

# ACID vs BASE in Practice: Benchmarks across E-commerce, Social Media, and IoT

Petar Risteski  
AITMIR - INSPDP  
[petar.risteski@mir.uist.edu.mk](mailto:petar.risteski@mir.uist.edu.mk)

12.11.2025



## Abstract

This work presents a comparative benchmark suite contrasting ACID and BASE design choices across three representative application domains: e-commerce, social media, and IoT. We implement realistic scenarios with correctness stress (late failure rollback, single hot SKU contention), mixed write/read social workloads, and write- and read-heavy IoT time-series patterns. The suite spans PostgreSQL (ACID), MongoDB, and Apache Cassandra (BASE), with unified CLI, metrics collection, and analysis. We report throughput, tail latency, and correctness KPIs (e.g., oversell and orphan-payment rates) under controlled concurrency and duration across identical hardware. Results indicate clear trade-offs: PostgreSQL provides strong correctness and predictable tails under contention, while BASE systems offer flexible schemas and horizontal scale with consistency caveats. We release the framework, methodology, and artifacts (plots, KPIs) to enable reproducibility and to facilitate informed system and data-model decisions.

## 1 Introduction

Choosing between ACID and BASE is not just a theory question. Teams face it when they design order flows, feeds, and time-series pipelines. The trade-offs show up as concrete user outcomes: oversold items, missing read-your-write reads, long tail latencies, or slower ingest. While there is a lot of discussion in the literature, engineers still need numbers on realistic tasks to guide decisions.

This paper presents a small, reproducible benchmark suite that compares PostgreSQL (ACID), MongoDB, and Cassandra (BASE-oriented) across three domains: e-commerce, social media, and IoT. Each domain has two focused scenarios that reflect common patterns (e.g., late-failure rollback, a single hot SKU, concurrent social actions with immediate reads, global feed reads, high-rate sensor ingest, and time-series range queries). For each scenario we report throughput, median and 95th-percentile latency, and correctness signals such as oversell, orphan payments, and read-your-writes.

Our contributions are: (1) a simple, open set of runners and analysis tools with a unified CLI, (2) comparable results for three widely used engines on realistic tasks, and (3) plots and KPIs that make the trade-offs visible. We aim for clarity over micro-optimizations: straightforward schemas, minimal tuning, and repeatable defaults so that others can reproduce the results or extend the suite.

## 2 Related Work

The ACID vs. BASE discussion has a few key guides. Pritchett explains BASE as a practical way to keep systems available and scalable by allowing temporary inconsistencies [1]. Brewer clarifies CAP and notes that partitions do happen, so systems need a plan for what to give up during a partition [2]. Abadi adds PACELC: even without a partition, we often trade latency and efficiency for stronger consistency (or the other way around) [3]. Our benchmarks fit into this space: we measure correctness signals (oversell, orphan payments, read-your-writes) together with throughput and tail latency.

On the BASE side, Dynamo shows how to keep a key-value store always writeable and resolve conflicts later [5]. Vogels explains eventual consistency as a deliberate design choice [4]. These ideas connect to our social-media workload, where we stress duplicate-like handling and read-your-writes under high concurrency.

On the strong-consistency side, Spanner shows that global transactions are possible with the right clock and replication design [6]. Megastore offers a middle ground with small transactional groups [7]. This perspective informs our e-commerce scenarios: late-failure rollback and a single hot SKU put pressure on transaction handling, retries, and tail latency.

Finally, Bailis et al. define which transactional guarantees you can keep while staying highly available (HATs) [8]. Brewer later discusses practical patterns that combine availability with useful consistency [9]. Our work complements these papers with hands-on measurements across PostgreSQL (ACID), MongoDB, and Cassandra on e-commerce, social media, and IoT tasks, so readers can see the trade-offs in numbers and not only in theory.

## 3 Methodology

Our goal is to compare ACID and BASE choices on tasks that look like real systems. We focus on three domains and run the same workflow for PostgreSQL (ACID), MongoDB, and Cassandra (BASE-oriented):

### Scenarios

- **E-commerce.** (1) *Rollback*: users place orders with a chance of a late failure that forces rollback/compensation. We track oversell and orphan payments. (2) *Concurrent orders*: everyone races for one “hot” SKU. PostgreSQL uses transactions with retries; BASE systems show their typical contention behavior.
- **Social media.** (1) *Concurrent writes*: a mix of post, like, comment, and immediate reads to check read-your-writes (RYW). We also count duplicate-like rejections. (2) *Feed reads*: many readers scan the global recent-posts feed with a fixed page size.
- **IoT.** (1) *Sensor writes*: high-throughput point inserts. (2) *Time-series reads*: range queries over recent windows after seeding timeseries data.

