The Data Model Architectural Style

A Comprehensive Reference for Information Structure Design

Contents

1 Overview

The data model style is an information structure architectural style that describes the structure of the data entities and their relationships within a system. Unlike component-and-connector views that show runtime behavior or module views that show code organization, data model views focus on the information that the system creates, stores, manipulates, and communicates.

The data model is fundamental to system architecture because data outlives the software that processes it. Applications are rewritten, technologies change, and user interfaces evolve, but the core data often persists and grows throughout an organization's lifetime. A well-designed data model enables systems to evolve, integrate, and scale while maintaining data integrity and supporting business needs.

Data modeling bridges the gap between business concepts and technical implementation. Business stakeholders think in terms of customers, orders, products, and transactions. Technical implementations require tables, documents, keys, and indexes. The data model provides a shared vocabulary and formal representation that both communities can understand and use to communicate.

1.1 Scope and Applicability

The data model style applies to any system that manages persistent or structured data. This includes transactional systems that create, update, and query business data; analytical systems that aggregate and analyze data for decision support; integration platforms that exchange data between systems; content management systems that organize and serve digital content; master data systems that maintain authoritative records across the enterprise; event-driven systems that capture and process streams of event data; and knowledge management systems that store and relate information for retrieval and reasoning.

The style is particularly valuable when data must persist beyond individual transactions or sessions, when multiple applications or components share the same data, when data integrity and consistency are critical, when data must be queried and reported in various ways, when data structures must evolve over time, and when compliance or audit requirements govern data management.

1.2 Historical Context

Data modeling has evolved through several generations of technology and methodology.

Hierarchical and network models dominated early database systems in the 1960s and 1970s. These models represented data as trees or graphs with explicit navigation paths.

The relational model, introduced by Edgar Codd in 1970, revolutionized data management by representing data as mathematical relations (tables) with declarative query languages. Relational databases became the dominant paradigm for decades.

Entity-relationship modeling, developed by Peter Chen in 1976, provided graphical notation for conceptual data modeling independent of implementation technology.

Object-oriented databases emerged in the 1980s and 1990s, attempting to bridge the gap between object-oriented programming and persistent storage.

NoSQL databases emerged in the 2000s to address scalability and flexibility requirements that relational databases struggled to meet. Document databases, key-value stores, column-family stores, and graph databases each optimize for different access patterns.

Modern data architecture often employs polyglot persistence, using different data models and storage technologies for different aspects of a system based on their specific requirements.

Understanding this evolution helps architects select appropriate data modeling approaches for their context.

1.3 Relationship to Other Styles

The data model style relates to several other architectural views and styles.

It supports the shared-data style by defining the structure of data in shared repositories.

It informs the module view by identifying data entities that modules must access and manage.

It connects to the deployment view by influencing how data is distributed and replicated.

It relates to domain-driven design by aligning data structures with bounded contexts and domain concepts.

It supports the service-oriented style by defining data contracts between services.

The data model is often documented alongside but separately from behavioral views. While component-and-connector views show how data flows through the system, the data model shows what that data looks like.

1.4 Conceptual, Logical, and Physical Models

Data modeling typically proceeds through three levels of abstraction.

Conceptual models represent business concepts and relationships at the highest level of abstraction. They use business terminology, focus on what data exists rather than how it is stored, and are independent of technology. Conceptual models facilitate communication with business stakeholders.

Logical models add detail to conceptual models, specifying attributes, keys, and precise relationship semantics. They remain independent of specific database technology but are detailed enough to drive implementation. Logical models are the primary focus of data architecture.

Physical models specify implementation details for a particular database technology. They include table definitions, column types, indexes, partitioning, and storage parameters. Physical models are derived from logical models with optimizations for specific technologies and workloads.

This document focuses primarily on logical data modeling, which is the level most relevant to software architecture.

2 Elements

The data model style comprises data entities as its primary elements, along with their attributes and properties.

2.1 Data Entities

A data entity is an object that holds information that needs to be stored or somehow represented in the system. Entities represent the things about which the system maintains data.

2.1.1 Types of Data Entities

Data entities can be categorized by their role and characteristics.

Core entities represent the fundamental business concepts around which the system is built. Examples include Customer, Product, Order, and Account. These entities are central to business operations and typically have complex relationships.

