Chapter 2: Data models

# The Importance of Data Models

**Data models** are essentially blueprints for how data is stored, organized, and managed in a system. They help define the structure of data, including how it’s related and how it can be used. Think of a data model like a map that guides you through a city; it shows you where everything is located and how different places are connected. In the same way, a data model shows how different pieces of data are connected within a database or a software application. It’s a plan that developers and database administrators use to build databases that are efficient, reliable, and scalable. Data models are crucial because they ensure that all the data used by a business, or an application is structured in a way that makes sense for the processes that need to use it.

# Data Model Basic Building Blocks

Data models are essential frameworks that define how information is structured and connected within a database. They are crucial for ensuring that data is organized logically and can be used effectively by applications and users.

**Entities** are the primary components of a data model. They represent real-world objects or concepts that have significance within the context of the database. For example, in a school database, entities might include ‘Student’, ‘Teacher’, and ‘Course’. These entities can be tangible, like a ‘Book’ in a library system, or intangible, like a ‘Course’ offered at a university.

**Attributes** are the properties or characteristics that define an entity. They provide specific details about the entity, much like descriptors. In the case of a ‘Student’ entity, attributes could include ‘Student ID’, ‘Name’, ‘Date of Birth’, and ‘Major’. These attributes hold the data that is pertinent to the entity they describe, such as a student’s name or their major field of study.

**Relationships** are the connections between entities, showing how they interact with each other. They are fundamental to relational databases because they define how entities relate to one another. Data models use three types of relationships: one-to-many, many-to-many and one-to-one. Database designers usually use the shorthand notations 1:, : and 1:1, respectively. The following examples illustrate the distinctions among the three:

* **One-to-many (1:) relationship**. A painter paints many different paintings, but each one of them is painted by only one painter. Thus, the painter (the one) is related to the paintings (the many). Therefore, database designers label the relationship PAINTER paints PAINTING as 1:. (Note that entity names are often capitalised as a convention, so they are easily distinguished.) Similarly, a customer (the one) may generate many invoices, but each invoice (the many) is generated by only a single customer. The CUSTOMER generates INVOICE relationship would also be labelled 1:.
* **Many-to-many (:) relationship**. An employee may learn many job skills, and each job skill may be learnt by many employees. Database designers label the relationship EMPLOYEE learns SKILL as :. Similarly, a student can take many classes and each class can be taken by many students, thus yielding the : relationship label for the relationship expressed by STUDENT takes CLASS.
* **One-to-one (1:1) relationship**. A retail company’s management structure may require that each of its stores be managed by a single employee. In turn, each store manager, who is an employee, manages only a single store. Therefore, the relationship EMPLOYEE manages STORE is labelled 1:1.

**Constraints** are rules that enforce certain conditions on the data to maintain its integrity and validity. They ensure that the data adheres to specific requirements, such as a ‘Salary’ attribute for an ‘Employee’ entity being within a certain range. Constraints are vital for preventing incorrect or invalid data from entering the system, which could lead to errors and inconsistencies. Constraints are normally expressed in the form of rules, for example:

* The employees salary must have values that are between 6 000 and 350 000.
* A students grade must be between 0 and 100.
* Each class must have one and only one teacher.

To properly identify entities, attributes, relationships, and constraints, one must start with a clear understanding of the business rules and the nature of the data. This involves analysing the real-world objects or concepts the database needs to represent and determining how they interact with each other. By doing so, a database designer can create a data model that accurately reflects the organization’s operations and provides a solid foundation for managing its data effectively.

# Data Model Basic Building Blocks

In the realm of database design, the establishment of business rules is a critical step that informs the structure and function of the data model. **Business rules** are the guidelines that dictate how data is managed within an organization, and they are derived from a comprehensive understanding of the organization’s operations. These rules are not just limited to businesses; they apply to any entity that manages data, including government units, religious groups, and research laboratories.

Business rules serve as the foundation for defining the entities, attributes, relationships, and constraints within a database. They are the written expressions of the policies, procedures, and principles that govern the organization’s data-related activities. For instance, a business rule stating that “a customer may generate many invoices” establishes the entity ‘Customer’, the entity ‘Invoice’, and a one-to-many relationship between them. Another rule might set a constraint, such as “a training session must have at least ten but no more than thirty employees,” which also implies entities and relationships.

Discovering business rules involves engaging with various sources within the organization, such as managers, policy makers, and existing documentation. Direct interviews with end users can also provide insights, although their perceptions may need verification due to potential discrepancies in understanding the operational processes.