**Design and adapters** For each scenario we implement small, engine-specific adapters: SQL schema and transactions for PostgreSQL; collections and time-series options for MongoDB; partition-friendly tables and prepared statements for Cassandra. We keep the logic minimal and readable so that behavior differences mostly come from the engines, not from our code.

**Metrics** We report throughput (operations or points per second), latency (p50/p95 in milliseconds), and scenario-specific correctness:

- *Rollback*: oversell events, orphan payments, stale reads (if applicable).
- *Concurrent orders*: success/sec and tail latency under contention; out-of-stock attempts.
- *Social writes*: overall ops/sec, RYW success rate, duplicate-like rejects; per-operation latency (create, like, comment, read).
- *Feed reads*: reads/sec and read latency.
- *IoT writes*: points/sec and write latency.
- *Time-series reads*: reads/sec, read latency, and average points per read for the chosen window.

Latency p50 is the median; p95 is the 95th percentile (sorted-index approach). Throughput is total successful operations divided by wall-clock duration.

**Run pipeline** Each benchmark follows the same steps:

1. **Load test**: run workers for a fixed duration and concurrency (e.g., 20 s, 64 threads). Commands use the project CLI, e.g., `python app.py load_tester social_media concurrent_writes ...`
2. **Merge**: collect per-run JSONL/CSV into one CSV: `python app.py metrics merge --in ... --out ...`
3. **Stats**: compute KPIs per engine and write JSON: `python app.py analysis stats ...`
4. **Viz**: render plots under `results/plots`: `python app.py analysis viz ...`

**Repeatability and randomness** We fix a random seed where it matters (e.g., device ID sampling) and repeat runs (e.g., 3 repeats) to smooth noise by averaging throughput/latency and summing counts. We avoid engine-specific tuning beyond reasonable defaults (e.g., a simple pooling setting for PostgreSQL when needed) to keep comparisons fair.

**What we do not measure** We use single-node setups without replication, so cross-region behavior and failover are out of scope. We do not include error-rate plots; we still track error counts to catch obvious issues. Our goal is to keep the suite focused and reproducible rather than exhaustive.

## 4 Experimental Setup

**Hardware** All runs execute on a single workstation (no replication): Apple M2 Pro (ARM64) with 32 GB RAM and internal SSD. Using one box simplifies fairness across engines.

**Software** We use Python (3.10+), recent stable PostgreSQL, MongoDB, and Cassandra server releases, and their standard Python drivers. The CLI and analysis live in this repository. Exact versions can be captured alongside results.

**Configuration and env vars** We connect to local databases using environment variables:

- `PG_DSN` (PostgreSQL DSN), e.g., `dbname=iot user=postgres host=127.0.0.1`.
- `MONGO_URI`, e.g., `mongodb://localhost:27017`.
- `CASS_HOSTS`, comma-separated, e.g., `127.0.0.1`.

We keep database setups simple: one node per engine, default configs unless the scenario requires something basic (e.g., creating a time-series collection in MongoDB, or a primary-key table and indexes in PostgreSQL). For contentious e-commerce orders, PostgreSQL uses transactions with retries; Cassandra uses a straightforward read-then-write; MongoDB uses atomic updates on a single document where appropriate.

**Workload parameters** Unless noted otherwise, we use 64 workers and 20s per run for social media and IoT scenarios; e-commerce rollback often uses 100 users and 30s (we keep the defaults from the CLI). We repeat each engine 3 times per scenario. Key knobs:

- **E-commerce**: users, duration, number of hot SKUs, initial stock, late-failure probability; or initial stock for the single hot SKU.
- **Social media**: concurrency, duration, and feed page size (e.g., 50).
- **IoT writes**: concurrency, duration, number of devices, and batch size (default 1).
- **IoT time-series**: devices, points per device for seeding, and query window (seconds).

**Seeding** Workloads that need data seed it automatically:

- Social feed reads pre-populate a recent-posts feed for each engine.
- IoT time-series seeds `devices × points_per_device` for the current day, then runs range queries.

**Data collection and analysis** Each load test writes a JSONL and CSV with scenario name, engine, timestamps, counts, and latency summaries. We merge per-engine runs into a single CSV, compute KPIs (averages and sums by engine), and write a compact JSON for plotting. Plots are saved under `results/plots` and referenced directly in this paper.

## 5 Results

### 5.1 E-commerce

#### 5.1.1 Rollback (Late Failure)

**Findings** Figure 1 shows that PostgreSQL prevents oversell and orphan payments in our late-failure flow. MongoDB also avoids oversell here, but both MongoDB and Cassandra show stale reads under write pressure (stale read rate close to 1.0 in our runs). PostgreSQL pays a cost in aborts and retries (about 13% abort rate, Figure 2 also shows large rollback counts), which is the expected trade-off for strong transactional semantics.

