# Introduction to Databases and Data Management

Full text manuscript based on the lecture slides

Michael T. M. Emmerich

Faculty of Information Technology
University of Jyväskylä

Draft based on Lecture Slides

March 19, 2026

# Preface

This book grows out of a lecture series on databases and data management. The goal is simple: give a clear path from the basic idea of a database to practical SQL, solid schema design, and modern large scale data systems. The text follows the lecture order. Each chapter corresponds to one lecture set of slides. This makes it easy to connect reading, slides, lecture notes, and later online publication.

The style of this manuscript is intentionally direct. We use simple English, many examples, and careful transitions between topics. A database course can feel split into separate worlds: conceptual modeling, relational theory, SQL, design theory, and distributed systems. In practice these worlds belong together. Good database work moves back and forth between them. You model a domain, transform that model into tables, query those tables, protect correctness with constraints and transactions, and later scale or reorganize the data when the system grows.

This book is also meant to be useful online. In a WordPress version, each chapter can have links to its PDF slides, LaTeX slide source, and chapter PDF. The present LaTeX file is self contained so that it can be compiled on Overleaf without depending on external images. The same source can later be tracked in Git, versioned openly, and published in multiple formats.

The intended reader is a student who is new to databases, but the manuscript also aims to be a helpful reference for project work. We therefore explain terminology carefully, keep the main thread visible, and add small jokes in a gentle format. Databases are serious enough without sounding gloomy all the time.

# Contents

# Chapter 1

# Introduction: Databases and Data Management

**Chapter guidance**

This chapter introduces the course, the core vocabulary, and the reason databases matter. We begin with the big picture before moving to precise definitions. Read this chapter first if you want to understand what a database is, what a DBMS does, and why data management is much more than storing rows in a table.

## 1.1 Why this course matters

Almost every information system depends on stored data. A webshop stores customers and orders. A university system stores students, courses, and grades. A hospital stores patients, appointments, and treatments. A streaming service stores media, users, and recommendations. The data is often more valuable than the application code because the data captures the state of the organization and its activities.

A database matters because data has to remain useful under pressure. The system must answer queries quickly, support many users at the same time, protect confidentiality, and avoid corruption. A good database design makes future work easier. A poor design makes every feature harder.

## 1.2 Two systems with different requirements

It is tempting to think that all data systems look alike. They do not. A small personal note app and a national tax system both store data, but their requirements are very different. One may care mainly about convenience. The other must care deeply about reliability, security, auditability, and legal correctness.

This contrast already teaches an important lesson: database design depends on context. There is no single perfect design for all settings. Instead, we choose structures and technologies that fit the workload and the risks.

## 1.3 Course structure and study strategy

A database course usually combines theory and practice. The theory explains what correct structure means. The practical side shows how to build and query database systems. In this manuscript, the flow is as follows:

1. We start with the basic idea of databases and data management.
2. We learn conceptual modeling with ER diagrams.
3. We move to the relational model.
4. We learn SQL for querying and database programming.
5. We study relational algebra as a formal query language.
6. We refine schemas with normal forms.
7. We broaden the view toward warehousing, distributed data, and big data systems.

A useful way to study is to move repeatedly between concrete examples and general principles. When you meet a new term, ask both, “What does it mean?” and “Where would I use it?”

Another good habit is to keep the three levels of modeling in mind while reading the book. In chapter 2 we speak about the world in domain terms such as student, course, and invoice. In chapter 3 these ideas become relations, keys, and constraints. In chapter 4 and chapter 5 we finally ask and update real data with SQL. Many confusions disappear once you ask yourself which level you are currently working on.

The field also has a human story. File systems and ad hoc programs did useful work, but they made data sharing painful. The early database pioneers wanted something more disciplined: a way to store data once, define meaning clearly, and let many applications use the same trusted source. That motivation still explains much of modern data management.

## 1.4 A short history of information management

Before database systems became dominant, many organizations used file based systems. Each application kept its own files in its own format. This often led to duplication, inconsistency, and hard to maintain software. The relational model changed this landscape by offering a clean logical structure, a strong theory, and a flexible query language.

Later, new needs appeared. Web scale systems, analytics, semi structured data, graph data, and streaming data pushed beyond the classic setting. Still, the relational model remains central because it offers clarity, discipline, and mature tooling. Even when organizations adopt non relational systems, they continue to rely on ideas developed in database theory.

## 1.5 Key vocabulary

**Definition 1.1: Database**

A *database* is an organized collection of related data together with its intended meaning and structure.

**Definition 1.2: DBMS**

A *database management system* or *DBMS* is software that defines, stores, queries, updates, protects, and administers databases.

**Definition 1.3: Database system**

A *database system* consists of the database, the DBMS, the application programs, and the users or administrators who interact with them.

A useful distinction is between *data*, *information*, and *knowledge*. Data is raw recorded content. Information is data placed in context. Knowledge is understanding that supports action. Database systems mainly manage data, but they do so in ways that make information retrieval possible.

## 1.6 Database versus random data

Not every pile of files counts as a well managed database. A true database has structure, rules, and deliberate organization. Data is related. Meanings are known. Constraints define what is allowed. Queries return answers in a controlled way. Recovery mechanisms help after failures. Access rights restrict who may see or change what.

A spreadsheet can be useful, but it is usually not a DBMS. It lacks many of the mechanisms that serious multi user data management needs, such as concurrency control, transaction support, and schema based integrity enforcement.

## 1.7 The tasks of a DBMS

A DBMS performs many tasks at once:

- It stores data efficiently on persistent media.
- It offers data definition tools for schemas and constraints.
- It supports queries and updates.
- It manages concurrent access by many users.
- It recovers from crashes.
- It enforces security and access control.
- It provides administration tools, such as backup and performance tuning.

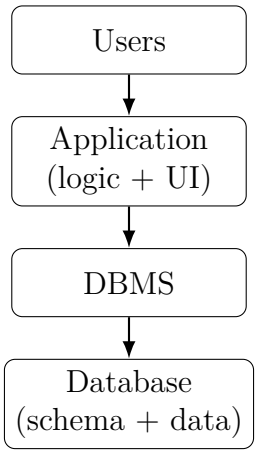
## 1.8 Levels of data models

Database thinking often distinguishes three levels. The *conceptual level* describes the domain in human terms. The *logical level* describes the formal database structure, for example with relations and keys. The *physical level* describes how data is actually stored and accessed on hardware.

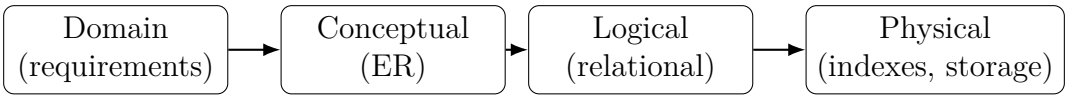
**Definition 1.4: Data model**

A *data model* is a collection of concepts used to describe the structure of data, relationships between data items, allowed constraints, and often supported operations.

The same application may have all three levels. A university domain may be modeled conceptually with students and courses, logically with tables and foreign keys, and physically with files, pages, and indexes.




A lecture figure showing the system stack that turns user requests into persistent storage operations.




This is the path followed by the course: from requirements to conceptual modeling, then to the relational schema and finally to implementation details.

## 1.9 Why SQLite is useful in teaching

SQLite is a practical teaching tool because it is lightweight, easy to install, and widely used. It supports real SQL features while avoiding heavy setup. It is not the only DBMS, but it gives students a direct way to experiment. At the same time, one must remember that commercial and enterprise systems often add richer transaction, security, and administration features.

## 1.10 Records, metadata, and data management

A *record* is a structured unit of stored information. *Metadata* is data about data, for example schema definitions, column types, or access permission information. *Data management* is the broader discipline that includes modeling, storage, querying, quality control, governance, integration, and lifecycle issues.

Data management matters because data does not stay still. It is created, changed, copied, merged, audited, archived, and reused. Good data management therefore aims not only at storage, but also at long term trustworthiness.

## 1.11 Review questions and small exercises

1. Explain the difference between a database, a DBMS, and a database system.
2. Why is a spreadsheet usually not considered a full DBMS?
3. Give one example of a conceptual, logical, and physical description for the same application.
4. Why does database design depend on context? Illustrate with two different applications.
5. What role does metadata play in data management?
6. List four important tasks of a DBMS and explain why each matters.
7. Why is SQLite useful in teaching, and what are its limits compared with enterprise systems?

## 1.12 Further reading

This chapter gives the broad entry point to the subject. For a classical and very influential treatment of database concepts and architecture, the text by Elmasri and Navathe is a standard starting point. Silberschatz, Korth, and Sudarshan give a balanced introduction with both systems and theory. Ramakrishnan and Gehrke provide many practical examples and a clean explanation of why database design choices matter.

# Bibliography

[1] R. Elmasri and S. B. Navathe. *Fundamentals of Database Systems*. Pearson.

[2] A. Silberschatz, H. F. Korth, and S. Sudarshan. *Database System Concepts*. McGraw-Hill.

[3] R. Ramakrishnan and J. Gehrke. *Database Management Systems*. McGraw-Hill.

## 1.13 Wrap-up

This chapter introduced the purpose of database systems and the vocabulary used in the rest of the book. The next chapter moves one step earlier in the design process. Before we create tables, we first model the real world that the database should represent.

# Chapter 2

# Conceptual Modeling with ER Diagrams

**Chapter guidance**

This chapter is about modeling the world before we decide on tables. It explains entities, attributes, relationships, cardinality constraints, weak entities, and extended ER features such as specialization and aggregation. Read it as the bridge between informal requirements and formal design.

## 2.1 Why conceptual modeling comes first

A database should represent a domain, not just a software convenience. If we rush directly to tables, we may hard code accidental choices too early. Conceptual modeling gives a cleaner start. It asks what kinds of objects exist, what properties they have, and how they are related.

ER modeling is especially useful when talking to domain experts. The notation is more human friendly than SQL and more precise than a paragraph of prose.

This is one reason conceptual modeling usually comes before implementation. A hospital expert may be able to tell us that a patient can have many appointments, that an appointment belongs to exactly one patient, and that some appointments may later be cancelled. That person may not care yet whether the final system uses SQLite, PostgreSQL, or a cloud platform. ER modeling lets us talk about the meaning first.

Historically, this chapter is closely connected with the work of Peter Chen, who introduced the entity relationship model in the 1970s. The lasting success of the model comes from its balance: it is simple enough for discussion, yet structured enough to guide later formal design. The next chapter will show exactly how these modeling choices are turned into relations and keys.

## 2.2 Structure of a database system

Let us briefly look how this fits into the bigger picture of a database design. At a high level, users or applications send commands to the DBMS. The DBMS interprets these

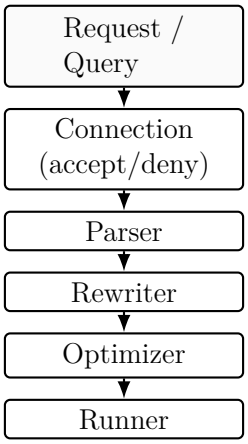



Figure 2.1: A high-level flow of a database command inside a DBMS. A request is accepted or rejected, parsed, rewritten, optimized, and then executed.

commands, checks schemas and permissions, accesses stored data, and returns results. A conceptual model does not show all internal DBMS machinery, but it helps define what the data should mean before implementation details enter the picture.

A DBMS is expected to provide more than just storage. It should support create, read, update, and delete operations, concurrent use by many users, integrity, and a degree of physical and logical data independence. In other words, the designer should be able to improve storage structures or implementation details later without changing the meaning of the data model every time. This is one reason why conceptual design should be kept separate from low-level implementation choices.

Although Figure 2.1 is simplified, it is pedagogically useful. It reminds us that the user sees queries and results, but inside the DBMS there are several layers. Conceptual modeling belongs before these inner stages. It clarifies what the data means so that later query processing, optimization, integrity checks, and storage decisions are built on a sound foundation.

## 2.3 Conceptual modelling

Conceptual modelling is an early stage of database design. At this stage we identify domain concepts, constraints, and assumptions, but we do not yet commit to one concrete implementation model. In that sense conceptual modeling is data-model independent. The goal is to distinguish essential data from non-essential detail and to make assumptions explicit.

The ER model is the classical tool for this stage. It gives a vocabulary for talking about the domain in terms of things, properties, and associations. Later, the relational model will translate these ideas into tables, keys, foreign keys, and constraints. For now, however, we focus on understanding the real world that the database should describe.

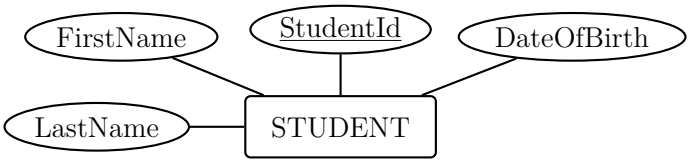



Figure 2.2: A basic ER fragment with one entity set, several attributes, and an underlined key attribute.

## 2.4 Entity sets and attributes

Roughly speaking, in the ER diagram, entities refer to objects of the real world. In an entity set they are of the same type and they share the same attributes, unless inheritance is considered. Relationship sets contain relationships between entities, whereas the relationships in a relationship set connect the same entities.

**Definition 2.1: Entity and entity set**

An *entity* is a distinguishable object in the domain of discourse. An *entity set* is a collection of similar entities, such as all students or all courses. Entities of the same entity set have the same attributes (see next definition).

**Definition 2.2: Attribute**

An *attribute* is a property used to describe an entity or relationship, such as a student's name or a course's credit count.

For example, in a student information system, `STUDENT` may be an entity set with attributes `StudentId`, `FirstName`, `LastName`, and `DateOfBirth`. Figure 2.2 shows this classic Chen style pattern: a rectangle for the entity set, ovals for attributes, and an underlined key attribute.

The figure also illustrates a practical point: an ER diagram should use names from the domain, not from accidental software choices. A domain expert naturally understands words such as `STUDENT`, `COURSE`, and `CompletionDate`. The later database implementation may rename, abbreviate, or encode things differently, but good conceptual models keep the meaning visible.

## 2.5 Key attributes

**Definition 2.3: Key attribute**

A *key attribute* is an attribute whose values uniquely identify entities within an entity set. In the ER diagram it is underlined. Sometimes, multiple attributes in combination form a key. Then they are all underlined in the ER diagram. The combination is called a composite key.

Names often do not work as keys because they are not guaranteed to be unique. A student number is usually a better choice. In ER diagrams, a key attribute is underlined.

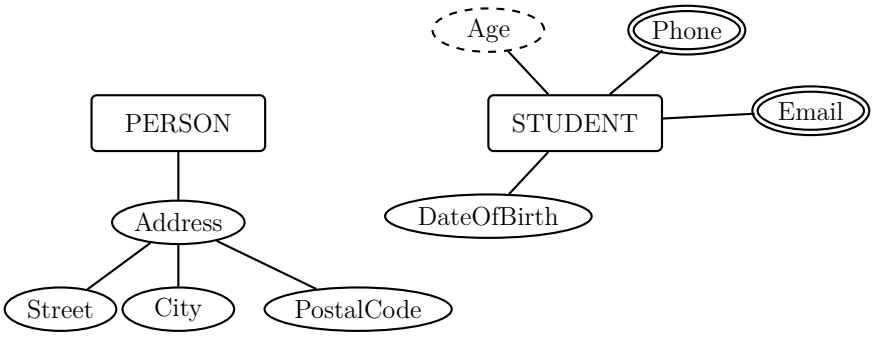



Figure 2.3: Composite, derived, and multivalued attributes in ER notation. Address is composite, Age is derived, and Phone and Email are multivalued.

Every entity set has exactly one key or composite key in an ER diagram. The key needs to be carefully chosen. For instance, `Name` alone might not be a good choice because there might be two persons with the same name in the database, whereas the combination of `Name` and `BirthDate` as a composite key might work better.

A practical warning is that safe key management can become difficult in large-scale databases. A theoretically unique attribute is not always a practically good key. Good keys should be stable, easy to manage, and hard to confuse. This is why designers often prefer identifiers such as `StudentId` rather than names alone. The problem is not merely technical. A poor key choice creates errors in identity, merging, deduplication, and updates.

## 2.6 Composite, derived, and multivalued attributes

Some attributes have internal structure. An address can be split into street, postal code, and city. Such an attribute is called *composite*. Some attributes can be derived from others. Age can be derived from date of birth. Such an attribute is *derived*. Some attributes can have several values for one entity, such as multiple phone numbers. These are *multivalued attributes*.

These distinctions matter because not every concept should be stored in the same way. Derived values, for example, are often better computed than stored. Multivalued attributes are also a warning sign. They often indicate that the later relational design may need a separate relation instead of trying to pack several values into one field. Figure 2.3 links the three ideas directly to their ER notation.

A useful modeling habit is to ask what should really be stored and what can be computed when needed. Age changes over time, while date of birth does not. That makes `DateOfBirth` the more stable fact. Similarly, a multivalued attribute may be acceptable in a conceptual diagram, but it should make the designer alert. In later chapters we will see why repeating values fit badly into flat relations.

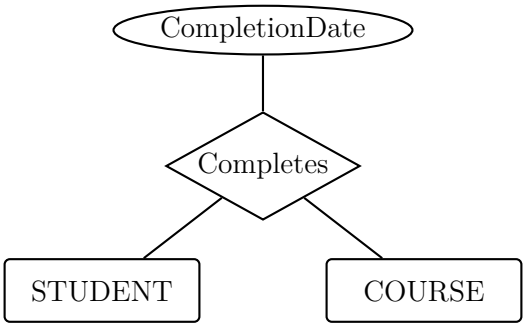



Figure 2.4: A relationship set may have attributes of its own. Here CompletionDate belongs to the relationship Completes rather than to STUDENT or COURSE alone.

## 2.7 Relationship sets

**Definition 2.4: Relationship set**

A *relationship set* describes associations among entities, such as a student enrolling in a course or an employee working in a department.

A binary relationship connects two entity sets. A ternary relationship connects three. Ternary relationships should not be replaced casually by several binary ones, because meaning may be lost. For example, the fact that a teacher evaluates a student in a course is naturally ternary.

Two practical details are important here. First, relationship sets are often named by verb phrases, because they describe how entities are associated. Second, a relationship set may itself have attributes. For example, if a student completes a course, then `CompletionDate` belongs naturally to the relationship rather than to the student or the course alone. In this way a relationship set can be understood as a set of tuples such as $(s, c, d)$, where $s \in$ `STUDENT`, $c \in$ `COURSE`, and $d$ belongs to a suitable date domain.

Figure 2.4 is important because it prevents a common modeling mistake. If `CompletionDate` were attached to `STUDENT`, it would suggest one completion date per student. If it were attached to `COURSE`, it would suggest one completion date per course. Neither expresses the intended meaning. The attribute belongs to the relationship instance.

## 2.8 Degree of a relationship

The *degree* of a relationship is the number of participating entity sets. Unary relationships connect an entity set to itself. Binary relationships connect two entity sets. Ternary relationships connect three. The degree matters because it affects both meaning and later translation into relations.

Figure 2.5 shows a unary relationship with role names. Such labels are helpful because one entity set can participate in more than one role. Figure 2.6 shows a ternary relationship. A ternary relationship is not, in general, equivalent to three binary ones. If we split the ternary fact into separate binary associations, we may lose which product

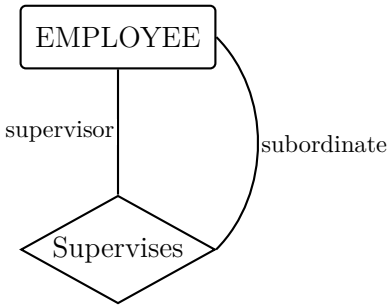



Figure 2.5: A unary relationship. EMPLOYEE participates twice in Supervises, once in the role supervisor and once in the role subordinate.

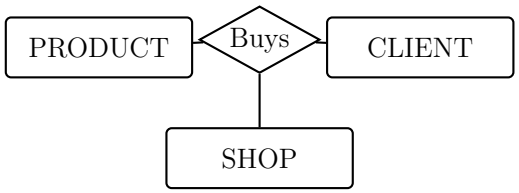



Figure 2.6: A ternary relationship. The fact that a client buys a product in a particular shop is not captured by separate binary relationships alone.

was bought by which client in which shop. If the domain really requires the joint fact, the ternary relationship is the honest model.

## 2.9 Degree and participation

Participation constraints describe whether participation is optional or mandatory, while cardinality constraints specify how often an entity may take part in a relationship. For example, if the participation of entity set $A$ in a relationship set $R$ has cardinality $(N, M)$, then each entity in $A$ must participate in at least $N$ and at most $M$ relationship instances of $R$. Together, these constraints capture patterns such as one-to-one, one-to-many, and many-to-many.

These constraints are important because they later influence the relational design. A one-to-many relationship is translated differently from a many-to-many relationship. The worked example in figure 2.13 makes these cardinalities visible instead of leaving them as prose alone.

A helpful practical formulation is this: ask how many times an entity from a given entity set must participate, and how many times it may participate, in the relationship set. This simple question often resolves modeling uncertainty. The minimum is often 0 or 1, and the maximum is often 1 or $N$, but more specific ranges are also possible.

## 2.10 Cardinality: min and max participation

Cardinality expresses *how many* entities can or must participate in a relationship. We usually place a minimum and maximum near each participating entity set. The minimum tells us whether participation is optional or mandatory. The maximum tells us whether one or many relationship instances are possible.

Figures 2.7 and 2.8 illustrate why minimum and maximum both matter. In the first case, a person may own no car at all, but a car must have an owner. In the second case, the maximum is not merely "many." It is exactly 11. This reminds us that cardinalities can capture real business or domain rules, not just generic one-to-many patterns.


Figure 2.7: A cardinality example. A person may own zero to many cars, while each car has exactly one owner.


Figure 2.8: A more domain-specific example. A team has exactly 11 players, while a player belongs to at most one team.

## 2.11 Different cardinality notations

In practice, several notations are used for cardinality constraints. Some use labels such as `1` and `N`. Others use min–max pairs such as `0..1`, `1..*`, or more generally `N..M`. The notation differs, but the modeling idea is the same: specify the minimum and maximum number of relationship instances in which an entity may or must participate.

Several traditions coexist, including Chen notation, UML style notations, and crow's foot notation. It is therefore less important to memorize one visual convention than to understand the semantics underneath. Whatever notation we use, we should be consistent within one book, course, or project.

From a mathematical point of view, these constraints correspond to ideas such as one-to-one, many-to-one, and many-to-many. But the ER notation is often more expressive because it can show both the minimum and the maximum. This is why the practical question quoted above is so helpful: *How many times must, and how many times can, an entity from this entity set participate in the relationship set?*

## 2.12 Weak entity sets

**Definition 2.5: Weak entity set**

A *weak entity set* is an entity set whose entities cannot be uniquely identified by their own attributes alone and therefore depend on another entity set for identification.

A classic example is `DEPENDENT` in an insurance or HR system. A dependent may be identified only relative to an employee, for example by `EmployeeId` together with `DependentName`. The identifying relationship is essential here. Figure 2.9 shows the visual cues: a double rectangle for the weak entity and a double diamond for the identifying relationship.

