DRAFT - ICT Project Guidance

Data Schema Design

Version:

0.2

Author:

Sky Sigal, Solution Architect

## Purpose

This document provides guidance for considerations to be made when developing a data schema whether it be for storage only or an API.

## Synopsis

The development of a data schema involves an understanding of schemas, entitities, attribtues

## Contents

[Purpose 1](#_Toc184802468)

[Synopsis 1](#_Toc184802469)

[Contents 2](#_Toc184802470)

[Context 4](#_Toc184802471)

[Schema Design Recommendations 5](#_Toc184802472)

[Schemas: mapping the problem space 5](#_Toc184802473)

[The Importance of Boundary Mapping 5](#_Toc184802474)

[Technical Boundary 5](#_Toc184802475)

[Techniques for Identifying Sub-domains 6](#_Toc184802476)

[The Broader Context: Beyond Today’s Problem 6](#_Toc184802477)

[From Subdomains to Schemas 6](#_Toc184802478)

[Conclusion 7](#_Toc184802479)

[Entities: correctly modelling the problem space 7](#_Toc184802480)

[Types 8](#_Toc184802481)

[Entities and Relationships 9](#_Toc184802482)

[Foreign Key based Relationships 10](#_Toc184802483)

[Attributes: including necessary information 11](#_Toc184802484)

[Unshared Information 11](#_Toc184802485)

[Primary Keys 12](#_Toc184802486)

[Deletion, State, and Archiving: A Modern Approach 12](#_Toc184802487)

[Deletion 12](#_Toc184802488)

[State 13](#_Toc184802489)

[Archiving 13](#_Toc184802490)

[Versioning 14](#_Toc184802491)

[Avoid oversharing State Flags 15](#_Toc184802492)

[Database Use 17](#_Toc184802493)

[Caching: An Integral Part of Database Design 17](#_Toc184802494)

[SQL Injection: A Persistent Threat 18](#_Toc184802495)

[Defer Saving 18](#_Toc184802496)

[ORMs 19](#_Toc184802497)

[Appendices 21](#_Toc184802498)

[Appendix A - Document Information 21](#_Toc184802499)

[Versions 21](#_Toc184802500)

[Images 21](#_Toc184802501)

[Tables 21](#_Toc184802502)

[References 21](#_Toc184802503)

[Review Distribution 21](#_Toc184802504)

[Audience 21](#_Toc184802505)

[Structure 21](#_Toc184802506)

[Diagrams 22](#_Toc184802507)

[Standards 22](#_Toc184802508)

[Acronyms 22](#_Toc184802509)

[Terms 22](#_Toc184802510)

## Context

…

# Schema Design Recommendations

## Domains: mapping the problem space

In system and schema design, a **namespace** is a logical container that provides a structured way to group and distinguish identifiers, such as object names, attributes, or types. It ensures uniqueness within a defined scope, preventing naming conflicts when multiple entities coexist in the same system. This concept is foundational for organising complex systems and breaking down problems into manageable and meaningful domains. At its core, the use of namespaces—or their equivalents—represents a method for delineating problem boundaries. By explicitly defining domains, systems gain clarity about their functional and business areas, fostering better alignment between the system’s structure and its purpose.

This deliberate segmentation mirrors the precision of cartography, mapping distinct aspects of the problem space into well-defined regions. These problem boundaries provide a framework for modular thinking, enabling the reuse of established patterns and reducing the need for novel, ad hoc solutions that introduce risk and inefficiency.

Too often domain mapping starts with and then remains within a single namespace to manage all components. While this may appear convenient at first, it quickly reveals a poor understanding of the problem space and results in foundational disorganisation. A single namespace conflates unrelated concerns, creating a monolithic structure where distinctions blur, making the system harder to maintain, scale, or secure. Avoiding this “big ball of mud” trap is critical for ensuring that the design reflects the system’s complexity and supports its long-term adaptability.

The concept of namespaces, while abstract, manifests across various disciplines and tools. In programming and processing languages, such as XML, SOAP, Kubernetes, or Java, namespaces provide structure by organising related components and separating concerns. Similarly, modules, packages, or schemas offer analogous functionality by enabling systems to evolve without unintended overlap or conflicts. This universality highlights the value of mapping problem spaces into distinct areas that are federated yet autonomous, much like nations working collaboratively while preserving their individual identities.

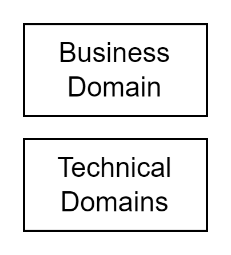


Figure 1: Consider the boundary between the Business and Technical Domains

When determining these boundaries, it is useful to distinguish between technical and business domains.