The documentation of business rules is vital for several reasons:

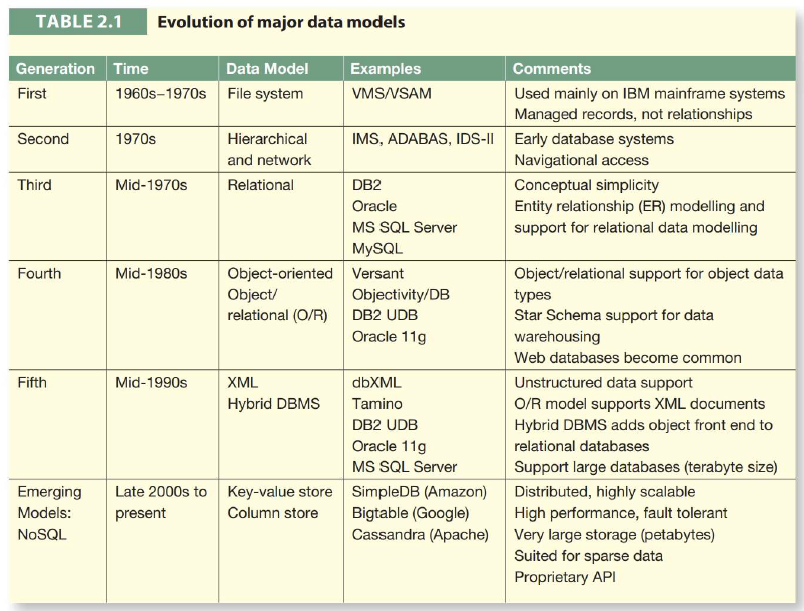
* It standardizes the organization’s view of its data.
* It facilitates communication between users and database designers.
* It enables designers to grasp the nature, role, and scope of the data, as well as the underlying business processes.
* It guides the creation of an accurate data model with appropriate rules and constraints.

However, not all business rules are directly translatable into data model components. Some, like a rule limiting a pilot’s flying hours, may be enforced through application software rather than the database model itself.

Translating business rules into data model components involves identifying **nouns as entities** and **verbs as relationships**. For example, from the rule “a customer may generate many invoices,” we deduce that ‘Customer’ and ‘Invoice’ are entities, and ‘generate’ indicates a relationship. To determine the type of relationship, we consider its bidirectionality—how many instances of one entity relate to another. Asking questions like “How many classes can one student enroll in?” and “How many students can enroll in one class?” helps identify the relationship as many-to-many (*:*).

This process of translating business rules into data model components is essential for developing a data model that accurately reflects the organization’s needs and facilitates effective data management.

# The Evolution of Data Models



## The Hierarchical and Network Model

The hierarchical and network models are two of the earliest types of database models that were developed to organize and manage data effectively.

**Hierarchical Model**:

* The hierarchical data model organizes data in a tree-like structure, where each record has a single parent, but can have multiple children.
* This model is similar to a family tree and was used by IBM in the 1960s.
* It is based on parent-child relationships, which are easy to understand and navigate. However, this model has limitations, such as difficulty in representing many-to-many relationships and a lack of flexibility due to its rigid structure.

**Network Model**:

* The network model is an extension of the hierarchical model that allows more complex relationships with its graph-like structure[.](https://www.geeksforgeeks.org/difference-between-hierarchical-and-network-data-model/)
* In this model, a child can have more than one parent, which is useful for representing more complex relationships.
* It uses records and sets (similar to tables and relationships in relational models) to represent and manage data.
* Despite its flexibility compared to the hierarchical model, the network model is still less used today due to its complexity and the advent of more advanced models like the relational model.

While the network database model is generally not used today, the definitions of standard database concepts that emerged with the network model are still used by modern data models:

* 1. **Schema**:
* A schema is the overall logical structure of the entire database as defined by the database administrator.
* It includes the definitions of tables, fields, relationships, and constraints that dictate how data is stored and organized.
  1. **Subschema**:
* A subschema is a subset of the schema that defines how the application programs view the data in the database.
* It allows different applications to have customized views of the database, which is particularly useful when different users need to see different aspects of the data.
  1. **Data Manipulation Language (DML)**:
* DML is a group of commands used to retrieve, insert, update, and delete data in a database.
* It is a sublanguage of SQL and includes commands like SELECT, INSERT, UPDATE, and DELETE.
  1. **Schema Data Definition Language (DDL)**:
* DDL consists of SQL commands used to define and modify the database schema.
* It includes commands like CREATE, ALTER, and DROP, which are used to create and alter the structure of database objects.