There are two good reasons for this modeling pattern. First, key management becomes easier. Second, data consistency is enforced: when the owning entity disappears, the dependent entity usually disappears as well. This is exactly the kind of semantic dependency that a good conceptual model should show clearly. The maximum cardinality on the weak side is often 1 in the identifying relationship, because the weak entity is identified through one owning entity.

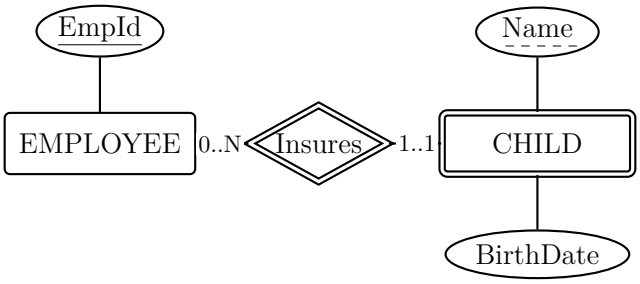



Figure 2.9: A weak entity example. CHILD is identified only together with its owning EMPLOYEE through the identifying relationship Insures.

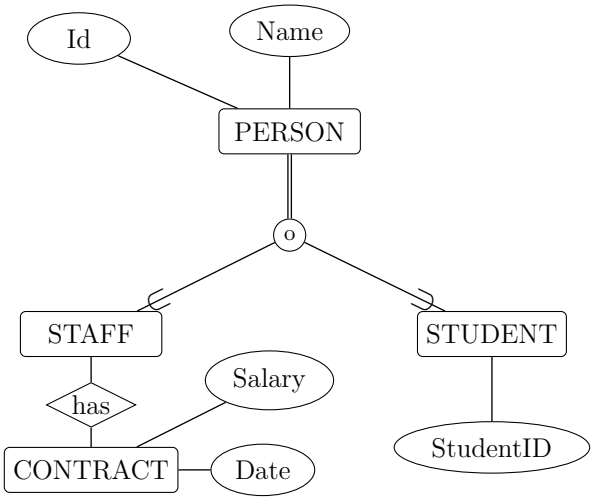



Figure 2.10: An EER specialization example. The supertype PERSON is specialized into STAFF and STUDENT. The label *o* indicates overlapping subtypes, and the double line indicates total coverage.

## 2.13 Extended ER modeling

Basic ER notation can be extended. Two important extensions are specialization and generalization.

**Definition 2.6: Specialization and generalization**

*Specialization* divides a broad entity set into more specific sub entity sets. *Generalization* goes in the opposite direction by recognizing a common supertype.

For example, `PERSON` may be specialized into `STUDENT` and `STAFF`. The designer may also specify whether the subtypes are disjoint or overlapping and whether coverage is total or partial.

Figure 2.10 follows a common EER notation. The circle label records whether subtype membership is *disjoint* or *overlapping*. The line from the supertype to the specialization circle records whether coverage is *partial* or *total*. The rotated ∪ marks

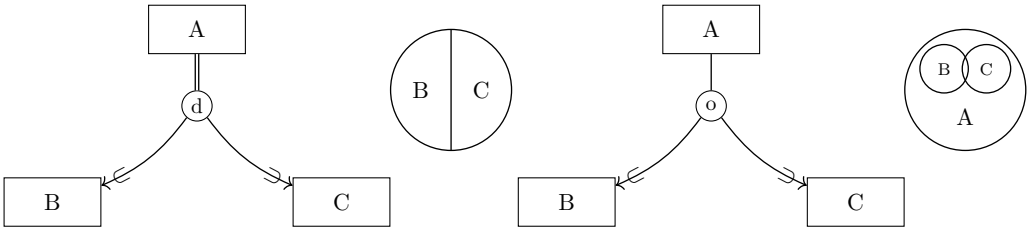



Figure 2.11: Subtype constraints in EER modeling. On the left, the specialization is disjoint and total. On the right, it is overlapping and partial.

on the child links visually indicate subtype branches. The point of this notation is not ornament. It makes semantic assumptions explicit.

## 2.14 Disjoint versus overlapping, total versus partial

These distinctions deserve separate emphasis, because they are easy to overlook and very important for correct design.

- **Disjoint** means that an entity belongs to at most one subtype.
- **Overlapping** means that an entity may belong to more than one subtype.
- **Total** means that every entity in the supertype belongs to at least one subtype.
- **Partial** means that some supertype entities may belong to no subtype at all.

Figure 2.11 is especially useful because students often remember the words but forget their interaction. Disjoint or overlapping tells us whether subtype memberships can coexist. Total or partial tells us whether every supertype entity must be covered. These are independent design choices. For example, a person can be both `STUDENT` and `STAFF`, which would be overlapping. But it may still be false that every person in the model belongs to one of those subtypes, which would make coverage partial.

## 2.15 Aggregation

**Definition 2.7: Aggregation**

*Aggregation* treats a relationship, together with its participating entities, as a higher level conceptual unit that can itself participate in another relationship.

Aggregation is useful when a relationship itself must be connected to something else. It helps keep the model readable when the real world contains layered interactions. In figure 2.12, the relationship `Wrote` is treated as one conceptual unit so that a reviewer can be linked to the whole act of a student writing a homework assignment.

This is often easier to understand than flattening everything into one large relationship immediately. Aggregation says that the interaction itself matters. In the example, a reviewer does not simply review a student, nor simply a homework assignment. The

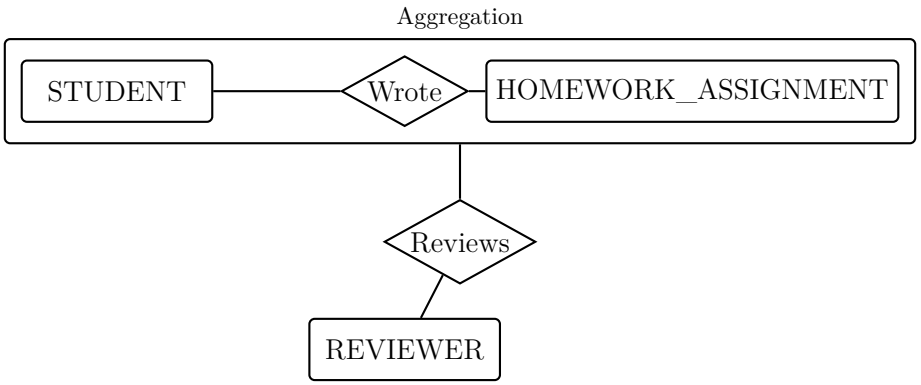



Figure 2.12: Aggregation lets a relationship participate in another relationship. The review concerns the whole writing event, not just one isolated entity.

reviewer evaluates the event in which that student wrote that assignment. Aggregation makes this semantics visible.

## 2.16 Phases of conceptual modeling

Conceptual modeling is not a single shot activity. A common workflow is this:

1. Read and discuss the requirements.
2. Identify candidate entity sets.
3. Identify attributes and likely keys.
4. Identify relationships and constraints.
5. Refine naming and remove ambiguity.
6. Review the model with stakeholders.

This process is iterative. One does not get the model perfect on the first pass. It is normal to refactor: add missing entities, rename unclear relationships, revise cardinalities, and decide whether some apparent attributes are in fact entities of their own. A practical tip from modeling work is to write down constraints that are hard to express in the diagram. A diagram is powerful, but not every business rule fits neatly inside it.

## 2.17 Worked mini example

Suppose we model a small university registration domain. We identify `STUDENT`, `COURSE`, and `TEACHER`. We add a many to many relationship `ENROLLED_IN` between students and courses. We also add a one to many relationship `TEACHES` from teacher to course. If we later need grades awarded by a specific teacher in a specific course to a specific student, a ternary relationship may be more natural than three separate binary ones.

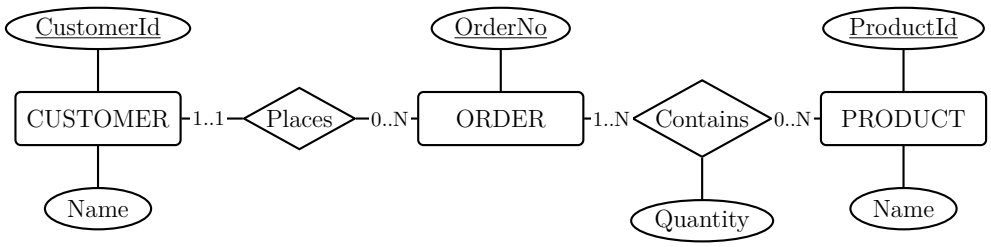



Figure 2.13: A worked ER mini example with explicit cardinalities and a relationship attribute.

A compact order-oriented example is also helpful because it shows clearly how requirements can be translated into an ER model. We start with a textual description: customers have identifiers and names, products have identifiers and prices, orders have order numbers, and an order can contain several products with quantities. From this description we identify three entity sets, two relationship sets, and one relationship attribute. A compact example is shown in figure 2.13; it is useful because the cardinalities already hint at how the later relational tables will be formed.

This example is good training because it forces us to make modeling choices explicit. Is `Quantity` an attribute of `PRODUCT`? No, because different orders may contain the same product with different quantities. Is `ORDER` an entity in its own right? Yes, because it has its own identity, here `OrderNo`. These are exactly the kinds of decisions that conceptual modeling is meant to clarify.

## 2.18 Checklist for finalizing an ER model

When finishing an ER model, ask the following questions in plain language:

- Does every important object in the domain appear?
- Are keys sensible and stable?
- Are cardinalities correct?
- Are weak entities really weak?
- Are subtype constraints explicit?
- Are hidden business rules written down somewhere?

Some further useful checks are these. Are entities and relationships named clearly from the domain perspective? Are there attributes that would be better modeled as separate entities? Are there derived attributes that should not be stored? Would EER abstraction reduce duplication and improve clarity? This sort of checklist is valuable because conceptual errors become expensive later.

## 2.19 A short historical note

Peter Chen's ER model became influential because it offered a strong visual language for conceptual design. It helped connect the world of users and domain experts with the later formal world of relations and constraints.

One can think of the ER model as a language of meaningful building blocks. Entities are the semantic building blocks, relationships describe how they combine, and attributes and constraints add further meaning. This is one reason the ER model has remained pedagogically powerful for decades.

## 2.20 Review questions and small exercises

1. What is the purpose of conceptual modeling before table design begins?
2. Explain the difference between an entity type and an entity instance.
3. What is the difference between a simple attribute and a composite attribute?
4. Give an example of a weak entity and explain why it is weak.
5. What does cardinality mean in a relationship? Give examples of 1:1, 1:N, and M:N.
6. Why are keys already important at the ER level?
7. Draw or describe an ER model for a library with books, members, and loans.

## 2.21 Further reading

For ER modeling, the original paper by Peter Chen is historically essential because it introduced the central ideas in a clear way. Modern textbooks then expand these ideas with richer notation, specialization, aggregation, and design guidance. This chapter is best followed by reading one classical source and one modern textbook treatment.

# Bibliography

[1] P. P. Chen. The Entity-Relationship Model: Toward a Unified View of Data. *ACM Transactions on Database Systems*, 1(1), 1976.

[2] R. Elmasri and S. B. Navathe. *Fundamentals of Database Systems*. Pearson.

[3] T. Teorey, S. Lightstone, T. Nadeau, and H. V. Jagadish. *Database Modeling and Design*. Morgan Kaufmann.

## 2.22 Wrap-up

Conceptual modeling gives the structure of the domain in a form humans can discuss. It is deliberately implementation independent, but it is not vague. A good ER model records entities, attributes, relationships, keys, cardinalities, weak entities, subtype structures, and aggregation in a way that can later be translated into a relational design. In the next chapter we move from ER thinking to the relational model, where the design becomes more formal and directly implementable.

# Chapter 3

# The Relational Model and the Transformation from ER

**Chapter guidance**

This chapter introduces relations, tuples, attributes, keys, foreign keys, and integrity constraints. It also explains how an ER model is transformed into a relational schema. Read it as the formal foundation that prepares us for SQL and relational algebra.

## 3.1 Basic structure of the relational model

The relational model was proposed by Edgar F. Codd in 1970. That historical fact matters because it explains the spirit of the model. Codd wanted data descriptions that were logically clean and independent from the low level storage tricks used by particular programs. In other words, the relational model is not merely a table drawing style. It is a proposal for how to think clearly about data.

When students first see relations, they often think only of spreadsheet-like tables. That is a useful starting image, but a relation is more disciplined than an ordinary sheet. Attribute names matter, keys matter, and integrity constraints matter. The purpose of this discipline becomes very practical in later chapters. It supports querying in chapter 4, safe updates in chapter 5, and schema refinement in chapter 7.

**Definition 3.1: Relation**

A *relation* is a set of tuples over a fixed set of attributes. In everyday practice it is often displayed as a table with rows and columns.

**Definition 3.2: Tuple and attribute**

A *tuple* is one row of a relation. An *attribute* is one named column.

The relational model is a logical level model. It tells us what is stored and how it is structured, not exactly how bytes sit on disk. A good way to think about it is this: the

| SupplierID | Name | City | FoundedYear |
|---|---|---|---|
| 31 | Worchware | Kaeli | 2003 |
| 42 | Fycbow | Shimo-carnia | 1993 |
| 33 | Myjice | Yarghe-pfuhar | 2008 |
| 54 | Bivoke | Lucan | 1986 |
| 25 | Vakoj | Idgipurn | 2000 |

Figure 3.1: An example relation shown in table form. The relational model is often visualized this way, although it carries stricter semantics than an ordinary spreadsheet.

relational model describes meaning and structure, while storage structures and access paths belong to a lower implementation level.

## 3.2 A more formal view of a relation

A relation can be described a little more formally as a pair $\langle H, r \rangle$, where $H$ is the header and $r$ is the body. The header contains the attribute names together with their intended domains, and the body is the set of tuples currently present in the relation.

This formal view explains why a relation is more than a rectangle on a screen. The header tells us what each position means. The body tells us which facts are currently stored. In later chapters, this separation becomes important when we distinguish between a schema and a current database state.

## 3.3 Example relation

Consider a relation `SUPPLIER(SupplierID, Name, City, FoundedYear)`. Each tuple represents one supplier. The attribute names describe the meaning of the components. A relation instance is the current set of tuples stored at some time.

Figure 3.1 is helpful as a first picture. We see one relation name, four attributes, and several tuples. But the formal reading is slightly richer than the visual one. The rows are facts, the columns are named attributes, and the relation as a whole is a set of such tuples.

## 3.4 Core definitions and auxiliary terms

A *schema* describes the name of a relation and its attributes. A *domain* is the set of allowed values for an attribute. A *relation instance* is the actual content currently stored. A *database schema* is a collection of relation schemas together with integrity constraints.

Two further terms are useful. The *degree* of a relation is the number of attributes in its schema. The *cardinality* of a relation is the number of tuples currently stored in it. These two notions are easy to confuse, so it is worth pausing over them. Degree concerns the width of the relation. Cardinality concerns the number of rows.

For example, the relation `SUPPLIER(SupplierID, Name, City, FoundedYear)` has degree 4. If it currently stores five tuples, then its cardinality is 5. One may also speak informally of the cardinality of an attribute in the sense of the number of distinct values

currently appearing in that column, but the primary relational usage of cardinality refers to the number of tuples in the relation.

## 3.5 A worked example: the DELIVERY relation

A useful running example is

$$\texttt{DELIVERY}(\texttt{supplier\_no}, \texttt{part\_no}, \texttt{project\_no}, \texttt{quantity}).$$

Here the schema has four attributes, so the degree of the relation is 4. A tuple such as

$$(1,\ 2,\ \texttt{p123},\ 17)$$

states that supplier 1 delivers part 2 to project `p123` in quantity 17.

This example is useful because it already hints at keys and references. The attributes `supplier_no`, `part_no`, and `project_no` will naturally connect to other relations such as `SUPPLIER`, `PART`, and `PROJECT`. The attribute `quantity` is different. It is not an identifier but a descriptive fact about one delivery.

## 3.6 NULL values

SQL uses the special marker `NULL` to represent missing, unknown, or inapplicable information. This is practical, but it must be handled with care because it does not behave like an ordinary value. Later in SQL we will see that tests involving `NULL` require special syntax.

It is important not to confuse `NULL` with the value 0. If the quantity in a delivery tuple is 0, then the quantity is exactly zero. If the quantity is `NULL`, then the quantity is missing, unknown, or not recorded. These are different meanings. Treating `NULL` as zero can change aggregates, comparisons, and logical conclusions.

For example,

$$(1,\ 2,\ \texttt{p123},\ \texttt{NULL}) \qquad \text{and} \qquad (1,\ 2,\ \texttt{p123},\ 0)$$

are not equivalent. The first says that the quantity is not known from the stored data. The second says that the quantity is known and equals zero.

## 3.7 Core properties of relations

In the pure relational model, a relation is a set. This means duplicate tuples are not conceptually allowed. The order of rows and columns is not part of the meaning. Real SQL systems sometimes relax these ideas in practical ways, but the clean mathematical picture is still important.

More precisely, a relation should satisfy the following basic ideas.

1. Every tuple has one value for each attribute in the schema.
2. The order of tuples does not matter.

3. Duplicate tuples are not allowed in the pure model.
4. The order of attributes is not part of the meaning.
5. Attribute names should clearly indicate the meaning of the stored values.

These rules explain why a relation is not just any grid of cells. If a table uses row position as part of the meaning, or if it contains repeated identical rows, or if its columns have unclear names, then it is drifting away from the clean relational ideal.

## 3.8 Why badly structured relations cause trouble

A bad relation may mix several meanings into one table, repeat the same facts many times, or store multiple values in one field. Such designs make updates error prone. One goal of database design is to keep relations disciplined and easy to maintain.

For instance, a badly designed table might rely on the first row being "the best supplier" and the second row being "the second best supplier." That would make row order part of the meaning, which is not relational. The proper relational fix would be to add an explicit attribute such as `Rank`. Similarly, if a table stores several phone numbers in one field separated by commas, then one attribute cell is no longer atomic in the intended sense. Later chapters on normalization will show why such choices create anomalies.

## 3.9 Integrity constraints

**Definition 3.3: Integrity constraint**

An *integrity constraint* is a rule that specifies which database states are allowed.

Important kinds of integrity constraints include domain constraints, key constraints, entity integrity, and referential integrity.

Integrity constraints are important because a database should not merely store values. It should store only those states that make sense in the modeled world. If a salary becomes negative, or a product reference points to a non-existing product, the stored data no longer describes a valid state of the domain.

## 3.10 Primary keys and candidate keys

**Definition 3.4: Superkey, candidate key, primary key**

A *superkey* is a set of attributes that uniquely identifies tuples. A *candidate key* is a minimal superkey. A *primary key* is the candidate key chosen as the main identifier of the relation.

For example, in a student registry, a student number may be chosen as the primary key. A national identifier might also be unique, but policy or privacy reasons may make it a poor primary key in practice.

It helps to distinguish the three notions carefully. A superkey guarantees uniqueness, but it may contain unnecessary extra attributes. A candidate key is a superkey with no unnecessary part. A primary key is the candidate key that the designer chooses as the main identifier.

Consider

$$\texttt{STUDENT}(\underline{std\_no}, ssn, first\_name, last\_name, postal\_code, city).$$

In many settings, both {`std_no`} and {`ssn`} might be candidate keys. But {`std_no`, `ssn`} would not be a candidate key, because it is not minimal. It is unique, but it contains an unnecessary extra attribute.

A good primary key should not only be unique. It should also be stable, always present, easy to manage, and small enough to work well as a reference. This is why artificial identifiers such as sequential numbers, UUIDs, or similar system-generated values are often used in practice.

## 3.11 Entity integrity

A primary key plays a special role in the relation. Because it is the chosen identifier, its value must always be present and must uniquely identify a tuple. This idea is often summarized as *entity integrity*: primary key values must not be `NULL`.

This is logically natural. If the primary key is missing, then the tuple lacks the very identifier that is supposed to distinguish it from all other tuples. By contrast, other non-key attributes may in some situations be unknown or absent.

## 3.12 Foreign keys

**Definition 3.5: Foreign key**

A *foreign key* is an attribute set in one relation whose values refer to a candidate key or primary key in another relation.

Foreign keys connect relations and preserve cross table consistency. If an invoice line refers to a product, the corresponding product should exist.

A foreign key does not need to be unique. Many tuples may refer to the same target tuple. That is exactly what happens in one-to-many structures. For example, many employees may refer to the same department, or many invoice lines may refer to the same product.

Figure 3.2 shows a common pattern. The relation `DELIVERY` represents an association among three other relations. Its key is composite, and each part of that key also acts as a foreign key. This kind of structure appears frequently when many-to-many or higher-arity associations are translated into the relational model.

| |
|---|
| SUPPLIER(supplier_no, name, city) |
| PART(part_no, name) |
| PROJECT(project_no, name, budget) |
| DELIVERY(supplier_no,part_no,project_no, quantity) |
| DELIVERY.supplier_no → SUPPLIER.supplier_no |
| DELIVERY.part_no → PART.part_no |
| DELIVERY.project_no → PROJECT.project_no |

Figure 3.2: An example schema with primary keys and foreign keys. The relation DELIVERY refers to SUPPLIER, PART, and PROJECT.

## 3.13 Referential integrity

Foreign keys lead naturally to the idea of *referential integrity*. If a tuple in one relation refers to a tuple in another, then the referenced tuple should exist. Otherwise the reference points nowhere.

This simple idea has practical consequences. If a referenced tuple is updated or deleted, the DBMS must decide what to do with dependent tuples. Common policies include restricting the change, cascading it, or setting the foreign key to `NULL` if that is allowed. The important point here is conceptual: foreign keys are not just convenient columns. They are constraints that preserve consistency across relations.

## 3.14 Other integrity constraints

Not all rules are keys. Some rules restrict ranges, such as salary being nonnegative. Others express enumerated possibilities, such as an invoice status being one of a small fixed set. Still others are more involved business rules that may need extra logic.

Examples include:

- a phone number must start with a given country code,
- a manager must be an employee of the same department,
- a salary must not exceed a permitted range,
- an attribute may be declared `NOT NULL`.

These examples matter because they show that good schema design is not only about tables and keys. Constraints are part of the meaning of the database.

## 3.15 Atomic values and a preview of normalization

In standard relational design, attribute values are usually treated as atomic. That means one attribute position is intended to contain one value of the appropriate kind, not an internal list or nested structure. This assumption does much of the quiet work behind clean design.

If a relation violates this expectation, it may be poorly structured or unnormalized. Often such relations can be improved by decomposition into better relations. We will

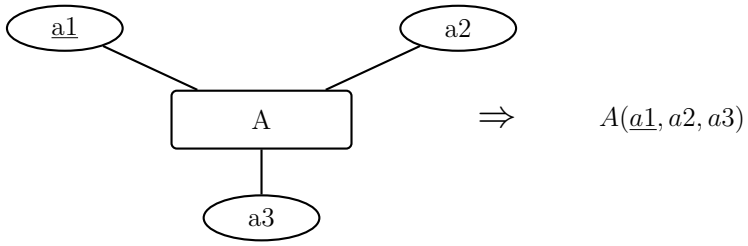



Figure 3.3: Transformation of a strong entity set into a relation. The key attribute becomes the primary key of the resulting relation.

return to that topic in the normalization chapter. For now, it is enough to remember that the relational model prefers clear, well-typed, and atomic attribute values.

