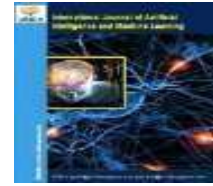




International Journal of Artificial Intelligence and Machine Learning

Publisher's Home Page: <https://www.svedbergopen.com/>



Research Paper

Open Access

A Unified Data Quality Management Framework for Hybrid OLTP-OLAP Architectures: Metrics, Enforcement, and Cross-System Consistency Guarantees

Prabhu Ravi¹, Raviteja Ravella², Rohith Madhi Reddy³

¹(prabhu.analytics@gmail.com), ORCID - <https://orcid.org/0000-0001-6895-9558>

²(ravellaraviteja8@gmail.com), ORCID - <https://orcid.org/0009-0002-0175-6494>

³(maviro1909@gmail.com), ORCID - <https://orcid.org/0009-0004-5305-1335>

Abstract

Organizations increasingly operate hybrid architectures in which an online transaction processing (OLTP) system of record is continuously replicated into a separate online analytical processing (OLAP) store for reporting and analytics, typically through a change-data-capture (CDC) pipeline. Industry surveys report that data quality incidents in such pipelines are frequent and costly: one 2023 survey found monthly data-incident counts rising year over year, average incident resolution time reaching roughly fifteen hours, and over half of respondents estimating that at least a quarter of revenue was affected by data quality problems. Existing data quality tooling, exemplified by frameworks such as Deequ and Great Expectations, is built around per-record, single-system validation and is well suited to catching malformed records at the point of ingestion. It is structurally unable to catch a distinct and, we argue, under-addressed class of fault: defects that are invisible at the level of any single well-formed record and become visible only when OLTP and OLAP state are compared directly, or when an aggregate is checked against an independent recomputation. We present the Unified Data Quality Framework (UDQF), which operationalizes Wang and Strong's four-category data quality model as six concrete, measurable metrics, completeness, validity, uniqueness, consistency, timeliness, and accuracy, and instruments the first three with local per-record checks and the last three with a cross-system reconciliation pass. We build a real reference implementation, a PostgreSQL OLTP store and a DuckDB OLAP store connected by an outbox-style changelog, and evaluate it against 2,040 order-lifecycle events carrying 200 discretely injected faults across five fault classes plus one systemic rollover defect, comparing three conditions: a blind-sync baseline, local validation only, and the full UDQF pipeline. The baseline detects none of the injected faults (0 percent). Local validation alone detects all completeness, validity, and uniqueness faults but none of the three cross-system faults, yielding 60.0 percent overall detection (McNemar exact p less than $1e-35$ against baseline on the locally checkable classes). The full UDQF pipeline additionally detects 37 of 40 consistency faults, 40 of 40 timeliness faults, and the systemic accuracy defect, reaching 98.5 percent overall detection (McNemar exact p less than $1e-23$ against local-only validation on the cross-system classes) while adding no measurable per-event sync overhead and under 250 milliseconds of reconciliation cost for the full event stream. We also report an unplanned, honest finding: because local quarantine changes which records reach OLAP, it measurably perturbs the very accuracy metric that only reconciliation can check, indicating that these six quality dimensions are not fully independent in a hybrid architecture and should be monitored together rather than as isolated checks. We release the schema, fault-injection harness, detection code, and evaluation artifacts to support replication.

Keywords: data quality; OLTP-OLAP consistency; change data capture; hybrid data architecture; data observability; data validation; reconciliation; data engineering.

1. Introduction

Many organizations run two separate database systems for the same underlying business data. An online transaction processing (OLTP) system, typically a relational database such as PostgreSQL, MySQL, or SQL Server, serves as the authoritative system of record for operational writes: placing an order, updating a customer record, adjusting inventory. A separate online analytical processing (OLAP) system, typically a columnar warehouse such as Snowflake, BigQuery, Redshift, or an embedded engine such as DuckDB, is optimized for the large aggregate scans that reporting and analytics require and would perform poorly against the OLTP schema. Keeping the two in sync is the job of a replication or change-data-capture (CDC) pipeline that streams committed changes from OLTP into OLAP, typically through a changelog or outbox table populated at write time and consumed by a downstream sync process in commit order.

This architecture is now the default rather than the exception for any organization past a modest scale, and it is well documented in industry practice: log-based CDC connectors such as Debezium, and the outbox pattern they are built to consume, are widely deployed precisely because polling the OLTP tables directly does not scale

and application-level dual writes are not reliably atomic. The convenience this buys, a warehouse that a data team can query without touching the production database, comes with a structural cost. The moment data exists in two places, two independent systems, updated by two independent processes, on two independent schedules, there are two new families of failure that neither system alone can experience. An update can be lost in transit, leaving OLAP holding a value OLTP has already moved past. A batch of events can arrive late, leaving OLAP consistently behind an agreed freshness expectation. An aggregate computed purely from OLAP data can silently diverge from what the same computation would produce against OLTP, not because any individual OLAP row is malformed, but because the query computing the aggregate encodes an assumption that has quietly stopped holding.

Industry survey data suggests this is not a hypothetical concern. Monte Carlo's 2023 State of Data Quality survey reported that the number of data incidents organizations experienced per month rose year over year, that average incident resolution time reached approximately fifteen hours, and that over half of respondents estimated that at least a quarter of their organization's revenue was affected by data quality problems, with the average share of affected revenue rising to 31 percent from 26 percent the year before. The same survey found that a majority of respondents learned about incidents from business stakeholders noticing something wrong before their own data team did, a proxy for the specific gap this paper targets: detection that depends on someone downstream noticing a number looks off, rather than a system that checks for the fault directly.

