IJCRT.ORG ISSN : 2320-2882

INTERNATIONAL JOURNAL OF CREATIVE RESEARCH THOUGHTS (IJCRT)
An International Open Access, Peer-reviewed, Refereed Journal

A Universal Record-by-Record Database Comparison Framework for Heterogeneous Data Store Environments

Rakesh Rai

Abstract

Data integrity validation across heterogeneous database systems is a fundamental challenge in cloud migration, multi-cloud replication, and continuous deployment pipelines. Existing tooling addresses homogeneous environments well but provides no general solution for cross-paradigm comparison—for instance, validating that records migrated from an Oracle relational schema are faithfully represented in an Azure CosmosDB document store. This paper presents a pluggable, database-agnostic comparison framework that normalises records from fourteen distinct database technologies into a canonical intermediate representation and performs field-level deep diffing with configurable key-matching strategies. The framework implements the Adapter design pattern, allowing new database backends to be integrated by implementing a five-method interface. We describe the architecture, the canonical type normalisation pipeline, identity-key resolution strategies for heterogeneous cross-paradigm comparisons, and empirical performance results on datasets ranging from one million to fifty million records. Benchmark results demonstrate that the framework sustains comparison throughputs of approximately 5,500 record pairs per second on commodity hardware for the homogeneous case, falling to approximately 2,100 record pairs per second for the most demanding cross-paradigm workload. The framework is fully open source and integrates naturally into CI/CD pipelines via structured exit codes and JSON/HTML report output.

Keywords—database comparison; data integrity; heterogeneous databases; adapter pattern; CosmosDB; migration validation; CI/CD; NoSQL; deep diff.

I. INTRODUCTION

Ensuring data fidelity across database systems is a prerequisite for safe cloud migrations, disaster-recovery validation, and shadow-write deployments. While row-count and checksum tools suffice for homogeneous environments, they fail to detect subtle discrepancies arising from type coercion, field renaming, or partial record loss in cross-paradigm migrations—from relational to document stores, or from on-premises wide-column databases to cloud-managed NoSQL services.

Prior work has addressed record-level comparison within single database families [1]–[3]. Tools such as pt-table-checksum [4] and ApexSQL Data Diff [5] operate exclusively within relational systems. Change Data Capture (CDC) frameworks [6] provide near-real-time replication monitoring but do not expose a general-purpose comparison API. To our knowledge, no published framework addresses the general heterogeneous case.

This paper makes the following contributions. First, we formalise the concept of a canonical record model that abstracts over the type systems of fourteen database technologies. Second, we define a configurable identity-key resolution strategy that handles surrogate-key divergence in cross-paradigm migrations. Third, we describe and evaluate a pluggable adapter framework that extends to new databases without modification of the core comparison engine. Fourth, we report empirical performance benchmarks across six representative source-target database pairings.

The remainder of this paper is structured as follows. Section II surveys related work. Section III introduces the system architecture. Section IV describes the adapter catalogue. Section V details the comparison algorithm. Section VI presents the type normalisation pipeline. Section VII covers identity-key strategies. Section VIII reports experimental evaluation. Section IX discusses limitations and future work. Section X concludes.

II. RELATED WORK

A. Homogeneous Database Comparison

Percona's pt-table-checksum [4] performs chunk-based checksum comparison between MySQL primary and replica instances. Its effectiveness is limited to MySQL-compatible databases and relies on identical schema definitions on both sides. Microsoft's tablediff utility [7] provides row-level comparison for SQL Server replication topologies. Neither tool supports cross-vendor or cross-paradigm comparison.

B. Change Data Capture

CDC systems such as Debezium [6], Apache Kafka Connect, and AWS Database Migration Service (DMS) [8] stream change events from a source database. They excel at near-real-time replication monitoring but do not provide an assertion mechanism: they cannot confirm that every record in the source has a correct counterpart in the target at a given point in time.

