value stores per node .....                        | 9  |
| 7.2 Other storage designs .....                                      | 9  |
| 7.3 Tiered and object storage for cold data .....                    | 9  |
| 7.4 Analytical replicas and HTAP .....                               | 10 |
| 8. PostgreSQL compatibility .....                                    | 10 |
| 8.1 Three approaches .....                                           | 10 |
| 8.2 What typically differs .....                                     | 10 |
| 8.3 Migration considerations .....                                   | 11 |
| 9. Comparative analysis .....                                        | 11 |
| 10. Open challenges and future opportunities .....                   | 12 |
| 10.1 Cross-region latency versus consistency .....                   | 12 |
| 10.2 Tail latency and hot spots under skew .....                     | 13 |
| 10.3 Elasticity and serverless economics .....                       | 13 |
| 10.4 Compatibility gaps and migration effort .....                   | 13 |
| 10.5 HTAP freshness in distributed SQL .....                         | 13 |
| 10.6 Operational complexity and observability .....                  | 14 |
| 10.7 Clock and failure-model assumptions .....                       | 14 |
| 10.8 The cost of strong consistency for AI and agent workloads ..... | 14 |
| 11. What this means for platform decisions .....                     | 14 |
| References .....                                                     | 15 |

## 1. Abstract

A single PostgreSQL primary is a strong default for transactional workloads, but it has one writer, one failure domain for writes, and a capacity ceiling set by one machine. Distributed SQL databases remove these limits by splitting tables into shards, replicating each shard with a consensus protocol, and running transactions across shards, while keeping a SQL interface and, increasingly, PostgreSQL compatibility. This paper examines how they do so. It describes the three layers these systems share (SQL, transactions and distributed key-value storage) and contrasts shared-nothing designs with disaggregated compute and storage. It then covers consensus replication, leader leases and follower reads; hash and range sharding, automatic splitting and multi-region placement; distributed commit, clocks and isolation levels; the storage engines underneath; and the three main approaches to PostgreSQL compatibility. It closes with a qualitative comparison, open challenges framed as research directions and opportunities, and notes for platform decisions that also conclude the series.

## 2. Introduction

### 2.1 What distributed SQL adds to a single PostgreSQL primary

A single-node relational database scales up rather than out. Read replicas add read capacity and standbys restore service after a failure, but writes still pass through one node, and asynchronous standbys can lose recently committed transactions when they are promoted. Distributed SQL systems change three things. They partition data into shards that different nodes serve, so write capacity and storage grow with the cluster. They replicate each shard synchronously with a consensus protocol, so the loss of a node, zone or region need not lose committed data or stop service. And they provide transactions that span shards, so applications are not forced to shard data and coordinate commits themselves.

Google's Spanner established the pattern: data sharded across many Paxos groups, synchronous replication across data centers, and externally consistent transactions built on a clock API that exposes its own uncertainty [1]. Spanner later gained a full SQL query layer with distributed execution that tolerates resharding [2]. CockroachDB [3], TiDB [4] and YugabyteDB [5] apply related ideas with the Raft consensus protocol on commodity infrastructure.

### 2.2 Scope

This paper covers the storage, replication, sharding, transaction and compatibility mechanisms of distributed SQL databases, with disaggregated cloud databases as a contrasting design. It concentrates on peer-reviewed work from 2021 to 2026 and on official documentation, and cites earlier work where it introduced an idea still in use. Statements about products reflect documentation accessed on 26 October 2026. It does not report benchmark figures, latency figures or recovery objectives. Where a paper reports results, we describe them qualitatively and attribute them to the authors.

### 2.3 How this paper builds on papers 1 to 5

Paper 1 classified storage engines by their dominant trade-off. Distributed SQL is not a separate class in that taxonomy; it is a distribution layer built over storage engines, most often Class A LSM-trees. Paper 2 described those engines (RocksDB, Pebble, the RocksDB fork in YugabyteDB's DocDB, and TiKV's use of RocksDB), and this paper refers to it rather than

repeating compaction and tuning detail. Paper 3's columnar formats reappear in the analytical replicas of HTAP systems (Section 7.4). Paper 4's vector workloads and paper 5's object-storage tables reappear in the open challenges.

## 3. Architecture building blocks

### 3.1 Three layers

Most distributed SQL systems separate a SQL layer, a transaction layer and a distributed key-value storage layer. The SQL layer is usually stateless, so any node can accept a connection; the transaction layer assigns timestamps, detects conflicts and makes commits atomic; and the storage layer maps rows and index entries to keys, splits the key space into shards and replicates each shard. CockroachDB describes its architecture as such a layered stack over a replicated, transactional key-value store [3]. TiDB places a stateless SQL server over TiKV, a transactional key-value store organized into Regions, with a Placement Driver that holds cluster metadata, schedules data placement and issues timestamps [4], [6]. YugabyteDB divides itself into a query layer and DocDB, a document store built on a customized RocksDB, with a YB-Master service for the catalog and orchestration [5].

### 3.2 Shared-nothing distributed SQL

In shared-nothing designs, each node stores and serves some shards, and no storage is shared between nodes. Scaling out adds nodes that take over shards, and fault tolerance comes from replicating each shard to several nodes. Spanner, CockroachDB, TiDB, YugabyteDB, OceanBase and PolarDB-X follow this model in different forms [1], [3], [4], [5], [7], [8].

### 3.3 Disaggregated compute and storage

A second family keeps a single writer but separates the database engine from a shared, replicated storage service. Amazon Aurora pushes redo processing into a multi-tenant storage service and sends only log records over the network; its authors argue that the network, rather than compute or storage, had become the central constraint [9]. Socrates, the architecture behind Azure SQL Database Hyperscale, separates compute, a log service, page servers and durable storage, so that durability and availability are handled by different tiers [10], [11]. Taurus separates compute and storage in a similar way and contributes its own replication and recovery algorithms [12]. Neon applies the idea to PostgreSQL: compute nodes run PostgreSQL, safekeepers make the write-ahead log durable through Paxos, pageservers reconstruct page versions on demand, and object storage holds long-term history [13]. PolarDB Serverless extends disaggregation to a remote memory pool as well as storage [14].

These systems scale storage and read replicas independently, but writes remain bounded by one primary. Wang and Zhang survey the research challenges of such disaggregated systems [15], and Pang and Wang use a PostgreSQL-based prototype to measure the effect of individual design principles, such as log-as-the-database and log replay strategies [16].

### 3.4 Unbundled, sharded and serverless designs

The two families are converging. FoundationDB unbundles a transactional key-value store into separately scaled roles (a sequencer, proxies, conflict resolvers, log servers and storage servers) and leaves SQL to layers built on top [17], [18]. Amazon Aurora DSQL separates query processing, commit adjudication, a replicated journal and storage into independent services;

query processors run a customized PostgreSQL engine, and storage is sharded by key range [19]. Aurora PostgreSQL Limitless Database takes a third route: routers accept connections and coordinate distributed transactions, while shards are Aurora PostgreSQL instances that each hold part of the hash-sharded tables [20]. Citus distributes PostgreSQL itself, with a coordinator routing queries to worker nodes that hold shards [21], [22].