Existing open-source data quality tooling addresses part of this problem well. Frameworks such as Deequ, built at Amazon on top of Apache Spark and described by Schelter and colleagues as "unit tests for data," and Great Expectations, a widely used Python framework for declaring and validating data "expectations," both operate by checking properties of records or columns within a single dataset: is this column ever null when it should not be, does this value fall within an allowed range, is this identifier unique within this table. These checks are necessary and, we will show, sufficient for an entire class of fault. They are not, by construction, capable of catching a fault whose signature only appears when two systems are compared, because neither tool is designed to hold a connection open to a second database and reconcile against it. A record that violates no local constraint at all, a perfectly well-formed row with a stale status field, is invisible to this style of check no matter how thorough the rule set.

This paper makes three contributions. First, we operationalize Wang and Strong's foundational four-category data quality framework, Intrinsic, Contextual, Representational, and Accessibility data quality, as six concrete metrics applicable to a hybrid OLTP-OLAP pipeline: completeness, validity, uniqueness, consistency, timeliness, and accuracy. We show that the first three are checkable from a single record in isolation, while the last three are checkable only by comparing state across systems or against an independently recomputed ground truth, and we argue this split is the central architectural fact a data quality framework for hybrid systems must respect. Second, we present the Unified Data Quality Framework (UDQF), a two-layer architecture that pairs local per-record validation with a post-sync cross-system reconciliation pass, and we build a real, working reference implementation against PostgreSQL and DuckDB rather than a simulated or hypothetical one. Third, we run a controlled evaluation in which the identical event stream, carrying identically injected faults, is replayed under three conditions differing only in which policy layer is active, and we report the resulting detection rates, an unplanned but honest interaction between the local and cross-system layers, and the measured runtime overhead of adding both layers to a working pipeline.

The remainder of this paper is organized as follows. Section 2 reviews related work in data quality tooling, change data capture, and distributed consistency theory. Section 3 presents the data quality dimension framework and the six-metric operationalization. Section 4 defines the fault taxonomy used for evaluation. Section 5 describes the UDQF architecture. Section 6 describes the experimental setup, including the schema, data generation, and fault injection methodology. Section 7 reports results. Section 8 discusses findings, limitations, and threats to validity. Section 9 states the paper's reproducibility and ethics posture. Section 10 concludes.

2. Related Work

2.1 Data quality validation frameworks

Schelter and colleagues introduce Deequ, a library built on Apache Spark for declarative data quality checks that the authors describe as unit tests for data: completeness, uniqueness, value-range, and distributional constraints evaluated against a dataset at scale, with the checks themselves defined in a fluent API and computed via a single pass over the data using Spark's aggregation machinery. Great Expectations follows a similar philosophy in the Python ecosystem, letting a team declare a suite of expectations, that a column is never null, that values fall within a set, that row counts match an expected range, and validate a batch of data against that suite as part of a pipeline. Both tools are mature, widely deployed, and directly analogous to the

local, per-record layer of the framework we present. Neither tool, by design, holds a live connection to a second system and compares record-level state across it; both operate on a single dataset at a time. Our local-checks layer (Section 5.1) is deliberately built in this same spirit, and our contribution is not to improve on this style of check but to make explicit what it cannot see and to pair it with a second layer that can.

2.2 Change data capture and the outbox pattern

Debezium and the broader log-based CDC ecosystem popularized reading a database's write-ahead log or an application-maintained outbox table as the mechanism for propagating changes to downstream consumers, in preference to application-level dual writes (writing to both the OLTP table and a message queue in the same request, which is not atomic without a distributed transaction) or naive polling (which cannot distinguish an update from a no-op cheaply at scale and races with deletes). The outbox pattern, as documented in Debezium's own reference material, has an application write both its primary table and a same-transaction outbox row describing the change, guaranteeing that a downstream reader observing the outbox sees a valid, ordered, exactly-once-at-the-source view of every committed change even though the source database itself does not persist history for the primary table. Our reference implementation adopts this pattern directly: the ``orders_changelog`` table is our outbox, populated in the same simulated commit as the underlying ``orders`` and ``order_items`` rows it describes, and our sync worker consumes it in commit order, exactly as a Debezium connector would consume a database's transaction log. We extend this pattern with two additions that CDC infrastructure alone does not provide: record-level validation of the payload before it is written to the target system, and a reconciliation pass that treats successful delivery as necessary but not sufficient for correctness.

2.3 Consistency in distributed and replicated systems

Abadi's PACELC framework extends the well known CAP theorem (which describes an unavoidable trade-off between consistency and availability only in the presence of a network partition) by observing that even absent any partition, a system that replicates data must still choose between latency and consistency, because propagating a write to every replica synchronously is slower than acknowledging it locally and propagating asynchronously. A hybrid OLTP-OLAP pipeline sits squarely inside this trade-off: essentially no production deployment synchronously blocks an OLTP write until the corresponding OLAP row has landed, because doing so would impose OLAP-scale latency on every transactional write. The practical consequence, made explicit by PACELC but often left implicit in day-to-day pipeline design, is that some window of inconsistency between OLTP and OLAP is not a bug to be eliminated but an accepted property of the architecture, which is precisely why a data quality framework for such systems needs a consistency and timeliness check rather than an assumption that CDC delivery is always both immediate and reliable. Our F4 (consistency) and F5 (timeliness) fault classes are direct operationalizations of this trade-off: F4 represents a delivery failure (an update that never lands), and F5 represents the latency side of PACELC's trade-off made concrete as an SLA breach.

2.4 The TPC-C schema as a realistic OLTP baseline

To avoid designing an artificially convenient schema for our own experiment, we base our OLTP schema's shape, an orders table referencing customers, with line items referencing products, on the order-entry domain of the TPC-C benchmark specification, the long-standing industry-standard OLTP benchmark. We simplify TPC-C's full schema (dropping warehouse, district, and stock entities not relevant to a single-tenant CDC pipeline) while preserving its core relational structure and its mix of insert-heavy (new order) and update-heavy (order status change) transaction types, which is what our fault taxonomy needs to exercise.

