Beyond One-Size-Fits-All: A Survey of NoSQL Databases and Polyglot Persistence Patterns

Hamish Burke
School of Engineering and Computer Science
Victoria University of Wellington
Wellington, New Zealand
burkehami@myvuw.ac.nz

***Abstract*—The exponential growth of web-scale applications has exposed fundamental limitations in traditional relational databases, driving the emergence of diverse NoSQL solutions. This paper provides a comprehensive survey of NoSQL databases, examining the limitations of traditional relational database systems that led to their development and analysing the different families of NoSQL data models. We explore the trade-offs between consistency, availability, and partition tolerance that characterise different NoSQL approaches, discuss their querying capabilities and performance considerations, and examine a possible future: polyglot persistence. We'll outline different architectural patterns for polyglot persistence and, through analysis of Data Access Object patterns and automated mediation approaches, demonstrate how modern applications can leverage multiple database technologies within unified architectures. We conclude by analysing current limitations and identifying polyglot persistence as a key direction for future database system design, moving beyond one-size-fits-all approaches toward specialised tools working in concert.**

***Index Terms*—NoSQL, databases, scalability, data models, distributed systems, CAP theorem, polyglot persistence**

I. Introduction

The rise of Web 2.0 applications, along with the rapid increase in data volume, velocity, and variety, has revealed significant limitations in traditional relational database management systems (RDBMS). As organisations manage tens of millions of customers globally and strive to sustain continuous growth, reliability has become a critical requirement [1]. Even minor outages can lead to substantial financial consequences. In this context, the rigid schema structure and strict ACID (Atomicity, Consistency, Isolation, Durability) guarantees of relational databases have often turned into obstacles rather than advantages [2].

This survey explores the landscape of NoSQL ("Not Only SQL") databases, analysing the specific challenges they address, the various architectural approaches they utilise, and their current limitations. We examine polyglot persistence as an alternative to choosing either RDBMS or NoSQL exclusively. We discuss different architectural patterns of polyglot persistence and demonstrate how this approach can lead to a more flexible and powerful architecture.

II. The Problem: Limitations of Relational Databases

Traditional relational databases face several fundamental limitations when dealing with modern web-scale applications and big data workloads. These limitations have led many organisations to adopt polyglot persistence strategies [3], using different database technologies for different aspects of their applications rather than forcing all data through a single system type.

A. Data Model Rigidity

Relational databases impose a strict, predefined tabular structure that is often ill-suited for the semi-structured or unstructured data typical in Web 2.0 applications. Consider Twitter's challenge of storing tweets with varying metadata—hashtags, mentions, media attachments, and location data—all of which can be absent or present in different combinations. Traditional RDBMS would require either numerous nullable columns or complex normalisation across multiple tables, both creating performance and maintenance challenges.

While some modern relational database management systems (RDBMS) have introduced support for formats like JSON [4], these formats are not part of their native storage paradigm. However, rigid schemas do provide certain advantages. For example, without a schema, the presentation layer, especially in an MVC architecture [5], would have no way of determining the format of incoming data. This situation essentially places all responsibility for data integrity on the application rather than the RDBMS itself [6].

B. Object-Relational Impedance Mismatch

The difficulty of transforming complex data structures from the object-oriented programming paradigm, like graphs, trees, or nested objects, into flat relational tables creates significant development overhead. LinkedIn's professional network, for instance, requires representing multi-level relationship hierarchies (connections, second-degree connections, company affiliations) that map unnaturally to relational structures. This mismatch often necessitates complex and performance-intensive [7] object-relational mapping (ORM) frameworks, which add another layer of abstraction and potential for inefficiency.

C. The Challenge of ACID in Distributed Systems

RDBMS systems are defined by their adherence to ACID properties:

- **Atomicity:** All operations in a transaction succeed or none do.

- **Consistency:** A transaction brings the database from one valid state to another.
- **Isolation:** Concurrent transactions produce the same state as if they were executed sequentially.
- **Durability:** Once a transaction is committed, it remains so.