## 4. Replication and consensus

### 4.1 A consensus group per shard

Distributed SQL systems replicate each shard with its own consensus group rather than replicating whole nodes. Chandra et al. describe how much engineering a production Paxos implementation at Google required beyond the algorithm itself [23]. Raft decomposes consensus into leader election, log replication and safety, with an explicit mechanism for membership changes, and was designed to be easier to implement correctly [24]. Spanner runs a Paxos state machine per tablet [1]; CockroachDB, TiKV and YugabyteDB run one Raft group per range, Region or tablet [25], [4], [5].

With a replication factor of 3, a group tolerates the loss of one replica; with 5, it tolerates two. In general, surviving the failure of  $f$  fault domains requires at least  $2f + 1$  of them, so surviving a region failure with synchronous replication requires three regions [26], [25].

### 4.2 Leaders, leases and consistent reads

A leader orders writes for its group. Serving a consistent read from the leader alone is safe only if the leader knows it has not been replaced, which requires either a quorum check per read or a time-based lease. CockroachDB designates one replica per range as leaseholder, which serves reads without a consensus round and proposes writes; its current leader-lease design keeps the leaseholder and the Raft leader on the same replica and bases lease validity on store-level liveness support from a quorum, removing an earlier dependency on a single liveness range [25]. YugabyteDB's Raft implementation adds leader leases and serves strongly consistent reads and writes from the tablet leader [5]. Davis, Demirbas and Deng observe that prior lease protocols were vaguely specified and that many Raft systems implement them incorrectly or not at all; their LeaseGuard protocol, specified in TLA+, builds on Raft's election guarantees and improves availability during failover [27].

### 4.3 Follower reads and read replicas

Reading from a nearby follower avoids a trip to a distant leader, at some cost in freshness or coordination. CockroachDB advances a closed timestamp per range: once it passes a time, no new writes can land at or below it, so any replica can serve reads at or below that time. Exact-staleness reads choose a fixed past timestamp, and bounded-staleness reads let the system pick the freshest locally servable time [28]. CockroachDB's multi-region work adds global tables, whose writes take longer so that strongly consistent reads can be served locally in every region [29], [28]. Spanner distinguishes strong reads from stale reads at an earlier timestamp, which do not block writes [30]. Non-voting replicas in CockroachDB and read replicas in YugabyteDB receive the replication stream without joining the write quorum; YugabyteDB describes the data they serve as timeline-consistent, possibly slightly stale [25], [26].

## 4.4 Asynchronous replication between clusters

When three regions are unavailable or cross-region write latency is unacceptable, two clusters can be linked asynchronously. YugabyteDB's xCluster replicates write-ahead log records from a source universe to a target universe; its transactional mode keeps the target read-only and exposes source transactions atomically and in commit order, while unplanned failover can lose transactions not yet replicated [26]. Hyperscale offers geo-secondary replicas with separate storage for disaster recovery [11]. GaussDB-Global, from the research literature, uses asynchronous replication with synchronized clocks and supports reads on asynchronous replicas with tunable freshness [31]. In each case, lower write latency and fewer sites are bought with a window of possible data loss and stale reads that teams must size and test.

# 5. Sharding and data placement

## 5.1 Hash and range sharding

Hash sharding spreads keys evenly and suits point lookups but scatters ranges; range sharding keeps keys in order for scans but concentrates sequential inserts on the last shard. YugabyteDB supports both, choosing per table or index through the primary key definition [5].

CockroachDB, TiKV and Spanner split the ordered key space into ranges [3], [4], [1]. Aurora DSQL partitions storage by range to preserve locality and its authors describe a planned hybrid that distributes sub-ranges of write-heavy keys by hash [19]. Limitless Database and Citus use hash distribution on a declared shard key or distribution column, and both let related tables be co-located on the same key, while small tables can be replicated to every shard as reference tables [20], [22].

## 5.2 Splitting, rebalancing and hot spots

Shards must split as data grows and move as nodes join or fail. YugabyteDB splits tablets automatically when they exceed size thresholds that rise as the number of tablets per node grows [5]. CockroachDB also splits ranges by load: when a range is busy, it samples keys and splits only if the split balances load and does not make many requests cross the new boundary, and it rebalances leases and replicas by load [32]. Aurora DSQL's control plane tracks heat and size and splits, merges and replicates shards [19].

Splitting cannot fix every hot spot. A single popular row cannot be split, and monotonically increasing keys always write to the newest range; CockroachDB's logs distinguish these cases [32], and TiDB documents write hot spots from sequential keys and monotonic index columns, mitigated by scattering row identifiers or using randomly distributed keys [33]. Aurora DSQL's documentation advises random primary keys to avoid contention [34]. Elasticity also has a cost in transactions: Accordion showed that placement must consider which partitions are accessed together, because scaling out can turn local transactions into distributed ones [35].

## 5.3 Multi-region placement and geo-partitioning

Multi-region deployments decide, per table or per row, where replicas and leaders live. VanBenschoten et al. describe how CockroachDB maps declarative SQL statements about data locality and survival goals to replica placement and partitioning, how every layer from the optimizer to replication was extended for multi-region workloads, and a transaction protocol for local, strongly consistent reads from any replica [29]. YugabyteDB offers preferred leader regions, follower reads, row-level geo-partitioning through tablespaces with per-region

placement, xCluster and read replicas as distinct patterns with different failure behavior [26]. Geo-partitioning serves both data residency and latency, but a partition whose replicas all sit in one region becomes unavailable if that region fails, and queries that touch several regions still pay cross-region round trips [26].

## 6. Distributed transactions and time

### 6.1 Commit across shards

A transaction that writes to several shards must commit on all of them or none. Most systems run a form of two-phase commit in which each participant, and often the commit record, is itself replicated by consensus. Percolator showed how to build multi-row transactions on a sharded store with a primary lock and timestamps from a central oracle [36]; TiDB adopts this model, with timestamps issued by the Placement Driver as a physical and a logical part [4], [6]. CockroachDB writes provisional values called write intents, tracks each transaction in a record, and uses a parallel commit protocol that shortens commit to one round of consensus [37]. YugabyteDB stores provisional records in a separate RocksDB instance per tablet and tracks transaction state in a replicated transaction status tablet [38]. Aurora DSQL buffers writes in the query processor until commit, when adjudicators check write-write conflicts and choose a commit time, and the transaction is written atomically to a journal [19]. Xu et al. describe a tree-structured two-phase commit for OceanBase that coalesces co-located partitions into one participant and handles participants that move during a transaction [39].

