IT Project Guidance

On Datastores

Version:

0.3

## Purpose

The purpose of this document is to provide technical decision makers with a high-level overview of the different types of datastores available for IT projects. It outlines typical use cases, key advantages, and essential factors to consider when selecting the most appropriate data storage solution.

## Synopsis

Modern IT projects benefit from a multi-database approach, leveraging different Datastores—such as document stores, relational databases, and time-series solutions—each tailored for specific needs. Rather than replacing traditional databases, new technologies like NoSQL complement established options, expanding the range of possibilities. Selecting the right combination of Datastores allows technical decision makers to ensure performance, scalability, and flexibility as project requirements evolve.

## Contents

[Purpose 1](#_Toc206589092)

[Synopsis 1](#_Toc206589093)

[Contents 2](#_Toc206589094)

[Purpose and Audience 3](#_Toc206589095)

[Background 3](#_Toc206589096)

[NoSQL 4](#_Toc206589097)

[Modern System Design 5](#_Toc206589098)

[Configuration Data Store 5](#_Toc206589099)

[Relational Datastores 7](#_Toc206589100)

[File System Datastores 8](#_Toc206589101)

[Object Datastores 10](#_Toc206589102)

[Key Management Service (KMS) Datastores 11](#_Toc206589103)

[Key Value Datastores 13](#_Toc206589104)

[Document Datastores 14](#_Toc206589105)

[Event Store 16](#_Toc206589106)

[Graph Datastores 16](#_Toc206589107)

[Search Service Datastores 18](#_Toc206589108)

[Recommendations 19](#_Toc206589109)

[Conclusion 20](#_Toc206589110)

[Appendices 21](#_Toc206589111)

[Appendix A - Document Information 21](#_Toc206589112)

[Versions 21](#_Toc206589113)

[Images 21](#_Toc206589114)

[Tables 21](#_Toc206589115)

[References 21](#_Toc206589116)

[Review Distribution 21](#_Toc206589117)

[Audience 21](#_Toc206589118)

[Structure 21](#_Toc206589119)

[Diagrams 22](#_Toc206589120)

[Acronyms 22](#_Toc206589121)

[Terms 22](#_Toc206589122)

[Appendix B - Event-Based versus State-Based Information Management 23](#_Toc206589123)

[When to Consider Use Event Store Architectures 25](#_Toc206589124)

[Appendix C – Future Trends 26](#_Toc206589125)

[Core Browser-Based Storage Options 26](#_Toc206589126)

# Purpose and Audience

This document aims to equip technical decision makers with clear criteria and context for selecting the most appropriate type of datastore for a given scenario. While the landscape of data storage has rapidly expanded to include a diverse array of solutions—most notably various NoSQL technologies alongside traditional relational databases—making an informed and effective choice requires understanding not only the strengths of each, but also the risks inherent in misapplication.

# Background

Relational Datastores have dominated the landscape since the 1970s[[1]](#footnote-2),[[2]](#footnote-3), not merely by virtue of market momentum, but because they consistently deliver robust solutions to well-understood problems. Their enduring popularity stems from their reliability and the depth of functionality they provide—features essential to many business-critical systems.

Yet, working effectively with relational databases demands a significant investment in upfront design. Architects and developers must carefully craft schemas, mapping out entities and their interrelationships with precision. The initial migration of data, as well as the ongoing maintenance and evolution of these schemas, requires forethought and discipline, as even minor missteps can lead to complex challenges down the line.

Mastery of relational database design is a specialized skill set, distinct from general coding or application logic. Developers unfamiliar with these principles often find themselves frustrated by unexpected complexities or unintended consequences—issues that arise not from the inherent limitations of relational technology, but from a lack of familiarity with its rigorous demands. Successful system development in this context hinges as much on understanding the science of data modelling and schema management as it does on writing functional code.

The lessons learned from decades of experience with relational systems continue to inform best practices and highlight the importance of blending technical proficiency with strategic design, ensuring that foundational choices support both immediate project needs and long-term sustainability.

Relational databases remain the gold standard when transactional integrity, complex multi-table queries, and a robust, time-tested ecosystem are critical to business needs.

However, they do have issues in areas they were not designed for.

# Not Only SQL (NoSQL)

A close-up of a sign

AI-generated content may be incorrect.

Figure 1: Not Only SQL (No SQL) storage options

Since the early 2000s, the landscape of data storage has evolved dramatically, giving rise to an array of models broadly grouped under the banner of "Not Only SQL" (NoSQL). This new generation of Datastores encompasses a variety of paradigms—key/value memory cache databases, media blob storage databases, JSON document databases, and graph databases, to name only the most prominent. Each of these technologies specializes in addressing specific requirements, filling distinct niches within the broader realm of data storage.

A frequent misstep in system design is assuming that NoSQL databases, due to their more recent emergence, inherently surpass traditional relational databases in modernity, capability, flexibility, or scalability. This belief, though understandable, oversimplifies the true relationship between these technologies.

In practice, NoSQL solutions are best understood not as wholesale replacements for relational databases, but as strategic complements. They are adept at resolving challenges that relational databases were not originally built to solve—such as efficiently managing unstructured or semi-structured data, scaling horizontally across distributed systems, and enabling rapid development cycles or prototyping in dynamic environments. These strengths make NoSQL an invaluable asset in situations where data models are fluid, enormous volumes of information must be processed in near real-time, or the strict structure of relational schemas would hinder progress.

However, they have other non-trivial disadvantages that cannot and should not be overlooked, from larger storage needs, expensive correction of data, lack of migration tooling.

Their mature tooling, comprehensive support for data consistency, and deep-rooted best practices ensure a level of reliability and predictability that is often indispensable for mission-critical applications.

# Modern System Design

A black and white diagram

AI-generated content may be incorrect.

Figure 2: Modern system design storage connections

The most effective architectural designs recognize the strengths and limitations of both approaches. Rather than defaulting to one technology over another based on trends or assumptions, sophisticated systems are built by deliberately matching each datastore’s unique characteristics to the specific functional and non-functional requirements of the project at hand. In this way, organizations can leverage the full spectrum of data storage possibilities, maximizing both immediate effectiveness and long-term sustainability.

The core purpose of this guide is to prevent the costly missteps that occur when system design prematurely pivots to a single datastore model—often NoSQL—without fully appreciating what is being gained and, more importantly, what is being lost. By illuminating the complementary strengths and appropriate use cases of each datastore type, this section will help ensure that architectural decisions are grounded in strategic understanding, rather than driven by hype or misconception.

