Data Management

Mario Bergés

2025-12-02

Table of contents

Preface

This book (or website) contains a collection of lecture notes for the Fall 2025 edition of 12-741: Data Management at Carnegie Mellon University.

Notably, the lecture notes are —to a great extent— generated through GenAI tools (mostly Claude Code, with Sonnet 4.5) after careful prompting and many rounds of revision. This effort represents an experiment and, as any experiment, should be treated with humility and an open mind.

My hypothesis for the experiment is that it should be possible for me to carefully extract the relevant knowledge that is stored in modern LLMs (through a lossy compression process) in order to generate useful lecture notes that go well beyond what I would be able to write myself in the same time, all while providing additional utility to the students. This additional utility would come in two flavors: (1) the lecture notes can now contain interactive elements based on code snippets that can facilitate building intuition and understanding difficult concepts; (2) the students are encouraged to be consciously and actively suspicious of the material that they are reading, given that it could contain errors.

This hypothesis might be proven correct, or incorrect. I don't know what the answer is but I am curious about it. Obviously, with a small sample size of the students in this class and only self-reported measures of success, it will be hard to assess the hypothesis in general. But it's worth trying.

To incentivize students to be more critical readers of the material contained in this book, I am awarding each student 1 bonus point (worth 1% of your final grade) for each errors found in the content (submitted to me via e-mail) up to a total of 5 points per student.

If you're wondering about the website design and layout of this book: I'm using Quarto. To learn more about Quarto books visit https://quarto.org/docs/books.

Part I

Logistics

1 Syllabus

12-741 Section A2 - Fall 2025

2 Course Information

- **Meeting Times:** TR 5:00PM - 6:20 PM
- **Location:** Baker Hall (BH) A53
- **Instructor:** Mario Bergés
- **Office:** PH 123G
- **Phone:** x8-4572
- **Office Hours:** Wednesdays 2:00 PM - 3:00 PM and by appointment
- **Office Hours Location:** PH 123G
- **Teaching Assistants:** TBD
- **Office Hours:** TBD
- **Office Hours Location:** TBD

2.1 Textbooks (Optional)

1. H. Garcia-Molina, J.D. Ullman, J. Widom (2009), *Database systems: the complete book*, Prentice Hall
2. Barber, D. (2012). *Bayesian Reasoning and Machine Learning*. Cambridge UP. [URL]

3 Course Description

Because of the increasing capabilities to generate and collect data, civil and environmental engineers are exposed to a flood of information. This course will introduce graduate students to concepts and methods for organizing and analyzing data.

Topics covered will include: representation and processing of data collected by sensors and sensor networks; data (pre)-processing; feature extraction; relational and no-SQL databases.

The student will learn how to access a database, how to understand and design basic databases, and how to find and process data for inferring trends and draw conclusions from this data.

4 Grading

Component	Weight
Assignments	40%
Final Project	30%
- Written Report	15%
- Individual Oral Presentation	15%
Final Exam	30%

5 Policies

5.1 Assignments

A total of four assignments will be published. The topics covered in each assignment will closely follow the ones listed in the schedule of classes.

All assignments are to be solved individually. Discussions and conversations with other students regarding the problem sets are encouraged. However, the final solutions along with the reasoning behind them need to come from you and be clearly explained in the submitted documents.

Each assignment will be worth 10%.

5.1.1 Late Submissions

All assignments have due dates indicated on the syllabus. In general, submitting assignments on time lets the instructional team provide feedback in a more timely and efficient manner. Assignments build on each other, so timely submissions are crucial to your progress in the class. However, sometimes life happens. If you cannot submit an assignment on time, the default will be that you will be eligible for 90% of the grade the first 48 hours that the assignment is late. If you have to submit beyond 48 hours past the due date, please contact me as soon as possible so we can make arrangements. If you do not contact me before the 48 hours and you do not submit it, your assignment will receive a grade of 0%.

5.2 Group Projects

A portion of the grades for the course will be based on a group project, which consists of a written report and an oral presentation (individually). This course project is an opportunity to investigate deeper the methods covered in the course, develop new techniques, and/or apply them to real problems. The project will be completed by groups of 1-3 people. A 4-5 student group will be allowed only in special circumstances, e.g. for ambitious projects. Members are expected to contribute to the project with equal effort.

5.3 Final Exam

There will be a final examination. The date for the exam is still being decided. The exam will be closed book and electronic devices, but students can bring up to five pages of notes (following Chris Hendrickson's approach, and Matteo Pozzi's approach, who taught this class before me, I also think that preparing short notes is an opportunity to reflect upon major points of the course).

- **Final Exam Date:** Friday, December 9, at 8:30am ET
- **Final Project Report Due:** Friday, December 5, at 11:59pm ET

5.4 Software To Be Used

Students are free to use Mathworks Matlab, Octave or Python to solve the assignments. I personally prefer Python.

Matlab is available (see http://www.cmu.edu/computing/software/all/matlab/index.html) on all of the CMU computing clusters, and students are permitted to download Matlab to private machines so long as it is used on CMU's campus (www.cmu.edu/myandrew). University licensed toolboxes are also available, e.g. for statistics. They are downloaded as part of the Carnegie Mellon version of Matlab. Microsoft Access is part of the Office package. MySQL is a free software available at: http://www.mysql.com/.

5.5 Collaboration

Collaboration is expected within the limits of discussing concepts and problems. However, each student must produce his/her own solution to the problems.

Copying from another student's assignment is clearly plagiarism. Using information directly from websites, books, papers and other literary sources without appropriate attribution is also plagiarism. Assignments submitted for this class will be reviewed by the instructor and TA and may be scanned through web-based academic integrity software. Occurrences of cheating or plagiarism will be handled according to the university policy on Academic Integrity, https://www.cmu.edu/policies/documents/Academic%20Integrity.htm. Students are expected to have read this policy and conform to the highest standards of academic integrity. For incidents of academic misconduct, the University Academic Disciplinary Actions Policy, found at https://www.cmu.edu/student-affairs/theword/acad_standards/creative/disciplinary.html, will be followed.

5.6 Class Participation

Students are expected to be in class on time and participate in class discussions. Participation will be loosely monitored and used to calculate the participation grade. If you cannot make class, please inform your instructors and group members ahead of time. In class, students are expected to be courteous and respectful of the views and needs of other students and instructors.