### 6.2 Clocks: TrueTime, hybrid logical clocks and bounded uncertainty

Multi-version concurrency control needs timestamps that respect real-time order across nodes. Spanner's TrueTime returns an interval that contains the true time; a transaction waits out the uncertainty before its commit becomes visible, which yields external consistency [1], [30]. Hybrid logical clocks combine physical time with a logical counter, so timestamps stay close to physical time while preserving causality without special hardware [40]. CockroachDB and YugabyteDB use hybrid logical clocks [37], [38]. Without a tight bound, they handle uncertainty differently: CockroachDB restarts reads that encounter values within an uncertainty window and stops a node whose clock drifts too far [37], and YugabyteDB notes that conflict resolution depends on the maximum clock skew, which can raise conflicts or latency [38].

Bounded-uncertainty clocks are now available outside Google. ClockBound reports an interval of earliest and latest possible time from the local clock's error bound [41]. YugabyteDB can use ClockBound as its time source on supported clouds, optionally with the Precision Time Protocol [42], and the Aurora DSQL paper ties transaction start timestamps to the provider's precision time infrastructure, noting that clock error beyond expected bounds can cost linearizability but not isolation [19]. Datacenter clock synchronization research continues to narrow these bounds [43].

### 6.3 Isolation levels

PostgreSQL defaults to Read Committed; its Repeatable Read is snapshot isolation; and its Serializable level adds Serializable Snapshot Isolation, which detects dangerous dependency patterns with non-blocking predicate locks [44], [45]. Distributed SQL systems map these levels differently. CockroachDB defaults to Serializable and also supports Read Committed, which

avoids most client retries [37]. YugabyteDB supports Serializable, Snapshot (as Repeatable Read) and Read Committed with PostgreSQL's semantics; Read Committed is enabled by default for new universes from v2025.2 when deployed with yugabyted or YugabyteDB, Inc.'s managed tools, and otherwise falls back to Snapshot [38]. Aurora DSQL fixes isolation at strong snapshot isolation, equivalent to PostgreSQL Repeatable Read, and resolves conflicts optimistically at commit with a serialization error [19], [34]. Spanner provides external consistency with serializable isolation by default [30]. TiKV provides snapshot isolation by default [6].

The same isolation name can therefore imply different retry and locking behavior on each system, so applications should test these paths explicitly.

## 6.4 Deterministic and geo-optimized alternatives

Deterministic databases agree on the order of transactions before executing them, which removes two-phase commit. Calvin replicates transaction inputs rather than effects [46]. SLOG assigns each item a home region so that most transactions commit locally while multi-home transactions remain strictly serializable [47], and Detock adds deadlock resolution that, its authors report, keeps multi-region transactions efficient under high contention [48]. These designs usually require knowing a transaction's read and write sets in advance and submitting it in one request. Nguyen et al. studied open-source applications and found the second assumption weaker in practice than the first: many transactions are interactive [49]. Other research reorders conflicting transactions by priority across regions, as in Natto [50], or replicates every update to multiple masters with epoch-based merging, as in GeoGauss [51].

# 7. Storage engines underneath

## 7.1 LSM-based key-value stores per node

Most shared-nothing distributed SQL systems store each node's shards in an LSM-tree. As paper 2 described, CockroachDB uses Pebble, TiKV uses RocksDB, and YugabyteDB's DocDB runs a customized RocksDB per tablet and disables the RocksDB write-ahead log because the Raft log already provides durability. The LSM suits the workload: replicated writes arrive as ordered batches, and MVCC adds versions that compaction later removes. Paper 2's concerns (write stalls, compaction debt, tombstones and space amplification) therefore apply per node, multiplied across replicas.

## 7.2 Other storage designs

Not every system uses an LSM. Spanner stores data in a blockwise-columnar format described in its SQL paper [2]. FoundationDB's storage servers have used a B-tree derived from SQLite, with the Redwood engine as its successor [17], [18]. OceanBase uses its own storage engine and, in the Paetica design, runs the same architecture as a single machine or a distributed cluster [7], [52]. In disaggregated systems the storage engine is a service: Aurora's storage tier applies redo log records [9], and Hyperscale page servers replay the log for their slices of the database [11].

## 7.3 Tiered and object storage for cold data

Object storage is entering the transactional stack as the durable layer behind caches. Neon's pageservers are backed by object storage that holds history while compute reads only from

caches and pageservers [13]. Hyperscale stores data files and log records in Azure storage [11]. Aurora DSQL relies on its journal and object storage for durability rather than on the storage nodes [19]. Spanner lets locality groups place data on SSD or HDD, with an optional age-based policy that moves older data during compaction [30]. A miss to a colder tier adds latency that service objectives must account for.

## 7.4 Analytical replicas and HTAP

Consensus logs can feed analytical copies. TiDB replicates Raft logs asynchronously to learner replicas in TiFlash, which convert rows to columns; before serving a read, a TiFlash replica confirms with the leader that it has caught up to the read timestamp, so analytical queries see consistent data [4], [53]. F1 Lightning adds HTAP to existing transactional systems through change data capture into a columnar store queried by a federated engine [54]. PolarDB-IMCI adds in-memory column indexes to read-only nodes of a disaggregated database [55]. Zhang et al. classify such designs by storage architecture and identify data synchronization and freshness as recurring challenges [56].

# 8. PostgreSQL compatibility

## 8.1 Three approaches

Systems claim PostgreSQL compatibility at different depths. The first approach reuses PostgreSQL's own query layer. YugabyteDB's YSQL reuses a fork of the PostgreSQL query layer over DocDB; stable series from v2025.1 are based on PostgreSQL 15, and earlier series on PostgreSQL 11 [57]. Aurora DSQL reuses PostgreSQL's parser, planner, executor and wire protocol but replaces storage, transaction processing, the buffer pool and the lock manager [19]. Neon runs PostgreSQL compute over a new storage service [13], the same principle Aurora applies with its own storage tier [9].

The second approach reimplements SQL behind the PostgreSQL wire protocol. CockroachDB speaks version 3.0 of the protocol and supports much of PostgreSQL's syntax with its own SQL engine [58]. Spanner's PostgreSQL interface supports a subset of the PostgreSQL dialect and the wire protocol, directly or through the PGAdapter proxy, and does not support extensions, user-defined types or stored procedures [59].

The third approach extends PostgreSQL itself. Citus is an extension that distributes tables, queries and transactions across PostgreSQL servers, so each worker runs ordinary PostgreSQL and the coordinator routes, splits and merges queries across them [21], [22]. Limitless Database follows a similar router-and-shard model built on Aurora PostgreSQL [20].

## 8.2 What typically differs

Even the closest reuse leaves differences, because storage, locking and DDL must work across nodes. Documented gaps fall into recurring groups:

- **Constraints and indexes.** YugabyteDB lists exclusion and deferrable constraints, GiST indexes and some index types as unsupported [57]; CockroachDB lists range types and some schema operations [58].
- **Locking and isolation.** CockroachDB supports transaction-scoped but not session-scoped advisory locks, and reports deadlocks as retry errors [58]; Aurora DSQL implements `SELECT

... FOR UPDATE` through commit-time conflict detection rather than locks [34]. Retry handling becomes a normal code path.