Intended for architects, lead developers, and technical product owners, this guidance stresses the importance of a nuanced, scenario-driven approach to datastore selection—recognizing that the best outcomes are achieved not by defaulting to a single technology, but by thoughtfully matching datastore characteristics to project requirements and long-term goals.

# Configuration Data Store

Configuration files are a basic yet crucial form of data storage, containing deployment-time parameters such as service URLs and environment variables. Their immutability ensures stability, as changes occur only through redeployment. It’s important to keep configuration (static, set at deployment) separate from settings (dynamic, changed at runtime), with the latter stored in primary datastores.

Sensitive information like credentials or keys should not be kept in configuration files; secure vaults are preferred. Common formats—JSON, YAML, XML, INI—each offer trade-offs in readability and tooling. In large or cloud-based systems, centralized configuration management helps maintain consistency and streamline updates.

**Advantages**

* Simple to implement and maintain.
* Human-readable formats enable easy manual editing and review.
* Facilitates rapid deployment and environment-specific customization.
* Immutable by design, reducing risk of unexpected change during runtime.

**Considerations**

* Distinction between configuration (immutable, deployment-time) and settings (mutable, runtime) must be rigorously maintained.
* Format choice impacts readability, error-proneness, and compatibility with automation tooling.
* Version control for configuration files is essential to track changes and facilitate rollbacks.
* Centralized management may be required for large-scale or distributed systems.

**Disadvantages**

* Not suitable for storing sensitive information due to potential security risks.
* Manual edits can introduce errors, especially in complex formats like YAML.
* Requires redeployment to alter configuration, limiting in-the-moment flexibility.
* Lack of runtime adaptability compared to dynamic settings managed in databases or other stores.

**Maintainability**

Maintainability depends on clear separation of concerns, standardized formats, and thorough documentation. Configuration files should be well-organized and annotated for easy updates. Version control enables collaboration and traceability, allowing teams to track and revert changes. Automated validation and linting catch errors early, ensure consistency, and reduce maintenance burdens. In distributed systems, centralized management simplifies updates and audits.

**Security Considerations**

Prioritizing security in configuration management is critical, especially when handling sensitive data. Avoid storing secrets in plain files, as they risk exposure through broad access or version control. If credentials are present in configuration files and were not mistakenly introduced during deployment (they should be injected into a KMS), this strongly suggests that those credentials are exposed in the code repository itself—an exceptionally serious security vulnerability. Use secure vaults or secret stores with strict access controls to mitigate this risk. Regular audits help catch inadvertent leaks, and access should be limited and monitored. Encryption and strong authentication further safeguard configuration details, protecting operational integrity.

# Relational Datastores

Effective database design demands careful schema definition and strategic indexing for both performance and maintainability. Indexes on primary keys (PKs), foreign keys (FKs), and frequently queried reference data are essential—they enable fast data retrieval and efficient joins, minimizing costly full table scans.

However, indexing comes with trade-offs. Each index must be updated with every insert, update, or delete, so unnecessary indexes can slow down write operations and bloat storage. Select indexes deliberately, guided by real-world query usage and database statistics.

Prioritize high-cardinality fields—those with many unique values, such as user IDs or order numbers—which offer the greatest performance gains from indexes. Indexing low-cardinality fields (like simple status flags) often yields little benefit and may actually degrade efficiency.

A balance is critical: too few indexes result in slow queries, too many hurt write performance and inflate system overhead. Regularly monitor usage patterns and adjust indexes as application needs evolve.

Relational databases excel at storing structured metadata about media, but are not ideal for large media files themselves. Storing bulky media objects in the database can slow down backups and restores, and complicate disaster recovery.

**Advantages**

* Rapid query performance via strategic indexing
* Reliable metadata storage and relational integrity
* Optimized support for high-cardinality field lookups

**Considerations**

* Monitor query patterns to refine index selection
* Maintain indexes on PKs, FKs, and high-cardinality fields
* Limit database use to metadata, not large media storage

**Disadvantages**

* Write operations slowed by excessive or unnecessary indexes
* Storage overhead increases with more indexes
* Poor performance for storing or retrieving large media files

**Maintenance considerations**

Regular database maintenance is essential for performance and data integrity. Indexes—especially on high-cardinality and frequently queried fields—should be periodically reviewed and tuned to match evolving data and query patterns. Routine backups and integrity checks are necessary to prevent data loss. Monitoring the schema for consistency with application and compliance requirements helps avoid drift. Plan maintenance windows to minimize user disruption during changes or large data operations.

**Security considerations**

Robust security is essential for protecting metadata in databases such as SQL Server and PostgreSQL. Begin by implementing access controls based on the principle of least privilege: grant only necessary permissions, restrict administrative privileges to a trusted group, and limit database access to specific application servers. Remote administration should be blocked except over secure channels—allow only the required ports (e.g., 5432 for PostgreSQL or 1433 for SQL Server), and use firewalls to restrict inbound traffic.

Encrypt data both at rest and in transit. For client connections, enable TLS/SSL (such as sslmode=require for PostgreSQL or forcing encrypted connections in SQL Server via Force Encryption settings). Use disk-level encryption features provided by your operating system or cloud provider to safeguard stored data.

Conduct regular audits of user accounts and privileges, scheduling reviews of roles and connection logs. Employ built-in audit features (like SQL Server Audit or PostgreSQL’s log\_connections) and centralize critical events for monitoring and compliance.

Apply patches and updates promptly—subscribe to security bulletins for SQL Server and PostgreSQL, and test updates in staging before deploying to production. Segment database servers on separate networks or subnets, keeping production environments isolated from development and test systems, and block all unnecessary ports.

For integrations, use service accounts with unique, strong credentials, and rotate passwords or keys regularly. When interfacing with external applications, ensure secure connections via SSL/TLS and validate both endpoints.

By focusing on targeted security controls—access restrictions, encryption, auditing, patching, and architectural segmentation—you can effectively protect sensitive data in SQL Server and PostgreSQL environments without unnecessary complexity.

# File System Datastores

File system storage, once standard for user-uploaded media, relied on shared drives or local disks. However, as organisations transitioned to cloud architectures, storing media on device-specific disks introduced significant risks: unauthorised access due to shared infrastructure, limited or short-lived audit logs, and the potential for irretrievable data loss when devices changed or access was lost. Replication was not automatic; custom replication strategies often ran into technical constraints, such as a maximum number of files per folder. These issues made file system storage—shared or isolated—ill-suited to scalable, secure, and resilient modern environments. Continued reliance on file system storage impedes cloud migration and robust, redundant architecture.