Reference entities provide lookup values and classifications used by other entities. Examples include Country, Currency, Status, and Category. Reference entities change infrequently and are often shared across systems.

Transaction entities capture business events and activities. Examples include Sale, Payment, Shipment, and Transfer. Transaction entities typically have timestamps and references to other entities involved in the transaction.

Association entities represent many-to-many relationships between other entities, often with additional attributes. Examples include OrderLine (linking Order and Product) and Enrollment (linking Student and Course).

Aggregate entities combine related information into a single unit for specific purposes. Examples include CustomerProfile (aggregating customer data from multiple sources) and OrderSummary (aggregating order details).

Temporal entities track changes over time, maintaining history of entity state. Examples include PriceHistory, AddressHistory, and EmploymentHistory.

2.1.2 Essential Properties of Data Entities

When documenting data entities, architects should capture several property categories.

Identity properties define how entities are identified. Entity name provides a clear, business-meaningful name for the entity. Primary key specifies the attribute or attributes that uniquely identify each instance. Natural keys are business-meaningful identifiers like Social Security Number or ISBN. Surrogate keys are system-generated identifiers like auto-increment integers or UUIDs.

Structural properties define the entity's data content. Data attributes list the individual data elements the entity contains. Attribute types specify the data type and constraints for each attribute. Required versus optional indicates which attributes must have values. Default values specify values used when no explicit value is provided.

Behavioral properties describe entity semantics. Business rules define constraints and behaviors specific to the entity. Validation rules specify conditions that data must satisfy. Derivation rules specify how derived attributes are calculated.

Access properties govern how the entity is used. Access permissions define rules to grant users permission to access the entity. Audit requirements specify what access and changes must be logged. Privacy classification indicates sensitivity and handling requirements.

Lifecycle properties describe entity evolution. Creation rules specify how and when entities are created. Modification rules specify how entities can be changed. Deletion rules specify how entities are removed or archived. Retention requirements specify how long entities must be kept.

2.2 Attributes

Attributes are the individual data elements that comprise entities. Each attribute has properties that define its structure and behavior.

2.2.1 Attribute Types

Attributes vary in their data types and characteristics.

Primitive attributes hold single values of basic types. String attributes hold text data with optional length constraints. Numeric attributes hold integers, decimals, or floating-point numbers. Date and time attributes hold temporal values with various precision. Boolean attributes hold true/false values. Binary attributes hold raw byte data.

Composite attributes combine multiple related values. Address might combine street, city, state, and postal code. Name might combine given name, family name, and title. These can be modeled as separate attributes or as structured types.

Multi-valued attributes hold collections of values. Phone numbers might allow multiple values per entity. Tags or categories might be lists. These may be modeled as separate related entities or as array types depending on the technology.

Derived attributes are calculated from other attributes. Age might be derived from birth date. Total might be derived from line item amounts. Derived attributes may be stored (denormalized) or calculated on demand.

2.2.2 Attribute Properties

Attributes have properties that constrain their values and behavior.

Data type specifies the kind of data the attribute holds.

Length or precision specifies size constraints for the value.

Nullability indicates whether the attribute can have no value.

Uniqueness indicates whether values must be unique across entities.

Default value specifies the value used when none is provided.

Constraints specify additional rules that values must satisfy.

Sensitivity classification indicates privacy or security handling.

2.3 Keys

Keys identify and locate entities within the data model.

2.3.1 Types of Keys

Different key types serve different purposes.

Primary keys uniquely identify each entity instance. Every entity must have a primary key. Primary keys should be immutable and never null. Simple primary keys use a single attribute. Composite primary keys combine multiple attributes.

Candidate keys are attributes or combinations that could serve as primary keys. An entity may have multiple candidate keys. One is chosen as the primary key; others remain as alternate keys.

Foreign keys reference the primary key of another entity, implementing relationships. Foreign key values must match existing primary key values (referential integrity). Foreign keys may allow null values for optional relationships.

Natural keys are meaningful business identifiers. Examples include email address, product SKU, or tax ID. Natural keys are human-readable but may change over time.

Surrogate keys are system-generated identifiers with no business meaning. Examples include auto-increment integers, UUIDs, or hash values. Surrogate keys are stable but require lookups for business identification.

2.3.2 Key Design Considerations

Key design significantly affects system behavior.

Stability requires that keys should not change after creation. Changes require updating all references, which is expensive and error-prone.

