SQL vs NoSQL in Polyglot Persistence Architectures

Florin-Răzvan SOARE, Anda-Elena SPĂTARU, Miruna ȘOȘEA
Department of Economic Informatics and Cybernetics
Bucharest University of Economic Studies
Bucharest, ROMANIA
soareflorin21@stud.ase.ro, spataruanda21@stud.ase.ro, soseamiruna21@stud.ase.ro

Polyglot persistence is increasingly adopted because large-scale applications combine correctness-critical state with high-throughput, latency-sensitive workloads that cannot be served efficiently by a single datastore. This paper contrasts SQL and NoSQL systems in terms of transactional guarantees, recovery behavior, data modeling, query expressiveness, schema evolution, and operational scaling constraints. Based on this comparison, we derive datastore selection criteria that distinguish system-of-record components from derived serving models. We then discuss integration mechanisms for polyglot architectures, emphasizing explicit data ownership, change propagation via CDC and log-based replication, and saga-style coordination with compensations to manage cross-store failures.

Keywords: *Polyglot persistence, SQL, NoSQL, scalability, CDC, sagas, ACID, BASE*

1 Introduction

Data-intensive software rarely faces a single, uniform persistence problem. Modern applications must simultaneously support correctness-critical state changes and high-volume user-facing interactions, often under global traffic and continuous delivery constraints. Social networks and large consumer platforms illustrate this tension: the same product must enforce strict rules for identity and access control, apply payments and entitlements reliably, and maintain auditability for moderation decisions, while also serving personalized feeds with low latency, collecting telemetry at high throughput, and storing semi-structured content metadata that evolves as features change. These requirements impose different demands on storage engines. Some workloads require coordination to guarantee invariants, whereas others benefit from partition-local operations to sustain responsiveness.

Relational database management systems remain foundational because the relational model introduced a disciplined way to represent data and express queries while maximizing independence between application logic and machine-level representation [1]. This separation enables long-lived systems to evolve: storage layouts can change, indexes can be added, and physical organization can be optimized without rewriting application code, as long as the logical schema and constraints remain stable. Consequently, relational systems are well suited for authoritative datasets that must be shared across services and remain interpretable over time, such as financial records, account states, and compliance-sensitive entities.

At the same time, large-scale distributed environments treat partial failure as normal. Nodes crash, hardware degrades, latency fluctuates, and network partitions occur. CAP highlights why these conditions matter: in replicated shared-data systems, partitions force explicit trade-offs between strong consistency and availability for affected operations [5]. Brewer's later clarification emphasizes that the engineering task is to define acceptable behavior under partitions rather than rely on a simplistic "two out of three" interpretation [6]. This motivates separating correctness-critical state transitions, which can afford coordination,

from latency-sensitive serving paths that benefit from partition-local designs.

A further motivation for polyglot persistence is workload specialization. When a single general-purpose DBMS is used for all workloads, it may become overburdened by heterogeneous access patterns or force compromises that degrade performance and maintainability. The “one size fits all” critique argues that specialized engines often outperform general-purpose designs when workloads differ substantially [4]. As a result, modern persistence layers are frequently structured as a portfolio: a transactional system of record, a scalable store for events or documents, and derived read models tailored to user-facing queries.

Within this context, polyglot persistence refers to deliberately using multiple datastore technologies within the same application and assigning each technology to the domain component whose requirements best match its guarantees and performance profile [11].

Furthermore, the architectural shift from monolithic software design to distributed microservices has acted as a primary catalyst for the widespread adoption of polyglot persistence. In traditional monolithic architectures, a single, centralized database, typically relational, served as the universal integration layer for all application modules. While this centralized approach simplified operational maintenance, it created a tight coupling that hindered independent scaling, bottlenecked continuous delivery, and forced all workloads into a single data model.

Concurrently, the proliferation of cloud-native infrastructure and Database-as-a-Service (DBaaS) offerings has drastically lowered the operational barrier to managing heterogeneous datastores. Historically, provisioning, tuning, and maintaining high availability for multiple distinct database engines within an on-premises data center required prohibitive operational overhead and highly specialized personnel. Today, managed cloud environments allow teams to provision a highly available relational instance alongside a serverless, horizontally scalable NoSQL table within minutes [25]. This infrastructural democratization directly addresses the demands of the modern data ecosystem, which is characterized by the "3Vs": Volume, Velocity, and Variety. Because traditional, uniformly structured systems struggle to efficiently ingest high-velocity unstructured data streams while simultaneously executing complex relational transactions, a hybridized, polyglot approach to data persistence is no longer a luxury, but an engineering necessity.

This approach can reduce architectural friction, but it introduces an integration problem: once data is split across stores, correctness depends on clear ownership boundaries, propagation semantics, and well-defined failure behavior. To address these concerns systematically, the next sections establish the rationale for SQL and NoSQL choices, compare their trade-offs under realistic workloads, and then formalize integration patterns that preserve coherence across datastore boundaries.

2 Comparison between SQL and NoSQL

SQL and NoSQL systems reflect different assumptions about data, workloads, and failure modes, and the most useful - comparison is therefore workload-driven rather than categorical. SQL systems are rooted in the relational model and a declarative query language intended to preserve stable meaning while decoupling application logic from physical storage choices [1]. Many NoSQL designs target large-scale distributed environments where partial failures and network partitions are routine, and where limiting coordination can improve availability and responsiveness under disruption [5][6].

A first differentiator is how each family represents data and preserves semantics over time. In SQL, an explicit schema (types,

keys, constraints, and relationships) acts as a shared contract that constrains how data can be written and interpreted, which supports long-lived system evolution without silent semantic drift across teams or services [1]. In NoSQL, the data model depends on the family, such as key-value, document, wide-column, or graph databases, and schemas are often flexible or implicit. This flexibility reduces friction when payloads evolve rapidly, but it shifts semantic enforcement toward the application layer (validation, versioning, and monitoring), increasing the risk of divergence if governance is weak [12].

