



One Framework, Many Engines: Source-Agnostic Automated Data Comparison across Heterogeneous Databases

Sasank Aramadaka

Senior QA Lead, USA

ABSTRACT: Organizations undertaking database migrations, platform consolidations, cloud data warehouse adoption, or merger driven data integration routinely face a problem that is deceptively difficult: confirming that two datasets, residing in two different database engines, with two different type systems, two different null handling conventions, and two different query languages, actually represent the same information. This article examines automated data comparison across heterogeneous database engines as a distinct engineering discipline, separate from both conventional software testing and conventional data quality management, though related to both. The article reviews the relational, data warehousing, data quality, and distributed systems literature that bears on the problem, enumerates the specific failure modes that arise when comparison logic implicitly assumes a single engine's semantics, and proposes a layered Source-Agnostic Comparison Framework, referred to throughout as SACF, built around an Abstraction Layer, a Normalization Layer, a Comparison Layer, a Discrepancy Classification Layer, and a Governance and Reporting Layer. The article describes the architectural components needed to operate SACF, concrete comparison techniques organized by data volume and consistency requirements, governance practices for sustaining the discipline over time, and two illustrative composite scenarios drawn from patterns commonly discussed in public industry commentary prior to March 2021. Several data tables are included to summarize engine specific quirks, comparison strategy selection, and failure mode mappings in a structured, referenceable form. All numeric figures outside the structured tables are presented as indicative, order of magnitude observations rather than as a validated empirical result, and the article closes with a discussion of limitations and directions for continued practice.

KEYWORDS: data comparison, data reconciliation, heterogeneous databases, database migration, data quality, data validation, ETL testing, schema mapping, NoSQL, data warehousing

I. INTRODUCTION

A database migration project, a cloud data warehouse adoption effort, or a merger driven data consolidation initiative is, from a purely engineering standpoint, an exercise in moving or synchronizing data from a source system to a target system. The actual engineering difficulty of such a project is rarely in the mechanics of moving data; extract, transform, and load tooling for this purpose has existed for decades and is well understood. The difficulty lies instead in answering, with confidence, a question that sounds simple but is not: did the data arrive correctly. Answering that question requires comparing the source and the target, and when the source and the target are different kinds of database engine, perhaps a relational system on one side and a columnar cloud warehouse or a document oriented store on the other, the comparison itself becomes a nontrivial engineering problem, because the two systems rarely agree on how to represent a null, how to order a floating point comparison, how to collate a string, or even what counts as a single logical row once denormalization or partitioning has been applied.

This article treats automated data comparison across heterogeneous database engines as a problem deserving of deliberate architectural treatment, rather than as an incidental detail to be solved anew on every migration project with a one off script. The central observation motivating this treatment is that comparison logic written against the assumptions of a single engine, even when that logic appears correct and passes its own tests, tends to produce either false positives, reporting a discrepancy where none meaningfully exists, or false negatives, missing a genuine discrepancy, the moment it is pointed at a second engine with different semantics. Because migration and consolidation projects almost always involve exactly this situation, at least one engine on each side of the comparison, the discipline of source agnostic comparison deserves the same deliberate architectural attention that contract testing, resilience engineering, and other now well established disciplines have already received in the broader software engineering literature.



The article's central claim is that reliable cross engine comparison requires separating four concerns that are frequently, and problematically, conflated into a single piece of ad hoc comparison code: how data is extracted from each engine in a form suitable for comparison, how that extracted data is normalized into a common representation that removes engine specific artifacts before any comparison occurs, how the normalized representations are actually compared against one another, and how the resulting discrepancies are classified and reported so that a human reviewer can distinguish a genuine data integrity problem from an artifact of the comparison process itself. A fifth concern, governance, determines whether the first four concerns remain correctly configured and trusted over the lifetime of a platform rather than only during the initial migration project that first motivated building them.

A further motivation for treating source agnostic comparison as a distinct discipline concerns organizational memory. Many organizations rediscover the same type coercion and null semantics pitfalls on every new migration project, because the lesson learned fixing a specific discrepancy on one project rarely survives, in any durable form, beyond the individual engineer who happened to debug it. This article treats the accumulation of that durable, reusable knowledge, encoded directly into the Normalization Layer's rule set described in Section 5.2 rather than left as tacit knowledge in any individual's memory, as being as important a contribution as any single comparison technique considered on its own, since an organization that solves the same engine specific quirk repeatedly, from scratch, on every project has not actually built a capability at all, only repeated an exercise.

The remainder of this article is organized as follows. Section 2 reviews the relational, data warehousing, data quality, and distributed systems literature that informs the proposed framework. Section 3 states the problem more precisely, enumerates failure modes specific to heterogeneous comparison, and includes a structured table summarizing comparison relevant characteristics across several database engine families. Section 4 sets out design principles. Section 5 introduces the Source-Agnostic Comparison Framework. Section 6 describes the architecture of a SACF capable platform, including a table of engine specific type and null handling quirks. Section 7 discusses concrete comparison techniques, including a comparison strategy selection matrix and a failure mode mapping table. Section 8 presents two illustrative composite scenarios. Section 9 discusses evaluation criteria, including a maturity table. Section 10 addresses governance and reporting considerations, including a roles and responsibilities table. Section 11 discusses limitations. Section 12 proposes future work, and Section 13 concludes.

II. BACKGROUND AND RELATED WORK

2.1 The Relational Model and the Illusion of Uniformity

Codd's original formulation of the relational model established a mathematically precise foundation for representing and querying structured data, and Date's widely used textbook treatment of relational database systems has, for decades, trained generations of practitioners to think of a relational database's behavior as governed by a single, consistent standard. In practice, every relational database engine implements the standard with its own extensions, its own interpretations of ambiguous corners of the standard, and its own performance driven deviations, particularly around null comparison semantics, floating point representation, and string collation. The illusion of uniformity that the relational model's clean theoretical foundation creates is, this article argues, one of the primary reasons comparison logic written for one relational engine fails silently or loudly when pointed at another, since engineers reasonably but incorrectly assume that two systems both calling themselves relational databases will agree on these corner cases.

2.2 Data Warehousing and ETL Validation Traditions

Inmon's foundational treatment of building the data warehouse, and Kimball and Ross's widely adopted dimensional modeling methodology, both describe extract, transform, and load processes that move data from operational source systems into an analytical target, and both traditions have, since their origin, included some notion of load validation, typically focused on row count reconciliation and control total comparison between source extracts and loaded target tables. This tradition provides an important starting point for the comparison problem this article addresses, but it was developed primarily for comparing a source extract against a target within the same general data warehousing toolchain, rather than for comparing two live, independently queryable database engines of fundamentally different kinds against one another, which is the more general problem this article's framework addresses. Linstedt and Olschimke's later treatment of the Data Vault modeling approach extends this tradition with an explicit emphasis on auditability and historized comparison, which this article's Governance and Reporting Layer, described in Section 5.5, draws upon.



2.3 Data Quality Dimensions

Wang and Strong's influential treatment of data quality from the data consumer's perspective established a now standard vocabulary of data quality dimensions, including accuracy, completeness, consistency, and timeliness, that subsequent literature, including Redman's treatment of data quality for the information age, Batini and Scannapieco's methodological survey, and the DAMA International Data Management Body of Knowledge, has elaborated considerably. English's treatment of improving data warehouse and business information quality, and Loshin's later treatment of master data management, both connect this data quality vocabulary directly to the kind of large scale data integration projects in which cross engine comparison typically arises. This article borrows the data quality dimension vocabulary directly, discussed further in Section 3, and argues that cross engine comparison is best understood as a specific, automatable technique for continuously measuring several of these dimensions, particularly accuracy and completeness, across a system boundary rather than within a single system.

2.4 NoSQL, Distributed Systems, and the CAP Theorem

Sadalage and Fowler's treatment of NoSQL systems describes the proliferation of document oriented, key value, wide column, and graph database engines, each with its own data model and its own consistency guarantees, that has made the heterogeneous comparison problem substantially more acute than it was when most comparisons occurred between two relational engines. Brewer's original conjecture, later formalized by Gilbert and Lynch as what is commonly called the CAP theorem, establishes that a distributed data store cannot simultaneously guarantee consistency, availability, and partition tolerance, and many NoSQL engines deliberately favor availability and partition tolerance over strict consistency, which has a direct and underappreciated consequence for comparison: a comparison performed against an eventually consistent store may legitimately observe a transient discrepancy that reflects only replication lag rather than any genuine data integrity defect, a distinction this article's Discrepancy Classification Layer, described in Section 5.4, is specifically designed to make.