5.7 Students with Disabilities

Students requesting classroom accommodation must first register with the Dean of Students Office. The Dean of Students Office will provide documentation to the student who must then provide this documentation to the Instructor when requesting accommodation.

5.8 Recording of Class Sessions

No recording or taping of any classroom activity is permitted without the express written consent of Prof. Bergés. Any student who needs to record or tape classroom activities because of a disability should contact the Carnegie Mellon Office of Equal Opportunity Services to request an appropriate accommodation.

5.9 Posting of Course Materials

All the material used in the course (syllabus, readings, problem sets, reports) is intended for use in the class only. No unauthorized posting, publication or redistribution is expected. Uploading course materials to Course Hero or other web sites is not an authorized use of the course material.

5.10 Take Care of Yourself

In general, do your best to maintain a healthy lifestyle this semester by eating well, exercising, getting enough sleep and taking some time to relax. This will help you achieve your goals and cope with stress.

All of us benefit from support during times of struggle. You are not alone. There are many helpful resources available on campus and an important part of the college experience is learning how to ask for help. Asking for support sooner rather than later is often helpful.

If you or anyone you know experiences any academic stress, difficult life events, or feelings like anxiety or depression, we strongly encourage you to seek support. Counseling and Psychological Services (CaPS) is here to help: call 412-268-2922 and visit their website at http://www.cmu.edu/counseling/. Consider reaching out to a friend, faculty or family member you trust for help getting connected to the support that can help.

6 Topics Covered (Lecture by Lecture)

6.1 Lecture 1: Introduction

- **Motivation for data management**: Understanding why data management practices are critical in civil and environmental engineering problems.
- **Real-world applications**: Identify specific examples of existing engineering systems that leverage data management solutions (e.g., smart infrastructure, environmental monitoring systems, transportation networks).
- **Course overview**: Familiarize yourself with the course syllabus, grading breakdown, policies, and expectations.
- **Core concepts**: Define what a database is in your own words and understand its role in organizing and accessing data.
- **Course logistics**: Understanding assignment structure, final project requirements, and collaboration policies.

Project Milestones: Understand the grading breakdown and expectations of the final project, including the written report and individual oral presentation components.

6.2 Lecture 2: Sensors and Time-series Data

- **Time series decomposition**: Be able to decompose a time series into seasonal, trend, and residual components to better understand patterns in data collected from sensors.
- **Noise characterization**: Understand different types of noise in sensor measurements and how noise propagates through both direct and virtual measurements.
- **Virtual sensors**: Be able to work with virtual sensors and understand how they derive measurements from other sensors or models.
- **Data quality**: Recognize the importance of understanding measurement uncertainty and error in sensor-based systems.

Project Milestones: Identify your teammates for the group project (groups of 1-3 people).

Assignment #1 Out

6.3 Lecture 3: Processing Time Series Data

- **Error propagation for linear Gaussian models**: Be able to calculate error propagation for linear Gaussian models.
- **Non-linear models**: Understand the reasons for why error propagation is different when models are not linear.
- **Outlier removal**: Be able to conduct outlier removal in time series using Chauvenet's criterion.
- **Linear regression derivation**: Be able to follow the derivation of linear regression with linear and non-linear basis functions.
- **Time series interpolation and extrapolation**: Use linear regression to interpolate or extrapolate from the time-series data that is available (estimating unobserved values in the time domain).

Project Milestones: Post on Piazza your specific group assignments and/or your need to join a group for the final project.

6.4 Lecture 4: Entity Relationship Diagrams and Set Theory

- **Entity Relationship Diagrams**: Understand the structure and visual components of these diagrams, and the motivation for using them.
  - **Components of an ER Diagram**: Be able to distinguish between attributes, entity sets and relationships.
  - **Weak Entity Sets**: Be able to identify weak entity sets and how to display them
  - **Roles and Subclasses in ER diagrams**: Understand the need and use of roles and sub-classes in ER diagrams.
- **Set Theory**: Recognize the value of set theory for describing and working with data records.
- **Definition of sets**: Be able to define sets in your own words, along with derivative terms such as sub-sets and super-sets.
- **Basic operations on sets**: Be able to apply (and recognize the notation for) the basic set operations of *intersection*, *union*, *set difference*, *complement*, *cartesian product*, *cardinality*
- **Venn Diagrams**: Recognize the value of visual diagramming approaches to sets such as Venn diagrams, and identify equivalences between set theory expressions and their corresponding Venn diagrams.
- **Properties of Set Operations**: Understand and be able to apply commutative, associative, distributive properties of set operations. Similarly, understand what are complimentary sets, and disjoint sets.

Project Milestones: Come up with 3 ideas for your final project, regardless of whther or not you are already assigned to a team.

6.5 Lecture 5: The Relational Model

Here are the topics covered, along with the learning objectives for each one:

- **Introduction to the Relational Model**: Understand the motivation behind the development of the relational model, and be able to provide a definition of it in your own words. Be able to describe the high-level connection between set theory and the relational model.
- **Relational Model Components**: Be able to identify and relate the following concepts: relations, attributes, tuples and types; as well as connect them to their corresponding analgous terms: tables, columns, rows and domain. Be able to describe the meaning of *key* attributes and understand the difference between a schema and an instance of a database.
- **Relational Database Creation Process**: Understand the steps of design (using a DDL), initial data load, and query execution.
- **Queries and Query Languages**: Be able to identify the function of a query and how natural language queries for relatioanl databases can be formalized with, e.g., relational algebra. Understand the link between relational algebra and SQL, as well as the fact that queries return relations.
- **Relational Algebra Operators**: Be able to apply and reason through the application of the following operators and their compositions: *select*, *project*, *cross product*, *natural join*, *theta join*, *union*, *difference*, *intersection*, *rename*.
- **From ER Diagrams to Relational Models**: Be able to transition from E/R diagrams to relational models of databases.

Project Milestones: Come up with 3 ideas for your final project, regardless of whther or not you are already assigned to a team.