A second differentiator concerns concurrency and correctness, particularly for invariants that span multiple entities. SQL databases commonly provide transactions as a programming model for safe state transitions, supporting atomicity and durability so that multi-step updates do not expose partial effects under concurrency or failures [2]. This is central when business rules cannot be decomposed without risk, or when multiple records must change consistently. Mature recovery designs such as ARIES further reinforce this model by providing disciplined write-ahead logging with redo/undo recovery, improving confidence in crash recovery and interpretability after incidents [3]. NoSQL systems frequently limit coordination to keep operations partition-local; when invariants cross partitions or replicas, applications often rely on idempotent commands, conflict handling, and compensating workflows rather than assuming a single coordinated transaction for all affected data. This direction aligns with the argument that distributed transactions across heterogeneous components can be operationally fragile and costly at scale, motivating alternative composition models such as saga-style sequences of local transactions with compensations [14][15].

A third differentiator is behavior under network partitions and the associated consistency-availability trade-offs in distributed replication. CAP emphasizes that partitions are not hypothetical at scale; they are a reality that forces explicit design choices in replicated shared-data systems [5]. Brewer's later clarification stresses that the practical goal is to define acceptable behavior during partitions, rather than relying on a simplistic "two out of three" rule [6]. Many NoSQL designs embed these choices into replication and coordination strategies: by avoiding global coordination on the common path, they can remain available and responsive during partial failures, often at the cost of exposing temporary inconsistency. The concept of eventual consistency characterizes this inconsistency window and clarifies what the application must handle, including retries, reordering, and convergence, when replicas are not instantly synchronized [7]. SQL systems are more commonly used where strong consistency is expected for authoritative state; in such settings, the system may prefer coordination (and potentially reduced availability for certain operations) to avoid returning conflicting or stale results for correctness-critical decisions.

Performance differences are best framed in terms of which operations are optimized. SQL reads are typically strongest when the workload requires expressive, ad-hoc queries over relationships: joins, filtering across multiple entity types, and aggregations that support auditing, support investigations, and reconciliation. This follows directly from the relational model's objective of enabling high-level queries while allowing the system to choose physical strategies internally [1]. NoSQL reads are typically strongest when the workload is predictable and can be served from a single partition or a pre-composed representation. Document and key-value approaches can make reads fast by

fetching a whole denormalized object in one operation, avoiding join-time costs at the expense of duplication and more complex update propagation. Wide-column systems explicitly shape storage around partitioned access patterns (often key- and range-oriented) to achieve predictable serving behavior at large scale [9][10].

Write performance similarly depends on coordination and workload shape. SQL writes must often pay for transactional semantics, constraint checks, and index maintenance; under contention, concurrency control can add latency, but the benefit is centralized enforcement of invariants and strong recovery guarantees [2][3]. Many NoSQL systems emphasize high write throughput by making writes partition-local and by using replication strategies suited to always-on operation under failure. Dynamo's design exemplifies an availability-focused write/read approach in which replication and conflict resolution are designed to keep the system operating during node failures and network instability [8]. Cassandra emphasizes decentralized, failure-tolerant operation without a single point of failure and is frequently associated with ingestion-heavy patterns where the read model is designed around known access paths [10]. In practical terms, NoSQL tends to outperform when writes are high-volume, mostly independent by key/partition, and the system can tolerate weaker immediate consistency or needs to prioritize availability; SQL tends to outperform when writes must preserve multi-entity invariants and when correctness is more valuable than avoiding coordination. Scalability and tail latency under load are another axis where the design philosophies diverge. "One size fits all" critiques argue that general-purpose DBMS designs can be forced into compromises when serving heterogeneous workloads at extreme scale, motivating specialized systems for distinct access patterns [4]. Many NoSQL stores operationalize specialization by constraining query patterns and scaling horizontally through partitioning and replication, which helps preserve predictable tail latency for serving workloads and ingestion pipelines [8][9][10]. SQL systems can scale substantially, but the combination of cross-entity relational queries and strong invariants can complicate horizontal distribution, especially when constraints or joins span partitions.

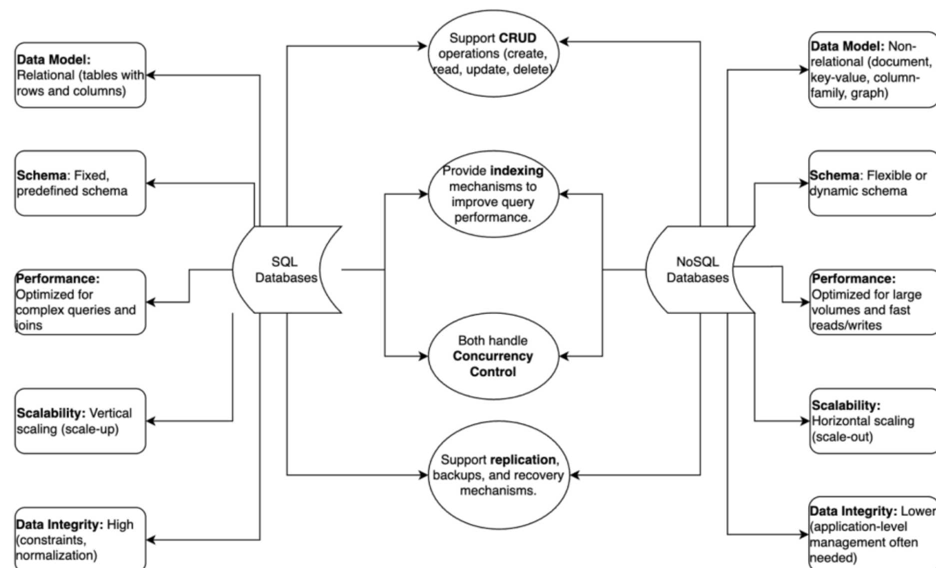

Fig. 3. Comparison between SQL and NoSQL Databases

To provide a clear and structured comparison of the concepts elaborated in the previous paragraphs, Figure 1 visually synthesizes the fundamental differences and shared characteristics between SQL and NoSQL databases. The diagram highlights how SQL systems are modeled in relational data models with fixed schemas, strong data integrity, and optimization for complex queries and joins[1], whereas NoSQL systems emphasize flexible schemas, horizontal scalability, and high-performance read/write operations for large-scale workloads[12]. At the same time, the illustration underscores important commonalities, such as support for CRUD operations, indexing mechanisms, concurrency control, and replication capabilities. By consolidating these aspects into a single conceptual view, the figure facilitates a more intuitive understanding of how the two paradigms diverge in design philosophy while converging on core database functionalities.