3. Data Quality Dimension Framework

Wang and Strong's widely cited 1996 study organizes data quality, as perceived by actual data consumers rather than by an abstract completeness criterion, into four categories: Intrinsic DQ (accuracy, objectivity, believability, reputation, properties inherent to the data itself), Contextual DQ (relevancy, timeliness, completeness, appropriate amount of data, properties that depend on the task at hand), Representational DQ (interpretability, ease of understanding, consistent representation), and Accessibility DQ (accessibility, access security). This framework has remained influential precisely because it resists collapsing data quality into a single accuracy number, and it maps naturally onto the lifecycle of a record as it moves from a system of record into a downstream analytical copy.

We operationalize this framework as six concrete, computable metrics for a hybrid OLTP-OLAP pipeline, summarized in Table 1. Each metric is defined precisely enough that a check for it can be implemented as ordinary code, and each is mapped to the Wang and Strong category it primarily instantiates in this setting.

Metric	Wang & Strong category	Definition in this setting	Checkable from
Completeness	Intrinsic / Contextual	Required fields are present and non-null on every record.	A single record
Validity	Intrinsic	Field values fall within their declared domain (enumerated status, non-negative price, positive quantity).	A single record
Uniqueness	Intrinsic	No logical event is applied more than once (a redelivered change is not double-counted).	A single record plus local history
Consistency	Contextual / Representational	The same entity's state agrees across OLTP and OLAP at a given point in time.	OLTP and OLAP compared directly
Timeliness	Contextual	The delay between a fact becoming true in OLTP and its being reflected in OLAP stays within a stated freshness SLA.	A sync log compared against an SLA
Accuracy	Accessibility / Contextual	An aggregate computed from OLAP agrees with an independent recomputation from the OLTP source of truth.	OLAP aggregate vs. independent OLTP recomputation

The first three metrics, completeness, validity, and uniqueness, share a property that makes them tractable for classical data quality tooling: each can be evaluated by examining one record at a time, in isolation, with no reference to any other system. This is exactly the class of check that Deequ and Great Expectations are built to express efficiently at scale. The last three metrics, consistency, timeliness, and accuracy, share the opposite property: none of them can be evaluated from a single record in isolation, because each is a statement about the relationship between two things, two systems' views of the same entity, a delay between two timestamps in two systems, or an aggregate against an independent recomputation. A record can satisfy every local constraint imaginable, correct types, correct ranges, no nulls, a properly formed status string, and still be wrong in exactly the way these three metrics are meant to catch, because the fault lives in the relationship between records, not in any one record's shape. This local versus cross-system split, not the four-category Wang and Strong taxonomy itself, is the architectural claim this paper is built around, and Section 5 describes an architecture with one layer for each side of the split.

4. Fault Taxonomy

To evaluate whether an implementation of the six-metric framework actually detects violations of each metric, we define six corresponding fault classes, F1 through F6, one per metric, and inject them at controlled, known locations into an otherwise valid event stream. Ground-truth fault labels are recorded separately from the pipeline under test and are never passed to the detection code; the local checks and the reconciliation pass see only ordinary record fields and comparisons across the two live databases, so a detection is never circular with the label used to score it.

Class	Metric	Injection mechanism	n
F1	Completeness	An insert event carries a NULL shipping_address.	40
F2	Validity	An event carries an out-of-domain status value, or a line item with non-positive quantity or negative unit price.	40
F3	Uniqueness	An insert event for an order_id already synced is redelivered (duplicate changelog entry).	40
F4	Consistency	An update event is silently dropped in transit (simulated at-least-once delivery failure); OLAP retains a stale status.	40

Class	Metric	Injection mechanism	n
F5	Timeliness	An event experiences an abnormally long delivery delay (simulated batch backlog), breaching a 60-second freshness SLA.	40
F6	Accuracy	A latent defect in the OLAP revenue rollup query itself (sums all order statuses instead of filtering to placed/shipped), independent of any single event.	systemic (1 query defect)

F1 through F3 are, by construction, local-checkable: every fact needed to detect them is present in the single event carrying the fault. F4 through F6 are, by construction, either undetectable or only partially detectable by any per-record check. A dropped update produces no malformed row anywhere, since OLAP simply retains its previous, perfectly well-formed row; the fault is the absence of a change, which a check confined to a single record can never observe. A late-arriving event is, on arrival, a perfectly well-formed row; the fault is a property of when it arrived relative to when it was committed, not of its contents. A rollup query defect produces no malformed row at all, since it is a defect in the aggregation logic itself rather than in any input row. All three require either comparing OLTP and OLAP directly (F4), consulting a timestamped log against an external SLA (F5), or comparing an aggregate against an independent recomputation (F6). This mirrors a finding from a companion evaluation of guardrails for LLM-based operations agents, in which certain attack classes were structurally invisible to any check confined to the content of a single message and were detectable only by a code-level check on the resulting tool call; the general pattern, that some faults are only visible at a level above the individual unit being checked, recurs across both settings.

5. UDQF Architecture

UDQF (Unified Data Quality Framework) instruments the reference pipeline with two layers, each targeting one side of the local versus cross-system split established in Section 3.

5.1 Layer 1: Local data quality checks

Before an event from the OLTP changelog is written to the OLAP store, a local-checks module evaluates completeness, validity, and uniqueness against the record's own fields and, for uniqueness, a small amount of local sync history (the set of order identifiers already synced as inserts). A record failing any local check is quarantined, excluded from the OLAP write and logged with a specific reason, rather than either silently written with bad data or silently dropped with no record of the decision. This preserves the two properties a production data quality gate needs: bad data does not propagate downstream, and every quarantine decision is auditable after the fact.