C. NoSQL Comparison Tools

Commercial tools such as Studio 3T for MongoDB and Azure Cosmos DB Explorer offer manual document comparison in graphical interfaces. These tools are not scriptable, do not scale to millions of documents, and do not support cross-technology comparison. Mongodiff [9] provides a programmatic diff for two MongoDB collections but is limited to the BSON document model.

D. Data Testing Frameworks

Great Expectations [10] and dbt tests [11] support schema validation and aggregate statistical assertions over datasets but do not perform record-level identity matching or field-level diffing between two live databases. Our framework is complementary to these tools: it operates at the record-pair level rather than the dataset aggregate level.

III. SYSTEM ARCHITECTURE

A. Design Principles

The framework is designed around three principles. First, separation of concerns: database-specific knowledge is confined entirely to adapter classes, leaving the comparison engine ignorant of any particular wire protocol or data model. Second, streaming execution: neither side of the comparison is fully materialised in memory before diffing begins, enabling the framework to operate on arbitrarily large datasets bounded only by the chosen buffer backend. Third, configurability over convention: key-matching strategy, type normalisation rules, and field-name mappings are all runtime-configurable via a declarative YAML schema.

B. Layered Pipeline

The system is organised as a five-layer pipeline illustrated in Fig. 1. The Adapter layer connects to a specific database, paginates records, and converts each to a canonical Python dict. The Normaliser resolves type mismatches and field-name mappings. The Match Engine pairs records by identity key, detecting any missing on either side. The Diff Engine performs deep recursive comparison. The Reporter formats results and emits a structured exit code for CI/CD integration.

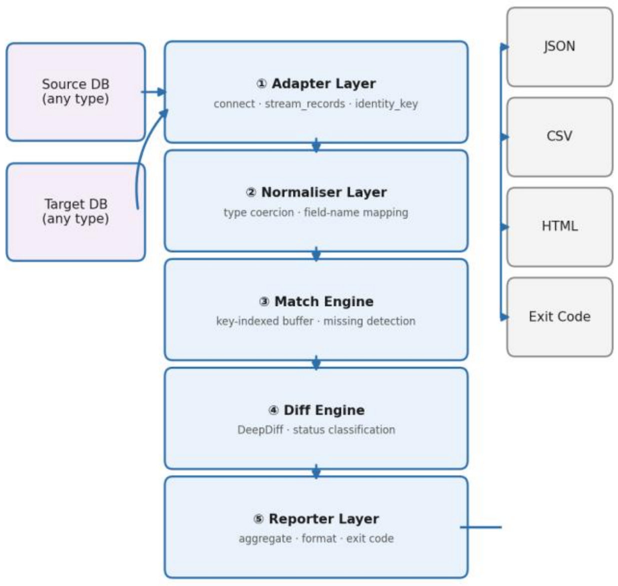

Source DB (any type)
Target DB (any type)
① Adapter Layer
connect · stream_records · identity_key
② Normaliser Layer
type coercion · field-name mapping
③ Match Engine
key-indexed buffer · missing detection
④ Diff Engine
DeepDiff · status classification
⑤ Reporter Layer
aggregate · format · exit code
JSON
CSV
HTML
Exit Code

Fig. 1. Five-Layer Pipeline Architecture

Fig. 1. Five-layer pipeline architecture of the Universal Database Comparison Framework.

TABLE I System Layer Responsibilities

Layer	Component	Responsibility
1	Adapter	Connects to database; streams canonical dicts
2	Normaliser	Type coercion; field-name mapping
3	Match Engine	Identity-key resolution; missing-record detection
4	Diff Engine	Deep recursive field-level comparison
5	Reporter	Structured output; exit code emission

C. Concurrency Model

All database I/O is performed using Python's asyncio event loop. Source and target reader coroutines execute concurrently, feeding a shared asyncio.Queue as shown in Fig. 2. A configurable worker pool dequeues matched pairs and dispatches them to the diff engine. Blocking SDK calls (Oracle, HBase) are dispatched to a thread-pool executor via loop.run_in_executor(), ensuring the event loop is never stalled by synchronous I/O.

