Research Square

Preprints are preliminary reports that have not undergone peer review.
They should not be considered conclusive, used to inform clinical practice,
or referenced by the media as validated information.

DriftShard: Drift-Aware Adaptive Sharding for Distributed Industrial IoT Time-Series Databases

Kavindu Karunarathne

k.karunarathne@yorksj.ac.uk

York St John University

Nalinda Somasiri

York St John University

Research Article

Keywords:

Posted Date: November 13th, 2026

DOI: https://doi.org/10.21203/rs.3.rs-11584427/v1

License: This work is licensed under a Creative Commons Attribution 4.0 International License.
Read Full License

Additional Declarations: No competing interests reported.

DriftShard: Drift-Aware Adaptive Sharding for Distributed Industrial IoT Time-Series Databases

Kavindu Karunarathne[1*] and Nalinda Somasiri[1]

[1*]Department of Computer Science and Data Science, York St John University London Campus, 6th Floor, Export Building, 1 Clove Crescent, London, E14 2BA, United Kingdom.

*Corresponding author(s). E-mail(s): k.karunarathne@yorksj.ac.uk;
Contributing authors: n.somasiri@yorksj.ac.uk;

Abstract

Distributed databases that store Industrial Internet of Things (IIoT) telemetry are usually sharded once, by device identifier or by time, and left in that state. The queries that reach them do not stay fixed: operator drill-downs, real-time monitoring and fleet-level analytics alternate over a working day, and each favours a different placement of the data. Re-sharding can recover lost performance, but moving millions of rows is expensive, and a system that re-shards on every change may spend more time migrating than it saves. This paper presents DriftShard, a framework that monitors the mix of query classes reaching a Citus cluster, detects regime changes with a total variation distance test, predicts the latency of every candidate layout with a gradient-boosted cost model, and migrates only when the predicted saving over a planning horizon exceeds the predicted migration time. DriftShard was implemented on PostgreSQL 16 with Citus 12.1 and evaluated with real migrations on a four-node cluster, using a synthetic sensor fleet (0.71 million rows) and the TON IoT telemetry collection (3.58 million rows). Across 40 randomised regime sequences, DriftShard was 42% and 30% faster than a static device-hash layout on the two datasets, and 31% and 53% faster than drifttriggered re-sharding that ignores migration cost. It did not match the best static layout chosen in hindsight, and its benefit depended on cost-model accuracy: with measured rather than predicted costs it improved by a further 6% and 13%. The results identify when adaptive sharding pays off and where learned cost models limit it.

Keywords: Distributed databases, Data partitioning, Sharding, Workload drift, Industrial Internet of Things, Learned cost models, Citus

1 Introduction

Industrial Internet of Things (IIoT) deployments produce a steady stream of timestamped readings from sensors, controllers and gateways. Storing this stream in a distributed relational database is attractive because it keeps SQL, transactions and secondary indexes available while spreading storage and query work across several nodes (Sisinni et al. 2018; Jensen et al. 2017). Citus, an extension that turns PostgreSQL into a sharded, shared-nothing system, is a common choice for this pattern (Cubukcu et al. 2021). In Citus, as in most sharded systems, the administrator chooses a distribution

column when a table is created. The rows are hashed on that column into a fixed number of shards, and every later query is routed according to the same choice.

For IIoT data the natural candidates are the device identifier, the time interval and the sensor family. Each candidate suits a different kind of question. Distributing by device keeps the full history of one machine on one shard, so a maintenance engineer who inspects a single pump touches one node. Distributing by time keeps a recent window together, so a monitoring dashboard that summarises the last hour across the whole plant reads from one or two shards rather than all of them. Distributing by sensor family places, for example, all thermostats together, which favours analytics that compare members of one family. When a query does not filter on the distribution column, the coordinator must send it to every shard and merge the partial results. On a small cluster this fan-out costs a few milliseconds per query; on a busy one it multiplies connection, planning and merge work.

The difficulty is that the workload reaching an IIoT database is not fixed. During a shift change, engineers drill into individual assets; during production, monitoring and ingest dominate; at the end of a period, analysts run family-level reports. These regimes recur, and each favours a different layout. A static layout must therefore be chosen as a compromise, and a poor choice is paid for on every query for as long as it remains in place. Workload-aware partitioning advisors address the initial choice (Curino et al. 2010; Nehme and Bruno 2011; Pavlo et al. 2012), and elastic partitioning systems react to load imbalance in transactional workloads (Taft et al. 2014; Serafini et al. 2016). Neither line of work addresses the question that arises when the analytical mix itself drifts: whether a change in the workload justifies the cost of physically re-sharding the table, given that the new regime may not last long.

This paper studies that question with DriftShard, a drift-aware re-sharding framework for Citus. DriftShard observes the classes of queries arriving at the coordinator in fixed-size windows, confirms a regime change when the total variation distance between the current and reference mixtures stays above a threshold, and then estimates the per-query latency of every candidate layout with a learned cost model. A migration is triggered only when the predicted saving over the expected remaining regime, plus a planning horizon priced with the long-run mixture, exceeds the predicted migration time multiplied by a safety factor. The migration itself is a complete physical rebuild: a new distributed table is created under the target layout, rows are copied with a distributed `INSERT ... SELECT`, indexes are rebuilt and the active table is swapped. The contributions of the paper are as follows.

1. A formulation of adaptive sharding for drifting analytical IIoT workloads, in which each re-sharding decision balances predicted latency savings over an uncertain regime length against a measured migration cost.
2. The DriftShard framework, which combines a total variation distance drift monitor with persistence, a gradient-boosted latency model with layout-aware features, a migration cost model fitted to measured rebuilds, and a planning decision rule.

3. An implementation on PostgreSQL 16 and Citus 12.1 with four hash layouts, seven IIoT query classes and four workload regimes, together with a benchmark protocol in which the cost model is never trained on the data size it is asked to predict.
4. An evaluation with real migrations on a live cluster and with trace-driven replay over 40 randomised regime sequences on two datasets, including ablations of the decision rule and of cost-model accuracy and sensitivity analyses of regime length, safety factor and detection threshold.

The evaluation gives a mixed but informative picture. DriftShard was consistently better than every policy that cannot see the future, including static device sharding, periodic re-optimisation, a one-off workload-aware placement and drift-triggered resharding that ignores migration cost. It was not better than the single static layout that, with hindsight, happened to suit all regimes, and on the TON IoT data that layout was within 10% of an oracle that migrates for free. The accuracy of the cost model, rather than drift detection or the decision rule, was the main limit on performance. These findings indicate when adaptive sharding is worth its cost and which component most needs improvement.

The rest of the paper is organised as follows. Section 2 reviews related work. Section 3 defines the problem. Section 4 describes DriftShard. Section 5 presents the experimental setup and Section 6 the results. Section 7 discusses their implications, Section 8 considers threats to validity and Section 9 concludes.

2 Related work

2.1 Workload-driven partitioning

Automated physical design has long treated partitioning as an optimisation over a known workload. Agrawal et al. (2004) integrated horizontal and vertical partitioning into a physical design advisor for a single-node system. For shared-nothing systems, Nehme and Bruno (2011) proposed a partitioning advisor driven by the query optimiser's own cost estimates, and Pavlo et al. (2012) searched for partitioning schemes that account for access skew in parallel OLTP systems. Schism modelled a transactional workload as a graph over tuples and used graph partitioning to minimise distributed transactions (Curino et al. 2010). More recently, Hilprecht et al. (2020) trained a reinforcement learning agent to choose partitioning schemes for cloud data warehouses, learning from cost estimates and then refining with measured runtimes. These systems produce a layout for a representative workload. They do not decide when the representative workload has changed enough to justify building a new layout, which is the question this paper addresses.