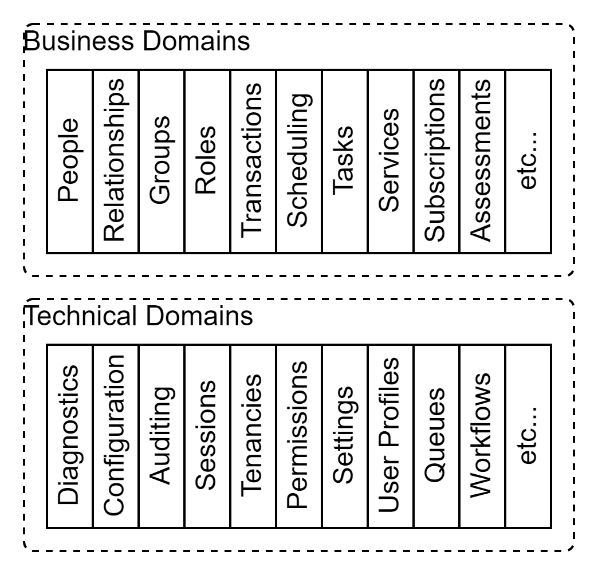


Figure 2: Example Technical & Business Subdomains

Technical domains address system-level concerns that are common across many systems, such as diagnostics for logging and monitoring, audit trails, fixed settings, and user preferences. These concerns are independent of any specific industry or business purpose and can often be abstracted into reusable solutions.

Business domains focus on the specific needs of the organisation or sector the system serves. These domains might involve managing entities like people, their relationships, resources they use, schedules they follow, and assessments they undergo. Careful consideration of these categories Designing subdomains within these overarching domains requires both insight and discipline. One effective technique is stacked abstraction, which involves identifying shared patterns across systems to define generalised subdomains. For example, systems frequently manage interactions with people—whether customers, employees, or service providers—and track related events, transactions, or assessments. Recognising these commonalities reduces the need for bespoke designs while aligning the system with established solutions.

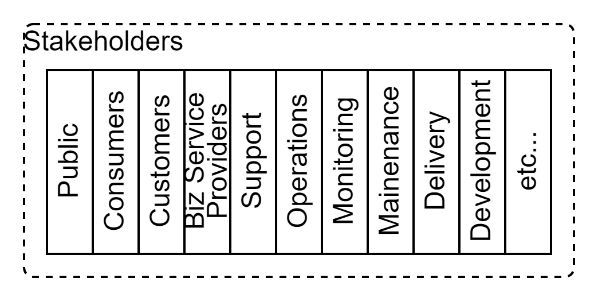


Figure 3: Stakeholders

An optional step is to check any elicited domains against a list of stakeholder roles and responsibilities, which often correspond to distinct operational areas within the system. This alignment not only improves usability and maintainability but also strengthens security by clearly delineating access based on roles. ensures that the system design is clear, organised, and extensible.

A robust domain based design also considers the broader ecosystem in which the system operates, including adjacent or complementary domains that may intersect with its purpose. For instance, an educational system might not only address immediate needs, such as course scheduling or assessment tracking, but also account for its integration with other phases of education—such as Early Childhood Education before it, vocational or lifelong learning after it, and parallel domains like health and equity services. Anticipating these relationships allows the system to accommodate future integration needs and avoids embedding design debt that could constrain growth and adaptability.

In relational databases, the concept of namespaces is realised through schemas. A schema is not merely a tool for grouping tables, views, and functions; it serves as a means of representing and respecting the system’s problem domains. By structuring data and relationships into distinct schemas, databases gain modularity, security, and clarity. For instance, separating technical domains like diagnostics and settings from business domains like people and schedules ensures the database reflects the system’s problem space with precision. While some systems may default to a single schema, such as dbo in SQL Server, this approach mirrors the same challenges of a single namespace in other contexts, creating a "ball of mud" that impedes adaptability and scalability. Instead, thoughtfully designed schemas align with the broader conceptual framework of namespaces, ensuring robustness and flexibility.

Schemas, whether in databases or conceptualised more broadly as namespaces, represent the cartography of the problem space. They guide system design by defining boundaries that clarify relationships, modularise functionality, and align the system with both immediate requirements and the broader organisational environment.

## Entities: correctly modelling the problem space

While there are many diagramming conventions to choose from Entity-Relation (ER) model diagrams remain a widely used tool for visualizing database structures, offering annotations for entities, attributes, keys, and relationships.

### Types

ER model diagrams, however, while they can be helpful in documenting a schema, their value for guiding schema development is often limited from lacking the nuance required to shape thinking around the *types* of entities necessary for a well-structured schema, particularly in complex systems.

A robust schema design requires careful differentiation between various types of entities, which ER diagrams often fail to visually differentiate, thereby not adding to the design process. Yet these distinctions would help ensure clarity, maintainability, and alignment with the system’s purpose.