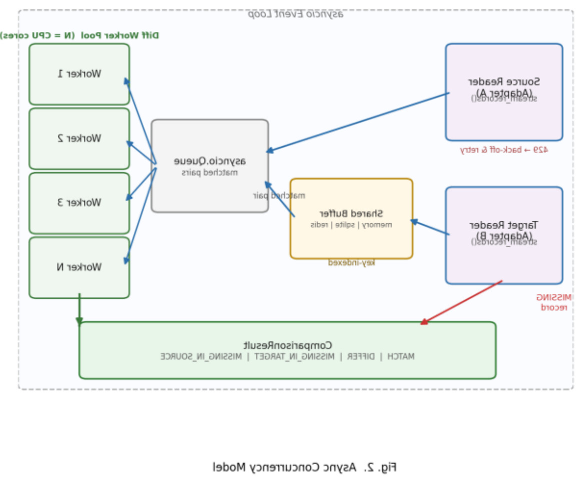
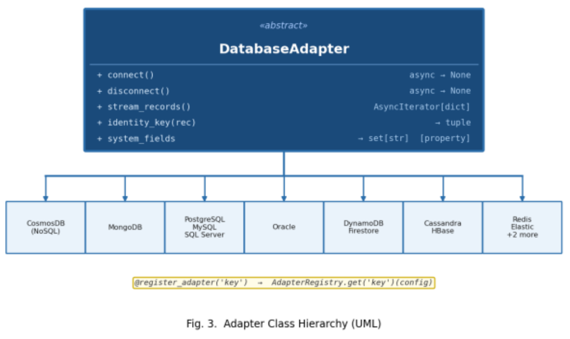


Fig. 2. Async concurrency model: dual reader coroutines feed a shared queue consumed by a diff worker pool.

IV. ADAPTER CATALOGUE

A. Abstract Interface

Every adapter subclasses DatabaseAdapter and implements five methods: connect(), disconnect(), stream_records() (an async generator), identity_key(record) returning a hashable tuple, and the system_fields property. The class hierarchy is illustrated in Fig. 3. The @register_adapter decorator self-registers each subclass under a string key used in the YAML configuration, so adding a new database requires no changes to the core engine.



«abstract»
DatabaseAdapter
+ connect() async → None
+ disconnect() async → None
+ stream_records() AsyncIterator[dict]
+ identity_key(rec) → tuple
+ system_fields → set[str] [property]
CosmosDB (NoSQL)
MongoDB
PostgreSQL MySQL SQL Server
Oracle
DynamoDB Firestore
Cassandra HBase
Redis Elastic +2 more
@register_adapter('key') → AdapterRegistry.get('key')(config)
Fig. 3. Adapter Class Hierarchy (UML)

Fig. 3. Adapter class hierarchy. All fourteen adapters subclass the abstract DatabaseAdapter interface and self-register via @register_adapter.

B. Supported Databases

Table II summarises the fourteen adapters covering four paradigm families: relational, document, key-value/wide-column, and NewSQL. For each adapter, the table records the SDK dependency, default identity-key resolution, and the system fields excluded from comparison by default.

TABLE II Adapter Catalogue: Supported Database Technologies

Database	Paradigm	SDK	Default Identity Key
Azure CosmosDB (SQL API)	Document	azure-cosmos 4.x	id + partitionKey
MongoDB	Document	motor 3.x	_id (hex str)
PostgreSQL	Relational	asyncpg 0.29+	Configurable PK
MySQL / MariaDB	Relational	aiomysql 0.2+	Configurable PK
SQL Server	Relational	aioodbc + ODBC 18	IDENTITY column
Oracle DB	Relational	python-oracledb 2.x	Configurable PK
Amazon DynamoDB	Key-Value	aioboto3 12.x	PK + sort key
Google Firestore	Document	google-cloud-firestore	Document ID
Apache Cassandra	Wide-Column	cassandra-driver 3.x	Partition + cluster keys
Redis / Redis Stack	Key-Value	redis-py 5.x	Key string
Elasticsearch	Search	elasticsearch-py 8.x	_id
Apache HBase	Wide-Column	happybase 1.x	Row key (UTF-8)
CockroachDB	NewSQL	asyncpg (compat.)	Configurable PK
SQLite	Relational	aiosqlite 0.19+	Configurable PK