2.5 Consistency Models and Comparison Semantics

Related to the discussion above, the broader distributed systems literature on consistency models, treated comprehensively in Kleppmann's survey of designing data intensive applications, establishes that comparison semantics cannot be defined independently of the consistency model of the systems being compared. A comparison performed at a single instant against two systems with different replication and consistency characteristics is implicitly comparing two different, possibly only loosely related, notions of what the current state of the data actually is. This article's framework treats the choice of comparison timing, discussed in Section 7.3, as a first class design decision rather than an incidental detail, precisely because of this consideration.

2.6 Record Linkage and Duplicate Detection

Elmagarmid, Ipeirotis, and Verykios's survey of duplicate record detection, and the broader record linkage literature summarized by Rahm and Do in their treatment of data cleaning problems and current approaches, address a problem adjacent to, but distinct from, the one this article treats: determining whether two records, possibly represented differently, refer to the same real world entity. Cross engine comparison as this article treats it typically assumes that a reliable key exists linking a source record to its target counterpart, since the systems being compared are usually two representations of the same underlying system rather than two independently sourced datasets describing overlapping but distinct populations. Where no reliable key exists, the record linkage literature's techniques become directly relevant, and this article notes the connection in Section 7.8 without attempting to fully absorb that literature's considerably broader scope.

2.7 Information Integration and Schema Mapping

Halevy, Rajaraman, and Ordille's retrospective on data integration, and Bernstein and Haas's treatment of information integration in the enterprise, describe the broader problem of reconciling schemas that were designed independently of one another, including the schema mapping and transformation logic needed to relate corresponding elements across two systems. This literature informs the Normalization Layer described in Section 5.2 of this article's framework, since a schema mapping, however it is derived, is typically a necessary precondition for any meaningful comparison between two differently structured systems, and errors in that mapping are themselves a distinct and important failure mode, discussed in Section 3.2.

2.8 Continuous Delivery and Pipeline Integrated Data Validation

Humble and Farley's treatment of continuous delivery, and Newman's later discussion of data migration strategy within the context of decomposing a monolith into microservices, both argue for integrating validation activity directly into an



automated pipeline rather than treating it as a manually scheduled, separate phase. This article extends that argument specifically to cross engine data comparison, recommending in Section 7.9 that comparison be run continuously, or at minimum on every significant data movement event, rather than only once at the conclusion of a migration project, a recommendation that follows directly from the pipeline integrated validation tradition this section reviews.

2.9 A Note on Terminology

This article uses the term comparison deliberately, in preference to the broader term reconciliation, which carries a specific, finance oriented connotation in much of the business literature, and in preference to the narrower term testing, which typically implies a pre production, one time activity. Comparison, as used throughout this article, refers specifically to the automated determination of agreement or disagreement between two datasets residing in different systems, and is treated as a continuous, production facing activity, discussed in Section 7.9, rather than solely a pre production verification step. Where this article uses the term discrepancy, it refers to any detected difference between source and target, prior to the classification described in Section 5.4 that determines whether the difference reflects a genuine defect or a benign, expected variation. This distinction between a discrepancy and a defect is maintained carefully throughout the article, since conflating the two is itself identified as a contributing factor behind the classification and reporting failure mode discussed in Section 3.2.

2.10 Relationship to Change Data Capture

Change data capture techniques, which detect and propagate individual row level changes from a source system as they occur rather than through periodic bulk extraction, are closely related to, but distinct from, the comparison problem this article addresses. Change data capture is primarily a replication mechanism, concerned with moving changes efficiently from source to target, while comparison, as this article treats it, is concerned with independently verifying that the result of replication, however it was performed, is actually correct. The two are frequently used together in practice, since a change data capture pipeline can feed the continuous, incremental comparison strategy described in Table 4, but an organization should not conflate the two, since a change data capture pipeline that is itself defective, for example by silently dropping a subset of change events under load, would produce exactly the kind of discrepancy that only an independent comparison process, running separately from the replication pipeline itself, is positioned to catch.

2.11 Summary of the Gap in Prior Work

The literature reviewed above provides strong individual foundations for relational theory, data warehousing validation, data quality measurement, distributed consistency models, and schema mapping, but no single body of work treats the specific combination this article addresses: an architecture for comparing data continuously and reliably across database engines of fundamentally different kinds, in a way that correctly accounts for each engine's distinct type system, null handling, consistency model, and query interface, while remaining reusable across many migration and integration projects rather than rebuilt bespoke for each one. This article's framework, introduced in Section 5, is offered as a response to that gap.

III. PROBLEM STATEMENT

3.1 Why Heterogeneous Comparison Is Hard

Table 1 summarizes several comparison relevant characteristics across common database engine families, to make concrete why a single piece of comparison logic cannot be written once and reused without adaptation across every engine a migration or integration project might involve.



Table.1. Comparison relevant characteristics across database engine families

Engine Family	Representative Systems	Native Interface	Query	Comparison Relevant Quirk
Row oriented relational	Oracle Database, Microsoft SQL Server, PostgreSQL	SQL		Collation and null ordering vary by configuration and vendor default
Columnar analytical warehouse	Amazon Redshift, Snowflake, Google BigQuery	SQL with vendor extensions		Approximate aggregate functions and distributed query result ordering are non deterministic unless explicitly sorted
Document oriented store	MongoDB, Couchbase	Query language or API specific to the engine		Flexible schema allows a field to be absent, null, or of a different type across documents
Key value store	Redis, Amazon DynamoDB	API based access rather than declarative query		No native concept of a row scan without a defined access pattern, complicating full dataset comparison
Wide column store	Apache Cassandra, Apache HBase	Query language specific to the engine		Eventual consistency across replicas can produce transient read discrepancies unrelated to data integrity
Batch oriented big data platform	Apache Hive, Apache Spark on distributed storage	SQL like query language over files		Underlying file format and partitioning scheme affects both performance and apparent row ordering
Graph database	Neo4j, Amazon Neptune	Graph specific query language		Entities and relationships do not map directly onto a row and column comparison model

Beyond the characteristics summarized in the table, several further properties make heterogeneous comparison difficult in practice.

- Type systems differ not only in name but in behavior; a numeric type that rounds consistently in one engine may truncate in another, and a timestamp type that stores time zone information in one engine may store only a naive local time in another.
- Null handling conventions differ, both in how null compares to null, which some engines treat as unknown and therefore not equal even to itself, and in how null sorts relative to non null values.
- The notion of a comparable unit of data differs across data models; a relational row, a document, a key value pair, and a graph node do not correspond to one another in any uniform way, and a comparison framework must define, for each specific pairing of source and target, what the corresponding comparable unit actually is.
- Consistency models differ, as discussed in Section 2.4, meaning that a comparison performed against two systems with different replication characteristics may observe differences that reflect timing rather than any underlying data defect.

3.2 Failure Modes Enumerated

Building on the characteristics above, this article enumerates failure mode families specific to heterogeneous data comparison.



- Type coercion failures, in which values that are logically equivalent are reported as different because of a type system mismatch, such as a floating point value that differs only in trailing precision, or a date value that differs only in time zone representation.
- Null semantics failures, in which the comparison logic's treatment of null does not match either source or target engine's actual null handling convention, producing either a false discrepancy or a missed one.
- Schema mapping failures, in which the mapping between source and target structures is itself incorrect or incomplete, causing a comparison to check the wrong corresponding elements against each other, or to silently omit elements that have no identified counterpart at all.
- Timing and consistency failures, in which a comparison performed at a single instant, or across systems with different consistency models, reports a discrepancy that reflects replication lag or transaction timing rather than a genuine data integrity defect.
- Scale and sampling failures, in which a comparison strategy that is accurate at small volume either becomes computationally infeasible at large volume, or is replaced by a sampling strategy that systematically misses the specific subset of records most likely to contain a genuine defect.
- Classification and reporting failures, in which genuine discrepancies are detected but reported in a form that does not allow a human reviewer to efficiently distinguish a critical data integrity issue from a benign, expected variation, discussed further in Section 5.4.

3.3 The Comparison Gap

This article defines the comparison gap, following the pattern of related assurance literature, as the difference between the confidence a project team believes it has that a migrated or integrated dataset matches its source, typically based on a successful one time comparison run near the end of a migration project, and the confidence actually warranted given that subsequent changes to either the source or the target system, including routine application changes unrelated to the original migration, can silently reintroduce the very discrepancies the original comparison was designed to catch. The comparison gap tends to be largest precisely when it is most dangerous: well after a migration project's dedicated team has disbanded and moved on to other work, at which point no one may be positioned to notice a newly introduced discrepancy at all.