**Advantages**

* Simple to set up for small-scale or legacy environments
* Supports legacy applications and workflows
* Provides hierarchical file organisation familiar to users

**Considerations**

* Custom replication and backup required for resilience
* Limited, transient access auditing
* Scalability is constrained by file system limits (e.g., max files per folder)
* May hinder integration with cloud-native services

**Disadvantages**

* Risk of data loss during device change or failure
* Auditing and monitoring capabilities are minimal
* Challenging to support geo-redundancy and disaster recovery
* Potential for unauthorised access on shared infrastructure

**Maintenance Considerations**

Object storage solutions, while highly durable, demand thoughtful maintenance planning. Regular verification of backup routines for both object data and its linked metadata is essential, as accidental deletions or provider outages can still result in data loss. Lifecycle management policies—such as automatic tiering or deletion of objects based on age—should be implemented to optimize costs and storage consumption. Periodic reviews of access logs and storage metrics can reveal usage patterns and help forecast capacity needs, preventing unexpected overruns. Additionally, regular testing of restoration procedures ensures that recovery processes are reliable and meet organizational recovery time objectives (RTO). For example, running quarterly drills to restore a sample media set from backups can surface procedural gaps before an actual incident occurs.

**Security Considerations**

Securing object storage involves a layered approach. All access should be governed by strict identity and access management (IAM) policies, granting the minimum necessary permissions to users and services. Encryption at rest and in transit is a must: most providers support server-side encryption by default (such as Azure’s SSE or AWS KMS), but sensitive data may require client-side encryption prior to upload. Access keys and tokens should be rotated regularly, and unused credentials promptly revoked. Public access must be tightly controlled; even seemingly innocuous objects can leak metadata or facilitate enumeration attacks if inadvertently exposed. Audit logging should be enabled to track all operations—uploads, downloads, deletions, and permission changes—and these logs should be regularly reviewed for anomalies. For instance, configuring alerts for repeated failed access attempts or unexpected public object exposure can provide early warning of misconfiguration or malicious activity.

# Object Datastores

Object storage—such as Blob Storage (Azure) and S3 (AWS)—is the preferred solution for storing serialized media streams in cloud environments. Critical metadata (e.g., hashtype, hash value, media type, creation/upload dates, original filename, classification, uploader’s userId, last malware scan date) should be retained in a relational database. This separation ensures media remains accessible and unaffected by device allocation changes, while metadata is efficiently managed.

This division of duties between storage types leverages their respective strengths. However, it requires robust backup and restoration strategies for both the primary relational database and the object storage layer. While many organizations rely on the inherent guarantees of their cloud providers, it is essential to review and validate these assurances.

A related option, file storage (e.g., Azure Files, AWS EFS), provides hierarchical structures and access via SMB or NFS, making it suitable for replicating legacy LAN conditions in the cloud. The choice between Blob and File Storage hinges on specific use cases.

**Advantages**

* High-speed access to large objects and media files
* Geo-redundancy is built-in, supporting high availability
* Separation of media and metadata boosts resilience and manageability
* Supports both public and secure backchannel access for flexible integration
* File storage subtypes allow seamless migration of legacy workflows

**Considerations**

* Comprehensive backup and restoration plans must cover both object and metaDatastores
* Geo-redundancy, while robust, should not replace independent off-cloud backups
* Data transfer costs can accrue, especially through secure backchannel access
* Integration with data warehouses and archiving systems may require additional planning
* Access models (public vs. backchannel) should be clearly defined and secured

**Disadvantages**

* Reliance on provider geo-redundancy can lead to complacency and lack of independent backups
* Potential risk during disaster recovery exercises if environments are deleted without additional off-cloud safeguards
* Configuration and management complexity increases with multiple storage types

**Maintenance considerations**

Regular maintenance of object and metadata storage systems requires thorough backup strategies that include independent, off-cloud backups to guard against cloud provider outages or accidental deletions. Administrators must actively monitor data transfer costs, especially when using secure backchannel access, as these can accumulate unexpectedly. Integration with downstream systems such as data warehouses or archiving solutions demands careful versioning and schema management to ensure compatibility and prevent data loss. Furthermore, clear documentation and automation of backup, restore, and configuration procedures can minimize human error and streamline disaster recovery exercises. For example, in environments using Azure Blob Storage or AWS S3, scheduled automated snapshots and regular restoration tests are essential to validate backup integrity and recovery speed.

**Security consideration**

Securing object and metaDatastores hinges on limiting access to storage endpoints and enforcing robust authentication methods. Public versus backchannel (private) access models must be clearly defined, with preference given to private endpoints shielded from the public internet. Encryption of data at rest and in transit is mandatory; utilizing built-in features like Azure Storage Service Encryption or AWS S3 Server-Side Encryption strengthens protection. Role-based access controls should restrict privileges to the least necessary, avoiding broad or lingering permissions. Additionally, audit logging must be enabled to track access and configuration changes, providing traceability in case of security incidents. For instance, configuring access policies with minimal privilege in AWS S3 or applying network rules to restrict Azure Blob Storage access can significantly reduce the risk of unauthorized exposure.

# Key Management Service (KMS) Datastores

As relational databases and blob storage are remote services, to access them, credentials (name and secret) are required to identify oneself and gain access to their services.

A Key Management Service (KMS) is used (keyvault in Azure, Secret Manager in AWS).

These confidential integration credentials are not to be confused with integration configuration settings, which do not require confidentialiaty. So, while one can put the url of the remote service in the on-device configuration store, one must store the confidential credentials in a more secure environment – a specialised keystore.

Vice versa, a Secure Store is not a configuration store, so any temptation of reusing it to storing non-confidential integration settings must be avoided.

Keys entered are not visible once put in. Access – specifically Changes (replacement of key values) are audited.

User account access to key vaults should be tightly limited to the admin who configures the key/value pairs, and that of the delivery pipeline that accesses it to retrieve secrets to establish identity to remote services (relational databases, blob storage, etc.). Nobody else – in any environment (ST, UAT, PP, PROD, etc.) by anyone, including testers wishing to access the database to ‘check data’.