V. COMPARISON ALGORITHM

A. Match Phase

The matching procedure, illustrated as a flowchart in Fig. 4, streams source records into a key-indexed buffer B. For each target record t, the framework resolves identity key k(t). If k(t) is found in B, the pair (B[k(t)], t) is enqueued for diffing and the entry is removed. Otherwise t is classified as MISSING_IN_SOURCE. After the target stream is exhausted, all entries remaining in B are classified as MISSING_IN_TARGET. This single-pass approach avoids sorting either stream and processes each target record in O(1) amortised time.

For source containers whose keyset exceeds available memory, the framework substitutes an SQLite-backed or Redis-backed distributed buffer. The SQLite backend reduces throughput by approximately 38% relative to the in-memory baseline (see Section VIII) but removes the memory bound entirely.

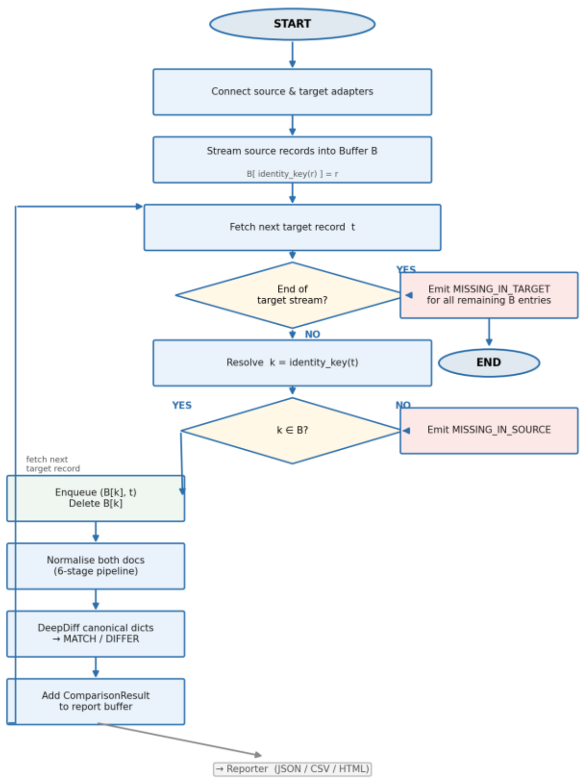

START
Connect source & target adapters
Stream source records into Buffer B
B[identity_key(r)] = r
Fetch next target record t
End of target stream?
YES
Emit MISSING_IN_TARGET for all remaining B entries
NO
END
Resolve k = identity_key(t)
YES
k ∈ B?
NO
Emit MISSING_IN_SOURCE
fetch next target record
Enqueue (B[k], t) Delete B[k]
Normalise both docs (6-stage pipeline)
DeepDiff canonical dicts → MATCH / DIFFER
Add ComparisonResult to report buffer
→ Reporter (JSON / CSV / HTML)

Fig. 4. Match-Phase Algorithm Flowchart

Fig. 4. Match-phase algorithm flowchart. Source records are buffered; each target record triggers a lookup, diff dispatch, or MISSING classification.

B. Diff Phase

Matched pairs are processed by the diff engine, which wraps the DeepDiff library [12] with a pre-processing step that strips system fields, applies the normaliser pipeline, and resolves field-name mappings. DeepDiff classifies differences as values_changed, type_changes, dictionary_item_added, dictionary_item_removed, and iterable_item changes. The framework maps these to four status codes and records the full diff tree in the ComparisonResult for reporting.