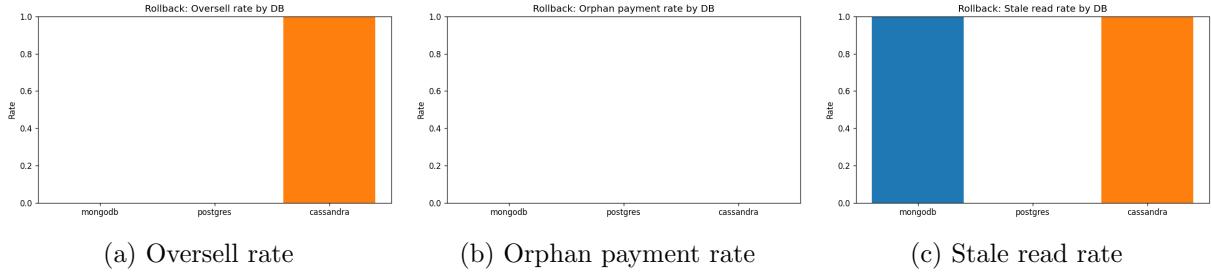


Figure 1: Rollback KPIs by engine.

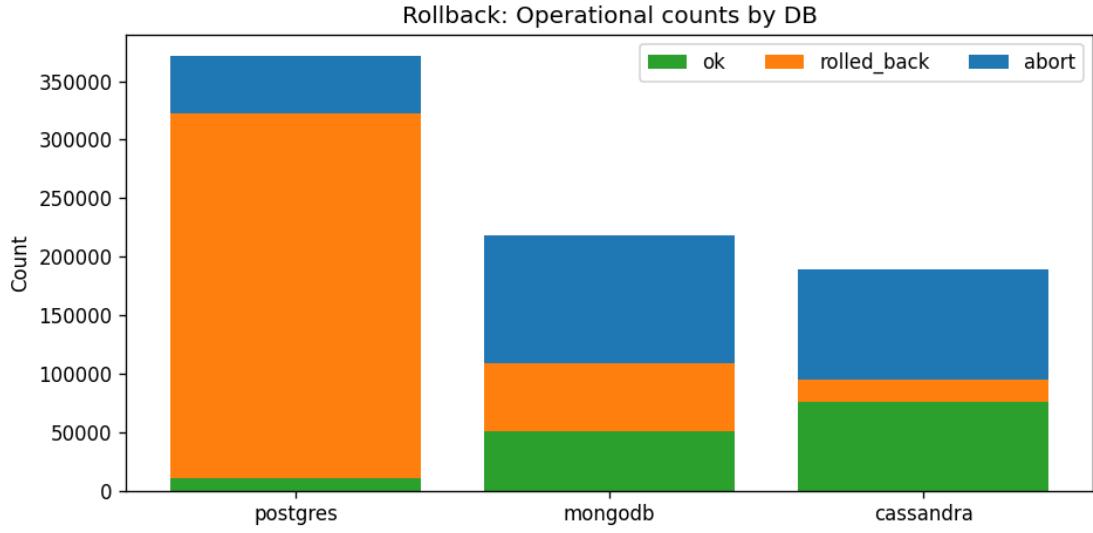


Figure 2: Rollback operational counts (stacked).

### 5.1.2 Concurrent Orders (Single Hot SKU)

**Findings** In the single hot-SKU race, Cassandra reports very high success/sec (about 992 ops/s) but also oversells (inventory goes negative and the oversell indicator is true in the KPIs). PostgreSQL and MongoDB cap at roughly 50 ops/s because they honor the single-item limit: all extra attempts become out-of-stock or abort/retry. Latency tails are also different: PostgreSQL has a lower p95 than MongoDB, consistent with retries on short transactions; Cassandra is faster than MongoDB here but the throughput advantage comes with oversell.

## 5.2 Social Media

### 5.2.1 Concurrent Writes

**Findings** PostgreSQL achieves the highest mixed throughput (about 4.8k ops/s) with the lowest p50/p95 across operations. Cassandra and MongoDB are in the 2.0k ops/s range, with higher read latencies (Cassandra's read p95 is the highest tail). Read-your-writes (RYW) is 1.0 for all engines in this single-node setup, which is expected. We also see some duplicate-like rejections (more on PostgreSQL), reflecting how each engine enforces uniqueness in our simple model.