2.2 Online repartitioning and live migration

Elastic transactional systems move data while the system is running. E-Store monitors tuple-level access frequency and redistributes hot tuples in a main-memory OLTP system (Taft et al. 2014), and Clay extends this to multi-table schemas by moving groups of co-accessed tuples together (Serafini et al. 2016). Squall provides the mechanism for fine-

grained live reconfiguration in partitioned main-memory databases (Elmore et al. 2015), and Albatross migrates tenants between nodes in shared-storage cloud databases with little interruption (Das et al. 2011). This body of work is concerned mainly with load balance under transactional skew and with the mechanics of moving data with minimal disruption. DriftShard is complementary. It targets read-heavy analytical workloads whose query mix, rather than their load, changes; it chooses among whole-table layouts; and it treats migration cost as a quantity to be predicted and weighed against benefit rather than minimised by a better mechanism. A livemigration mechanism such as Squall could reduce the migration cost that DriftShard trades against.

2.3 Learned database tuning and cost models

Machine learning has been applied to many self-managing database functions, including knob tuning, query optimisation, indexing and cardinality estimation (Zhou et al. 2022; Pavlo et al. 2017). OtterTune reuses measurements from earlier tuning sessions to configure new deployments (Van Aken et al. 2017). Neo and Bao learn query plans from execution feedback (Marcus et al. 2019, 2021), and learned index structures replace traditional index nodes with models of the key distribution (Kraska et al. 2018). Evaluations of learned optimisers have shown that apparent gains depend strongly on how training and test workloads are separated (Lehmann et al. 2024). DriftShard uses a learned model in a narrower role, as a predictor of per-class latency under layouts that are not currently deployed, and its evaluation follows the same caution by testing the model only on a data size it has not seen.

2.4 Drift detection

Concept drift research studies changes in the distribution of a data stream and the adaptation of models to them (Gama et al. 2014; Lu et al. 2019). Detectors range from fixed-window statistical tests to adaptive windows such as ADWIN, which grows and shrinks a window according to observed change (Bifet and Gavalda` 2007). In DriftShard the monitored distribution is the categorical mix of query classes, so total variation distance between successive windows provides a bounded and directly interpretable statistic. The persistence requirement used here is a simple guard against reacting to a single unusual window.

2.5 IIoT and time-series storage

Purpose-built time-series stores such as Gorilla compress and serve recent metrics at high throughput (Pelkonen et al. 2015), and the survey by Jensen et al. (2017) describes the range of time-series management systems. Many industrial deployments nevertheless keep telemetry in relational systems so that it can be joined with asset and maintenance records. The authors' earlier work examined learned index selection and adaptive indexing for IIoT workloads in PostgreSQL (Karunarathne 2026; Karunarathne and Somasiri 2026), and a rule-based framework that adjusts indexing, partitioning and stream-processing settings in a PostgreSQL, Kafka and Flink pipeline to lower the latency

of real-time analytics (Karunarathne et al. 2026). Those studies adapted the physical design within a single database node, where changes are cheap enough to apply as soon as a rule fires. The present study moves to data placement across nodes, where a change is far more expensive and the cost of acting must therefore be modelled explicitly.

3 Problem formulation

Consider a telemetry table with n rows distributed over a cluster. Let L be a finite set of candidate layouts, each defined by a distribution column, a shard count and, optionally, a native partitioning scheme. Let Q be a set of query classes, where each class is a parameterised SQL template. A workload is a stream of queries $x_1, x_2, \ldots$, each belonging to a class $q(x_t) \in Q$.

Let $c_\ell(q)$ denote the expected latency of a query of class q when the table has layout ℓ. For a mixture $\mathbf{p}$ over Q, the expected per-query latency of layout ℓ is

$$C_\ell(\mathbf{p}) = \sum_{q \in Q} p_q \, c_\ell(q). \tag{1}$$

The workload is assumed to move between regimes, each of which has a stable mixture $\mathbf{p}^{(r)}$ and an unknown duration. Changing the layout from ℓ to ℓ' takes time $M(\ell \to \ell', n)$, during which the table is rebuilt.

A re-sharding policy chooses a layout sequence $\ell_1, \ell_2, \ldots$ using only the queries observed so far. Its cost over a stream of length T is

$$J = \sum_{t=1}^{T} \mathrm{lat}_{\ell_t}(x_t) + \sum_{t:\ell_{t+1} \neq \ell_t} M(\ell_t \to \ell_{t+1}, n), \tag{2}$$

that is, the end-to-end time to serve the stream including migrations. Two reference points bound the problem. A static policy never migrates and pays $C_\ell(\mathbf{p}^{(r)})$ in every regime. An oracle that knows each regime in advance and migrates for free pays $\min_\ell C_\ell(\mathbf{p}^{(r)})$ in every regime; it is not achievable but bounds what any policy can gain.

If a regime with mixture $\mathbf{p}$ will last H more queries, moving from ℓ to ℓ' pays off only if

$$\big(C_\ell(\mathbf{p}) - C_{\ell'}(\mathbf{p})\big)\, H > M(\ell \to \ell', n). \tag{3}$$

Three quantities in this inequality are unknown at decision time: the new mixture $\mathbf{p}$, the remaining length H, and the costs C and M themselves. DriftShard estimates each of them.

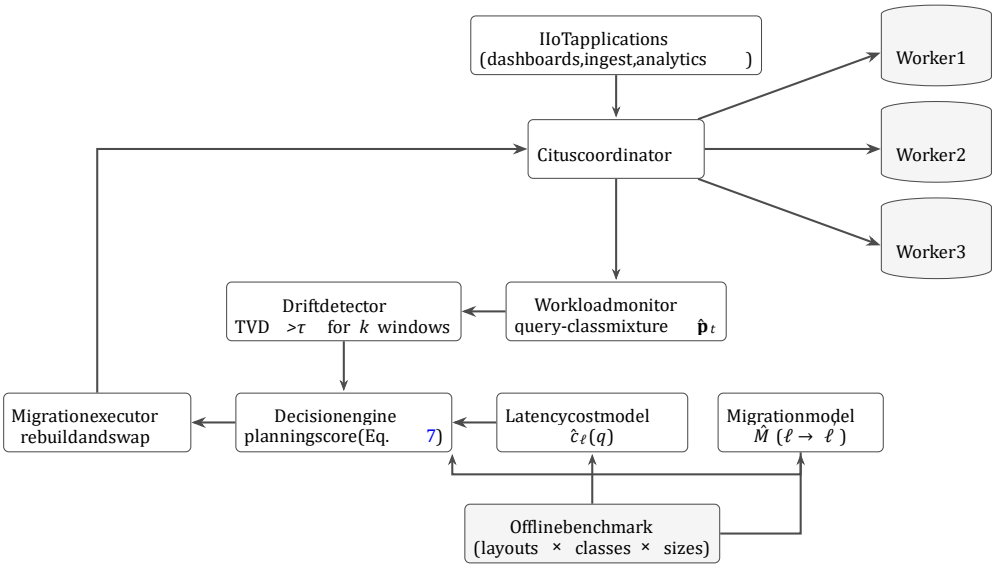


Fig. 1 DriftShard components. The monitor and decision path run beside the Citus coordinator; the cost models are trained offline from a layout benchmark

4 The DriftShard framework

4.1 Overview

Fig. 1 shows the components of DriftShard and their place beside the Citus coordinator. The workload monitor classifies each incoming query and maintains the mixture of the current window. The drift detector compares that mixture with a reference. When drift is confirmed, the decision engine scores every layout with the latency cost model and the migration cost model. If a different layout wins, the migration executor rebuilds the table and swaps it in. Both cost models are trained offline from a benchmark of the candidate layouts.