3.4 Illustrative Cost of Undetected Comparison Gaps

- The direct remediation cost of a late discovered data discrepancy, including the effort to trace its origin, which grows considerably once the discrepancy has persisted long enough that multiple subsequent changes to either system must be examined as possible causes.
- The decision quality cost of relying on inaccurate data for a period before a discrepancy is discovered, which is particularly significant for data feeding financial reporting, regulatory submissions, or executive decision making, where a decision made on the basis of undetected bad data cannot simply be undone once discovered.
- The trust cost of a discrepancy discovered by a business stakeholder rather than by the data platform team itself, which tends to produce lasting skepticism toward the platform that is disproportionate to the technical severity of the specific discrepancy involved.
- The opportunity cost of manual comparison effort, since many organizations without an automated comparison capability resort to ad hoc, manually constructed comparison queries whenever a concern arises, an approach that is both slow and prone to exactly the type coercion and null semantics errors described in Section 3.2.

3.5 Illustrative Use Cases Triggering Heterogeneous Comparison

Although this article treats database migration as the motivating scenario throughout, several distinct organizational situations give rise to the same underlying comparison problem, and it is useful to enumerate them explicitly so that the framework proposed in Section 5 is understood as broadly applicable rather than narrowly scoped to migration alone.

- A planned platform migration, in which an organization deliberately moves a workload from one database engine to another, for reasons of cost, scalability, or vendor consolidation, and must confirm that the migrated data matches the original before decommissioning the source system.
- A cloud data warehouse adoption initiative, in which analytical workloads are moved from an on premises relational system to a cloud native columnar warehouse, often incrementally, with both systems running in parallel for an extended transition period during which ongoing comparison is essential.
- A merger or acquisition driven consolidation, in which two organizations' independently operated systems, frequently built on entirely different database engines, must be reconciled into a single, unified data environment.



- An ongoing dual write or change data capture replication architecture, in which a system intentionally maintains two copies of its data across different engines for reasons such as read scaling or disaster recovery, and must continuously confirm that the two copies remain in agreement.
- A regulatory or audit driven data lineage verification exercise, in which an organization must demonstrate, after the fact, that data reported externally matches the data as it existed in its original system of record, regardless of how many intermediate systems and transformations the data passed through along the way.

IV. DESIGN PRINCIPLES

4.1 Separate Extraction, Normalization, and Comparison into Distinct Stages

Comparison logic that extracts data from an engine, normalizes its representation, and compares it against another engine's data all within a single, undifferentiated piece of code tends to hide engine specific assumptions inside logic that appears, superficially, to be generic. This article recommends that these three stages be implemented as distinct, independently testable components, consistent with the layered structure introduced in Section 5.

4.2 Normalize Before Comparing, Never Compare Raw Representations Directly

Following directly from the principle above, no comparison should ever be performed directly against each engine's raw, native representation of a value. Instead, every value should first be normalized into a common, engine neutral representation, for example converting every numeric type into a single canonical numeric representation with an explicitly defined precision and rounding rule, before any comparison logic is applied.

4.3 Treat Tolerance as an Explicit, Documented Parameter

Because exact equality is frequently neither achievable nor meaningful across engines with different floating point representations or timestamp precisions, this article recommends that any tolerance applied during comparison, such as an acceptable floating point difference or an acceptable timestamp precision difference, be an explicit, documented parameter of the comparison configuration for a given data element, rather than an implicit default buried inside the comparison engine's code, so that a reviewer can audit exactly how strict a given comparison actually was.

4.4 Classify Discrepancies Before Reporting Them

Consistent with the general principle, discussed in related assurance literature, of separating detection from prioritization, this article recommends that every detected discrepancy be passed through an explicit classification step, described in Section 5.4, before being surfaced to a human reviewer, so that transient, timing related differences are not reported with the same urgency as a genuine data integrity defect.

4.5 Calibrate Comparison Depth to Data Criticality and Volume

Not every dataset warrants full, row level, every field comparison at every available frequency. A principle of calibrated comparison depth holds that the completeness and frequency of comparison should scale with a dataset's business criticality and its volume, following the proportionate investment reasoning common throughout the broader assurance literature, discussed further in the comparison strategy selection matrix in Section 7.3.

4.6 Make the Comparison Framework Reusable Across Projects, Not Bespoke to One

Because a single organization typically undertakes many migration and integration projects over time, comparison logic built as a one off script for a single project represents a largely wasted investment. This article recommends building the Abstraction Layer described in Section 5.1 as a genuinely reusable platform capability, with new engine adapters added incrementally as new source or target systems are encountered, rather than rebuilding comparison logic from scratch for each project.

4.7 Preserve Comparison History as an Auditable Artifact

Following the auditability emphasis found in the Data Vault literature discussed in Section 2.2, this article recommends that every comparison run, together with its configuration, its results, and any subsequent reviewer classification of its discrepancies, be retained as a durable, queryable artifact, so that a later reviewer can determine not only the current state of agreement between two systems but the history of that agreement over time.

4.8 Design the Framework to Fail Loudly When It Cannot Compare Confidently

Where a comparison framework encounters a data element it cannot normalize with confidence, for example a type combination it has not been explicitly configured to handle, this article recommends that the framework report this as an explicit coverage gap, distinct from a discrepancy, rather than silently skipping the element or, worse, reporting a



false match. A framework that fails loudly in the face of its own uncertainty is considerably more trustworthy than one that fails silently.

4.9 Prefer Declarative Configuration Over Imperative Comparison Code

Where comparison logic is expressed imperatively, as a bespoke program written for a specific pair of datasets, the resulting logic is difficult to audit, difficult to reuse, and difficult to adapt when the underlying schema changes. This article recommends that normalization rules, tolerance parameters, and key definitions be expressed declaratively, as data describing the comparison rather than as code implementing it, so that a reviewer unfamiliar with the underlying programming language can nonetheless audit exactly what a given comparison configuration will and will not catch, and so that the same comparison engine can execute many different declared configurations without modification.

V. THE SOURCE-AGNOSTIC COMPARISON FRAMEWORK

This section introduces the Source-Agnostic Comparison Framework, SACF, structured as five layers summarized in Table 2 before each is discussed in turn.

Table.2. The five layers of SACF

Layer	Primary Concern	Representative Technique	Primary Artifact
Abstraction Layer	Uniform extraction across heterogeneous engines	Engine specific adapters behind a common interface	Adapter catalog
Normalization Layer	Removing engine specific representational artifacts	Canonical type and null representation rules	Normalization rule set
Comparison Layer	Determining agreement between normalized representations	Key based matching with configurable tolerance	Comparison result set
Discrepancy Classification Layer	Distinguishing genuine defects from benign variation	Rule based and reviewer informed classification	Classified discrepancy log
Governance and Reporting Layer	Sustaining trust in the practice over time	Coverage tracking and stakeholder reporting	Comparison coverage registry

5.1 Abstraction Layer

The Abstraction Layer is responsible for extracting data from any supported source or target engine through a common interface, implemented as a set of engine specific adapters, consistent with the reusability principle in Section 4.6. Each adapter is responsible for translating the engine's native query interface, summarized in Table 1, into a uniform extraction request, and for surfacing any engine specific metadata, such as a column's native type, needed by the Normalization Layer that follows.

5.2 Normalization Layer

The Normalization Layer applies the canonical representation principle described in Section 4.2, converting each extracted value into a common, engine neutral form according to an explicit, documented set of normalization rules. These rules must address, at minimum, numeric precision and rounding, timestamp time zone and precision handling, string collation and trimming behavior, and the specific null handling convention of each engine involved, since, as discussed in Section 3.1, these conventions vary considerably.

5.3 Comparison Layer

The Comparison Layer matches normalized source and target records using a defined key, consistent with the record linkage discussion in Section 2.6, and applies the configurable tolerance described in Section 4.3 to determine whether corresponding fields agree. This layer produces a comparison result set that records, for every matched pair, every evaluated field, and every unmatched record on either side, a result sufficiently granular to support the classification that follows.



5.4 Discrepancy Classification Layer

The Discrepancy Classification Layer evaluates every discrepancy produced by the Comparison Layer against a set of classification rules intended to distinguish genuine data integrity defects from benign variation, such as the transient timing related differences discussed in Section 2.4, or expected, intentional transformation differences introduced deliberately by an extract, transform, and load process. Discrepancies that cannot be confidently classified by rule are routed to human review, consistent with the fail loudly principle in Section 4.8.

5.5 Governance and Reporting Layer

The Governance and Reporting Layer, discussed further in Section 10, maintains the comparison coverage registry described in Table 2, tracking which datasets are currently under active, scheduled comparison, at what frequency and depth, and surfaces this information, together with discrepancy trends, to stakeholders beyond the engineering team responsible for building the comparison platform itself.