## 3.16 Transformation from ER to relations

A key part of database design is turning a conceptual ER model into a relational schema. The basic rules are systematic.

The general strategy is straightforward. We first transform entity sets, then deal with attributes, then transform relationship sets, and finally handle EER structures such as specialization. In practice, several rules may interact, so schema design is partly systematic and partly judgment based.

### 3.16.1 Strong entity sets

Each strong entity set becomes a relation. Its simple attributes become columns. The chosen key becomes the primary key. Composite attributes are usually broken into their meaningful components before the table is created. Derived attributes are often not stored at all if they can be recomputed safely.

Figure 3.3 captures the simplest and most common mapping step. A strong entity set with ordinary attributes becomes one relation with the same essential information.

### 3.16.2 Weak entity sets

A weak entity set becomes a relation that includes its partial key together with the key of the owner entity set. Together they form the primary key. This mirrors the ER idea from figure 2.9: the weak entity does not have a complete identity without its owner.

The corresponding relational schema is

$$\texttt{EMPLOYEE}(\underline{\texttt{EmpId}}, \texttt{EmpName})$$

and

$$\texttt{CHILD}(\underline{\texttt{EmpId}}, \underline{\texttt{ChildName}}, \texttt{BirthDate}),$$

with $\texttt{CHILD.EmpId} \rightarrow \texttt{EMPLOYEE.EmpId}$.

This is a good example of how conceptual meaning survives translation. The weak entity does not become independent during mapping. Its dependence is preserved in the composite primary key and the foreign key.

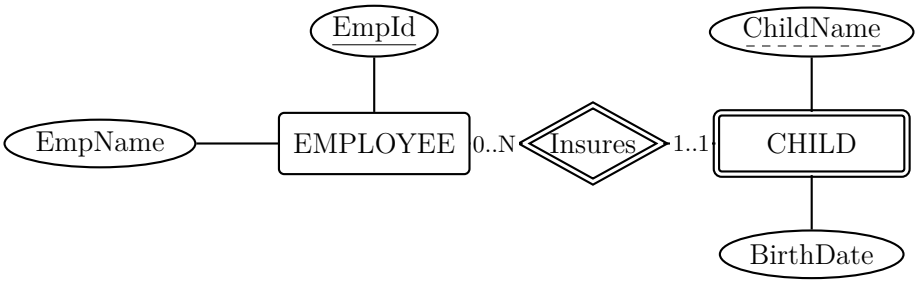



Figure 3.4: A weak entity and its identifying relationship. The weak entity CHILD is identified relative to EMPLOYEE.

### 3.16.3 One to one relationships

A one to one relationship can often be represented by placing a foreign key on one side, usually the side with total participation or the side that leads to a cleaner design. If the relationship has its own descriptive attributes or if future evolution is likely, a separate relation can also be justified.

The reason is simple. A one-to-one relationship does not automatically need its own relation if one of the existing relations can store the reference cleanly without introducing repetition or ambiguity. But when the relationship itself has attributes, or when optional participation and later changes matter, a separate relation may be the clearer choice.

### 3.16.4 One to many relationships

A one to many relationship is typically represented by placing a foreign key on the many side. This is one of the most important mapping rules in practice because it appears constantly in operational schemas. For example, if one department has many employees, then `Employee` receives a foreign key that points to `Department`.

A common intuition is that the many side is where the repeated references naturally occur. Many employees may belong to one department, so the employee tuples carry the department identifier. This gives a compact design and avoids creating an unnecessary extra relation.

### 3.16.5 Many to many relationships

A many to many relationship becomes a new relation whose primary key often consists of the primary keys of the participating relations. Additional relationship attributes are stored there as well. Thus an ER relationship such as `ENROLLED_IN` between `STUDENT` and `COURSE` naturally becomes its own table, perhaps with extra attributes such as grade or enrollment date.

From Figure 3.5 we obtain a schema such as

$$\texttt{STUDENT}(\underline{\texttt{StudId}}, \ldots), \qquad \texttt{COURSE}(\underline{\texttt{CourseId}}, \ldots),$$

$$\texttt{EnrolledIn}(\underline{\texttt{StudId}}, \underline{\texttt{CourseId}}, \ldots),$$

with foreign keys from `EnrolledIn` to `STUDENT` and `COURSE`. This is one of the standard relational patterns and appears constantly in practice.


Figure 3.5: A many-to-many relationship between STUDENT and COURSE. This kind of relationship is mapped to its own relation in the relational model.


Figure 3.6: A one-to-many relationship. The identifier of PERSON can be stored as a foreign key in CAR.


Figure 3.7: A relationship with maximum cardinality one on the PLAYER side. Such structures often allow a compact foreign-key-based mapping.

### 3.16.6 Relationship cardinality and mapping choices

The mapping of a relationship depends strongly on its maximum cardinalities. A many-to-many relationship usually becomes its own relation. A one-to-many relationship is usually absorbed by a foreign key on the many side. A one-to-one relationship leaves more freedom.

Minimum cardinalities are also semantically important, but they are not always captured directly by the table structure alone. For example, the fact that every car must have an owner is a minimum participation requirement. Such requirements may need explicit integrity constraints in addition to the basic foreign key structure.

### 3.16.7 Transforming attributes

Simple attributes usually become columns in the obvious way. Composite attributes are decomposed into their components. Multivalued attributes are moved into separate relations because one row should not contain an unbounded set of repeated values. Derived attributes are often omitted from storage if they can be recomputed from base data.

This gives a useful design rule: store what is fundamental, derive what is safely derivable.

Figure 3.8 shows why multivalued attributes usually lead to separate relations. The key of the owner entity is repeated, and the repeated values become tuples of their own. This keeps the resulting design relational and avoids packing lists into one cell.

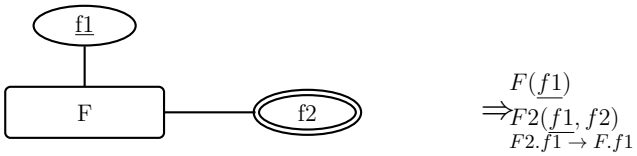



Figure 3.8: Transformation of a multivalued attribute. The multivalued attribute is moved into a separate relation together with the owner's key.

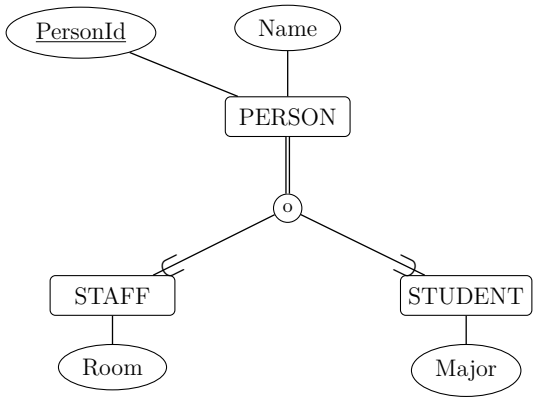



Figure 3.9: An EER hierarchy that can be translated in several reasonable ways, depending on whether we emphasize clarity, compactness, or avoidance of NULL values.

### 3.16.8 Specialization and generalization

There is no single mandatory translation for inheritance structures. Common strategies include one relation for the supertype and one for each subtype, one relation for each subtype carrying all inherited attributes, one relation for the whole hierarchy with type information, or related alternatives. The best choice depends on constraints, redundancy, and expected queries.

A practical rule of thumb is this. If subtype specific attributes are many and the subtype distinction matters semantically, separate subtype tables are often clearer. If the hierarchy is shallow and queries almost always need all data together, one table with type information may be simpler. As in the ER chapter, the conceptual choice and the physical design choice are related but not identical.

For the structure in Figure 3.9, two common mappings are:

$$\texttt{PERSON}(\underline{\texttt{PersonId}}, \texttt{Name}), \quad \texttt{STAFF}(\underline{\texttt{PersonId}}, \texttt{Room}), \quad \texttt{STUDENT}(\underline{\texttt{PersonId}}, \texttt{Major}),$$

with foreign keys from `STAFF` and `STUDENT` to `PERSON`, or a single-table design such as

$$\texttt{PERSON}(\underline{\texttt{PersonId}}, \texttt{Name}, \texttt{Room}, \texttt{Major}, \texttt{IsStaff}, \texttt{IsStudent}).$$

The first mapping is usually cleaner semantically and avoids unnecessary `NULL` values. The second can be simpler for some applications, especially when the hierarchy is small and data is often accessed together. But it may introduce many empty fields.

## 3.17 Choosing among several transformations

Real ER diagrams often trigger several rules at once, and sometimes more than one transformation is possible. In such cases it is useful to compare the alternatives from the database point of view.

Questions worth asking include:

- Does the design avoid unnecessary repetition?
- Does it avoid many avoidable `NULL` values?

- Are keys and foreign keys clear?
- Does the schema match the important queries and updates?
- Is the resulting structure easy to understand and maintain?

One robust fallback is to transform a relationship into its own relation. This always works for many situations, although it may later need normalization. Likewise, the "one supertype relation plus one relation per subtype" strategy is widely applicable for EER structures, even if it is not always the most compact.

## 3.18 Key design principle

Use keys to enforce identity cleanly and avoid redundancy. This principle sounds small, but it has large consequences. Once identity is badly handled, updates, joins, and constraints all become harder.

A good relational design is therefore not merely a collection of tables. It is a disciplined structure in which tuples can be identified, related, and constrained without ambiguity.

## 3.19 Review questions and small exercises

1. What is a relation in the relational model?
2. What is the difference between a schema and an instance?
3. Explain the role of domains in relational databases.
4. What is a superkey, and how does it differ from a candidate key?
5. Why do foreign keys matter when transforming ER diagrams into tables?
6. Describe a standard mapping of a 1:N relationship from ER to the relational model.
7. How can an M:N relationship be represented in a relational schema?

## 3.20 Further reading

The relational model is foundational. Codd's original paper is still worth reading because it shows the conceptual leap away from application-specific file structures. Textbooks then explain how the formal ideas translate into practical schema design and ER-to-relational mapping.

# Bibliography

[1] E. F. Codd. A Relational Model of Data for Large Shared Data Banks. *Communications of the ACM*, 13(6), 1970.

[2] C. J. Date. *An Introduction to Database Systems*. Addison-Wesley.

[3] R. Elmasri and S. B. Navathe. *Fundamentals of Database Systems*. Pearson.

## 3.21 Wrap-up

The relational model gives the formal language for tables, keys, and constraints. It also gives the bridge from conceptual modeling to implementable schemas through systematic transformation rules. In the next chapter we turn from structure to querying. SQL will let us ask questions and retrieve information from relational databases.

# Chapter 4

# SQL Basics: Querying Relational Databases

**Chapter guidance**

This chapter introduces SQL as a practical language for querying relational databases. The focus is on reading data: selecting columns, filtering rows, ordering results, grouping rows, and joining tables. The chapter also explains the difference between the mathematical relational model and the SQL view of tables.

## 4.1 What is SQL?

SQL stands for Structured Query Language. It is the standard language family used in relational database systems. SQL is not just for one task. It includes data definition, data manipulation, access control, and transaction related commands. In this chapter, however, we focus on query basics.

SQL is largely declarative. You describe *what* result you want, not step by step *how* to compute it. This is one of the reasons database systems can optimize queries internally.

The language has an interesting history. SQL grew out of the SEQUEL language designed by Donald Chamberlin and Raymond Boyce at IBM. The syntax that became standard is not identical to Codd's mathematical notation, but it made relational ideas usable by a much wider audience. This is also why SQL sometimes looks a little English-like and a little mathematical at the same time.

A beginner often expects a query language to behave like a programming language with loops and explicit control flow. SQL is different. In ordinary querying, we do not tell the system to inspect the first row, then the second row, then the third row. Instead we describe the set of rows we want. This set-oriented view is one of the great strengths of relational databases.

## 4.2 SQL and the relational model

The relational model gives the theoretical foundation. SQL is the practical language used in real systems. They are closely related, but not identical. For example, SQL

tables can contain duplicate rows unless constraints prevent that. SQL also uses `NULL`, which does not fit neatly into the clean set based mathematics of the pure relational model.

This difference is important for clear thinking. In the pure relational model, a relation is a set, so duplicate tuples are not part of the meaning. In SQL, however, the result of a query is often best understood as a table-like object that may contain repeated rows unless `DISTINCT` is used or keys make repetition impossible. This is one reason why SQL should be learned both as a practical language and as an implementation of deeper relational ideas rather than as their exact copy.

## 4.3 SQL dialects and sublanguages

Different DBMS products support slightly different SQL dialects. SQLite, PostgreSQL, MySQL, SQL Server, and Oracle all speak SQL, but they differ in details. This is why examples often emphasize general ideas first.

A common division of SQL is:

- DDL for schema definition
- DML for data manipulation
- DCL for access control
- transaction related commands for safe changes

In practice, these parts are closely connected. A useful database user may define tables, insert and update data, grant access rights, and then query the data. Still, it is pedagogically helpful to separate these roles. This chapter concentrates on reading data by queries. The next chapter will continue with data definition, updates, constraints, and transactions.

## 4.4 A running example schema

We use a small business style example with customers, invoices, products, and invoice lines. A simple schema might contain:

- `CUSTOMER(customer_id, customer_name, city, customer_type, district)`
- `INVOICE(invoice_id, customer_id, year, invoice_total, status)`
- `PRODUCT(product_id, product_name, model, unit_price, color)`
- `INVOICE_LINE(invoice_id, product_id, quantity)`

This example is small enough to understand and rich enough to illustrate joins. The same schema is visualized in figure 4.1, where the foreign-key arrows show how invoice lines connect invoices and products, and how invoices connect customers.

A quick look at figure 4.1 already suggests common questions. Which customers have invoices? Which products appear on which invoices? What is the total quantity sold per product? Such questions will lead us naturally from single-table queries to joins and grouping.

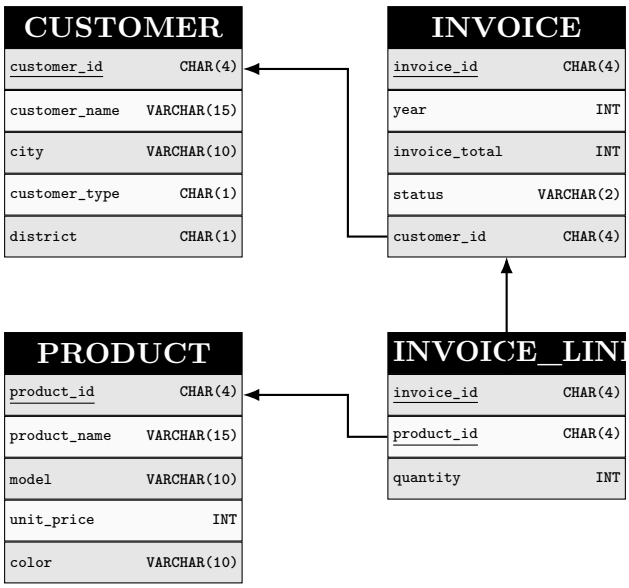



Figure 4.1: The running SQL example as a small relational schema with foreign-key links. INVOICE refers to CUSTOMER, and INVOICE_LINE refers to both INVOICE and PRODUCT.

## 4.5 Basic query form

A simple SQL query often has the form

```
SELECT column_list
FROM table_name
WHERE condition;
```

A beginner should read this query in the logical order rather than in the written order. First the DBMS looks at the relation named in the `FROM` clause. Then it keeps only those rows that satisfy the `WHERE` condition. After that it forms the requested output columns from the surviving rows. Finally, clauses such as `ORDER BY` and `LIMIT` shape the presentation of the result. The optimizer may execute the query differently under the hood, but this logical reading order is a very useful mental model.

A slightly richer logical reading order is this:

$$\texttt{FROM} \rightarrow \texttt{WHERE} \rightarrow \texttt{GROUP BY} \rightarrow \texttt{HAVING} \rightarrow \texttt{SELECT} \rightarrow \texttt{ORDER BY}.$$

This is not necessarily the physical execution order inside the DBMS, but it is an excellent way to understand the meaning of a query.

## 4.6 Selecting columns

To return only selected columns, list them explicitly.

```
SELECT customer_name, city
FROM CUSTOMER;
```

To return all columns, use `*`. That is often convenient when exploring a table, but explicit column lists are usually better in serious work. They make the intention clearer, reduce accidental dependence on future schema changes, and avoid returning unnecessary data.

Sometimes we want the result column to have a more readable name. Then we can use a column alias.

```
SELECT customer_name AS name, city AS hometown
FROM CUSTOMER;
```

Aliases do not change the underlying schema. They only rename columns in the result of the query.

## 4.7 Removing duplicates with DISTINCT

Because SQL query results may contain duplicate rows, it is often useful to remove them explicitly.

```
SELECT DISTINCT city
FROM CUSTOMER;
```

This query asks for the set of cities represented in the customer table. Without `DISTINCT`, the same city could appear many times if many customers live there.

This is a good example of the difference between relational theory and SQL practice. In relational algebra, projection automatically removes duplicates because the result is a relation, hence a set. In SQL, ordinary `SELECT` does not do that unless `DISTINCT` is requested.

## 4.8 Expressions in the SELECT clause

The `SELECT` list can contain not only stored columns but also expressions. We can compute derived output values directly in the query.

```
SELECT product_name, unit_price, unit_price * 1.25 AS
   price_with_tax
FROM PRODUCT;
```

This is useful because many reports need calculated values rather than raw stored data alone. The important idea is that the expression affects the displayed result, not necessarily the stored database.

## 4.9 Filtering rows with WHERE

The `WHERE` clause restricts rows.

```
SELECT customer_name, city
FROM CUSTOMER
WHERE city = 'Jyväskylä';
```

Conditions can use comparisons, conjunctions, disjunctions, and negation. Parentheses matter, because operator precedence matters.

```
SELECT customer_name, city, customer_type
FROM CUSTOMER
WHERE city = 'Jyväskylä'
  AND customer_type = 'R';
```

The meaning is simple: only rows satisfying the condition survive. All others are discarded before the final result is formed. This makes `WHERE` one of the most important clauses in practical SQL. A missing or incorrect condition can turn a precise query into a large and misleading result.

## 4.10 Comparison operators

SQL commonly uses the comparison operators `=`, `<>`, `<`, `<=`, `>`, and `>=`. These are used in `WHERE` conditions and sometimes in `HAVING` conditions later.

```
SELECT product_name, unit_price
FROM PRODUCT
WHERE unit_price > 100;
```

```
SELECT invoice_id, status
FROM INVOICE
WHERE status <> 'PA';
```

The first query keeps only expensive products. The second keeps invoices whose status is not `PA`. Such examples are simple, but they train the basic habit of reading a query as a logical filter over rows.

## 4.11 Range tests with BETWEEN

When a value should lie within an interval, `BETWEEN` can be clearer than writing two comparisons.

```
SELECT invoice_id, invoice_total
FROM INVOICE
WHERE invoice_total BETWEEN 100 AND 500;
```

This is roughly equivalent to saying that `invoice_total >= 100` and `invoice_total <= 500`. The main benefit is readability.

## 4.12 Pattern matching and membership

SQL offers pattern matching with `LIKE` and list membership with `IN`.

```
SELECT product_name
FROM PRODUCT
WHERE product_name LIKE 'A%';
```

```
SELECT customer_name
FROM CUSTOMER
WHERE city IN ('Turku', 'Tampere', 'Helsinki');
```

In the first example, `%` stands for any sequence of characters, so the query finds product names beginning with `A`. In the second example, `IN` is simply a compact way of writing several alternatives connected by `OR`.

Another useful pattern is the underscore character `_`, which stands for one single character.

```
SELECT customer_id
FROM CUSTOMER
WHERE customer_id LIKE 'C___';
```

This query looks for four-character identifiers beginning with `C`.

## 4.13 NULL semantics

Because `NULL` means missing or unknown information, comparisons require care. To test whether a value is null, use `IS NULL` or `IS NOT NULL`.

```
SELECT customer_name
FROM CUSTOMER
WHERE district IS NULL;
```

A very common beginner error is to write

```
WHERE district = NULL
```

but this does not work as intended. The reason is conceptual: `NULL` is not an ordinary value. It represents missing or unknown information. Therefore SQL provides the special predicates `IS NULL` and `IS NOT NULL`.

## 4.14 Ordering and limiting results

The `ORDER BY` clause sorts output. `LIMIT` is useful when only a few rows are needed.

```
SELECT customer_name, city
FROM CUSTOMER
ORDER BY city, customer_name
LIMIT 10;
```

By default, sorting is ascending. We may also specify descending order.

```
SELECT invoice_id, invoice_total
FROM INVOICE
ORDER BY invoice_total DESC;
```

Ordering is applied late in the logical interpretation of the query, after the rows to be returned have already been determined. This is why `ORDER BY` is best understood as shaping the final presentation of the result rather than changing which rows qualify.

## 4.15 Why multiple tables?

One table is rarely enough because different kinds of facts belong in different places. Products and customers are different things. Orders connect them. When data is split across related tables, joins allow us to reconstruct useful combinations.

This is not a weakness of the relational design. It is one of its strengths. Good schemas separate different kinds of facts so that updates are cleaner and redundancy is reduced. Joins then rebuild meaningful combined views only when needed.

## 4.16 Table aliases

When queries involve more than one table, or even one table more than once, short aliases are very useful.

```
SELECT c.customer_name, i.invoice_id
FROM CUSTOMER AS c
JOIN INVOICE AS i
  ON c.customer_id = i.customer_id;
```

Aliases make queries shorter and often clearer. They are especially important when different tables contain attributes with the same name, such as `customer_id` or `invoice_id`. They also help when the same table is joined to itself, which we will encounter later in relational algebra and SQL self-joins.

## 4.17 Joins

A join combines rows from two tables according to a matching condition.

Joins are central because normalized databases usually store facts in separate places. A customer name belongs naturally in the customer table. Invoice totals belong naturally in the invoice table. The whole point of a join is to reconstruct a meaningful view when a question spans several kinds of facts. In that sense, joins are the bridge between good schema design and useful query results.

```
SELECT c.customer_name, i.invoice_id, i.invoice_total
FROM CUSTOMER AS c
JOIN INVOICE AS i
  ON c.customer_id = i.customer_id;
```

This query returns invoice data together with the customer name. The key idea is that the foreign key in `INVOICE` matches the primary key in `CUSTOMER`.

It helps to walk through the query slowly. First we take one row from `CUSTOMER` and one row from `INVOICE`. Next we test whether their `customer_id` values are equal. If not, that pair is discarded. If yes, the pair is kept, and the selected output columns are produced. Conceptually this sounds like many pairwise checks, although a real DBMS will normally use smarter join algorithms. The same logic later appears again in formal form in chapter 6.