Simplicity favors single-attribute keys over composite keys. Simple keys are easier to use in queries and foreign key relationships.

Size affects performance. Smaller keys require less storage and faster comparisons. Integer keys are typically more efficient than string keys.

Meaningfulness trades off against stability. Natural keys are meaningful but may change. Surrogate keys are stable but meaningless.

3 Relations

Relations in the data model style define logical associations between data entities.

3.1 Cardinality Relations

Cardinality relations specify how many instances of one entity relate to instances of another entity.

3.1.1 One-to-One Relationships

One-to-one relationships associate exactly one instance of each entity.

One-to-one relationships often represent entity decomposition, where one entity's data is split across two tables for normalization or performance reasons.

They may represent optional extensions, where additional data exists only for some entities.

Implementation typically places the foreign key in either table or uses the same primary key in both tables.

Examples include Person to Passport (each person has at most one passport), Employee to EmployeeDetails (separating frequently and infrequently accessed data), and User to UserPreferences (separating core and optional data).

3.1.2 One-to-Many Relationships

One-to-many relationships associate one instance of the parent entity with multiple instances of the child entity.

This is the most common relationship type in data models.

Implementation places a foreign key in the child entity referencing the parent.

Examples include Customer to Order (one customer has many orders), Department to Employee (one department has many employees), and Category to Product (one category contains many products).

3.1.3 Many-to-Many Relationships

Many-to-many relationships associate multiple instances of each entity with multiple instances of the other.

Implementation requires an association (junction) table containing foreign keys to both entities.

The association table may have additional attributes describing each relationship instance.

Examples include Student to Course (students enroll in many courses; courses have many students), Product to Supplier (products come from many suppliers; suppliers provide many products), and Author to Book (authors write many books; books have many authors).

3.1.4 Relationship Properties

Relationships have properties beyond cardinality.

Participation indicates whether relationship participation is mandatory or optional. Total participation requires every entity instance to participate. Partial participation allows entity instances without relationships.

Identifying relationships contribute the parent's key to the child's primary key, creating a strong dependency.

Non-identifying relationships reference the parent but the child has its own independent key.

Cascading behavior specifies what happens when parent entities change. Cascade delete removes child entities when parent is deleted. Cascade update propagates key changes to foreign keys. Restrict prevents parent changes if children exist.

3.2 Generalization and Specialization

Generalization and specialization indicate an *is-a* relation between entities, representing inheritance hierarchies.

3.2.1 Semantics of Generalization

Generalization groups common attributes and relationships of multiple entity types into a supertype.

Specialization defines subtypes that inherit supertype attributes and add specialized attributes.

Subtypes may have exclusive or overlapping membership. Exclusive (disjoint) specialization means each instance belongs to at most one subtype. Overlapping specialization allows instances in multiple subtypes.

Subtypes may have total or partial coverage. Total coverage means every supertype instance must belong to some subtype. Partial coverage allows supertype instances without subtype membership.

3.2.2 Examples of Generalization

Person generalizes Employee and Customer, sharing name and contact information, with each subtype adding specialized attributes.

Account generalizes CheckingAccount, SavingsAccount, and InvestmentAccount, sharing balance and owner, with each subtype having specific features.

Vehicle generalizes Car, Truck, and Motorcycle, sharing registration and owner, with each having type-specific attributes.

3.2.3 Implementation Strategies

Several strategies implement generalization in databases.

Single table inheritance uses one table with all attributes of all types, with null values for inapplicable attributes and a discriminator column indicating the type. This is simple but wastes space and allows invalid nulls.

Class table inheritance uses separate tables for supertype and each subtype, joined by shared primary key. This is normalized but requires joins for complete data.

Concrete table inheritance uses separate tables for each subtype with duplicated supertype attributes. This avoids joins but duplicates structure and complicates polymorphic queries.

3.3 Aggregation

Aggregation turns a relationship into an aggregate entity, representing a whole-part relationship or treating the relationship itself as an entity.

3.3.1 Semantics of Aggregation

Aggregation represents stronger coupling than simple association. The aggregate owns its parts. Parts may not exist independently of the aggregate. Operations on the aggregate may cascade to parts.

Composition is a strong form of aggregation. Parts cannot exist without the whole. Deleting the whole deletes the parts. Parts belong to exactly one whole.