These components are integral to the functioning of modern databases, providing the means to define, access, and manipulate data within a structured environment. While the hierarchical and network models are not commonly used today, their principles have laid the groundwork for the relational model and other contemporary database systems.

## The Relational Model

The relational model, introduced by E.F. Codd in 1970, marked a significant shift in the way databases were conceptualized and managed. It simplified the database structure by presenting data as a collection of tables, or relations, which could be easily understood and manipulated by both users and designers. This simplicity was akin to shifting from a manual to an automatic transmission in a car, making databases more accessible to a broader range of users.

Initially, the relational model’s simplicity came at the cost of higher computational overhead, which was a concern given the limited computing power available at the time. However, as technology advanced, with exponential increases in computing power and efficiency, coupled with decreasing costs, the relational model became practical and widely adopted. Today, sophisticated relational database software runs on devices far less expensive than the mainframes of the past.

A **relational database management system (RDBMS**) implements the relational model and offers several advantages over its hierarchical and network predecessors. One key advantage is **abstraction**; the RDBMS hides the complexities of the database’s physical storage from the user. Users interact with the data through a logical framework of tables, which appear as simple row and column matrices. These tables are interrelated through shared fields, allowing for easy data association while minimizing redundancy.

For example, a CUSTOMER table might share a field like AGENT\_CODE with an AGENT table, enabling users to associate customers with their respective agents seamlessly. This controlled redundancy is a deliberate design choice to eliminate the excessive and often problematic redundancies found in file systems.

The relational model also provides **complete data and structural independence**, meaning the physical storage of data is irrelevant to users and designers—it’s the logical perception that matters. This independence is a cornerstone of the relational model’s success, as it allows for flexibility and scalability without burdening the user with implementation details.

Another factor contributing to the relational model’s dominance is **SQL (Structured Query Language)**, a powerful and flexible query language that enables users to specify what data they want without detailing how to retrieve it. The RDBMS uses SQL to translate user queries into actions, simplifying data retrieval. SQL’s declarative nature allows users to focus on the logical aspects of the database, leaving the RDBMS to handle the physical complexities.

In a typical SQL-based relational database application, there are three main components:

* **The End-User Interface**: This interface allows users to interact with the data, often auto-generating SQL code for operations. It can be vendor-provided or custom-designed using application generators.
* **A Collection of Tables Stored in the Database**: Data is perceived as being stored in tables, which present information in an easily understandable format. Each table is independent, but rows across tables can be related through common values in shared attributes.
* **SQL Engine**: Operating largely unseen by the end user, the SQL engine executes all data queries and requests. It’s a part of the DBMS software that translates user requests into the necessary instructions to access and maintain the data.

The RDBMS’s ability to manage the physical details of data storage while providing a logical, table-based interface revolutionized database design and usage. It allowed users to concentrate on the logical structure and relationships within the data, fostering a more intuitive and effective way to manage and query information. This logical focus is explored further in subsequent chapters, along with detailed discussions on SQL and its applications in database design and performance management.

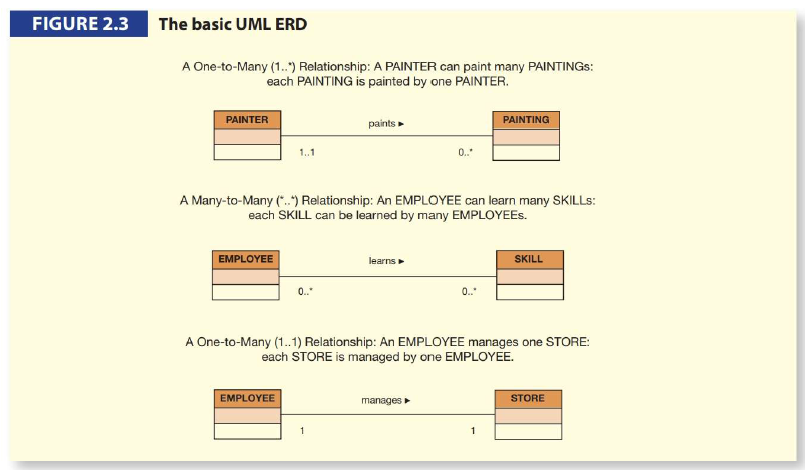
## The Entity Relational Model

The relational database model’s conceptual simplicity was a game-changer in the world of database technology. It made databases more accessible and easier to manage, much like an automatic transmission made driving more accessible compared to a manual transmission. This simplicity led to a surge in demand for Relational Database Management Systems (RDBMSs), which in turn required more sophisticated tools for database design, especially as the complexity of transactions and information grew.