6.6 Lecture 6: The Structured Query Language (SQL)

Here are the topics covered, along with the learning objectives for each one:

- **Introduction to the SQL**: Understand what SQL is and where it come from as well as its prevalence in industry and computing infrastructure today. Be able to distinguish between the Data Description Language and Data Manipulation Language groups of statements.
- **SQL Statements**: Be able to formulate queries using the following statements, functions and clauses, as well as their relational algebra counterparts (where applicable): `SELECT`, `SELECT ... WHERE`, `SELECT ... GROUP BY`, `SELECT ... JOIN`, `SELECT ... SELECT` (i.e., nested selects), `SUM()`, `MAX()`, `AVG()`, `CREATE TABLE`, `INSERT`, `DELETE`, `ALTER`, `UPDATE`.
- **Local DBMS servers**: Be able to install and issue queries to a local DBMS such as MySQL (and the MySQL Workbench as a client), or SQLite and the CLI interface to it.

Project Milestones: Submit a set of 5 slides for your group's final project proposal presentation:

1) Title of the project, and its members
2) Motivation for your project
3) Specific database problem you are aiming to solve and expected goals
4) Proposed solution (and how it draws from concepts learned in this course)
5) Remaining questions for your team about how to complete the project

6.7 Lecture 7: Database Design

Here are the topics covered, along with the learning objectives for each one:

- **Principles of database design**: Be able to explain how relational databases are designed and connnect the theory behind it to earlier concepts such as Entity Relationship Diagrams and Relational Algebra.
- **Properties and normal forms**: Be able to explain how the properties of the data lead to certain dependencies that allow for it to be structured. Understand how different structures lead to different design choices named "normal forms". Be able to decompose a relation up to its Boyce-Codd Normal Form (BCNF), when possible.
- **Functional dependencies**: Be able to derive functional dependencies given prioperties of the data, and draw these dependencies in a diagram.
- **Keys and closures**: Understand the concept of closures and keys of relations.

Project Milestones: Meet with your team to agree on a set of milestones and deliverables that are needed to complete the final project. This will be required as part of Assignment 4.

6.8 Lecture 8: Data Representation and Compression

Here are the topics covered, along with the learning objectives for each one:

- **Data Representation**: Be able to explain why different choices of how to represent (or compress) data/information offer tradeoffs that we should be aware of.
- **Filesystems and ASCII encoding**: Be able to predict the size of a simple ASCII text file based on its content.
- **Types of data compression**: Be able to describe the differences between lossless and lossy compression schemes.
- **Frequency Domain Representation of Data**: Understand the benefits of frequency domain representations for certain types of data.
- **Huffman coding**: Be able to apply Huffman coding to an arbitray text string.
- **JPEG and other application**: Be aware of the compression algorithms used for JPEG and in other domains. Are LLMs a compression algorithm?

Project Milestones: Schedule a meeting with the instructor for any clarifications needed before you go on Thanksgiving break.

6.9 Lecture 9: Database Design Practicum

Here are the topics covered, along with the learning objectives for each one:

- **ER Diagrams in Practice**: Be able to take a description of a dataset and convert it into an E/R model of it.
- **The Relational Model**: Be able to translate the E/R diagram to a relational database schema.
- **Implementing a Relational Database**: Be able to write an SQL script to implement the schema in SQLite.
- **Data Ingestion**: Be able to populate an existing databse with data contained in CSV files using SQL statements.
- **CRUD Using SQL**: Be able to create, read, update and delete (CRUD) data in the SQLite database using SQL.
- **CRUD using a Web Framework**: Understand that SQL statements (and the database schema under it) can be abstracted into higher-level structures such as a RESTful API, or a web application.
- **CRUD using Natural Language**: Understand that higher levels of abstraction are possible, particularly through the use of LLMs and their ability to parse natural language.

Project Milestones: Define the structure of your final report, and set milestones for completing each section.

Part II

Lecture Notes for Modules

7 Module 1: Introduction

7.1 Learning Objectives

By the end of this module, students will be able to:

- Understand why data management practices are critical in civil and environmental engineering problems
- Identify specific examples of existing engineering systems that leverage data management solutions (e.g., smart infrastructure, environmental monitoring systems, transportation networks)
- Familiarize yourself with the course syllabus, grading breakdown, policies, and expectations
- Define what a database is in your own words and understand its role in organizing and accessing data
- Understanding assignment structure, final project requirements, and collaboration policies

7.2 Topics Covered

- Motivation for data management
- Real-world applications
- Course overview
- Core concepts
- Course logistics

7.3 Project Milestones

Understand the grading breakdown and expectations of the final project, including the written report and individual oral presentation components.

7.4 Source Material

7.4.1 Why Data Management Matters in Civil and Environmental Engineering

Civil and environmental engineers increasingly face challenges related to the volume, variety, and complexity of data generated in modern engineering projects. From sensor networks monitoring structural health of bridges, to environmental monitoring systems tracking air and water quality, to transportation systems collecting real-time traffic data, the ability to effectively manage and query large datasets has become essential.

Without proper data management practices, engineers face several challenges:

- **Data loss and corruption**: Project data stored in ad-hoc formats (spreadsheets, text files, personal computers) is vulnerable to loss and inconsistency
- **Difficulty in data sharing**: Multiple team members working on the same project need coordinated access to current data
- **Inefficient data retrieval**: Finding specific information within large datasets becomes time-consuming without structured organization
- **Data integrity issues**: Ensuring data remains accurate and consistent across multiple uses and users
- **Scalability limitations**: As projects grow, simple file-based approaches become unmanageable

Databases and database management systems provide systematic solutions to these challenges, enabling engineers to store, retrieve, and analyze data efficiently and reliably.

7.4.2 What is a Database?

A **database** is a collection of persistent and structured data with a programming interface and transaction management. Let's break down this definition:

7.4.2.1 Persistent Data

Data is stored in a way that remains available after a computer session is terminated. Unlike data held in RAM (random access memory), which disappears when a program closes or a computer shuts down, database data persists on storage media such as hard drives or solid-state drives.

7.4.2.2 Structured Data

Data is stored in a format that is easily separable into logical parts. This is fundamentally different from unstructured formats like word processing documents or arbitrary text files. Structured data has a defined organization—typically rows and columns in relational databases—that makes it possible to efficiently query specific pieces of information.