These contrasting strengths and trade-offs are not merely theoretical distinctions but have direct architectural implications. In practice, these trade-offs often motivate polyglot persistence, where different datastores are assigned to different bounded contexts according to their required guarantees and access patterns [11][13]. The remaining problem is ensuring coherent behavior once data and responsibilities span multiple stores, which motivates the integration mechanisms discussed next.

3 Advanced Data Modeling and Consistency Trade-offs

To fully appreciate the architectural divergence between relational and non-relational datastores, it is necessary to examine the underlying methodologies governing data modeling and the granular spectrum of consistency models. The transition from SQL to NoSQL is not merely a change in database engines; it dictates a fundamental paradigm shift in how developers interact with data and how distributed systems agree on state.

3.1 The Object-Relational Impedance Mismatch vs. Aggregate-Oriented Storage

Relational database models organize data into two-dimensional tables, requiring normalized relations mapped via foreign keys. However, modern application logic is overwhelmingly object-oriented, dealing with complex, nested, and hierarchical data structures. Bridging this gap requires Object-Relational Mapping (ORM) frameworks (e.g., Hibernate, Entity Framework). While ORMs simplify application development by abstracting SQL queries, they frequently introduce the "Object-Relational Impedance Mismatch." Complex object graphs require multiple resource-intensive joins or result in the notorious N+1 query problem, where the application executes a disproportionate number of secondary queries to fetch related entities.

Conversely, many NoSQL systems, specifically Document and Key-Value stores, adopt an aggregate-oriented storage model [21]. An aggregate is a collection of related objects that are treated as a single unit for data manipulation. By storing the entire aggregate as a single JSON or BSON document, NoSQL databases eliminate the impedance mismatch. An application can retrieve a complex entity (e.g., a student profile with an embedded list of historical test preferences) in a single disk seek. However, this optimization shifts the burden of data integrity to the application layer. If an embedded property changes (e.g., a teacher's contact email embedded within thousands of test records), updating it requires a bulk write operation across multiple documents, whereas a normalized SQL schema would require a single row update in a Teachers table.

3.2 Entity-Driven vs. Access-Driven Schema Design

The philosophical difference between SQL and NoSQL is most apparent during the schema design phase. Relational databases employ an *entity-driven* modeling approach.

The database is designed to reflect the pure relational state of the business domain, aiming for the Third Normal Form (3NF) to eliminate data redundancy and insertion anomalies. The exact queries that will be executed are treated as an afterthought, with the assumption that SQL's expressive power, coupled with secondary indexes, can efficiently join tables to answer any future ad-hoc analytical question.

NoSQL data modeling, particularly in wide-column and partitioned key-value stores like DynamoDB or Cassandra, mandates an *access-driven* (or query-driven) approach [22]. Because distributed databases lack native support for cross-partition joins, the database schema must be meticulously designed around the specific queries the application will execute. This gives rise to paradigms such as "Single-Table Design," where heterogeneous entity types (e.g., Students, Courses, and Enrollments) are co-located within the exact same table using generic partition keys and sort keys. By pre-computing joins and utilizing materialized paths or adjacency lists, NoSQL systems achieve predictable O(1) or O(log N) read complexity regardless of the dataset's size. However, this creates a highly rigid structure; any new access pattern not anticipated during the initial design phase may require a complete data migration or the creation of costly secondary indexes.

3.3 Granular Consistency Models and Quorum Mechanics

While the CAP theorem and the concept of "eventual consistency" provide a high-level vocabulary for distributed systems, practical polyglot architectures rely on granular, tunable consistency models based on quorum mechanics. The binary distinction between "strong" and "eventual" consistency is an oversimplification.

In distributed databases, consistency is often dictated by the replication factor (N), the write quorum (W), and the read quorum (R). If the system is configured such that R+W>N, it can guarantee strict consistency, as any read operation will overlap with the most recent write operation [8]. However, setting high read and write quorums significantly increases latency and reduces availability during node failures.

To optimize performance, NoSQL systems allow architects to lower the read quorum, intentionally returning data from a single, potentially out-of-date replica. Consequently, applications must be engineered to handle intermediate consistency states [23]. These include *Read-After-Write Consistency* (ensuring a user can see their own updates immediately, often achieved by routing the read to the leader node), *Monotonic Reads* (ensuring a user never reads an older version of data after having read a newer version), and *Consistent Prefix Reads* (ensuring operations are seen in the order they were applied). For example, DynamoDB offers developers a boolean parameter to request a strongly consistent read. Doing so bypasses the storage cache and consumes twice the read capacity units, illustrating the direct financial and performance cost of demanding relational-style strong consistency in a distributed environment.

3.4 Multi-Region Active-Active Topologies and Conflict Resolution

As cloud-native applications scale globally, polyglot persistence architectures frequently extend across multiple geographic regions to ensure low-latency access and robust disaster recovery. SQL databases typically employ asynchronous primary-replica architectures for cross-region replication. In this topology, the primary region remains the single source of truth; if it fails, writes are halted until a replica is promoted. This guarantees serializable transaction logs but creates a regional bottleneck for write throughput.

In contrast, distributed NoSQL systems often support Active-Active (multi-master)

topologies, such as DynamoDB Global Tables or Cassandra's multi-datacenter replication. These systems allow concurrent writes to the same logical record in different regions, maximizing availability but making write conflicts inevitable. Resolving these conflicts requires algorithmic interventions that do not rely on global locking. Common strategies include "Last Writer Wins" (LWW) based on physical or logical timestamps, or application-defined merge logic utilizing Conflict-free Replicated Data Types (CRDTs) [24]. CRDTs guarantee that regardless of the order in which network packets arrive, all database nodes will eventually converge to the exact same state without coordination. Understanding these conflict resolution heuristics is paramount when orchestrating saga compensations across diverse datastores, as delayed cross-region replication can inadvertently cause compensating transactions to execute against stale data, leading to severe logical corruption.

4 Experimental Setup and Case Study (PostgreSQL vs DynamoDB)

In practice, complex applications often require rich relationships between entities, which are naturally supported by relational databases through joins, constraints, and structured schemas. At the same time, some application features are largely independent of cross-entity relationships and can be better served by NoSQL systems optimized for high-throughput, low-latency access patterns. To illustrate why combining relational and non-relational datastores can be beneficial in polyglot persistence architectures, we designed a small empirical evaluation based on an e-learning application scenario.