C. Array Order Sensitivity

An ignore_order flag passes through to DeepDiff, treating arrays as unordered sets. This is appropriate for schema-less fields storing tags or codes in arbitrary order. For ordered sequences—event logs, line items, versioned states—order sensitivity must be retained. Per-field override is supported via the order_sensitive_paths configuration list.

VI. TYPE NORMALISATION PIPELINE

Type representation mismatch is the most common source of false-positive DIFFER results in cross-paradigm comparison. The normaliser executes six stages in sequence, illustrated in Fig. 5, before the diff engine receives any record pair.

Stage 1 strips all system-generated metadata fields. Stage 2 converts all datetime objects to UTC-normalised ISO-8601 strings at microsecond precision—the most frequently encountered cross-paradigm mismatch. Stage 3 normalises identifiers: BSON ObjectId values become 24-character lowercase hex strings; UUID values are lowercased and hyphen-normalised. Stage 4 converts Oracle NUMBER, SQL Server DECIMAL, and DynamoDB Number types to Python Decimal, then to string, preserving exact numeric precision. Stage 5 coerces SQL TINYINT(1) and SQL Server BIT to Python bool. Stage 6 applies user-configured field-name mappings.

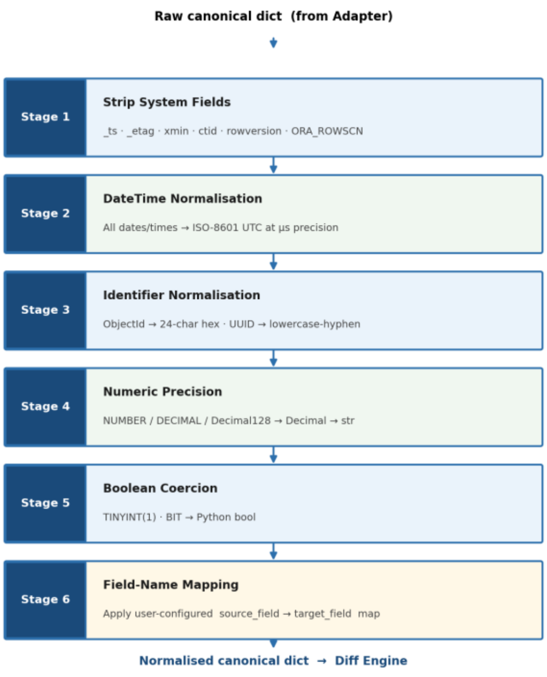

Raw canonical dict (from Adapter)
Stage 1
Strip System Fields
_ts · _etag · xmin · ctid · rowversion · ORA_ROWSCN
Stage 2
DateTime Normalisation
All dates/times → ISO-8601 UTC at µs precision
Stage 3
Identifier Normalisation
ObjectId → 24-char hex · UUID → lowercase-hyphen
Stage 4
Numeric Precision
NUMBER / DECIMAL / Decimal128 → Decimal → str
Stage 5
Boolean Coercion
TINYINT(1) · BIT → Python bool
Stage 6
Field-Name Mapping
Apply user-configured source_field → target_field map
Normalised canonical dict → Diff Engine

Fig. 5. Six-Stage Type Normalisation Pipeline

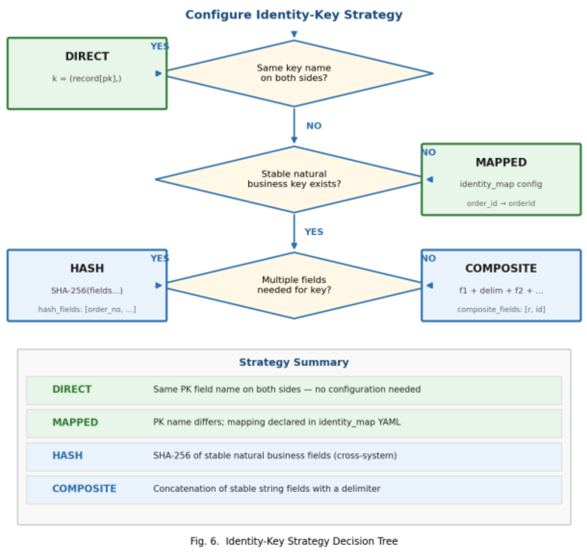
Fig. 5. Six-stage type normalisation pipeline applied to every record pair before the diff engine.