7.4.2.3 Database Management System (DBMS)

The database concept is embodied in specialized software called a **Database Management System (DBMS)**. A DBMS provides:

- **Data organization**: Tools to define how data is structured and related
- **Data storage**: Efficient mechanisms for storing large amounts of data
- **Data retrieval**: Query languages that allow users to extract specific information
- **Data integrity**: Constraints and validation to ensure data accuracy
- **Concurrency control**: Management of simultaneous access by multiple users
- **Security**: Access control and authentication mechanisms

7.4.2.4 Programming Interface

A critical feature of databases is their **programming interface**, which allows users or application programs to access and modify data through a powerful query language. The most common query language is SQL (Structured Query Language), which provides a standardized way to:

- Retrieve specific data based on criteria
- Insert new data
- Update existing data
- Delete data
- Aggregate and summarize information
- Combine data from multiple sources

7.4.3 A Brief History of Databases

The body of knowledge and technology that constitutes modern database systems has developed since the 1960s.

The first Database Management Systems and the associated ideas were developed in the late 1960s. Some of the earliest database models were based on a hierarchical data model. Specifically, hierarchical databases organized data similar to the structure of a directory tree, with parent-child relationships forming a strict hierarchy. While hierarchical databases were useful

for certain applications, they had significant limitations. Data retrieval was constrained by the predefined hierarchical structure, making it difficult to represent many-to-many relationships or to query data in ways not anticipated in the original design.

The field of databases lacked a firm mathematical basis until **E.F. Codd** published a groundbreaking paper in 1970 introducing the relational model. Codd later formalized his ideas in "Codd's 12 rules" (1974), which defined what constitutes a truly relational database system.

The relational model offered several advantages:

- **Mathematical foundation**: Based on set theory and relational algebra
- **Flexibility**: Data could be queried in ways not predetermined by the database structure
- **Data independence**: The logical structure of data was separated from its physical storage
- **Declarative queries**: Users could specify what data they wanted, not how to retrieve it

The first commercial relational DBMS was **Oracle**, released in 1978, followed by **IBM DB2** (as a relational system). With the emergence of standards like SQL, relational databases are now employed within the majority of industrial and commercial applications.

Today, relational databases remain the dominant technology for structured data management, though they are increasingly complemented by NoSQL databases for specific use cases involving unstructured data, real-time applications, or massive scalability requirements. Understanding the fundamentals of relational databases provides a foundation for working with any data management system.

8 Module 2: Sensors and Time-series Data

8.1 Module Overview

8.1.1 Learning Objectives

By the end of this module, students will be able to:

- Be able to decompose a time series into seasonal, trend, and residual components to better understand patterns in data collected from sensors
- Understand different types of noise in sensor measurements and how noise propagates through both direct and virtual measurements
- Be able to work with virtual sensors and understand how they derive measurements from other sensors or models
- Recognize the importance of understanding measurement uncertainty and error in sensor-based systems

8.1.2 Topics Covered

- Time series decomposition
- Noise characterization
- Virtual sensors
- Data quality

8.1.3 Project Milestones

Identify your teammates for the group project (groups of 1-3 people).

8.2 Source Material

8.2.1 Introduction to Signals

In civil and environmental engineering, we often collect data from sensors that measure physical quantities over time. These measurements form what we call **signals**.

8.2.1.1 Continuous-Time Signals

A continuous-time signal is a function that maps time to a measured value. We denote such a signal as:

$$x(t), \quad t \in \mathbb{R}$$

where t represents continuous time and $x(t)$ represents the value of the signal at time t. For example, the temperature measured by a sensor at any instant in time could be represented as $T(t)$.

8.2.1.2 Discrete-Time Signals

In practice, sensors sample physical quantities at discrete time intervals. A discrete-time signal is represented as:

$$x[n], \quad n \in \mathbb{Z}$$

or equivalently:

$$x_n, \quad n = 0, 1, 2, ...$$

where n is the sample index and x_n represents the value of the signal at the n-th sample. The relationship between continuous and discrete time is given by:

$$x[n] = x(t_n) = x(n \cdot \Delta t)$$

where Δt is the sampling interval. For instance, if we sample temperature every 5 minutes, then $\Delta t = 5$ minutes and T_n represents the temperature at the n-th five-minute interval.

8.2.2 Signal Decomposition

Time series data from sensors often contain multiple underlying patterns. A common and useful approach is to decompose a signal into additive components that capture different temporal characteristics.

8.2.2.1 Additive Decomposition Model

We can express a continuous-time signal $x(t)$ as the sum of multiple components:

$$x(t) = x_{\text{seasonal}}(t) + x_{\text{trend}}(t) + x_{\text{residual}}(t)$$

where:

- **Seasonal component** $x_{\text{seasonal}}(t)$: Captures periodic patterns that repeat over fixed intervals (e.g., daily cycles, annual cycles). This can be modeled as a moving average over a long temporal window (e.g., annual timescale).
- **Trend component** $x_{\text{trend}}(t)$: Captures medium-term trends and variations (e.g., day-of-year patterns, seasonal weather changes). This can be modeled as a moving average over an intermediate temporal window.
- **Residual component** $x_{\text{residual}}(t)$: Captures short-term fluctuations and noise that remain after removing the seasonal and trend components. Instead of modling this term explicitly (i.e., deriving it from the measurements through some function like we did for the trend and seasonal components) we instead directly define it as the *residual* of the signal: whatever is left after removing seasonal and trend components.

Each component can be thought of as a moving average with a different window size:

$$x_{\text{component}}(t) = \frac{1}{W} \int_{t-W/2}^{t+W/2} x(\tau)\, d\tau$$

where W is the window length appropriate for that component.

This decomposition helps us understand and isolate different sources of variation in sensor data, making it easier to identify patterns, detect anomalies, and build predictive models.

8.2.3 Noise in Sensor Measurements

Sensor measurements are never perfect. They are affected by various sources of noise and uncertainty that we must characterize and account for.

8.2.3.1 The Additive Noise Model

We model the relationship between a measurement, the true physical quantity, and noise as:

$$y_i = q_i + n_i$$

where:

- y_i is the measured value at discrete time index i
- q_i is the true physical quantity we are trying to measure
- n_i is the measurement noise

The noise n_i represents all sources of error: sensor imprecision, environmental interference, quantization errors, etc.

8.2.3.2 Statistical Characterization of Noise

Since noise is random, we characterize it using probability distributions. In general, the noise follows some probability density function (PDF):

$$f_N(n)$$

which describes the likelihood of observing different noise values.

8.2.3.2.1 Gaussian (Normal) Noise

A common assumption is that measurement noise follows a Gaussian distribution:

$$f_N(n) = \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\frac{(n-\mu)^2}{2\sigma^2}\right)$$

where μ is the mean and σ^2 is the variance of the noise. For many sensors, we assume zero-mean noise ($\mu = 0$), meaning the noise doesn't systematically bias measurements in one direction.