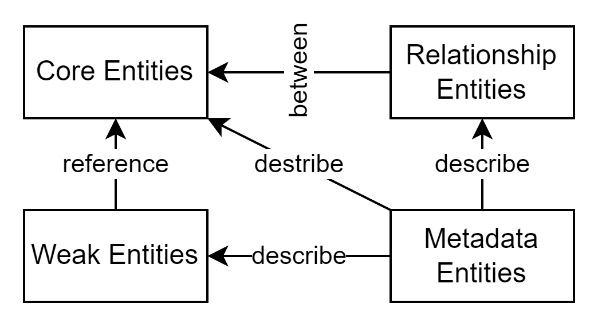


Figure 4: Entity Types

Consider understanding the domains concepts as one of the following:

* **Core Entities (Root Entities)**: represent the fundamental core objects of a domain. Core entities have their own identity (a Primary Key (“PK”)) and are the primary subjects of relationships. In Domain Driven Design (DDD), they are analogous to “Root entities”. Examples might include Product, Customer, or Order.
* **Weak Entities (Value Objects)**:  
  Often dependent on core entities, weak entities provide supplementary detail to a root/core entity (via a Foreign Key (“FK”) reference) but lack an independent identity (a “Primary Key”) themselves. In domain-driven design, they are analogous to “value objects”. Examples might include Address or Measurement, which derive their context and meaning from a related core entity.
* **Relation Entities (Join Tables)**:  
  These entities model many-to-many relationships or complex associations between core entities. For instance, an Enrollment entity connecting Student and Course entities can include additional attributes such as enrollment date or status, elevating it beyond a simple join table.
* **Descriptor/Metadata Entities for Objects**:  
  These entities provide additional descriptive or contextual information about core entities. Examples include Tag, Category, or Attribute, which can classify or detail a core entity without modifying its structure.
* **Descriptor/Metadata Entities for Relationships**:  
  Similar to descriptors for objects, these entities describe or add context to relationships. For example, a ConnectionType entity might describe the nature of the relationship between two people in a SocialNetwork schema, such as "friend" or "colleague."

### Entities and Relationships

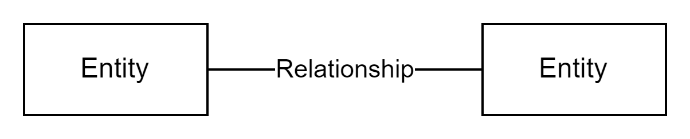


Figure 5: Entity Constraint based Relationship

Entity Relationship (ER) models simplify the process of associating two entities by representing their relationship as a simple line. While simple and easy to understand, this simplicity often hides the complexity required to accurately model real-world systems. Relationships between entities are rarely static nor permanent; they typically include temporal properties, such as start and end dates, or status indicators, like whether the relationship is currently active.

When relationships are modelled merely as constraints (e.g., a primary key–foreign key relationship) without a dedicated place for this metadata, developers frequently make the error of attaching these properties to one or other of the entities themselves.

For example, marking a Person as "Enabled" or "Disabled" conflates metadata about the relationship with properties of the entity, violating the principle that permanent entities should not carry contextual or temporal metadata. This misstep compromises the clarity and integrity of the model.

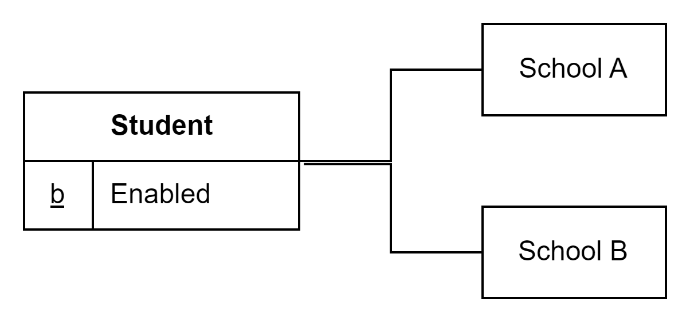


Figure 6: Embedding metadata in an entity (Ex. 1)

As an example, embedding the state within the Person makes it impossible to disable a student from one school without also disabling them from all schools.

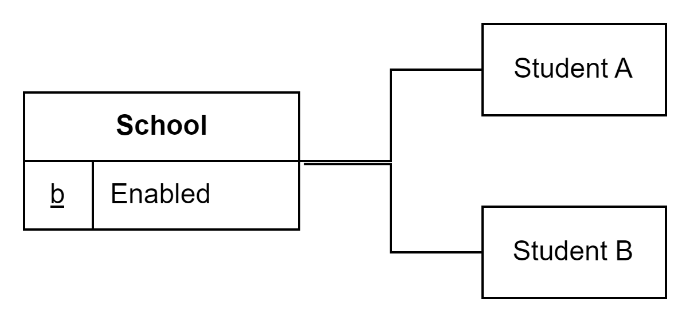


Figure 7: Embedding metadata in an entity (ex. 2)

Vice versa, embedding the state with a School makes it impossible to disable a single student without disabling all students and teachers.

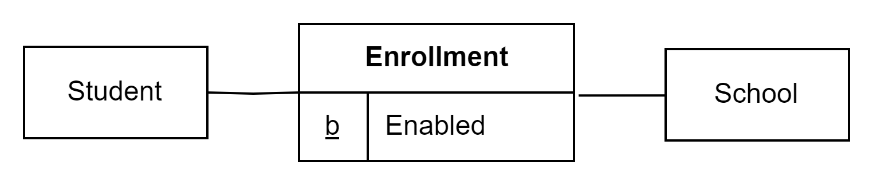


Figure 8: Embedding metadata in a relationship

Understanding the first-class importance of relationships – and avoiding relying on only the simpler constraint-based relationships -- is closely tied to recognizing the distinction between **permanent** and **contextual** entities.