In the reference e-learning application, the primary operational dataset is naturally relational: students, classes, tests, problem banks, and teacher-owned assessments form a strongly connected domain that benefits from a uniform schema, referential integrity, and expressive join-based queries. For these correctness-critical and relationship-heavy entities, a SQL database is the appropriate system of record.

In contrast, storing and serving a student's solved-test history is largely an append-only, read-by-student workload that does not require complex joins or cross-entity constraints at query time. This makes it a good candidate for a non-relational, partition-oriented store. Therefore, our evaluation focuses on this "history" component and compares write performance under concurrent load when implemented on PostgreSQL (Amazon RDS) versus DynamoDB, highlighting how polyglot persistence assigns each datastore to the workload it matches best.

We created a DynamoDB table using the default baseline settings available under the AWS Free Tier, without workload-specific tuning, and added a Global Secondary Index on the idStudent attribute.

idSolvedProblem *(String)*	answers	checkedAnswers	correctAnswers	idProblem	idStudent	mainQuestion	mark	problemType	questions	solvingDate
60000000899		[{"BOOL": true}...	[{"BOOL": false ...	1911654	1879223	Main: Bo9h5pu...	22	GRILA	[{"S": "Q: ...	2025-12-30T11:53:33.275587+00:00
16000003557		[{"BOOL": false ...	[{"BOOL": false ...	1893290	371266	Main: sg94BWc...	100	GRILA	[{"S": "Q: ...	2025-12-22T04:25:24.618345+00:00
23000000969	[{"S": "A: p...			619926	1017489	Main: W01ILvF...	89	NONGRILA	[{"S": "Q: ...	2025-12-12T22:51:26.685552+00:00

Fig. 4. DynamoDB table structure

Figure 2 illustrates a snapshot of the resulting table within the AWS infrastructure, highlighting the semi-structured nature of the stored items. Each record encapsulates solved-test data, including identifiers, answer collections, correctness indicators, and metadata such as timestamps and scores, all organized in a flexible, attribute-based format. This representation reflects DynamoDB's

schema-less design, allowing heterogeneous attributes to coexist within the same table while supporting high-throughput insert operations and efficient retrieval via indexed access patterns.

Global secondary indexes (1) Info

Delete Create index

Find indexes

Name	Status	Partition key	Sort key	Read capacity	Write capacity	Projected attributes	Size	Item count
idx_solved_problems_student	Active	idStudent (Number)	-	On-demand	On-demand	All	0 bytes	0

Fig. 5. DynamoDB index on idStudent

To illustrate the performance of joins and queries, we created a partitioning index. Figure 3 presents the configuration of the Global Secondary Index defined on the DynamoDB table. The index, named *idx_solved_problems_student*, uses *idStudent* as its partition key and operates in on-demand capacity mode, aligning with the baseline, auto-scaling configuration of the table. By projecting all attributes, the index enables efficient query execution without requiring additional lookups to the base table. This design supports the primary access pattern of retrieving solved-test history per student, illustrating how DynamoDB leverages secondary indexes to optimize read operations in scenarios where relational joins are not applicable[8].

The exploratory analysis on different architectural paradigms, includes an analysis on performance of a SQL database as a comparison to the previous example of NoSQL. Therefore, for PostgreSQL, we adopted a relational design that remains normalized while supporting heterogeneous problem formats (multiple-choice vs. free-text). Specifically, we created two tables:

1) `solved_problems`, which stores the shared "header" fields for each solved problem (e.g., `id_student`, `id_problem`, `mark`, `solving_date`, `problem_type`);

2) `solved_problems_data`, a volume table that stores variable-length components as rows (questions, selected answers, correct answers, or free-text answers).

This design allows each solved problem to have a different number of questions/answers without relying on JSON columns, while keeping the core metadata relational and queryable. To optimize the student-history use case, we created an index on `solved_problems.id_student` and an index on `solved_problems_data.id_solved_problem` (foreign key) to accelerate retrieval and joins. Figure 4 illustrates the table definition and indexes.

```sql
DO $$
BEGIN
  IF NOT EXISTS (SELECT 1 FROM pg_type WHERE typname = 'problem_type') THEN
    CREATE TYPE problem_type AS ENUM ('GRILA', 'NONGRILA');
  END IF;

  IF NOT EXISTS (SELECT 1 FROM pg_type WHERE typname = 'solved_data_type') THEN
    CREATE TYPE solved_data_type AS ENUM (
      'QUESTION',
      'CHECKED_ANSWER',
      'CORRECT_ANSWER',
      'ANSWER'
    );
  END IF;
END $$;

CREATE TABLE IF NOT EXISTS solved_problems (
  id_solved_problem BIGINT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,

  main_question TEXT NOT NULL,
  id_student    BIGINT NOT NULL,
  id_problem    BIGINT NOT NULL,

  mark          SMALLINT NOT NULL CHECK (mark BETWEEN 10 AND 100),
  solving_date  TIMESTAMPTZ NOT NULL,

  problem_type  problem_type NOT NULL
);

CREATE TABLE IF NOT EXISTS solved_problems_data (
  id_solved_problem_data BIGINT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,

  id_solved_problem BIGINT NOT NULL
    REFERENCES solved_problems(id_solved_problem)
    ON DELETE CASCADE,

  data_type solved_data_type NOT NULL,

  pos INT NOT NULL CHECK (pos >= 1),

  text_value TEXT,
  bool_value BOOLEAN,

  CONSTRAINT ck_value_by_type CHECK (
    (data_type IN ('QUESTION','ANSWER') AND text_value IS NOT NULL AND bool_value IS NULL)
    OR
    (data_type IN ('CHECKED_ANSWER','CORRECT_ANSWER') AND bool_value IS NOT NULL AND text_value IS NULL)
  )
);

CREATE INDEX idx_solved_problems_student
ON solved_problems (id_student);

CREATE INDEX idx_solved_problems_data_parent
ON solved_problems_data (id_solved_problem);
```

Fig. 4. Postgres tables and indexes code

As a comparison with the previously described DynamoDB configuration, figure 5 presents the configuration of the PostgreSQL instance deployed on AWS RDS. Similar to the DynamoDB setup, the database was provisioned using a baseline configuration within the AWS Free Tier, without workload-specific tuning. The instance is based on a *db.t4g.micro* class with limited compute and memory resources, single-AZ deployment, and general-purpose SSD storage with autoscaling enabled. Additional features such as encryption at rest, automated backups, and basic monitoring (Performance Insights and Enhanced Monitoring) are also configured by default. This setup reflects a typical lightweight relational deployment, providing a controlled environment for evaluating performance and behavior relative to the DynamoDB implementation under comparable resource constraints.