## 4.18 From two-table joins to longer join chains

Many practical questions need more than one join. In our running schema, an invoice line connects an invoice to a product, and the invoice connects to the customer. So a question about which customer bought which product naturally leads to a chain of joins.

```
SELECT c.customer_name, p.product_name, l.quantity
FROM CUSTOMER AS c
JOIN INVOICE AS i
  ON c.customer_id = i.customer_id
JOIN INVOICE_LINE AS l
  ON i.invoice_id = l.invoice_id
JOIN PRODUCT AS p
  ON l.product_id = p.product_id;
```

A good way to read this is one join at a time. First match customers with their invoices. Then match those invoices with their invoice lines. Finally match each line with the corresponding product. Each join adds another layer of meaning.

## 4.19 Cartesian products and why join conditions matter

A join condition is not optional decoration. Without it, combining tables can easily produce a Cartesian product, that is, every row of one table paired with every row of the other.

If `CUSTOMER` has 100 rows and `INVOICE` has 500 rows, then their Cartesian product has 50 000 pairs. Most of those pairs are meaningless. The join condition removes the meaningless combinations and keeps only the pairs that satisfy the intended relationship.

This is why careful matching conditions are essential. Incorrect joins may not produce error messages. They may simply produce wrong answers.

## 4.20 Aggregates and grouping

Practical SQL also supports aggregate functions such as `COUNT`, `SUM`, `AVG`, `MIN`, and `MAX`. The `GROUP BY` clause groups rows before aggregation.

```
SELECT customer_id, SUM(invoice_total) AS total_sales
FROM INVOICE
GROUP BY customer_id;
```

A `HAVING` clause can then filter groups.

```
SELECT customer_id, SUM(invoice_total) AS total_sales
FROM INVOICE
GROUP BY customer_id
HAVING SUM(invoice_total) > 1000;
```

The logical idea is simple. First the rows are partitioned into groups with the same `customer_id`. Then the aggregate is computed separately for each group. Finally `HAVING` keeps only those groups that satisfy the stated condition.

## 4.21 WHERE versus HAVING

It is useful to distinguish `WHERE` from `HAVING`. The `WHERE` clause filters individual rows before grouping. The `HAVING` clause filters whole groups after grouping.

For example:

```
SELECT customer_id, SUM(invoice_total) AS total_sales
FROM INVOICE
WHERE year = 2025
GROUP BY customer_id
HAVING SUM(invoice_total) > 1000;
```

This query first keeps only invoices from 2025. Then it groups them by customer. Finally it keeps only those customers whose 2025 total exceeds 1000. The distinction is important because grouping changes what kind of object we are filtering: rows before grouping, groups after grouping.

## 4.22 COUNT(*) and COUNT(column)

A small but important detail is the difference between `COUNT(*)` and `COUNT(column)`.

```
SELECT COUNT(*)
FROM CUSTOMER;
```

counts rows, while

```
SELECT COUNT(district)
FROM CUSTOMER;
```

counts only those rows in which `district` is not `NULL`. This matters because `NULL` values are ignored by many aggregate functions.

## 4.23 A note on style

Readable SQL matters. Use clear aliases, sensible indentation, and explicit join conditions. Good style prevents many mistakes.

One practical habit is to put each major clause on its own line and indent join conditions clearly. Another is to choose aliases that are short but meaningful, such as `c` for `CUSTOMER` and `i` for `INVOICE`. These may look like small matters, but they improve correctness as well as readability.

## 4.24 Review questions and small exercises

1. What is the difference between projection in relational thinking and `SELECT` in SQL?
2. Why is the `WHERE` clause important in avoiding unintended large result sets?
3. Explain the effect of `ORDER BY` on query results.

4. What is an alias, and why can aliases make SQL clearer?
5. Write a query that returns all rows from a table `Student` with a grade greater than 3.
6. What is the difference between `COUNT(*)` and `COUNT(column)`?
7. Why do joins need attention to matching conditions?
8. What is the purpose of `GROUP BY`?

## 4.25 Further reading

This chapter introduces the practical core of SQL. A standards-oriented view helps later when different DBMS products behave slightly differently. Textbooks by Beaulieu and Viescas are useful for learning idiomatic SQL, while broader database texts explain how SQL connects to relational ideas.

# Bibliography

[1] A. Beaulieu. *Learning SQL*. O'Reilly.

[2] J. Viescas and D. Hernandez. *SQL Queries for Mere Mortals*. Addison-Wesley.

[3] A. Silberschatz, H. F. Korth, and S. Sudarshan. *Database System Concepts*. McGraw-Hill.

## 4.26 Wrap-up

We have now seen how SQL asks questions about stored data. The most important ideas are selecting columns, filtering rows, combining tables with joins, grouping rows for aggregation, and interpreting results carefully in the presence of duplicates and `NULL` values. The next chapter continues with database programming tasks such as table creation, constraints, updates, access control, and transactions.

# Chapter 5

# Database Programming in SQL: DDL, DML, Access Control, and Transactions

**Chapter guidance**

This chapter moves from reading data to managing and changing it. It covers table creation, constraints, schema evolution, views, indexes, inserts, updates, deletes, privileges, and transactions. Read it as the chapter that turns SQL from a query language into a full database workbench.

## 5.1 From querying to programming

Many students first meet SQL through `SELECT`. That is only the beginning. Real database work also means defining schemas, protecting data quality, controlling permissions, and ensuring correct updates under failure or concurrency.

This chapter is where databases begin to feel like engineering rather than only querying. A schema is a promise about structure. Constraints are promises about valid states. Transactions are promises about how changes behave even when multiple users work at the same time or when the system fails halfway through an update. That is why this material is worth reading carefully even if query examples feel more immediately exciting.

A useful mental model is this:

- the *schema* says what data may exist,
- *rights and privileges* say who may do what,
- *transactions* say how change remains correct.

These three ideas belong together. Defining a good schema without good access control is incomplete. Allowing updates without transactions is risky. Protecting a system with permissions but not with constraints still leaves data quality exposed.

## 5.2 SQL data types in practice

Before defining tables, it helps to recall that SQL is typed. Each column is declared with a type, and that type is part of the meaning of the data.

Common type families include:

- string types such as `CHAR(n)`, `VARCHAR(n)`, and `TEXT`,
- integer types such as `INT` and `INTEGER`,
- exact numeric types such as `NUMERIC(p,s)` and `DECIMAL(p,s)`,
- boolean types such as `BOOLEAN`,
- date and time types such as `DATE` and `TIMESTAMP`.

These types are not only syntactic labels. They influence comparison, indexing, storage, permitted values, and the kinds of constraints that can be expressed naturally. For example, money should usually be stored in an exact numeric type rather than a floating-point type, because financial calculations should not depend on rounding surprises.

## 5.3 DDL: creating tables

Data Definition Language describes the structure of the database.

```
CREATE TABLE CUSTOMER (
  customer_id   INTEGER PRIMARY KEY,
  customer_name TEXT NOT NULL,
  city          TEXT,
  customer_type TEXT,
  district      TEXT
);
```

A table definition specifies columns, types, and constraints. Constraints are the first line of defense for data quality.

A good way to read a `CREATE TABLE` statement is line by line. First identify the table name. Then scan the columns and ask what each column means in the domain. After that inspect the constraints: which column may never be missing, which values must be unique, which references must point to existing rows, and which business conditions are checked directly by the DBMS? Reading DDL in this order turns a block of syntax into a design explanation.

## 5.4 Primary keys and foreign keys in DDL

```
CREATE TABLE INVOICE (
  invoice_id     INTEGER PRIMARY KEY,
  customer_id    INTEGER NOT NULL,
  year           INTEGER,
  invoice_total  DECIMAL(10,2) CHECK (invoice_total >= 0),
```

```
  status          TEXT DEFAULT 'OK',
  FOREIGN KEY (customer_id) REFERENCES CUSTOMER(customer_id)
);
```

This example shows that keys are not just conceptual ideas. They become executable constraints in the DBMS.

The `PRIMARY KEY` clause makes `invoice_id` the main identifier of each invoice. The `NOT NULL` constraint ensures that every invoice refers to some customer. The foreign-key clause says that the referenced customer must exist in the `CUSTOMER` table. The `CHECK` constraint ensures that totals cannot become negative. The `DEFAULT` clause gives a standard status when no explicit value is provided.

Taken together, these constraints move important assumptions out of application code and into the database itself.

## 5.5 Composite keys and referential actions

Some tables naturally require more than one attribute for identification. This often happens in relationship tables.

```
CREATE TABLE INVOICE_LINE (
  invoice_id  INTEGER,
  product_id  INTEGER,
  quantity    INTEGER CHECK (quantity > 0),
  PRIMARY KEY (invoice_id, product_id),
  FOREIGN KEY (invoice_id) REFERENCES INVOICE(invoice_id) ON
     DELETE CASCADE,
  FOREIGN KEY (product_id) REFERENCES PRODUCT(product_id) ON
     DELETE SET NULL
);
```

Here the pair `(invoice_id, product_id)` forms a composite primary key. That makes sense if each product should appear at most once per invoice line list.

The example also introduces referential actions:

- `RESTRICT` denies a delete or update in the parent when dependent rows exist,
- `ON DELETE SET NULL` replaces the foreign-key value in the child by `NULL`,
- `ON DELETE CASCADE` propagates the delete to the child rows,
- `ON DELETE SET DEFAULT` assigns the default value if one exists.

These options matter because foreign keys are not only about checking references at insertion time. They also govern what happens when referenced rows later change or disappear. The best choice depends on the semantics of the application. Invoice lines often disappear when the invoice disappears, which makes cascade behavior understandable. In other situations, automatic cascading may be too dangerous.

## 5.6 Column constraints beyond keys

Useful column constraints include `NOT NULL`, `UNIQUE`, `CHECK`, and `DEFAULT`. A good rule of thumb is to encode stable business rules in constraints and leave volatile policy details to the application layer.

```
CREATE TABLE CUSTOMER (
  customer_id   CHAR(4)       PRIMARY KEY,
  customer_name VARCHAR(15)   NOT NULL,
  email         VARCHAR(50)   UNIQUE,
  customer_type CHAR(1)       DEFAULT 'A',
  district      CHAR(1)       CHECK (district IN ('I','L','1','2'
     ,'3'))
);
```

This design prevents several classes of mistakes. A customer without a name is rejected. Duplicate email addresses are forbidden if email is supposed to identify or at least distinguish customers. The customer type gets a default classification. The district must lie in a permitted set.

A useful design principle is to store only data states that already make sense. The more clearly this is enforced in the schema, the less fragile the surrounding software becomes.

## 5.7 Schema evolution with ALTER TABLE

Databases evolve. Requirements change. Columns are added, renamed, or removed. Constraints are tightened. The exact syntax differs between DBMS products, but the idea is common.

```
ALTER TABLE CUSTOMER ADD COLUMN email TEXT;
```

Schema evolution should be done carefully because production data already exists.

A more elaborate example is:

```
CREATE TABLE INVOICE_LINE (
  invoice_id INTEGER,
  product_id INTEGER
);

ALTER TABLE INVOICE_LINE
  ADD COLUMN quantity INTEGER;
```

In some DBMS products, constraints such as primary keys can also be added later. In others, the support is more limited. This is why schema migration should be planned rather than improvised. One should always ask how existing rows will behave under the new rule. Adding a new `NOT NULL` column, for instance, may require either a default value or a backfill step for old rows.

## 5.8 Views

Views are especially helpful when we want to present data in a simpler or safer form than the underlying base tables. For instance, an accountant may need invoice totals and customer names but not every internal customer attribute. A view can package the relevant join and hide the rest. In this way, SQL supports not only storage and querying but also careful presentation of data to different groups of users.

**Definition 5.1: View**

A *view* is a stored query that behaves like a virtual table.

Views help with reuse, abstraction, and access control.

```
CREATE VIEW v_customer_public AS
SELECT customer_id, city
FROM CUSTOMER;
```

A public view can expose only those columns that some users should see.

Views can also express derived data, not just hidden subsets of columns. For example:

```
CREATE VIEW v_city_sales AS
SELECT customer_id, SUM(invoice_total) AS total_sales
FROM INVOICE
GROUP BY customer_id;
```

Such a view packages a useful computation behind a stable name. This is convenient because many users may need the same derived information, and it keeps the logic in one place rather than duplicating it across applications.

Another valuable use of views is privacy-preserving aggregation. A view that shows averages by city, department, or year can reveal useful statistics while hiding individual-level records.

## 5.9 Indexes

**Definition 5.2: Index**

An *index* is an auxiliary data structure that speeds up access to rows based on search conditions, at the cost of extra storage and update overhead.

Indexes can greatly speed up queries, especially on search columns and join columns.

```
CREATE INDEX idx_invoice_customer
ON INVOICE(customer_id);
```

An index is not magic. If data changes often, indexes also have a maintenance cost. Conceptually, several indexing ideas are common:

- a sorted access structure, which makes searches and ranges efficient but can be costly to maintain,
- a B+-tree style index, which supports efficient updates and also range queries,

- a hash index, which is strong for equality lookups but not for range conditions.

For example, an index on a birth date or invoice date can help a query such as:

```
SELECT invoice_id, year
FROM INVOICE
WHERE year BETWEEN 2020 AND 2025;
```

This kind of query benefits from an index that preserves ordering information. By contrast, an equality lookup such as finding one customer by identifier may be handled efficiently by several different indexing strategies.

A useful design question is not “Should I index everything?” but rather “Which queries matter enough to justify index maintenance cost?” Too many indexes can slow inserts, updates, and deletes significantly.

## 5.10 From DDL to DML

After the schema has been defined, we start manipulating its contents. Data Manipulation Language changes relation instances.

A convenient application-oriented summary is the CRUD pattern:

- Create rows with `INSERT`,
- Read rows with `SELECT`,
- Update rows with `UPDATE`,
- Delete rows with `DELETE`.

This mapping is simple, but it is a useful bridge between database language and software design.

## 5.11 INSERT

```
INSERT INTO CUSTOMER(customer_id, customer_name, city)
VALUES (1, 'Aino Aaltonen', 'Jyväskylä');
```

It is usually better to include the explicit column list, even if all columns are being filled. That makes the statement more robust against future schema changes and easier to read.

Multiple-row inserts are also common:

```
INSERT INTO CUSTOMER (customer_id, customer_name, city,
   customer_type, district)
VALUES
  (2, 'Ben Berg', 'Turku', 'A', 'L'),
  (3, 'Cleo Laine', 'Helsinki', 'B', '2');
```

Columns not mentioned in the insert either take their default values or remain `NULL`, depending on the schema. Typical reasons for failure include duplicate primary keys, missing parent rows in foreign-key references, or violated `CHECK` conditions.

## 5.12 UPDATE

```
UPDATE CUSTOMER
SET city = 'Turku'
WHERE customer_id = 1;
```

An update changes all rows that satisfy the `WHERE` condition. This is why every update should be read twice before execution. The condition determines the scope of the change.

We may also update several columns at once or use expressions:

```
UPDATE INVOICE
SET invoice_total = invoice_total + 10,
    status = 'OK'
WHERE year = 2025;
```

A practical safety habit is to run the corresponding `SELECT ... WHERE ...` first. That shows which rows will be affected before the change becomes real. This habit prevents many accidental mass updates.

## 5.13 DELETE

```
DELETE FROM CUSTOMER
WHERE customer_id = 1;
```

A `DELETE` statement removes rows, not the table structure itself. Removing the table definition would require `DROP TABLE` instead.

Again, the `WHERE` clause is crucial. Without it, all rows are deleted:

```
DELETE FROM INVOICE;
```

That statement leaves the table in place but empties it completely. Foreign-key constraints may block deletion, propagate it through cascading rules, or force a more careful sequence of operations. In real systems, destructive changes are often performed inside explicit transactions. Some applications avoid hard deletion altogether and instead mark rows as inactive by a status column. This is often called a soft delete.

## 5.14 Roles, users, and privileges

Real databases support permissions. Some users may read data. Others may insert or update. Administrators may create schemas or grant rights to others.

The SQL standard often describes these access-control subjects in terms of *roles*. A role can represent one user, a group of users, or a responsibility such as analyst, billing clerk, or administrator. This is useful because rights can then be assigned by responsibility rather than one by one to individual people.

Typical privileges include:

- `CONNECT` for connecting to the database,
- `SELECT` for reading data,

- `INSERT` for adding rows,
- `UPDATE` for modifying rows or columns,
- `DELETE` for removing rows,
- `EXECUTE` for routines,
- `ALL PRIVILEGES` as a broad shorthand in systems that support it.

## 5.15 GRANT and REVOKE

```
GRANT SELECT ON CUSTOMER TO analyst_role;
REVOKE UPDATE ON CUSTOMER FROM analyst_role;
```

Roles are useful because rights can be grouped by responsibility rather than assigned one by one to each user.

The general pattern is:

```
GRANT privilege[, privilege]*
ON object
TO role[, role]*
[WITH GRANT OPTION];
```

For example:

```
GRANT SELECT, UPDATE
ON INVOICE
TO analyst_role
WITH GRANT OPTION;
```

The phrase `WITH GRANT OPTION` means that the recipient may further grant those rights to others. This is powerful but should be used carefully, because rights can then propagate.

Some systems also allow fine-grained, column-level permissions:

```
GRANT UPDATE (invoice_total, status)
ON INVOICE
TO billing_clerk;
```

The matching removal form is `REVOKE`. In systems where rights were further delegated, revoking rights from an intermediary can also remove derived rights farther downstream. This is one reason the principle of least privilege is so important: give only the rights that are genuinely needed.

## 5.16 Role hierarchies and views as protection layers

Roles can often be granted to roles, creating hierarchies. This simplifies administration because one grant may affect many users at once.

Views also play an important role in access control. Instead of granting direct access to a sensitive base table, one may expose only a safer view. For example, a reporting

role may be allowed to read public customer information but not private contact details. In that sense, views are not only a convenience mechanism. They are also a security design tool.

## 5.17 Transactions

**Definition 5.3: Transaction**

A *transaction* is a logical unit of work that should be executed completely or not at all.

Typical transaction commands are `BEGIN`, `COMMIT`, and `ROLLBACK`.

```
BEGIN TRANSACTION;
UPDATE ACCOUNT SET balance = balance - 100 WHERE account_id = 1;
UPDATE ACCOUNT SET balance = balance + 100 WHERE account_id = 2;
COMMIT;
```

If something goes wrong in the middle, the transaction should be rolled back.

A transaction groups several steps into one coherent action. That matters because many real-world operations are meaningful only as a whole. A bank transfer is the classic example. Deducting money from one account without crediting it to the other is not a valid finished state. The database must therefore treat the transfer as one unit.

Many systems also provide an autocommit mode, where each individual statement becomes its own transaction unless an explicit `BEGIN` starts a larger one. This is convenient, but it can hide the need for explicit transaction boundaries in more complex operations.

## 5.18 A guarded update example

Consider a business rule stating that an account balance must never go below zero. Then a withdrawal or transfer must include a check before the transaction is allowed to complete.

Conceptually, the workflow is:

1. begin the transaction,
2. read the current balance,
3. perform the update,
4. check the new balance,
5. commit if the rule is still satisfied,
6. otherwise roll back.

This example shows that transactions and business rules work together. Atomicity alone is not enough. The application must also decide whether the final state is acceptable.

## 5.19 ACID properties

**Definition 5.4: ACID**

The acronym *ACID* stands for Atomicity, Consistency, Isolation, and Durability.

Atomicity means all or nothing. Consistency means constraints remain satisfied. Isolation means concurrent transactions should not interfere in unacceptable ways. Durability means committed changes survive failures.

These four ideas deserve a moment of reflection.

- **Atomicity** protects against half-finished work.
- **Consistency** means that valid constraints and rules remain valid after the transaction.
- **Isolation** addresses concurrency by making interleaved execution behave as if transactions had been run in some acceptable serial order.
- **Durability** means that once a commit succeeds, the result is meant to survive crashes, power loss, or restart.

ACID does not mean that every transaction is automatically correct. It means that the DBMS offers guarantees under a well-defined model of transaction behavior. The transaction itself must still encode sensible logic.

## 5.20 Isolation and concurrency anomalies

DBMS products often offer several isolation levels. Stronger isolation gives safer behavior but may reduce concurrency. The reason is that parallel transactions can interfere in subtle ways.

Classical anomalies include:

- **lost update**: two transactions overwrite each other so that one update disappears,
- **dirty read**: one transaction reads changes made by another transaction before those changes are committed,
- **non-repeatable read**: the same row is read twice and yields different values because another transaction committed a change in between,
- **phantom read**: the same range query is executed twice and yields a different set of rows because another transaction inserted or deleted matching rows.

These anomalies explain why concurrency control is a central part of database systems rather than an optional extra.

## 5.21 Isolation levels

A common summary of the standard isolation levels is:

| Isolation level | Dirty reads | Non-repeatable reads | Phantoms |
|---|---|---|---|
| SERIALIZABLE | no | no | no |
| REPEATABLE READ | no | no | yes |
| READ COMMITTED | no | yes | yes |
| READ UNCOMMITTED | yes | yes | yes |

This table captures the standard intent, although exact behavior may differ by DBMS. The general lesson is simple: looser isolation often allows more concurrency and may improve performance, but it also permits more anomalies.

A typical SQL pattern is:

```
BEGIN;
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
-- do work
COMMIT;
```

The exact syntax varies by system, but the underlying design choice is the same. One chooses an isolation level to balance correctness guarantees against concurrency and speed.

## 5.22 Locking and serializability

To implement isolation, DBMS products use concurrency-control techniques such as locking, timestamp ordering, or optimistic methods. A useful high-level idea is the notion of a *schedule*, which is the interleaving of actions from concurrent transactions.

A schedule is called *serializable* if it is equivalent in effect to some serial execution order. This is a deep theoretical idea, but its practical meaning is very intuitive: even if transactions run concurrently, the final result should be one that could have arisen from some safe one-after-another order.

## 5.23 Two-phase locking

A classical protocol for achieving serializable behavior is two-phase locking, often abbreviated as 2PL.

Its basic idea is simple:

- in the *expanding phase*, a transaction acquires locks and does not release them yet,
- in the *shrinking phase*, it releases locks and acquires no new ones.

Important variants include:

- **conservative 2PL**, where all needed locks are acquired before the transaction starts its real work,
- **strict 2PL**, where write locks are held until commit or rollback,
- **strong strict 2PL**, where all locks are held until commit or rollback.

Strict 2PL is especially important in practice because it avoids certain rollback cascades and provides a cleaner transactional behavior.

## 5.24 Waiting and lock upgrades

Concurrency is not only about writers blocking writers. Even readers may matter. Suppose one transaction reads an item under a shared lock and later wants to write it, which requires upgrading to an exclusive lock. If another transaction is still holding a shared lock on that item, the upgrade must wait.

This explains why concurrency behavior can feel surprising. A transaction that initially looks harmless because it only reads can later block a writer or an upgrade. Good DBMS design handles such situations systematically, but application developers should still understand that these waits are normal.