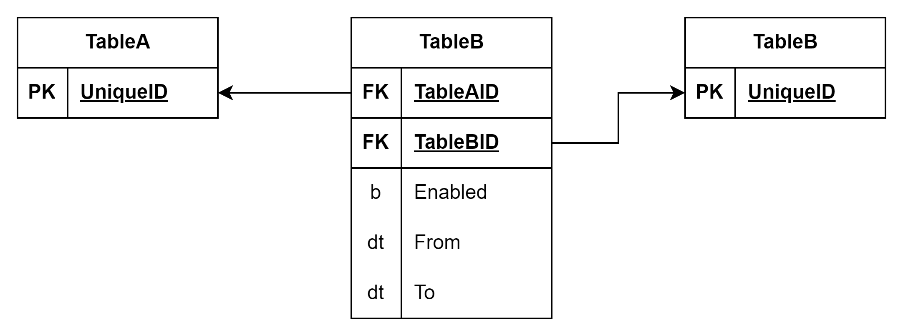


Figure 9: Relationship Object

Roles such as Student or Customer are typically contextual—they exist in relation to another entity, such as a specific School or Store. Rather than modelling these roles as standalone entities, they should be replaced with dedicated relationship objects, such as StudentEnrollment or CustomerOf, which represent the contextual connection between a new permanent object, like Person, and the associated School or Store.

Making this distinction and removing the misclassification of contextual entities enables richer, more accurate modelling of real-world complexity while maintaining flexibility, consistency, and scalability. For instance, the schema can now support a Person who is simultaneously a Customer of multiple stores, a Parent to another person, and a Student at a school. By prioritizing relationship objects and adhering to a clear differentiation between permanent and contextual entities, the system achieves a robust, extensible foundation capable of accommodating complex and evolving requirements.

### Foreign Key based Relationships

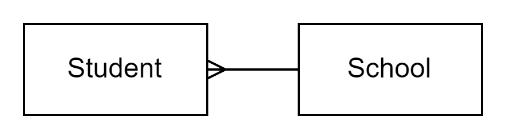


Figure 10: Example of a simplistic constraint based Relationship

While relationships can be established in their simplest form as a foreign key constraint pointing to a remote entities surrogate key as a primary key, this approach is generally not recommended except in the most straightforward or naïve circumstances.

However, a relationship between entities is rarely a simple, unadorned connection; it almost always carries additional context or metadata that makes the relationship meaningful. For this reason, modelling relationships with dedicated join objects (also known as join tables or relationship entities) is a better practice.

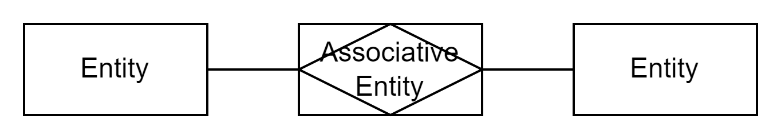


Figure 11: Associative Relationship Entity

Consider the example of associating multiple phone numbers with a person. A simple foreign key relationship between a Person entity and a PhoneNumber entity might seem sufficient at first glance. However, this model quickly fails to capture necessary context. For instance, the purpose of a phone number—such as "Home" or "Business"—is not intrinsic to the phone number itself but is instead metadata describing the relationship between the person and the number. Additionally, a phone number might be "Home" for one person and "Work" for another, further complicating a simplistic foreign key approach. This ambiguity highlights the need for a join object, such as PersonPhoneNumber, which can include attributes like purpose or type.

A similar issue arises in other common scenarios. For example, in a system where people belong to groups, it might be tempting to model the relationship with a simple foreign key. However, this fails to account for critical metadata such as the start and end dates of the membership, the role of the person in the group, or their status within the group. A dedicated join object like GroupMembership not only represents the connection but also provides a proper place to store these contextual details.

Foreign key constraints remain a valuable tool for enforcing referential integrity and should be applied within join objects to link them to the related primary entities. At the same time, these join objects provide the correct location for metadata about relationships, ensuring that contextual data is not misplaced within the entities themselves. By adopting this approach, schemas become more flexible and scalable, as relationships can accommodate complex requirements—such as multiple roles, states, or time-based attributes—without compromising clarity or maintainability. Modelling relationships in this way allows for a more accurate representation of real-world complexities and positions the system to evolve seamlessly with future needs.

## Attributes: including necessary information

### Unshared Information

As mentioned earlier, best practice in database and API schema design discourages embedding metadata directly into an entity, particularly when the entity is exposed as a Data Transfer Object (DTO) over an API.

The reasoning behind this guidance is to maintain a clear separation between the core properties of the entity and the additional contextual or operational information (metadata) that might change based on the entity's usage or relationship to other systems.

However, there are practical limitations to this approach, and exceptions are sometimes necessary to balance maintainability, usability, and performance.

For instance, the datastore's unique identifier for an entity (commonly referred to as its ID) is typically exempted from this rule. Including the ID in an DTO entity is often necessary for the API consumer to perform operations such as updates, deletes, or links to other entities, even though the ID is technically metadata.