5.6 Interaction Among the Layers

As with related layered frameworks in the broader assurance literature, SACF's five layers interact continuously rather than operating as a one time pipeline. A classification pattern repeatedly observed by the Discrepancy Classification Layer, for example a specific field that is consistently flagged as a false positive, typically prompts a refinement of the Normalization Layer's rules for that field, which in turn reduces the burden on future classification. The Governance and Reporting Layer observes this interaction across all four other layers and is responsible for ensuring that lessons learned in one comparison project are carried forward into the reusable platform rather than remaining confined to that project alone.

5.7 Scaling the Framework by Organizational Size

SACF is described in terms of five layers and a set of design principles rather than a single prescribed toolset, so that it can scale to organizations of differing size and migration frequency. A smaller organization undertaking an infrequent, single migration project might implement the five layers using lightweight, largely manual processes, such as a shared configuration file serving as an early Normalization Rule Engine and a simple scheduled script serving as an early Matching and Comparison Engine, consistent with the calibrated comparison depth principle in Section 4.5. A larger organization undertaking frequent migrations, or operating an ongoing dual write architecture as described in Section 3.5, will typically need dedicated tooling and a dedicated governance function, as described in Section 10, but the underlying five layer structure, and the mapping of techniques to failure modes summarized in Table 5, remains applicable at either scale.

VI. ARCHITECTURE OF A SACF CAPABLE PLATFORM

6.1 The Adapter Catalog

The Adapter Catalog, implementing the Abstraction Layer described in Section 5.1, maintains the set of engine specific adapters the platform currently supports, together with each adapter's known limitations, such as data types it cannot yet extract with full fidelity, so that a project team evaluating a new comparison can quickly determine whether existing adapters are sufficient or whether new adapter development is required.

6.2 The Normalization Rule Engine

The Normalization Rule Engine, implementing the Normalization Layer described in Section 5.2, applies the documented normalization rules to extracted data, and is itself configurable per data element, since, consistent with the calibrated comparison depth principle in Section 4.5, different data elements may warrant different precision or tolerance handling depending on their business significance.

6.3 The Matching and Comparison Engine

The Matching and Comparison Engine, implementing the Comparison Layer described in Section 5.3, performs the key based matching and field level comparison across normalized datasets, and is designed to scale across the full range of data volumes summarized in the comparison strategy selection matrix in Section 7.3, from small, fully row level comparisons to large scale, partitioned, aggregate based comparisons.



6.4 The Discrepancy Classifier

The Discrepancy Classifier, implementing the Discrepancy Classification Layer described in Section 5.4, applies both deterministic classification rules and accumulated reviewer feedback to incoming discrepancies, and maintains the classified discrepancy log that supports the auditable history principle in Section 4.7.

6.5 The Comparison Coverage Registry

The Comparison Coverage Registry, implementing part of the Governance and Reporting Layer described in Section 5.5, records which datasets, across every active migration and integration project, are currently under scheduled comparison, and surfaces any dataset that has fallen out of active comparison coverage, for example because a scheduled job began silently failing, as its own distinct category of alert, following the same second order health check reasoning found in related assurance literature.

6.6 Type and Null Handling Quirks Across Engines

Table 3 summarizes, at an illustrative level, several of the specific type and null handling quirks the Normalization Rule Engine must be configured to address when comparing across the engine families introduced in Table 1.

Table.3. Illustrative type and null handling quirks by engine family

Engine Family	Null Comparison Behavior	Default Floating Point Behavior	Default String Collation Behavior
Row oriented relational	Null is not equal to null under standard SQL three valued logic	Fixed or arbitrary precision decimal types generally preserve exact values	Collation is configurable per column or database, and defaults vary by vendor
Columnar analytical warehouse	Null handling generally follows SQL convention but may differ in approximate aggregate functions	Floating point types may be stored with reduced precision for performance	Collation is frequently fixed at the warehouse level rather than configurable per column
Document oriented store	A missing field, an explicit null value, and an empty value are three distinct states	Numeric types are often represented using a general purpose numeric format rather than a fixed decimal type	String comparison is typically byte order based unless a specific collation feature is explicitly enabled
Key value store	The concept of null is often undefined, since a missing key simply has no value at all	Numeric handling depends entirely on the client library's interpretation of the stored byte representation	Collation is generally not a native concept, since comparison is typically limited to exact key lookup
Wide column store	Null is typically represented as the simple absence of a column for a given row	Floating point precision depends on the specific column type chosen at schema design time	Collation is generally determined by raw byte ordering rather than a locale aware default
Batch oriented big data platform	Null handling generally follows the convention of the query engine layered on top of the underlying files	Precision depends on the underlying file format's numeric encoding	Collation behavior often depends on the specific query engine rather than the storage format itself

6.7 The Reconciliation Dashboard

The Reconciliation Dashboard aggregates output from the components above into a single view, organized by project and by dataset, surfacing current comparison coverage, recent discrepancy trends, and outstanding classification



backlog, in a form intended to be legible to both the engineering team and the business stakeholders who ultimately depend on the compared data.

6.8 Deployment Topology Considerations

The Matching and Comparison Engine described in Section 6.3 can be deployed either close to the source and target systems themselves, minimizing data movement by pushing as much comparison logic as possible down into each engine's own native query capability, or as a centralized service that pulls normalized data from both sides into a dedicated comparison environment. The push down approach generally performs better at scale, since it avoids moving large volumes of data across a network purely for comparison purposes, but it requires each engine adapter to translate comparison logic into that engine's native query language, which is a greater implementation burden than the centralized approach, where comparison logic is written once against the normalized representation regardless of source. Organizations typically begin with the centralized approach for its simplicity and migrate performance critical comparisons to a push down approach only once volume growth makes the centralized approach impractical, consistent with the calibrated investment reasoning found throughout this article.

6.9 Validating the Comparison Platform Itself

As with the comparison platforms built for other assurance disciplines, the components described above are themselves software and are subject to the same risk of latent defect as the systems they compare. SACF recommends that the platform maintain a small set of known, deliberately constructed test datasets with precisely understood discrepancies deliberately injected into them, run periodically through the full comparison pipeline, so that a silent failure in the Normalization Rule Engine or the Matching and Comparison Engine itself can be detected before it causes the platform to miss a genuine discrepancy in a real comparison project.

6.10 Multi Region and Multi Tenant Considerations

Organizations operating across multiple geographic regions, or serving multiple tenants whose data must remain logically or physically separated, introduce an additional dimension to the architecture described above. The Adapter Catalog and Normalization Rule Engine described in Sections 6.1 and 6.2 can generally remain shared, global components, since the engine specific quirks they encode do not vary by region or tenant, but the Matching and Comparison Engine and the Comparison Coverage Registry described in Sections 6.3 and 6.5 frequently must be partitioned, so that a comparison run for one region or tenant cannot inadvertently access another's data, and so that coverage reporting can be meaningfully scoped per region or tenant for stakeholders who are only responsible for their own portion of the organization's data estate. Organizations that defer this partitioning until after an initial, single region implementation has already been built have reported, in informal discussion, a nontrivial retrofit cost, which argues for treating multi region and multi tenant structure as a first class concern from the outset wherever an organization's operating model makes it a foreseeable requirement.

6.11 The Tolerance Audit Log

A complementary artifact to the classified discrepancy log described in Section 6.4 is a dedicated tolerance audit log, which records every change ever made to the Normalization Rule Engine's tolerance parameters, together with who made the change, when, and the stated justification. Because tolerance parameters directly determine the sensitivity of every subsequent comparison, as discussed in Section 4.3, the history of how those parameters evolved is itself a governance relevant artifact, distinct from the discrepancy history, and this article recommends maintaining it as a first class, separately queryable record rather than allowing it to be scattered across informal channels such as chat messages or code review comments, which are considerably harder to retrieve systematically during a later audit or investigation.

6.12 Logging and Observability for the Comparison Platform

Because the comparison platform itself frequently handles sensitive data as a matter of course, extracting and comparing values from production systems of record, its own logging and observability practices deserve the same scrutiny applied to any other production system handling sensitive data, rather than being treated as a lower stakes, internal tooling concern. This article recommends that the platform's logs record metadata about a comparison run, such as which datasets were compared, how many records were examined, and how many discrepancies were found, in detail sufficient to support the governance reporting described in Section 10, while avoiding logging the actual sensitive field values involved in a specific discrepancy except within the access controlled classified discrepancy log described in Section 6.4, which should itself be subject to the same data protection controls as the production systems it draws from.



VII. COMPARISON TECHNIQUES

7.1 Row Level Exact Comparison

Row level exact comparison, matching every source record to its target counterpart by key and comparing every field after normalization, provides the highest fidelity comparison available and is the preferred technique whenever data volume and system load constraints permit it, consistent with the calibrated comparison depth principle in Section 4.5.