Considerations are that once data has been retrieved, that it be kept in memory, and not written into any other datastore (configuration, database, etc.) as those stores are readable if the system is compromise. If in the database, restored backups also provide unaudited access to credentials.

**Maintenance considerations**

Key value datastores require careful maintenance to ensure optimal performance and reliability. Regular monitoring of memory usage is essential, especially as multiple stores may be created for different culture-language combinations, which can quickly consume available resources. Administrators should establish policies for cache eviction and expiration to avoid stale or unused data lingering in memory, setting appropriate cache durations ranging from seconds to minutes based on access patterns. For example, in Redis, automated eviction policies such as Least Recently Used (LRU) can help manage memory by removing old or infrequently accessed keys.

It is also important to plan for persistence settings according to system requirements. While key value stores like Redis offer options for snapshotting and append-only file persistence, these should be configured judiciously, recognizing that the primary function of such stores is performance, not long-term durability. Maintenance routines should include periodic reviews of cache hit rates and latency metrics, adjustments to shard or cluster size as data volumes grow, and routine software updates to safeguard against vulnerabilities and to benefit from performance improvements.

**Security consideration**

Securing key value datastores centres on limiting access to endpoints and enforcing robust authentication and authorization controls. As these stores often contain sensitive session and application data, it is vital to restrict access through network rules—such as firewall settings or private endpoints—so that only trusted services and users can connect. For instance, Redis on Azure recommends using Virtual Network integration with private endpoints, while AWS Elasticache can be deployed within a VPC with subnet-based access controls.

Authentication should be enabled and enforced, utilizing mechanisms like Redis AUTH or IAM policies in AWS, to prevent unauthorized access. Data encryption in transit, via TLS, and at rest (when persistence is enabled) should be configured to protect against interception or data leakage. Role-based access controls must be applied so that only necessary accounts have rights to read or write data; administrative commands, especially those that modify the cache structure or flush data, should be limited to privileged users.

Audit logging, where available, should be turned on to record changes in configuration and access patterns, aiding in traceability during security reviews or incident investigations. Regularly rotating credentials and reviewing access permissions further decreases the risk of exposure. As an example, Redis Enterprise allows for integration with external authentication providers and supports granular access control lists for command-level permissions, bolstering overall security posture.

# Key/Value Datastores

Key value stores primarily function as shared cache services, persisting values in memory with optional writes to persistent storage. Their main purpose is to cache data close to users, in a format tailored for immediate use, thus reducing repeated access to slower datastores and avoiding redundant computations. Examples include Redis cache, available on both Azure and AWS.

Unlike other datastores, key value stores do not guarantee that new values will persist in the event of a system crash; their role is to enable rapid restoration of most system functionality, not to serve as a source of record. This design focuses on performance rather than durability.

**Advantages**

* Significantly improves system performance, especially for distributed services.
* Supports storing data in its final, ready-to-use form—post-calculation, post-formatting, and in the language of the session—using language-tagged keys (e.g., ‘XYZ\_en’).
* Facilitates rapid access for high-frequency requests by caching under different culture-language codes.
* Enables flexible cache durations: short-term (1 second, for repeat queries in a single request), medium-term (5 seconds, suitable for up to 20,000 requests), long-term (30 seconds, up to 120,000 requests), and very long-term (5 minutes, for up to 1 million requests).

**Considerations**

* Multiple stores for various culture-language codes increase memory usage.
* Store data in the most frequently accessed “shape” to minimise transformation costs on retrieval.
* Potential data loss is limited to edge cases, requiring only re-fetch and re-caching of the affected data.

**Disadvantages**

* Relatively high operational cost due to memory usage-based billing.
* Lack of durability—recently cached data may be lost in the event of a crash, as persistence is not guaranteed.

**Maintenance considerations**

Regular monitoring and management of cache configurations are essential to ensure optimal performance and cost control. For systems supporting multiple culture-language codes, periodic audits can help identify under-utilized stores that unnecessarily consume memory. Automated pruning strategies, such as least recently used (LRU) eviction or time-to-live (TTL) expiry policies, prevent stale data from accumulating and causing inefficiencies. Testing cache refresh scenarios is important, especially in edge cases where data loss may occur and re-fetched data must be reliably re-cached. In practice, this means simulating crash recovery and validating that the system gracefully replenishes lost cache segments without impacting user experience. Maintenance is further streamlined when cache keys and schemas are standardized, allowing for easier bulk updates and compatibility checks as the underlying data shapes evolve with business requirements.

**Security consideration**

Protecting cached data, especially when it contains user-specific or sensitive session information, demands careful access control and encryption. For example, storing language-tagged keys such as ‘user\_profile\_en’ may inadvertently expose personal preferences or identifiers, so it's prudent to prevent unauthorized cache queries or memory dumps. Employing encryption for in-memory data and enforcing strict authentication for administrative cache operations mitigates risk in environments lacking persistence and durability. Furthermore, cache invalidation routines should be designed to avoid leaking residual data during store rotations or system restarts. Regular vulnerability assessments—including penetration tests targeting cache interfaces—are vital for identifying and remediating attack vectors. In distributed systems, network-level protections like TLS and firewall rules help ensure that inter-node cache traffic is shielded from interception or tampering.

# Document Datastores

Document datastores represent a distinct paradigm in modern data management, prioritizing flexibility and scalability over rigid, pre-defined schemas. Unlike traditional relational databases, document stores organize information as collections of self-describing documents—most commonly in formats like JSON, BSON, or XML—allowing varied and evolving data structures to coexist within the same data set. This model is especially well-suited for applications where requirements change frequently or where rich, nested objects are commonplace. By centering data around entire documents rather than normalized tables, these systems simplify both development and retrieval, making it easier to model real-world entities and support agile product evolution.

**Advantages**

* Efficient bulk or key-based retrieval, often outperforming relational joins for certain access patterns.
* Flexible schema: JSON (or BSON/XML) document storage allows rapid iteration and adapts to heterogeneous data structures.
* Designed for distributed, scalable architectures, enabling natural scale-out and high availability.
* Supports nesting of arrays and sub-documents, simplifying data modelling and retrieval without complex joins.
* Backup and restoration, when managed by traditional RDBMS, are straightforward and trusted for disaster recovery.

**Considerations**

* Schema flexibility accelerates development but increases the need for thorough testing and ongoing maintainability.
* JSON Schema validation may be implemented to ensure data integrity.
* If operating in a single region or with limited internationalisation, replication and complex distributed architectures may be unnecessary overhead.
* Relational databases now offer limited document support, but features may lag behind dedicated document stores.
* Traditional data warehouses expect relational data, complicating analytics on loosely structured documents.