Similarly, single-value classification (as opposed to categorization) can sometimes be included in DTOs when doing so simplifies API design and reduces complexity for consumers.

Conversely, categorization, which often involves referencing multiple values or broader groupings, should typically point to distinct entities in their own right, rather than being embedded directly within a DTO.

The inclusion of unshared metadata information in DTOs always requires careful consideration. For example, while including operational metadata like FromDateTime or ToDateTime risks "polluting" the primary entity's representation. Instead, such metadata is better exposed through separate mechanisms, such as shadow APIs or auxiliary endpoints, to keep the main API focused on core business data.

### Primary Keys

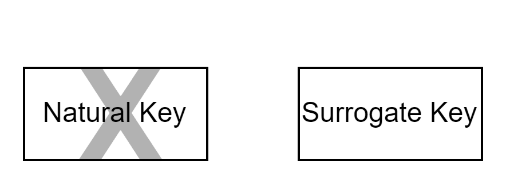


Figure 12: Primary Key Options

The choice of data storage primary keys plays a critical role in database design, impacting performance, maintainability, and scalability. A well-chosen primary key ensures efficient data access and preserves the integrity of relationships between tables.

Natural keys, while intuitive, should be avoided. They are tied to business rules that may evolve over time, introducing the risk of cascading updates and broken relationships. Additionally, their logic depends on stored procs that introduce for every insert both a significant performance costs and an ensuing longer table lock duration, affecting performance.

Similarly, while a common feature of most database management system (DBMS), surrogate primary keys generated by the DBMS, such as integers or long integers, introduce performance bottlenecks. These keys often require sequential locking, which degrades performance in high-concurrency environments.

DBMS generated UUIDs -- particularly v4 -- also present challenges due to their random nature combined with poor maintenance processes[[1]](#footnote-2).

### Deletion, State, and Archiving: A Modern Approach

#### Deletion

Figure 13: Logical State Attributes

Avoid deletion of records wherever possible—ideally, never delete them. Data should be treated as immutable, preserving every version of a record to maintain a consistent history. Space is cheap. This approach allows users to make changes without fear of permanent loss, enabling them to undo their actions if necessary. Importantly, "undo" should not mean deleting the most recent record but creating a new record based on the values of the prior version. This ensures that the system maintains a complete and accurate record of all changes and undos, which is crucial for auditing, troubleshooting, and maintaining trust in the data's integrity.

#### State

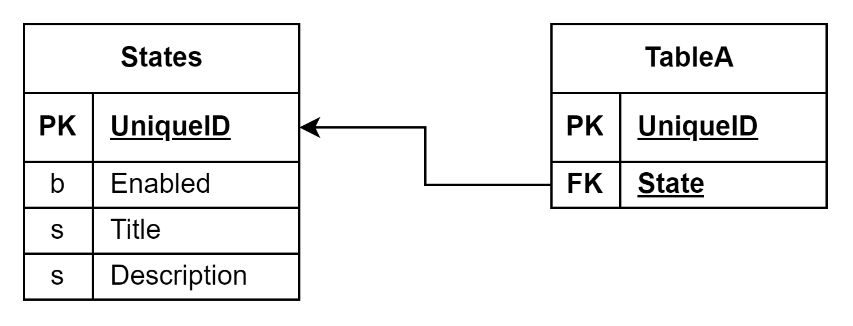


Figure 14: Logical State Flags

Deletion should be conceptualized as a logical state rather than an irreversible action. Logical states, such as "Draft," "ForReview," "Rejected," "Released," "Merged," "Removed," "Restored," and "Retired," allow for flexible workflows and maintain traceability. These states can be tailored to the system’s requirements, skipping irrelevant ones in simpler workflows.

One particularly interesting state worth considering is "Merged," which enables two records to be combined into a new one. Each merged record retains a reference to the new record it was merged into, preserving the historical trail. This approach even allows for "Unmerging," where a previously merged record can be disentangled and restored to its original state. By managing deletion and modification as logical states, systems become more resilient and adaptable to user needs.

#### Archiving

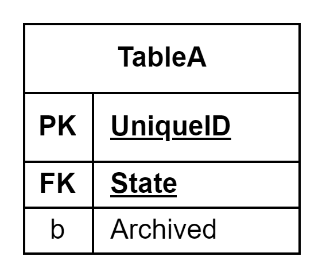


Figure 15: Archive Flag

The concept of archiving stems from an era when storage was prohibitively expensive, necessitating the segregation of rarely accessed data. However, modern storage costs have rendered this driver obsolete for most systems. The perceived need for archiving often arises in discussions about managing database size and adhering to data retention policies. While extremely large databases can still impact backup and restoration *times*, this is a rare concern for most applications and should not drive design decisions prematurely.

Moving data to secondary storage makes it undiscoverable and in accessible without a secondary system that doesn’t exist, and restoration of legacy data into an operational system so fraught as to it begs the question as to consider very carefully what use case the data is being kept.