3.3.2 Examples of Aggregation

Order aggregates OrderLine. Order lines do not exist independently. Deleting an order deletes its lines.

Document aggregates Section. Sections belong to their document. The document lifecycle governs section lifecycle.

Organization aggregates Department. Departments exist within the organization structure.

3.3.3 Aggregation in Document Databases

Document databases naturally support aggregation by nesting documents.

Related data is embedded within the aggregate document.

Aggregates are retrieved and updated atomically.

Aggregate boundaries align with consistency boundaries.

4 Constraints

Constraints define rules that data must satisfy to maintain integrity and quality.

4.1 Integrity Constraints

Integrity constraints ensure data correctness.

Entity integrity requires that primary keys are never null and always unique. Every entity instance must be uniquely identifiable.

Referential integrity requires that foreign key values match existing primary key values or are null. References must point to valid entities.

Domain integrity requires that attribute values conform to their defined domains. Values must be of the correct type and satisfy domain constraints.

4.2 Normalization and Functional Dependencies

Functional dependencies should be avoided to prevent data anomalies.

4.2.1 Functional Dependencies

A functional dependency exists when the value of one attribute determines the value of another. If A determines B, then for any given value of A, there is exactly one corresponding value of B.

Problematic functional dependencies occur when non-key attributes depend on other non-key attributes, leading to update anomalies, insertion anomalies, and deletion anomalies.

4.2.2 Normal Forms

Normalization organizes data to reduce redundancy and dependency.

First Normal Form (1NF) requires atomic attribute values (no repeating groups or arrays) and unique rows identified by primary key.

Second Normal Form (2NF) requires 1NF plus no partial dependencies, meaning non-key attributes must depend on the entire primary key, not just part of it.

Third Normal Form (3NF) requires 2NF plus no transitive dependencies, meaning non-key attributes must depend only on the key, not on other non-key attributes.

Boyce-Codd Normal Form (BCNF) strengthens 3NF by requiring that every determinant is a candidate key.

Higher normal forms (4NF, 5NF) address multi-valued dependencies and join dependencies.

4.2.3 Denormalization

Controlled denormalization may be appropriate for performance.

Denormalization introduces redundancy intentionally to reduce joins and improve query performance.

Denormalized data must be kept synchronized, adding update complexity.

Common denormalization techniques include storing calculated values, duplicating foreign key attributes, and creating summary tables.

The decision to denormalize should be driven by measured performance requirements, not premature optimization.

4.3 Business Rule Constraints

Business rules impose domain-specific constraints.

Value constraints restrict attribute values. Salary must be positive. Status must be one of defined values. End date must be after start date.

Cardinality constraints restrict relationship counts. An order must have at least one line item. A project must have exactly one manager.

State constraints restrict valid entity states. An order cannot be shipped before it is paid. An account cannot be closed with a non-zero balance.

Cross-entity constraints span multiple entities. Total order value cannot exceed customer credit limit. Employee salary cannot exceed manager salary.

4.4 Temporal Constraints

Temporal constraints govern time-related data.

Valid time represents when facts are true in the real world.

Transaction time represents when facts are recorded in the database.

Bi-temporal models track both valid time and transaction time.

Temporal constraints may specify that time periods cannot overlap, that history cannot be modified, or that future-dated changes are allowed.

5 What the Style is For

The data model style supports several essential purposes in system architecture.

5.1 Describing Data Structure

The primary purpose is describing the structure of the data used in the system.

Entity identification captures what things the system needs to track. What are the core business concepts? What information must be stored?

Attribute specification captures what is known about each entity. What properties do we need to record? What are their types and constraints?

Relationship mapping captures how entities relate. How do customers relate to orders? How do products relate to categories?

This description provides a foundation for database design, application development, and system integration.

5.2 Impact Analysis and Extensibility

The data model enables impact analysis of changes to the data model and extensibility analysis.

Change impact assessment determines what is affected by proposed changes. Adding an attribute affects which applications? Changing a key affects which relationships? Modifying constraints affects which processes?

Dependency tracing follows relationships to understand cascading effects. Changing customer identifier affects orders, payments, and shipments that reference customers.

Extensibility planning identifies how the model can grow. Where can new entities be added? How can existing entities be extended? What patterns support future needs?

5.3 Enforcing Data Quality

The data model supports enforcing data quality by avoiding redundancy and inconsistency.