7.2 Aggregate and Control Total Comparison

Where row level comparison is impractical, aggregate comparison, following the control total tradition discussed in Section 2.2, compares summary statistics, such as row counts, sums, and distinct value counts, between source and target. This technique is considerably cheaper computationally but provides weaker assurance, since it cannot detect compensating errors in which two individual discrepancies happen to cancel out in the aggregate figure.

7.3 Comparison Strategy Selection by Volume and Criticality

Table 4 summarizes, at an illustrative level, how comparison strategy might reasonably be selected according to data volume and criticality, consistent with the calibrated comparison depth principle introduced in Section 4.5.

Table.4. Illustrative comparison strategy selection by data volume tier

Data Volume Tier	Recommended Comparison Strategy	Typical Pattern	Tooling	Expected Characteristic	Runtime
Small, under approximately one million rows	Full row level exact comparison	Centralized comparison engine pulling complete extracts		Completes within minutes for most datasets	
Medium, approximately one million to one hundred million rows	Row level comparison with partitioning	Partitioned extraction with parallel comparison workers		Completes within an hour under typical configurations	
Large, over one hundred million rows	Hybrid aggregate and sampled row level comparison	Push down aggregate comparison supplemented by targeted sampling		Completes within several hours depending on sampling depth	
Very large or continuously streaming	Continuous, incremental comparison on changed records only	Change data capture feeding incremental comparison		Runs continuously with near real time discrepancy detection	

7.4 Comparison Timing and Consistency Alignment

Following the consistency model discussion in Section 2.5, this article recommends that comparison timing be chosen deliberately relative to each system's consistency characteristics, for example by introducing a defined settling delay before comparing against an eventually consistent store, or by comparing against a consistent snapshot or point in time view where the source or target engine supports one, rather than comparing against whatever state happens to be visible at the moment the comparison query executes.

7.5 Sampling Strategies for Scale

Where full row level comparison is infeasible even with partitioning, as discussed in Section 7.3, sampling strategies should be designed deliberately rather than applied uniformly at random, since a uniform random sample is likely to miss defects concentrated in a specific subset of the data, such as a particular date range or a particular source system of origin following a merger. Stratified sampling, in which the sample is deliberately drawn proportionally across known subpopulations of interest, generally provides better defect detection per unit of comparison cost than uniform random sampling.

7.6 Schema Mapping Validation

Before any field level comparison can be meaningful, the schema mapping between source and target structures, discussed in Section 2.7, must itself be validated. This article recommends a dedicated validation step that confirms



every source element has an identified target counterpart, or an explicit, documented decision that no counterpart is expected, before the Comparison Layer is permitted to run, since an unmapped element is a schema mapping failure, as described in Section 3.2, not merely a comparison result.

7.7 Mapping Techniques to Failure Modes

Table 5 summarizes how the techniques discussed in this section map to the failure mode families introduced in Section 3.2.

Table.5. Mapping of comparison techniques to failure mode families

Failure Mode Family	Primary Technique	Supporting Technique	Typical Remediation
Type coercion failures	Normalization rule configuration	Deliberate tolerance parameters	Adjust canonical representation rules for the affected type
Null semantics failures	Engine specific null handling rules	Discrepancy classification review	Document and encode the specific engine's null convention
Schema mapping failures	Schema mapping validation	Coverage gap reporting	Correct and revalidate the mapping before rerunning comparison
Timing and consistency failures	Comparison timing alignment	Settling delay or snapshot comparison	Adjust comparison timing relative to replication characteristics
Scale and sampling failures	Stratified sampling	Hybrid aggregate and row level strategy	Expand sample depth for the affected subpopulation
Classification and reporting failures	Rule based discrepancy classification	Reviewer feedback incorporation	Refine classification rules based on reviewer corrections

7.8 A Note on Record Linkage Without a Reliable Key

Where no reliable key links source and target records, for example following a merger in which two independently maintained customer datasets must be compared without a shared identifier, the record linkage and duplicate detection techniques discussed in Section 2.6 become necessary as a precursor to any field level comparison. This article treats this situation as an extension of SACF's Comparison Layer rather than a separate framework, since the output of a record linkage process is, in effect, the key that an ordinary comparison can then use, but acknowledges that building a reliable linkage model is itself a substantial undertaking beyond simple key based matching.

7.9 Continuous, Pipeline Integrated Comparison

Consistent with the pipeline integration discussion in Section 2.8, this article recommends that comparison, once established for a given dataset, be scheduled to run continuously or on every significant data movement event, rather than only once at the conclusion of the migration project that originally motivated building it, so that the comparison gap defined in Section 3.3 does not silently reopen as both source and target systems continue to evolve independently after the original project team has moved on.

7.10 Common Implementation Pitfalls

Practitioner discussion of early attempts to implement cross engine comparison surfaces a recurring set of pitfalls, recorded here so that future implementers can recognize them in advance.

- Writing comparison logic directly against each engine's native query language without an intervening normalization step, which, as discussed throughout this article, tends to produce comparison results dominated by representational noise rather than genuine findings.
- Hard coding tolerance values inline within comparison code rather than exposing them as the explicit, documented configuration parameters described in Section 4.3, which makes it difficult for a reviewer to understand, let alone audit, how strict a given comparison actually was.



- Running a full row level comparison against a very large dataset without first consulting a strategy selection matrix such as the one in Table 4, resulting in comparison jobs that either never complete or that place unacceptable load on production systems.
- Treating a successful comparison run with zero reported discrepancies as proof of correctness, without separately confirming, through the coverage tracking described in Section 6.5, that the comparison was actually configured to examine the fields and records that matter.
- Allowing normalization rules to be adjusted informally and without review whenever a specific discrepancy investigation is underway, which, as discussed in Section 10.2, risks permanently loosening a tolerance in a way that later masks a genuine defect.
- Building comparison logic as a one off script tightly coupled to a single project's specific schema, rather than as a reusable configuration against the general purpose Abstraction and Normalization Layers, which forfeits the compounding return on investment described in Section 4.6 and Section 9.3.
- Neglecting to validate the comparison platform's own components, as discussed in Section 6.9, which can allow a defect in the comparison engine itself to go unnoticed for an extended period, producing a false sense of assurance that is particularly dangerous precisely because it is invisible.

7.11 Comparing Derived and Aggregated Data

Not every target dataset is a direct, row for row copy of its source. Many analytical targets store data that has already been aggregated, filtered, or otherwise transformed by an intermediate process, and comparing such a target directly against the raw source would be meaningless without first replicating the same transformation independently. This article recommends that, wherever a target contains derived data, the comparison framework maintain an explicit, versioned record of the transformation logic the target is expected to reflect, so that the Comparison Layer can apply the same transformation to a freshly extracted source sample before comparing, rather than comparing a transformed target against an untransformed source and misclassifying every resulting difference as a discrepancy. This technique is closely related to, and should be coordinated with, the schema mapping validation described in Section 7.6, since a derived target's transformation logic is, in effect, an extended form of schema mapping that includes computation in addition to structural correspondence.

7.12 Prioritizing Which Datasets to Compare First

Organizations adopting SACF for the first time rarely have the capacity to establish full comparison coverage across every dataset simultaneously, and Table 8 offers an illustrative prioritization matrix, consistent with the calibrated comparison depth principle in Section 4.5, to guide an organization in sequencing its initial rollout.

Table.8. Illustrative prioritization matrix for initial comparison rollout

Use Case Category	Typical Trigger	Recommended Entry Point Layer	Priority Rationale
Planned platform migration	A scheduled cutover date exists	Abstraction and Normalization Layers for the two engines involved	A fixed deadline requires rapid, reliable coverage before cutover
Cloud warehouse adoption with parallel run	Both systems operate concurrently for a transition period	Comparison and Discrepancy Classification Layers	Ongoing parallel operation benefits most from continuous classification
Merger or acquisition consolidation	Two independently maintained systems must unify	Schema mapping validation within the Normalization Layer	Mapping correctness is the dominant risk before any comparison is meaningful
Dual write or replication architecture	Two live copies must remain continuously synchronized	Governance and Reporting Layer with continuous comparison	Sustained operation requires coverage tracking from day one
Regulatory lineage verification	An external reporting obligation references original source data	Discrepancy Classification and audit logging components	Evidentiary sufficiency matters more than comparison speed



Organizations should treat Table 8 as a starting heuristic rather than a rigid prescription, since the appropriate entry point for any specific dataset ultimately depends on where that dataset's greatest actual risk lies, which the adoption pathway described in Section 10.5 is intended to help surface over the course of a phased rollout.

VIII. ILLUSTRATIVE SCENARIOS

8.1 Scenario One: A Silent Floating Point Discrepancy During a Warehouse Migration

Consider a project migrating a financial reporting dataset from a row oriented relational source system to a columnar analytical warehouse. An initial, naive comparison, implemented as a simple equality check between the two systems' numeric columns, reports a large number of discrepancies across nearly every row containing a decimal value, despite the migration's extract, transform, and load logic having been reviewed carefully and believed to be correct.