## 5.25 Deadlocks

A direct consequence of locking is the possibility of deadlock.

A *deadlock* is a cycle of waiting: each transaction is waiting for a lock held by another transaction in the cycle. For example:

- transaction $T_1$ locks item $A$ and then waits for item $B$,
- transaction $T_2$ locks item $B$ and then waits for item $A$.

Neither can continue. The system must intervene.

**Joke box**

Two very polite transactions meet in a very narrow corridor. One says, “After you.” The other replies, “No, after you.” Both keep holding their locks. Nothing moves. The DBMS finally sighs and says, “One of you must roll back.”

The joke is light, but the point is real. Deadlocks do not usually resolve by themselves. The DBMS must detect or prevent them and then abort one transaction, releasing its locks so that the other can proceed.

## 5.26 Handling deadlocks

Typical handling strategies include:

- **prevention**, for example by requiring a fixed global lock order,
- **timestamp-based protocols**, such as wait-die or wound-wait,
- **detection**, often via a wait-for graph in which cycles indicate deadlock,
- **timeouts**, where a transaction waiting too long is aborted and retried.

From an application point of view, the practical lesson is that deadlocks are not unusual failures. They are normal events in concurrent systems. Robust applications therefore treat “transaction aborted” as something that may require retry logic rather than panic.

**Remark 5.1: Practical lesson**

Transactions are not an optional luxury. They are what separate reliable data management from hopeful guessing.

## 5.27 Review questions and small exercises

1. What is the difference between DDL and DML?
2. Why are integrity constraints important even if application code already performs checks?
3. Explain the roles of `INSERT`, `UPDATE`, and `DELETE`.
4. What is a transaction, and why is it useful?
5. State the four ACID properties in simple words.
6. Why can concurrent transactions cause problems without control mechanisms?
7. What is the role of `COMMIT` and `ROLLBACK`?
8. Why is access control part of data management and not an afterthought?

## 5.28 Further reading

To go deeper into transactions and recovery, systems-oriented textbooks are especially valuable. They explain why correctness in multi-user environments requires more than good intentions. Practical SQL books are useful for syntax, but transaction theory becomes clearer in broader database systems texts.

# Bibliography

[1] A. Silberschatz, H. F. Korth, and S. Sudarshan. *Database System Concepts*. McGraw-Hill.

[2] R. Ramakrishnan and J. Gehrke. *Database Management Systems*. McGraw-Hill.

[3] T. Härder and A. Reuter. *Principles of Transaction-Oriented Database Recovery*. Morgan Kaufmann.

## 5.29 Wrap-up

This chapter completed the practical core of SQL use. We can now define tables, protect them with constraints, expose or hide data through views and privileges, update data safely, and reason about concurrency through transactions and isolation. The next chapter steps back into formal theory and asks how queries can be expressed as algebraic operations.

**Joke box**

- **DDL** walks into a bar and says: "I would like to `CREATE TABLE`."
- **DCL** replies (smiling): "Only if I `GRANT` you permission."
- **DML** jumps in: "Great, I will `INSERT` some customers, `UPDATE` the bill, and `DELETE` the evidence."

**A QUERY** squints at the menu and says:

```
SELECT beer
FROM fridge
WHERE cold = TRUE
ORDER BY foam DESC
LIMIT 1;
```

**TxCL** sighs (kindly): "BEGIN... and if this gets messy, we ROLLBACK."

Kind reminder: be nice to your future self – use transactions, and always leave the database consistent.

# Chapter 6

# Relational Algebra

**Chapter guidance**

This chapter presents relational algebra as an operational formalism for queries. It explains the core operators, joins, division, relational completeness, and the role of algebra in query optimization. Read it as the mathematical engine that sits behind declarative querying.

## 6.1 Why relational algebra matters

Relational algebra matters for at least three reasons. First, it gives a precise language for describing queries. Second, SQL queries can be translated into algebraic form. Third, algebraic equivalences support optimization, which is how DBMS systems improve performance without changing meaning.

Many students discover here that a query is not magic. Even a friendly SQL statement can be understood as a composition of smaller logical operations. That insight is useful later when a query gives the wrong result or performs badly. Instead of guessing, we can ask: which rows are filtered first, which attributes are kept, and where is the join introduced?

The notation also trains an important habit: read a query as a sequence of meaning-preserving transformations. Instead of staring at one long SQL statement, we can say: first combine these relations, then filter, then project, then rename if needed. That is why E. F. Codd's algebraic view mattered so much. It made query reasoning and optimization mathematically discussable.

Relational algebra is also important because its operators are closed over relations: a relation goes in, and a relation comes out. This means we can build large expressions from smaller ones without leaving the formal framework. That closure property is one of the main reasons relational algebra is so elegant and so useful for optimizers.

## 6.2 Declarative versus operational query languages

SQL is mainly declarative. Relational algebra is operational in the sense that it describes query computation by composing operators. Both views are valuable. The declarative view is easy for the user. The operational view is useful for reasoning and optimization.

This distinction should not be misunderstood. “Operational” does not mean that relational algebra tells us which index is used or which bytes move first on disk. It means that an algebra expression explicitly shows the logical operations from which the result is built. By contrast, a declarative SQL query focuses on the intended result and leaves the DBMS free to choose an execution strategy.

Relational calculus offers another formal view of querying. It is declarative like SQL: one describes which tuples should belong to the result rather than building the result step by step. Classical database theory shows that relational algebra and safe relational calculus have the same expressive power. This is one reason the algebra is so central: it is both mathematically expressive and operationally useful.

## 6.3 Basic syntax

The operands of relational algebra are relations. The results are also relations. This *closure property* makes it possible to compose operations freely.

A relational algebra query therefore takes one or more relation instances as input and returns a relation instance as output. The schema of the output depends only on the input schemas and on the operator parameters. This is a very useful discipline. It lets us reason about expressions before we even run them.

In practice, two notational styles are common:

- a positional notation, where attributes are identified by position, and
- a named-attribute notation, where attributes are identified by name.

Named-attribute notation is usually easier to read and is the style used in this chapter.

## 6.4 Core operators overview

The classical core operators are selection, projection, union, set difference, cartesian product, and renaming. Derived operators include joins and intersection.

We use the notation

$$\sigma_c(R), \qquad \pi_{A_1,\ldots,A_k}(R), \qquad \rho_{S(B_1,\ldots,B_k)}(R), \qquad R \cup S, \qquad R \setminus S, \qquad R \times S, \qquad R \bowtie_\theta S.$$

Here $R$ and $S$ are relations. The condition $c$ in $\sigma_c(R)$ is a Boolean predicate over attributes of $R$. The list $A_1, \ldots, A_k$ in projection tells us which attributes remain visible. Renaming is crucial when the same relation appears twice in one query, as in self-joins.

It is worth pausing over closure again. Each of these operators returns a relation. That means we can freely nest them:

$$\pi_{Name}\Big(\sigma_{Dept='IT'}(Employee \bowtie Department)\Big).$$

This is one of the great strengths of algebraic query notation: complex expressions are built from a small number of composable parts.

**Definition 6.1: Selection**

Given a relation $R$, the selection $\sigma_c(R)$ returns those tuples of $R$ that satisfy condition $c$.

**Definition 6.2: Projection**

Given a relation $R$, the projection $\pi_{A_1,\ldots,A_k}(R)$ returns the tuples formed by keeping only the listed attributes. Duplicate tuples are removed in the pure algebra.

## 6.5 A running example instance

To make the operators concrete, it is useful to work with a small running instance:

$s =$

| *sid* | *sname* | *age* | *rating* |
|---|---|---|---|
| 22 | *dustin* | 35 | 7 |
| 31 | *lubber* | 55 | 8 |
| 44 | *guppy* | 35 | 5 |

$r =$

| *sid* | *bid* | *day* |
|---|---|---|
| 22 | 101 | 2026-02-01 |
| 22 | 102 | 2026-02-03 |
| 31 | 102 | 2026-02-02 |

$b =$

| *bid* | *color* |
|---|---|
| 101 | *red* |
| 102 | *green* |
| 103 | *red* |

with schemas

$$S(sid, sname, age, rating), \qquad R(sid, bid, day), \qquad B(bid, color).$$

These small relations are enough to illustrate most of the important operators. The examples below will repeatedly return to sailors, reservations, and boats because that setting makes joins, set operations, and division easy to visualize.

## 6.6 Selection and projection

Selection filters rows. Projection filters columns. Together they already express a large class of useful queries.

The distinction is easy to say and important to internalize. A selection changes *which tuples* survive. A projection changes *which attributes* remain visible. If you confuse these two, many later algebra expressions become hard to read. Happily, the names become intuitive with practice: selection selects records, projection projects them onto fewer columns.

For example, if `Employee(Eid, Name, Dept, Salary)` is a relation, then

$$\sigma_{Dept='IT'}(Employee)$$

returns all IT employees, while

$$\pi_{Name,Dept}(Employee)$$

returns only names and departments.

Using the running instance, selection works as follows:

$\sigma_{age<50}(s) =$

| *sid* | *sname* | *age* | *rating* |
|---|---|---|---|
| 22 | *dustin* | 35 | 7 |
| 44 | *guppy* | 35 | 5 |

The tuple for sailor 31 disappears because the condition $age < 50$ is false there.

Projection keeps chosen attributes and removes duplicates:

$$\pi_{age}(s) = \begin{array}{|c|}\hline age \\ \hline 35 \\ 55 \\ \hline \end{array}$$

Notice the set semantics: age 35 appears only once in the result, even though two sailors have that age. This is a good place to remember again that pure relational algebra is set-based.

## 6.7 Union, intersection, and difference

Set operations require union compatibility. This means the participating relations must have matching attribute structure.

**Definition 6.3: Union compatibility**

Two relations are *union compatible* if they have the same number of attributes and the corresponding attributes have matching domains and intended meanings.

- $R \cup S$ contains tuples in either relation.
- $R \cap S$ contains tuples in both relations.
- $R \setminus S$ contains tuples in $R$ but not in $S$.

Because relational algebra is set-based, duplicate tuples are not counted repeatedly. This is mathematically clean and is one reason students should keep apart the pure algebra and practical SQL implementations, where duplicates may persist unless we request `DISTINCT`.

Using the reservation relation $r$, let

$$r_1 = \sigma_{bid=101}(r), \qquad r_2 = \sigma_{bid=102}(r).$$

Then

$$r_1 = \begin{array}{|c|c|c|}\hline sid & bid & day \\ \hline 22 & 101 & 2026\text{-}02\text{-}01 \\ \hline \end{array} \qquad r_2 = \begin{array}{|c|c|c|}\hline sid & bid & day \\ \hline 22 & 102 & 2026\text{-}02\text{-}03 \\ 31 & 102 & 2026\text{-}02\text{-}02 \\ \hline \end{array}$$

and therefore

$$r_1 \cup r_2 = \begin{array}{|c|c|c|}\hline sid & bid & day \\ \hline 22 & 101 & 2026\text{-}02\text{-}01 \\ 22 & 102 & 2026\text{-}02\text{-}03 \\ 31 & 102 & 2026\text{-}02\text{-}02 \\ \hline \end{array}.$$

Intersection gives the tuples common to both inputs. For example, sailors who reserved boat 101 *and* boat 102 can be expressed as

$$\pi_{sid}(\sigma_{bid=101}(r)) \ \cap \ \pi_{sid}(\sigma_{bid=102}(r)).$$

Here

$$\pi_{sid}(\sigma_{bid=101}(r)) = \begin{array}{|c|}\hline sid \\ \hline 22 \\ \hline \end{array}, \qquad \pi_{sid}(\sigma_{bid=102}(r)) = \begin{array}{|c|}\hline sid \\ \hline 22 \\ 31 \\ \hline \end{array},$$

so the result is

$$\begin{array}{|c|}\hline sid \\ \hline 22 \\ \hline \end{array}.$$

Difference is often especially useful in query writing. For instance, sailors with no reservations are

$$\pi_{sid}(s) \setminus \pi_{sid}(r).$$

Since

$$\pi_{sid}(s) = \begin{array}{|c|}\hline sid \\ \hline 22 \\ 31 \\ 44 \\ \hline \end{array}, \qquad \pi_{sid}(r) = \begin{array}{|c|}\hline sid \\ \hline 22 \\ 31 \\ \hline \end{array},$$

the result is

$$\begin{array}{|c|}\hline sid \\ \hline 44 \\ \hline \end{array}.$$

## 6.8 Cartesian product and renaming

The cartesian product $R \times S$ pairs every tuple of $R$ with every tuple of $S$. This is powerful but potentially huge. It is rarely used alone in practical query thinking. It is usually followed by a selection.

If

$$\pi_{sid,sname}(s) = \begin{array}{|c|c|}\hline sid & sname \\ \hline 22 & dustin \\ 31 & lubber \\ 44 & guppy \\ \hline \end{array} \qquad \text{and} \qquad \pi_{bid}(b) = \begin{array}{|c|}\hline bid \\ \hline 101 \\ 102 \\ 103 \\ \hline \end{array},$$

then

$$\pi_{sid,sname}(s) \times \pi_{bid}(b) = \begin{array}{|c|c|c|}\hline sid & sname & bid \\ \hline 22 & dustin & 101 \\ 22 & dustin & 102 \\ 22 & dustin & 103 \\ 31 & lubber & 101 \\ 31 & lubber & 102 \\ 31 & lubber & 103 \\ 44 & guppy & 101 \\ 44 & guppy & 102 \\ 44 & guppy & 103 \\ \hline \end{array}.$$

The size of the result is the product of the input sizes. Even in this tiny example, $3 \times 3 = 9$. In real databases the explosion can be enormous. This is why the warning "do not write a cartesian product unless you really mean it" is good advice.

Renaming, written with $\rho$, is important when attribute names would otherwise clash, especially in self joins. In many derivations, a join is best understood as a filtered cartesian product:

$$R \bowtie_\theta S = \sigma_\theta(R \times S).$$

This identity explains why joins are not mysterious extra operations. They are disciplined combinations of product and selection.

**Definition 6.4: Renaming**

The renaming operator changes the name of a relation and or its attributes without changing the underlying tuples.

Renaming becomes indispensable when we compare a relation with itself. For example, to compare sailors pairwise by age, we create two renamed copies:

$$s_1 = \rho_{sid \to sid1,\ age \to age1}(s), \qquad s_2 = \rho_{sid \to sid2,\ age \to age2}(s).$$

Now the two copies can be joined without ambiguity.

## 6.9 Joins

**Definition 6.5: Theta join**

A theta join combines two relations and keeps only those tuple pairs that satisfy a given condition. It can be defined as a selection on top of a cartesian product.

An equi join is a theta join using equality. A natural join additionally merges attributes with the same name.

For example,

$$INVOICE \bowtie_{INVOICE.customer\_id = CUSTOMER.customer\_id} CUSTOMER$$

combines invoice tuples with the matching customer tuples. The join condition tells us exactly what it means for two tuples to fit together. That explicitness is one reason relational algebra is so good for teaching and optimization.

Using the running instance, an example of a theta join is the "older-than" relation:

$$t = s_1 \bowtie_{age1 > age2} s_2.$$

If we project only the relevant attributes, we obtain

$\pi_{sid1,age1,sid2,age2}(t) =$

| $sid1$ | $age1$ | $sid2$ | $age2$ |
|---|---|---|---|
| 31 | 55 | 22 | 35 |
| 31 | 55 | 44 | 35 |

.

This tells us that sailor 31 is older than sailors 22 and 44.

## 6.10 Equi-join and natural join

An equi-join is a theta join in which the condition is a conjunction of equalities. A natural join goes one step further: it automatically joins on all common attribute names and removes one copy of each common join attribute.

For example,

$$s \bowtie_{sid} r$$

is an equi-join on the common identifier `sid`. It yields

| *sid* | *sname* | *age* | *rating* | *bid* | *day* |
|---|---|---|---|---|---|
| 22 | *dustin* | 35 | 7 | 101 | 2026-02-01 |
| 22 | *dustin* | 35 | 7 | 102 | 2026-02-03 |
| 31 | *lubber* | 55 | 8 | 102 | 2026-02-02 |

.

A natural join example is

$$r \bowtie b,$$

where the common attribute name is `bid`. The result is

| *sid* | *bid* | *day* | *color* |
|---|---|---|---|
| 22 | 101 | 2026-02-01 | *red* |
| 22 | 102 | 2026-02-03 | *green* |
| 31 | 102 | 2026-02-02 | *green* |

.

Natural join is compact and elegant, but it should be used with care. If two relations happen to share an attribute name accidentally, then a natural join may impose an unintended equality condition. In longer derivations, explicit join conditions are often safer and clearer.

## 6.11 Join examples

Relational algebra becomes easier once one sees several join patterns.

A non-equality theta join can express comparison queries. For instance, to list pairs where one sailor is older than another:

$$\pi_{sid1,sid2}\big(s_1 \bowtie_{age1>age2} s_2\big).$$

An equi-join can combine several relations by following explicit key equalities. For example, sailor names together with reserved boat colors can be written as

$$\pi_{sname,color}\Big(\big(s \bowtie_{sid} r\big) \bowtie_{bid} b\Big).$$

If we also want the reservation day, we simply project one more attribute:

$$\pi_{sname,color,day}\Big(\big(s \bowtie_{sid} r\big) \bowtie_{bid} b\Big).$$

Self-joins are often useful for "exists" or comparison queries. For example, sailors who are not the youngest are those for whom someone younger exists:

$$\pi_{sid1,sname}\big(s_1 \bowtie_{age1>age2} s_2\big).$$

Similarly, sailors who share the same age with someone else are

$$\pi_{sid1,sname}\big(s_1 \bowtie_{age1=age2 \wedge sid1 \neq sid2} s_2\big).$$

## 6.12 Self join example: one stop train connection

Suppose we have `Train(origin, destination, start_time, arrival_time)` and want to find one stop connections. We rename one copy of the relation to describe the first leg and another for the second. Then we join them on matching intermediate station and valid timing. This example shows why renaming is essential.

Let

$$t_1 = \rho_{origin \to o1,\ destination \to d1,\ start\_time \to s1,\ arrival\_time \to a1}(Train)$$

and

$$t_2 = \rho_{origin \to o2,\ destination \to d2,\ start\_time \to s2,\ arrival\_time \to a2}(Train).$$

To find connections from `Jyvaskyla` to `Oulu` with exactly one intermediate stop, we can write

$$\pi_{d1,s1,a1,s2,a2}\Big(\sigma_{o1='Jyvaskyla' \wedge d2='Oulu'}\big(t_1 \bowtie_{d1=o2 \wedge s2>a1} t_2\big)\Big).$$

The meaning is best read step by step. The condition $d1 = o2$ says that the destination of the first leg is the origin of the second leg, so the two legs meet at the same intermediate station. The condition $s2 > a1$ says that the second train departs after the first arrives, so the transfer is temporally feasible. Finally, the outer selection fixes the overall origin and final destination.

If the instance is

$Train =$

| *origin* | *destination* | *start_time* | *arrival_time* |
|---|---|---|---|
| *Jyvaskyla* | *Tampere* | 08:00 | 10:30 |
| *Tampere* | *Oulu* | 11:10 | 14:40 |
| *Jyvaskyla* | *Seinajoki* | 09:00 | 10:20 |
| *Seinajoki* | *Oulu* | 10:10 | 13:00 |
| *Seinajoki* | *Oulu* | 10:40 | 13:30 |

,

then the result is

| $X$ | $s1$ | $a1$ | $s2$ | $a2$ |
|---|---|---|---|---|
| *Tampere* | 08:00 | 10:30 | 11:10 | 14:40 |
| *Seinajoki* | 09:00 | 10:20 | 10:40 | 13:30 |

.

The pair `Jyvaskyla→Seinajoki (arrive 10:20)` with `Seinajoki→Oulu (depart 10:10)` is rejected because the transfer condition $s2 > a1$ fails.

This example is pedagogically excellent because it shows three important ideas at once: renaming for self-comparison, joining on a structural condition, and filtering on an additional semantic constraint.

## 6.13 Division

**Definition 6.6: Division**

The division operator answers "for all" type queries. If $R(X, Y)$ and $S(Y)$ are suitable relations, then $R \div S$ returns those $x$ values that are paired in $R$ with every $y$ in $S$.

A classic example is finding students who have completed all courses in a given set. If $R(Student, Course)$ records completions and $S(Course)$ is the required set, then $R \div S$ returns exactly those students paired with every course in $S$. Division feels unusual at first, but it captures an important pattern: not just some of them, but all of them.

More formally, if $P(x_1, \ldots, x_m, y_1, \ldots, y_k)$ and $R(y_1, \ldots, y_k)$, then $P/R$ has schema $(x_1, \ldots, x_m)$ and is defined by

$$P/R = \{\, \mathbf{x} \in \pi_{x_1,\ldots,x_m}(P) \mid \text{for all } \mathbf{y} \in R,\ \mathbf{x} \circ \mathbf{y} \in P \,\},$$

where $\mathbf{x} \circ \mathbf{y}$ denotes tuple concatenation.

A useful sanity check is

$$(A \times B)/B = A,$$

because every $x \in A$ appears together with every $y \in B$ in $A \times B$.

## 6.14 Division as the “for all” operator

Division answers queries of the form

$$\text{find } x \text{ such that for all } y \in Y,\ (x, y) \in P.$$

That is why a good habit is: if a query contains the phrase “for every” or “for all required items,” then division is often the right conceptual operator.

Using the running instance, suppose we want sailors who have reserved boats of *all colors* that occur in the boat relation. First form the relation of actual $(sid, color)$ pairs:

$$p = \pi_{sid,color}(r \bowtie_{bid} b).$$

Then form the set of required colors:

$$y = \pi_{color}(b).$$

These are

$$p = \begin{array}{|c|c|}\hline sid & color \\ \hline 22 & red \\ 22 & green \\ 31 & green \\ \hline \end{array} \qquad y = \begin{array}{|c|}\hline color \\ \hline red \\ green \\ \hline \end{array}.$$

Now division gives

$$p/y = \begin{array}{|c|}\hline sid \\ \hline 22 \\ \hline \end{array}.$$

So only sailor 22 has reserved at least one red boat and at least one green boat.

## 6.15 Youngest sailor using division

Division can also encode comparison queries in a less obvious but elegant way. To find the youngest sailors, reformulate the task as follows: keep those sailors whose age is less than or equal to the age of *every* sailor.

Build the relation

$$p(sid1, sid2) = \{(sid1, sid2) \mid age(sid1) \leq age(sid2)\}.$$

In algebraic form:

$$p = \pi_{sid1,sid2}\Big(\rho_{sid\to sid1,\ age\to age1}(s) \bowtie_{age1\leq age2} \rho_{sid\to sid2,\ age\to age2}(s)\Big).$$

Then divide by the set of all second identifiers:

$$youngest = p \ / \ \pi_{sid2}\Big(\rho_{sid\to sid2}(s)\Big).$$

For the running instance, the relation $p$ becomes

| $sid1$ | $sid2$ |
|---|---|
| 22 | 22 |
| 22 | 31 |
| 22 | 44 |
| 31 | 31 |
| 44 | 22 |
| 44 | 31 |
| 44 | 44 |

and

$$\pi_{sid2}(s_2) = \begin{array}{|c|}\hline sid2 \\ \hline 22 \\ 31 \\ 44 \\ \hline \end{array}.$$

Therefore

$$p/\pi_{sid2}(s_2) = \begin{array}{|c|}\hline sid1 \\ \hline 22 \\ 44 \\ \hline \end{array}.$$

So the youngest sailors are 22 and 44, both with age 35.