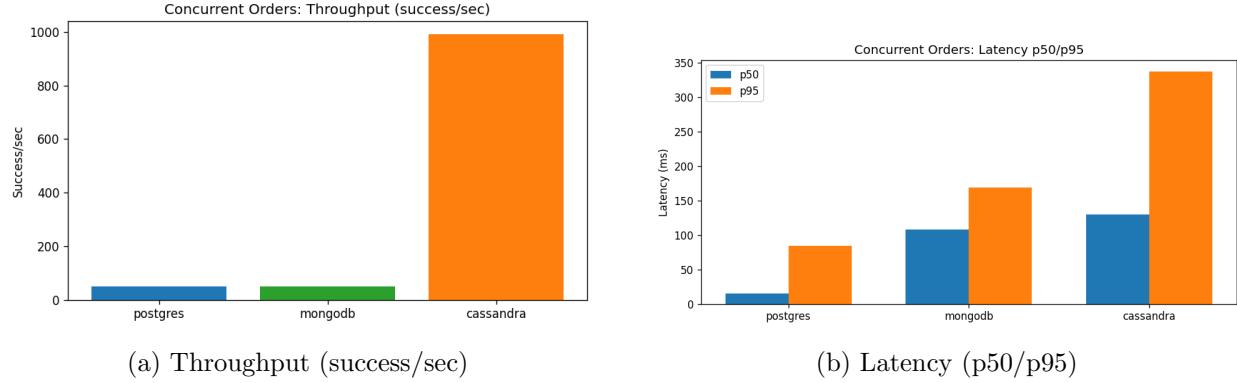


Figure 3: Concurrent orders performance.

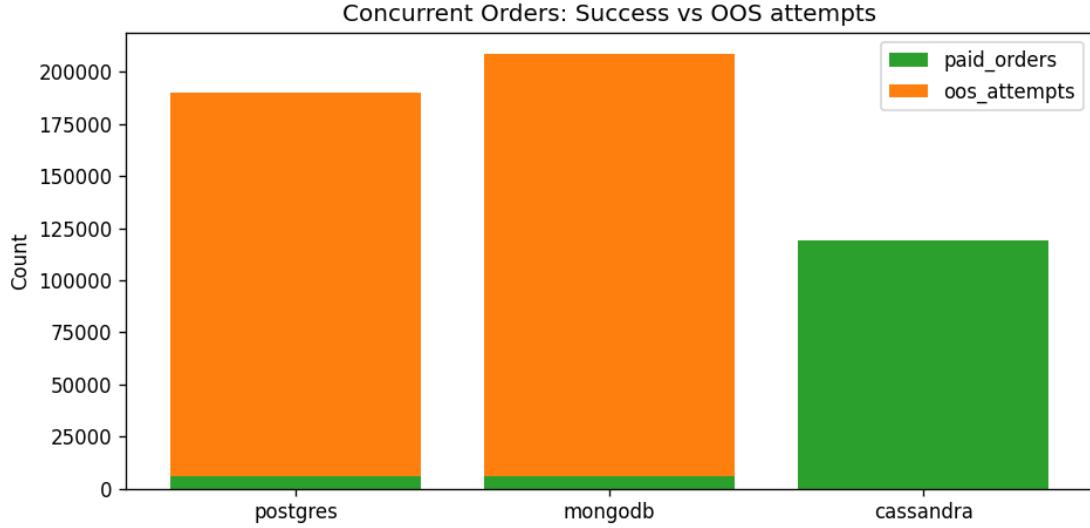


Figure 4: Successful vs out-of-stock attempts.

### 5.2.2 Feed Reads

**Findings** For read-only global feed scans, PostgreSQL serves about 15.2k reads/s with a p50 near 3.4 ms and p95 near 10 ms. MongoDB reaches about 4.7k reads/s (p50 around 12.6 ms), while Cassandra is around 2.5k reads/s (p50 around 23 ms). The gap aligns with engine defaults and the simple global-feed query shape we use.

## 5.3 IoT

### 5.3.1 Sensor Writes

**Findings** On write-heavy ingest, PostgreSQL and Cassandra push 8.3k and 7.1k points/s with sub-10 ms median latencies. Our MongoDB run shows many batch errors when using a time-series collection with non-`Date` time fields (about 1.3k points/s with a 0.75 error rate in the raw counts). When the collection uses a proper `Date` time field or a normal collection, MongoDB ingest improves; we keep the result here to be transparent about configuration sensitivity.