**Disadvantages**

* Increased storage requirements due to key/value duplication across documents, impacting backup and restoration times.
* Some databases store documents as plain text rather than structured data, leading to significant performance penalties (up to 20x slower on SQL Server).
* Potential lack of durability: in-memory document caches may lose recent data upon crash if not persisted.
* Data transformation costs may rise if documents are not cached in their most frequently accessed “shape.”

**Maintenance considerations**

Document-oriented databases ease development by supporting flexible schemas, but this flexibility can complicate long-term maintenance. As schemas evolve, legacy documents may not conform to new validation rules, requiring periodic data migrations or transformation scripts to ensure consistency. For example, a MongoDB collection that began with a flat structure may later incorporate nested documents, necessitating updates to historical data for compatibility with new application features. Backup and restore operations can be time-consuming due to the increased storage footprint from key/value duplication and the potential for numerous, variably structured documents. Additionally, monitoring performance is crucial, as storage engines may degrade over time if documents are frequently rewritten in different “shapes.” Tools like MongoDB’s Compass or Elasticsearch’s Kibana can aid in tracking document growth, index usage, and anomalous patterns to pre-empt performance bottlenecks.

**Security consideration**

Document databases introduce unique security challenges that must be addressed to protect sensitive information. Because JSON or BSON documents can store arbitrary fields, strict access controls should be enforced at both the collection and field level. For instance, in a Couchbase or MongoDB deployment, role-based access control (RBAC) can restrict read or write permissions to approved users or applications. Data at rest should be encrypted, and if documents contain personally identifiable information (PII), field-level encryption or redaction may be required to comply with regulations like GDPR. Additionally, since some document databases expose RESTful APIs for CRUD operations, ensuring secure authentication (such as using OAuth or API keys) and guarding against injection attacks becomes critical. Regular audits, patching, and the use of secure transport protocols (TLS/SSL) are essential to maintaining a robust security posture.

# Event Data Store

A specialisation of Document Stores, see the analyses of Event and State management systems in the appendices.

# Graph Datastores

Graph databases are a specialised class of NoSQL databases designed to model, store, and query highly interconnected data. Instead of using tables or documents, graph databases represent information as nodes (entities) and edges (relationships), enabling intuitive traversal of complex networks such as social connections, recommendation systems, or supply chains.

Graph query languages such as Cypher (used by Neo4j) and Gremlin are common, but GraphQL—a query language and runtime—is increasingly prominent. While GraphQL is not itself a database, it provides a unified API to query graph-like structures, abstracting underlying storage (which may or may not be a native graph database). GraphQL excels at enabling clients to specify precisely what data they require, reducing over-fetching and enhancing performance, especially in microservices and federated environments.

**Advantages**

* Efficiently captures and queries relationships and connections that would be complex and expensive to model in relational or document databases.
* Enables rapid traversal of deeply nested or many-to-many relationships, supporting advanced analytics (e.g., recommendation, fraud detection).
* Flexible schema—nodes and relationships can evolve over time without major refactoring.
* GraphQL, as an API layer, supports decoupled, client-driven data access and can aggregate multiple sources seamlessly.
* Visual modelling and querying can be more intuitive for domain experts.

**Considerations**

* Graph databases shine in scenarios where connections are paramount, but may be overkill for flat or mostly tabular data.
* Performance can degrade with very large graphs unless indexed and sharded carefully; some systems struggle with massive, densely connected datasets.
* GraphQL endpoints, if not thoughtfully designed, can expose underlying complexity, risk inefficient queries, or become bottlenecks.
* Requires careful schema design to avoid "overfitting" everything into the graph model—attempting to represent all business logic or infrastructure as a graph leads to maintainability headaches.
* Integration with existing data platforms (e.g., ETL, analytics) may require custom connectors or pipelines.
* Security model must be adapted; explicit access control over nodes, edges, and traversal depth is essential, but standards are still evolving in some implementations.

**Disadvantages**

* Steep learning curve for teams new to graph theory, query languages, or graph-based APIs.
* Fewer mature, enterprise-ready solutions than relational or document stores; vendor lock-in is common.
* Not a one-size-fits-all solution: forcing all system data or logic into a graph leads to brittle, over-complicated architectures.
* Backup/restore and horizontal scaling may require specialised approaches—particularly for very large, distributed graphs.
* Tools for monitoring, debugging, and performance tuning are less standardised than in relational ecosystems.

**Maintainability considerations**

It’s easy to get caught up in the excitement of graphs’ expressive power. However, not all data or workflows are suited to graph representation, and these technologies are best deployed as complementary extension services—not as all-encompassing systems replacing every other storage or application layer. For most use cases, graphs best augment existing solutions rather than serve as the universal foundation for data storage.

**Security considerations**

As with other datastores, credentials and sensitive access information for graph databases or GraphQL endpoints should be managed via a Key Management Service (KMS). Avoid embedding secrets in application configuration files, version control, or within the database itself. Always enforce least-privilege access and monitor for unusual traversal behaviour to mitigate potential lateral movement risks.

# Search Service Datastores

Search Service Datastores are utilised for retrieving records based on approximate or phonetically similar spellings, enabling robust search capabilities even when data entry errors occur. Although not traditionally viewed as a primary datastore, these services operate as supplementary stores—persisting phonetic tokens and corresponding record IDs. Upon record creation, a queue is populated for asynchronous tokenisation across relevant locales, ensuring multi-lingual support and resilience. It is crucial that this queue is reliably maintained until processing completes, preventing data loss if exceptions arise during tokenisation.

Credential management for these services should be securely handled, ideally via a Key Management Service (KMS). When records are retrieved, the search service returns the correct record ID based on user queries, facilitating efficient and error-tolerant discovery.

A well-designed schema is essential, specifying which columns and tables are indexed for search. Consideration must be given to phonetic nuances across different cultures and languages (e.g., 'wh' in Māori pronounced as ‘f’, ‘homme’ in French as ‘om’), requiring thoughtful configuration. This aspect of discovery is often incorporated late in development, risking costly UX rework. Increasingly, AI-driven search services are supplanting traditional approaches, offering more intelligent matching but demanding deeper integration and schema design.