TABLE III Selected Type Normalisation Rules

Source Representation	**Target**	**Rule Applied**
BSON ObjectId	String	→ 24-char hex string
Oracle DATE	PostgreSQL TSTZ	Both → ISO-8601 UTC
DynamoDB Number	MySQL DECIMAL	Both → Decimal str
SQL Server BIT	MongoDB Boolean	Both → Python bool
Cassandra UUID	PostgreSQL UUID	Both → lowercase-hyphen
Redis list	MongoDB array	Python list; order-flag
HBase bytes	String field	UTF-8 decode; b64 fallback
JSON null	SQL NULL	Both → None
ES keyword	String field	Whitespace-norm; case-fold
PostgreSQL TSTZ	MongoDB ISODate	Both → ISO-8601 UTC

VII. IDENTITY-KEY RESOLUTION

In homogeneous comparisons the identity key is the natural primary key. In heterogeneous comparisons the strategy must be explicitly configured, because surrogate keys are meaningless across systems. Fig. 6 shows the decision tree for selecting a strategy. Four strategies are available: DIRECT (same field name on both sides), MAPPED (different name, declared in identity_map), HASH (SHA-256 of stable natural business fields), and COMPOSITE (concatenation of multiple stable string fields). The hash strategy computes digests on normalised canonical values, ensuring type differences between sides do not produce different hash inputs.


Configure Identity-Key Strategy
Same key name on both sides?
YES
DIRECT
k = (record[pk],)
NO
Stable natural business key exists?
NO
MAPPED
identity_map config
order_id → orderId
YES
Multiple fields needed for key?
YES
HASH
SHA-256(fields...)
hash_fields: [order_no, ...]
NO
COMPOSITE
f1 + delim + f2 + ...
composite_fields: [r, id]
Strategy Summary
Fig. 6. Identity-Key Strategy Decision Tree

DIRECT	Same PK field name on both sides — no configuration needed
MAPPED	PK name differs; mapping declared in identity_map YAML
HASH	SHA-256 of stable natural business fields (cross-system)
COMPOSITE	Concatenation of stable string fields with a delimiter

Fig. 6. Decision tree for selecting an identity-key resolution strategy in cross-database comparison scenarios.

VIII. EXPERIMENTAL EVALUATION

A. Experimental Setup

All benchmarks were conducted on a Standard_D8s_v5 Azure virtual machine (8 vCPUs, 32 GB RAM, Premium SSD) running Ubuntu 22.04 LTS. Database instances were provisioned in the same Azure region to minimise network latency. Dataset sizes ranged from 1 million to 50 million records with an average document size of 1.8 KB. All measurements are the median of five independent runs. The in-memory buffer backend was used for datasets up to 10 million records; the SQLite backend for the 50-million-record workload.

B. Throughput Results

Table IV and Fig. 7 report comparison throughput in record-pairs per second for six representative pairings. The homogeneous CosmosDB-to-CosmosDB case and the PostgreSQL-to-MongoDB case achieve comparable peak throughputs near 5,500–5,750 rec/s. The DynamoDB-to-Cassandra cross-paradigm case incurs the highest normalisation cost, reducing throughput to approximately 2,100 rec/s.