8.2.3.2.2 Variance of the Mean

When we have multiple independent measurements $y_1, y_2, \ldots, y_N$ of the same quantity, we often compute their average (called the sample mean):

$$\bar{y} = \frac{1}{N} \sum_{i=1}^{N} y_i$$

If the noise is independent and identically distributed (i.i.d.) with variance σ^2, the variance of the mean is:

$$\text{Var}(\bar{y}) = \frac{\sigma^2}{N}$$

> Tip
>
> **Active Learning**
> You may want to prove that the variance of the sample mean is indeed $\frac{\sigma^2}{N}$. There is a sketch of the proof in the lecture slides.

This shows that averaging N measurements reduces the variance (and thus uncertainty) by a factor of N. The standard deviation decreases by a factor of $\sqrt{N}$.

Example: Suppose a temperature sensor has noise with standard deviation $\sigma = 0.5°C$. If we take a single measurement, the uncertainty is $0.5°C$. If we average 100 measurements, the uncertainty of the average becomes:

$$\text{SD}(\bar{y}) = \frac{0.5}{\sqrt{100}} = \frac{0.5}{10} = 0.05°C$$

This tenfold reduction in uncertainty illustrates the power of averaging to improve measurement quality.

8.2.4 Virtual Sensors and Uncertainty Propagation

In many engineering applications, we don't directly measure the quantity of interest. Instead, we derive it from other measurements using mathematical relationships.

8.2.4.1 What is a Virtual Sensor?

A **virtual sensor** is a measurement derived from one or more physical sensor measurements through a mathematical function. For example:

- Average temperature across multiple rooms: $T_{\text{avg}} = \frac{1}{3}(T_1 + T_2 + T_3)$
- Total energy consumption: $E_{\text{total}} = E_1 + E_2 + E_3$
- Heat flux from temperature difference: $q = k \cdot \Delta T$

These derived quantities are "virtual" measurements that carry uncertainty from the original measurements.

8.2.4.2 Uncertainty Propagation for Linear Combinations

Consider a virtual measurement that is a linear combination of M sensor measurements:

$$z = \sum_{j=1}^{M} a_j y_j$$

where a_j are constants and y_j are the individual measurements. Each measurement has the form $y_j = q_j + n_j$.

If we assume the noise terms n_j are independent and identically distributed with zero mean and variance σ^2, the variance of the virtual measurement is:

$$\text{Var}(z) = \sum_{j=1}^{M} a_j^2 \sigma^2 = \sigma^2 \sum_{j=1}^{M} a_j^2$$

Special case - Simple average: For an average of M measurements (where $a_j = 1/M$ for all j):

$$z = \frac{1}{M} \sum_{j=1}^{M} y_j$$

$$\text{Var}(z) = \sigma^2 \sum_{j=1}^{M} \left(\frac{1}{M}\right)^2 = \sigma^2 \cdot M \cdot \frac{1}{M^2} = \frac{\sigma^2}{M}$$

This confirms our earlier result: averaging reduces variance by a factor of M.

Understanding how uncertainty propagates through virtual measurements is critical for:

- Assessing the reliability of derived quantities
- Designing sensor networks with appropriate redundancy
- Making informed decisions based on uncertain data

8.3 Learn-by-Doing Activities

> **Note**
>
> The following activity needs to be completed in an interactive version of this document (i.e, the HTML output served by a webserver).

8.4 Interactive Exploration: Virtual Sensors and Uncertainty

This interactive widget allows you to explore how uncertainty in individual sensor measurements affects a virtual measurement (the average of multiple sensors).

The scenario: Three temperature sensors measure the temperature in three different rooms of a building. Measurements are taken every 5 minutes for one full day (288 samples). Each sensor measures the true room temperature plus some random noise. We create a virtual sensor that computes the average temperature across all three rooms.