This is a beautiful example because division here expresses a universal comparison: keep those identifiers that are appropriately related to all other identifiers.

## 6.16 Division using only the basic operators

Although division is very useful conceptually, it is not one of the minimal classical primitive operators. It can be expressed using projection, product, and set difference:

$$P/R \ = \ \pi_X(P) \ \setminus \ \pi_X\Big((\pi_X(P) \times R) \ \setminus \ P\Big).$$

This formula is worth understanding. Start with all candidate $X$-values. Then build all pairs that *should* exist if a candidate satisfies the "for all" condition. Subtract the pairs that actually exist in $P$. What remains are the missing required pairs. Projecting $X$ identifies the bad candidates, and subtracting them from all candidates leaves exactly the good ones.

This derivation also shows that division adds convenience and conceptual clarity, but not fundamentally new expressive power.

## 6.17 Relational completeness

The classical result is that relational algebra and relational calculus have the same expressive power for the standard class of safe queries. This is often summarized by saying that SQL is relationally complete when it can express the operations corresponding to relational algebra.

It is also helpful to distinguish relational completeness from Turing completeness. A language is *relationally complete* if it can express everything classical relational algebra can express. A language is *Turing complete* if it can compute any computable function. Relational algebra is not Turing complete. For example, plain classical relational algebra cannot express arbitrary transitive closure. But that limitation is not a defect. It reflects the fact that algebra is a carefully designed query formalism rather than a general-purpose programming language.

## 6.18 More examples

A few short examples help consolidate the notation.

Names and ages of sailors with rating greater than 4:

$$\pi_{sname,age}(\sigma_{rating>4}(s)).$$

Sailors with at least one reservation:

$$\pi_{sid}(s \bowtie_{sid} r).$$

Sailors with no reservation:

$$\pi_{sid}(s) \setminus \pi_{sid}(r).$$

Sailors who reserved a green boat *or* a red boat:

$$\pi_{sid}(\sigma_{color='green'}(r \bowtie b)) \cup \pi_{sid}(\sigma_{color='red'}(r \bowtie b)).$$

Sailors who reserved a green boat *and* a red boat:

$$\pi_{sid}(\sigma_{color='green'}(r \bowtie b)) \cap \pi_{sid}(\sigma_{color='red'}(r \bowtie b)).$$

These examples are small, but they illustrate a major strength of algebra: each query pattern becomes a recognizable combination of a few basic operators.

## 6.19 Expression trees and optimization

An algebraic query can be drawn as a tree. Leaves are base relations. Internal nodes are operators. This view makes it easier to reason about alternative but equivalent plans.

Pushing selections and projections early is often beneficial. If fewer rows and columns survive intermediate stages, later operations become cheaper.

**Example 6.1: Selection pushdown**

Suppose we want employees in the IT department together with their department names. A plan that first joins all employees with departments and then filters may do more work than a plan that filters employees first and joins later.

**Theorem 6.1: A common equivalence**

If a selection condition $c$ refers only to attributes of relation $R$, then

$$\sigma_c(R \times S) = \sigma_c(R) \times S.$$

*Proof.* Both sides keep exactly those tuple pairs $(r, s)$ for which $r \in R$, $s \in S$, and $r$ satisfies $c$. Since the condition does not mention attributes of $S$, the membership test is independent of $s$. □

Another useful equivalence is associativity of join:

$$A \bowtie (B \bowtie C) \equiv (A \bowtie B) \bowtie C.$$

This allows the optimizer to reorder joins logically. Different join orders may have very different costs even though they are semantically equivalent.

Projection pushdown is also important, provided we keep the attributes needed later for joins or output. The general rule of thumb is simple and powerful: select and project as early as possible.

## 6.20 Equivalence of relational algebra expressions

**Definition 6.7: Equivalence of relational algebra expressions**

Two relational algebra expressions $E$ and $F$ are equivalent, written $E \equiv F$, if they produce the same result for all valid inputs.

This notion is central for optimization. The DBMS is allowed to replace one algebra tree by an equivalent one if the new form is expected to be cheaper.

For example,

$$A \cap B \equiv A \setminus (A \setminus B)$$

and

$$\sigma_{a=10 \wedge b=a}(A) \equiv \sigma_{a=b}\Big(\sigma_{b=10}(A)\Big).$$

These identities show that the same query meaning can often be written in different but equivalent ways.

## 6.21 RA trees

RA expressions can be represented as trees and evaluated bottom-up. For example,

$$b \bowtie (r \bowtie \sigma_{age \leq 40}(s))$$

corresponds to a tree whose leaves are $b$, $r$, and $s$, and whose internal nodes are $\sigma$ and $\bowtie$.

This representation is more than a picture. It is the form in which optimizers typically reason about query plans. The DBMS can transform one tree into another equivalent tree and estimate the cost of each candidate.

A basic translation from SQL to algebra helps make this concrete. The SQL pattern

$$\texttt{SELECT } a_1, \ldots, a_k \texttt{ FROM } R_1, \ldots, R_m \texttt{ WHERE } C$$

corresponds, at the core level, to

$$\pi_{a_1,\ldots,a_k}\Big(\sigma_C(R_1 \times \cdots \times R_m)\Big).$$

An optimizer then recognizes that explicit joins are better than raw products followed by selections, pushes filters downward, and looks for a low-cost join order.

## 6.22 Join trees and search space

For a query involving many joins, there are often many equivalent join trees. For instance, with four relations $A, B, C, D$, the following are equivalent in meaning:

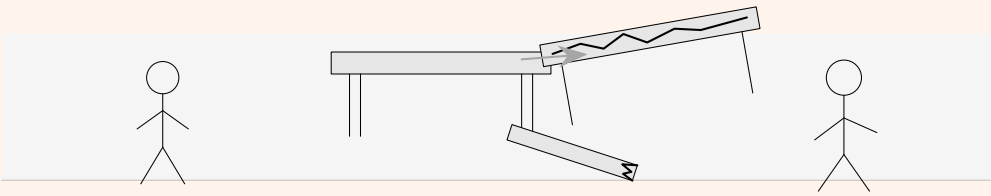
$$((A \bowtie B) \bowtie C) \bowtie D, \qquad (A \bowtie (B \bowtie C)) \bowtie D, \qquad (A \bowtie B) \bowtie (C \bowtie D).$$

But they may have very different costs.

The number of full binary join trees grows according to the Catalan numbers. This growth is rapid, which is why optimizers cannot simply enumerate all possible plans in large queries without guidance. They use algebraic equivalences, heuristics, and cost estimates based on statistics and indexes.

This is one of the main practical reasons relational algebra matters. It is not only a teaching language. It is also the language in which query plans become mathematically manipulable.

**Joke box**



**A mess** has appeared in the kitchen. One kitchen table is smashed into another, and a table is broken in half.
**Child:** "The teacher was telling us how to ***divide*** tables by using other tables."
**Mother:** "Oh no..."
**Child:** "But don't worry, mother. Our teacher also told us also how to ***join*** the tables again."

## 6.23 Review questions and small exercises

1. Why is relational algebra useful even if we already have SQL?
2. Explain the difference between selection and projection.
3. What does union compatibility mean?
4. Why can the Cartesian product be dangerous if used carelessly?
5. What is the difference between a theta join and a natural join?
6. Express in words what $\pi_A(\sigma_c(R))$ does.
7. Build a relational algebra expression for students enrolled in a course named Databases.

## 6.24 Further reading

For relational algebra and formal query thinking, the most valuable sources are foundational relational texts and theory-oriented database books. They help explain optimization, equivalence of expressions, and the connection between algebra and declarative query languages.

# Bibliography

[1] C. J. Date. *An Introduction to Database Systems*. Addison-Wesley.

[2] J. D. Ullman. *Principles of Database and Knowledge-Base Systems*. Computer Science Press.

[3] R. Elmasri and S. B. Navathe. *Fundamentals of Database Systems*. Pearson.

## 6.25 Wrap-up

Relational algebra gives us a precise language for what SQL systems do under the hood. It clarifies query meaning, supports equivalence-based optimization, and explains why good plans often filter and project early and choose join orders carefully. The next chapter uses this formal mindset to improve schemas. We study redundancy, dependencies, and normal forms.

# Chapter 7

# Schema Refinement and Normal Forms

**Chapter guidance**

This chapter explains why some schemas create redundancy and anomalies and how decomposition can repair them. It introduces functional dependencies, closure, keys, lossless join, dependency preservation, BCNF, 3NF, and 4NF. Read it carefully, because this is where design quality becomes a precise topic.

## 7.1 The evils of redundancy

Redundancy means that the same fact is stored several times. Redundancy is not always evil, but uncontrolled redundancy often causes anomalies. An insertion anomaly appears when one fact cannot be stored without another unrelated fact. A deletion anomaly appears when deleting one fact accidentally removes another. An update anomaly appears when the same fact must be changed in many places and may become inconsistent.

This chapter is often the point where database design becomes intellectually satisfying. Up to now, we have used good judgment when choosing entities, keys, and tables. Normalization adds formal tools that explain why one design is safer than another. The historical development is also worth knowing. Codd introduced early normal forms, Boyce and Codd sharpened the ideas further, and later work such as Fagin's extended the theory to multivalued dependencies and beyond.

A simple example shows the problem. Suppose a relation stores

$$Employee(EmpId, EmpName, Dept, Manager).$$

If each employee belongs to exactly one department, and each department has exactly one manager, then the manager name is repeated in every tuple of that department. If the manager changes, many rows must be updated. If the last employee of a department is deleted, the information about the department manager may disappear as well. This is the basic motivation for schema refinement.

## 7.2 A note on 1NF and 2NF

The normal forms are historically layered. It is useful to situate BCNF and 3NF among them.

- **First Normal Form (1NF)** rules out repeating groups and nested list structure. In this book, 1NF is usually assumed from the start, because we work with atomic attribute values.
- **Second Normal Form (2NF)** addresses partial dependency on part of a composite key. It is historically important, but weaker than 3NF and BCNF.
- **Third Normal Form (3NF)** and **BCNF** are the main focus here because they address redundancy caused by functional dependencies in a more systematic way.

This is why many modern treatments move quickly to BCNF and 3NF after briefly recalling 1NF and 2NF.

## 7.3 Functional dependencies

**Definition 7.1: Functional dependency**

Let $R$ be a relation schema and let $X$ and $Y$ be sets of attributes of $R$. We say that $X \to Y$ is a *functional dependency* if in every valid instance of $R$, any two tuples that agree on $X$ also agree on $Y$.

Functional dependencies capture a key kind of semantic rule. If employee ID determines employee name and department, then `EmpId` $\to$ `Name, Dept`.

It is important to read an FD semantically, not procedurally. The statement $X \to Y$ does not mean that the database computes $Y$ from $X$ by a formula. It means that the value of $X$ determines the value of $Y$ in the modeled world. For example, if a student number uniquely identifies a student, then the student number determines the student's name. This is a statement about meaning and uniqueness, not about arithmetic derivation.

## 7.4 Example: hourly employees

Suppose a relation stores employee, department, and hourly wage information. If an employee belongs to exactly one department, then employee determines department. If department determines manager, another dependency appears. These dependencies may create repeated information and motivate decomposition.

Consider for instance

$$HourlyEmp(EmpId, EmpName, Dept, Manager, HourlyWage).$$

If the intended semantics are

$$EmpId \to EmpName, Dept, HourlyWage \qquad \text{and} \qquad Dept \to Manager,$$

then the manager of a department is repeated for every employee in that department. This is exactly the kind of redundancy normalization aims to remove.

## 7.5 Full, partial, and transitive dependency

To understand why 2NF and 3NF were introduced historically, it is helpful to name three common dependency patterns.

**Definition 7.2: Full functional dependency**

A dependency $X \to Y$ is a *full functional dependency* if $Y$ depends on all attributes of $X$, and on no proper subset of $X$.

**Definition 7.3: Partial dependency**

A dependency $X \to Y$ is a *partial dependency* if some proper subset of $X$ already determines $Y$.

**Definition 7.4: Transitive dependency**

A dependency $X \to Z$ is *transitive* if there is an intermediate set $Y$ such that $X \to Y$ and $Y \to Z$, where $Y$ is not itself just a trivial reformulation of $X$.

A standard example of partial dependency is

$$Enrollment(StudentId, CourseId, StudentName, Grade)$$

with key $(StudentId, CourseId)$ and FD

$$StudentId \to StudentName.$$

Here `StudentName` depends only on part of the composite key, namely `StudentId`, so the dependency is partial.

A standard example of transitive dependency is

$$EmpId \to Dept \qquad \text{and} \qquad Dept \to Manager,$$

which together imply

$$EmpId \to Manager.$$

The manager depends on the employee only through the department.

## 7.6 Armstrong's axioms

Armstrong's axioms give a sound and complete inference system for functional dependencies.

**Theorem 7.1: Armstrong's axioms**

The following inference rules are sound for functional dependencies:

1. Reflexivity: if $Y \subseteq X$, then $X \to Y$.
2. Augmentation: if $X \to Y$, then $XZ \to YZ$.

3. Transitivity: if $X \to Y$ and $Y \to Z$, then $X \to Z$.

They are also complete, meaning every logically implied functional dependency can be derived from them.

These three rules are small in number but very powerful. Reflexivity says that an attribute set always determines any part of itself. Augmentation says that adding the same context to both sides preserves implication. Transitivity allows chains of determination to be combined.

Derived rules such as union and decomposition are convenient in practice, even though they follow from the three basic axioms.

**Theorem 7.2: Derived inference rules**

The following rules follow from Armstrong's axioms.

1. **Union**: if $X \to Y$ and $X \to Z$, then $X \to YZ$.
2. **Decomposition**: if $X \to YZ$, then $X \to Y$ and $X \to Z$.

These two rules are used constantly in design exercises. Union lets us combine consequences with the same left hand side. Decomposition lets us break a large right hand side into smaller pieces that are easier to test against BCNF or 3NF definitions.

A third convenient derived rule is often called *pseudotransitivity*: if $X \to Y$ and $WY \to Z$, then $WX \to Z$. One can derive it from the basic axioms, but in longer exercises it is often mentally used directly.

## 7.7 Closure of attribute sets

Computing a closure is best learned as a small mechanical procedure. Start with the attributes already in $X$. Then repeatedly scan the dependency set and add every attribute that becomes implied by what you already have. Stop only when one full pass adds nothing new. This may sound simple, and it is, but that simplicity is exactly why closure is such a useful practical test in design exercises.

**Definition 7.5: Closure of an attribute set**

Given a set of functional dependencies $F$ and an attribute set $X$, the *closure* $X^+$ is the set of all attributes functionally determined by $X$ under $F$.

Closures help us test whether a set is a superkey and whether a dependency follows from a given dependency set.

It is useful to distinguish $X^+$ from $F^+$. The symbol $X^+$ means the closure of one attribute set under the dependency set $F$. The symbol $F^+$ means the set of all FDs implied by $F$. The first is a set of attributes. The second is a set of dependencies.

**Example 7.1: Closure computation**

Let

$$R(A, B, C, D, E), \qquad F = \{A \to B,\ B \to C,\ AC \to D,\ D \to E\}.$$

Compute $A^+$.
Start with $A^+ = \{A\}$.
From $A \to B$, add $B$:

$$A^+ = \{A, B\}.$$

From $B \to C$, add $C$:

$$A^+ = \{A, B, C\}.$$

Now $A$ and $C$ are both present, so $AC \to D$ applies:

$$A^+ = \{A, B, C, D\}.$$

From $D \to E$, add $E$:

$$A^+ = \{A, B, C, D, E\}.$$

Thus $A^+ = ABCDE$, so $A$ is a superkey for this schema.

## 7.8 Keys and superkeys revisited

A set of attributes is a superkey if its closure contains all attributes of the relation. It is a candidate key if it is a minimal superkey. This connects key reasoning directly with functional dependencies.

For example, if $R(A, B, C, D)$ and $F = \{A \to B,\ A \to C,\ A \to D\}$, then

$$A^+ = ABCD,$$

so $A$ is a superkey. Since no proper subset of $A$ exists, it is also a candidate key.

With larger sets one must also check minimality. If $AB^+ = ABCD$, then $AB$ is a superkey. But if already $A^+ = ABCD$, then $AB$ is not a candidate key because $B$ is unnecessary.

## 7.9 Lossless join decomposition

**Definition 7.6: Decomposition**

A *decomposition* of a relation schema $R$ is a replacement of $R$ by smaller relation schemas whose attributes together cover the attributes of $R$.

**Definition 7.7: Lossless join decomposition**

A decomposition of a relation schema into smaller schemas is *lossless join* if joining the decomposed relations always recreates exactly the original relation for every valid instance.

Lossless join is essential. If decomposition loses information or creates spurious tuples, it is not acceptable.

**Theorem 7.3: Binary lossless join test**

A decomposition of relation schema $R$ into $R_1$ and $R_2$ is lossless with respect to a dependency set $F$ if

$$(R_1 \cap R_2) \to R_1 \quad \text{or} \quad (R_1 \cap R_2) \to R_2$$

can be derived from $F$.

This criterion is very practical. The common attributes must determine one whole side of the decomposition. Intuitively, the overlap must be strong enough to glue the two parts back together without ambiguity.

**Example 7.2: Lossless join test**

Let $R(ABCE)$ and $F = \{C \to E\}$. Consider the decomposition into $ABC$ and $BCE$.
The common attributes are

$$ABC \cap BCE = BC.$$

Since $C \to E$, augmentation gives $BC \to BCE$. Therefore

$$(ABC \cap BCE) \to BCE,$$

so the decomposition is lossless.

## 7.10 A lossy join example and spurious tuples

Lossless decomposition is so important because a bad decomposition can create tuples that never existed in the original data.

Consider the relation instance

$$r(A, B, C) = \begin{array}{|c|c|c|} \hline A & B & C \\ \hline 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 2 & 8 \\ \hline \end{array}$$

and decompose it into $AB$ and $BC$:

$$\text{proj}_{AB}(r) = \begin{array}{|c|c|} \hline A & B \\ \hline 1 & 2 \\ 4 & 5 \\ 7 & 2 \\ \hline \end{array} \qquad \text{proj}_{BC}(r) = \begin{array}{|c|c|} \hline B & C \\ \hline 2 & 3 \\ 5 & 6 \\ 2 & 8 \\ \hline \end{array}.$$

Now join them again:

$$\text{proj}_{AB}(r) \bowtie \text{proj}_{BC}(r) = \begin{array}{|c|c|c|}\hline A & B & C \\ \hline 1 & 2 & 3 \\ 1 & 2 & 8 \\ 4 & 5 & 6 \\ 7 & 2 & 3 \\ 7 & 2 & 8 \\ \hline \end{array}.$$

The tuples $(1, 2, 8)$ and $(7, 2, 3)$ were not in the original relation. They are *spurious tuples*. So this decomposition is lossy:

$$\text{proj}_{AB}(r) \bowtie \text{proj}_{BC}(r) \neq r.$$

This example is worth remembering because it shows concretely what can go wrong. A decomposition is not justified merely because the smaller tables look tidy. They must also preserve the original information exactly.

## 7.11 BCNF and 3NF

**Definition 7.8: BCNF**

A relation schema is in *Boyce Codd Normal Form* if for every nontrivial functional dependency $X \to Y$, the set $X$ is a superkey.

**Definition 7.9: Third Normal Form**

A relation schema is in *Third Normal Form* if for every nontrivial functional dependency $X \to A$, either $X$ is a superkey or $A$ is a prime attribute, meaning it belongs to some candidate key.

BCNF is stricter. 3NF is slightly weaker but often useful because it can preserve dependencies in cases where BCNF decomposition cannot.

A quick comparison helps. In BCNF, every determinant must be a superkey. In 3NF, an FD is also allowed when the right-hand side is prime. That one extra allowance is exactly what makes 3NF more flexible in dependency-preserving design.

Also note that

$$\text{BCNF} \implies \text{3NF},$$

but the converse is not always true.

## 7.12 Why BCNF is needed

Suppose $R(A, B, C)$ has FDs

$$A \to B, \qquad B \to C.$$

If $A$ is a key, then $A \to B$ is harmless with respect to BCNF, because the determinant $A$ is a key. But $B \to C$ violates BCNF if $B$ is not a superkey.

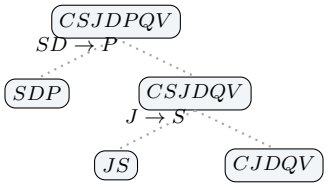



Figure 7.1: One BCNF decomposition tree for the example $CSJDPQV$ with $SD \to P$ and $J \to S$.

The redundancy is easy to see. If the same $B$-value appears in many tuples, then the corresponding $C$-value is repeated in every one of them. Updating $C$ requires many changes, and inconsistencies become possible. BCNF eliminates exactly this sort of dependency-based redundancy.

## 7.13 Dependency preservation

**Definition 7.10: Dependency preservation**

A decomposition is *dependency preserving* if the original dependencies can be enforced by checking the decomposed relations individually, without requiring a join.

Dependency preservation is practically important because checking dependencies across joins may be expensive.

The trade-off with BCNF is fundamental. BCNF can always be reached by lossless decomposition, but not always in a dependency-preserving way. By contrast, 3NF always allows a dependency-preserving decomposition, which is one reason it remains important in practice.

## 7.14 BCNF decomposition

BCNF decomposition repeatedly splits a relation on dependencies that violate BCNF. The result is lossless, but not always dependency preserving. This trade off is central to the design discussion.

A common binary rule is this: if $U \to V$ holds and violates BCNF, then decompose $R$ into

$$UV \qquad \text{and} \qquad R - V.$$

This decomposition is lossless. The procedure is then repeated recursively until every resulting relation is in BCNF.

A useful example starts from a relation with attributes $CSJDPQV$ and functional dependencies $SD \to P$ and $J \to S$. If we decompose first on $SD \to P$, we obtain the lossless path shown in figure 7.1. The key point is that the common attributes in each split are sufficient to reconstruct the original information by join.

The resulting decomposition is

$$\{SDP,\ JS,\ CJDQV\},$$

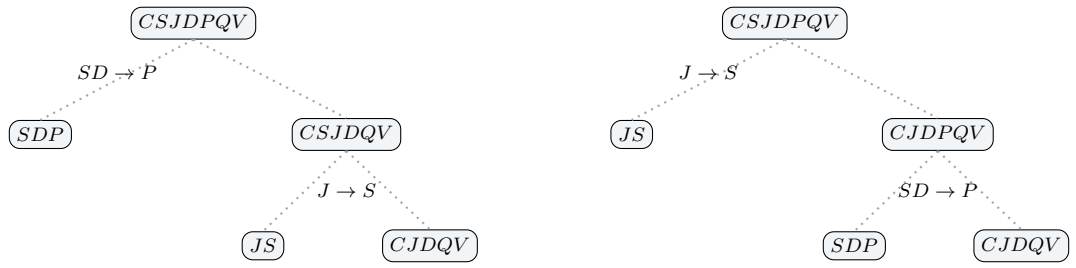



Figure 7.2: BCNF decomposition is not unique. Different violating dependencies can be chosen first.

and each relation is in BCNF.