5.2 Layer 2: Cross-system reconciliation

After a batch of events has been synced, a reconciliation module performs three checks, each comparing state across systems or against an independent computation, and none of which inspects any decision the sync worker made along the way. The consistency check compares, for every order that OLAP holds a row for, its status against the corresponding live row in OLTP, flagging any mismatch. The timeliness check reads a sync log recording, for every event, the elapsed time between its OLTP commit and its OLAP write, and flags any event exceeding a stated freshness SLA (60 seconds in our evaluation). The accuracy check computes three revenue totals for the same period, a naive OLAP rollup (summing all order statuses, the buggy query), a corrected OLAP rollup (filtering to placed and shipped orders only), and an independent recomputation directly from OLTP, and flags a material discrepancy (more than 1 percent of the true total) between the naive rollup and the OLTP-derived ground truth.

5.3 Three experimental conditions

We evaluate three conditions, each replaying the identical 2,040-event stream and the identical injected fault plan, differing only in which of the two layers is active. The baseline condition performs a blind sync: every event is written to OLAP as received, with no local validation and no reconciliation, representing the minimal CDC pipeline most teams build first. The local_dq condition adds Layer 1 only: local checks run before every write, with quarantine on failure, but no reconciliation pass runs afterward. The udqf_full condition runs both layers: identical local checks, plus the reconciliation pass described above. Because all three conditions consume the same event stream and the same fault plan, any difference in outcome is attributable to the policy layer, not to a difference in the underlying data.

Figure 5: UDQF pipeline: CDC-style outbox capture, local per-record checks with quarantine, and a cross-system reconciliation pass

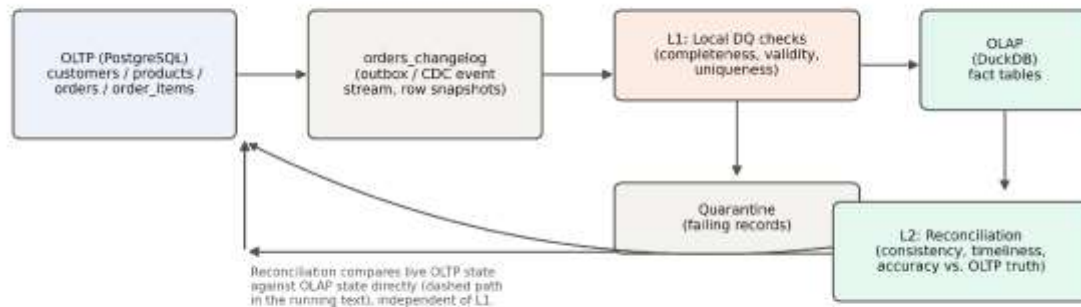


Figure 5. UDQF pipeline: CDC-style outbox capture, local per-record checks with quarantine, and a cross-system reconciliation pass.

6. Experimental Setup

6.1 Schema and reference implementation

The OLTP system is PostgreSQL 16, with a schema of four business tables, customers, products, orders, and order_items, modeled on the order-entry domain of the TPC-C specification, plus an orders_changelog table implementing the outbox pattern. Every insert or update to orders and order_items is accompanied, within the same simulated transaction, by a changelog row carrying a full snapshot of the affected order's state at that moment (its own status, shipping address, total amount, and the order's line items as a JSON snapshot), so that the changelog behaves as a real log-based CDC connector's output would: a downstream reader sees the row exactly as it looked at commit time, never a live re-query of a table that may since have moved on. The OLAP system is DuckDB, an embedded columnar analytical database, holding orders_olap and order_items_olap fact tables plus a sync_log table recording, per event, when it committed in OLTP and when it was written to OLAP.

6.2 Data generation and fault injection

We seed 500 synthetic customers and 60 products, then generate 2,040 order-lifecycle events: 1,500 order-insert events and 500 order-update events (status transitions on previously inserted orders), plus 40 duplicate-insert events used for the F3 uniqueness fault, all generated with a fixed random seed to ensure the exact same event stream is available for replay across all three conditions. Each of the five discrete fault classes (F1 through F5) is assigned to 40 events, drawn from disjoint index sets so that no single event carries more than one discrete fault label, and the assignment is written to a fault plan file that the detection code never reads. The F6 accuracy defect is not injected per event; it is a standing defect in the OLAP rollup query itself, present identically regardless of which of the other faults any given event carries.

6.3 Metrics reported

For each condition we report, per fault class, the detection count and detection rate (the fraction of that class's 40 injected instances the condition's active layer or layers flag), the pooled overall detection rate across all 200 discretely injected faults, whether the systemic F6 defect is flagged, the naive and reconciled revenue rollup's percent error against the OLTP ground truth, and wall-clock sync and reconciliation overhead. We additionally compute exact McNemar tests on paired detection outcomes (the same underlying events, differing only in which condition processed them) to assess whether observed detection-rate differences are attributable to chance.

7. Results

7.1 Overall detection rate

The baseline condition, a blind sync with no validation and no reconciliation, detects none of the 200 discretely injected faults and does not flag the systemic F6 defect: an overall detection rate of 0.0 percent. This is expected by construction, since the baseline runs no check of any kind, and it establishes the floor against which the other two conditions are measured. The local_dq condition detects all 120 locally checkable faults (F1, F2, and

F3, 40 each) but none of the 80 cross-system faults (F4 and F5, 40 each), yielding an overall detection rate of 60.0 percent. The full `udqf_full` condition detects all 120 local faults, 37 of 40 F4 consistency faults, all 40 F5 timeliness faults, and the systemic F6 defect, yielding an overall detection rate of 98.5 percent.