Normalization eliminates redundant storage of the same facts, preventing inconsistencies when data is updated in one place but not another.

Constraints prevent invalid data from being stored, catching errors at the database level rather than relying on application logic.

Referential integrity ensures relationships remain valid, preventing orphaned records and broken references.

Domain definitions ensure attribute values are appropriate, preventing type mismatches and invalid values.

5.4 Guiding Implementation

The data model serves as a blueprint for implementation, guiding implementation of modules that access the data.

Database schema design derives directly from the logical data model, with physical optimizations for the chosen technology.

Object-relational mapping configures how application objects correspond to data entities.

API design reflects data structures in request and response formats.

Query design follows relationship paths defined in the model.

Validation logic implements constraints specified in the model.

5.5 Enabling Integration

The data model enables system integration.

Data contracts define the structure of shared data.

Mapping specifications describe how data transforms between systems.

Canonical models provide common data structures for enterprise integration.

Master data management establishes authoritative sources for shared entities.

5.6 Supporting Analytics

The data model supports analytical use cases.

Dimensional modeling organizes data for analytical queries with facts and dimensions.

Data warehouse design derives from operational data models.

Report design follows entity and relationship structures.

Data lineage traces data from source through transformations.

6 Notations

Data models can be represented using various notations.

6.1 Entity-Relationship Diagrams

Entity-relationship (ER) diagrams are the classic notation for data modeling.

Entities are shown as rectangles with the entity name.

Attributes are shown as ovals connected to entities, or listed within the entity rectangle.

Relationships are shown as diamonds connected to related entities, or as lines between entities.

Cardinality is indicated through various conventions (crow's foot, min-max notation, or UML multiplicity).

6.1.1 Chen Notation

Chen notation is the original ER notation.

Entities are rectangles. Attributes are ovals. Relationships are diamonds. Primary key attributes are underlined. Cardinality uses labels (1, M, N) near connections.

Chen notation is clear and educational but can become cluttered for large models.

6.1.2 Crow's Foot Notation

Crow's foot notation (also called IE notation or Martin notation) is widely used in database tools.

Entities are rectangles with attributes listed inside. Relationships are lines between entities. Cardinality uses symbols at line ends: a single line for one, crow's foot (three-line fork) for many, circle for optional, bar for mandatory.

Crow's foot notation is compact and widely supported by tools.

6.1.3 IDEF1X Notation

IDEF1X is a standard notation used in government and some industries.

Entities are rectangles with rounded or square corners distinguishing independent and dependent entities. Relationships are lines with dots indicating cardinality. Identifying relationships use solid lines; non-identifying use dashed lines.

IDEF1X is precise and formal but less commonly used than crow's foot notation.

6.2 UML Class Diagrams

UML class diagrams can represent data models.

Entities are shown as classes with attributes and optional operations.

Relationships are shown as associations with multiplicity annotations.

Generalization uses hollow triangle arrowheads.

Aggregation uses diamond shapes (hollow for aggregation, filled for composition).

UML is familiar to developers and integrates with other UML diagrams.

6.3 Textual Notations

Data models can be represented textually.

Data Definition Language (DDL) defines database schemas in SQL or similar languages, providing precise, executable specifications.

Schema definition languages like JSON Schema, XML Schema, or Protocol Buffers define data structures for APIs and documents.

Documentation formats describe entities, attributes, and relationships in structured prose or tables.

6.4 Data Dictionaries

Data dictionaries provide comprehensive attribute-level documentation.

Entity definitions describe each entity's purpose and scope.

Attribute definitions describe each attribute with type, constraints, and business meaning.

Relationship definitions describe associations between entities.

Code tables document reference data values and their meanings.

7 Quality Attributes

Data model design significantly affects system quality attributes.

7.1 Data Integrity

The data model directly affects data integrity.

Entity integrity ensures entities are uniquely identifiable through primary key design.

Referential integrity ensures relationships are valid through foreign key design.

Domain integrity ensures values are valid through attribute constraints.

Business rule integrity ensures data satisfies business rules through check constraints and triggers.

7.2 Performance

Data model design significantly affects performance.

Query performance depends on how data is organized and related. Normalized models may require many joins. Denormalized models may enable faster queries.

Write performance depends on constraints and indexes. More constraints and indexes slow writes while enabling faster reads.