What matters here is not only the final answer but also the process: find a violating dependency, decompose, and repeat.

What matters pedagogically is that BCNF decomposition is not unique. Different violating dependencies may be chosen first. The alternative in figure 7.2 starts with $J \to S$ instead. Both decompositions are lossless, but their intermediate shapes differ.

## 7.15 Minimal covers and the 3NF synthesis idea

A minimal cover is a cleaned up version of a dependency set that keeps the same meaning while removing unnecessary parts. The algorithm is worth remembering because it is used in systematic 3NF synthesis.

1. Decompose every right hand side so that each FD has a single attribute on the right.
2. Try to remove extraneous attributes from left hand sides.
3. Remove redundant dependencies one by one if they are implied by the rest.

Different orders can lead to different minimal covers, but all correct results are equivalent in meaning. Once a minimal cover is available, one standard 3NF procedure creates a relation for each left hand side together with the attributes it determines, and then makes sure that some candidate key is represented. This is why 3NF is so attractive: unlike BCNF, a dependence-preserving decomposition is always obtainable.

The ordering of the minimal-cover procedure matters. First split right-hand sides, then simplify left-hand sides, and only after that test whole dependencies for redundancy. If one changes the order carelessly, it becomes harder to see what is truly extraneous.

**Example 7.3: Minimal cover**

Let

$$F = \{AB \to CD,\ D \to E,\ A \to BE\}.$$

**Step 1: split right-hand sides.**

$$AB \to C, \quad AB \to D, \quad D \to E, \quad A \to B, \quad A \to E.$$

**Step 2: remove extraneous attributes from left-hand sides.** Since $A \to B$ already holds, the $B$ in $AB \to C$ is extraneous, so

$$AB \to C \quad \Rightarrow \quad A \to C.$$

Likewise,

$$AB \to D \quad \Rightarrow \quad A \to D.$$

**Step 3: remove redundant dependencies.** Now $A \to E$ is redundant because

$$A \to D \quad \text{and} \quad D \to E$$

already imply $A \to E$.
So one minimal cover is

$$F_0 = \{A \to C,\ A \to D,\ D \to E,\ A \to B\}.$$

## 7.16 Why 3NF is sometimes preferred

If preserving dependencies directly is very important, a 3NF design may be chosen. This is one reason 3NF continues to matter even though BCNF is cleaner from a redundancy point of view.

In practice, the question is often this: is it more important to reduce redundancy as far as possible, or to enforce the important dependencies locally on the decomposed tables? BCNF gives the stronger cleanliness condition. 3NF gives more flexibility for dependency preservation. Good design balances both concerns.

## 7.17 Multivalued dependencies and 4NF

**Definition 7.11: Multivalued dependency**

A *multivalued dependency* $X \twoheadrightarrow Y$ expresses that for each value of $X$, the set of $Y$ values is independent of the remaining attributes.

A standard example involves restaurants, pizza varieties, and delivery areas. If the pizza varieties offered by a restaurant are independent of the delivery areas it serves, then storing all three together causes unnecessary repetition.

More concretely, if a restaurant offers two pizza varieties and serves two delivery areas, then the combined table must contain all four pairings, even though the variety choice and area choice are logically independent. That is redundancy of a different kind from ordinary functional-dependency redundancy.

**Definition 7.12: Fourth Normal Form**

A relation schema is in *Fourth Normal Form* if for every nontrivial multivalued dependency $X \twoheadrightarrow Y$, the set $X$ is a superkey.

A small instance makes the redundancy visible. Suppose

$$R(Restaurant, Pizza, Area)$$

contains

| $r$ | $p$ | $d$ |
|---|---|---|
| *napoli* | *margherita* | *center* |
| *napoli* | *margherita* | *north* |
| *napoli* | *diavola* | *center* |
| *napoli* | *diavola* | *north* |

because the restaurant *napoli* independently offers pizzas $\{margherita, diavola\}$ and serves areas $\{center, north\}$. Then the 4NF decomposition is

$$RP(Restaurant, Pizza) \qquad \text{and} \qquad RD(Restaurant, Area).$$

This removes the combinatorial repetition.

**Joke box**

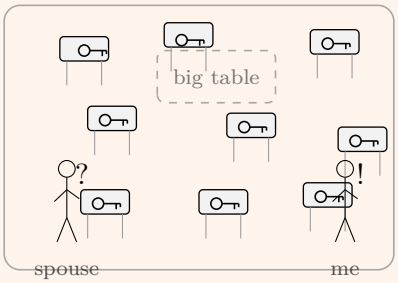
A mess: the big kitchen table is gone. Now there are many small tables all over the kitchen.
The keys that were once neatly on the single kitchen table are distributed across them.

**Spouse:** “Oh, darling, what did you do with the kitchen?!”

**Me:** “No worries, Kulta, I just wanted to normalize our relationship.”

**Spouse:** “...?”

(Moral: decomposition reduces redundancy, but too many relationships can make life complicated.)




## 7.18 Review questions and small exercises

1. What problem does normalization try to solve?
2. Explain the difference between a full functional dependency and a partial dependency.
3. What is a transitive dependency?
4. What does 2NF remove that 1NF does not?

5. What is the key idea of 3NF?
6. Why is BCNF stricter than 3NF?
7. What does lossless decomposition mean?
8. Why can very strong normalization sometimes create practical costs?

## 7.19 Further reading

Normalization is one of the most elegant parts of database design because it turns vague design quality into concrete tests. Date gives a careful conceptual view, while textbook treatments by Elmasri and Navathe and by Ramakrishnan and Gehrke connect theory to examples and decomposition algorithms.

# Bibliography

[1] C. J. Date. *Database Design and Relational Theory*. O'Reilly.

[2] R. Elmasri and S. B. Navathe. *Fundamentals of Database Systems*. Pearson.

[3] R. Ramakrishnan and J. Gehrke. *Database Management Systems*. McGraw-Hill.

## 7.20 Wrap-up

Normal forms teach disciplined schema design. They show that design quality is not just taste. It can be analyzed. In the next chapter we zoom out from schema refinement toward warehousing, distribution, and broader database paradigms.

# Chapter 8

# Data Warehousing, Distribution, and Database Paradigms

**Chapter guidance**

This chapter broadens the view from operational databases to analytical systems and distributed settings. It explains data warehouses, ETL, data marts, star schemas, client server architecture, parallel hardware styles, replication, sharding, and the rise of multiple database paradigms.

## 8.1 Core abbreviations

This chapter uses common abbreviations such as OLTP, OLAP, ETL, DW, and NoSQL. A summary table appears at the end of the book.

A few of these abbreviations are especially central here:

- **OLTP** stands for Online Transaction Processing and refers to operational systems that process day-to-day business events.
- **OLAP** stands for Online Analytical Processing and refers to read-heavy analysis over large collections of historical data.
- **ETL** stands for Extract, Transform, Load.
- **DW** stands for Data Warehouse.
- **NoSQL** is often read as "Not only SQL," emphasizing that the modern ecosystem includes several non-relational paradigms alongside relational systems.

## 8.2 What is a data warehouse?

**Definition 8.1: Data warehouse**

A *data warehouse* is a repository designed primarily for integrated, historical, and analytical use rather than day to day transaction processing.

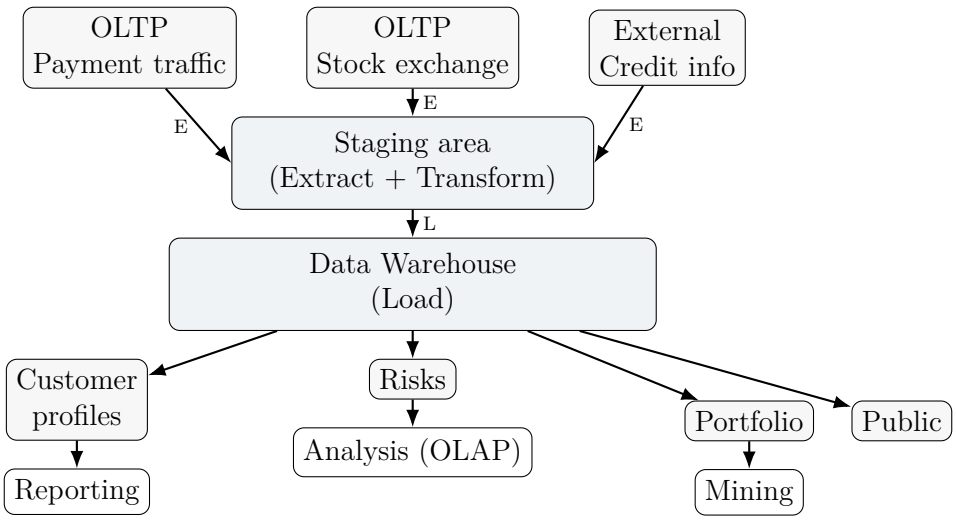



Figure 8.1: A typical warehouse architecture with ETL, a central warehouse, and data marts.

Operational systems usually record current business events. A warehouse supports analysis across time and across source systems.

The difference in purpose is fundamental. An operational system is designed to record that one payment happened, one order was placed, or one account balance changed. A warehouse is designed to support questions such as these:

- How did sales evolve by region over the last three years?
- Which customer groups became more risky during the last quarter?
- Which product categories perform best by season and channel?

These are cross-sectional, historical, and often aggregating questions. A warehouse is therefore not merely a backup copy of an operational database. It is an analytical environment with different design goals.

## 8.3 Warehouse architecture and ETL

A common warehouse architecture includes source systems, a staging area, ETL processes, a central warehouse, and sometimes specialized data marts.

**Definition 8.2: ETL**

*ETL* stands for Extract, Transform, Load. It is the process of taking data from source systems, cleaning and transforming it, and loading it into a warehouse.

The staging area exists because raw incoming data often needs validation, cleaning, harmonization, and integration before it can be trusted. Source systems may use different codes, naming conventions, date formats, currencies, or levels of precision. ETL turns these into a more coherent structure.

Figure 8.1 shows the standard flow. Several source systems feed a staging area. The staging area performs extraction and transformation work. The cleaned and integrated

result is loaded into the warehouse. From there, focused marts and analytical applications are supplied.

## 8.4 Staging area: why it exists

The staging area deserves special emphasis because it is often misunderstood. It is not just a temporary parking place. It is where the difficult integration work happens.

Typical staging tasks include:

- unifying formats and codes,
- filtering erroneous or poor-quality data,
- reconciling duplicates,
- deriving new attributes from business rules,
- matching data from heterogeneous source systems.

Data is often loaded periodically in batch mode, for example daily, weekly, monthly, or quarterly. In some organizations, corrected data may even be fed back into operational systems after cleaning and reconciliation.

## 8.5 Data marts

One motivation for data marts is organizational reality. Different parts of an organization often ask different questions. Finance may ask about monthly revenue, overdue invoices, and budget variance. Logistics may ask about stock levels, delivery times, and returns. A shared warehouse can feed both, but each team often benefits from a smaller, more focused analytical view.

A *data mart* is a focused subset of warehouse data for a specific department or analysis need. Sales, finance, and logistics teams may each use their own mart. This can improve clarity and performance.

A mart can be understood as a thematic slice of the larger warehouse. In some architectures, marts are built strictly from the enterprise warehouse. In others, they may combine warehouse data with selected external sources. The central idea stays the same: each mart is organized around a particular analytical purpose rather than the whole enterprise at once.

## 8.6 Star, snowflake, and starflake schemas

Analytical schemas often separate fact tables from dimension tables. A *star schema* keeps dimensions mostly denormalized. A *snowflake schema* normalizes dimensions more. A *starflake* lies somewhere between them.

The choice affects simplicity, redundancy, and query performance.

A fact table stores measurable events such as sales amounts, quantities, balances, or transfers. Dimension tables store descriptive context such as customer, product, region, channel, or time. This makes many business questions easy to formulate as aggregations over facts grouped by dimensions.

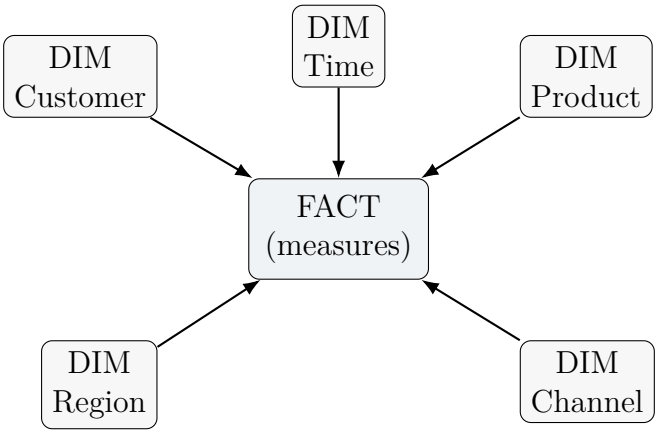



Figure 8.2: A star-style analytical schema: one fact table connected to several dimension tables.

In a star schema, dimensions are kept broad and descriptive, which often simplifies analytical queries. In a snowflake schema, dimension tables are normalized further, which can reduce redundancy but also adds joins. A starflake schema is a mixed compromise: some dimensions remain denormalized while others are normalized more deeply.

## 8.7 OLTP versus OLAP

The contrast between OLTP and OLAP is easier to remember through examples. A webshop that records one order, one payment, or one address change at a time is doing OLTP work. A manager who asks for revenue by region over the last three years is doing OLAP work. The first workload values short correct updates. The second values broad summaries across many rows. This difference explains why design and tuning choices often diverge.

**Definition 8.3: OLTP and OLAP**

*OLTP* stands for Online Transaction Processing and emphasizes many small updates under correctness constraints. *OLAP* stands for Online Analytical Processing and emphasizes large read heavy analytical queries.

These workloads differ enough that the same schema style is not always ideal for both.

A compact comparison is useful:

- OLTP focuses on daily business events, small updates, and current state.
- OLAP focuses on historical data, aggregation, analysis, planning, and decision support.
- OLTP tends to favor highly normalized schemas.
- OLAP often favors dimensional schemas such as star or snowflake.

## 8.8 Master data

Master data refers to core business entities such as customers, products, and suppliers. A difficult real world problem is that the same customer may appear in several operational systems with slightly different identifiers or attributes. Master data management tries to establish trusted reference records.

Two useful terms are:

- **SOE**, or *system of entry*, the accepted source in which some category of data is entered or maintained.
- **SOR**, or *system of record*, the trusted version regarded as correct for enterprise-wide use.

For example, a web shop may be the SOE for delivery addresses, a CRM system may be the SOE for marketing preferences, and a billing system may be the SOE for invoicing addresses. The organization still needs one trusted customer record for reporting and cross-system use. That is the role of the SOR.

## 8.9 From warehouse to data lake

A data lake stores large amounts of raw or semi processed data in more flexible form. This is attractive for exploratory analytics and machine learning, but it also brings risk.

The contrast can be summarized as follows:

- A **data warehouse** aims for clean, structured, integrated, curated data under a fairly explicit target schema.
- A **data lake** accepts raw, semi-structured, or weakly structured data in more flexible form.

This flexibility is useful when the future analytical questions are not yet known in advance. But flexibility without discipline can destroy usability.

**Remark 8.1: Lake versus swamp**

A data lake without metadata discipline, governance, and quality control can become a *data swamp*. The nice name disappears, but the problem stays.

A swamp is not defined by size alone. It is defined by lack of metadata, lack of trusted provenance, unclear semantics, and poor governance. In such a state, data exists physically but becomes hard to find, trust, or reuse.

## 8.10 Three tier architecture

Traditional file server or tightly coupled two tier architectures often struggle to scale and maintain clean separation of concerns. A three tier architecture separates presentation, application logic, and data management. This supports better maintainability and often better scalability.

The three layers are usually understood as:

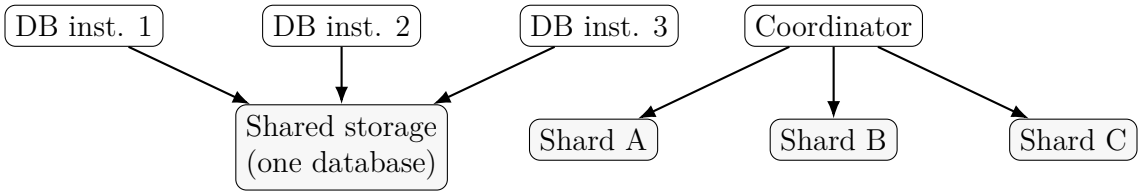



Figure 8.3: Shared disk on the left and shared nothing on the right. Shared nothing is especially important for horizontal scaling.

- **presentation tier**, where users interact with the system,
- **application tier**, where business logic is executed,
- **data tier**, where persistent data is stored and queried.

This separation is valuable because it keeps user interface concerns apart from business rules and storage concerns. It also makes evolution easier: one can change the presentation layer without redesigning the data layer, or improve the data layer without changing the visible interface immediately.

## 8.11 Parallel hardware architectures

Distributed and parallel data systems can use several styles:

- shared memory
- shared disk
- shared nothing

Shared nothing architectures are especially influential in large scale systems because they allow horizontal scaling by partitioning data and work across nodes.

A brief intuition helps:

- In **shared memory**, many processors share the same main memory. This is common inside a single machine.
- In **shared disk**, several DB instances access the same storage.
- In **shared nothing**, each node owns its own local resources, and the data is partitioned across nodes.

## 8.12 Replication and sharding

**Definition 8.4: Replication and sharding**

*Replication* means storing copies of data on multiple nodes. *Sharding* means partitioning data so different nodes store different subsets.

Replication improves availability and read scalability. Sharding improves write scalability and total capacity. Both create consistency questions.

It helps to compare them directly:

- In replication, each copy stores all or almost all of the relevant data.
- In sharding, each node stores only a slice of the overall dataset.

Typical large-scale systems often combine both ideas. Data is first partitioned into shards, and each shard is then replicated for fault tolerance.

Hash-based routing is common for sharding. A key is mapped to a shard, often through a hash or consistent-hashing style mechanism, so that requests can be routed efficiently and rebalancing remains manageable when nodes join or leave.

## 8.13 Replication configurations

A common distinction is between master–slave and peer-to-peer replication.

In a **master–slave** or **primary–secondary** configuration, writes go to one primary node and are propagated to secondaries. Reads may often be served from secondaries. This is conceptually simple and widely used.

In a **peer-to-peer** configuration, several nodes may accept reads and writes. This increases flexibility, but conflict handling and consistency management become harder.

Another important distinction is:

- **synchronous replication**, where a write is acknowledged only after replication to other nodes or a quorum,
- **asynchronous replication**, where the write is acknowledged immediately and replicas catch up later.

Synchronous replication improves consistency but may increase latency. Asynchronous replication improves latency and often availability but may tolerate temporary inconsistency.

## 8.14 Sharding and routing

Sharding needs a routing mechanism. An application or coordinator must know which shard should receive a given request.

Figure 8.4 shows the basic idea. A routing layer maps the shard key to the responsible shard. This can be done by range partitioning, hashing, or related strategies. The main design challenge is to balance efficient routing, even distribution of load, and manageable rebalancing as the cluster changes.

## 8.15 CAP style trade offs

In distributed systems, designers often discuss trade offs among consistency, availability, and partition tolerance. Even without going into every formal detail, the practical message is clear: distribution makes life more powerful and more complicated at the same time.

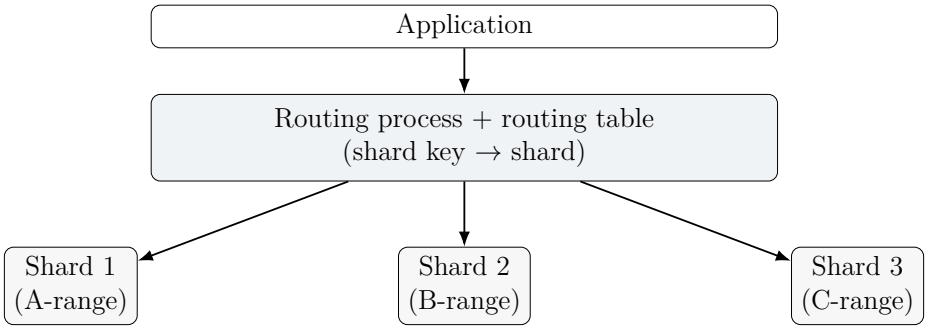



Figure 8.4: A simple sharding picture: requests are routed to the correct shard through a shard key.

The CAP intuition is this:

- **Consistency**: all clients observe data that is mutually coherent according to the chosen consistency model.
- **Availability**: requests receive responses rather than simply failing when the system is under stress.
- **Partition tolerance**: the system continues operating even when communication between some nodes is broken.

The key practical point is that under network partition, difficult design decisions must be made. Distribution therefore creates trade-offs that do not arise in the same way on a single machine.

A related acronym is **BASE**: Basically Available, Soft state, Eventually consistent. This vocabulary is often used to describe systems that relax strong immediate consistency in exchange for scalability and availability.

## 8.16 Database paradigms

Not all useful databases are relational. Key value stores, document stores, column family stores, graph databases, and time series systems all serve different needs. The rise of NoSQL did not kill SQL. It widened the ecosystem.

The relational model remains central because it is mathematically clean and highly expressive for many business applications. But other paradigms fit better when the shape of the data, the scale of the workload, or the dominant query pattern differs.

It is also useful to place object-oriented and object-relational systems in this landscape:

- **Object-oriented databases** store objects with identity, attributes, and methods more directly.
- **Object-relational databases** extend the relational model with richer type support and object-like features while preserving a strong relational core.

## 8.17 NoSQL families

Four common NoSQL paradigms are especially worth knowing:

- **key-value stores**, which map keys to values and are optimized for fast key-based access,
- **document stores**, which store structured documents such as JSON and support queries inside documents,
- **graph databases**, which emphasize nodes, edges, and relationship traversal,
- **column-family stores**, which organize data into rows with sparse column families and are designed for large-scale distributed workloads.

Each of these paradigms supports a different style of workload:

- fast direct lookup,
- flexible semi-structured records,
- relationship-heavy traversal,
- wide sparse data at scale.

This is why "best database" is usually the wrong question. The better question is which model matches the data and access pattern best.

## 8.18 Polyglot persistence

**Definition 8.5: Polyglot persistence**

*Polyglot persistence* means using different data storage technologies in one system because different components have different needs.

A recommendation engine, a billing system, and a social graph may each fit different storage choices.

For example:

- a billing or banking core may require a relational DBMS with strong transactional guarantees,
- a recommendation or session cache may fit a key-value store,
- a content or product catalog may fit a document store,
- a social or dependency network may fit a graph database.