Use the sliders below to adjust the true temperature (mean) and measurement noise (standard deviation) for each room, and observe how these changes affect both the individual measurements and the average temperature.

```
//| echo: false

// Import plotting library
Plot = import("https://cdn.jsdelivr.net/npm/@observablehq/plot@0.6.0/+esm")
d3 = import("https://cdn.jsdelivr.net/npm/d3@7/+esm")

// Number of samples (1 day at 5-minute intervals = 288 samples)
samplesPerDay = 288

// Time array (in hours)
timeHours = Array.from({length: samplesPerDay}, (_, i) => i * 5 / 60)

// Generate Gaussian random numbers using Box-Muller transform
function randn() {
  let u = 0, v = 0;
```

```
  while(u === 0) u = Math.random();
  while(v === 0) v = Math.random();
  return Math.sqrt(-2.0 * Math.log(u)) * Math.cos(2.0 * Math.PI * v);
}

// Function to generate temperature time series for a room
function generateRoomTemperature(mean, stddev, n) {
  return Array.from({length: n}, () => mean + stddev * randn());
}

// Sliders for Room 1
viewof room1Mean = Inputs.range([15, 25], {
  value: 20,
  step: 0.5,
  label: "Room 1 Mean Temperature (°C)"
})

viewof room1Std = Inputs.range([0, 2], {
  value: 0.5,
  step: 0.1,
  label: "Room 1 Noise Std Dev (°C)"
})

// Sliders for Room 2
viewof room2Mean = Inputs.range([15, 25], {
  value: 21,
  step: 0.5,
  label: "Room 2 Mean Temperature (°C)"
})

viewof room2Std = Inputs.range([0, 2], {
  value: 0.5,
  step: 0.1,
  label: "Room 2 Noise Std Dev (°C)"
})

// Sliders for Room 3
viewof room3Mean = Inputs.range([15, 25], {
  value: 19,
  step: 0.5,
  label: "Room 3 Mean Temperature (°C)"
})
```

```
viewof room3Std = Inputs.range([0, 2], {
  value: 0.5,
  step: 0.1,
  label: "Room 3 Noise Std Dev (°C)"
})

// Button to regenerate noise
viewof regenerate = Inputs.button("Regenerate Noise", {
  value: 0,
  reduce: (v) => v + 1
})

// Generate temperature data for each room
room1Temps = {
  regenerate; // This makes the cell reactive to button clicks
  return generateRoomTemperature(room1Mean, room1Std, samplesPerDay);
}

room2Temps = {
  regenerate;
  return generateRoomTemperature(room2Mean, room2Std, samplesPerDay);
}

room3Temps = {
  regenerate;
  return generateRoomTemperature(room3Mean, room3Std, samplesPerDay);
}

// Compute average (virtual sensor)
averageTemps = room1Temps.map((v1, i) => (v1 + room2Temps[i] + room3Temps[i]) / 3)

// Prepare data for plotting
plotData = timeHours.flatMap((t, i) => [
  {time: t, temperature: room1Temps[i], sensor: "Room 1"},
  {time: t, temperature: room2Temps[i], sensor: "Room 2"},
  {time: t, temperature: room3Temps[i], sensor: "Room 3"},
  {time: t, temperature: averageTemps[i], sensor: "Average (Virtual)"}
])

// Create the plot
Plot.plot({
  width: 900,
```

```
  height: 400,
  marginLeft: 60,
  marginRight: 100,
  x: {
    label: "Time (hours)",
    grid: true
  },
  y: {
    label: "Temperature (°C)",
    grid: true
  },
  color: {
    domain: ["Room 1", "Room 2", "Room 3", "Average (Virtual)"],
    range: ["steelblue", "orange", "green", "red"],
    legend: true
  },
  marks: [
    Plot.line(plotData, {
      x: "time",
      y: "temperature",
      stroke: "sensor",
      strokeWidth: d => d.sensor === "Average (Virtual)" ? 2.5 : 1.5,
      opacity: d => d.sensor === "Average (Virtual)" ? 1 : 0.7
    })
  ]
})

// Display statistics
md`
### Observed Statistics

**Individual Sensors:**
- Room 1: Mean = ${d3.mean(room1Temps).toFixed(2)}°C, Std Dev = ${d3.deviation(room1Temps).to
- Room 2: Mean = ${d3.mean(room2Temps).toFixed(2)}°C, Std Dev = ${d3.deviation(room2Temps).to
- Room 3: Mean = ${d3.mean(room3Temps).toFixed(2)}°C, Std Dev = ${d3.deviation(room3Temps).to

**Virtual Sensor (Average):**
- Mean = ${d3.mean(averageTemps).toFixed(2)}°C
- Std Dev = ${d3.deviation(averageTemps).toFixed(2)}°C

**Theoretical Prediction:**
- If all rooms had the same noise level  , the average should have std dev   /√3 = ${(Math.ma
```

```
- Actual average std dev: ${d3.deviation(averageTemps).toFixed(2)}°C
`
```

8.4.1 Questions to Consider

1. **Effect of noise**: Try setting all three rooms to the same mean temperature but different noise levels. How does the averaging reduce the noise in the virtual sensor?
2. **Variance reduction**: Set all three rooms to have the same noise standard deviation (e.g., 1.0°C). Compare the standard deviation of the individual rooms to the standard deviation of the average. Is it close to the theoretical prediction of $/\sqrt{3}$?
3. **Different means**: Set the three rooms to have different mean temperatures. What is the mean of the average temperature? How does it relate to the individual room temperatures?
4. **High vs. low noise**: Compare the case where all rooms have low noise (std dev = 0.1°C) versus high noise (std dev = 2.0°C). Click "Regenerate Noise" several times for each case. How much does the average temperature fluctuate between regenerations?

9 Module 3: Processing Time Series Data

9.1 Module Overview

9.1.1 Learning Objectives

By the end of this module, students will be able to:

- Be able to calculate error propagation for linear Gaussian models
- Understand the reasons for why error propagation is different when models are not linear
- Be able to conduct outlier removal in time series using Chauvenet's criterion
- Be able to follow the derivation of linear regression with linear and non-linear basis functions
- Use linear regression to interpolate or extrapolate from the time-series data that is available (estimating unobserved values in the time domain)

9.1.2 Topics Covered

- Error propagation for linear Gaussian models
- Non-linear models and error propagation
- Outlier removal using Chauvenet's criterion
- Linear regression derivation (linear and non-linear basis functions)
- Time series interpolation and extrapolation

9.1.3 Project Milestones

Post on Piazza your specific group assignments and/or your need to join a group for the final project.

9.2 Source Material

9.2.1 Recap: Uncertainty of Measurements Under Noise

In Module 2, we introduced the concept that real-world sensors produce measurements corrupted by noise. Let's revisit this fundamental idea and make it more concrete, as understanding measurement uncertainty is critical for everything we'll do in processing time series data.

9.2.1.1 The Measurement Model

Imagine we want to measure the temperature in a room using a digital thermometer. The true temperature (which we'll never know exactly) is some value q. When we take a measurement with our sensor, we get a reading y. If our sensor were perfect, we'd have $y = q$. But real sensors aren't perfect—they're affected by electronic noise, environmental factors, and inherent measurement limitations.

We can model this imperfection mathematically as:

$$y = q + n$$

where:

- q is the true physical quantity we're trying to measure
- y is our actual measurement (what the sensor reports)
- n is the measurement noise

For many practical sensors, this noise n can be well-approximated as **additive white Gaussian noise**. This means:

1. The noise is **additive** (it simply adds to the true value)
2. It's **white** (noise at one time point doesn't depend on noise at other time points)
3. It follows a **Gaussian (Normal) distribution** with mean zero and some variance σ^2

We write this as: $n \sim \mathcal{N}(0, \sigma^2)$

9.2.1.2 Why Gaussian Noise?

You might wonder: why do we assume Gaussian noise? While not all noise is perfectly Gaussian, there are good reasons this assumption is both practical and theoretically justified:

1. **Central Limit Theorem**: When noise comes from many small, independent sources (thermal noise, quantization errors, electromagnetic interference, etc.), their sum tends toward a Gaussian distribution
2. **Mathematical tractability**: Gaussian distributions have nice mathematical properties that make analysis possible
3. **Empirical observation**: Many real sensors do exhibit approximately Gaussian noise

9.2.1.3 Understanding Uncertainty

Now, what does this noise model tell us about uncertainty? There are two perspectives we might care about:

Forward uncertainty - $P(y|q)$: "Given that the true value is q, what measurement y might I observe?"

Since $y = q + n$ and $n \sim \mathcal{N}(0, \sigma^2)$, we have:

$$P(y|q) = \mathcal{N}(q, \sigma^2) \tag{9.1}$$

This says: if the true value is q, our measurements will be normally distributed around q with variance σ^2.