A final reason provided is the reduction of query times. This is usually not a valid reason, as it is distracting from the work that does need doing – improving data schema design and indexing.

Additionally, moving sensitive data to separate storage mechanisms does not reduce overall storage requirements; it just doubles the risk by placing sensitive data in a less monitored and potentially less secure environment. Keeping all data within the primary system ensures centralized oversight and reduces vulnerabilities.

Finally, removal of sensitive data should not focus on archiving but instead focus on **de-personalizing** information rather than relocating it.

### Versioning

As mentioned earlier, logical deletion, as opposed to physical deletion, serves as the primary mechanism to enable data rollback after user or system errors. This approach ensures that historical data is preserved, allowing systems to list sets or access individual instances of previous record versions, making them accessible as needed, including through APIs.

Using an incrementing version number to track record versions adds limited value while introducing potential performance challenges. Incrementing numbers require a mechanism for maintaining sequential order, which can lead to table locking and contention, especially in high-concurrency environments.

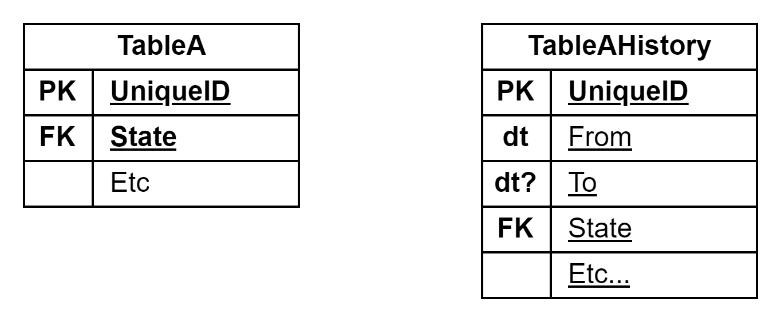


Figure 16: Temporal Tables

Modern databases, such as SQL Server (introduced in 2016), provide a more robust solution in the form of temporal history tables. These tables use From and To date columns to track the validity period of records. Unlike traditional versioning, temporal tables do not immediately store data in the history table upon record insertion. Instead, historical data is added to the temporal table only when a record is updated or physically deleted, with the main table retaining the current active record.

While temporal tables simplify version tracking and reduce implementation effort, they can grow substantially if records are frequently updated. This growth requires careful management to maintain performance. Strategies such as logical flag archiving, combined with composite indexing, can ensure efficient querying while managing data volume.

A practical challenge of versioning is making historical records accessible via APIs. Directly exposing temporal data, including From and To metadata, in primary integration modules can complicate their design and reduce usability. A recommended approach is to implement a "shadow API" specifically for interacting with temporal history tables. This API focuses exclusively on serving historical data, preserving the primary API for current records. By abstracting temporal data through a dedicated API, the system maintains a clean separation of concerns and provides simplified, context-specific interfaces for downstream consumers.

For database systems that do not natively support temporal tables, the same functionality can be achieved manually by including explicit FromDateTimeUTC and nullable ToDateTimeUTC fields in the schema. In this model, the ToDateTimeUTC field is populated only when a new version of a record is created, with the latest record retaining a null value in this column. To ensure data integrity and efficient querying, a composite unique index on ID, FromDateTimeUTC, and ToDateTimeUTC is essential.

Both approaches—temporal tables and custom From/To date tracking—are valid and effective strategies, with the choice depending on the specific requirements of the system and the capabilities of the database being used. In either case, success depends on careful attention to indexing, managing data growth, and designing APIs that effectively abstract versioning logic while maintaining performance and usability.

The critical principle is to handle historical data through a separate API, ensuring the main API remains streamlined and focused on current operations, while the shadow API provides full access to historical versions without overcomplicating the primary interface. This separation facilitates cleaner integration and more maintainable system design.

### Avoid oversharing State Flags

Best practice dictates avoiding the exposure of state flags and datetime metadata through APIs unless absolutely necessary. These elements are typically system-level metadata, designed to assist with internal data storage and operations. They are not intrinsic to the entity's purpose or meaning and are often irrelevant—or even confusing—to external consumers.

For example, state flags such as IsDeleted, IsArchived, or IsEnabled might be essential for managing the lifecycle of entities within the system. Similarly, fields like CreatedDate, UpdatedDate, or LastAccessedDate are useful for internal auditing, synchronization, or debugging. However, exposing these details in an API risks overcomplicating the interface, leaking implementation details, or inadvertently creating dependencies that constrain future design changes.

By refraining from sharing these metadata elements directly, APIs maintain a clean abstraction that focuses on the entity's core business value. If metadata is required by specific consumers, consider creating dedicated endpoints or metadata-specific APIs to provide this information in a controlled and purpose-driven manner.

Carefully managing the exposure of state flags and metadata ensures APIs remain concise, relevant, and adaptable to evolving system requirements, while minimizing the risk of unnecessary complexity for external consumers.

## Commands

Another form of entity is those of commands that change the state of other entities or their relationships.

Unlike entity-specific attributes that describe an entity or relationship itself, attributes that describe the state of them —effectively metadata—should not be directly manipulated by modifying properties.