Polyglot persistence is not about fashion. It is about fitting the storage model to the problem rather than forcing every problem into one model.

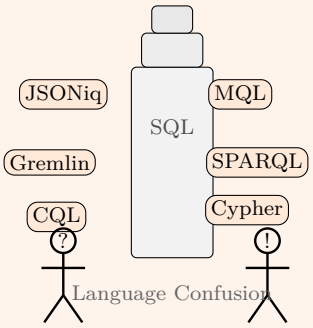
**Joke box**

Students, frustrated:
"Before NoSQL, we only had to learn SQL in the databases course.
Now, since NoSQL started, there are all kinds of languages and we have to learn them all!"

**Student 1:** "Let's start a counter movement!"

**Student 2:** "How shall we name it?"

All Students: **"NoNoSQL"**




## 8.19 Review questions and small exercises

1. What is the main difference between OLTP and OLAP workloads?
2. Why is ETL needed before data enters a warehouse?
3. What is a data mart?
4. Compare a star schema and a snowflake schema.
5. What is meant by master data?
6. Explain replication and sharding in simple terms.
7. Why can a data lake become a data swamp?
8. What is polyglot persistence, and why might an organization adopt it?

## 8.20 Further reading

Data warehousing and distributed data management form a bridge between classical database systems and modern analytics platforms. Kimball's work is central for dimensional modeling, while Inmon's writing shaped the warehouse perspective from an architectural angle. Broader systems books then help connect warehousing to replication, partitioning, and modern non-relational paradigms.

# Bibliography

[1] R. Kimball and M. Ross. *The Data Warehouse Toolkit*. Wiley.

[2] W. H. Inmon. *Building the Data Warehouse*. Wiley.

[3] M. T. Özsu and P. Valduriez. *Principles of Distributed Database Systems*. Springer.

## 8.21 Wrap-up

This chapter expanded the view from the single operational database to a larger data ecosystem. It connected warehousing, ETL, marts, distributed architectures, replication, sharding, and multiple data paradigms into one broader picture of modern data management. The next chapter continues that expansion into the world of big data, Hadoop, and MapReduce.

# Chapter 9

# Big Data, Hadoop, and MapReduce (not covered in the lecture)

**Chapter guidance**

This chapter introduces big data processing in distributed environments. It explains why traditional single system approaches may fail at scale, how Hadoop and HDFS organize storage, how the MapReduce model works, and why joins become difficult in distributed batch computation.

## 9.1 Big data motivation

Big data is not defined by one exact size threshold. It is better understood as data that becomes hard to store, process, or analyze with conventional single machine approaches. Volume, velocity, and variety are common dimensions, though the exact list varies.

The key point is relative difficulty, not fashion. A dataset that is trivial for a large cloud system may still be a big data problem for a laptop. Likewise, the same organization may use a classic relational DBMS for one workload and a distributed batch framework for another. This chapter therefore complements earlier chapters rather than replacing them. In many real systems, SQL and distributed processing live side by side.

A useful way to think about the problem is this: once the data no longer fits comfortably on one machine, or once one machine becomes too slow, too expensive, or too fragile for the workload, the architecture must change. At that point, storage, computation, communication, and fault tolerance become joint design problems rather than separate concerns.

Organizations also want more than simple storage. They want analytics at scale:

- **descriptive** analytics asks what happened,
- **predictive** analytics asks what is likely to happen,
- **prescriptive** analytics asks what actions should be taken.

These goals often require processing large logs, sensor streams, click traces, or document collections that are naturally distributed.

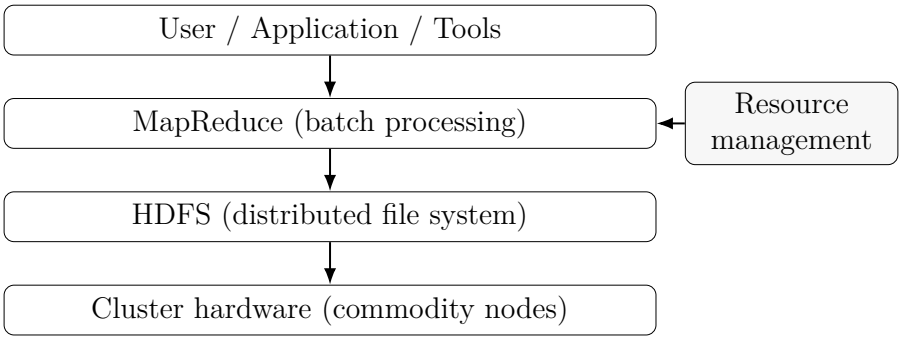



Figure 9.1: A compact view of the Hadoop stack. Applications use a batch processing layer on top of HDFS, which itself runs on a cluster of commodity machines.

## 9.2 Why traditional DBMS alone may not scale

Traditional DBMS systems are powerful, but very large scale workloads may require storage and computation across many machines. Large web logs, click streams, sensor data, or massive document collections can exceed the practical limits of one server.

This does not mean that relational DBMS technology becomes useless. It means that additional distributed frameworks may become necessary.

The reasons are several. A single server may run out of disk capacity, memory, network throughput, or compute power. Even when the raw storage fits, long-running scans and aggregations may take too long. Reliability also becomes a concern: if a single machine fails, the entire service or batch job may stop. Distributed systems address this by spreading both data and work across many commodity nodes.

A good example is log analysis. A company may collect terabytes of web logs each day. The logs are append-heavy, read in large sequential scans, and often processed by batch aggregation jobs. This is very different from a workload centered on short transactions and indexed point lookups. The workload shapes the architecture.

## 9.3 Hadoop ecosystem overview

Hadoop popularized an open ecosystem for distributed storage and batch processing. Two core ideas are HDFS for storage and MapReduce for computation. Figure 9.1 and figure 9.4 summarize the stack and the batch dataflow.

The stack in figure 9.1 separates concerns clearly. HDFS provides large-scale distributed storage. MapReduce provides a computation model that works over the distributed data. Resource management coordinates who gets to run where. This separation made Hadoop influential because it offered a practical, modular answer to large-scale batch analytics.

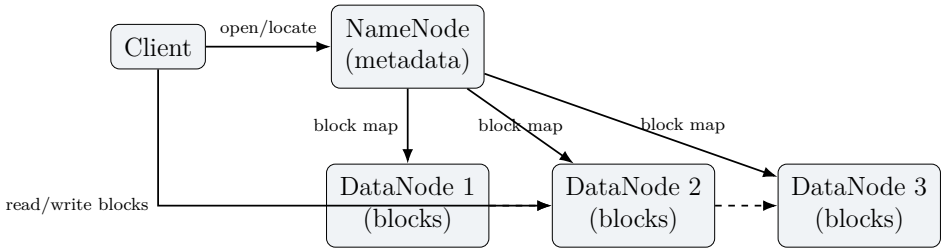



Figure 9.2: The basic HDFS architecture. The NameNode stores file metadata, while DataNodes store the actual file blocks.

## 9.4 HDFS

**Definition 9.1: HDFS**

The *Hadoop Distributed File System* stores large files across many machines by splitting them into blocks and replicating those blocks for fault tolerance.

A central metadata component tracks file structure, while worker nodes store the actual blocks. The details evolved over time, but the basic idea is data distributed over many nodes with redundancy.

A client normally asks the NameNode where the relevant blocks are located, then communicates directly with the DataNodes for reading and writing. This avoids turning the metadata server into the path for all data transfer. Figure 9.2 therefore separates metadata traffic from bulk data traffic.

Large blocks are a deliberate design choice. They reduce metadata overhead and support streaming access over huge files. HDFS is therefore especially well suited to batch-oriented workloads that scan large files sequentially. It is less well suited to many tiny files or many small random updates.

Replication is another core idea. If one DataNode fails, other replicas of the same block still exist. This gives HDFS robustness on clusters built from ordinary hardware, where failures are expected rather than exceptional.

## 9.5 HDFS scalability and practical constraints

HDFS solves the problem of large-file storage well, but it also has practical constraints.

- The metadata service must track the file namespace and block mappings.
- Large numbers of very small files can create significant metadata overhead.
- Replication improves fault tolerance but also increases storage cost and background traffic.
- Block size and replication factor affect both performance and reliability.

These design choices reflect the intended workload. HDFS is optimized for large files, streaming reads, and batch jobs. It is not primarily a general-purpose low-latency file system.

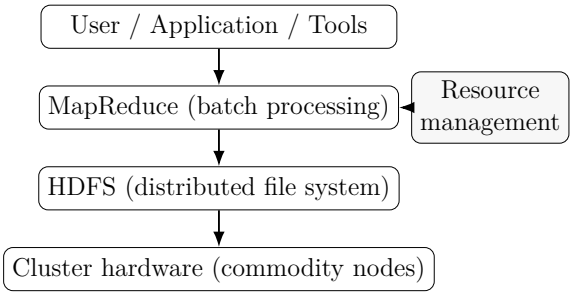



Figure 9.3: The Hadoop stack in one picture.

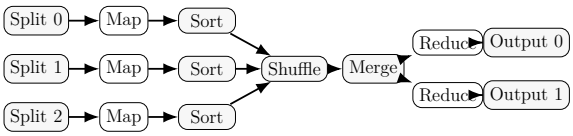



Figure 9.4: The MapReduce pipeline from input splits to reducer outputs.

## 9.6 MapReduce idea

**Definition 9.2: MapReduce**

*MapReduce* is a distributed programming model in which a map phase emits key value pairs and a reduce phase processes all values associated with the same key.

The framework handles data partitioning, movement, scheduling, and fault recovery.

The elegance of MapReduce lies in the narrow interface it offers to the programmer. One writes a map function and a reduce function, while the framework deals with the distributed systems work around them. This includes reading input splits, scheduling tasks, moving intermediate data, grouping values by key, and re-running failed tasks when needed.

## 9.7 MapReduce dataflow

The overall flow of a MapReduce job is easier to understand when broken into stages.

1. Input data is split into pieces that can be processed in parallel.
2. Each map task reads one split and emits intermediate key-value pairs.
3. Intermediate pairs are sorted and partitioned by key.
4. During the shuffle, all values belonging to the same key are brought together.
5. Each reducer processes one key group at a time and emits final output.

This is why figure 9.4 contains both a map side and a reduce side. The map side is embarrassingly parallel: each input split can be processed independently. The reduce side depends on regrouping all equal keys, which is more communication-heavy.

The model became famous through work by Jeffrey Dean and Sanjay Ghemawat at Google. Its importance was not only technical but also conceptual. It encouraged engineers to think in terms of data locality, repeated failures, and large-scale aggregation. Even when newer systems move beyond classic MapReduce, these ideas remain important.

## 9.8 Classic word count example

Word count is the traditional first example. The mapper reads text and emits pairs of the form `(word,1)`. The framework groups values by word. The reducer sums the counts for each word.

It is worth spelling out the stages carefully. Imagine the input is split across several machines. Each mapper works on its own piece and emits many small pairs such as `(database,1)` and `(query,1)`. During the shuffle, all pairs with key `database` are brought together, regardless of which machine produced them. Then one reducer receives the list of ones for `database` and adds them. The same happens for every other word. The example is simple, but it already shows the three central ideas: local processing, regrouping by key, and aggregate computation.

In pseudocode, one may write:

```
map(key, value):  for each word w in value emit(w,1)

reduce(key=w, values=[1,1,1,...]):  emit(w, sum(values))
```

A small instance makes the idea concrete. Suppose three splits contain:

Split 0: database query database
Split 1: query algebra
Split 2: database algebra query

The map outputs include:

$$(database, 1),\ (query, 1),\ (database, 1),\ (query, 1),\ (algebra, 1), \ldots$$

After shuffle and grouping:

$$database \mapsto [1, 1, 1], \qquad query \mapsto [1, 1, 1], \qquad algebra \mapsto [1, 1]$$

and the reducers emit:

$$(database, 3), \qquad (query, 3), \qquad (algebra, 2).$$

## 9.9 Other common MapReduce patterns

Word count is only the first step. Several other recurring batch patterns fit the same model well.

- **Distributed grep**: map emits matching lines, reduce may be trivial.
- **Reverse web graph**: map emits $(target, source)$, reduce collects inbound links.
- **Inverted index**: map emits $(word, docid)$, reduce aggregates the document list for each word.

For an inverted index, the logic is:

- **Map**: for each word occurrence in a document, emit $(word, docid)$.
- **Reduce**: for each word, output the unique set or list of associated document identifiers.

These examples show that MapReduce is especially natural for large scans, regrouping by key, and aggregation over very large inputs.

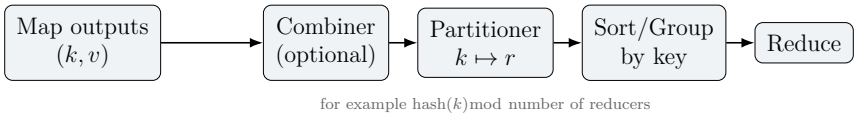



Figure 9.5: The main components between map output and reduce input: optional local combining, partitioning, sorting, grouping, and final reduction.

## 9.10 MapReduce runtime and job execution

A MapReduce job is more than user code. It is also a runtime process involving scheduling, execution, and recovery.

A high-level job lifecycle is:

1. submit the job,
2. assign map and reduce tasks,
3. execute tasks on workers,
4. collect outputs and commit the final result.

In the classic Hadoop v1 architecture, the central coordination role was played by the *JobTracker*, while worker nodes ran *TaskTrackers*. The JobTracker assigned tasks and monitored progress through heartbeats. The workers executed map and reduce tasks. This master-worker structure made the model easy to understand, though the master role also became a scaling and fault-tolerance concern.

## 9.11 The shuffle

Between map and reduce lies the *shuffle*. This is the stage where intermediate data is grouped by key and moved to the reducers that need it. In practice the shuffle often dominates cost. Network traffic and disk activity matter greatly here.

The reason the shuffle is expensive is now clear. Intermediate results from many mappers must often cross the network, be written and read, sorted, and merged. Even a simple logical task can therefore become costly if the intermediate data volume is large.

## 9.12 Partitioner, combiner, and sorting

A partitioner decides which reducer receives a key. A combiner can perform local pre aggregation when valid. Sorting is typically part of bringing equal keys together. These support efficiency, but they also impose constraints on algorithm design.

A good rule of thumb is that a combiner is useful when the reduce operation is associative and commutative, such as sum or count. In those cases, performing local partial aggregation before the shuffle can significantly reduce intermediate traffic. But a combiner is an optimization, not a semantic requirement. A correct algorithm must still produce the right result even if the system chooses not to run the combiner.

The partitioner is equally important. It determines load distribution across reducers. A poor partitioning strategy can create severe skew, where one reducer receives far more data than the others. Then the whole job may be limited by one overloaded reducer.

## 9.13 Fault tolerance

Distributed systems must assume that some machine, disk, or task will fail. MapReduce handles this by re running failed tasks and by relying on replicated data in the underlying file system. This model is robust for large batch jobs.

A few typical failure cases are:

- a task crashes because of buggy user code,
- a worker node disappears,
- a disk becomes unavailable,
- a network disruption interrupts communication.

The framework responds by re-executing failed tasks elsewhere when possible. This works well because the model is based on deterministic functions over stored input data. If the same map or reduce task is run again on the same input, it should produce the same result.

## 9.14 Determinism and output commit

Determinism matters because fault recovery depends on safe re-execution. A task should not produce different logical results merely because it is run again after failure.

To avoid duplicated final output, many distributed batch systems use an output commit protocol. Tasks write to temporary locations first. Only successful task completion leads to final committed output. This separation between temporary work and committed results is important for correctness.

## 9.15 Why joins are hard in MapReduce

Joins are natural in relational databases because DBMS systems are built around them. In MapReduce, joins are possible, but often expensive. Large datasets may need to be repartitioned and shuffled on join keys. Some special cases allow map side joins or bucket aligned joins, but the general lesson is that relational style operations become more costly once data is distributed widely.

The difficulty comes from data placement. In a relational DBMS, join algorithms are deeply integrated into the system, together with indexes, buffer management, and query optimization. In MapReduce, matching tuples may initially be on completely different machines. To join them, the system often has to bring them together explicitly.

## 9.16 Reduce-side joins

The most general strategy is the *reduce-side join*. Suppose we want to join

$$R(A, B) \bowtie S(B, C)$$

on attribute $B$. The map phase tags each tuple with its source relation and emits the join key as intermediate key:

$$\texttt{emit}(B, \langle R, A\rangle) \qquad \text{or} \qquad \texttt{emit}(B, \langle S, C\rangle).$$

After the shuffle, all tuples for the same $B$-value meet at the same reducer. The reducer then outputs all matching combinations.

This approach is general and easy to understand, but it can be expensive because both sides of the join may need to be shuffled.

## 9.17 Map-side joins

If one side of the join is small enough, a *map-side join* may be much cheaper. The small relation is loaded into memory, usually as a hash table, inside each mapper. The large relation is then streamed through the mappers, and matching tuples are joined locally without a large reduce-side shuffle of both inputs.

This is attractive when one side is a small dimension table and the other is a large fact-like dataset. But the strategy only works if the small side truly fits into mapper memory.

## 9.18 Partition and bucket joins

If both inputs are already partitioned compatibly by the join key, then each corresponding pair of partitions can be joined directly. This is the idea behind *partition joins*. A related idea is the *bucket join*, where tuples are assigned into hash buckets and corresponding buckets are joined.

These strategies can reduce shuffle cost substantially, but they require preprocessing or structural alignment of the data. The price is therefore paid earlier in the pipeline.

## 9.19 N-way map-side joins

A particularly useful special case arises in star-schema style settings. One large fact table is joined with several small dimension tables. If all dimensions are small enough, each mapper can load the dimensions into memory and then stream over the fact table, producing joined results locally. This avoids heavy multi-way reduce-side shuffling.

The limitation is obvious but important: all dimension tables must be small enough, and the memory requirements must remain practical at mapper scale.

## 9.20 Join strategy in perspective

The main join strategies can be summarized as follows:

| Strategy | When useful | Main cost or limit |
|---|---|---|
| Reduce-side join | general case | large shuffle and sort |
| Map-side join | one side small | broadcast and memory |
| Partition join | both sides aligned | preprocessing requirements |
| Bucket join | bucketed inputs | structural constraints |
| N-way map-side join | fact + small dimensions | memory and distribution |

This table shows why joins are one of the most revealing operations in distributed data processing. They expose the tension between generality and communication cost.

## 9.21 MapReduce in perspective

MapReduce was historically very important, but modern systems such as Spark often offer more flexible in memory processing and richer APIs. Even so, the key ideas remain educationally valuable because they explain the basic problems of distributed data processing.

MapReduce is particularly strong for:

- batch ETL,
- large scans,
- aggregation jobs,
- workloads in which latency is not the main concern.

It is less attractive for:

- low-latency interactive analytics,
- iterative machine learning with many repeated passes over data,
- complex pipelines that benefit from richer in-memory optimization.

Later systems often improve on these limitations, but they still inherit many of the same distributed systems concerns.

## 9.22 Discussion

The real lesson of this chapter is not to worship one framework. It is to understand why scale changes the problem. Once data lives across many machines, storage, communication, scheduling, and fault recovery become central design concerns.

This also connects back to earlier parts of the book. Relational algebra and SQL explained what a query means. Query optimization explained how different equivalent

execution strategies can differ in cost. MapReduce shows what happens when execution itself must be distributed across many machines with failures and communication overhead. In that sense, the topic extends the same database logic into a larger systems setting.

## 9.23 Review questions and small exercises

1. What makes a data problem a big data problem?
2. What is the role of HDFS in Hadoop?
3. Explain the map and reduce phases in simple words.
4. Why is the shuffle stage often expensive?
5. What does a combiner do, and when is it useful?
6. Why are joins harder in MapReduce than in a relational DBMS?
7. Why is fault tolerance central in distributed batch systems?
8. Why is MapReduce still worth studying even if newer tools exist?

## 9.24 Further reading

The original MapReduce and Bigtable papers remain excellent reading because they explain the design problems very clearly. For broader Hadoop-era context, white papers and systems texts help show how distributed storage and computation evolved into more flexible platforms such as Spark and cloud-native data processing systems.

# Bibliography

[1] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Communications of the ACM*, 51(1), 2008.

[2] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google File System. *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, 2003.

[3] F. Chang et al. Bigtable: A Distributed Storage System for Structured Data. *ACM Transactions on Computer Systems*, 26(2), 2008.

## 9.25 Wrap-up

We have now completed the lecture order from basic database concepts to distributed big data processing. The remaining material at the end of the book collects symbols, abbreviations, and index information for easy reference.

# Symbol Table

| Symbol | Meaning |
|---|---|
| $\sigma_c(R)$ | Selection of tuples in relation $R$ that satisfy condition $c$. |
| $\pi_{A_1,\ldots,A_k}(R)$ | Projection of relation $R$ to the listed attributes. |
| $\rho(R)$ | Renaming operator in relational algebra. |
| $R \cup S$ | Union of two union compatible relations. |
| $R \cap S$ | Intersection of two union compatible relations. |
| $R \setminus S$ | Set difference. |
| $R \times S$ | Cartesian product. |
| $R \bowtie S$ | Join of two relations. |
| $X \to Y$ | Functional dependency from attribute set $X$ to attribute set $Y$. |
| $X^+$ | Closure of attribute set $X$ under a set of dependencies. |
| $X \twoheadrightarrow Y$ | Multivalued dependency. |

# Abbreviation Table

| Abbreviation | Meaning |
|---|---|
| ACID | Atomicity, Consistency, Isolation, Durability. |
| BCNF | Boyce Codd Normal Form. |
| DBMS | Database Management System. |
| DCL | Data Control Language. |
| DDL | Data Definition Language. |
| DML | Data Manipulation Language. |
| DW | Data Warehouse. |
| EER | Extended Entity Relationship. |
| ER | Entity Relationship. |
| ETL | Extract, Transform, Load. |
| FD | Functional Dependency. |
| HDFS | Hadoop Distributed File System. |
| MVD | Multivalued Dependency. |
| OLAP | Online Analytical Processing. |
| OLTP | Online Transaction Processing. |
| RA | Relational Algebra. |
| SQL | Structured Query Language. |
| 3NF | Third Normal Form. |
| 4NF | Fourth Normal Form. |

# How this manuscript can be published online

A WordPress version of this book can mirror the lecture order exactly. A useful pattern is to create one page per chapter. Each chapter page can contain a short introduction, the chapter text as HTML, and a small resource panel with links to the slide PDF, the slide LaTeX source, the chapter PDF, and the Git repository for version tracking.

A landing page can include the current release date, a complete PDF of the book, citation information, and version links to earlier releases. This is close in spirit to modern online technical books that publish chapter drafts openly and improve them through feedback.

Recommended Git structure:

- one repository for the full book source
- one repository for slide sources, or a clear subdirectory structure inside a single repository
- release tags for dated public versions
- issue tracking for typos, improvements, and content requests

This chapter is only a brief note for later publication work. The main task of the present step was to create a self contained full text LaTeX manuscript from the lecture material.

# Index