While these guarantees are crucial for transactional integrity, they become a major bottleneck for horizontal scaling. Enforcing ACID across a distributed cluster, particularly through two-phase commit protocols, requires significant coordination between nodes. Operations like distributed joins and locks introduce high latency and create single points of failure, severely impacting performance and availability as the system scales out [2].

D. Scalability and Availability Constraints

RDBMS systems were primarily designed for vertical scaling (scaling up by adding resources to a single server). They do not scale horizontally (scaling out by distributing the load across multiple servers) as effectively due to their normalised data model and the aforementioned challenges of distributed ACID guarantees. For applications requiring 99.99% uptime, the strict consistency models of RDBMS can become limiting, as a network partition or node failure can render parts of the system unavailable. Systems like Amazon's Dynamo were explicitly designed to prioritise availability over immediate consistency, ensuring that read and write operations complete within milliseconds even in the face of network partitions [1].

III. NoSQL Database Families

NoSQL databases address these limitations through diverse architectural approaches. Four main families have emerged, each optimised for specific use cases. Rather than replacing relational databases entirely, these diverse NoSQL approaches enable organisations to select the most appropriate tool for each specific use case, forming the foundation for polyglot persistence architectures.

A. Key-Value Stores

These are the simplest models, where data is addressed by a unique key. The corresponding value is treated as an opaque blob of data (e.g., a byte array), which the database does not interpret.

- **Architecture:** A simple hash map.
- **Examples:** Redis [8], which keeps data in memory and supports advanced data structures like lists and sets within its values; Amazon DynamoDB [1].
- **Use Cases:** Caching, session management, real-time leaderboards.
- **Real-World Application:** GitHub uses Redis for caching user sessions and repository metadata, achieving sub-millisecond response times for frequently accessed data.

B. Document Stores

Document stores extend the key-value concept by storing semi-structured documents, typically in formats like JSON or BSON. Crucially, the value (document) is not opaque; the database understands its structure and can query based on internal fields.

- **Architecture:** (Key, Document) pairs with flexible schema.
- **Examples:** MongoDB [9], CouchDB, Firebase Firestore [10].
- **Use Cases:** Content management, e-commerce platforms, logging.
- **Real-World Application:** The Guardian newspaper uses MongoDB to store and serve articles with varying metadata structures (tags, authors, multimedia content) without requiring schema migrations for new content types.

C. Column-Family Stores

Inspired by Google's BigTable [11], these stores organise data into rows, but each row can have a different set of columns. Columns are grouped into predefined "column families." This architecture creates a sparse, distributed, persistent multidimensional sorted map.

- **Architecture:** A two-level map structure: a row key maps to column families, which in turn maps column names to values.
- **Examples:** Apache Cassandra [12], HBase.
- **Use Cases:** Write-heavy workloads, time-series data, IoT, large-scale messaging.
- **Real-World Application:** Netflix uses Cassandra to store viewing histories and recommendations for over 200 million users, handling millions of writes per second across global data centers while maintaining 99.99% availability.

D. Graph Databases

Graph databases are designed for highly interconnected data. Relationships (edges) are treated as first-class citizens, just like the data entities (nodes). Both nodes and edges can have properties, enabling the efficient execution of queries that traverse complex relationship networks [13].

- **Architecture:** Nodes, relationships, and properties form a Property Graph.
- **Examples:** Neo4j [14], JanusGraph.
- **Use Cases:** Social networks, recommendation engines, fraud detection, network topology.
- **Real-World Application:** LinkedIn uses graph databases to power their "People You May Know" feature, efficiently traversing relationship networks to suggest connections based on mutual contacts, shared employers, and educational backgrounds.