For instance, flipping a "deleted" flag directly can lead to inconsistent or untraceable changes. Instead, state changes should be executed through explicit messages or commands, such as a "DeleteStudent" command, which encapsulate the intent behind the change. This approach ensures that state changes are logged, auditable, and validated against business rules before being applied. It decouples the intent from the implementation, enabling consistent handling of side effects like notifications or cascading updates. By treating state changes as deliberate, logged actions, systems gain greater traceability, maintainability, and alignment with event-driven or domain-centric design principles.

The same applies for making relationships between entities.

### Command Argument Packages

Related to Command entities, a final type of entity is the **argument package**, which groups all the arguments required by a command.

This approach – known as the **Command Pattern** -- improves auditing by ensuring that all inputs to a command are clearly defined and logged, enhances queueing by enabling commands to be processed asynchronously or in batches, and provides a foundation for implementing advanced capabilities, such as undo or rollback functionality.

By structuring arguments into dedicated packages, systems achieve greater modularity, traceability, and extensibility in their handling of commands.

# Database Use

The design of a system has to be considered not in isolation, but also how it will be used.

## Caching: An Integral Part of Database Design

Caching is often treated as a separate consideration from database design, but it shouldn’t be. Caching is integral to the problem of optimizing data access and delivery, and its implementation should be deeply intertwined with the database's role in the system. At its core, the goal of caching is simple: cache as much data as possible as close to the end user as possible, in the form they will consume it.

In its simplest implementation, caching begins at the end user’s device. Wherever feasible and without compromising security, data should be cached directly in the end user’s browser or client application. This eliminates unnecessary requests to the server, providing immediate responsiveness. Beyond the end user, the next layer of caching should be within the server’s memory. If there is a presentation tier, an in-memory cache server shared across that tier can further improve efficiency. When the data must be accessed by the application tier, a shared in-memory cache server can act as an intermediary, reducing the frequency of database queries. While an in-memory cache server resembles a database server in that it serves multiple requests, it doesn’t compute answers; it simply returns data associated with a given key, significantly reducing computational overhead.

The persistence duration of cached items depends on their nature and usage. Data that changes infrequently, such as system settings that require a server redeployment or reset to update, can be cached for longer periods, such as 20 minutes or more. Session-specific user settings, which remain consistent for the duration of a user session, might be cached for shorter periods, such as one minute. Data that changes over time but can be reused across multiple sessions can also be cached, provided it can be invalidated and refreshed reliably.

For optimal performance, data should be cached in forms ready for consumption. If the presentation layer requires data in a specific format—such as JSON-encoded fragments—caching it in that format reduces redundant encoding or processing steps. Similarly, caching multiple versions of the same data, such as translations in different languages, can enhance performance even if it results in duplicate entries in the cache. These trade-offs are minor compared to the benefits of faster response times and reduced server load.

Caching is not inherently complex, but it does require precision. Careful thought must be given to cache invalidation, expiration policies, and security considerations. The more data you can cache at appropriate levels, the less strain is placed on the server and database. This improves the system’s scalability and responsiveness, directly benefiting end users. In essence, caching isn’t just a convenience—it’s a necessity for modern systems that in turn puts requirements on the database design schema, in order to remove load from the database server.

## SQL Injection: A Persistent Threat

Year after year, SQL injection remains a top-10 security risk according to industry reports, such as those published by OWASP. Despite decades of awareness, the fundamental problem—and the associated risk—has not materially diminished in environments where developers write raw SQL. The issue stems from improperly sanitized inputs being incorporated into dynamically generated SQL queries, allowing attackers to manipulate the query to access or alter unauthorized data.

Over the past 30 years, despite improvements in tooling and education, the prevalence of SQL injection vulnerabilities underscores the challenge of consistent implementation of secure coding practices. The core issue persists: when developers handle SQL directly, the risk of introducing vulnerabilities remains high. Even experienced developers can inadvertently overlook edge cases or fail to sanitize inputs adequately, leaving the system exposed.

At a minimum, organizations must implement static code analysis or similar tools to ensure that SQL queries consistently use parameterized statements or variables, which prevent malicious inputs from altering the intended structure of a query. However, this approach alone places the burden of security on individual developers and is prone to human error.

The preferred solution, particularly in environments without deep SQL expertise, is to rely on Object-Relational Mappers (ORMs) for database interactions. ORMs abstract SQL generation and enforce the use of parameterized queries by default, effectively eliminating the risk of SQL injection for most use cases. Moreover, ORMs provide additional benefits, such as simplifying database operations, improving code maintainability, and enabling secure integration with the database.

By leveraging ORMs correctly, developers can mitigate the risks associated with SQL injection while streamlining their workflow. However, it’s essential to follow ORM best practices, such as avoiding raw SQL queries through the ORM unless absolutely necessary, as this can reintroduce vulnerabilities. In most scenarios, relying on well-established ORMs represents the most practical and secure approach to database management, allowing developers to focus on building functionality without compromising on security.