However, despite its improvements over previous models, the relational model alone wasn’t sufficient as a design tool. It lacked certain features that would make it truly effective for complex database design tasks. To address this, database designers turned to graphical tools that could visually represent the entities and their relationships within a database, leading to the adoption of the **Entity-Relationship (ER) model**.

Introduced by Peter Chen in 1976, the ER model provided a graphical representation of entities and relationships, complementing the relational model and becoming a standard for data modeling. The ER model’s strength lies in its clear distinction between entities (things about which data are collected) and relationships (how these things are associated). Initially, Chen’s notation allowed relationships to have their own attributes, sparking debates about the nature of entities and relationships. Over time, more graphical versions of Chen’s notation, such as the Crow’s Foot notation, were developed to represent these relationships more clearly.

The **Crow’s Foot model**, developed by Gordon Everest and popularized by Clive Finkelstein and James Martin, uses graphical symbols to represent relationships, with a three-pronged symbol indicating the ‘many’ side of a relationship. While there’s a shift towards using the Unified Modeling Language (UML), many organizations, especially those with legacy systems, still use the Crow’s Foot notation.



**UML**, an object-oriented modeling language standardized by the Object Management Group (OMG) in 1997, has also been used to produce ER models. Although UML was developed for object-oriented design, its class diagrams are increasingly used as the industry standard for data modeling. UML is not a methodology but a language that describes a set of diagrams and symbols for graphically modeling systems.

In the ER model:

* **Entities** are represented by rectangles and described by a set of attributes.
* **Relationships** describe associations among entities and are represented by lines connecting the entities, with the relationship type (one-to-many, many-to-many, one-to-one) indicated by the connectivity.

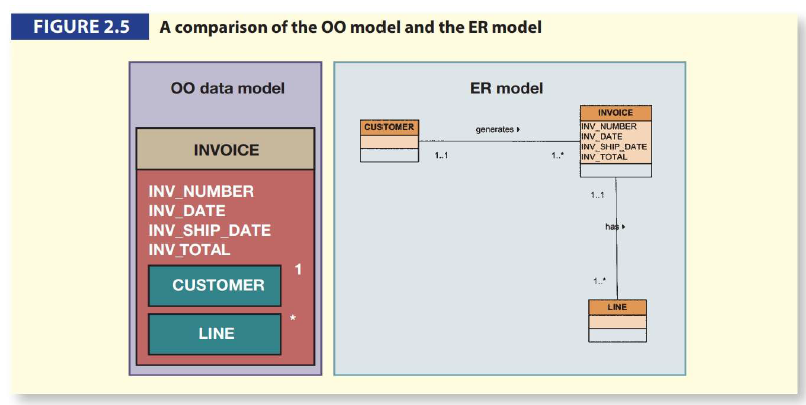
The ER model and the relational database model together provide a solid foundation for structured database design, allowing for the creation of intuitive and logical representations of complex data structures. As database modeling tools evolve, they continue to support various notations, including UML, enabling designers to choose the most appropriate method for their specific needs. The subsequent chapters will delve deeper into data modeling with ER diagrams, exploring both Crow’s Foot notation and UML class diagram notation in more detail.

## The Object-Oriented (O-O) Model

The **Object-Oriented Data Model (OODM)** emerged as a response to the need for a data model that more accurately mirrors the complexities of the real world. In an OODM, data and their relationships are encapsulated within objects, which are instances of classes. This model is the foundation of the Object-Oriented Database Management System (OODBMS), which manages the storage and manipulation of objects.

The OODM differs from traditional entity-based models in that it includes not only data (attributes) but also the relationships between data and the operations (methods) that can be performed on the data. This encapsulation makes objects a powerful and self-contained unit of data management, leading to the term “semantic data model,” where ‘semantic’ refers to the meaning that is inherently part of the objects and their interactions.

In the OODM, objects are abstractions of real-world entities, and they are grouped into classes based on shared characteristics. **Classes** define the structure (attributes) and behavior (methods) of their objects. The OODM also introduces the concept of inheritance, where subclasses can inherit attributes and methods from their parent classes, forming a class hierarchy.