- **DDL.** DDL inside an explicit transaction does not behave as in PostgreSQL on YugabyteDB, according to its migration known-issues list [57]; Aurora DSQL requires DDL and DML in separate transactions [34].
- **Extensions and procedural code.** Spanner's interface does not support extensions [59]; Aurora DSQL supports SQL functions but not PL/pgSQL [34]; YSQL supports many extensions, but unsupported ones fail at schema import [57].
- **Sequences and identifiers.** Monotonic identifiers create hot spots on range-sharded data and need coordination across nodes, so systems recommend random or hash-distributed keys and document sequence behavior separately [34], [33], [58].
- **Performance characteristics.** Joins, foreign-key checks and index lookups that are local on one node can cross the network. Plans that suit PostgreSQL may not suit a distributed system, and Spanner's documentation expects rework for query optimization and primary-key design [59].

### 8.3 Migration considerations

Compatibility is a property of the application, not of the product. A service that uses plain tables, indexes and transactions may move with little change; one that relies on extensions, advisory locks, transactional DDL or deferrable constraints needs design decisions first. A sound migration inventories schema objects and queries with the target project's assessment tools, tests against the exact release and configuration that production will run, redesigns primary keys for distribution, rehearses data movement with reconciliation, and keeps a fallback path until cutover is proven. Tools cannot see code paths that never ran during assessment.

## 9. Comparative analysis

Table 1 compares architecture families discussed in this paper, with representative systems. The ratings are qualitative and relative to each other. Actual behavior depends on version, configuration, topology, schema design and workload.

| Architecture family (examples)                                                                | Consistency model                                                                               | Scaling model                                           | Multi-region support                                                                     | PostgreSQL compatibility approach                          | Best-fit workloads                                                   |
|-----------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------|---------------------------------------------------------|------------------------------------------------------------------------------------------|------------------------------------------------------------|----------------------------------------------------------------------|
| <b>Single-node PostgreSQL with streaming replicas (baseline)</b>                              | Serializable available; strong on the primary; replicas may lag                                 | Scale up; read replicas for reads                       | Low (asynchronous standbys; failover can lose recent commits)                            | Native                                                     | Workloads one primary can serve, with standby-based recovery         |
| <b>Extension-based sharding (Citus; Aurora Limitless as a related router-and-shard model)</b> | PostgreSQL semantics per shard; distributed transactions coordinated by a coordinator or router | Scale out by shard key; co-located and reference tables | Low to Medium (inherits the replication options of the underlying PostgreSQL deployment) | PostgreSQL itself on each node, plus distribution metadata | Multi-tenant and analytics-heavy applications with a clear shard key |

| Architecture family (examples)                                    | Consistency model                                                 | Scaling model                                                 | Multi-region support                                                                       | PostgreSQL compatibility approach                                                 | Best-fit workloads                                                                               |
|-------------------------------------------------------------------|-------------------------------------------------------------------|---------------------------------------------------------------|--------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------|
| <b>Shared-nothing, PostgreSQL query layer reused (YugabyteDB)</b> | Raft per tablet; Serializable, Snapshot and Read Committed        | Scale out; hash or range sharding; automatic tablet splitting | High (synchronous across three or more regions, geo-partitioning, xCluster, read replicas) | Forked PostgreSQL query layer over distributed storage                            | PostgreSQL applications needing zone or region resilience or write scale                         |
| <b>Shared-nothing, PostgreSQL wire protocol (CockroachDB)</b>     | Raft per range; Serializable by default, Read Committed available | Scale out; range sharding with size- and load-based splits    | High (declarative multi-region tables, follower reads)                                     | Own SQL engine behind the PostgreSQL wire protocol                                | Geo-distributed OLTP with strong consistency requirements                                        |
| <b>Synchronized-clock global database (Spanner)</b>               | Paxos per tablet; external consistency                            | Scale out; range sharding with automatic resharding           | High (synchronous replication across data centers)                                         | PostgreSQL dialect subset and wire protocol via an interface                      | Globally distributed OLTP on a managed service                                                   |
| <b>Disaggregated single writer (Aurora, Hyperscale, Neon)</b>     | PostgreSQL or SQL Server semantics on one writer                  | Scale storage and read replicas; one writer                   | Medium (geo-secondary or cross-region replicas, typically asynchronous)                    | PostgreSQL itself over replicated storage (Aurora PostgreSQL, Neon)               | PostgreSQL workloads needing elastic storage, fast replicas or branching without write scale-out |
| <b>Serverless disaggregated distributed SQL (Aurora DSQL)</b>     | Strong snapshot isolation with optimistic commit                  | Independent scaling of query, commit and storage services     | High (active-active multi-region)                                                          | PostgreSQL parser, planner and executor over new storage; reduced feature surface | New applications designed for optimistic retries and serverless operation                        |
| <b>MySQL-compatible HTAP (TiDB, for contrast)</b>                 | Raft per Region; snapshot isolation by default                    | Scale out; range sharding with Region splits                  | Medium (Raft replicas can span zones or regions)                                           | None (MySQL protocol)                                                             | Mixed transactional and analytical workloads in the MySQL ecosystem                              |

Table 1: Qualitative comparison of distributed SQL architecture families and related designs.

## 10. Open challenges and future opportunities

### 10.1 Cross-region latency versus consistency

**Challenge.** Synchronous replication across regions puts at least one cross-region round trip on every write, and strongly consistent reads must reach a leaseholder or wait for closed timestamps [29], [28]. **Research direction.** Home-region designs such as SLOG and Detock make most transactions local [47], [48]; priority-aware ordering reduces tail latency for

important transactions [50]; global tables trade write latency for local strong reads [29]; and active-active designs such as Aurora DSQL confine cross-region coordination to commit [19]. **Opportunity.** Placement that adapts to observed access locality, per table or per row, would let systems move leaders and home regions as traffic shifts, rather than relying on static declarations.

## 10.2 Tail latency and hot spots under skew

**Challenge.** Skewed keys, sequential inserts and popular rows concentrate load on one shard, and splitting does not help a single hot key [32], [33]. Optimistic designs turn contention into aborts and retries [34]. **Research direction.** Load-based splitting with balance and boundary-crossing checks [32], affinity-aware placement [35], hybrid range-and-hash partitioning [19] and contention-aware deterministic ordering [48] each address part of the problem.

**Opportunity.** Systems that report contention per key and per index, and suggest schema changes such as key randomization or co-location, would shorten diagnosis, which today depends on log messages and expertise.