4.2 Workload monitor and drift detection

The monitor assigns every query to one of the classes in Q. In the implementation each class corresponds to an application template, so classification is a lookup; in a production deployment it could be done by normalising the SQL text and hashing the resulting template. After every window of w queries the monitor computes the empirical mixture $\hat{\mathbf{p}}_t$.

The detector keeps a reference mixture $\mathbf{r}$ and computes the total variation distance

$$D_t = \tfrac{1}{2} \, \|\hat{\mathbf{p}}_t - \mathbf{r}\|_1, \tag{4}$$

which lies between 0 and 1. Drift is confirmed when $D_t > \tau$ for k consecutive windows. The reference is then replaced by the mean of those k windows, so that it represents the new regime. While the workload is stable, the reference is updated slowly as $\mathbf{r} \leftarrow 0.8\mathbf{r} + 0.2\hat{\mathbf{p}}_t$, which absorbs gradual change without triggering a decision. The experiments use $w = 50$, $\tau = 0.25$ and $k = 2$.

4.3 Latency cost model

The decision engine must estimate $c_\ell(q)$ for layouts that are not deployed, so the model cannot rely on runtime statistics of the current table. Each prediction is therefore made from features that can be computed from the layout definition, the query class and its parameters, and the data size:

- the expected number of shards the query touches, derived from the predicates on the distribution column (for k distinct hash keys over S shards, $S\,(1 - (1 - 1/S)^k)$); the number of shards that hold data, which is smaller than S when the distribution column has few distinct values; the time window length, whether the query fixes a device or a sensor family, and an estimated selectivity; the fraction of time partitions that survive pruning for partitioned layouts; whether the query groups by the distribution column, so that aggregation can complete on the workers; whether the query is an insert, the logarithm of the row count, the number of workers, and one-hot indicators for layout and class.

A LightGBM regressor (Ke et al. 2017) is trained on the logarithm of measured latency, which stabilises the variance of a heavy-tailed target. Because Eq. (2) depends on mean latency, predictions on the log scale are converted back with Duan's smearing estimator (Duan 1983), applied separately for each query class:

$$\hat{c}_\ell(q) = \exp\big(f(\mathbf{z})\big)\,\frac{1}{|I_q|}\sum_{i \in I_q}\exp(e_i) - 1 \quad , \tag{5}$$

where f is the fitted model, $\mathbf{z}$ the feature vector and e_i the training residuals of class q. Without this correction the model predicts typical rather than expected latency, and the difference is large for classes with long tails. The table $\hat{c}_\ell(q)$ is computed once for all layouts and classes by averaging predictions over 40 sampled parameter settings per class.

4.4 Migration cost model

The time to rebuild the table under a new layout is modelled as

$$\hat{M}(\ell \to \ell', n) = a + b\,\frac{n}{10^6} + c\,\mathbf{1}[\ell' \text{ is partitioned}], \tag{6}$$

with non-negative coefficients fitted by least squares to the migrations recorded during the benchmark. The partition term captures the extra cost of creating and indexing many partition shards.

Algorithm 1 DriftShard decision step, executed after every window of w queries

Require: window classes W, current layout ℓ, row count n, cost table $\hat{c}$, migration model $\hat{M}$

$\hat{\mathbf{p}} \leftarrow$ empirical mixture of W; append $\hat{\mathbf{p}}$ to history; $\bar{\mathbf{p}} \leftarrow$ mean of history
if not Detector.Update($\hat{\mathbf{p}}$) and not first window **then**
    return no action
end if
$\hat{\mathbf{p}}_{\text{cur}} \leftarrow \mathbf{r}$ (detector reference); record drift position
$\hat{H} \leftarrow \frac{1}{2} H_0 + \frac{1}{2}$ mean gap; $\hat{H}_{\text{rem}} \leftarrow \max(0, \hat{H} - kw)$
for all $\ell' \in \mathsf{L}$ **do**
    $S(\ell') \leftarrow \left(C_{\ell'}(\hat{\mathbf{p}}_{\text{cur}})\, \hat{H}_{\text{rem}} + C_{\ell'}(\bar{\mathbf{p}})\, H_f\right)/1000$
    if $\ell' \neq \ell$ **then** $S(\ell') \leftarrow S(\ell') + \kappa \hat{M}(\ell \to \ell', n)$
    end if
end for
$\ell^* \leftarrow \operatorname{argmin}_{\ell'} S(\ell')$
if $\ell^* \neq \ell$ **then return** migrate to ℓ^*
else return no action
end if

4.5 Decision rule

When drift is confirmed, a myopic version of the rule applies Eq. (3) directly: it selects the layout that is best for the new mixture and migrates if the predicted saving over the expected remaining regime exceeds $\kappa \hat{M}$, where κ is a safety factor. This rule ignores the fact that regimes recur. A layout that is slightly worse for the current regime but much better for the others may be the better destination.

DriftShard therefore scores every layout ℓ', including the current one, over two horizons:

$$S(\ell') = C_{\ell'}(\hat{\mathbf{p}}_{\text{cur}})\hat{H}_{\text{rem}} + C_{\ell'}(\bar{\mathbf{p}})H_f + \kappa \hat{M}(\ell \to \ell', n)\mathbf{1}[\ell' \neq \ell], \qquad (7)$$

where costs are in milliseconds and scores in seconds. Here $\hat{\mathbf{p}}_{\text{cur}}$ is the confirmed mixture of the new regime and $\bar{\mathbf{p}}$ is the mean of all windows observed so far, which serves as the

long-run mixture. $\hat{H}_{\text{rem}}$ is the expected remaining length of the current regime, and H_f is a planning horizon over which the long-run mixture is assumed to hold. The layout with the lowest score is selected, and a migration happens only if that layout differs from the current one.

The expected regime length is learned online. Starting from a prior H_0, DriftShard records the positions of confirmed drifts and sets $\hat{H} = \frac{1}{2}H_0 + \frac{1}{2}\bar{g}_{-}$ $_{-}$, where $\bar{g}$ is the mean gap between drifts; shrinking towards the prior prevents a single short regime from dominating the estimate. Since detection lags the true change by about k windows, $\hat{H}_{\text{rem}} = \max(0, \hat{H} - kw)$. The experiments use $H_0 = 800$ queries, $H_f = 1600$ queries and $\kappa = 1$. Algorithm 1 summarises the procedure.

The decision step evaluates a small matrix product over |L| × |Q| entries once per window, and its overhead was below measurement resolution in all experiments.

4.6 Migration executor

A migration creates a new table with the target layout, distributes it with create distributed _table, copies all rows from the active table with a distributed INSERT ... SELECT, builds the secondary indexes, runs ANALYZE, drops the old table and directs subsequent queries to the new one. Queries are not served during this rebuild, so the measured migration time is charged in full to the policy. This is a conservative choice; Section 7 returns to it.

5 Experimental setup

5.1 Cluster and software

All experiments ran on a cloud virtual machine with 2 vCPUs and 7 GB of memory.
The cluster consisted of one Citus coordinator and three workers, each a separate PostgreSQL 16 instance on its own port, with Citus 12.1.5 built from source. Every table used 32 shards. The instances were configured with 256 MB of shared buffers and 32 MB of work memory; durability settings were relaxed (fsync off, asynchronous commit) because the experiments measure query and rebuild time rather than recovery. The framework was written in Python and drives the cluster through the psycopg driver.