This model’s semantic richness allows for a more intuitive representation of complex real-world scenarios. For example, in an invoicing system, an object representing an invoice would include all related objects, such as customer and line items, within the same object structure. This contrasts with the Entity-Relationship (ER) model, where separate entities and relationships would be used to represent the same scenario.

The OODM’s influence extends beyond database design to programming, with many contemporary languages incorporating object-oriented concepts. This has allowed for the development of applications that can handle complex objects and relationships in innovative ways.

It’s important to recognize that different data models serve different purposes. Conceptual models like the ER model are ideal for high-level data modelling, focusing on the broader organizational aspects. In contrast, implementation models like the hierarchical and network models are more concerned with the technical details of data storage and access. The relational model and the OODM have the flexibility to be used as both conceptual and implementation models, bridging the gap between high-level design and practical application.

In summary, the OODM offers a comprehensive approach to data modelling that aligns closely with real-world entities and their interactions, providing a robust framework for managing complex data structures and relationships within modern database systems.

## Other Models

The evolution of database technology has been driven by the need to manage increasingly complex data in a way that reflects the real world more closely. The relational model, which revolutionized data management with its simplicity and mathematical foundation, faced limitations when it came to representing complex objects and relationships inherent in certain domains.

To bridge this gap, the **Extended Relational Data Model (ERDM)** was developed, incorporating features from the Object-Oriented (OO) model into the relational framework. This hybrid approach led to the creation of **Object-Relational Database Management Systems (ORDBMS)**, which combine the best of both worlds: the conceptual simplicity and robustness of the relational model with the rich semantics and flexibility of the OO model.

**ORDBMS**s have gained significant market share, particularly in Online Transaction Processing (OLTP) and Online Analytical Processing (OLAP) applications, due to their enhanced capabilities. They support complex data types, inheritance, and encapsulation of data and methods within database objects. Meanwhile, Object-Oriented Database Management Systems (OODBMS) have found their niche in specialized areas like CAD/CAM, GIS, and multimedia, where the intricate representation of objects is crucial.

The relational model’s adaptability was further demonstrated with the advent of the internet, which necessitated the management of unstructured data such as documents, emails, and web pages. **XML** emerged as a standard for data exchange, and databases evolved to manage XML data natively. **ORDBMSs** extended their functionality to include support for XML, showcasing the relational model’s ability to incorporate new data types and structures.

Another specialized area that required a different approach is **spatial data management**. Spatial data, which includes multi-dimensional objects like cities or forests, is not efficiently handled by traditional relational databases due to its complexity and size. **Spatial Database Management Systems (SDBMS)** were developed to address this, introducing spatial data types and indexing to efficiently store, retrieve, and manage spatial data. These systems are crucial for applications like route optimization and urban planning, with popular implementations seen in GIS applications such as Google Earth.

In summary, the relational model’s core principles have proven to be extensible, allowing it to evolve and integrate new features like object orientation and XML support. This has enabled databases to continue meeting the diverse and complex data management needs of modern organizations, while specialized systems like SDBMS cater to specific domains where traditional models may fall short.

## Emerging Data Models: Big Data and NoSQL

The advent of **Big Data** has fundamentally changed the landscape of data management. As organizations grapple with the deluge of data generated from web interactions, social media, and various online activities, traditional relational database management systems (RDBMS) are being pushed to their limits. The structured nature of RDBMS, organized into rows and columns, struggles to accommodate the diverse and unstructured nature of Big Data, which includes everything from text and images to complex user behavior analytics.

The challenges posed by Big Data are multifaceted:

* **Structural Rigidity**: The relational model’s structured format is not conducive to the varied and unstructured nature of social media data.
* **Volume and Velocity**: The sheer volume of data being generated and the speed at which it accumulates demand more agile and scalable solutions.
* **Cost**: The cost of scaling traditional RDBMS to handle Big Data can be prohibitive, requiring significant investment in hardware, storage, and software licenses.
* **Data Analysis**: While OLAP tools are effective for structured data, they fall short when it comes to extracting insights from unstructured data sources.

In response to these challenges, **NoSQL** databases have emerged as a viable alternative. They offer a more flexible schema-less architecture, capable of handling the variety and complexity of Big Data. NoSQL databases are designed to scale out by distributing the data across many servers, and they often provide cost-effective solutions due to their open-source nature and the ability to run on commodity hardware.