IV. Consistency, Availability, and the CAP Theorem

A fundamental concept in distributed systems is the CAP theorem, first formally proven by Gilbert and Lynch [15],

TABLE I
COMPARISON OF NOSQL DATABASE FAMILIES

Category	Key-Value	Document	Column-Family	Graph
Data Model	(Key, Opaque Value)	(Key, Interpreted Document)	Rows with column families	Nodes, Edges, Properties
Schema	Schema-less	Flexible schema per document	Predefined families, flexible columns	Flexible schema per node/edge
Primary Query	By key	By key or document content	By row key or column range	Graph traversal (e.g., Cypher, Gremlin)
Use Cases	Caching, sessions	Content mgmt, logs	Big data, IoT, writes	Social networks, fraud detection
Examples	Redis, DynamoDB	MongoDB, CouchDB	Cassandra, HBase	Neo4j, JanusGraph

which states that it is impossible for a distributed data store to provide more than two of the following three guarantees simultaneously:

- **Consistency (C):** Every read receives the most recent write or an error.
- **Availability (A):** Every request receives a (non-error) response, without the guarantee that it contains the most recent write.
- **Partition Tolerance (P):** The system continues to operate despite an arbitrary number of messages being dropped (or delayed) by the network between nodes.

Since network partitions are a reality in any large-scale distributed system, the trade-off is effectively between Consistency and Availability. The theorem has evolved to be understood less as a strict binary choice and more as a spectrum of trade-offs that system designers must navigate [16]. Many NoSQL systems are designed as AP systems, prioritising availability by relaxing strong consistency in favour of *eventual consistency*.

A. Tunable Consistency and Eventual Consistency

Rather than a binary choice, modern NoSQL systems like Cassandra offer *tunable consistency* [12]. Developers can specify the required consistency level on a per-operation basis, balancing latency and data correctness. Common levels include:

- **ONE:** Ensures the write is sent to at least one replica. Fastest but least consistent.
- **QUORUM:** Requires a majority of replicas ($(N/2)+1$, where N is the replication factor) to acknowledge the operation. This provides a strong guarantee of consistency if the condition $R+W > N$ is met (where R and W are the number of nodes for read and write quorums).
- **ALL:** Requires all replicas to acknowledge. Strongest consistency but lowest availability.

Facebook's approach illustrates practical CAP theorem trade-offs: their social graph prioritises availability (users can always post content) over immediate consistency (friends might see posts at slightly different times), while their financial systems prioritise consistency for payment processing.

B. Mechanisms for Achieving Eventual Consistency

AP systems use several mechanisms to converge on a consistent state over time:

- **Hinted Handoff:** If a node is down during a write, a coordinator node stores a "hint" and delivers the write when the target node recovers.
- **Read Repair:** During a read request, the coordinator can detect inconsistencies among replicas and repair the stale data in the background.
- **Anti-Entropy (Merkle Trees):** Nodes periodically compare their data using Merkle trees—a hash tree over key ranges—to efficiently identify and synchronise differences without transferring the entire dataset.

V. LIMITATIONS AND ONGOING CHALLENGES

Despite their advantages, NoSQL databases face several significant challenges that limit their adoption and effectiveness.

A. Querying and Data Access

The absence of a standardised query language represents one of the most significant barriers to NoSQL adoption. Unlike relational databases with SQL, each NoSQL system implements proprietary APIs or query languages—MongoDB uses its Query API [9], while Neo4j employs Cypher [14]. This fragmentation creates steep learning curves and increases the risk of vendor lock-in [17].

Several standardisation efforts have emerged to address this challenge. Generic abstraction APIs like SOS [18] attempt to decouple application logic from specific database choices, but their broad compatibility comes at the cost of limited expressivity, offering only basic GET, PUT, and DELETE endpoints. More specialised approaches show greater promise: SPARQL [19] provides SQL-like expressiveness for graph databases and has achieved successful standardisation by major institutions [20], though its scope remains limited to a single database model.