## 10.3 Elasticity and serverless economics

**Challenge.** Scaling out moves data, and scaling to zero requires that no durable state live in compute. Shared-nothing systems must rebalance replicas; serverless systems must restore caches and connections quickly. **Research direction.** CockroachDB Serverless virtualizes clusters for many tenants on shared infrastructure, with scaling from zero and isolation between tenants [60]. Marlin moves cluster coordination into the cloud database itself rather than an external coordination service [61]. PolarDB Serverless, Neon and Aurora DSQL separate state from compute [14], [13], [19]. **Opportunity.** Cost models that price data movement, replication and cold starts alongside compute would let teams predict the cost of elasticity before deployment.

## 10.4 Compatibility gaps and migration effort

**Challenge.** Every system in this paper documents PostgreSQL features it lacks or implements differently (Section 8.2), and the lists change with each release [57], [58], [34], [59]. **Research direction.** Reusing the PostgreSQL query layer narrows syntax and semantic gaps [57], [19], and migration assessment tools, such as YugabyteDB Voyager's assessment report, inventory unsupported schema features and query constructs before a move [57]. **Opportunity.** A shared, versioned conformance suite for PostgreSQL behavior (isolation, locking, DDL, sequences and common extensions), published per system and release, would replace broad claims of compatibility with testable statements.

## 10.5 HTAP freshness in distributed SQL

**Challenge.** Analytical replicas fed from consensus logs or change streams lag the transactional copy, and reconciling freshness with isolation is harder when data spans many shards [56]. **Research direction.** TiFlash learners check replication progress against the leader before serving reads [53]; F1 Lightning adds HTAP to existing systems without owning them [54]; and PolarDB-IMCI keeps column indexes on read-only nodes [55]. **Opportunity.** Exposing freshness as an explicit, per-query parameter, as follower reads already do for staleness [28], would let analytical users trade freshness for isolation deliberately. Paper 5's open table formats are a related target for exporting consistent snapshots to the lakehouse.

## 10.6 Operational complexity and observability

**Challenge.** A distributed database has many more components than a single primary: consensus groups, leases, placement policies, clocks, background splits and cross-cluster replication. Failures of interaction between them are hard to observe. **Research direction.** Elle checks client-observed transaction histories, such as those recorded under fault injection, for anomalies that violate a database's stated isolation level [62]; formal specification, as in LeaseGuard's TLA+ model, makes protocols checkable [27]. **Opportunity.** Built-in tools that explain why a transaction retried, why a range moved or why a read was stale, tied to the placement and isolation settings in force, would reduce the expertise needed to run these systems.

## 10.7 Clock and failure-model assumptions

**Challenge.** Correctness and performance rest on assumptions about clock error and failure behavior. Systems based on hybrid logical clocks depend on a configured maximum offset [37], [38]. Gray failures, in which components degrade without failing outright, evade failure detectors [63], and metastable failures persist after their trigger is removed [64]. YugabyteDB's documentation notes that a leader with a stalled log can keep its leadership while it still heartbeats [5]. **Research direction.** Bounded-uncertainty clocks [41], [42], [43], store-level liveness for leases [25] and rigorously specified lease protocols [27] tighten these assumptions. **Opportunity.** Fault-injection suites that include clock drift, slow disks and partial partitions, run before each upgrade, would test the assumptions that availability claims depend on.

## 10.8 The cost of strong consistency for AI and agent workloads

**Challenge.** AI agents generate many speculative queries and writes [65], and branchable databases face a tension between fast branching and fast data operations [66]. Strong consistency across regions adds latency and cost to every write, whether or not an agent's exploratory change will be kept. **Research direction.** Agent-first data systems research proposes new interfaces, query processing and memory stores for this workload [65]; copy-on-write branching in disaggregated storage lets agents work on isolated copies [13], [66]. **Opportunity.** Distributed SQL systems that offer cheap, isolated branches with explicit consistency levels per branch, and promote only accepted changes to strongly consistent storage, would match the cost of consistency to the value of the data. Paper 4's case for keeping vectors close to governed data applies equally.

# 11. What this means for platform decisions

Start with PostgreSQL when one primary, with standbys and tested recovery, meets the workload's capacity and resilience needs. Consider YugabyteDB, which Bassirah supports through independent open-source engineering (Bassirah is not a YugabyteDB, Inc. partner), or another distributed SQL option when zone or region resilience through synchronous replication, or write scale beyond one node, is a requirement, and budget for schema redesign, compatibility testing and the latency of synchronous replication. In each case, validate compatibility, failure behavior and performance on representative workloads before adoption, and assume no outcome from published results.

This paper closes the Data Storage Engines series. Paper 1 argued that the class of engine a workload needs follows from its dominant trade-off; papers 2 to 6 examined those trade-offs in

LSM-trees, columnar engines, vector indexes, open table formats and distributed SQL. The same lesson recurs in every paper: choose the engine class before the product, then validate the product against the workload.

## References

1. Corbett, J. C., Dean, J., Epstein, M., Fikes, A., Frost, C., Furman, J. J., Ghemawat, S., Gubarev, A., Heiser, C., Hochschild, P., et al. (2012). Spanner: Google's Globally-Distributed Database. 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI). (<https://www.usenix.org/conference/osdi12/technical-sessions/presentation/corbett>)
2. Bacon, D. F., Bales, N., Bruno, N., Cooper, B. F., Dickinson, A., Fikes, A., Fraser, C., Gubarev, A., Joshi, M., Kogan, E., Lloyd, A., Melnik, S., Rao, R., Shue, D., Taylor, C., van der Holst, M. and Woodford, D. (2017). Spanner: Becoming a SQL System. SIGMOD, 331–343. (<https://doi.org/10.1145/3035918.3056103>)
3. Taft, R., Sharif, I., Matei, A., VanBenschoten, N., Lewis, J., Grieger, T., Niemi, K., Woods, A., Birzin, A., Poss, R., et al. (2020). CockroachDB: The Resilient Geo-Distributed SQL Database. SIGMOD, 1493–1509. (<https://doi.org/10.1145/3318464.3386134>)
4. Huang, D., Liu, Q., Cui, Q., Fang, Z., Ma, X., Xu, F., Shen, L., Tang, L., Zhou, Y., Huang, M., et al. (2020). TiDB: A Raft-based HTAP Database. PVLDB, 13(12), 3072–3084. (<https://doi.org/10.14778/3415478.3415535>)
5. YugabyteDB, Inc. (2026). YugabyteDB documentation: Architecture; Query layer; Hash and range sharding; Tablet splitting; Raft consensus; Replication in DocDB. Official documentation sources, accessed 26 October 2026. ([https://raw.githubusercontent.com/yugabyte/yugabyte-db/master/docs/content/stable/architecture/\\_index.md](https://raw.githubusercontent.com/yugabyte/yugabyte-db/master/docs/content/stable/architecture/_index.md); [https://raw.githubusercontent.com/yugabyte/yugabyte-db/master/docs/content/stable/architecture/query-layer/\\_index.md](https://raw.githubusercontent.com/yugabyte/yugabyte-db/master/docs/content/stable/architecture/query-layer/_index.md); <https://raw.githubusercontent.com/yugabyte/yugabyte-db/master/docs/content/stable/architecture/docdb-sharding/sharding.md>; <https://raw.githubusercontent.com/yugabyte/yugabyte-db/master/docs/content/stable/architecture/docdb-sharding/tablet-splitting.md>; <https://raw.githubusercontent.com/yugabyte/yugabyte-db/master/docs/content/stable/architecture/docdb-replication/raft.md>; <https://raw.githubusercontent.com/yugabyte/yugabyte-db/master/docs/content/stable/architecture/docdb-replication/replication.md>)
6. PingCAP (2026). TiDB Architecture; TimeStamp Oracle (TSO) in TiDB. Official documentation, accessed 26 October 2026. (<https://docs.pingcap.com/tidb/stable/tidb-architecture/>; <https://docs.pingcap.com/tidb/stable/tso/>)
7. Yang, Z., Yang, C., Han, F., Zhuang, M., Yang, B., Yang, Z., Cheng, X., Zhao, Y., Shi, W., Xi, H., et al. (2022). OceanBase: A 707 Million tpmC Distributed Relational Database System. PVLDB, 15(12), 3385–3397. (<https://doi.org/10.14778/3554821.3554830>)
8. Cao, W., Li, F., Huang, G., Lou, J., Zhao, J., He, D., Sun, M., Zhang, Y., Wang, S., Wu, X., et al. (2022). PolarDB-X: An Elastic Distributed Relational Database for Cloud-Native Applications. IEEE ICDE, 2859–2872. (<https://doi.org/10.1109/ICDE53745.2022.00259>)
9. Verbitski, A., Gupta, A., Saha, D., Brahmadesam, M., Gupta, K., Mittal, R., Krishnamurthy, S., Maurice, S., Kharatishvili, T. and Bao, X. (2017). Amazon Aurora: Design Considerations for