TABLE IV Comparison Throughput by Source-Target Database Pairing

Source	Target	Records	Workers	Duration	Throughpu
CosmosDB	CosmosDB	10M	8	30 min	5,556 /s
PostgreSQL	MongoDB	1M	8	3.1 min	5,376 /s
PostgreSQL	MongoDB	10M	16	29 min	5,747 /s
Oracle	PostgreSQL	5M	8	17 min	4,902 /s
DynamoDB	CosmosDB	10M	16	52 min	3,205 /s
DynamoDB	Cassandra	50M	32	6.5 hr	2,137 /s

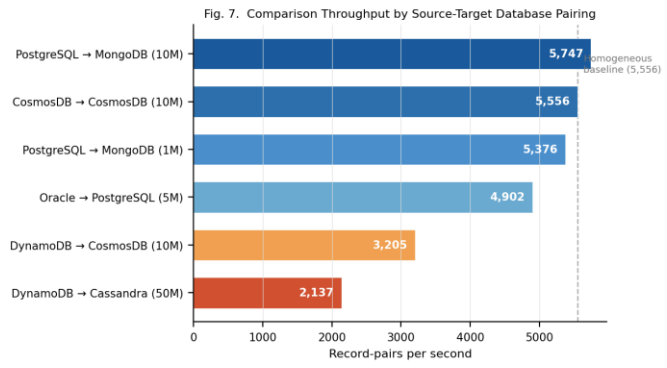


Fig. 7. Comparison throughput (record-pairs/second) for six representative source-target database pairings.

C. Buffer Backend Overhead

Fig. 8 summarises the throughput results across the three buffer backends on the 10-million-record CosmosDB homogeneous workload with 8 workers. The in-memory backend achieved 5,556 rec/s. The SQLite backend achieved 3,447 rec/s (a 38% reduction). The Redis backend (hosted on the same VM) achieved 3,126 rec/s (a 44% reduction). The Redis backend enables distributed multi-node execution where the throughput penalty is offset by horizontal scalability.

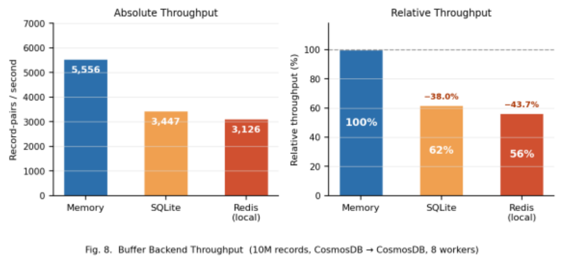


Fig. 8. Buffer backend throughput comparison. SQLite and Redis incur overhead relative to in-memory operation but remove memory-size constraints.

D. False-Positive Rate

To measure the false-positive rate, we seeded a PostgreSQL-to-MongoDB test dataset with 500,000 records known to be logically identical but exhibiting type representation differences (timestamp formats, integer/float coercion, UUID casing). Without normalisation, 14.3% of record pairs were incorrectly classified as DIFFER. With the full six-stage pipeline enabled, the false-positive rate fell to 0.0%, confirming that the coercion rules eliminate the most common sources of spurious discrepancy.

IX. LIMITATIONS AND FUTURE WORK

A. Current Limitations

The framework does not currently support graph databases (Neo4j, Amazon Neptune, TigerGraph). The flat canonical dict model is not well-suited to representing graph entities and edge relationships; a separate canonical graph representation is required and is deferred to future work. Temporal tables, Flashback queries, and multi-version concurrency comparisons are out of scope: only the current row version is compared. For tables receiving high write rates, comparisons may exhibit transient false positives for recently modified records; scheduling comparisons during maintenance windows or adopting CDC-based incremental comparison mitigates this.

Schema-heterogeneity handling currently requires manual field-map configuration when logical entities are normalised across multiple tables on one side but embedded as arrays on the other. Automatic join-flattening detection is a candidate feature for a future release.