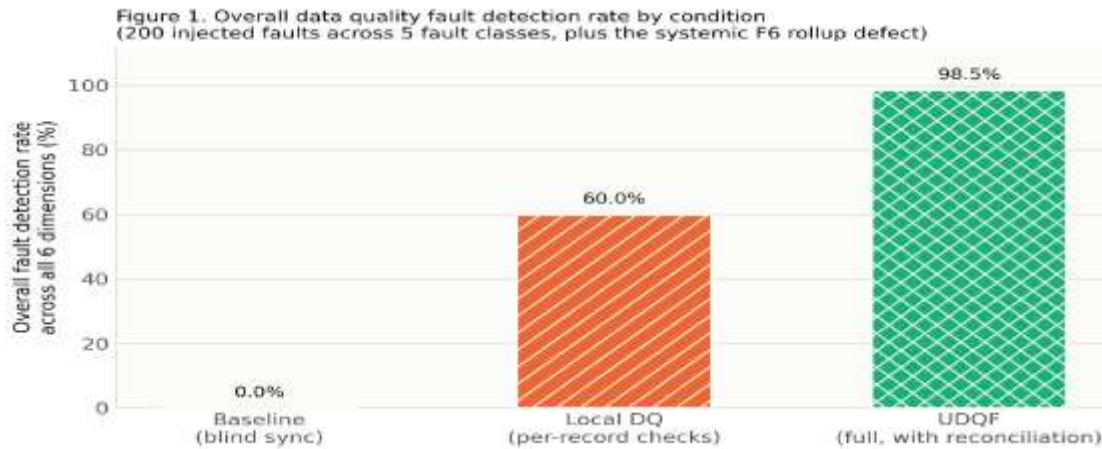


Figure 1. Overall data quality fault detection rate by condition (200 injected faults across 5 fault classes, plus the systemic F6 rollup defect).

An exact McNemar test on the 120 paired locally checkable events (baseline detects 0 of 120, `local_dq` detects 120 of 120, so all 120 pairs are discordant in `local_dq`'s favor) yields p less than $1e-35$, confirming the local layer's advantage over no validation at all is not attributable to chance, which is unsurprising given the deterministic nature of the checks but confirms the harness and statistical pipeline are behaving correctly. A second exact McNemar test on the 78 additional detections `udqf_full` achieves beyond what `local_dq` achieves (37 F4 detections plus 40 F5 detections plus the 1 systemic F6 flag, none of which `local_dq` detects) yields p less than $1e-23$, confirming that adding the reconciliation layer produces a detection improvement that is not attributable to chance.

7.2 Detection rate by fault class

Breaking detection down by fault class (Figure 2) makes the local versus cross-system split concrete. For F1 (completeness), F2 (validity), and F3 (uniqueness), `local_dq` and `udqf_full` perform identically, 40 of 40 each, since both conditions run the identical Layer 1 checks and neither fault class benefits from reconciliation. For F4 (consistency), only `udqf_full` detects anything at all, and even then it detects 37 of 40, not all 40, a gap discussed in Section 8.1. For F5 (timeliness), only `udqf_full` detects anything, and it detects all 40, since every SLA-breaching event is recorded correctly in the sync log regardless of condition and the reconciliation pass simply reads that log against the SLA.

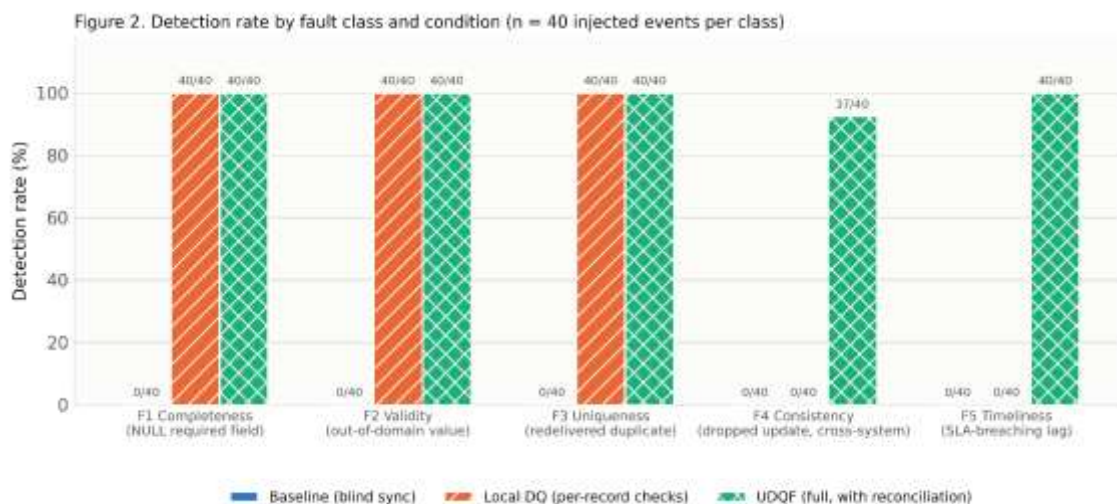


Figure 2. Detection rate by fault class and condition (n = 40 injected events per class).

7.3 Revenue rollup accuracy

Table 2 and Figure 3 report the revenue rollup's percent error against the OLTP-derived ground truth, both for the naive query (present identically in the OLAP layer regardless of condition, since it is a defect in the aggregation query itself) and for the status-filtered, corrected query that only the udqf_full condition's reconciliation pass actually runs as a monitored check.