Storage efficiency depends on normalization level. Normalized models minimize storage. Denormalized models trade storage for speed.

Scalability depends on data distribution possibilities. Some models partition easily; others have cross-partition dependencies.

7.3 Modifiability

Data models evolve over time, making modifiability important.

Schema evolution capability determines how easily the model can change. Adding attributes is usually easy. Changing keys or relationships is difficult.

Backward compatibility enables changes without breaking existing applications.

Data migration complexity determines the cost of model changes.

Extensibility patterns enable anticipated growth without schema changes.

7.4 Security

Data models support security through structure.

Access control granularity depends on entity organization. Separate entities can have separate permissions.

Data classification is represented through entity and attribute properties.

Audit capability depends on entity design for tracking changes.

Encryption scope aligns with entity and attribute boundaries.

7.5 Usability

Data model design affects developer and analyst usability.

Conceptual clarity depends on how well the model reflects business concepts.

Query simplicity depends on how directly queries map to model structure.

Documentation quality depends on how well the model is described.

Tooling support depends on how standard the model notation is.

7.6 Interoperability

Data models affect system integration.

Standard alignment enables integration with systems using common models.

Transformation complexity depends on how different source and target models are.

Canonical forms provide common ground for diverse systems.

8 Common Data Model Patterns

Several recurring patterns address common data modeling challenges.

8.1 Audit Trail Pattern

Audit trails track changes to data over time.

Audit attributes add created-by, created-date, modified-by, and modified-date to entities.

Audit tables store copies of changed records with change metadata.

Event sourcing captures all changes as immutable events.

Temporal tables maintain complete history with valid-time tracking.

8.2 Soft Delete Pattern

Soft delete preserves deleted records for recovery or audit.

A deleted flag or deleted-date attribute indicates logical deletion.

Queries filter out deleted records by default.

Deleted records can be restored or permanently purged later.

Referential integrity must consider soft-deleted records.

8.3 Type Code Pattern

Type codes distinguish entity subtypes without separate tables.

A type code attribute indicates the subtype.

Type-specific attributes may be nullable or use a generic structure.

This is simpler than multiple tables but less strictly typed.

8.4 Self-Referencing Pattern

Self-referencing entities model hierarchies within a single entity type.

A foreign key references the same entity's primary key.

This represents parent-child relationships like organizational hierarchy or category trees.

Queries may require recursive constructs to traverse the hierarchy.

8.5 Bridge Table Pattern

Bridge tables implement many-to-many relationships.

A separate table contains foreign keys to both related entities.

The bridge table may have additional attributes for the relationship.

Primary key is typically the combination of both foreign keys.

8.6 Multi-Tenant Pattern

Multi-tenant models support multiple customers in shared infrastructure.

Tenant identifier is included in primary keys or as a partition key.

Row-level security filters data by tenant.

Separate schemas or databases provide stronger isolation but more complexity.

8.7 Polymorphic Association Pattern

Polymorphic associations allow referencing different entity types.

A type indicator specifies which entity type is referenced.

An identifier references the specific instance.

This is flexible but cannot enforce referential integrity at the database level.

8.8 Entity-Attribute-Value Pattern

Entity-Attribute-Value (EAV) provides schema flexibility.

A single table stores entity identifier, attribute name, and attribute value.

Any attributes can be added without schema changes.

This is extremely flexible but sacrifices type safety, referential integrity, and query performance.

EAV is appropriate for sparse, highly variable attributes but should not be used for core structured data.

8.9 Slowly Changing Dimension Patterns

Slowly changing dimension (SCD) patterns track historical changes for analytics.

Type 1 overwrites old values with new values, losing history.

Type 2 adds new rows for changes, maintaining full history with version dates.

Type 3 adds columns for previous values, maintaining limited history.

Type 4 uses separate history tables.

8.10 Master Data Pattern

Master data patterns establish authoritative sources for shared entities.

Golden records represent the authoritative version of each entity.

Matching and merging processes consolidate data from multiple sources.

Survivorship rules determine which source values are authoritative.

Distribution processes propagate master data to consuming systems.

9 Data Modeling for Different Technologies

Data modeling approaches vary for different database technologies.

9.1 Relational Database Modeling

Relational modeling maps directly to SQL databases.

Entities become tables. Attributes become columns. Relationships become foreign keys. Constraints become database constraints.