Organizations are increasingly turning to NoSQL solutions to harness the potential of Big Data. By doing so, they can gain insights into customer behavior, improve decision-making, and maintain a competitive edge in a rapidly evolving digital marketplace. The lessons learned from companies that failed to adapt to technological shifts underscore the importance of staying ahead in data management capabilities.

As we move forward, the integration of Big Data analytics with NoSQL databases is likely to play a pivotal role in shaping business strategies and driving innovation. The ability to manage and analyze vast amounts of diverse data efficiently will continue to be a key differentiator for successful organizations in the digital age.

## NoSQL Databases

NoSQL databases have become integral to the functioning of many of the web services we use daily, such as Amazon, Facebook, YouTube, and Google Maps. These databases are designed to handle the vast amounts of data generated by these services, providing a level of performance, scalability, and flexibility that traditional relational databases cannot match.

**Characteristics of NoSQL Databases**:

* **Non-relational**: NoSQL databases do not follow the relational model, and there is no standard data model for NoSQL. They encompass a variety of data models, including document, graph, column, and key-value stores.
* **Distributed Architecture**: They are often built with distributed architecture in mind, using clusters of low-cost commodity servers to manage large volumes of data.
* **Scalability and Fault Tolerance**: NoSQL databases are designed to scale out by adding more nodes to the cluster, and they offer high availability and fault tolerance.
* **Sparse Data**: They are particularly well-suited for managing sparse data, where the number of attributes is large, but the actual number of data instances is low.
* **Performance over Consistency**: NoSQL databases often prioritize performance over strict transaction consistency, offering eventual consistency to ensure that all copies of the data will eventually be consistent.

**Key-Value Stores**:

* In a key-value store, data is organized as a collection of key-value pairs, where each key is associated with one or more values.
* This model is schema-less, meaning new attributes can be added without altering the overall database structure.
* Relationships and data integrity are managed by the application code, not the database itself.
* NoSQL databases typically use a native API with simple commands like put, read, and delete, rather than a declarative query language like SQL.

**Column Stores**:

* Some NoSQL databases, like Google’s Bigtable and Apache’s Cassandra, extend the key-value model into column stores, grouping related key-value pairs together.
* These databases can include features like versioning, using timestamps to manage multiple versions of the data.

**Challenges with NoSQL**:

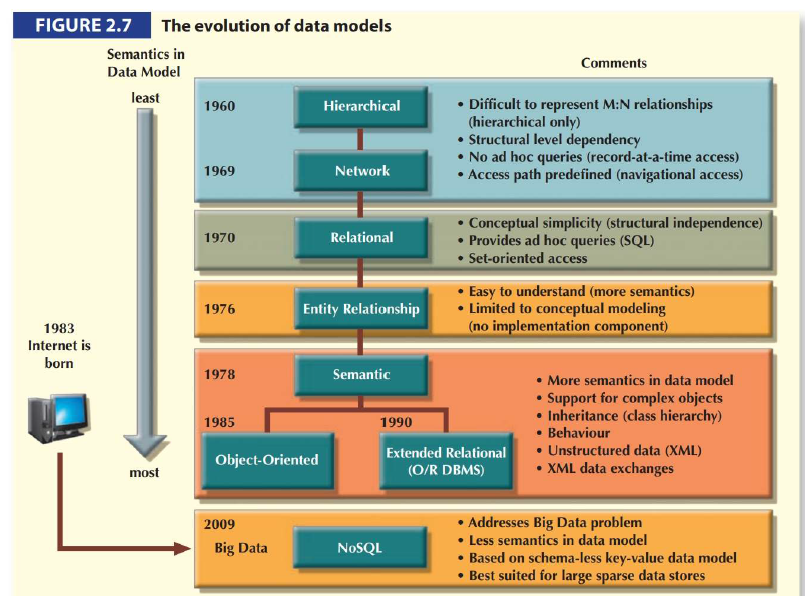
* Indexing and searching can be complex due to the variety of data types that can be stored in the value column.
* Managing relationships and ensuring data integrity can be more challenging compared to relational databases.

**NoSQL in the Big Data Era**:

* The rise of Big Data has led to an increased focus on NoSQL databases as they are well-equipped to handle the scale and complexity of web-generated data.
* NoSQL databases are particularly useful for organizations that need to manage large amounts of unstructured data, such as social media content, with high performance and cost-effectiveness.

NoSQL technology is a critical component of modern data management strategies, especially as organizations seek to leverage the wealth of information available from web sources. Understanding the strengths and limitations of NoSQL is essential for selecting the right database technology for specific applications and use cases.