5.2 Datasets

Synthetic fleet. A generator produced a fleet of 140 devices in seven sensor families (fridge, GPS, garage, Modbus, motion, thermostat and weather), each reporting every 120 s for seven days, giving 705,600 rows over 168 hourly buckets. Readings follow a daily cycle with a random walk, and injected bursts of anomalous readings label 1.8% of rows.

TON IoT. The processed telemetry streams of the TON IoT collection (Alsaedi et al. 2020) contain readings from seven IoT and IIoT sensors with attack labels. After parsing timestamps, 3,580,178 readings remained, covering 30 days (708 hourly buckets), of

which 14.5% are labelled as attacks. Each stream comes from a single physical sensor, which would leave only seven distinct device identifiers. To emulate a fleet, each stream was split into 40 virtual devices by assigning its readings to devices in round-robin order, giving 280 devices. This step preserves the timing and value distribution of the original data but not true per-device behaviour, and it is reported as a limitation.

Both datasets share one schema: device identifier, sensor family, timestamp, hour bucket, two numeric readings, a label and an attack type. The hour bucket is maintained by the application and serves as the routing key for time-based distribution.

5.3 Layouts and query classes

Table 1 lists the four candidate layouts. All use the same three secondary indexes, on (device, timestamp), (timestamp) and (family, timestamp), so that differences arise from placement rather than indexing. Table 2 lists the seven query classes. Because Citus prunes hash-distributed shards only on equality and IN predicates, every time-bounded

Table 1 Candidate sharding layouts

Layout	Distribution	Expected strength
DEV	hash on device identifier	single-device history
TIME	hash on hour bucket	recent windows, time-local ingest
DEV-TPART	hash on device identifier, range partitions on timestamp	device history with time pruning
TYPE	hash on sensor family	family-level analytics

Table 2 IIoT query classes

Class	Description	Window
Q1	hourly aggregates for one device	7 days
Q2	per-device aggregates across the fleet	1 hour
Q3	per-family statistics and anomaly counts	1 day
Q4	hourly trend for one sensor family	7 days
Q5	labelled anomalies, ordered by time, limit 500	6 hours
Q6	top 20 devices of one family by mean reading	6 hours
W1	multi-row insert of 200 readings	1 hour

Table 3 Query-class mixtures of the workload regimes

Regime	Q1	Q2	Q3	Q4	Q5	Q6	W1
Device operations	0.70	0.05			0.05	0.05	0.15
Real-time monitoring		0.35	0.10		0.25		0.30
Fleet analytics	0.10		0.15	0.40		0.35	
Mixed	1/7	1/7	1/7	1/7	1/7	1/7	1/7

query also carries a `time_bucket = ANY(...)` predicate derived from its window. An application can compute this predicate directly from the time range it requests.

5.4 Workload regimes

Four regimes were defined as mixtures over the query classes (Table 3). *Device operations* represents engineers inspecting individual assets, *real-time monitoring* represents control-room dashboards with heavy ingest, and *fleet analytics* represents comparisons within sensor families. A uniform *mixed* regime is used only for reference costs. Each workload stream moves between regimes, with phase lengths drawn uniformly within 25% of a nominal length.

5.5 Policies

DriftShard was compared with the following policies. All policies except the static ones start on DEV, which matches the common practice of distributing IIoT tables by device.

Static DEV, TIME, DEV-TPART and TYPE never migrate and start on their own layout at no cost.

Best-initial observes the first window and moves once to the layout predicted to be best for it, in the spirit of workload-driven advisors (Curino et al. 2010; Nehme and Bruno 2011).

Periodic re-optimises every 200 queries for the latest window and ignores migration cost.

Greedy-drift uses the same drift detector as DriftShard and moves to the predicted best layout on every confirmed drift, ignoring migration cost.

DriftShard-myopic applies the break-even test of Eq. (3) to the current regime only.

DriftShard (true costs) is DriftShard with measured mean latencies in place of the learned cost table, which isolates the effect of cost-model error.

Oracle knows every regime change and the measured costs and migrates at no charge. It is a lower bound, not a competitor.

5.6 Protocol

Benchmark. For each data size, the table was deployed under DEV and then migrated in turn to TIME, DEV-TPART, TYPE and back to DEV, recording every migration. After each migration the table was vacuumed and analysed, each query class was warmed up three times, and the classes were then executed in shuffled order for 15 repetitions (synthetic) or 25 repetitions (TON IoT). The synthetic fleet was measured at 50% and 100% of its devices and TON IoT at 25%, 50% and 100%. In total, 840 and 2,100 query latencies and 8 and 12 migrations were recorded.

Cost-model training. The model used by the policies was trained only on the smaller data sizes and applied to the full size. Its accuracy was also assessed by leaveone-size-out validation, reporting the median absolute percentage error, the Spearman rank correlation between predicted and measured layout orderings within each class, and the fraction of regimes for which the predicted best layout matches the measured one.

Live replay. A stream of 1,568 queries over five phases (device operations, realtime, fleet, real-time, device operations) was executed against the full-size table for five policies, with every query and migration run for real.

Trace-driven replay. To evaluate more workloads than the live cluster allows, a replay executor draws each query latency from the measured benchmark distribution for its layout and class, and each migration time from the measured migrations (or from the migration model for pairs not measured). Each policy was run on 20 seeds of the fixed five-phase schedule and on 40 randomised schedules of eight phases, in which regimes recur in random order without immediate repetition. The fidelity of the replay was checked against the live runs.

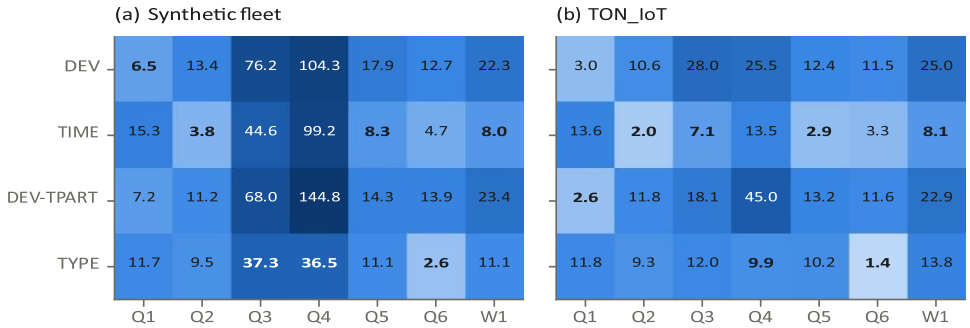


Fig. 2 Median latency (ms) per layout and query class at full data size. Bold marks the fastest layout for each class

Sensitivity. The nominal phase length was varied from 50 to 3,200 queries, κ from 0.25 to 4 and τ from 0.10 to 0.50, using 10 randomised schedules per setting.

The primary metric is end-to-end time, Eq. (2). Query time, migration time, the number of migrations and the 95th percentile latency are also reported, with 95% confidence intervals across schedules.

6 Results

6.1 Layout trade-offs

Fig. 2 shows median latency for every layout and query class at full size. The pattern matches the intended design of the classes. For single-device history (Q1), DEV and DEV-TPART answered from one shard, in 6.5 and 7.2 ms on the synthetic fleet, whereas TIME had to visit almost every shard and took 15.3 ms. For the one-hour fleet window (Q2), TIME touched one or two shards and took 3.8 ms, against 13.4 ms for DEV. TYPE was fastest for family-level queries: 36.5 ms for the seven-day family trend (Q4) against 104.3 ms for DEV, and 2.6 ms for the family top-20 query (Q6). Time-local ingest (W1) was almost three times faster under TIME than under DEV, because the 200 rows of a batch share one hour bucket but span many devices. TON IoT showed the same ordering, with TIME stronger on aggregate queries over windows: Q3 took 7.1 ms under TIME and 28.0 ms under DEV.