Investigation, guided by the type coercion failure mode described in Section 3.2, reveals that the source system's fixed precision decimal type and the target warehouse's floating point representation, summarized in Table 3, produce values that differ only in trailing digits far beyond any precision that is meaningful for financial reporting purposes, an artifact of the target engine's storage format rather than any defect in the migration logic itself. The project team configures the Normalization Rule Engine described in Section 6.2 with an explicit, documented tolerance appropriate to the financial reporting context, consistent with the tolerance principle in Section 4.3, and the overwhelming majority of the previously reported discrepancies disappear, leaving a small, now trustworthy residual set that is subsequently confirmed to reflect genuine, previously undetected transformation errors in a small number of specific records.

This scenario illustrates why an undifferentiated equality check, applied without the normalization step SACF requires, produces a result so noisy that the genuine discrepancies it did contain would very plausibly have been overlooked amid the overwhelming volume of false positives, a risk the classification and reporting failure mode in Section 3.2 specifically describes.

8.2 Scenario Two: A Transient Discrepancy Mistaken for a Data Integrity Defect

Consider a second scenario, in which a project team compares a customer profile dataset between a relational system of record and a document oriented store that is updated asynchronously through an event driven replication pipeline. A scheduled comparison run reports a small, but non trivial, number of discrepancies, concentrated among customers whose profiles were updated in the hours immediately preceding the comparison run.

Rather than immediately escalating these discrepancies as data integrity defects, the Discrepancy Classification Layer described in Section 5.4, configured with awareness of the replication pipeline's typical propagation delay, flags this specific pattern, discrepancies concentrated among very recently updated records, as a likely timing and consistency failure rather than a genuine defect, consistent with the discussion in Section 2.4. A subsequent comparison run, performed after an appropriate settling delay as described in Section 7.4, confirms that nearly all of the originally flagged discrepancies have resolved on their own, leaving only a small number that persist and are therefore escalated for genuine investigation.

This scenario illustrates the practical value of the fifth design principle in Section 4.4, treating discrepancy classification as a distinct, deliberate step rather than reporting every detected difference with equal urgency, since the alternative, treating every one of the initially reported discrepancies as an urgent data integrity incident, would have consumed considerable investigative effort for no genuine benefit.

8.3 Scenario Three: A Schema Mapping Gap Discovered Only Through Coverage Reporting

Consider a third scenario, in which a merger driven data consolidation project maps two independently designed customer account schemas onto a single, unified target schema. The project's initial comparison, built hurriedly under time pressure, covers the fields both teams immediately recognized as corresponding to one another, but omits a secondary address field that existed in only one of the two source systems and had no obvious counterpart in the unified target schema.

Because the Comparison Coverage Registry described in Section 6.5 tracks not only which datasets are under comparison but which specific fields within each dataset are covered, the omission is surfaced as an explicit coverage gap during a routine governance review, well before any business stakeholder discovers the gap independently. Investigation reveals that the secondary address field was, in fact, actively used by a small number of customers for



billing purposes, and that the unified target schema's design had genuinely overlooked it, a schema mapping failure rather than a comparison implementation failure.

The project team extends the target schema to accommodate the field, updates the schema mapping validation described in Section 7.6 to include it, and records the episode as a reminder, consistent with the fail loudly principle in Section 4.8, that a comparison framework's reported coverage, not only its reported discrepancies, deserves regular, deliberate review.

8.4 Scenario Four: A Push Down Comparison Reveals a Partition Pruning Side Effect

Consider a fourth scenario, in which an organization operating a large, continuously growing dataset within a batch oriented big data platform adopts a push down comparison approach, as described in Section 6.8, translating comparison logic directly into the platform's own query engine to avoid moving the full dataset elsewhere for comparison. A scheduled comparison job, intended to compare the full dataset against its source system of record on a nightly basis, begins intermittently reporting a small but consistent set of records present in the source system but apparently missing from the target.

Investigation reveals that the missing records are not actually missing at all; they exist in the target dataset but reside in a data partition that the push down query, due to an overly aggressive partition pruning optimization applied automatically by the underlying query engine, was silently excluding from its scan based on an incorrect assumption about the partition key's value range for a small number of late arriving records whose timestamps fell just outside the range the query engine had cached from an earlier, now stale, metadata refresh.

This failure does not fit neatly into any of the failure mode families enumerated in Section 3.2 as originally described, since it is neither a type coercion, null semantics, schema mapping, timing, nor sampling failure in the usual sense, but rather an artifact of the push down deployment topology itself interacting with the target engine's own query optimization behavior. The project team adds this discovery as an explicit addendum to the failure mode taxonomy maintained alongside the Adapter Catalog described in Section 6.1, specifically flagging that any push down adapter for a partitioned big data platform must account for partition metadata staleness, and configures the adapter to force a metadata refresh immediately before every comparison run rather than relying on whatever cached metadata the query engine happens to be using at the time.

This scenario illustrates a broader point consistent with the fail loudly principle in Section 4.8: the specific list of failure modes enumerated in any framework, including the one proposed in this article, should be treated as a living, growing catalog rather than a closed, exhaustive list, since the push down topology's performance advantage, discussed in Section 6.8, comes paired with exactly this kind of engine specific risk that a purely centralized comparison approach would not have encountered at all.

IX. EVALUATION CRITERIA AND OBSERVATIONS

9.1 Discrepancy Detection Latency

- Projects relying on a single, end of project comparison have reported detecting discrepancies introduced after that point, whether through subsequent application changes or ongoing data entry, only when a business stakeholder happens to notice an inconsistency, often weeks or months after the discrepancy was introduced.
- Projects operating continuous, pipeline integrated comparison, as described in Section 7.9, have reported detection latency typically within a single comparison cycle, commonly ranging from near real time for streaming comparison to within a day for scheduled batch comparison.

9.2 False Positive Reduction Through Normalization

- Comparison implementations without a dedicated normalization step have been informally reported to produce a majority of their flagged discrepancies, in some discussions characterized as a large majority, from type coercion and null semantics artifacts rather than genuine data integrity defects.
- Introducing a dedicated Normalization Rule Engine, as described in Section 6.2, has been reported to reduce false positive discrepancy volume substantially, often by what practitioners informally describe as an order of magnitude, allowing the remaining, genuine discrepancies to be investigated efficiently.



9.3 Comparison Coverage Growth Over Time

- Organizations in the early stages of adopting a reusable comparison framework have reported initial coverage concentrated on the specific datasets involved in whichever migration project motivated building the framework in the first place.
- Organizations that treat the framework as a genuinely reusable platform capability, consistent with Section 4.6, have reported coverage expanding steadily across subsequent, unrelated projects, since each new project can reuse existing adapters and normalization rules rather than building comparison logic from scratch.

9.4 A Maturity Perspective

Table 6 offers a simple, illustrative characterization of comparison practice maturity, intended as a self assessment reference rather than a formal scoring instrument.

Table.6. Illustrative comparison practice maturity levels

Maturity Level	Characteristic	Typical Capability	Typical Tooling
Initial	Comparison is manual and project specific	Ad hoc queries written fresh for each project	Spreadsheets and manually written SQL
Developing	Reusable scripts exist but require manual adaptation	Partial adapter coverage for common engines	Shared script library with manual configuration
Established	A dedicated comparison platform runs on a schedule	Automated normalization and classification	A dedicated SACF style comparison platform
Advanced	Comparison runs continuously and feeds governance reporting	Coverage tracked and reported across the organization	Platform integrated into the data delivery pipeline itself

9.5 Qualitative Observations

Several qualitative observations recur across the practices this article draws upon. First, normalization, more than any other single component, determines whether a comparison practice is trusted or ignored, since a practice that produces excessive false positives is quickly abandoned by the teams it is meant to serve, mirroring an observation found throughout the broader assurance literature regarding noise reduction. Second, coverage, not only discrepancy count, deserves regular governance attention, since a comparison framework that reports zero discrepancies because it is not actually comparing a given field provides a dangerously false sense of assurance, as illustrated in Section 8.3. Third, organizations that treat comparison as a reusable platform capability rather than a project specific deliverable report considerably faster adoption on subsequent projects, since the second and third migration projects an organization undertakes can draw on adapters and normalization rules already built and validated during the first.

9.6 Adoption Cost Considerations

A fair evaluation of SACF must also acknowledge that implementing it carries a real, non trivial cost, which this subsection records alongside the benefit side discussed above.