- High Throughput Cloud-Native Relational Databases. SIGMOD, 1041–1052. (<https://doi.org/10.1145/3035918.3056101>)
10. Antonopoulos, P., Budovski, A., Diaconu, C., Hernandez Saenz, A., Hu, J., Kodavalla, H., Kossmann, D., Lingam, S., Minhas, U. F., Prakash, N., et al. (2019). Socrates: The New SQL Server in the Cloud. SIGMOD, 1743–1756. (<https://doi.org/10.1145/3299869.3314047>)
  11. Microsoft (2026). Hyperscale distributed functions architecture (Azure SQL Database). Official documentation, accessed 26 October 2026. (<https://learn.microsoft.com/en-us/azure/azure-sql/database/hyperscale-architecture>)
  12. Depoutovitch, A., Chen, C., Chen, J., Larson, P., Lin, S., Ng, J., Cui, W., Liu, Q., Huang, W., Xiao, Y. and He, Y. (2020). Taurus Database: How to be Fast, Available, and Frugal in the Cloud. SIGMOD, 1463–1478. (<https://doi.org/10.1145/3318464.3386129>)
  13. Neon (2026). Neon architecture overview. Official documentation, accessed 26 October 2026. (<https://neon.com/docs/introduction/architecture-overview>)
  14. Cao, W., Zhang, Y., Yang, X., Li, F., Wang, S., Hu, Q., Cheng, X., Chen, Z., Liu, Z., Fang, J., et al. (2021). PolarDB Serverless: A Cloud Native Database for Disaggregated Data Centers. SIGMOD, 2477–2489. (<https://doi.org/10.1145/3448016.3457560>)
  15. Wang, J. and Zhang, Q. (2023). Disaggregated Database Systems. Companion of the 2023 International Conference on Management of Data (SIGMOD), 37–44. (<https://doi.org/10.1145/3555041.3589403>)
  16. Pang, X. and Wang, J. (2024). Understanding the Performance Implications of the Design Principles in Storage-Disaggregated Databases. Proceedings of the ACM on Management of Data (SIGMOD), 2(3). (<https://doi.org/10.1145/3654983>)
  17. Zhou, J., Xu, M., Shraer, A., Namasivayam, B., Miller, A., Tschannen, E., Atherton, S., Beamon, A. J., Sears, R., Leach, J., et al. (2021). FoundationDB: A Distributed Unbundled Transactional Key Value Store. SIGMOD, 2653–2666. (<https://doi.org/10.1145/3448016.3457559>)
  18. FoundationDB project (2026). FoundationDB Architecture. Official documentation, accessed 26 October 2026. (<https://apple.github.io/foundationdb/architecture.html>)
  19. Brooker, M., Bowes, M., Hershey, M., van der Merwe, Z., Morle, J. and Strydom, M. (2026). Aurora DSQL: Scalable, Multi-Region OLTP. arXiv preprint 2607.13276. (<https://arxiv.org/abs/2607.13276>)
  20. Amazon Web Services (2026). Aurora PostgreSQL Limitless Database architecture. Amazon Aurora User Guide, accessed 26 October 2026. (<https://docs.aws.amazon.com/AmazonRDS/latest/AuroraUserGuide/limitless-architecture.html>)
  21. Cubukcu, U., Erdogan, O., Pathak, S., Sannakkayala, S. and Slot, M. (2021). Citus: Distributed PostgreSQL for Data-Intensive Applications. SIGMOD, 2490–2502. (<https://doi.org/10.1145/3448016.3457551>)
  22. Citus Data (2026). Concepts: Nodes; Distributed Data. Citus documentation, accessed 26 October 2026. ([https://docs.citusdata.com/en/stable/get\\_started/concepts.html](https://docs.citusdata.com/en/stable/get_started/concepts.html))
  23. Chandra, T. D., Griesemer, R. and Redstone, J. (2007). Paxos Made Live: An Engineering Perspective. ACM Symposium on Principles of Distributed Computing (PODC), 398–407. (<https://doi.org/10.1145/1281100.1281103>)
  24. Ongaro, D. and Ousterhout, J. (2014). In Search of an Understandable Consensus Algorithm. USENIX Annual Technical Conference (ATC). (<https://www.usenix.org/conference/atc14/technical-sessions/presentation/ongaro>)