Fig. 3 combines these costs with the regime mixtures using mean latency, as in Eq. (1). No single layout was best for every regime on either dataset. On the synthetic fleet, the real-time regime was served best by TIME (12.5 ms per query) and fleet analytics by TYPE (22.4 ms), while the three layouts other than TIME were within 6% of each other for device operations. The spread between the best and worst layout reached a factor of 3.3 for fleet analytics. On TON IoT, DEV-TPART was best for device operations (9.2 ms against about 12.5 ms for the others), TIME for real-time monitoring (8.2 ms against 39.3 ms for DEV), and TIME and TYPE were nearly equal for fleet analytics (21.7 and 21.4 ms). Two layouts were robust: TYPE on the synthetic fleet was never more than 8% worse than the best layout in any regime, and TIME

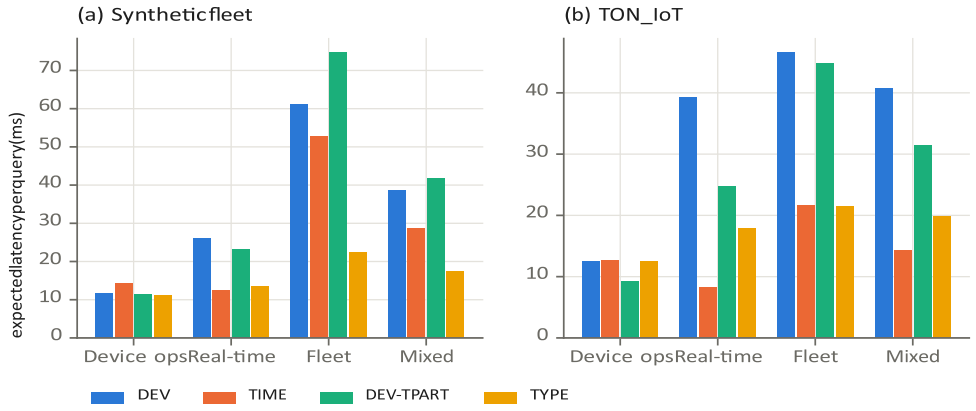


Fig. 3 Expected latency per query for each regime and layout, computed from measured mean latencies and the regime mixtures

on TON IoT was at most 37% worse. This robustness strongly shaped the end-to-end results that follow.

6.2 Migration cost

Fig. 4 shows the measured migration times. At full size, a rebuild took between 2.4 and 4.9 s for the 0.71 million synthetic rows and between 10.8 and 16.6 s for the 3.6 million TON IoT rows, moving roughly 130 MB and 520 to 580 MB of table and index data respectively. Index construction accounted for about half of the time, and the partitioned DEV-TPART target was the most expensive in every case because each partition is itself a distributed table with its own shards and indexes. The fitted migration models gave 3.3 and 3.2 s per million rows, with partition surcharges of 1.9 and 4.2 s. At full TON IoT size, one migration therefore costs as much as several hundred queries, which sets the minimum regime length for which re-sharding can pay off.

6.3 Cost-model accuracy

Table 4 reports the accuracy of the latency model on the full data size when trained on the smaller sizes. On the synthetic fleet the LightGBM model had a median absolute percentage error of 22.9%, ordered layouts within each class with a Spearman correlation of 0.94, and identified the best layout for three of the four regimes. On TON IoT the median error rose to 48.7% and the best layout was right for only two regimes. A ridge regression on the same features ranked layouts less well on both datasets. Fig. 5 shows that the model's errors are not random: it underestimates the slowest classes, particularly heavy-tailed scans under DEV, and it compresses the differences between layouts. Near-ties between layouts, such as TIME and TYPE for TON IoT fleet analytics, make the regime-level choice fragile even when per-class errors are moderate.

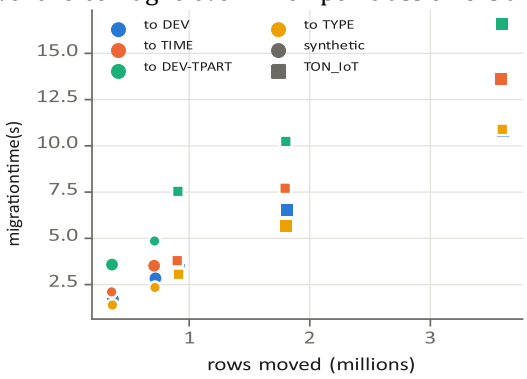


Fig. 4 Measured re-sharding time against rows moved, by target layout and dataset

Table 4 Latency model accuracy on the full data size, trained on smaller sizes only

Dataset	Model	Median APE	Mean APE	Spearman ρ	Best layout correct
Synthetic	LightGBM	22.9%	26.1%	0.94	3 of 4
	Ridge	21.0%	24.3%	0.79	3 of 4
TON IoT	LightGBM	48.7%	71.8%	0.86	2 of 4
	Ridge	64.2%	94.5%	0.80	2 of 4

Table 5 Live cluster results for a 1,568-query stream over five fleet times. Times in seconds

	Synthetic fleet				TON IoT			
Policy	Total	Query	Migr.	No.	Total	Query	Migr.	No.
Static DEV	40.1	40.1	0.0	0	36.4	36.4	0.0	0
Static TYPE	27.6	27.6	0.0	0	28.2	28.2	0.0	0
Greedy-drift	37.1	25.3	11.8	4	70.2	21.8	48.3	4
DriftShard-myopic	31.8	26.2	5.7	2	34.9	34.9	0.0	0
DriftShard	34.7	32.3	2.3	1	41.0	30.0	11.0	1

6.4 End-to-end results on the live cluster

Table 5 reports the live runs, in which every query and migration was executed on the cluster. On the synthetic fleet, DriftShard completed the stream in 34.7 s against 40.1 s for static DEV, a reduction of 13.5%, with a single migration to TYPE that took 2.3 s. Fig. 6 shows how that migration cut latency during the fleet-analytics phase and kept it low for the rest of the stream. Greedy-drift achieved the lowest query time of the adaptive policies (25.3 s) but spent 11.8 s on four migrations and finished at 37.1 s.

The TON IoT live run was less favourable. Greedy-drift spent 48.3 s on four migrations of 3.6 million rows and took almost twice as long as static DEV. DriftShard

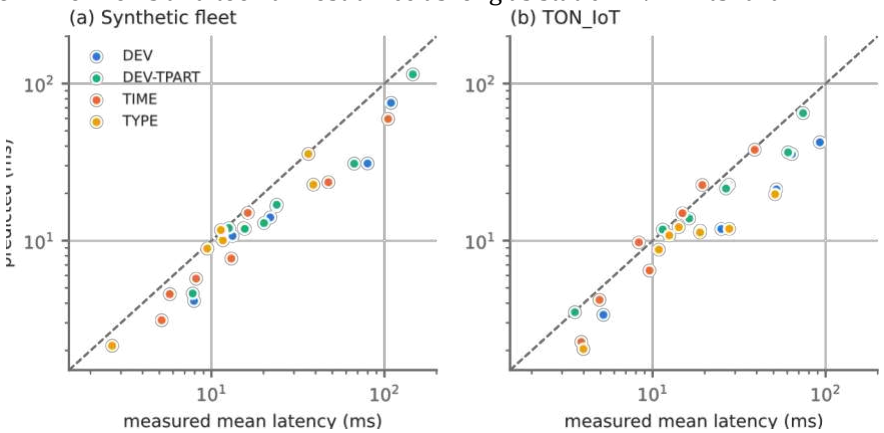


Fig. 5 Predicted against measured mean latency per layout and class on the unseen full data size