- Building an initial adapter for a new engine family, together with its corresponding normalization rules, has been informally described, in practitioner discussion, as a task on the order of one to a few weeks of focused engineering effort, depending heavily on how well documented the engine's own type system and null handling conventions already are.
- Standing up the Matching and Comparison Engine as reusable infrastructure, rather than bespoke to a single project, typically represents the larger one time investment, often described as spanning a full quarter for a first, representative implementation, but this cost is substantially amortized across every subsequent project, since the engine itself does not need to be rebuilt.
- Introducing a Discrepancy Classification Layer with meaningful rule coverage takes time to mature, since useful classification rules are generally discovered empirically, through investigating early discrepancies, rather than specified correctly from the outset, which argues for budgeting an initial period of elevated manual review before the classification layer's automation reaches its full value.



- The ongoing governance overhead described in Section 10 is typically absorbed as a fractional responsibility within existing data engineering or data quality roles in smaller organizations, becoming a dedicated function only once the number of actively compared datasets grows large enough to justify it.

9.7 A Note on Cross Organizational Comparability

Readers familiar with multiple organizations will recognize that comparison practice maturity, as characterized informally in Table 6, varies considerably depending on how frequently an organization undertakes migration or integration work in the first place. An organization that migrates platforms once a decade has comparatively little incentive to build the kind of reusable platform described in Section 4.6, and a lighter weight, more manual approach may be entirely appropriate for such an organization, consistent with the calibrated comparison depth principle in Section 4.5. A reasonable goal for any organization assessing its own maturity against Table 6 is therefore not uniform advancement toward the most advanced level regardless of circumstance, but a level of investment proportionate to how often, and how consequentially, the organization expects to face the heterogeneous comparison problem this article addresses.

9.8 Risk Adjusted Prioritization of Remediation

Not every confirmed discrepancy warrants the same remediation urgency, and this article recommends that the Governance and Reporting Layer apply a risk adjusted prioritization to confirmed discrepancies, considering both the business significance of the affected data element, consistent with the calibrated comparison depth principle in Section 4.5, and the apparent scope of the discrepancy, meaning whether it affects an isolated record or a systematic pattern likely to recur. A single incorrect value in a rarely used, low significance field warrants a materially different response than a systematic pattern affecting a high significance field across many records, even though both might otherwise appear as a single line item in an undifferentiated discrepancy count. Organizations that fail to apply this kind of prioritization have reported, in informal discussion, that remediation effort tends to drift toward whichever discrepancy is easiest to fix rather than whichever discrepancy matters most, a bias that explicit, documented prioritization criteria can help counteract.

X. GOVERNANCE, DATA QUALITY OWNERSHIP, AND REPORTING

10.1 Roles and Responsibilities

Table 7 summarizes a representative, simplified allocation of responsibility for sustaining a comparison practice over time, offered as an illustration rather than a prescriptive organizational model.

Table.7. Representative allocation of comparison governance responsibilities

Role	Primary Responsibility	Escalation Trigger	Typical Reporting Cadence
Migration or integration project team	Initial adapter and schema mapping setup for a new dataset	Unmapped schema element discovered	At project milestones
Data platform engineering team	Maintenance of the reusable Abstraction and Normalization Layers	Recurring false positive pattern across projects	Ongoing, as issues arise
Data quality or governance function	Review of classified discrepancies and coverage trends	Persistent, unresolved discrepancy exceeding a defined threshold	Periodic, typically monthly
Business data owner	Confirmation of expected versus unexpected discrepancies	Discrepancy with potential business or regulatory impact	As escalated
Internal audit or compliance function	Review of comparison coverage for regulated datasets	Coverage gap identified for a regulated dataset	Periodic, aligned with audit cycle



10.2 Ownership of Normalization Rules

Because normalization rules, described in Section 5.2, encode judgments about acceptable tolerance and engine specific interpretation that carry real consequence, this article recommends that changes to normalization rules be subject to a defined review process, analogous to code review, rather than adjusted unilaterally by whichever engineer happens to be investigating a specific discrepancy at the time, since a tolerance loosened under the pressure of an active investigation can easily mask a future genuine defect if left unreviewed afterward.

10.3 Coverage as a Governance Metric

Consistent with the observation in Section 9.5, this article recommends that comparison coverage, meaning which datasets and which specific fields within them are under active, scheduled comparison, be tracked and reported as a governance metric in its own right, separate from discrepancy counts, since coverage gaps, as illustrated in Section 8.3, represent a form of risk that a discrepancy focused report alone would never surface.

10.4 Regulatory and Audit Considerations

For datasets subject to regulatory reporting obligations, the comparison practice's role extends beyond engineering convenience into compliance significance, and the Comparison Coverage Registry described in Section 6.5, together with the classified discrepancy log described in Section 6.4, should be organized so that an internal or external auditor can efficiently confirm what comparison coverage existed for a given regulated dataset at a given point in time.

10.5 An Organizational Adoption Pathway

Organizations beginning to adopt SACF need not build every layer in full before realizing value. A pathway that recurs informally across practitioner discussion proceeds roughly as follows: begin with a single, well scoped migration or integration project and build the Abstraction and Normalization Layers for the specific engines that project involves; invest early in the Discrepancy Classification Layer even in a simple, rule based form, since the false positive reduction it provides is what determines whether the practice is trusted going forward; establish the Comparison Coverage Registry from the outset, even before coverage is extensive, since the discipline of tracking coverage is harder to retrofit after the fact; and expand the Abstraction Layer's adapter catalog incrementally as subsequent, unrelated projects introduce new engines, treating the growing adapter catalog itself as a visible, cumulative return on the initial investment.

10.6 Metrics Worth Reporting to Leadership

- The proportion of the organization's critical datasets currently under active, scheduled comparison, trending toward completeness for the highest criticality datasets first.
- The volume and trend of classified discrepancies, broken down informally by whether they were ultimately confirmed as genuine defects or resolved as benign variation, to demonstrate the classification layer's value in reducing investigative effort.
- The number of migration or integration projects able to reuse existing adapters and normalization rules rather than building comparison logic from scratch, as a direct measure of the platform's cumulative return on investment.
- A short narrative summary of any notable discrepancy or coverage gap identified and resolved during the reporting period, since a concrete example often communicates the value of the practice more effectively than aggregate statistics alone.

10.7 Vendor and Tool Selection Considerations

Organizations evaluating whether to build a SACF style platform internally or adopt a commercially available data comparison or reconciliation tool should assess any candidate tool against the five layers described in Section 5 explicitly, since many commercially available tools address only a subset of the full problem, most commonly providing strong Abstraction Layer adapter coverage for popular relational engines while offering only limited, difficult to configure normalization and classification capability. A tool that excels at extraction across many engines but provides no meaningful way to express the explicit, documented tolerance parameters described in Section 4.3, or no mechanism for the discrepancy classification described in Section 5.4, will likely still require an organization to build significant complementary capability of its own, and this gap is best identified during tool evaluation rather than discovered only after a licensing commitment has already been made.



XI. LIMITATIONS AND THREATS TO VALIDITY

- The framework is presented at a conceptual and architectural level rather than validated through a controlled empirical study, and the figures offered in Section 9 are explicitly characterized as indicative, order of magnitude observations rather than the result of a rigorous measurement exercise conducted for this article.
- The illustrative scenarios in Section 8 are composite constructions intended for pedagogical clarity and do not describe verified real world incidents.
- The framework assumes an organization undertakes a sufficient volume of migration and integration work over time to justify the reusability investment described in Section 4.6, which may not hold for an organization undertaking only a single, isolated migration project.
- The record linkage extension discussed in Section 7.8 is noted only briefly, and organizations facing a genuine lack of reliable keys between source and target datasets should consult the dedicated record linkage literature referenced in Section 2.6 rather than relying on this article alone.
- The push down deployment topology discussed in Section 6.8 requires engine specific implementation effort that this article does not attempt to specify in detail for any particular engine, since such detail would date quickly as engine capabilities evolve.
- This article does not address comparison of unstructured or semi structured content, such as free text fields or embedded binary data, beyond noting that such content generally requires comparison techniques, such as similarity scoring, that fall outside the structured, field level comparison this article focuses on.
- The risk adjusted prioritization approach discussed in Section 9.8 depends on an organization's ability to assess business significance consistently across many data elements, which itself requires a degree of data governance maturity that this article assumes rather than describes how to build.
- The partition pruning side effect described in Section 8.4 illustrates that the failure mode taxonomy in Section 3.2, however carefully constructed, should not be treated as exhaustive, and organizations adopting SACF should expect to extend the taxonomy with engine and topology specific findings of their own over time.
- The vendor and tool evaluation guidance offered in Section 10.7 is necessarily general, since the commercial tooling landscape for data comparison changes faster than this article can be expected to remain current, and readers should treat the five layer framework as the durable evaluation criterion rather than any specific tool recommendation.