Cost is a factor to weigh, as advanced search services and AI integration can be expensive. Continuous evaluation of alternatives may be beneficial.

**Advantages**

* Supports robust search even with misspelled or phonetically varied input
* Facilitates multi-lingual and locale-aware retrieval
* Improves user experience in data discovery

**Considerations**

* Requires schema development to define searchable columns and tables
* Phonetic diversity across cultures necessitates careful configuration
* Early integration in development avoids costly redesign
* Credential management should leverage KMS for security

**Disadvantages**

* Higher cost, especially with AI-driven solutions
* Not a standalone solution; only one facet of comprehensive discovery
* Potential for processing delays if queue management fails

**Maintenance considerations**

Modern Datastores, whether RDBMS, NoSQL, or hybrid platforms, require ongoing maintenance to ensure optimal performance and reliability. Schema evolution can introduce complexity, particularly as business requirements change and new features are added. For example, evolving from a simple relational schema to include document storage in PostgreSQL may necessitate careful planning to avoid data inconsistency and migration issues. Regular index optimization, query performance tuning, and scheduled backups are essential to prevent data loss and minimize downtime. Additionally, monitoring for storage utilization and proactively scaling resources helps avoid outages, especially in cloud-based environments where usage can spike unexpectedly. Automated tools for patching and version upgrades can reduce operational overhead but must be tested carefully to prevent unexpected disruptions.

**Security consideration**

Security within data storage solutions must be multi-layered and adapt to evolving threats. Protecting credentials through solutions such as Key Management Services (KMS) is vital, as improper management can expose sensitive data. For instance, integrating KMS with cloud Datastores ensures that encryption keys are not hardcoded or managed manually, reducing the risk of unauthorized access. Access controls should be granular, granting permissions on a need-to-know basis and enforcing least privilege principles. Encryption at rest and in transit is mandatory for compliance with regulations and for safeguarding data integrity. Furthermore, auditing and logging access events are critical, providing visibility for detecting anomalous activities. In distributed environments, regular security reviews and penetration testing can identify and mitigate vulnerabilities before they are exploited.

# Summary

The following is a summary of the advantages, considerations, disadvantages of the various types of data store discussed in this document.

Table : ACiD Summary of Data Storage Types

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Datastore Type | Use Cases | Advantages | Considerations | Disadvantages |
| Search Service Datastores | Full-text search, analytics | Optimized for indexing and querying text | Requires synchronization with src systems | Not a primary data store |
| Key Management Service (KMS) Datastores | Secure key storage, encryption mgmt. | Centralized control, compliance support | Not intended for general data storage | Cannot be used as a general datastore |
| Key-Value Store | Caching, session storage, high-speed lookups | Ultra-fast access, simple API, highly scalable | Limited querying, typically no support for multi-item transactions | Lack of structure, not suitable for relational data or analytics |
| Relational Database Management Systems (RDBMS) | General-purpose databases, transactional workloads | Mature technology, strong consistency, supports structured data, some support NoSQL/document storage (e.g., PostgreSQL, SQL Server) | Traditionally schema-bound, may require schema migration for flexibility | Potentially less flexible for unstructured or rapidly evolving data models |
| Object Storage | Large media files, backups, unstructured data | Virtually unlimited scalability, cost-effective for large objects, accessible via HTTP APIs | Not designed for transactional workloads, eventual consistency | High latency for small objects, limited metadata querying |
| Document Stores | Flexible schema, unstructured or semi-structured data, high scalability  Content management, catalogs, user profiles | Not limited by rigid schemas, can handle diverse data types  Schema-less, stores complex objects natively, easy to scale | Integration with other systems (e.g., backup, disaster recovery) must be planned from the outset Eventual consistency models, diverse query languages, operational maturity varies. Indexing strategy is critical | May lack some transactional guarantees or features of RDBMS, operational complexity, weaker integration challenges, data duplication |
| Graph Database | Social networks, relationship analysis, fraud detection | Optimized for relationship queries, visual data modeling, flexible schema | Specialized query languages, niche expertise required | Not ideal for tabular data, scaling can be challenging for very large graphs |
| Search Service Datastores | Full-text search, analytics | Optimized for indexing and querying text | Requires synchronization with source systems | Not a primary data store |
| Hybrid Datastores (e.g., multi-model, NewSQL) | Mixed workloads, modern applications needing both structured and unstructured data | Combines strengths of relational and NoSQL, scalability options, flexible storage | Complex configuration, evolving standards, operational expertise required | Higher costs, possible compromise on specialized features, learning curve |

# Recommendations

Modern software design often starts with a single data store—typically an RDBMS—and evolves to include additional services as requirements mature. Many cloud-based services are cost-effective and easily accessible, but their selection and integration require careful planning.

The choice shouldn’t be framed as SQL versus NoSQL (the latter standing for “Not Only SQL”). Instead, each data store should be selected based on its fit for specific workloads and complemented with robust backup and restoration capabilities for disaster recovery.

This leads some architects to favour RDBMS platforms that also offer NoSQL/document storage features (such as PostgreSQL or SQL Server). These can simplify schema migration and centralize data management. However, ease of disaster recovery should not be the sole criterion for technology choices.

It’s crucial to evaluate each service for security, scalability, and operational suitability, ensuring that the architecture can evolve gracefully while supporting both immediate and long-term project needs.

# Conclusion

Relying solely on a single datastore is often a sign that critical considerations—such as security (e.g., the absence of KMS), discovery (lack of search capability), and performance (no caching layer)—may have been overlooked. Choosing a technology because it is perceived as “more modern” should not outweigh evaluating its fit across the full project lifecycle, including both initial deployment and long-term operations. Prioritise solutions based on their concrete capabilities and alignment with project goals, not just current trends.

Appendices

Appendix A - Document Information

Authors & Collaborators

* Sky Sigal, Solution Architect

### Versions

* 1. Initial Draft
  2. Added Summary

### Images

[Figure 1: Not Only SQL (No SQL) storage options 4](#_Toc206668515)

[Figure 2: Modern system design storage connections 5](#_Toc206668516)

### Tables

[Table 1: ACiD Summary of Data Storage Types 20](#_Toc206668410)

[Table 2: Offline Storage Options by Platform 28](#_Toc206668411)

### References

**There are no sources in the current document.**

### Review Distribution

The document was distributed for review as below:

|  |  |
| --- | --- |
| Identity | Notes |
| Amy Orr, Enterprise Data Architect |  |
| Russell Campbell, Project Manager |  |
| Carl Klitscher, Solution Architect |  |
| Harry Nguyen, Lead Cloud Engineer |  |

### Audience

The document is technical in nature, but parts are expected to be read and/or validated by a non-technical audience.