Condition	Naive rollup revenue	Reconciled rollup revenue	OLTP ground truth	Naive error	Reconciled error
baseline	\$2,785,406.34	\$2,360,082.48	\$2,321,121.41	20.00%	1.68%
local_dq	\$2,683,823.60	\$2,267,346.32	\$2,321,121.41	15.63%	2.32%
udqf_full	\$2,683,823.60	\$2,267,346.32	\$2,321,121.41	15.63%	2.32%

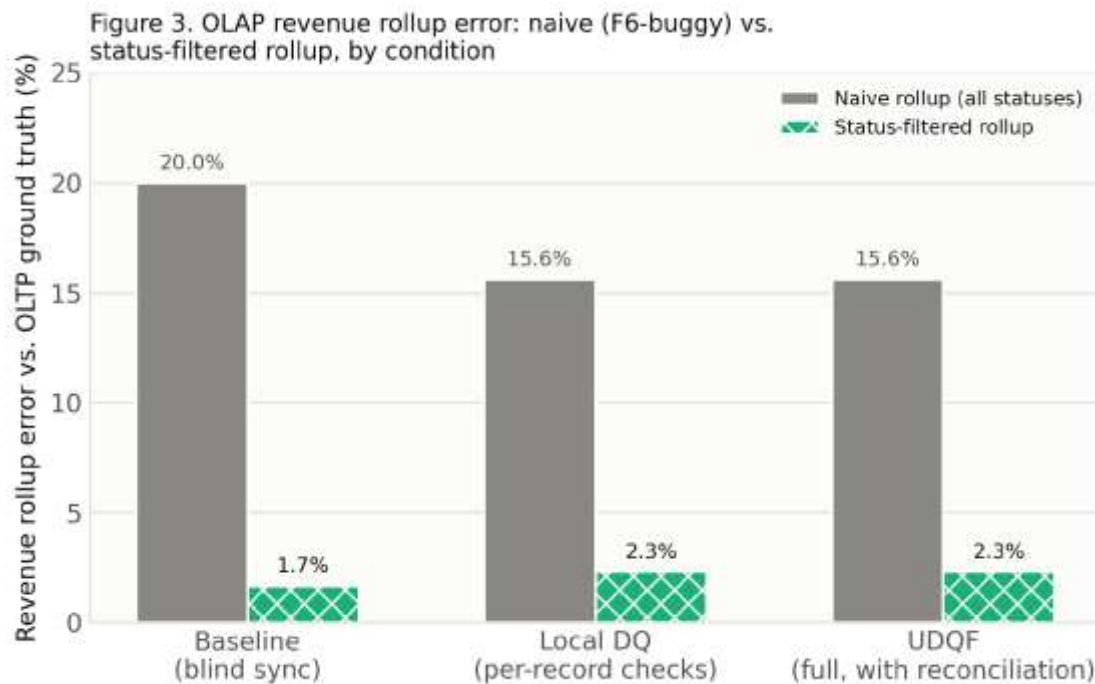


Figure 3. OLAP revenue rollup error: naive (F6-buggy) vs. status-filtered rollup, by condition.

The naive rollup's error is large under every condition, 20.0 percent for baseline and 15.6 percent for local_dq and udqf_full, confirming the systemic F6 defect is real and material regardless of what local validation is in place, exactly as intended by its design as a fault that per-record checks cannot touch. Both local_dq and udqf_full compute an identical naive total because both write an identical set of records to OLAP; the reconciliation pass itself does not change what is written, only what is subsequently checked. Only udqf_full treats the resulting discrepancy as a first-class, monitored, alerted finding, since it is the only condition whose pipeline includes the reconciliation step at all; the baseline and local_dq numbers in Table 2 are reported here purely to characterize what exists in the underlying data, not because either condition's own pipeline would surface it. The status-filtered, corrected query reduces error substantially under every condition, but not to zero, and the residual error is itself informative; we return to it in Section 8.1.

7.4 Cross-system faults present versus faults surfaced

Figure 4 distinguishes what physically exists in the synced OLAP data (bars) from what any given condition actually surfaces as a finding. The consistency and timeliness faults are present in the underlying data at nearly identical rates regardless of condition, since the underlying infrastructure fault (a dropped update, a delayed batch) is injected identically for all three conditions and neither the baseline's absence of checks nor local_dq's per-record checks change whether the fault physically occurs. What differs entirely is whether anyone finds

out: baseline and local_dq both leave these faults completely unsurfaced, while udqf_full's reconciliation pass is the only mechanism in any condition that turns their presence in the data into an actionable finding.