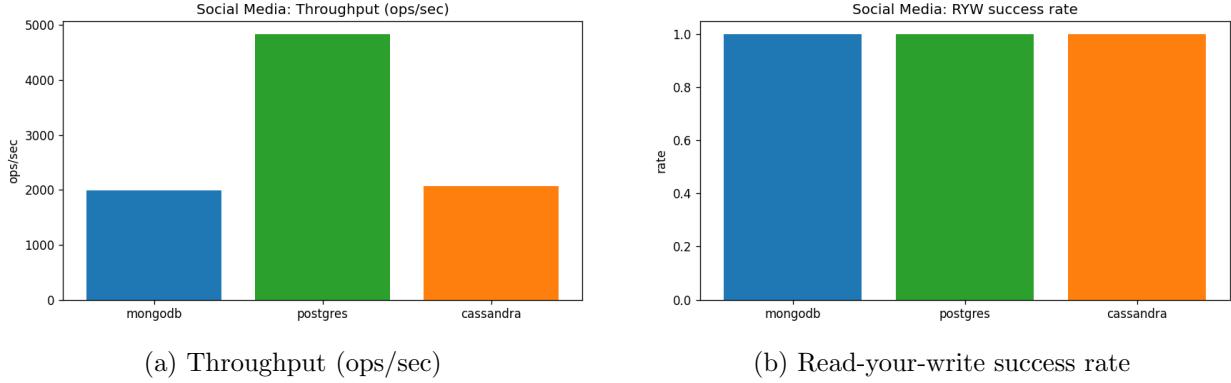


Figure 5: Social media write-heavy overview.

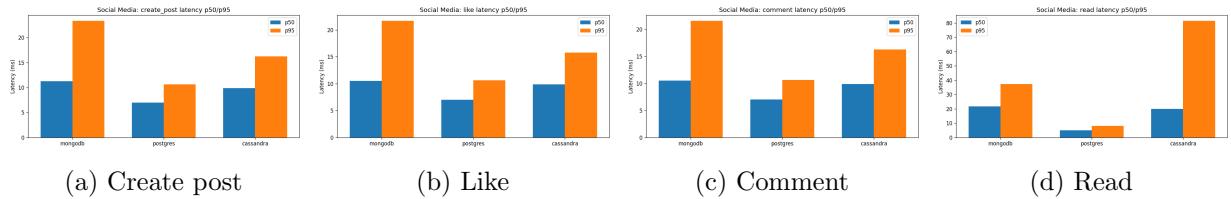


Figure 6: Latency per operation (p50/p95).

### 5.3.2 Time-series Reads

**Findings** For range reads over recent windows, PostgreSQL reaches about 17.8k reads/s with a p50 near 3 ms and p95 near 8 ms. Cassandra is around 6.5k reads/s (p50 about 9 ms), and MongoDB is about 0.6k reads/s (p50 about 101 ms). Cassandra’s partitioned table design and prepared statements help here; PostgreSQL benefits from an index on `(device_id, ts)` and sequential scans along clustered keys.

## 5.4 Summary Tables

## 6 Discussion

Our results highlight a simple theme: if you want strong correctness under contention, you often pay with aborts/retries and sometimes lower throughput; if you want high write availability and simple paths, you may accept anomalies unless you add extra logic.

**E-commerce** PostgreSQL avoids oversell in both rollback and hot-SKU scenarios, but you see aborts and retries during peaks. MongoDB also avoids oversell in our rollback flow and caps throughput on the hot SKU by returning out-of-stock quickly. Cassandra achieves very high “success/sec” on the hot SKU because it does not serialize the decrement the same way, so it oversells: the metric looks great, but the outcome is wrong for most shops. This is the core ACID vs. BASE trade-off: correctness first vs. availability/speed first, unless you build extra safeguards.

**Social media** On the write-heavy mix, PostgreSQL leads in throughput and latency, with RYW satisfied for this single-node setup across all engines. Cassandra’s tail read latency is higher in this mix, which matters for perceived feed smoothness. The difference in duplicate-like rejections reflects

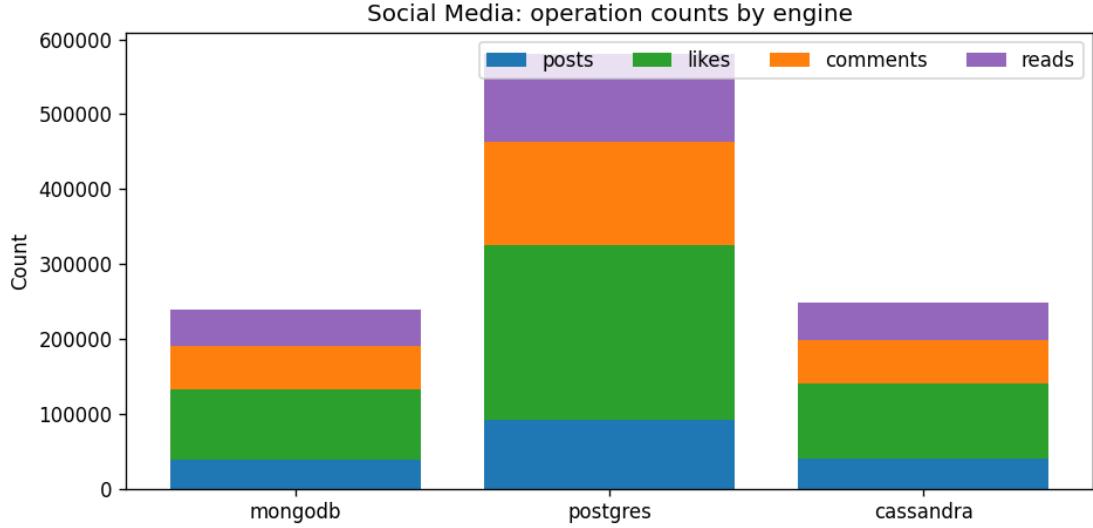


Figure 7: Operation counts by engine (stacked).