### Structure

Where possible, the document structure is guided by either ISO-\* standards or best practice.

### Diagrams

Diagrams are developed for a wide audience. Unless specifically for a technical audience, where the use of industry standard diagram types (ArchiMate, UML, C4), is appropriate, diagrams are developed as simple “box & line” monochrome diagrams.

### Acronyms

API

: [Application Programming Interface](#Term_ApplicationProgrammingInterface).

DDD

: Domain Driven Design

GUI

: [Graphical User Interface](#Term_ApplicationProgrammingInterface). A form of [UI](#Acronym_UI).

ICT

: acronym for Information & Communication Technology, the domain of defining Information elements and using technology to automate their communication between entities. [IT](#Acronym_IT) is a subset of ICT.

IT

: acronym for Information, using Technology to automate and facilitate its management.

UI

: User Interface. Contrast with [API](#Acronym_API).

### Terms

Refer to the project’s Glossary.

Application Programming Interface

: an Interface provided for other systems to invoke (as opposed to User Interfaces).

Capability

: a capability is what an organisation or system must be able to achieve to meet its goals. Each capability belongs to a domain and is realised through one or more functions that, together, deliver the intended outcome within that area of concern.

Domain

: a domain is a defined area of knowledge, responsibility, or activity within an organisation or system. It groups related capabilities, entities, and functions that collectively serve a common purpose. Each capability belongs to a domain, and each function operates within one.

Entity

: an entity is a core object of interest within a domain, usually representing a person, place, thing, or event that holds information and can change over time, such as a Student, School, or Enrolment.

Function

: a function is a specific task or operation performed by a system, process, or person. Functions work together to enable a capability to be carried out. Each function operates within a domain and supports the delivery of one or more capabilities.

Person

: a physical person, who has one or more Personas. Not necessarily a system User.

Persona

: a facet that a Person presents to a Group of some kind.

Quality

: a quality is a measurable or observable attribute of a system or outcome that indicates how well it meets expectations. Examples include reliability, usability, and performance. Refer to the ISO-25000 SQuaRE series of standards.

User

: a human user of a system via its UIs.

User Interface

: a system interface intended for use by system users. Most computer system UIs are Graphics User Interfaces ([GUI](#Acronym_GUI)) or Text/Console User Interfaces (TUI).

Appendix B - Event-Based versus State-Based Information Management

Event Store architectures represent a fundamental shift from the paradigms underpinning traditional data systems. Where classic relational database management systems (RDBMS) are built around persisting current state and auditing changes within that state—often via structured tables, history columns, and carefully managed metadata—event-based architectures focus instead on capturing a chronological log of what happened, not just what is. Each change, rather than being an update to a record, is represented as a discrete event: a message describing the occurrence and, crucially, containing a payload of arguments that detail the nature and context of the event.

Think of it as a “fire and remember” stack: when something happens—a user updates their profile, a purchase is made, or a sensor reports a reading—an event is generated and stored. The system does not overwrite previous data; rather, it accumulates a sequence of immutable records.

Unlike traditional systems, where the current state is always available and updated in-place, event stores require you to replay the entire history of events from the beginning to reconstruct the current state. This is both powerful and challenging, marking a clear departure in design philosophy.

**Advantages**

* Auditability and Replay: Every event is recorded in detail, providing a complete, tamper-evident log of all actions. This makes it possible to audit not only what changed and who changed it, but also to replay any (or all) events to reconstruct historical state, debug issues, or populate new services.
* Additive Schema Evolution: Since events are stored as JSON or similar flexible formats, schemas can typically be evolved in an additive manner. New fields or event types can be introduced without breaking legacy consumers, provided downstream processes are designed to handle unknown fields gracefully.
* Decoupled Systems and Powerful Integrations: Events can be consumed by multiple independent services, enabling sophisticated workflows, real-time analytics, and seamless data movement across boundaries.

**Considerations**

* Auditability in Practice: While RDBMS history tables and versioning mechanisms allow for a well-understood pattern of state auditing—tracking who changed what, and when—event-based systems theoretically offer even finer-grained insights. However, the why of a change (the intent or business context) is often only partially captured, unless explicitly included in the event payload. Notably, auditability of events themselves (e.g., “who appended this event, and under what authenticated context?”) can be tricky, and is sometimes overlooked.
* Purpose of Tracking: Event vs Change: Practitioners should ask: “Do we need to track the occurrence of every event, or just the net effect on state?” If only the most recent state matters, a traditional system may be more efficient and easier to maintain. If understanding the sequence of actions is critical—to diagnose issues, provide regulatory traceability, or enable time-travel analytics—event stores are more suitable.
* Long-Term Schema Management: Storing heterogeneous events and arbitrary payloads in a single JSON column can be convenient in the short term, but over years, this lack of schema creates headaches. Without careful governance, event formats may diverge, legacy consumers may break, and maintaining backward compatibility becomes increasingly complex.
* Performance and Complexity: Event storage grows monotonically, resulting in far greater storage requirements than systems that only persist current state. Replaying events to reconstruct state can be slow, especially for high-velocity event streams, requiring either aggressive snapshotting strategies or sophisticated caching layers. Exporting this data to analytics pipelines, data warehouses, or reporting services can be challenging, as such systems typically expect denormalized, flattened state rather than streams of events.
* Non-unique value proposition: It’s essential to note that event stores are not the only path to robust auditing—traditional banks, payroll platforms, and similar systems have relied on relational databases for over half a century, consistently delivering reliable audit capabilities. Before embracing event sourcing, teams should carefully weigh whether the unique attributes it offers justify departing from the proven stability, established practices, and trusted reputation of conventional systems.

**Disadvantages**

* Storage Overhead: Mirroring the drawbacks of document stores, event stores require significantly more disk space: every mutation, no matter how minor, is preserved forever unless archival or pruning strategies are implemented.
* Challenging Data Export and Analytics: Since most analytics and reporting tools are designed around the concept of current state, data from event stores must be transformed—flattened and aggregated—into a relational form before export. This can entail substantial development and operational effort, especially for complex domains.
* Purging: purging is problematic as without a modification and insertion of a current state record as a new starting point, removal of events changes the final calculated state.
* Maintainability Concerns: The absence of enforced schema (or schema-on-read patterns) can lead to a proliferation of event types and structures, making it difficult to reason about the system holistically over time. Team discipline and robust documentation (both external event architecture guidelines and internal standards) are essential.
* Operational Complexity: Replaying all events to reconstruct current state is fundamentally slower and more complex than accessing a single, up-to-date record. This can impact user experience, system reliability, and recovery times, and degrade over time such that ongoing reinvestments in infrastructure are required to compensate.