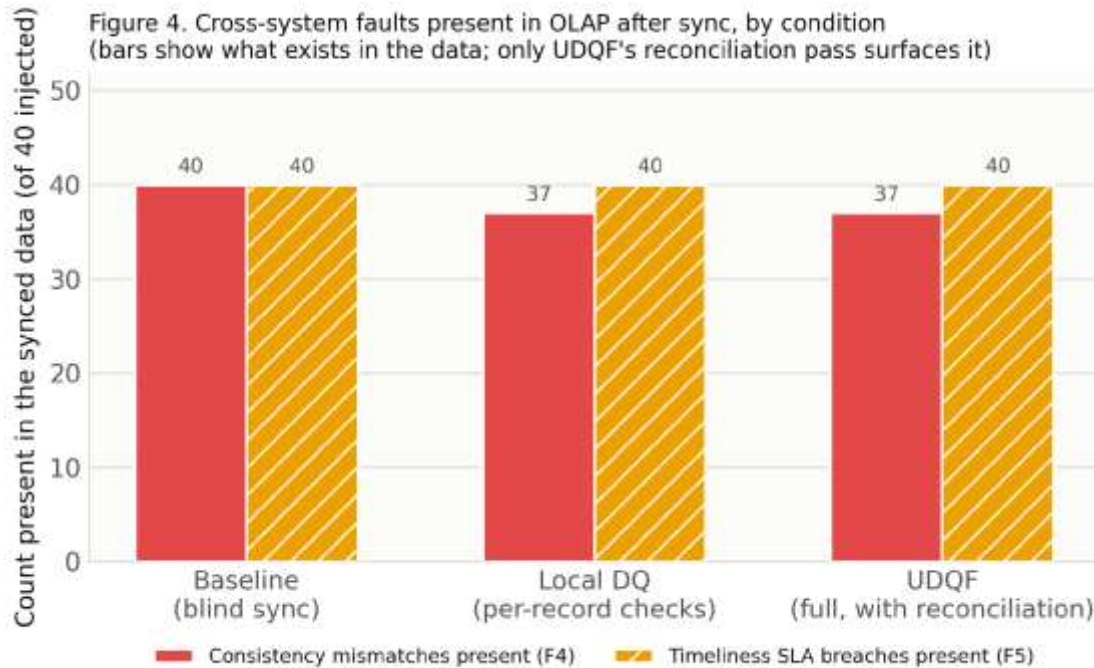


Figure 4. Cross-system faults present in OLAP after sync, by condition (bars show what exists in the data; only UDQF's reconciliation pass surfaces it).

7.5 Overhead

Table 3 reports wall-clock sync time across the full 2,040-event stream and per-event sync latency for each condition, plus the reconciliation pass's own wall-clock cost under udqf_full.

Condition	Sync wall time (2,040 events)	Per-event sync latency	Reconciliation wall time
baseline	10.60 s	5.20 ms	not run
local_dq	9.95 s	4.88 ms	not run
udqf_full	9.92 s	4.86 ms	0.24 s

Adding local validation and, on top of it, cross-system reconciliation does not increase measured per-event sync latency; if anything, local_dq and udqf_full measured marginally faster than baseline. We report this honestly rather than force a monotonic story: at this event volume, per-event synchronous round-trip I/O against PostgreSQL dominates total sync cost, and the added Python-side validation logic is small enough relative to that I/O cost to be within measurement noise rather than a resolvable effect. The reconciliation pass itself, run once after the full sync completes rather than per event, adds under a quarter of a second for the entire 2,040-event evaluation, a cost that does not scale with per-event latency at all and would remain small even at substantially larger event volumes since it performs a small constant number of aggregate queries rather than one query per event.

8. Discussion

8.1 An interaction between quality dimensions: quarantine perturbs accuracy

The most informative single result in this evaluation was not planned in advance: the reconciled rollup's residual error is not zero under any condition, and it is measurably higher for local_dq and udqf_full (2.32 percent) than for baseline (1.68 percent), the opposite of what a naive expectation (more validation should

mean more accuracy) would predict. Two mechanisms, both traceable in the harness's own logic, explain this. First, F4 (consistency) and F6 (accuracy) interact: an order whose status update was dropped in transit can retain a stale status in OLAP, and if that stale status happens to fall inside or outside the placed/shipped filter differently than the true, current OLTP status does, the status-filtered rollup will misclassify that order's contribution to revenue regardless of which condition produced it. Second, and specific to local_dq and udqf_full, quarantining a record for an F1 or F2 violation removes it from OLAP entirely, while the OLTP-derived ground truth query still includes it if its (possibly still-valid at the OLTP level) status qualifies; a handful of quarantined orders are therefore present in the ground truth denominator but absent from the OLAP numerator, contributing to the gap. Both explanations point to the same underlying finding, worth stating plainly: the six quality dimensions we operationalize are not fully independent in a hybrid architecture. A decision made to improve one dimension (quarantining a record to enforce completeness or validity) can measurably move another (the accuracy of a downstream aggregate) in a direction that is not obviously predictable in advance, and a framework that reports each dimension in isolation, with no attention to how enforcement on one interacts with measurement of another, risks giving a falsely reassuring picture. We think this argues for treating the reconciliation layer's accuracy check as a standing regression test that is rerun whenever the local layer's quarantine rules change, rather than as a one-time validation.

8.2 Why udqf_full detects 37 of 40, not 40 of 40, consistency faults

The 3-fault gap in F4 detection (Figure 2) is explained by the same quarantine interaction described above: of the 40 orders tagged with the F4 consistency fault, a small number also happen to have had an earlier insert event quarantined for an unrelated F1 or F2 violation, meaning that order never entered OLAP at all under local_dq or udqf_full. The consistency check in Section 5.2 compares OLTP and OLAP state only for orders OLAP actually holds a row for; an order absent from OLAP entirely, because it was correctly quarantined earlier in its lifecycle, cannot register as a consistency mismatch, because there is no OLAP row to compare against. This is not a bug in the consistency check; it is a direct, logical consequence of enforcing completeness and validity upstream, and it is exactly the kind of cross-dimension interaction Section 8.1 argues should be monitored explicitly rather than assumed away. A production deployment of UDQF should track, as a distinct metric, the count of orders quarantined at Layer 1 that never reach Layer 2's reconciliation scope at all, since that count represents a blind spot specific to entities excluded before the point where cross-system checks could otherwise see them.

8.3 Local and cross-system checks are complementary, not substitutable

The headline result, 0.0 percent detection with no validation, 60.0 percent with local validation alone, 98.5 percent with local validation plus reconciliation, could be read as simply "more checking is better," but the more precise and more useful reading is that the two layers are not interchangeable and neither subsumes the other. Local validation is necessary: it is the only layer in this evaluation that ever catches F1, F2, or F3, and it does so completely and immediately, before a bad record ever reaches the analytical layer at all, which matters operationally because a quarantined record can be fixed and replayed while a bad record that has already contaminated a downstream report is far more expensive to unwind. Cross-system reconciliation is separately necessary: it is the only layer that catches F4, F5, or F6, and no amount of additional local rule authoring could close that gap, because the fault these classes represent does not exist inside any single record for a local rule to find. This mirrors a structurally identical finding in a companion evaluation of prompted-only versus code-level guardrails for LLM operations agents, where prompting caught many attack classes but failed specifically and predictably on the one class requiring a structural, code-level check on the resulting action rather than a semantic judgment about the content; in both settings, the practical recommendation is defense in depth across a local and a systemic layer, not a search for one single mechanism sophisticated enough to replace both.