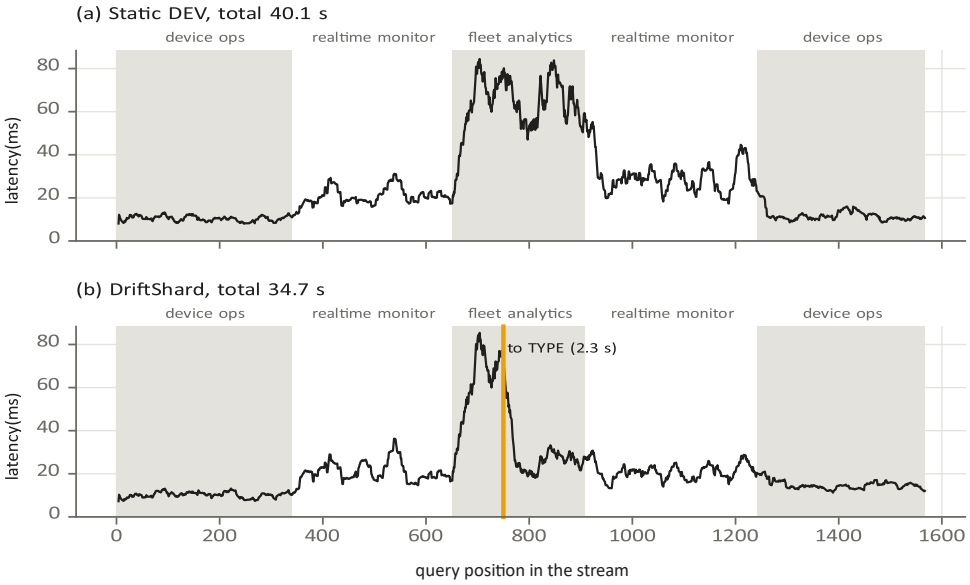


Fig. 6 Live latency (25-query moving mean) on the synthetic fleet for static DEV and DriftShard. The vertical bar marks DriftShard's migration and its duration

made one migration of 11.0 s; the layout it moved to reduced query time from 36.4 to 30.0 s, but the 1,568-query stream was too short to recover the migration, and the total of 41.0 s was 12.6% worse than static DEV. The myopic variant never migrated on this stream. In both datasets the static TYPE layout, which was not known to be the best choice before the experiment, gave the lowest total.

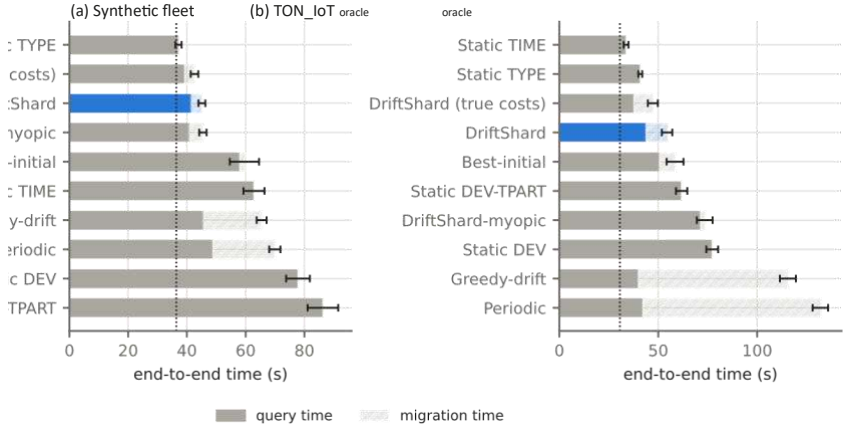


Fig. 7 End-to-end time over 40 randomised eight-phase schedules (mean and 95% confidence interval). Hatched segments are migration time; the dotted line is the oracle

Table 6 Trace-driven replay over 40 randomised eight-phase schedules: mean end-to-end time (s) with 95% confidence interval, and mean number of migrations

	Syntheticfleet		TON -IoT	
Policy	Total	Migr.	Total	Migr.
Oracle (lower bound)	36.4 ± 1.1	5.8	30.6 ± 1.3	8.0
Static DEV	77.8 ± 4.0	0	77.3 ± 3.0	0
Static TIME	62.8 ± 3.6	0	33.7 ± 1.2	0
Static DEV-TPART	86.3 ± 5.2	0	61.8 ± 2.9	0
Static TYPE	37.1 ± 1.1	0	40.9 ± 1.0	0
Best-initial	59.6 ± 5.0	0.6	58.5 ± 4.3	0.7
Periodic	69.9 ± 1.9	7.6	132.0 ± 3.9	8.2
Greedy-drift	65.4 ± 1.7	7.1	115.5 ± 4.0	7.0
DriftShard-myopic	45.5 ± 1.2	1.7	73.5 ± 4.0	0.2
DriftShard	45.1 ± 1.2	1.3	54.5 ± 2.6	1.0
DriftShard (true costs)	42.6 ± 1.3	1.2	47.3 ± 2.5	1.0

6.5 Replay over randomised regime sequences

A single live stream is one sample of regime order and length. Fig. 7 and Table 6 summarise 40 randomised eight-phase schedules per dataset.

Among policies that do not know the future, DriftShard had the lowest mean endto-end time on both datasets. On the synthetic fleet it took 45.1 s, which is 42.0% less than static DEV, 31.1% less than greedy-drift, 35.5% less than periodic re-optimisation and 24.3% less than best-initial. On TON IoT it took 54.5 s, 29.5% less than static DEV, 52.9% less than greedy-drift, 58.7% less than periodic and 6.9% less than bestinitial. DriftShard migrated about once per schedule, whereas greedy-drift and periodic migrated seven or eight times and spent between 20 and 90 s doing so.

Two comparisons qualify these gains. First, the best static layout chosen with hindsight remained ahead: TYPE on the synthetic fleet (37.1 s) and TIME on TON IoT (33.7 s). DriftShard took 21.5% and 61.5% longer respectively. On the synthetic fleet most of the gap is the one migration DriftShard needs to leave DEV, since its query time (41.5 s) was within 12% of static TYPE. On TON IoT, static TIME was within 10% of the oracle, so the regimes there gave an adaptive policy little to gain and a wrong move a great deal to lose. Second, accurate costs mattered. With measured costs, DriftShard improved by 5.6% on the synthetic fleet and by 13.2% on TON IoT, where the learned model was weaker.

The planning score of Eq. (7) and the myopic rule performed alike on the synthetic fleet (45.1 against 45.5 s, inside the confidence intervals). On TON IoT, the myopic rule rarely found a single regime long enough to justify a 3.6-million-row rebuild and stayed close to static DEV (73.5 s), whereas the planning score, which also prices the long-run mixture, moved once to a layout suited to the recurring regimes. The live TON IoT run, in

contrast, favoured the myopic rule. The evidence for the planning score is therefore positive on average but not uniform.