## Data Models Summary



The development of database management systems (DBMSs) has been a continuous journey towards capturing the complexity of the real world within digital systems. As data models have evolved, they’ve aimed to balance conceptual simplicity with the ability to represent the real world accurately and meaningfully.

**Conceptual Simplicity and Semantic Completeness**: A data model must strike a balance between being conceptually simple and semantically complete. It should not be more complex than the real-world scenarios it represents. This means that while the model should be easy to understand, it should also be rich enough to capture the nuances of the data it’s modeling.

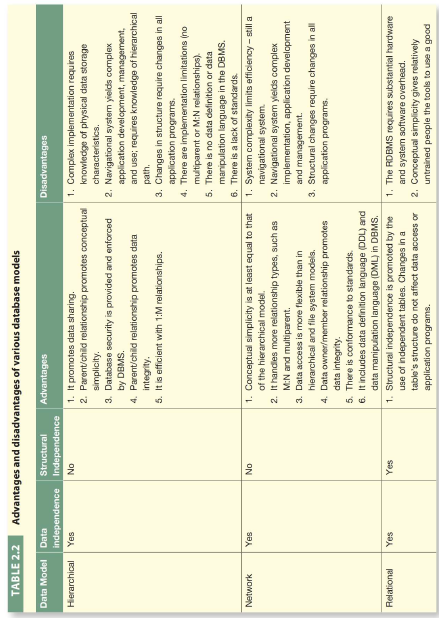
**Real-World Representation**: Data models should mirror the real world as closely as possible. This is achieved by incorporating semantics into the data representation, which involves not only the static aspects of data but also their dynamic behavior. The model should reflect real-world transformations while maintaining data consistency and integrity.

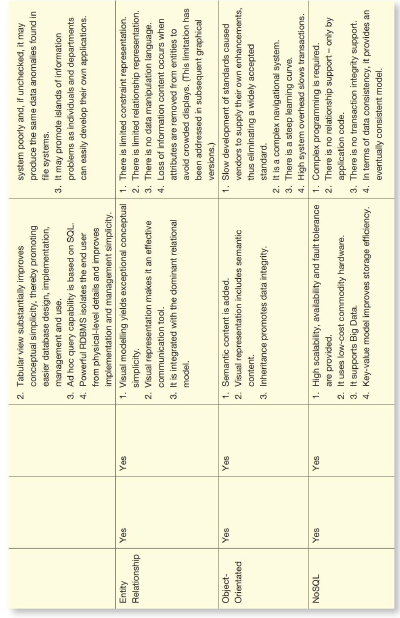
**Evolution of Data Models**:

* The **network model** made it easier to represent complex many-to-many relationships, improving upon the hierarchical model.
* The **relational model** brought simplicity in data representation, data independence, and an easy-to-use query language, becoming the dominant model for business applications.
* The **object-oriented (OO) data model** introduced support for complex data within a semantic framework.
* The **Extended Relational Data Model (ERDM)** integrated OO features into the relational model, enhancing its capabilities.
* The rise of **Big Data** has led to the development of new ways to model, store, and manage data, diverging from traditional data management approaches.

**Suitability for Tasks**: Not all data models are suitable for every task. Conceptual models like the **Entity-Relationship (ER) model** are ideal for high-level data modeling. In contrast, implementation models like the hierarchical and network models are better suited for managing stored data for implementation purposes. Some models, such as the relational model and the OO data model, can serve as both conceptual and implementation models, offering versatility in their application.

In summary, the evolution of data models reflects the ongoing effort to create systems that can handle the complexity of real-world data in a way that is both understandable and robust. Each model has contributed to this goal in its own way, building on the strengths and addressing the weaknesses of its predecessors. As we continue to face new data challenges, the development of data models will undoubtedly continue, each iteration bringing us closer to a perfect digital reflection of our complex reality.