Fig. 5. Postgres database configuration

Building on the previously described infrastructure configurations, we designed a controlled experiment to evaluate and compare the insert performance of the two datastores. Specifically, we implemented a concurrent insert benchmark in Python, where multiple worker threads insert synthetic solved-problem records in batches to simulate a write-intensive workload. The benchmark measures total wall-clock time and derives end-to-end throughput (inserts per second) under load. To ensure that the results reflect database performance rather than data preparation overhead, the reported “insert-only” time excludes record generation, focusing exclusively on the duration of write operations. As illustrated in the implementation, timing is initiated immediately before the database write call and stopped after its completion, while synthetic data generation and mapping are performed outside the measured interval.

```
def main():  1 usage
    args = parse_args()
    target = args.target

    total_ops = WORKERS * PER_WORKER

    pool = None
    seq_name = None
    ddb_table = None

    if target == "sql":
        pool = ConnectionPool(conninfo=PG_DSN, min_size=1, max_size=PG_POOL_MAX, timeout=30)
        with pool.connection() as conn:
            seq_name = get_sequence_name(conn)

    if target == "nosql":
        session_kwargs = dict(
            aws_access_key_id=AWS_ACCESS_KEY_ID,
            aws_secret_access_key=AWS_SECRET_ACCESS_KEY,
            region_name=AWS_REGION,
        )
        if AWS_SESSION_TOKEN:
            session_kwargs["aws_session_token"] = AWS_SESSION_TOKEN

        session = boto3.session.Session(**session_kwargs)
        ddb = session.resource( service_name: "dynamodb", config=DDB_BOTO_CONFIG)
        ddb_table = ddb.Table(DDB_TABLE)

    wall0 = time.perf_counter()

    insert_times = []
    with ThreadPoolExecutor(max_workers=WORKERS) as ex:
        if target == "sql":
            futures = [ex.submit(worker_sql,  *args: i, pool, seq_name) for i in range(WORKERS)]
        else:
            futures = [ex.submit(worker_nosql,  *args: i, ddb_table) for i in range(WORKERS)]

        for f in as_completed(futures):
            wid, ins = f.result()
            insert_times.append(ins)

    wall1 = time.perf_counter()
    wall = wall1 - wall0
```

Fig. 6. Main function of benchmark script

For reproducibility, the benchmark was executed separately for each datastore using the parameters *--target sql* and *--target nosql*. Both tests were run from the same client machine within the same AWS region (*eu-central-1*) to minimize network-induced variability. Each run used WORKERS = [114], PER_WORKER = [8000] (total inserts = WORKERS × PER_WORKER), and incremental batching with CHUNK = [500]. It is important to note that, in the PostgreSQL case, effective concurrency was constrained by the connection pool limit (PG_POOL_MAX = 15), whereas DynamoDB, due to its managed and distributed architecture, can accommodate a significantly higher level of concurrent client requests. For a better overview, of the script that performs the inserts, figure 6 depicts the main function of the benchmark script.

```
def worker_sql(worker_id: int, pool: ConnectionPool, seq_name: str):  1 usage
    rng = random.Random(worker_id * 99991 + int(time.time()))
    insert_sec = 0.0

    done = 0
    while done < PER_WORKER:
        n = min(CHUNK, PER_WORKER - done)

        docs = [gen_submission(rng) for _ in range(n)]
        templates = [precompute_sql_templates(d) for d in docs]

        idx = 0
        while idx < len(templates):
            b = templates[idx: idx + SQL_BATCH_SIZE]
            t0 = time.perf_counter()
            insert_batch_postgres(pool, seq_name, b)
            t1 = time.perf_counter()
            insert_sec += (t1 - t0)
            idx += SQL_BATCH_SIZE

        done += n

    return worker_id, insert_sec


def worker_nosql(worker_id: int, ddb_table):  1 usage
    rng = random.Random(worker_id * 99991 + int(time.time()))
    insert_sec = 0.0

    base = (worker_id + 1) * 1_000_000_000

    done = 0
    while done < PER_WORKER:
        n = min(CHUNK, PER_WORKER - done)

        docs = [gen_submission(rng) for _ in range(n)]
        items = [make_ddb_item(str(base + done + i), d) for i, d in enumerate(docs)]

        t0 = time.perf_counter()
        insert_batch_dynamodb(ddb_table, items)
        t1 = time.perf_counter()
        insert_sec += (t1 - t0)

        done += n

    return worker_id, insert_sec
```

Fig. 7. Functions to calculate time for inserts

Building on the previously described benchmark setup, we will analyse both the implementation details and the resulting performance metrics obtained for each datastore. Figure 7 illustrates the core worker functions used in the experiment, highlighting how insert time is accumulated strictly around the database write operations for both PostgreSQL and DynamoDB. In particular, each worker processes records in batches (CHUNK = 500), measuring only the execution time of the insert calls (*insert_batch_postgres* and *insert_batch_dynamodb*), thereby ensuring consistency with the "insert-only" metric defined earlier. Figures 8 and 9 report the aggregated outputs for the NoSQL and SQL runs, respectively. Under the same client-side workload configuration (114 worker threads × 8,000 inserts each, CHUNK = 500), DynamoDB completed the run in **190.61 s** total wall-clock time and achieved **4,784.61 inserts/s**, with an average per-worker **insert-only** time of **177.07 s**. PostgreSQL (Amazon RDS) required **629.88 s** total wall-clock time, corresponding to **1,447.89 inserts/s**, with an average per-worker **insert-only** time of **624.97 s**. Overall, DynamoDB achieved approximately **3.3× higher throughput** than PostgreSQL in this append-only write scenario (4,784.61 vs. 1,447.89 inserts/s). This outcome is consistent with DynamoDB's partition-oriented design for high write concurrency, whereas PostgreSQL incurs additional overhead from transactional processing and index maintenance and, in our setup, was further bounded by the connection pool (**PG_POOL_MAX = 15**), effectively capping the level of database-side concurrency despite the higher number of client threads.

```
(.venv) razvan@Razvans-MacBook-Pro DatabaseReadWriteComparison % python script.py --target nosql


TOTAL wall: 190.61s
Throughput end-to-end (submissions/sec): 4784.61
Avg worker insert_only: 177.07s
(.venv) razvan@Razvans-MacBook-Pro DatabaseReadWriteComparison %
```