25. Cockroach Labs (2026). Replication Layer. CockroachDB documentation (stable), accessed 26 October 2026.  
(<https://docs.cockroachlabs.com/docs/stable/architecture/replication-layer>)
26. YugabyteDB, Inc. (2026). YugabyteDB documentation: Multi-region deployments; Row-level geo-partitioning; xCluster replication; Read replicas. Official documentation sources, accessed 26 October 2026. ([https://raw.githubusercontent.com/yugabyte/yugabyte-db/master/docs/content/stable/explore/multi-region-deployments/\\_index.md](https://raw.githubusercontent.com/yugabyte/yugabyte-db/master/docs/content/stable/explore/multi-region-deployments/_index.md);  
<https://raw.githubusercontent.com/yugabyte/yugabyte-db/master/docs/content/stable/explore/multi-region-deployments/row-level-geo-partitioning.md>; <https://raw.githubusercontent.com/yugabyte/yugabyte-db/master/docs/content/stable/architecture/docdb-replication/async-replication.md>;  
<https://raw.githubusercontent.com/yugabyte/yugabyte-db/master/docs/content/stable/architecture/docdb-replication/read-replicas.md>)
27. Davis, A. J. J., Demirbas, M. and Deng, L. (2026). LeaseGuard: Raft Leases Done Right. Proceedings of the ACM on Management of Data (SIGMOD), 4(1).  
(<https://doi.org/10.1145/3786663>)
28. Cockroach Labs (2026). Follower Reads. CockroachDB documentation (stable), accessed 26 October 2026. (<https://docs.cockroachlabs.com/docs/stable/follower-reads>)
29. VanBenschoten, N., Ajmani, A., Gartner, M., Matei, A., Shah, A., Sharif, I., Shraer, A., Storm, A., Taft, R., Tan, O., Woods, A. and Walters, P. (2022). Enabling the Next Generation of Multi-Region Applications with CockroachDB. SIGMOD, 2312–2325.  
(<https://doi.org/10.1145/3514221.3526053>)
30. Google Cloud (2026). Spanner: TrueTime and external consistency; Tiered storage overview. Official documentation, accessed 26 October 2026.  
(<https://docs.cloud.google.com/spanner/docs/true-time-external-consistency>;  
<https://docs.cloud.google.com/spanner/docs/tiered-storage>)
31. Memarzia, P., Zhang, H., Ho, K., Grosman, R. and Wang, J. (2024). GaussDB-Global: A Geographically Distributed Database System. IEEE ICDE, 5111–5118.  
(<https://doi.org/10.1109/ICDE60146.2024.00383>)
32. Cockroach Labs (2026). Load-Based Splitting. CockroachDB documentation (stable), accessed 26 October 2026. (<https://docs.cockroachlabs.com/docs/stable/load-based-splitting>)
33. PingCAP (2026). Troubleshoot Hotspot Issues. TiDB documentation, accessed 26 October 2026. (<https://docs.pingcap.com/tidb/stable/troubleshoot-hot-spot-issues/>)
34. Amazon Web Services (2026). Concurrency control in Aurora DSQL; Migrating from PostgreSQL to Aurora DSQL. Amazon Aurora DSQL User Guide, accessed 26 October 2026.  
(<https://docs.aws.amazon.com/aurora-dsql/latest/userguide/working-with-concurrency-control.html>; <https://docs.aws.amazon.com/aurora-dsql/latest/userguide/working-with-postgresql-compatibility-unsupported-features.html>)
35. Serafini, M., Mansour, E., Aboulhaga, A., Salem, K., Rafiq, T. and Minhas, U. F. (2014). Accordion: Elastic Scalability for Database Systems Supporting Distributed Transactions. PVLDB, 7(12), 1035–1046. (<https://doi.org/10.14778/2732977.2732979>)
36. Peng, D. and Dabek, F. (2010). Large-scale Incremental Processing Using Distributed Transactions and Notifications. 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI). (<https://www.usenix.org/conference/osdi10/large-scale-incremental-processing-using-distributed-transactions-and>)

37. Cockroach Labs (2026). Transaction Layer. CockroachDB documentation (stable), accessed 26 October 2026.  
(<https://docs.cockroachlabs.com/docs/stable/architecture/transaction-layer>)
38. YugabyteDB, Inc. (2026). YugabyteDB documentation: Transactions; Transaction fundamentals; Transaction isolation levels; Distributed transactions. Official documentation sources, accessed 26 October 2026.  
([https://raw.githubusercontent.com/yugabyte/yugabyte-db/master/docs/content/stable/architecture/transactions/\\_index.md](https://raw.githubusercontent.com/yugabyte/yugabyte-db/master/docs/content/stable/architecture/transactions/_index.md);  
<https://raw.githubusercontent.com/yugabyte/yugabyte-db/master/docs/content/stable/architecture/transactions/transactions-overview.md>;  
<https://raw.githubusercontent.com/yugabyte/yugabyte-db/master/docs/content/stable/architecture/transactions/isolation-levels.md>;  
<https://raw.githubusercontent.com/yugabyte/yugabyte-db/master/docs/content/stable/architecture/transactions/distributed-txns.md>)
39. Xu, Q., Qian, C., Yang, C., Kong, F., Liu, G., Han, F. and Zhai, Z. (2026). A Tree-Structured Two-Phase Commit Framework for OceanBase: Optimizing Scalability and Consistency. PVLDB, 19(12), 4250–4262. (<https://doi.org/10.14778/3827998.3828030>)
40. Kulkarni, S. S., Demirbas, M., Madappa, D., Avva, B. and Leone, M. (2014). Logical Physical Clocks. Principles of Distributed Systems (OPODIS), LNCS 8878, 17–32.  
([https://doi.org/10.1007/978-3-319-14472-6\\_2](https://doi.org/10.1007/978-3-319-14472-6_2))
41. Amazon Web Services (2026). ClockBound repository README. Accessed 26 October 2026.  
(<https://github.com/aws/clock-bound>)
42. YugabyteDB, Inc. (2026). System configuration: Configure Precision Time Protocol; Configure ClockBound. YugabyteDB documentation source, accessed 26 October 2026.  
(<https://raw.githubusercontent.com/yugabyte/yugabyte-db/master/docs/content/stable/deploy/manual-deployment/system-config.md>)
43. Namyar, P., Li, Y., Wang, W., Dukkupati, N., Yap, K.-K., Gong, J., Chen, C., Gao, P., Ray, D., Kumar, G., Ma, Y., Govindan, R. and Vahdat, A. (2025). Firefly: Scalable, Ultra-Accurate Clock Synchronization for Datacenters. ACM SIGCOMM, 434–452.  
(<https://doi.org/10.1145/3718958.3750502>)
44. PostgreSQL Global Development Group (2026). PostgreSQL 18 Documentation, Section 13.2: Transaction Isolation. Accessed 26 October 2026.  
(<https://www.postgresql.org/docs/current/transaction-iso.html>)
45. Ports, D. R. K. and Grittnner, K. (2012). Serializable Snapshot Isolation in PostgreSQL. PVLDB, 5(12), 1850–1861. (<https://doi.org/10.14778/2367502.2367523>)
46. Thomson, A., Diamond, T., Weng, S.-C., Ren, K., Shao, P. and Abadi, D. J. (2012). Calvin: Fast Distributed Transactions for Partitioned Database Systems. SIGMOD, 1–12.  
(<https://doi.org/10.1145/2213836.2213838>)
47. Ren, K., Li, D. and Abadi, D. J. (2019). SLOG: Serializable, Low-latency, Geo-replicated Transactions. PVLDB, 12(11), 1747–1761. (<https://doi.org/10.14778/3342263.3342647>)
48. Nguyen, C. D. T., Miller, J. and Abadi, D. J. (2023). Detock: High Performance Multi-region Transactions at Scale. Proceedings of the ACM on Management of Data (SIGMOD), 1(2).  
(<https://doi.org/10.1145/3589293>)
49. Nguyen, C. D. T., Chen, K., DeCarolis, C. and Abadi, D. J. (2025). Are Database System Researchers Making Correct Assumptions about Transaction Workloads? Proceedings of the ACM on Management of Data (SIGMOD), 3(3). (<https://doi.org/10.1145/3725268>)