The querying limitations extend beyond language standardisation. Operations that are simple in SQL, such as ad-hoc joins across different entities, can become complex or even impossible in NoSQL systems. Developers often have to implement architectural workarounds, each of which comes with its own trade-offs:

- **Data Denormalisation:** Embedding related data within a single document reduces query complexity but introduces data redundancy and can lead to update anomalies.
- **Application-Side Joins:** Executing multiple queries and combining the results in application code shifts some of the database complexities to developers, which may negatively impact performance.

While initiatives to create uniform interfaces continue [18], [21], these solutions often introduce performance overhead and cannot fully abstract fundamental differences between data models. Consequently, architectural approaches have emerged to manage this complexity.

1) Data Access Patterns and Abstraction Layers: Beyond query language standardisation, architectural patterns have emerged to manage the complexity of diverse data stores. Data Access Objects (DAOs) serve as abstraction layers between application logic and database implementations, enabling greater flexibility in database selection and migration.

Pereira et al. [22] identify three primary patterns for implementing polyglot persistence:

- **Independent DAO:** Business logic directly calls separate DAOs for each database type, maintaining explicit control but increasing coupling.
- **Integrated Polyglot DAO:** A unified DAO interface handles multiple database types internally, simplifying business logic at the cost of increased DAO complexity.
- **Mediated DAO:** A mediator component routes operations to appropriate DAOs based on data characteristics or performance requirements, similar to the automated approaches discussed below.

These patterns address similar challenges to the generic APIs discussed earlier but focus on architectural flexibility rather than query language uniformity.

2) Automated Polyglot Persistence: Recent research has explored automating database selection decisions to reduce the operational overhead of managing multiple database systems. Schaarschmidt et al. [23] propose a Polyglot Persistence Mediator (PPM) that makes runtime routing decisions based on service level agreements and schema-based annotations. Their evaluation demonstrated 50-100% write performance improvements and reduced read latencies by automatically directing operations to the most suitable backend.

This automated approach addresses a key barrier to polyglot adoption: the expertise required to effectively manage multiple database systems. However, it introduces new challenges around mediator complexity, metadata management, and the need for sophisticated routing algorithms that understand workload characteristics.

B. Performance Considerations

NoSQL systems are not universally faster than RDBMS. Performance in database systems is fundamentally workload-dependent, making the choice between RDBMS and NoSQL a matter of matching system characteristics to specific requirements rather than assuming universal superiority.

1) Workload-Dependent Performance Characteristics: While NoSQL excels at high-throughput writes and simple reads by key, RDBMS often outperforms on complex analytical queries due to mature query optimisers [24]. The Yahoo! Cloud Serving Benchmark (YCSB) [25] provides standardised workloads for evaluating performance across different systems, revealing significant variations based on access patterns:

- **Read-Heavy Workloads:** Traditional RDBMS systems like MySQL demonstrate competitive read latencies, while some NoSQL systems like Cassandra and HBase show higher read latencies due to their write-optimised architectures.
- **Write-Heavy Workloads:** NoSQL systems designed for write optimisation, particularly Cassandra and HBase, achieve significantly lower update latencies compared to traditional relational systems.
- **Mixed Workloads:** Systems like PNUTS (Yahoo's distributed database) demonstrate more balanced performance across both read and write operations.

Uber's architecture exemplifies these trade-offs: they use Cassandra for high-write trip data, Redis for real-time driver locations, and PostgreSQL for transactional operations like payments, each optimised for specific workload characteristics.

2) Scalability and Elasticity Performance: YCSB evaluations reveal important differences in how systems handle scale:

Horizontal Scaling: Both PNUTS and Cassandra demonstrated effective scaling as the number of servers and workload increased proportionally. However, HBase exhibited more erratic performance characteristics during scaling operations, highlighting the importance of system-specific tuning and operational expertise.

Elastic Growth: While Cassandra, HBase, and PNUTS all supported elastic scaling during live workload execution, PNUTS provided the most stable latency characteristics during data repartitioning operations. This stability is crucial for production systems that cannot tolerate performance degradation during scaling events.