6.6 Effect of regime length

Fig. 8 shows end-to-end time relative to the oracle as the nominal regime length varies from 50 to 3,200 queries. Greedy-drift was the worst policy for short regimes, reaching 2.8 times the oracle on the synthetic fleet and 6.9 times on TON IoT at 100 queries per regime, because it migrated on every change regardless of how long the new regime would last. DriftShard approached the oracle steadily as regimes lengthened, from 1.97 to 1.11 times on the synthetic fleet and from 3.87 to 1.49 times on TON IoT. Greedydrift overtook DriftShard only for the longest regimes (3,200 queries), where almost any migration pays back. For very short regimes on TON IoT (50 queries), DriftShard was worse than static DEV, because its prior regime length of 800 queries overstated how long each regime would last and the online estimate had too few drifts to correct it.

6.7 Sensitivity to parameters

The detection threshold had little effect. Between $\tau = 0.10$ and $\tau = 0.50$, mean end-toend time varied between 45.5 and 46.3 s on the synthetic fleet and between 58.3 and 58.9 s on TON IoT (10 schedules per setting), because the regimes differ by a total variation distance well above the tested thresholds and the persistence requirement filtered isolated windows. The safety factor mattered more. Values of κ between 0.25 and 1 gave the best results on both datasets; at $\kappa = 4$ the time rose to 48.0 s on the synthetic fleet and to 79.2 s on TON IoT, as DriftShard became too reluctant to leave the initial layout.

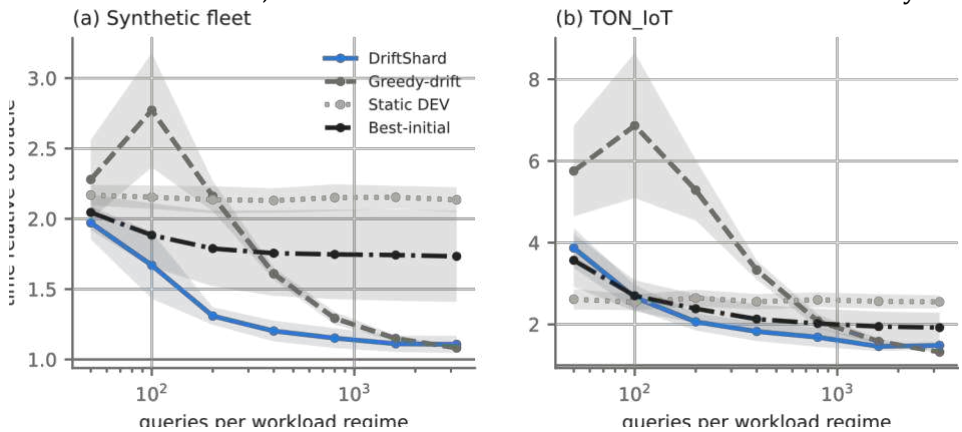


Fig. 8 End-to-end time relative to the oracle as a function of regime length (10 randomised schedules per point, 95% confidence bands)

6.8 Replay fidelity

The replay preserved the ranking of policies observed on the live cluster for the synthetic fleet, and its totals were within 1 to 9% of the live values for four of the five policies; it underestimated static TYPE by 22%. On TON IoT the agreement was weaker: the replay overestimated static DEV by 24% and the myopic policy by 30%, while greedy-drift agreed within 1%. The replay samples latencies independently from benchmark measurements and therefore misses caching and correlation effects that a long live stream contains. Its results are best read as comparisons between policies under identical conditions rather than as predictions of absolute time.

7 Discussion

When does adaptive sharding pay off? Three conditions emerged from the experiments. The regimes must conflict, so that no single layout is close to the best in all of them; on TON IoT the TIME layout was within 37% of the best in every regime, and adaptation could not beat it. The regimes must last long enough to repay a migration, which on the 3.6-million-row table meant several hundred queries. And the decision must rest on cost estimates that rank layouts correctly. When these conditions held, as for most of the synthetic schedules, DriftShard recovered most of the gap between the default layout and the best layout while migrating about once. When they did not, the migration-cost test limited the damage: DriftShard never approached the losses of greedy-drift or periodic re-optimisation, which spent up to 90 s migrating to save less than 40 s of query time.

The cost model is the weakest component. The drift detector was insensitive to its threshold, and the decision rule was simple to tune. By contrast, replacing predicted costs with measured ones improved DriftShard by 6% and 13%. The model's difficulties have identifiable causes: heavy-tailed latencies that a log-scale model underestimates even after smearing, near-ties between layouts at the regime level, and limited training data at only two or three sizes. Two remedies are practical. The first is online calibration, in which the latencies observed under the current layout correct the model's predictions for that layout as the system runs. The second is to train on more sizes and cluster configurations, so that the model can learn how fan-out and scan costs grow with data volume rather than extrapolating from one or two points.

Practical guidance. For a team operating an IIoT database on Citus, the results suggest benchmarking the candidate layouts against the expected regimes before deployment. If one layout is close to the best everywhere, as TIME was for TON IoT, it should simply be chosen and left alone. If the regimes conflict and last for hundreds of queries or more, a cost-aware adaptive policy is worthwhile, and its migration test matters more than the sophistication of its drift detector. A policy that re-shards on every change, without accounting for migration cost, should be avoided. This differs from adaptive tuning at the level of indexes and stream settings, where rule-based adaptation has been effective because each action is inexpensive (Karunarathne et al. 2026).

Migration mechanism. DriftShard charged each migration in full because queries were not served during the rebuild. A live migration mechanism, such as Citus shard moves combined with logical replication or the fine-grained reconfiguration of Squall (Elmore et al. 2015), would reduce the effective cost and move the break-even point towards shorter regimes. The decision rule is unchanged by such a mechanism; only $\hat{M}$ would need to reflect the reduced interference.

8 Threats to validity

Internal validity. The cluster ran on a single virtual machine with 2 vCPUs, so the coordinator and workers competed for the same processors and there was no network latency between them. Absolute latencies and migration times would differ on a physically distributed cluster, and the balance between fan-out and scan cost may shift. Each live policy was run once; the randomised replays supply variation across schedules but not across repeated live executions. Durability settings were relaxed, which shortens migrations relative to a production configuration.

Construct validity. End-to-end time with blocking migrations is one reasonable objective; a deployment that values tail latency or availability would weigh migrations differently. The query classes and regime mixtures were designed by the authors to represent typical IIoT roles and are not drawn from a production trace.

External validity. Only four hash layouts were considered, with a fixed shard count; composite, range and replicated layouts were not. The synthetic fleet is modest in size, and the TON IoT data required the emulation of a fleet from seven physical sensors, which removes genuine per-device behaviour. Results for larger tables and clusters with more workers may differ, and the experiments should be repeated at that scale.

Conclusion validity. Differences between DriftShard and its myopic variant were inside the confidence intervals on the synthetic fleet and changed sign between the replay and the live run on TON IoT, so no firm conclusion is drawn about the planning score beyond its average advantage.

9 Conclusion

This paper examined whether a distributed IIoT database should change its sharding layout when the analytical workload drifts, and presented DriftShard, which detects regime changes in the query mix and re-shards only when a learned estimate of the saving exceeds the measured cost of moving the data. On a live Citus cluster and across 40 randomised regime sequences on two datasets, DriftShard was faster than static device sharding by 30 to 42%, faster than drift-triggered re-sharding that ignores migration cost by 31 to 53%, and faster than a one-off workload-aware placement. It did not reach the best static layout chosen in hindsight, and on one short live stream a single unprofitable migration made it slower than the default. The experiments indicate that adaptive sharding pays off when regimes conflict and last long enough to repay a

migration, and that the accuracy of the latency model, rather than drift detection, limits what it can achieve.

Future work will add online calibration of the cost model from observed latencies, replace blocking rebuilds with live shard movement, extend the layout space to composite and range distribution, and repeat the evaluation on physically separate nodes with larger data volumes and three, five and eight workers.