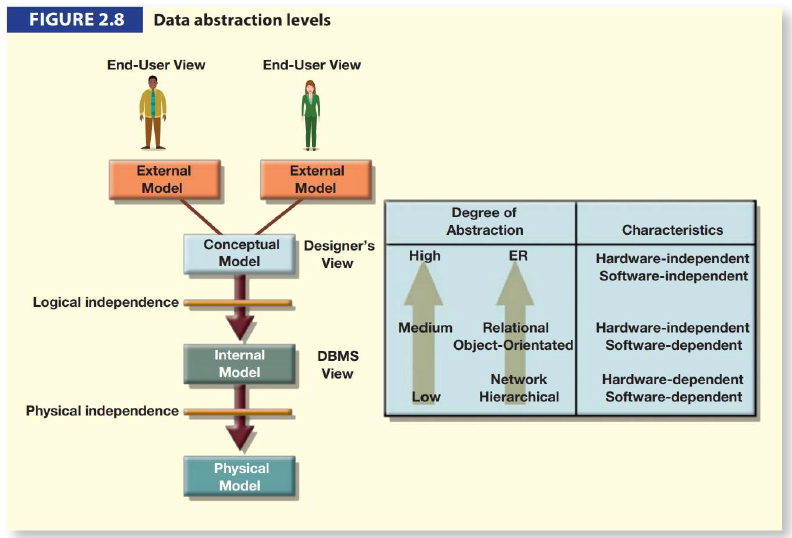




# Degrees of Data Abstraction

In the early 1970s, the American National Standards Institute (ANSI) Standards Planning and Requirements Committee (SPARC) defined a framework for data modeling based on varying degrees of data abstraction. This approach can be compared to the process of automotive design. Initially, a car designer conceptualizes the car's overall design. Engineers then refine this concept into detailed structures that can be produced. Finally, these engineering details are translated into precise production specifications for use on the factory floor. This progression from abstract design to detailed implementation is essential for producing a car. Similarly, in database design, a designer starts with an abstract overview of the data environment and gradually adds detail until the design is ready for implementation. This layered approach is particularly useful for integrating different views of data within an organization, ensuring a cohesive final design.

The ANSI/SPARC architecture, often referred to in database design, outlines three levels of data abstraction: external, conceptual, and internal. This framework helps in understanding database models and can be expanded to include a physical model to explicitly address physical implementation details.



## The External Model

The external model represents the end users' view of the data environment. End users are those who interact with application programs to manipulate data and generate information. Typically, they operate within specific business units such as sales, finance, and marketing, each with its own constraints and data requirements. Consequently, these users view their data subsets as distinct from those of other business units. In modeling data, Entity-Relationship (ER) diagrams are used to represent these external views, known as external schemas. For instance, in a university setting, the external schemas for student registration and class scheduling are distinct but share common entities like CLASS and COURSE. These schemas illustrate the relationships and constraints relevant to each business unit, ensuring that the data model supports all operational requirements and enhances security by limiting data access to relevant subsets.

## The Conceptual Model

The conceptual model integrates all external views into a single, unified view of the entire database. This model provides a global representation of data as seen by the entire organization, known as a conceptual schema. The ER model, commonly used for this purpose, is depicted through an ER diagram (ERD), which serves as the blueprint for the database. The conceptual model offers significant advantages, such as providing a comprehensive overview of the data environment and being independent of specific software or hardware. This independence ensures that changes in technology do not affect the conceptual design, making the model adaptable to different database management systems (DBMS) and hardware configurations.

## The Internal Model

Once a specific DBMS is chosen, the internal model maps the conceptual model to the database's implementation. The internal model represents the database as viewed by the DBMS, translating the conceptual model's characteristics and constraints into the chosen implementation model's constructs. For relational databases, this involves mapping entities to tables and expressing the schema using SQL. For example, in Tiny University, the internal model would include tables for LECTURER, COURSE, CLASS, STUDENT, ENROL, and ROOM, as depicted in the internal schema. The internal model's development is crucial for hierarchical or network models requiring precise data storage and access path specifications. Although relational models simplify this process by handling data access paths transparently, specifying data storage locations is still necessary, especially in mainframe environments. The internal model is software-dependent, meaning changes in DBMS software necessitate adjustments to the model. However, it remains hardware-independent, unaffected by changes in storage devices or operating systems.

## The Physical Model

The physical model operates at the lowest level of abstraction, detailing how data is stored on physical media like disks or tapes. This model requires defining both the storage devices and access methods, making it dependent on specific software and hardware. While early data models required designers to account for physical storage details, the relational model focuses more on logical structures, reducing the need for physical-level specifications. However, fine-tuning at the physical level can enhance performance, especially in large databases, without requiring detailed knowledge of physical data storage. The physical model's dependence on DBMS and hardware means that changes in storage methods or operating systems do not affect the internal model, ensuring physical independence.

In summary, the levels of data abstraction defined by the ANSI/SPARC framework provide a structured approach to database design, from high-level conceptualization to detailed implementation. This framework facilitates the integration of diverse data views, supports software and hardware independence, and enhances database performance and security.