3) Trade-offs in System Design: The YCSB results confirm hypothesised trade-offs between read and write optimisation in distributed systems. Systems optimised for write throughput (like Cassandra's log-structured merge trees) inherently sacrifice some read performance, while read-optimised systems may struggle with write-intensive workloads. This reinforces the argument for polyglot persistence, where different systems can be selected based on specific access patterns within the same application.

4) Benchmark Limitations and Real-World Considerations: While standardised benchmarks like YCSB provide valuable comparative data, they represent simplified workloads that may not capture the complexity of real-world applications. Specific studies, such as the comparison between MongoDB and Cassandra by Abramova and Bernardino [26], highlight how performance can vary dramatically based on data scale, query complexity, and consistency requirements. Production performance depends heavily on factors including:

- Data distribution and hotspot patterns
- Network topology and latency characteristics
- Consistency level requirements
- Concurrent user patterns and peak load characteristics

These real-world complexities emphasise the importance of application-specific performance testing and the value of polyglot architectures that can adapt to diverse performance requirements within a single system.

C. Operational Complexity

The operational expertise required to deploy, monitor, and maintain large-scale distributed NoSQL clusters is significant and less widely available than RDBMS expertise. Managing eventual consistency adds application-level complexity, as developers must account for stale reads and implement conflict resolution logic.

VI. Conclusion

The evolution of database systems reflects the changing requirements of modern applications rather than a fundamental superiority of one approach over another. NoSQL databases emerged to address specific challenges of scale, availability, and data model flexibility where traditional relational systems were not the optimal fit.

Each NoSQL family serves distinct use cases and makes different trade-offs, particularly between consistency and availability. Database choice should be driven by careful analysis of application requirements—data structure, query patterns, scalability needs, and transactional guarantees—rather than by general assumptions about performance.

The future of data management lies in "polyglot persistence," where different database technologies are leveraged for specific tasks within the same application ecosystem [3]. As demonstrated by the DAO patterns [22] and automated mediation approaches [23], architectural solutions are emerging to manage the complexity of such heterogeneous environments. These patterns enable organisations to leverage the strengths of both relational and non-relational systems while mitigating the operational overhead through abstraction layers and intelligent routing mechanisms.

This approach aligns with the long-standing argument that a "one size fits all" approach to database systems is suboptimal [27]. By understanding the deep technical trade-offs, including the operational complexities discussed, and by employing appropriate architectural patterns, we can build more robust, performant, and maintainable systems that adapt to diverse data requirements [17].

References

[1] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: Amazon's highly available key-value store," *ACM SIGOPS operating systems review*, vol. 41, no. 6, pp. 205–220, 2007.

[2] R. Hecht and S. Jablonski, "Nosql evaluation: A use case oriented survey," in *2011 International Conference on Cloud and Service Computing*, pp. 336–341, IEEE, 2011.

[3] P. P. Khine and Z. Wang, "A review of polyglot persistence in the big data world," *Information*, vol. 10, no. 4, p. 141, 2019.

[4] C. Chasseur, Y. Li, and J. M. Patel, "Enabling json document stores in relational systems.," in *WebDB*, vol. 13, pp. 14–25, 2013.

[5] J. Deacon, "Model-view-controller (mvc) architecture," *Online][Citado em: 10 de março de 2006.] http://www. jdl. co. uk/briefings/MVC. pdf*, vol. 28, p. 61, 2009.

[6] J. Celko, *Joe Celko's Complete Guide to NoSQL: What Every SQL Professional Needs to Know about Non-Relational Databases*. Newnes, 2013.

[7] D. Colley, C. Stanier, and M. Asaduzzaman, "The impact of object-relational mapping frameworks on relational query performance," in *2018 International Conference on Computing, Electronics & Communications Engineering (iCCECE)*, pp. 47–52, IEEE, 2018.