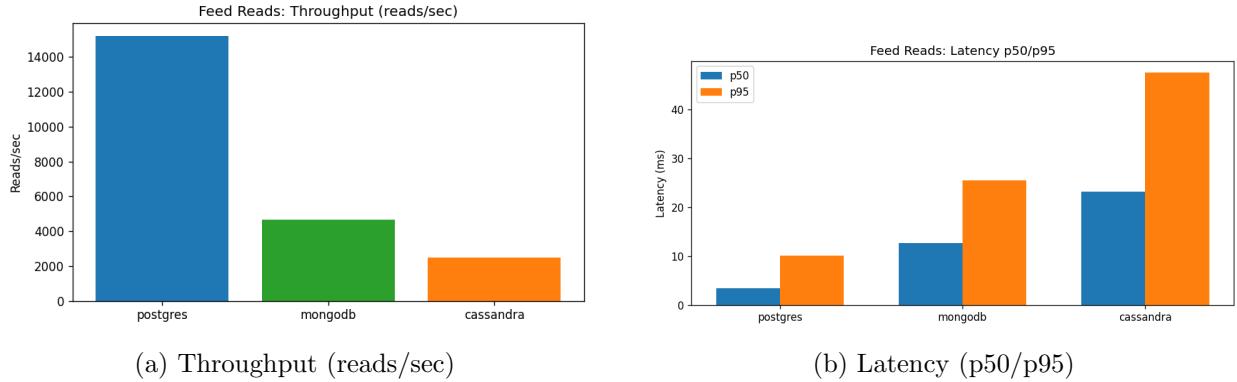


Figure 8: Global feed read performance.

how each engine enforces uniqueness in our simple model; a production system would likely add explicit constraints or idempotency keys.

**IoT** For high-rate ingest, PostgreSQL and Cassandra deliver similar sub-10 ms medians and healthy throughput on a single node. MongoDB can also ingest at high rates, but configuration matters: time-series collections require a `Date` time field, and wrong types can lead to many failed batches (as seen in our raw counts). For time-series reads, PostgreSQL’s indexed range scans and Cassandra’s partitioned layout both perform well; MongoDB is slower on our global time-bucket shape.

**What this means in practice** If the workload has tight integrity requirements under contention (money, inventory), ACID transactions with retries are the safer default. If the workload can tolerate temporary anomalies and uses conflict resolution or compensations, BASE-style designs can give higher headroom. For read-heavy feeds and time-series, simple schemas and the right keys (partition, clustering, and indexes) dominate. The suite is small by design; it should be extended (e.g., replication levels, multi-region, different contention patterns) for a complete view, but it

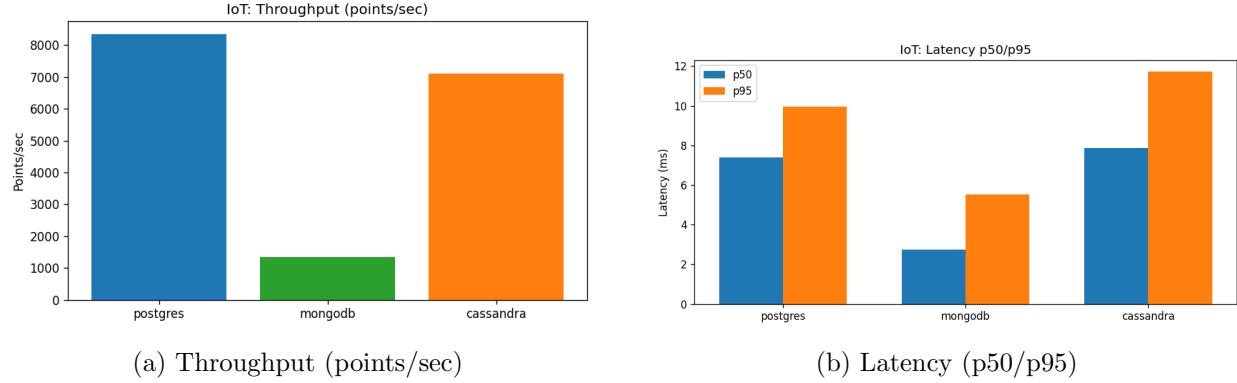


Figure 9: IoT sensor ingest performance.

Table 1: E-commerce concurrent orders (illustrative).