Fig. 8. Script output for NoSQL target parameter

```
(.venv) razvan@Razvans-MacBook-Pro DatabaseReadWriteComparison % python script.py --target sql


TOTAL wall: 629.88s
Throughput end-to-end (submissions/sec): 1447.89
Avg worker insert_only: 624.97s
(.venv) razvan@Razvans-MacBook-Pro DatabaseReadWriteComparison %
```

Fig. 9. Script output for SQL target parameter

The experimental results highlight the strong dependency between database performance and workload characteristics. In the evaluated append-only, write-intensive scenario, DynamoDB significantly outperformed PostgreSQL, achieving approximately 3.3× higher throughput. This outcome is consistent with the design principles of distributed NoSQL systems, which prioritize horizontal scalability, partitioned data distribution, and high write concurrency, as originally demonstrated in

systems such as Dynamo [8] and Bigtable [9]. By avoiding complex transactional coordination and leveraging eventual consistency models [7], DynamoDB can efficiently absorb a large volume of concurrent write requests.

In contrast, PostgreSQL's performance reflects the disadvantages of relational systems designed around strong consistency and transactional guarantees. Mechanisms such as write-ahead logging, locking, and index maintenance, fundamental to ensuring ACID properties [2][3], introduce additional latency during write operations. Furthermore, in the experimental setup, effective concurrency was limited by the connection pool, highlighting how resource management can become a bottleneck in vertically scaled systems. The results of the experimental benchmark reinforce the argument that "one size does not fit all" in data management [4]. While NoSQL systems excel in high-throughput, relationship-light workloads, relational databases remain better suited for scenarios requiring complex queries, strict consistency, and rich integrity constraints. Therefore, the observed performance gap should not be interpreted as a universal advantage of NoSQL, but rather as evidence of the importance of aligning data storage technologies with specific workload requirements.

5 Workload-Driven Data Management Implications

The experimental findings reinforce a fundamental principle in modern data management: datastore selection must be driven by workload characteristics rather than adherence to a single technological paradigm. As argued in prior work, the "one size fits all" approach is no longer viable in the presence of diverse application requirements and data access patterns [4]. Instead, systems must be designed by carefully aligning storage technologies with the operational profiles they are expected to support.

The evaluated scenario highlights a clear distinction between workload types. Append-only, write-intensive workloads with simple access patterns, such as the solved-test history analyzed in this study, benefit from NoSQL systems optimized for horizontal scalability and high ingestion throughput. These systems typically relax strict consistency guarantees in favor of availability and partition tolerance, consistent with the trade-offs described by the CAP theorem [5][6] and the eventual consistency model [7]. In contrast, workloads involving complex relationships, integrity constraints, and transactional semantics remain better suited to relational databases, where ACID properties and structured schemas ensure correctness and consistency [2].

This contradictory division suggests that data modeling should not be approached uniformly across an application. Instead, different components of the same system may exhibit fundamentally different requirements: some demand strong consistency and relational integrity, while others prioritize scalability, latency, and flexibility. As a result, architectural decisions should begin with a precise characterization of workload dimensions, including read/write ratios, concurrency levels, data interdependencies, and query complexity.

More important, these implications extend beyond performance considerations. Choosing an inappropriate datastore for a given workload can lead to unnecessary complexity, either by forcing relational systems to scale beyond their intended design or by requiring NoSQL systems to emulate relational behavior at the application level. Therefore, effective data management requires not only selecting the right tool for each task but also acknowledging the boundaries of each paradigm.

These observations naturally motivate the adoption of polyglot persistence, where multiple datastores coexist within a system, each serving a specific role aligned with its strengths. The following section explores how such heterogeneous systems can be integrated while preserving consistency and operational reliability.

Beyond performance and scalability limits, polyglot persistence fundamentally alters the financial and operational model of data management. The experimental setup highlights two distinct provisioning paradigms: capacity-based provisioning (PostgreSQL on RDS) and consumption-based, or serverless, scaling (DynamoDB).

In traditional relational deployments, resources (CPU, RAM, storage IOPS) must be provisioned to handle peak expected loads. This often leads to underutilization during off-peak hours, creating financial inefficiency. Furthermore, vertically scaling a relational database to alleviate connection pool bottlenecks, as observed in our PostgreSQL benchmar, incurs a steep cost curve and potential downtime during instance resizing.

In contrast, managed NoSQL systems like DynamoDB offer on-demand capacity models that align costs directly with application traffic. You pay per read/write request unit, which is highly cost-effective for spiky, high-throughput ingestion workloads such as telemetry or our test-history scenario. However, this financial model introduces new risks: poorly designed partition keys can lead to "hot partitions," causing massive read/write throttling and sudden cost spikes. Therefore, workload-driven data management must also encompass a "FinOps" (Financial Operations) perspective [18]. A polyglot architecture is often economically justified when the cost of offloading high-throughput, simple-query traffic to a serverless NoSQL datastore is lower than the cost of over-provisioning a monolithic SQL cluster to handle the same traffic alongside complex transactions.

6 Integrating SQL and NoSQL in Polyglot Persistence Architectures

Once multiple datastores are introduced, correctness depends on making ownership and propagation semantics explicit. A useful baseline is to designate a system of record for each business fact: exactly one datastore is authoritative for a given domain concept, while other stores contain derived projections, indexes, or caches built from that source [11][13]. This avoids “dual truth” scenarios in which multiple systems accept independent updates for the same fact, creating inconsistencies that are difficult to detect and reconcile. The integration question then becomes how committed changes from the system of record are propagated and applied to downstream models under failures and retries.