Declarations

Funding. The authors did not receive support from any organisation for the submitted work.

Competing interests. The authors have no competing interests to declare that are relevant to the content of this article.

Ethics approval. Not applicable. The study uses publicly available IoT telemetry and synthetic data and does not involve human participants or animals.

Consent for publication. All authors have read and approved the final manuscript and consent to its publication.

Data availability. The TON IoT telemetry data are publicly available from UNSW Canberra (Alsaedi et al. 2020). The synthetic data can be regenerated with the generator included in the code. Benchmark measurements, replay results and figure data are available from the corresponding author on reasonable request and will be deposited in a public repository upon acceptance.

Code availability. The DriftShard implementation, cluster scripts and experiment pipeline are available from the corresponding author on reasonable request and will be released in a public repository upon acceptance.

Use of generative AI. Generative AI tools were used to assist with parts of the software implementation and with language editing. The authors designed the study, verified the code and every reported result, and take full responsibility for the content of the manuscript.

Author contributions. Lakmali Karunarathne: Conceptualisation, Methodology, Software, Investigation, Formal analysis, Data curation, Visualisation, Writing (original draft). Nalinda Somasiri: Supervision, Validation, Writing (review and editing).

References

Agrawal S, Narasayya V, Yang B (2004) Integrating vertical and horizontal partitioning into automated physical database design. In: Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data, pp 359–370, https://doi.org/10.1145/1007568.1007609

Alsaedi A, Moustafa N, Tari Z, et al (2020) TON IoT telemetry dataset: a new generation dataset of IoT and IIoT for data-driven intrusion detection systems. IEEE Access 8:165130–165150. https://doi.org/10.1109/ACCESS.2020.3022862

Bifet A, Gavalda` R (2007) Learning from time-changing data with adaptive windowing. In: Proceedings of the 2007 SIAM International Conference on Data Mining, pp 443–448, https://doi.org/10.1137/1.9781611972771.42

Cubukcu U, Erdogan O, Pathak S, et al (2021) Citus: distributed PostgreSQL for data-intensive applications. In: Proceedings of the 2021 International Conference on Management of Data, pp 2490–2502, https://doi.org/10.1145/3448016.3457551

Curino C, Jones E, Zhang Y, et al (2010) Schism: a workload-driven approach to database replication and partitioning. Proceedings of the VLDB Endowment 3(1–2):48–57. https://doi.org/10.14778/1920841.1920853

Das S, Nishimura S, Agrawal D, et al (2011) Albatross: lightweight elasticity in shared storage databases for the cloud using live data migration. Proceedings of the VLDB Endowment 4(8):494–505. https://doi.org/10.14778/2002974.2002977

Duan N (1983) Smearing estimate: a nonparametric retransformation method. Journal of the American Statistical Association 78(383):605–610. https://doi.org/10.1080/01621459.1983.10478017

Elmore AJ, Arora V, Taft R, et al (2015) Squall: fine-grained live reconfiguration for partitioned main memory databases. In: Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, pp 299–313, https://doi.org/10.1145/2723372.2723726

Gama J, Zliobaite˙ I, Bifet A, et al (2014) A survey on concept drift adaptation. ACM˘ Computing Surveys 46(4):44:1–44:37. https://doi.org/10.1145/2523813

Hilprecht B, Binnig C, R¨ohm U (2020) Learning a partitioning advisor for cloud databases. In: Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data, pp 143–157, https://doi.org/10.1145/3318464.3389704

Jensen SK, Pedersen TB, Thomsen C (2017) Time series management systems: a survey. IEEE Transactions on Knowledge and Data Engineering 29(11):2581–2600. https://doi.org/10.1109/TKDE.2017.2740932

Karunarathne L (2026) Machine learning-based query performance optimization in large-scale IoT databases. Journal of Information Technology and Digital World 8(4):287–307. https://doi.org/10.36548/jitdw.2026.4.001

Karunarathne L, Somasiri N (2026) Adaptive database optimisation of industrial IoT workloads using machine learning and local generative AI. Journal of IoT in Social,

Mobile, Analytics, and Cloud 8(4):324–344. https://doi.org/10.36548/jismac.2026.4.001

Karunarathne L, Ganesan S, Karunarathne K, et al (2026) Database optimization for low-latency analytics with adaptive indexing. Journal of Data Science and Intelligent Systems https://doi.org/10.47852/bonviewJDSIS62027607, online first

Ke G, Meng Q, Finley T, et al (2017) LightGBM: a highly efficient gradient boosting decision tree. In: Advances in Neural Information Processing Systems, pp 3146–3154

Kraska T, Beutel A, Chi EH, et al (2018) The case for learned index structures. In: Proceedings of the 2018 International Conference on Management of Data, pp 489–504, https://doi.org/10.1145/3183713.3196909

Lehmann C, Sulimov P, Stockinger K (2024) Is your learned query optimizer behaving as you expect? a machine learning perspective. Proceedings of the VLDB Endowment 17(7):1565–1577. https://doi.org/10.14778/3654621.3654625

Lu J, Liu A, Dong F, et al (2019) Learning under concept drift: a review. IEEE Transactions on Knowledge and Data Engineering 31(12):2346–2363. https://doi.org/10.1109/TKDE.2018.2876857

Marcus R, Negi P, Mao H, et al (2019) Neo: a learned query optimizer. Proceedings of the VLDB Endowment 12(11):1705–1718. https://doi.org/10.14778/3342263.3342644

Marcus R, Negi P, Mao H, et al (2021) Bao: making learned query optimization practical. In: Proceedings of the 2021 International Conference on Management of Data, pp 1275–1288, https://doi.org/10.1145/3448016.3452838

Nehme R, Bruno N (2011) Automated partitioning design in parallel database systems. In: Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data, pp 1137–1148, https://doi.org/10.1145/1989323.1989444

Pavlo A, Curino C, Zdonik S (2012) Skew-aware automatic database partitioning in shared-nothing, parallel OLTP systems. In: Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data, pp 61–72, https://doi.org/10.1145/2213836.2213844

Pavlo A, Angulo G, Arulraj J, et al (2017) Self-driving database management systems. In: Proceedings of the 8th Biennial Conference on Innovative Data Systems Research (CIDR)

Pelkonen T, Franklin S, Teller J, et al (2015) Gorilla: a fast, scalable, in-memory time series database. Proceedings of the VLDB Endowment 8(12):1816–1827. https://doi.org/10.14778/2824032.2824078

Serafini M, Taft R, Elmore AJ, et al (2016) Clay: fine-grained adaptive partitioning for general database schemas. Proceedings of the VLDB Endowment 10(4):445–456. https://doi.org/10.14778/3025111.3025125

Sisinni E, Saifullah A, Han S, et al (2018) Industrial internet of things: challenges, opportunities, and directions. IEEE Transactions on Industrial Informatics 14(11):4724–4734. https://doi.org/10.1109/TII.2018.2852491

Taft R, Mansour E, Serafini M, et al (2014) E-store: fine-grained elastic partitioning for distributed transaction processing systems. Proceedings of the VLDB Endowment 8(3):245–256. https://doi.org/10.14778/2735508.2735514

Van Aken D, Pavlo A, Gordon GJ, et al (2017) Automatic database management system tuning through large-scale machine learning. In: Proceedings of the 2017 ACM International Conference on Management of Data, pp 1009–1024, https://doi.org/10.1145/3035918.3064029

Zhou X, Chai C, Li G, et al (2022) Database meets artificial intelligence: a survey. IEEE Transactions on Knowledge and Data Engineering 34(3):1096–1116. https://doi.org/10.1109/TKDE.2020.2994641