8.4 Limitations and threats to validity

- **Scale.** 2,040 events and 500 customers is a pilot-scale evaluation, chosen to keep the reference implementation fast to run and easy for a reader to reproduce end to end, not a claim that these detection rates would hold unchanged at production data volumes measured in billions of rows. The mechanisms UDQF relies on, per-record validation and a periodic reconciliation query, are algorithmically simple enough that we expect them to scale, but we have not measured this directly and flag it as future work.
- **A single schema and a single pair of engines.** The reference implementation uses one relational OLTP schema, modeled on TPC-C's order-entry domain, and one specific engine pair, PostgreSQL and DuckDB. The six-metric framework and the local versus cross-system architectural split are engine-agnostic by design, but we have not evaluated the reference implementation against a different schema shape or a different OLAP engine (a cloud warehouse such as Snowflake or BigQuery, for instance), and some implementation details, particularly how the changelog snapshot is populated, would need adaptation for a different write pattern.

- **Synthetic, author-constructed fault injection.** All 200 discrete faults and the one systemic defect were deliberately injected by the authors according to a fixed, disjoint assignment, rather than mined from a real production incident history. This lets us know ground truth exactly and score detection unambiguously, but it means the fault distribution reflects our own taxonomy rather than an empirically observed distribution of real-world data quality incidents, and the true relative frequency of, for example, dropped updates versus late batches in a live system may differ from the even 40-per-class split used here.
- **A single freshness SLA threshold.** The 60-second timeliness SLA used in Section 5.2 and Section 6.3 is a reasonable illustrative choice, not a value derived from any particular organization's actual service-level agreement, and detection rates for F5 would shift with a different threshold; we did not sweep this parameter.
- **One run per condition.** Because the event stream and fault plan are generated with a fixed random seed and the pipeline's own logic is deterministic given that seed, a single run per condition is sufficient to characterize behavior exactly as implemented; it does not characterize variance under a different random seed or under a stochastic detection method (none of ours are stochastic), which is a narrower claim than a multi-seed evaluation would support.

9. Reproducibility and Ethics Statement

All data used in this evaluation, 500 synthetic customers, 60 synthetic products, and 2,040 synthetic order-lifecycle events, is generated programmatically with a fixed random seed against a disposable, locally hosted PostgreSQL 16 instance and a local DuckDB file created solely for this evaluation. No real customer data, no production system, and no third-party service is read from, written to, or otherwise touched at any point in this work. The fault-injection plan is generated independently of, and is never read by, the local-checks or reconciliation code under test, so reported detection rates are not circular with the mechanism used to construct the faults. We release the full schema, data-generation scripts, fault-injection harness, detection code, and raw results as supplementary artifacts alongside this paper to support independent replication and extension. As with any small-scale pilot evaluation authored and run by the paper's own authors, we encourage readers to treat the specific numeric detection rates reported here as illustrative of the architectural argument rather than as a benchmark result to be compared directly against differently scaled or differently constructed evaluations elsewhere in the literature.

10. Conclusion

Hybrid OLTP-OLAP architectures are now the ordinary case rather than the exception, and the data quality tooling most teams reach for by default, per-record, single-system validation frameworks such as Deequ and Great Expectations, addresses one half of the resulting failure surface well while leaving the other half, faults that only appear when two systems are compared, structurally invisible. We operationalized Wang and Strong's data quality framework as six concrete metrics split cleanly along this local versus cross-system line, built a real reference implementation pairing local validation with cross-system reconciliation against live PostgreSQL and DuckDB instances, and showed in a controlled pilot evaluation that local validation alone plateaus at 60.0 percent overall fault detection while adding reconciliation reaches 98.5 percent, at negligible additional runtime cost. We also surfaced an unplanned but, we think, important finding: enforcing one quality dimension can measurably perturb another, meaning a hybrid system's six quality metrics should be monitored as an interacting system rather than as independent, isolated checks. We release our implementation and evaluation artifacts in the hope that this architectural framing, and not merely the specific numbers reported here, proves useful to practitioners building or auditing similar pipelines.

References

- [1] Wang, R. Y., & Strong, D. M. (1996). Beyond Accuracy: What Data Quality Means to Data Consumers. *Journal of Management Information Systems*, 12(4), 5-33.
- [2] Schelter, S., Biessmann, F., Lange, D., Rukat, T., Schmidt, P., Seufert, S., Brunelle, P., & Taptunov, A. (2019). Unit Testing Data with Deequ. *Proceedings of the 2019 ACM SIGMOD International Conference on Management of Data (SIGMOD '19)*, 1993-1996.
- [3] Great Expectations (GX Core). Open-source Python framework for declaring and validating data expectations. <https://greatexpectations.io/>
- [4] Transaction Processing Performance Council (2010). TPC BENCHMARK C Standard Specification, Revision 5.11.

- [5] Monte Carlo Data (2023). Data Downtime Nearly Doubled Year Over Year, Monte Carlo Survey Says (The State of Data Quality, 2023).
- [6] Debezium Community / Red Hat (2019). Reliable Microservices Data Exchange With the Outbox Pattern. Debezium Documentation, Outbox Event Router.
- [7] Abadi, D. (2012). Consistency Tradeoffs in Modern Distributed Database System Design: CAP is Only Part of the Story. *IEEE Computer*, 45(2), 37-42.