Normalization is the primary design principle, with selective denormalization for performance.

Indexing strategy significantly affects performance.

9.2 Document Database Modeling

Document modeling organizes data around aggregates.

Aggregates become documents. Related data is embedded within documents. References link documents when embedding is inappropriate.

Modeling decisions depend on access patterns. Embed data that is accessed together. Reference data that is accessed separately or shared.

Denormalization is common; consistency is managed at the application level.

9.3 Graph Database Modeling

Graph modeling emphasizes relationships.

Entities become nodes. Relationships become edges. Properties attach to both nodes and edges.

Modeling focuses on traversal patterns. What relationships need to be navigated? What paths need to be queried?

Graph models excel for highly connected data and relationship-centric queries.

9.4 Key-Value Store Modeling

Key-value modeling organizes data by access key.

Entities are stored as values under unique keys. Keys are designed for efficient access patterns. Values may be opaque blobs or structured documents.

Modeling focuses on key design. What queries must be supported? What key structures enable those queries?

Secondary indexes or denormalization support queries beyond primary key.

9.5 Column-Family Store Modeling

Column-family modeling organizes data for sparse, wide data.

Row keys identify entities. Column families group related attributes. Columns within families can vary per row.

Modeling focuses on query patterns. What data is accessed together? What queries must be efficient?

Denormalization is standard; data is modeled for specific query patterns.

9.6 Time-Series Database Modeling

Time-series modeling optimizes for temporal data.

Measurements are the primary entity with timestamp, metric, value, and tags.

Data is organized by time, typically partitioned into time ranges.

Aggregation and downsampling are built into the model.

10 Examples

Concrete examples illustrate data model concepts.

10.1 E-Commerce Data Model

An e-commerce system illustrates common patterns.

Core entities include Customer with attributes for identity, contact, and preferences; Product with attributes for identity, description, pricing, and inventory; Order with attributes for identity, customer reference, status, and totals; and OrderLine with attributes linking order and product with quantity and price.

Relationships include Customer to Order (one-to-many), Order to OrderLine (one-to-many, composition), and OrderLine to Product (many-to-one).

Reference entities include Category with hierarchical self-reference, PaymentMethod with types and details, and ShippingMethod with options and costs.

10.2 Healthcare Data Model

A healthcare system illustrates complex relationships and constraints.

Core entities include Patient with demographics and medical record number, Provider with credentials and specialties, Encounter representing patient visits, and Diagnosis linking encounters to conditions.

Relationships are complex with many-to-many between patients and providers through encounters, hierarchical diagnosis codes, and temporal validity for insurance coverage.

Constraints include privacy requirements for patient data, audit requirements for all access, and regulatory requirements for data retention.

10.3 Content Management Data Model

A content management system illustrates hierarchical and polymorphic data.

Core entities include Content as a base entity with metadata, Article extending Content with body and author, Media extending Content with file reference and format, and Folder organizing content hierarchically.

Generalization shows Content as supertype with Article, Media, and other subtypes.

Relationships include hierarchical folder structure through self-reference, many-to-many content-to-tag relationships, and content versioning through temporal patterns.

10.4 Financial Data Model

A financial system illustrates transaction and temporal modeling.

Core entities include Account with type, owner, and balance; Transaction representing money movement with type, amount, and timestamp; and Position tracking holdings with quantity and valuation.

Temporal aspects include effective dates for price changes, as-of queries for historical positions, and audit trails for all changes.

Constraints include balance rules that debits equal credits, referential rules that transactions reference valid accounts, and business rules for approval workflows.

11 Best Practices

Experience suggests several best practices for data modeling.

11.1 Start with Business Understanding

Data models should reflect business reality.

Engage business stakeholders to understand concepts and terminology.

Use business language in entity and attribute names.

Validate the model with business experts.

Align entity boundaries with business concept boundaries.

11.2 Model Iteratively

Data modeling is an iterative process.

Start with core entities and relationships.

Add detail incrementally as understanding grows.

Validate with use cases and queries.

Refine based on feedback and requirements changes.

11.3 Normalize, Then Denormalize Strategically

Start with normalization, then optimize.

Design for correctness first through normalization.

Identify performance requirements through testing.

Denormalize specifically for measured bottlenecks.

Document and manage denormalization carefully.

11.4 Design for Evolution

Data models will change; design for it.

Prefer additive changes (new entities, attributes, relationships).