[8] "Redis becomes the most popular database on AWS as complex cloud application deployments surge — theregister.com." https://www.theregister.com/2020/11/23/redis_the_most_popular_db_on_aws/. [Accessed 12-06-2025].

[9] A. Chauhan, "A review on various aspects of mongodb databases," *International Journal of Engineering Research & Technology (IJERT)*, vol. 8, no. 05, pp. 90–92, 2019.

[10] R. Kesavan, D. Gay, D. Thevessen, J. Shah, and C. Mohan, "Firestore: The nosql serverless database for the application developer," in *2023 IEEE 39th International Conference on Data Engineering (ICDE)*, pp. 3376–3388, IEEE, 2023.

[11] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Bigtable: A distributed storage system for structured data," *ACM Transactions on Computer Systems (TOCS)*, vol. 26, no. 2, pp. 1–26, 2008.

[12] A. Lakshman and P. Malik, "Cassandra: a decentralized structured storage system," *ACM SIGOPS operating systems review*, vol. 44, no. 2, pp. 35–40, 2010.

[13] R. Angles and C. Gutierrez, "Survey of graph database models," *ACM Computing Surveys (CSUR)*, vol. 40, no. 1, pp. 1–39, 2008.

[14] J. J. Miller, "Graph database applications and concepts with neo4j," in *Proceedings of the southern association for information systems conference, Atlanta, GA, USA*, vol. 2324, pp. 141–147, 2013.

[15] S. Gilbert and N. Lynch, "Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services," *Acm Sigact News*, vol. 33, no. 2, pp. 51–59, 2002.

[16] E. Brewer, "Cap twelve years later: How the" rules" have changed," *Computer*, vol. 45, no. 2, pp. 23–29, 2012.

[17] C. Mohan, "History repeats itself: sensible and nonsensql aspects of the nosql hoopla," in *Proceedings of the 16th International Conference on Extending Database Technology*, pp. 11–16, 2013.

[18] P. Atzeni, F. Bugiotti, and L. Rossi, "Uniform access to nosql systems," *Information Systems*, vol. 43, pp. 117–133, 2014.

[19] R. Angles and C. Gutierrez, "The expressive power of sparql," in *International Semantic Web Conference*, pp. 114–129, Springer, 2008.

[20] V. Santos and B. Cuconato, "Nosql graph databases: an overview," *arXiv preprint arXiv:2412.18143*, 2024.

[21] A. H. Chillón, M. Klettke, D. S. Ruiz, and J. G. Molina, "A generic schema evolution approach for nosql and relational databases," *IEEE Transactions on Knowledge and Data Engineering*, vol. 36, no. 7, pp. 2774–2789, 2024.

[22] F. Pereira, E. Guerra, and R. R. Rosa, "Patterns for polyglot persistence layer," in *Proceedings of the 29th Conference on Pattern Languages of Programs*, pp. 1–8, 2022.

[23] M. Schaarschmidt, F. Gessert, and N. Ritter, "Towards automated polyglot persistence," in *Datenbanksysteme für Business, Technologie und Web (BTW 2015)*, pp. 73–82, Gesellschaft für Informatik eV, 2015.

[24] A. Floratou, N. Teletia, D. J. DeWitt, J. M. Patel, and D. Zhang, "Can the elephants handle the nosql onslaught?," *arXiv preprint arXiv:1208.4166*, 2012.

[25] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with ycsb," in *Proceedings of the 1st ACM symposium on Cloud computing*, pp. 143–154, 2010.

[26] V. Abramova and J. Bernardino, "Nosql databases: Mongodb vs cassandra," in *Proceedings of the international C* conference on computer science and software engineering*, pp. 14–22, 2013.

[27] M. Stonebraker and U. Çetintemel, "" one size fits all" an idea whose time has come and gone," in *Making databases work: the pragmatic wisdom of Michael Stonebraker*, pp. 441–462, 2018.