# When to Consider Use Event Store Architectures

Event-based architectures certainly offer tangible value, particularly in terms of auditability. However, it’s important to recognize that audit trails—while different in approach—are not an exclusive advantage of event stores. Mature relational database systems have long enabled robust auditing capabilities, using history and versioning tables to track changes reliably.

When evaluating whether to record events versus state updates, it is our recommendation to adopt a hybrid approach if the audit requirements truly warrant it. Rather than fully committing your entire system to event sourcing, selectively record only those essential business events that genuinely require tracking. Avoid burdening the system with exhaustive event capture, as replaying state changes can be computationally intensive and time-consuming; such operations should be isolated to the smallest possible segment of your system’s resources.

The optimal strategy is to leverage a relational database management system (RDBMS) as the backbone for core system operations, where metadata and reference data reside. Use event storage as a purpose-built repository for those serialised events whose tracking is business-critical—much like how blob storage is reserved for serialised meta streams. This mix-and-match methodology ensures that each component plays to its strengths: RDBMS for structured, performant state management, and event stores for high-fidelity, specialised event capture. In summary, judiciously combine these technologies, implementing event storage only where its benefits clearly outweigh the costs.

In conclusion, event stores unlock remarkable power for auditability, replayability, and decoupling—but come with significant complexity, storage demands, and operational overhead. They are best adopted knowingly, with an appreciation for the trade-offs, a solid understanding of both external best practices and internal needs, and a clear view of your system’s long-term responsibilities. If you don’t need the full fidelity of event tracking, consider simpler alternatives. But for those rare domains where “what happened, when, why and what if?” is paramount, event stores are without equal as a specialised data store.

Appendix C – Future Trends

Offline storage, once a niche concern, is regaining relevance as applications increasingly demand reliability and rich offline experiences. The resurgence is most palpable in environments where service agents or browsers are responsible for storing data locally on the client. The technologies enable everything from offline-first progressive web apps to robust sync solutions and resilient mobile applications. Let us explore the landscape—highlighting what’s available in browsers, the role of Sqlite, and the principal options to consider.

# Core Browser-Based Storage Options

Three main technologies frequently come up in discussions of browser-based offline storage:

* LocalStorage: The classic key-value store provided by the Web Storage API. LocalStorage offers simple synchronous storage of strings, persistent across sessions. It is best for small amounts of data (generally up to 5MB), but lacks sophisticated querying or indexing capabilities.
* IndexedDB: The preferred option for complex offline scenarios. IndexedDB is a transactional, asynchronous, NoSQL-like embedded database. It allows you to store and retrieve objects, supports indexes, and is well-suited for larger datasets and structured storage. Most modern browsers support IndexedDB, making it the backbone for many advanced offline applications.
* Cache Storage: Used primarily by service workers, Cache Storage enables applications to store HTTP responses and resources for offline access and efficient repeated fetches. This is essential for building Progressive Web Apps (PWAs) and is accessible via the Cache API.

Other notable browser storage mechanisms include:

* SessionStorage: Similar to LocalStorage, but scoped to a browser tab/session—data is lost when the tab is closed.
* Cookies: Historically used for client-side persistence, but now mostly reserved for lightweight state and authentication tokens due to their size and security limitations.
* Service Worker Storage: Service workers themselves can leverage Cache Storage and IndexedDB for offline operations, but do not provide a unique storage API.

Regarding Sqlite: conventional browsers do not natively support Sqlite as a standalone, directly-accessible storage engine. However, there are important nuances:

* WebAssembly (WASM): Projects such as sql.js and related wrappers have succeeded in compiling Sqlite to WebAssembly, making it possible to run a Sqlite database entirely in-memory within the browser. The database can persist data using local browser storage, such as IndexedDB, as a backing store. This approach is increasingly popular for sophisticated client-side applications that require SQL capabilities, but it is not a built-in browser feature.
* Mobile Applications: Sqlite is ubiquitous in mobile environments. Both Android and iOS provide robust support for Sqlite as an embedded database, which serves as the foundation for many apps’ offline and sync features. Frameworks like React Native and Flutter expose Sqlite to developers through plugins and native APIs.

Table 2: Offline Storage Options by Platform

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Technology | Browser Support | WASM Support | Mobile Support | Notes |
| LocalStorage | Yes | Indirect (via JS API) | No | String key-value; 5MB typical limit |
| SessionStorage | Yes | Indirect (via JS API) | No | Tab/session-scoped; ephemeral |
| IndexedDB | Yes | Indirect (can be backing store for WASM) | No | Object store; structured, transactional |
| Cache Storage | Yes (via Service Workers) | No | No | Caches HTTP responses/resources |
| Cookies | Yes | No | Limited | Small, plain-text; mainly for auth/state |
| Sqlite | No (browser-native) | Yes (via WASM, e.g. sql.js) | Yes (native) | Full SQL DB; persistent and robust |

The revival of offline storage in browser-based environments is driven by the expectation of seamless offline capability, cross-device sync, and resilient architectures. While browsers do not natively offer Sqlite, the WASM approach increasingly fills this gap for advanced use cases. For most web applications, however, IndexedDB remains the workhorse for local structured data.

In summary, when architecting for offline capability, consider the strengths and constraints of each technology. IndexedDB, LocalStorage, and Cache Storage form the backbone of browser-based offline storage, while Sqlite—though absent natively from browsers—is a mainstay in mobile and, increasingly, WASM-powered apps for web. The choice depends on your requirements for structured queries, capacity, and portability—there is no one-size-fits-all solution, but a vibrant ecosystem to support every ambition.

**Note:** there is growing momentum that WASM take the place of service side containers, and becoming more universal across the three key spheres (servers, browsers, mobile).

1. Edgar F. Codd’s “A relational model for Data for Large Shared Data Banks” was published in 1970. [↑](#footnote-ref-2)
2. SQL was invented in the mid 70’s and Oracle released Oracle in 1979. [↑](#footnote-ref-3)