50. Yang, L., Yan, X. and Wong, B. (2022). Natto: Providing Distributed Transaction Prioritization for High-Contention Workloads. SIGMOD, 715–729.  
(<https://doi.org/10.1145/3514221.3526161>)
51. Zhou, W., Peng, Q., Zhang, Z., Zhang, Y., Ren, Y., Li, S., Fu, G., Cui, Y., Li, Q., Wu, C., et al. (2023). GeoGauss: Strongly Consistent and Light-Coordinated OLTP for Geo-Replicated SQL Database. Proceedings of the ACM on Management of Data (SIGMOD), 1(1).  
(<https://doi.org/10.1145/3588916>)
52. Yang, Z., Xu, Q., Gao, S., Yang, C., Wang, G., Zhao, Y., Kong, F., Liu, H., Wang, W. and Xiao, J. (2023). OceanBase Paetica: A Hybrid Shared-Nothing/Shared-Everything Database for Supporting Single Machine and Distributed Cluster. PVLDB, 16(12), 3728–3740.  
(<https://doi.org/10.14778/3611540.3611560>)
53. PingCAP (2026). TiFlash Overview. TiDB documentation, accessed 26 October 2026.  
(<https://docs.pingcap.com/tidb/stable/tiflash-overview/>)
54. Yang, J., Rae, I., Xu, J., Shute, J., Yuan, Z., Lau, K., Zeng, Q., Zhao, X., Ma, J., Chen, Z., et al. (2020). F1 Lightning: HTAP as a Service. PVLDB, 13(12), 3313–3325.  
(<https://doi.org/10.14778/3415478.3415553>)
55. Wang, J., Li, T., Song, H., Yang, X., Zhou, W., Li, F., Yan, B., Wu, Q., Liang, Y., Ying, C., et al. (2023). PolarDB-IMCI: A Cloud-Native HTAP Database System at Alibaba. Proceedings of the ACM on Management of Data (SIGMOD), 1(2). (<https://doi.org/10.1145/3589785>)
56. Zhang, C., Li, G., Zhang, J., Zhang, X. and Feng, J. (2024). HTAP Databases: A Survey. IEEE Transactions on Knowledge and Data Engineering, 36(11), 6410–6429.  
(<https://doi.org/10.1109/TKDE.2024.3389693>)
57. YugabyteDB, Inc. (2026). YugabyteDB documentation: PostgreSQL 15 features in YSQL; Enhanced PostgreSQL Compatibility Mode; Voyager known issues (PostgreSQL); Voyager migration assessment. Official documentation sources, accessed 26 October 2026.  
(<https://raw.githubusercontent.com/yugabyte/yugabyte-db/master/docs/content/stable/api/ysql/pg15-features.md>;  
<https://raw.githubusercontent.com/yugabyte/yugabyte-db/master/docs/content/stable/reference/configuration/postgresql-compatibility.md>;  
<https://raw.githubusercontent.com/yugabyte/yugabyte-db/master/docs/content/stable/yugabyte-voyager/known-issues/postgresql.md>;  
<https://raw.githubusercontent.com/yugabyte/yugabyte-db/master/docs/content/stable/yugabyte-voyager/migrate/assess-migration.md>)
58. Cockroach Labs (2026). PostgreSQL Compatibility. CockroachDB documentation (stable), accessed 26 October 2026. (<https://docs.cockroachlabs.com/docs/stable/postgresql-compatibility>)
59. Google Cloud (2026). PostgreSQL interface for Spanner. Official documentation, accessed 26 October 2026. (<https://docs.cloud.google.com/spanner/docs/postgresql-interface>)
60. Swenson, J., Kimball, A., Poss, R., Taft, R., Lim, J., Storm, A. J., Bhola, S., Bulkley-Logston, P., Tatlow, P., Harding, R., Shamim, R., Maru, A. and Sharif, I. (2025). CockroachDB Serverless: Sub-second Scaling from Zero with Multi-region Cluster Virtualization. Companion of the 2025 International Conference on Management of Data (SIGMOD), 648–661.  
(<https://doi.org/10.1145/3722212.3724432>)
61. Hu, W., Hu, G., Balakrishnan, M. and Yu, X. (2025). Marlin: Efficient Coordination for Autoscaling Cloud DBMS. Proceedings of the ACM on Management of Data (SIGMOD), 3(4). (<https://doi.org/10.1145/3749173>)

62. Kingsbury, K. and Alvaro, P. (2020). Elle: Inferring Isolation Anomalies from Experimental Observations. PVLDB, 14(3), 268–280. (<https://doi.org/10.14778/3430915.3430918>)
63. Huang, P., Guo, C., Zhou, L., Lorch, J. R., Dang, Y., Chintalapati, M. and Yao, R. (2017). Gray Failure: The Achilles' Heel of Cloud-Scale Systems. Workshop on Hot Topics in Operating Systems (HotOS), 150–155. (<https://doi.org/10.1145/3102980.3103005>)
64. Huang, L., Magnusson, M., Muralikrishna, A. B., Estyak, S., Isaacs, R., Aghayev, A., Zhu, T. and Charapko, A. (2022). Metastable Failures in the Wild. 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI), 73–90. (<https://www.usenix.org/conference/osdi22/presentation/huang-lexiang>)
65. Liu, S., Ponnappalli, S., Shankar, S., Zeighami, S., Zhu, A., Agarwal, S., Chen, R., Suwito, S., Yuan, S., Stoica, I., Zaharia, M., Cheung, A., Crooks, N., Gonzalez, J. E. and Parameswaran, A. G. (2026). Supporting Our AI Overlords: Redesigning Data Systems to be Agent-First. CIDR. (<https://vldb.org/cidrrdb/papers/2026/p32-liu.pdf>)
66. Ang, E., Weldon, S., Kim, I. K., Durand, K., Kaffes, K. and Wu, E. (2026). BranchBench: Aligning Database Branching with Agentic Demands. arXiv preprint 2604.17180. (<https://arxiv.org/abs/2604.17180>)

Product and project names are trademarks of their respective owners. Their mention describes publicly documented technology and does not imply a business relationship. Bassirah provides independent engineering for YugabyteDB and is not a partner of YugabyteDB, Inc.