Use surrogate keys that won't need to change.

Plan for data migration with major changes.

Version APIs that expose data structures.

11.5 Document Thoroughly

Data models require comprehensive documentation.

Document each entity's purpose and scope.

Document each attribute's meaning and constraints.

Document relationships and their semantics.

Document constraints and business rules.

Keep documentation synchronized with the model.

11.6 Consider Access Patterns

Model structure should support required access.

Identify key queries and reports.

Design relationships to support navigation patterns.

Index for query performance.

Partition for scalability.

11.7 Enforce Constraints Appropriately

Constraints should be enforced at the right level.

Enforce at the database level when possible.

Use application validation for complex rules.

Document constraints that cannot be enforced technically.

Test constraint enforcement.

12 Common Challenges

Data modeling presents several common challenges.

12.1 Balancing Normalization and Performance

Normalization and performance often conflict.

Fully normalized models require many joins.

Denormalization improves queries but complicates updates.

The right balance depends on workload characteristics.

Strategies include normalizing by default, measuring performance, denormalizing for specific bottlenecks, and considering read replicas for reporting.

12.2 Managing Schema Evolution

Schemas must evolve as requirements change.

Schema changes may require application changes.

Data migration may be complex and risky.

Backward compatibility constrains evolution.

Strategies include planning for evolution from the start, using additive changes when possible, implementing migration scripts and testing thoroughly, and versioning for breaking changes.

12.3 Handling Unstructured Data

Not all data fits neatly into structured models.

Documents, images, and other content resist tabular representation.

User-generated content has unpredictable structure.

Integration data may have varying formats.

Strategies include using appropriate storage (document stores, object storage), modeling metadata separately from content, using EAV patterns sparingly for truly variable attributes, and combining structured and unstructured storage.

12.4 Integrating Multiple Models

Enterprise data often spans multiple models and systems.

Different systems may model the same concepts differently.

Integration requires mapping between models.

Master data management adds complexity.

Strategies include establishing canonical models, defining clear ownership and authority, implementing robust mapping and transformation, and governance to maintain alignment.

12.5 Modeling Time

Temporal data adds complexity.

Point-in-time queries require historical data.

Multiple time dimensions (valid time, transaction time) complicate models.

Temporal joins and queries are complex.

Strategies include choosing appropriate temporal patterns, using database temporal features where available, designing for most common temporal queries, and documenting time semantics clearly.

12.6 Security and Privacy

Data security adds modeling constraints.

Sensitive data requires protection.

Access control affects model structure.

Privacy regulations constrain data handling.

Strategies include classifying data sensitivity, designing for row-level and column-level security, separating sensitive attributes, and planning for data minimization and deletion.

13 Conclusion

The data model style provides essential structure for the information that systems create, store, and use. By defining data entities, their attributes, and their relationships, data models enable systems to manage information effectively, maintain data quality, and support business operations.

Effective data modeling requires understanding business concepts, applying appropriate modeling techniques, and balancing various quality attributes including integrity, performance, and modifiability. The patterns and practices described in this document provide guidance for creating data models that serve both immediate needs and long-term evolution.

Data models are foundational to software architecture. While applications change, data endures. A well-designed data model provides stability that supports system evolution, integration, and growth over time.

Understanding data modeling equips architects to design information structures that serve business needs, guide implementation, enable integration, and maintain quality throughout the data lifecycle.

References

- Clements, P., Bachmann, F., Bass, L., Garlan, D., Ivers, J., Little, R., Merson, P., Nord, R., & Stafford, J. (2010). *Documenting Software Architectures: Views and Beyond* (2nd ed.). Addison-Wesley Professional.
- Simsion, G. C., & Witt, G. C. (2004). *Data Modeling Essentials* (3rd ed.). Morgan Kaufmann.
- Hay, D. C. (2006). *Data Model Patterns: A Metadata Map*. Morgan Kaufmann.
- Fowler, M. (2002). *Patterns of Enterprise Application Architecture*. Addison-Wesley Professional.
- Kleppmann, M. (2017). *Designing Data-Intensive Applications*. O'Reilly Media.
- Silverston, L. (2001). *The Data Model Resource Book* (Revised ed., Vol. 1). Wiley.
- Hoberman, S. (2009). *Data Modeling Made Simple* (2nd ed.). Technics Publications.