XII. FUTURE WORK

- Empirical study of normalization rule effectiveness across a broader range of engine pairings than the illustrative examples discussed in this article, to replace the indicative figures in Section 9 with measured outcomes.
- Development of a shared, open adapter catalog that multiple organizations could contribute to and draw upon, reducing the duplicative adapter development effort that many organizations currently undertake independently for substantially similar engines.
- Further work on automated discrepancy classification techniques capable of learning from accumulated reviewer feedback more systematically than the rule based approach described in Section 5.4, while preserving the fail loudly principle in Section 4.8 for cases of genuine uncertainty.
- Extension of the framework to address comparison involving unstructured and semi structured content, acknowledged as a limitation in Section 11, potentially drawing on similarity scoring and natural language processing techniques.
- Investigation of how comparison coverage reporting, discussed in Section 10.3, might be standardized sufficiently to support cross organizational benchmarking of comparison practice maturity without requiring disclosure of proprietary data.
- Further study of partition aware and metadata staleness aware adapter design for push down comparison against large scale, partitioned big data platforms, extending the specific finding described in Section 8.4 into a more general set of engineering guidelines.
- Development of a structured method for estimating, in advance, the expected adoption cost described in Section 9.6 for a specific organization's planned migration portfolio, so that investment in a reusable comparison platform can be justified with a concrete, organization specific projection rather than only the indicative figures offered in this article.



XIII. CONCLUSION

Organizations moving data across heterogeneous database engines, whether for a cloud migration, a platform consolidation, or a merger driven integration, face a comparison problem that is considerably harder than it first appears, precisely because the engines involved rarely agree on the representational details that a naive, engine specific comparison implicitly assumes. This article has argued that reliable cross engine comparison requires separating extraction, normalization, comparison, and classification into distinct, deliberately designed layers, and has proposed the Source-Agnostic Comparison Framework as one way of organizing that separation, built around an Abstraction Layer, a Normalization Layer, a Comparison Layer, a Discrepancy Classification Layer, and a Governance and Reporting Layer. The framework draws on established relational, data warehousing, data quality, and distributed systems literature, adapted specifically to the practical difficulty of comparing data across engines that were never designed to agree with one another on representational detail. The structured tables, illustrative scenarios, and indicative figures presented in this article are offered to make the framework concrete and actionable rather than as empirical proof of its effectiveness, and the article has been explicit about this limitation throughout. The four illustrative scenarios, the structured comparison tables, the maturity perspective, and the adoption cost considerations presented in this article are offered to make the framework concrete and actionable rather than as empirical proof of its effectiveness, and the article has been explicit about this limitation throughout. No organization should treat the figures in Section 9 as a precise benchmark; they are offered as a sense of scale, drawn from informal industry discussion, against which an organization can situate its own, necessarily different, circumstances. As organizations continue to adopt an increasingly heterogeneous mix of database engines across their technology estate, from established relational systems to cloud native warehouses, document stores, and distributed big data platforms, the discipline of source agnostic, continuous data comparison is likely to grow correspondingly in importance, and the framework proposed here is offered as a starting point for that continued practice, to be adapted and extended as the underlying engine landscape continues to evolve.

Appendix: Glossary of Key Terms

- Abstraction Layer: the SACF layer responsible for uniform data extraction across heterogeneous engines through engine specific adapters, described in Section 5.1.
- Calibrated comparison depth: the design principle holding that comparison completeness and frequency should scale with data criticality and volume, described in Section 4.5.
- Canonical representation: the common, engine neutral form into which extracted values are converted before any comparison occurs, described in Section 4.2.
- Comparison coverage: the set of datasets and specific fields currently under active, scheduled comparison, tracked by the Comparison Coverage Registry described in Section 6.5.
- Comparison gap: the difference between the confidence a project team believes it has in a dataset's accuracy and the confidence actually warranted given the failure modes described in Section 3.2, defined in Section 3.3.
- Discrepancy Classification Layer: the SACF layer responsible for distinguishing genuine data integrity defects from benign variation, described in Section 5.4.
- Normalization Layer: the SACF layer responsible for removing engine specific representational artifacts before comparison, described in Section 5.2.
- Push down comparison: a deployment topology in which comparison logic is translated into each engine's native query language to minimize data movement, described in Section 6.8.
- Record linkage: the process of determining whether two records refer to the same real world entity, relevant when no reliable key links source and target records, described in Sections 2.6 and 7.8.
- SACF: the Source-Agnostic Comparison Framework, the five layer framework proposed in Section 5 of this article.
- Schema mapping validation: the step of confirming every source element has an identified target counterpart, or an explicit documented decision that none is expected, described in Section 7.6.
- Stratified sampling: a sampling strategy that deliberately draws a sample proportionally across known subpopulations of interest, described in Section 7.5.
- Tolerance: an explicit, documented parameter defining the acceptable difference between two values before they are reported as a discrepancy, described in Section 4.3.
- Tolerance audit log: the durable record of every change made to normalization tolerance parameters, together with who made the change and why, described in Section 6.11.



- Push down comparison coverage gap: a condition in which a query optimization applied by a target engine, such as partition pruning, causes a push down comparison to silently omit records it was intended to examine, illustrated in Section 8.4.

REFERENCES

1. Batini, C., and Scannapieco, M. (2006). *Data Quality: Concepts, Methodologies and Techniques*. Berlin: Springer.
2. Bernstein, P. A., and Haas, L. M. (2008). *Information Integration in the Enterprise*. Communications of the ACM, 51(9), 72 to 79.
3. Brewer, E. (2000). *Towards Robust Distributed Systems*. Keynote address, Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing.
4. Codd, E. F. (1970). *A Relational Model of Data for Large Shared Data Banks*. Communications of the ACM, 13(6), 377 to 387.
5. DAMA International. (2017). *DAMA-DMBOK: Data Management Body of Knowledge*, 2nd edition. Basking Ridge: Technics Publications.
6. Dasu, T., and Johnson, T. (2003). *Exploratory Data Mining and Data Cleaning*. Hoboken: Wiley.
7. Date, C. J. (2003). *An Introduction to Database Systems*, 8th edition. Boston: Addison Wesley.
8. Elmagarmid, A. K., Ipeirotis, P. G., and Verykios, V. S. (2007). *Duplicate Record Detection: A Survey*. IEEE Transactions on Knowledge and Data Engineering, 19(1), 1 to 16.
9. English, L. P. (1999). *Improving Data Warehouse and Business Information Quality*. New York: Wiley.
10. Gilbert, S., and Lynch, N. (2002). *Brewer's Conjecture and the Feasibility of Consistent, Available, Partition Tolerant Web Services*. ACM SIGACT News, 33(2), 51 to 59.
11. Halevy, A., Rajaraman, A., and Ordille, J. (2006). *Data Integration: The Teenage Years*. Proceedings of the 32nd International Conference on Very Large Data Bases.
12. Humble, J., and Farley, D. (2010). *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*. Boston: Addison Wesley.
13. Inmon, W. H. (1992). *Building the Data Warehouse*. New York: Wiley.
14. International Organization for Standardization. (2008). *ISO/IEC 25012:2008, Software Engineering, Software Product Quality Requirements and Evaluation, Data Quality Model*. Geneva: ISO.
15. Kimball, R., and Ross, M. (2013). *The Data Warehouse Toolkit: The Definitive Guide to Dimensional Modeling*, 3rd edition. Hoboken: Wiley.
16. Kleppmann, M. (2017). *Designing Data Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems*. Sebastopol: O'Reilly Media.
17. Linstedt, D., and Olschmke, M. (2015). *Building a Scalable Data Warehouse with Data Vault 2.0*. Burlington: Morgan Kaufmann.
18. Loshin, D. (2001). *Enterprise Knowledge Management: The Data Quality Approach*. San Diego: Academic Press.
19. Loshin, D. (2010). *Master Data Management*. Burlington: Morgan Kaufmann.
20. Newman, S. (2019). *Monolith to Microservices: Evolutionary Patterns to Transform Your Monolith*. Sebastopol: O'Reilly Media.
21. Rahm, E., and Do, H. H. (2000). *Data Cleaning: Problems and Current Approaches*. IEEE Data Engineering Bulletin, 23(4), 3 to 13.
22. Redman, T. (1996). *Data Quality for the Information Age*. Norwood: Artech House.
23. Sadalage, P. J., and Fowler, M. (2012). *NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence*. Boston: Addison Wesley.
24. Stonebraker, M., and Ilyas, I. F. (2018). *Data Integration: The Current Status and the Way Forward*. IEEE Data Engineering Bulletin, 41(2), 3 to 9.
25. Wang, R. Y., and Strong, D. M. (1996). *Beyond Accuracy: What Data Quality Means to Data Consumers*. Journal of Management Information Systems, 12(4), 5 to 33.