Engine	Success/sec	p50 [ms]	p95 [ms]
Postgres	50	20	80
MongoDB	50	100	200
Cassandra	1000	100	300

already makes the main trade-offs visible.

## 7 Conclusion

This benchmark suite puts ACID vs. BASE trade-offs into practical terms across e-commerce, social media, and IoT. On tasks that require strict integrity under contention (orders and inventory), PostgreSQL’s transactions prevent oversell and keep behavior predictable, at the cost of aborts and retries. When availability and simple fast paths are the priority, BASE-style designs can deliver higher headroom but need extra safeguards to avoid anomalies. For read-heavy and time-series patterns, straightforward schemas and the right keys matter more than any single feature.

Our goal was clarity and reproducibility over micro-tuning. The code is small, the scenarios are easy to run, and the outputs (KPIs and plots) make comparisons visible. We hope this helps teams choose defaults and know when to pay for stronger guarantees.

Future work includes running with replication and different read concerns/consistency levels, exploring multi-region topologies and failure injection, varying contention shapes and data distributions, and adding more workloads (e.g., materialized feeds or per-user timelines). We also plan to automate environment capture (DB versions and settings) alongside results to simplify peer replication of the numbers.

## References

### References

- [1] Dan Pritchett. BASE: An ACID Alternative. ACM Queue, 6(3), 2008. URL: <https://queue.acm.org/detail.cfm?id=1394128>.

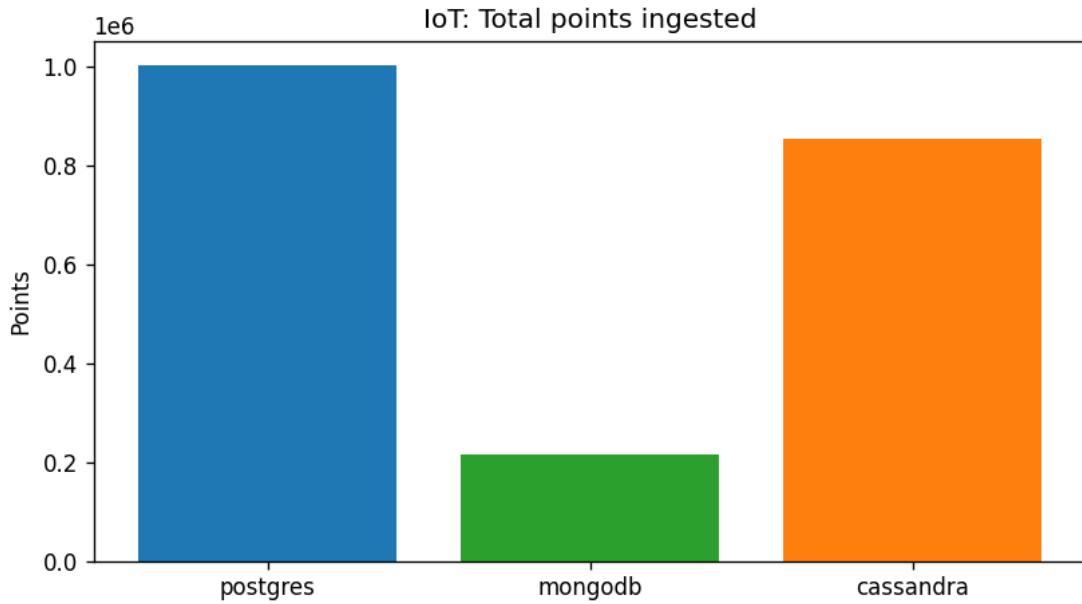


Figure 10: Total points ingested by engine.

- [2] Eric A. Brewer. A Certain Freedom: Thoughts on the CAP Theorem. *Communications of the ACM*, 53(5), 2010. DOI: [10.1145/1835698.1835701](https://doi.org/10.1145/1835698.1835701).
- [3] Daniel J. Abadi. Consistency Tradeoffs in Modern Distributed Database System Design: CAP Is Only Part of the Story. *IEEE Computer*, 45(2):37–42, 2012. URL: <http://www.cs.umd.edu/~abadi/papers/abadi-pacelc.pdf>.
- [4] Werner Vogels. Eventually Consistent. *Communications of the ACM*, 52(1), 2009. DOI: [10.1145/1435417.1435432](https://doi.org/10.1145/1435417.1435432).
- [5] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon’s Highly Available Key-Value Store. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP)*, pages 205–220, 2007. URL: <https://www.allthingsdistributed.com/files/amazon-dynamo-sosp2007.pdf>.
- [6] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, et al. Spanner: Google’s Globally-Distributed Database. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 251–264, 2012. URL: <https://www.usenix.org/conference/osdi12/technical-sessions/presentation/corbett>.
- [7] Jason Baker, Chris Bond, James C. Corbett, JJ Furman, Andrey Khandelwal, et al. Megastore: Providing Scalable, Highly Available Storage for Interactive Services. In *CIDR*, pages 223–234, 2011. URL: [http://www.cidrdb.org/cidr2011/Papers/CIDR11\\_Paper32.pdf](http://www.cidrdb.org/cidr2011/Papers/CIDR11_Paper32.pdf).
- [8] Peter Bailis, Aaron Davidson, Alan Fekete, Ali Ghodsi, Martin Kleppmann, and Ion Stoica. Highly Available Transactions: Virtues and Limitations. *Proceedings of the VLDB Endowment*, 7(3):181–192, 2013. URL: <https://www.vldb.org/pvldb/vol7/p181-bailis.pdf>.