## Defer Saving

A common inefficiency in application development arises from saving changes to the database immediately after every operation. While this approach might seem straightforward, it introduces significant performance and consistency issues that can scale poorly in real-world applications.

The first issue is the sheer number of write operations. In a team environment, especially with modern applications handling concurrent user requests, excessive writes can lead to a dramatic increase in database calls. It is not uncommon for an application to make upwards of 40 calls to the database for a single page request, with a substantial portion being write operations. Each write not only consumes resources but may also lock tables or rows, degrading the responsiveness of the application for other users. This pattern creates a bottleneck, particularly under high concurrency scenarios.

The second issue relates to database consistency. When each operation saves immediately, there is no transactional context wrapping the series of changes. If an error occurs partway through, the database may be left in an incomplete or inconsistent state. Developers often attempt to solve this by wrapping all operations within a transaction. However, this can lead to long-running transactions, keeping locks open for extended periods and further impacting database performance.

Object-Relational Mappers (ORMs) offer a powerful solution to this challenge. By keeping track of changes to entities in memory, ORMs defer the actual database writes until a save operation is explicitly triggered. This reduces the number of writes and allows them to be bundled efficiently into a single operation, minimizing the impact on performance. Furthermore, since the ORM manages these changes, the application can maintain consistency by ensuring that all operations within a transaction are committed or rolled back as a unit.

For this pattern to be effective, it is critical to remove the ability for developers to arbitrarily trigger saves. Instead, the save operation should be centralized and controlled, typically by a request handler at the end of the request lifecycle. This approach ensures that all changes made during the request are handled within a single transaction, keeping it open for the shortest possible duration. If an error occurs, the transaction can be safely rolled back, leaving the database in a stable and predictable state.

By deferring saves until the end of a request, applications benefit from improved performance and greater consistency. This pattern not only leverages the strengths of ORMs but also promotes a cleaner and more robust architecture, reducing the likelihood of database issues and making the system more resilient to errors.

## ORMs

Object-Relational Mappers (ORMs) are invaluable tools for managing database interactions in modern software development. They abstract complex SQL generation, allowing developers to focus on application logic and minimizing the likelihood of security vulnerabilities. However, their adoption often encounters resistance from traditional operations-focused DBAs.

The scepticism from DBAs is not entirely without merit but is rooted in the nature of their role. DBAs are tasked with diagnosing and resolving database performance issues, and encountering ORM-generated SQL in logs can be frustrating. The queries are often opaque, lacking the clarity of human developed SQL. This is intentional: ORM-generated SQL is optimized for machine-to-machine interaction, not human readability. While DBAs may view the quality of ORM-generated SQL as suboptimal, this perspective is opinionated rather than factual. The SQL generated by an ORM is designed to fulfil the requirements of the data model it represents, and its quality is directly tied to the quality of the underlying model.

This distinction is crucial. The focus should not be on dissecting ORM-generated SQL but on optimizing the data model itself. Proper indexing, normalization, and careful relationship design ensure that the SQL generated by the ORM performs optimally as intended. When the model is well-structured, the SQL is inherently optimized. Neglecting this aspect can lead to performance bottlenecks, but this is a failing of the development of the model, not the ORM.

ORMs also bring significant benefits beyond abstraction. They enhance security by enforcing parameterized queries, reducing the risk of SQL injection and encoding errors. Developers, who often lack expertise in relational database design and set theory, are better equipped to work within an ORM's structured framework. The use of ORMs also streamlines development workflows.

Code-first – as opposed to Model-first or even DB-first - approaches are particularly effective, enabling developers to design models directly in code and generating schemas automatically. This alignment between application logic and database design simplifies migrations, version control, deployment processes while decreasing the introduction of hard-to-find errors.

In summary, while traditional DBAs may challenge the use of ORMs, their concerns can be addressed by focusing on model optimization and collaborative design. Properly implemented, ORMs provide secure, maintainable, and performant systems that meet the demands of modern development.

Whether using ORMs or not, following the same principles outlined in previous sections regarding the proper design and development of schemas, entities, primary keys, and relationships remain critical for expected optimal operations.

Appendices

Appendix A - Document Information

### Versions

* 1. Initial Draft

### Images

[Figure 1: TODO Image 2](#_Toc144995112)

### Tables

[Table 1: TODO Table 3](#_Toc145048484)

[Table 2: TODO Table 2 3](#_Toc145048485)

### References

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

### Review Distribution

The document was distributed for review as below:

|  |  |
| --- | --- |
| Identity | Notes |
|  |  |
|  |  |
|  |  |

### 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.

### Standards

ISO-25010

: …

ISO-25012

: …

ISO-25022

: …

### Acronyms

Refer to the project’s Glossary.

IT

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

##### ICT

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

### Terms

Appendix B – Namespace Design Principles

Prefer embedding metadata in a separate entity.

Prefer metadata referring to entities rather than the other way around.

Prefer developing relationship objects to constraint based relationships.

Avoid natural keys

1. [Black Arts Index Maintenance 1.2 - Guids vs. Fragmentation | Jeff Moden](https://www.youtube.com/watch?v=rvZwMNJxqVo) [↑](#footnote-ref-2)