In the context of the evaluated e-learning system, this principle can be concretely applied by treating the relational database (PostgreSQL) as the system of record for core entities such as students, tests, and results, where strong consistency and integrity constraints are required. In contrast, DynamoDB acts as a derived store that maintains a denormalized, append-only history of solved tests per student, optimized for fast retrieval. Under this model, updates are never performed independently in both systems; instead, changes originate in the system of record and are subsequently propagated to downstream projections. The integration challenge then becomes ensuring that committed changes are reliably transferred and applied across systems, even in the presence of failures and retries.

Change propagation is a second pillar of integration. Derived stores must be kept sufficiently up to date for their intended user journeys, which requires reliable synchronization mechanisms. Log-based replication and change data capture are commonly used

to propagate committed changes incrementally, enabling downstream stores to update projections with low latency [16]. For example, when a student completes a test, the result is first persisted in PostgreSQL, after which a corresponding change event (e.g., *student_solved_test_created*) can be emitted and consumed to update the DynamoDB projection. However, CDC introduces engineering challenges: extracting and interpreting log records at scale is complex, and downstream consumers must address ordering, replay, and idempotent application to avoid duplicate effects [16]. Robust implementations therefore rely on consumer offsets, replay capability, dead-letter handling, and observability metrics such as lag and error rates.

To formalize the unidirectional flow of data from the system of record to derived stores, modern polyglot architectures frequently adopt the Command Query Responsibility Segregation (CQRS) pattern paired with Event Sourcing [19]. CQRS structurally separates the write models (Commands), handled by a strongly consistent SQL database to validate business rules, from the read models (Queries), which are materialized in NoSQL databases for high-speed delivery. Instead of just persisting the current state, Event Sourcing involves storing a sequence of immutable state-changing events. In our e-learning context, instead of merely capturing a row update, the system records an explicit domain event (e.g., TestGraded, MarkAssigned). This immutable event log acts as the ultimate source of truth, from which the DynamoDB projection can be deterministically rebuilt if destroyed or structurally altered.

Furthermore, integrating heterogeneous databases amplifies data governance and compliance complexities, particularly concerning data privacy frameworks like the GDPR. In a centralized relational database, executing a "Right to be Forgotten" request involves executing a single DELETE statement with cascading foreign keys. In a polyglot architecture driven by CDC and eventual consistency, a single deletion must propagate reliably through message brokers, updating not only the SQL system of record but also purging the corresponding denormalized records in the NoSQL serving layers and any downstream data lakes [20]. This necessitates automated reconciliation jobs that periodically sweep derived stores to detect and remediate orphaned data, preventing silent compliance violations.

Cross-store operations introduce a third challenge. While distributed transactions appear attractive, many large-scale systems avoid them due to coordination overhead and operational fragility, relying instead on message-driven workflows and compensating actions [14]. In this context, sagas provide a formal model: long-lived business activities are implemented as a sequence of local transactions paired with compensations that restore invariants when later steps fail [15]. In practice, this can be illustrated by a workflow where a solved test is first recorded in the SQL system, then propagated to the NoSQL store, and finally used to trigger user-facing updates. If one of the later steps fails, compensating actions, such as retrying the projection update or marking the operation for reconciliation, ensure that the system eventually converges to a consistent state. This approach requires explicit state management, idempotent command handling, and clearly defined recovery strategies.

Finally, broader work on polyglot data management highlights that heterogeneity increases integration complexity and makes semantic alignment a first-order concern [17]. Even when integration is implemented at the application layer, similar issues recur: mapping data models, reconciling inconsistencies, and managing operational complexity across systems with different guarantees. Resilient polyglot architectures therefore combine datastore specialization with explicit contracts for ownership, consistency

expectations, propagation guarantees, and recovery strategies. While this added complexity introduces engineering overhead, it enables systems to leverage the strengths of both relational and NoSQL paradigms, achieving a balance between consistency, scalability, and performance that would be difficult to obtain with a single datastore.

7 Conclusion

This paper demonstrates that database selection in a polyglot architecture should be fundamentally driven by workload characteristics rather than adherence to a single paradigm. Starting from the well-established distinction between relational and non-relational systems, the study shows that their differences are not merely theoretical but have direct implications for system design and performance. In the evaluated e-learning scenario, the core domain is most effectively modeled using SQL, where strong relational structures and integrity constraints are essential for maintaining consistency and correctness. In contrast, the solved-test history represents an append-only, student-centric access pattern that is inherently read-optimized and does not require complex joins at query time.

Empirical evaluation under identical concurrent insert workloads on AWS indicates that DynamoDB provides superior ingestion throughput compared to PostgreSQL (RDS) for this specific component. This result reflects the architectural differences between the two systems: DynamoDB's distributed, partition-oriented design enables efficient handling of high write concurrency, while PostgreSQL incurs additional overhead due to transactional processing, index maintenance, and connection management. However, these findings should be interpreted in the context of the evaluated workload, as relational systems remain more suitable for scenarios requiring complex queries, strong consistency, and rich data relationships.

Overall, the results reinforce the argument that modern applications benefit from combining multiple data storage paradigms within a unified architecture. Polyglot persistence allows each datastore to be used according to its strengths, enabling systems to achieve both scalability and correctness without compromising on either dimension. At the same time, this approach introduces additional complexity in terms of data integration, consistency management, and operational overhead, requiring careful design of ownership boundaries and data propagation mechanisms.

Furthermore, this research underlines that architectural decisions in data management extend beyond technical metrics to include operational predictability, cost efficiency, and data governance. While integrating systems via CDC, CQRS, and sagas mitigates the risks of distributed data, it shifts complexity from the database engine to the application and infrastructure layers. Teams must be prepared to handle eventual consistency, cross-store auditing, and complex compliance enforcement (such as distributed data deletion).

Future work may extend this study by evaluating mixed workloads, read performance, and consistency trade-offs under different deployment configurations, further refining the guidelines for workload-driven data management.

References

[1] E. F. Codd, "A Relational Model of Data for Large Shared Data Banks," *Communications of the ACM*, https://www.seas.upenn.edu/~zives/03f/cis550/codd.pdf, 1970.

[2] J. Gray, "The Transaction Concept: Virtues and Limitations," Tandem TR 81.3, https://jimgray.azurewebsites.net/papers/thetransactionconcept.pdf, 1981.