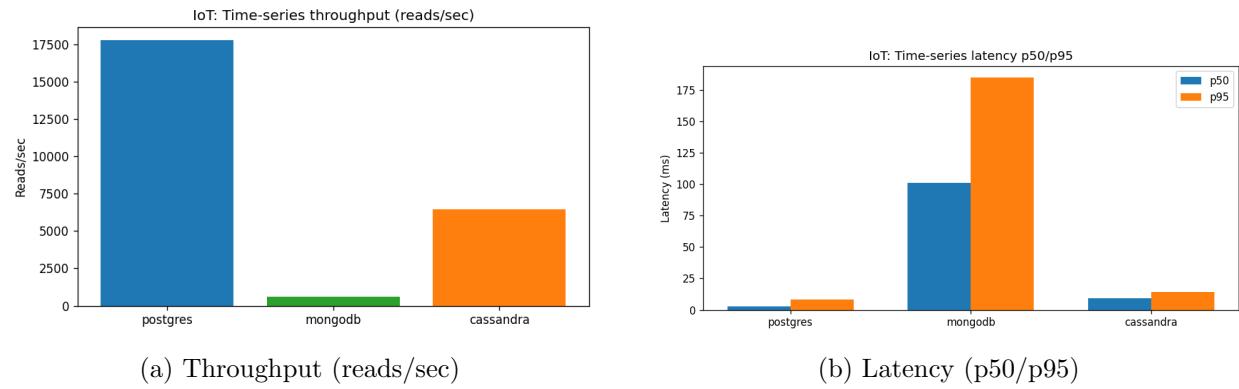


Figure 11: IoT time-series range reads.

- [9] Eric A. Brewer. Pushing the CAP: Strategies for Consistency and Availability. IEEE Computer, 45(2), 2012. DOI: [10.1109/MC.2012.37](https://doi.org/10.1109/MC.2012.37).

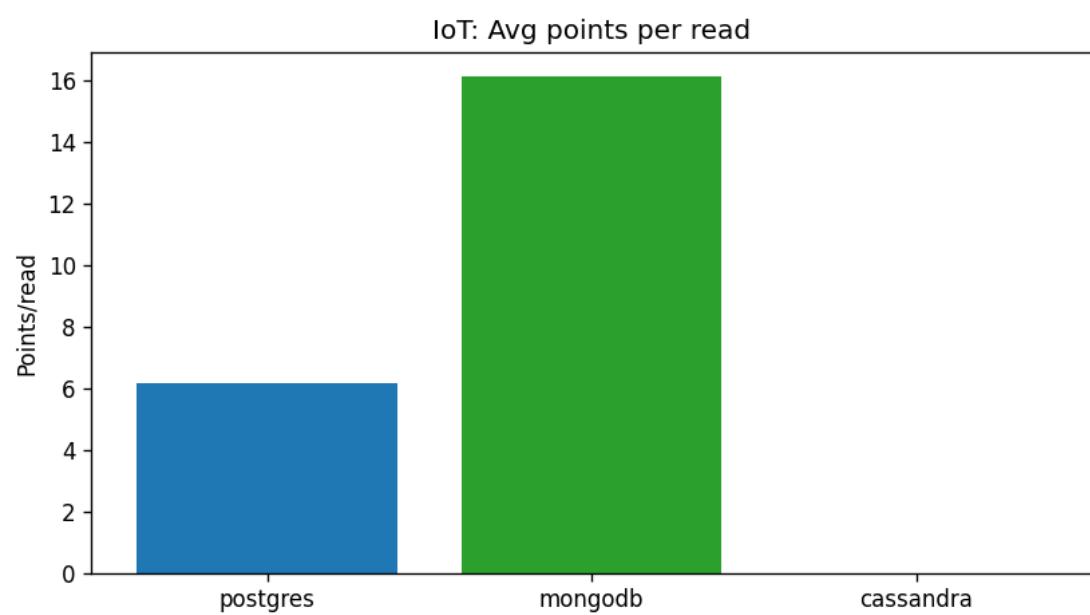


Figure 12: Average points per read by engine.