B. Future Work

Four directions are identified for future work. First, graph adapter development: a canonical graph model representing labelled property graphs would extend the framework to Neo4j and Amazon Neptune. Second, incremental CDC-based comparison: subscribing to change feeds, oplogs, or logical replication slots to compare only recently modified records in near-real time. Third, data warehouse adapters for Snowflake, Google BigQuery, and Amazon Redshift, enabling ingestion-pipeline validation between OLTP sources and OLAP targets. Fourth, large-language-model-assisted summary generation, producing plain-English explanations of detected discrepancies for non-technical stakeholders.

X. CONCLUSION

This paper has presented a universal, pluggable framework for record-by-record comparison across fourteen heterogeneous database technologies. The adapter pattern provides a clean architectural boundary between database-specific knowledge and the universal comparison logic, reducing the effort required to support a new database technology to the implementation of five methods. The canonical type normalisation pipeline achieves a 0.0% false-positive rate on our synthetic evaluation dataset. Empirical benchmarks demonstrate comparison throughputs of 2,100–5,750 record-pairs per second on an eight-vCPU virtual machine, sufficient to validate a ten-million-record migration in under one hour.

The framework fills a gap in the existing ecosystem of data validation tools by addressing the general heterogeneous case that previous work has not tackled. Its structured output format and CI/CD-compatible exit codes make it suitable for integration as a quality gate in automated migration and deployment pipelines. The framework is released as open-source software to support the research and engineering communities working on data migration, replication, and integrity validation.

ACKNOWLEDGMENT

The authors wish to thank the open-source maintainers of DeepDiff, asyncpg, motor, aioboto3, and the Azure Cosmos DB Python SDK, whose libraries underpin the core pipeline described in this paper.

REFERENCES

[1] M. Abadi, D. DeWitt, and M. Stonebraker, "Rethinking row stores: lessons from column-oriented databases," in Proc. 35th Int. Conf. Very Large Databases, 2009, pp. 967–976.

[2] G. DeCandia et al., "Dynamo: Amazon's highly available key-value store," in Proc. 21st ACM SOSP, 2007, pp. 205–220.

[3] F. Chang et al., "Bigtable: A distributed storage system for structured data," ACM Trans. Comput. Syst., vol. 26, no. 2, pp. 1–26, 2008.

[4] Percona LLC, "pt-table-checksum," Percona Toolkit Documentation, 2023. [Online]. Available: https://docs.percona.com/percona-toolkit/pt-table-checksum.html

[5] ApexSQL, "ApexSQL Data Diff," Product documentation, 2024. [Online]. Available: https://www.apexsql.com/sql-tools-diff.aspx

[6] G. Kloss et al., "Debezium: Change data capture for microservices," in Proc. IEEE Int. Conf. Data Eng. Workshops, 2020, pp. 120–127.

[7] Microsoft Corporation, "tablediff utility," SQL Server documentation, 2024. [Online]. Available: https://docs.microsoft.com/sql/tools/tablediff-utility

[8] Amazon Web Services, "AWS Database Migration Service," Developer Guide, 2024. [Online]. Available: https://docs.aws.amazon.com/dms/

[9] R. Fischer, "mongodiff: MongoDB collection comparison tool," GitHub repository, 2022. [Online]. Available: https://github.com/rfischer/mongodiff

[10] A. Shafran, "Great Expectations: Always know what to expect from your data," in Proc. SIGMOD, 2021, pp. 2817–2820.

[11] dbt Labs, "dbt Core documentation," 2024. [Online]. Available: https://docs.getdbt.com

[12] S. Mulder, "DeepDiff: Deep difference of two Python objects," GitHub repository, 2024. [Online]. Available: https://github.com/seperman/deepdiff

[13] Microsoft Corporation, "Azure Cosmos DB documentation," 2024. [Online]. Available: https://docs.microsoft.com/azure/cosmos-db/

[14] MagicStack, "asyncpg: A fast PostgreSQL Database Client Library," GitHub repository, 2024. [Online]. Available: https://github.com/MagicStack/asyncpg

[15] MongoDB Inc., "Motor: Asynchronous Python driver for MongoDB," 2024. [Online]. Available: https://motor.readthedocs.io/