[3] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz, "ARIES: A Transaction Recovery Method Supporting

Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging," *ACM Transactions on Database Systems*, https://web.stanford.edu/class/cs345d-01/rl/aries.pdf, 1992.
[4] M. Stonebraker et al., "'One Size Fits All': An Idea Whose Time Has Come and Gone," *ICDE*, https://cs.brown.edu/~ugur/fits_all.pdf, 2005.
[5] E. A. Brewer, "Towards Robust Distributed Systems," PODC Keynote, https://pld.cs.luc.edu/courses/353/spr11/notes/brewer_keynote.pdf, 2000.
[6] E. A. Brewer, "CAP Twelve Years Later: How the 'Rules' Have Changed," *Computer*, https://sites.cs.ucsb.edu/~rich/class/cs293b-cloud/papers/brewer-cap.pdf, 2012.
[7] W. Vogels, "Eventually Consistent," *Communications of the ACM*, https://15799.courses.cs.cmu.edu/fall2013/static/papers/p40-vogels.pdf, 2009.
[8] G. DeCandia et al., "Dynamo: Amazon's Highly Available Key-Value Store," *SOSP*, https://www.allthingsdistributed.com/files/amazon-dynamo-sosp2007.pdf, 2007.
[9] F. Chang et al., "Bigtable: A Distributed Storage System for Structured Data," *OSDI*, https://static.googleusercontent.com/media/research.google.com/en//archive/bigtable-osdi06.pdf, 2006.
[10] A. Lakshman and P. Malik, "Cassandra: A Decentralized Structured Storage System," *SIGOPS Operating Systems Review* / LADIS, https://www.cs.cornell.edu/projects/ladis2009/papers/lakshman-ladis2009.pdf, 2010.
[11] P. J. Sadalage and M. Fowler, "*NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence,*" Addison-Wesley, https://ptgmedia.pearsoncmg.com/images/9780321826626/samplepages/0321826620.pdf, 2012.
[12] A. B. M. Moniruzzaman and S. A. Hossain, "NoSQL Database: New Era of Databases for Big Data Analytics - Classification, Characteristics and Comparison," https://arxiv.org/pdf/1307.0191, 2013.
[13] P. P. Khine and Z. Wang, "A Review of Polyglot Persistence in the Big Data World", *Information*, https://www.mdpi.com/2078-2489/10/4/141, 2019.
[14] P. Helland, "Life Beyond Distributed Transactions: An Apostate's Opinion," *CIDR*, https://ics.uci.edu/~cs223/papers/cidr07p15.pdf, 2007.
[15] H. Garcia-Molina and K. Salem, "SAGAS," *ACM SIGMOD*, https://www.cs.cornell.edu/andru/cs711/2002fa/reading/sagas.pdf, 1987.
[16] D. Butterstein et al., "A Fast, Scalable Replication Solution for Near Real-Time (CDC/log-based replication)," *PVLDB*, https://www.vldb.org/pvldb/vol13/p3245-butterstein.pdf, 2020.
[17] F. Kiehn et al., "Polyglot Data Management: State of the Art & Open Challenges," *PVLDB*, https://www.vldb.org/pvldb/vol15/p3750-panse.pdf, 2022.
[18] J. Schleier-Smith et al., "What Serverless Computing Is and Should Become," *Communications of the ACM*, vol. 64, no. 5, pp. 76-84, 2021.
[19] M. Fowler, "CQRS," *MartinFowler.com*, https://martinfowler.com/bliki/CQRS.html, 2011.
[20] J. Duncan and C. O'Brien, "The engineering challenges of data privacy and GDPR compliance in distributed systems," *IEEE International Conference on Cloud Engineering (IC2E)*, https://www.researchgate.net/publication/351336495_The_Engineering_Challenges_of_Data_Privacy_and_GDPR_Compliance_in_Distributed_Systems ,2021.
[21] E. Evans, "Domain-Driven Design: Tackling Complexity in the Heart of

Software," Addison-Wesley Professional, https://www.domainlanguage.com/ddd/reference/ , 2003.
[22] A. Chebotko, A. Kashlev, and S. Lu, "A Big Data Modeling Methodology for Apache Cassandra," *IEEE International Congress on Big Data*, https://shiyong.eng.wayne.edu/papers/bigdata2015_andrey.pdf, 2015.
[23] D. B. Terry et al., "Session Guarantees for Weakly Consistent Replicated Data," *Proceedings of the Third International Conference on Parallel and Distributed Information Systems (PDIS)*, https://www.cs.utexas.edu/~lorenzo/corsi/cs380d/papers/p140-terry.pdf, 1994.
[24] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski, "Conflict-Free Replicated Data Types," *Symposium on Self-Stabilizing Systems (SSS)*, https://hal.inria.fr/inria-00609399/document, 2011.
[25] S. Newman, *Building Microservices: Designing Fine-Grained Systems*, 2nd ed., O'Reilly Media, https://samnewman.io/books/building_microservices_2nd_edition/, 2021.

Răzvan-Florin SOARE earned his bachelor's degree in Economic Informatics in 2024 and now he is a master's student at the Academy of Economic Studies from Bucharest, Databases – Support for Business. He will graduate from this program in 2026. Professionally, he works as a Dabase Developer in the banking sector, where he designs, optimizes, and manages complex data systems. Beyond his core expertise, he is deeply passionate about artificial intelligence and process automation.

Anda Elena Spătaru graduated at the Bucharest University of Economic Studies in 2024, earning a Bachelor's degree in Economic Informatics. She is currently pursuing a master's degree in Databases for Business Support, expected to be completed in 2026. Professionally, Anda has contributed to the development of digital solutions that enhance business efficiency and support data-driven decision-making. She began her career as a Data Engineer at PPC, later transitioning into data analysis and business intelligence development.

Miruna Șoșea graduated from the Faculty of Cybernetics, Statistics and Economic Informatics of the Academy of Economic Studies in 2024, obtaining a Bachelor's Degree in Economic Informatics. She is currently pursuing a master's degree in Databases – Business Support and is expected to graduate in 2026. She currently works as a Data Engineer, focusing on data pipelines, data warehousing, and large-scale data processing. Her main areas of interest include data engineering, data analysis, distributed systems, and database management.