Inverse uncertainty - $P(q|y)$: "Given that I observed measurement y, what might the true value q be?"

This is actually what we care about most! There are many ways to arrive at this quantity. A simple one is to re-arrange our $y = q + n$ equation. Since we were able to derive that $P(y|q)$ is Normal in (Equation 9.1), we can easily see that $q = y - n$ should also follow a distribution $P(q|y)$ that is normal with mean y and variance σ^2, namely:

$$P(q|y) = \mathcal{N}(y, \sigma^2)$$

This result shows that both distributions have the **same form** and the **same variance**. Given a measurement y, our best estimate of the true value q is simply y itself, but we remain uncertain by an amount characterized by σ^2.

9.2.1.4 Estimating the Noise Level

How do we determine σ^2 in practice? If we can take multiple measurements $y_1, y_2, \ldots, y_N$ of the same physical quantity q (which remains constant), we can estimate the variance from the sample variance:

$$\hat{\sigma}^2 = \frac{1}{N-1}\sum_{i=1}^{N}(y_i - \bar{y})^2$$

where $\bar{y} = \frac{1}{N}\sum_{i=1}^{N} y_i$ is the sample mean.

This gives us a practical way to characterize our sensor's uncertainty: collect repeated measurements under stable conditions and compute the sample variance.

9.2.2 Error Propagation for Virtual Sensors

In Module 2, we introduced the concept of **virtual sensors**—measurements that are computed from other sensor measurements rather than directly measured. For example, if we measure the water levels at the top and bottom of a dam, we might compute the pressure difference as a virtual sensor. A key question arises: **if our physical sensors have uncertainty, how does that uncertainty propagate to our virtual sensors?**

9.2.2.1 A Motivating Example: Averaging Two Measurements

Let's start with the simplest possible case. Suppose we have two sensors measuring the same physical quantity q:

$$y_1 = q + n_1, \quad n_1 \sim \mathcal{N}(0, \sigma_1^2)$$
$$y_2 = q + n_2, \quad n_2 \sim \mathcal{N}(0, \sigma_2^2)$$

where n_1 and n_2 are independent (noise in one sensor doesn't affect the other).

To get a better estimate, we might average the two measurements to create a virtual sensor:

$$z = \frac{1}{2}y_1 + \frac{1}{2}y_2$$

Question: What is the noise affecting z? Is it better or worse than using just one sensor?

Let's work this out. Substituting our measurement model:

$$z = \frac{1}{2}(q + n_1) + \frac{1}{2}(q + n_2) = q + \frac{1}{2}n_1 + \frac{1}{2}n_2$$

So the virtual sensor has noise $n_z = \frac{1}{2}n_1 + \frac{1}{2}n_2$. What are the properties of this noise?

Mean: Since both n_1 and n_2 have zero mean:

$$E[n_z] = \frac{1}{2}E[n_1] + \frac{1}{2}E[n_2] = 0$$

Great! The virtual sensor is still unbiased (zero mean noise).

Variance: Using the independence of n_1 and n_2:

$$\text{Var}(n_z) = \text{Var}\left(\frac{1}{2}n_1 + \frac{1}{2}n_2\right) = \frac{1}{4}\text{Var}(n_1) + \frac{1}{4}\text{Var}(n_2) = \frac{1}{4}\sigma_1^2 + \frac{1}{4}\sigma_2^2$$

Special case: If both sensors have the same noise level ($\sigma_1 = \sigma_2 = \sigma$), then:

$$\text{Var}(n_z) = \frac{1}{4}\sigma^2 + \frac{1}{4}\sigma^2 = \frac{1}{2}\sigma^2$$

By averaging two independent measurements, we've **reduced the variance by half**. The standard deviation (the square root of variance) is reduced by a factor of $\sqrt{2} \approx 0.707$. This is why taking multiple measurements and averaging them improves accuracy.

Distribution: Since n_1 and n_2 are Gaussian and we're taking a linear combination, n_z is also Gaussian (this is a key property of Gaussian distributions). Therefore:

$$z = q + n_z, \quad n_z \sim \mathcal{N}\left(0, \frac{\sigma^2}{2}\right)$$

9.2.2.2 The General Linear Case

Now let's generalize. Suppose we have m sensors measuring different (or the same) physical quantities:

$$y_k = q_k + n_k, \quad n_k \sim \mathcal{N}(0, \sigma_k^2), \quad k = 1, 2, \ldots, m$$

And we create a virtual sensor as a **linear combination** of these measurements:

$$z_i = \sum_{k=1}^{m} \alpha_k y_{k,i} + c$$

where α_k are weighting coefficients and c is a constant offset.

Following the same logic as before:

$$z_i = \sum_{k=1}^{m} \alpha_k (q_k + n_k) + c = \left(\sum_{k=1}^{m} \alpha_k q_k + c \right) + \sum_{k=1}^{m} \alpha_k n_k$$

The virtual sensor has noise:

$$n_z = \sum_{k=1}^{m} \alpha_k n_k$$

Mean: Still zero (since each n_k has zero mean)

Variance: Assuming independent noise across sensors:

$$\text{Var}(n_z) = \sum_{k=1}^{m} \alpha_k^2 \sigma_k^2$$

Distribution: $n_z \sim \mathcal{N}\left(0, \sum_{k=1}^{m} \alpha_k^2 \sigma_k^2\right)$

Key insight: The variance of the virtual sensor depends on:

1. The variances of the input sensors (σ_k^2)
2. The **squares** of the coefficients (α_k^2)

This means:

- Multiplying a measurement by a factor α multiplies its variance by α^2
- Adding independent noisy measurements adds their variances
- The constant c doesn't affect uncertainty at all (it's just a bias)

9.2.2.3 Interactive Exploration: Virtual Sensors in Action

Let's make this concrete with an interactive example. Imagine we have two sensors measuring water levels at different locations in a reservoir system:

- Sensor 1 measures the upper level: $y_{1,k} = q_U + n_{1,k}$
- Sensor 2 measures the lower level: $y_{2,k} = q_L + n_{2,k}$

From these, we create two virtual sensors:

1. **Average level**: $z_A = \frac{1}{2} y_1 + \frac{1}{2} y_2$ (measures $q_A = \frac{1}{2} q_U + \frac{1}{2} q_L$)
2. **Gradient (flow rate proxy)**: $z_C = \frac{1}{h} y_1 - \frac{1}{h} y_2$ where h is the vertical distance between sensors

Use the interactive visualization below to explore how changing the true levels (q_U, q_L) and noise levels (σ_U, σ_L) affects both the raw measurements and the virtual sensors:

> **Note**
>
> The following activity needs to be completed in an interactive version of this document (i.e, the HTML output served by a webserver).

```
//| echo: false

viewof qU = Inputs.range([0, 1000], {value: 500, step: 10, label: "True Upper Level (qU)"})
viewof qL = Inputs.range([0, 1000], {value: 300, step: 10, label: "True Lower Level (qL)"})
viewof sigmaU = Inputs.range([0, 200], {value: 150, step: 10, label: "Upper Sensor Noise (U)
viewof sigmaL = Inputs.range([0, 200], {value: 150, step: 10, label: "Lower Sensor Noise (L)
viewof h = Inputs.range([0.05, 0.5], {value: 0.1, step: 0.05, label: "Vertical Distance (h)"
```

```
//| echo: false

// Generate random normal samples using Box-Muller transform
function randn_bm() {
  let u = 0, v = 0;
  while(u === 0) u = Math.random();
  while(v === 0) v = Math.random();
  return Math.sqrt(-2.0 * Math.log(u)) * Math.cos(2.0 * Math.PI * v);
}

// Generate time series data
data = {
  const N = 51;
  const k = Array.from({length: N}, (_, i) => i);

  // Generate noisy measurements
  const y1 = k.map(() => qU + sigmaU * randn_bm());
  const y2 = k.map(() => qL + sigmaL * randn_bm());

  // Calculate virtual sensors
  const zA = y1.map((y1_val, i) => 0.5 * y1_val + 0.5 * y2[i]);
  const zC = y1.map((y1_val, i) => (1/h) * y1_val - (1/h) * y2[i]);

  // True physical quantities
  const qA_true = 0.5 * qU + 0.5 * qL;
  const qC_true = (1/h) * qU - (1/h) * qL;
```

```
  return {
    k: k,
    qU_true: k.map(() => qU),
    qL_true: k.map(() => qL),
    y1: y1,
    y2: y2,
    zA: zA,
    zC: zC,
    qA_true: k.map(() => qA_true),
    qC_true: k.map(() => qC_true)
  };
}
```

```
//| echo: false

// Plot 1: Raw sensor measurements
Plot.plot({
  title: "Raw Sensor Measurements",
  width: 800,
  height: 300,
  x: {label: "Time index (k)"},
  y: {label: "Water level"},
  marks: [
    Plot.line(data.k.map((k, i) => ({k, value: data.qU_true[i]})),
              {x: "k", y: "value", stroke: "#1f77b4", strokeWidth: 2, strokeDasharray: "5,5"
    Plot.line(data.k.map((k, i) => ({k, value: data.qL_true[i]})),
              {x: "k", y: "value", stroke: "#d62728", strokeWidth: 2, strokeDasharray: "5,5"
    Plot.dot(data.k.map((k, i) => ({k, value: data.y1[i]})),
             {x: "k", y: "value", fill: "#17becf", fillOpacity: 0.6, r: 4}),
    Plot.dot(data.k.map((k, i) => ({k, value: data.y2[i]})),
             {x: "k", y: "value", fill: "#ff7f0e", fillOpacity: 0.6, r: 4})
  ]
})
```

```
//| echo: false

html`<div style="display: flex; justify-content: center; gap: 20px; margin: 10px 0; font-size
  <div style="display: flex; align-items: center; gap: 5px;">
    <svg width="30" height="3"><line x1="0" y1="1.5" x2="30" y2="1.5" stroke="#1f77b4" stroke
    <span>q<sub>U</sub> (true)</span>
  </div>
```

```
  <div style="display: flex; align-items: center; gap: 5px;">
    <svg width="30" height="10"><circle cx="15" cy="5" r="4" fill="#17becf" opacity="0.6"/></
    <span>y<sub>1</sub> (measured)</span>
  </div>
  <div style="display: flex; align-items: center; gap: 5px;">
    <svg width="30" height="3"><line x1="0" y1="1.5" x2="30" y2="1.5" stroke="#d62728" stroke
    <span>q<sub>L</sub> (true)</span>
  </div>
  <div style="display: flex; align-items: center; gap: 5px;">
    <svg width="30" height="10"><circle cx="15" cy="5" r="4" fill="#ff7f0e" opacity="0.6"/></
    <span>y<sub>2</sub> (measured)</span>
  </div>
</div>`
```

```
//| echo: false

// Plot 2: Average virtual sensor
Plot.plot({
  title: "Virtual Sensor: Average Level (zA = 0.5·y1 + 0.5·y2)",
  width: 800,
  height: 300,
  x: {label: "Time index (k)"},
  y: {label: "Average level"},
  marks: [
    Plot.line(data.k.map((k, i) => ({k, value: data.qA_true[i]})),
              {x: "k", y: "value", stroke: "#2ca02c", strokeWidth: 2, strokeDasharray: "5,5"}
    Plot.dot(data.k.map((k, i) => ({k, value: data.zA[i]})),
             {x: "k", y: "value", fill: "#98df8a", fillOpacity: 0.7, r: 4})
  ]
})
```

```
//| echo: false

html`<div style="display: flex; justify-content: center; gap: 20px; margin: 10px 0; font-size
  <div style="display: flex; align-items: center; gap: 5px;">
    <svg width="30" height="3"><line x1="0" y1="1.5" x2="30" y2="1.5" stroke="#2ca02c" stroke
    <span>q<sub>A</sub> (true)</span>
  </div>
  <div style="display: flex; align-items: center; gap: 5px;">
    <svg width="30" height="10"><circle cx="15" cy="5" r="4" fill="#98df8a" opacity="0.7"/></
    <span>z<sub>A</sub> (measured)</span>
  </div>
```

```
</div>`
```

```
//| echo: false

// Plot 3: Gradient virtual sensor
Plot.plot({
  title: `Virtual Sensor: Gradient (zC = (1/${h.toFixed(2)})·y1 - (1/${h.toFixed(2)})·y2)`,
  width: 800,
  height: 300,
  x: {label: "Time index (k)"},
  y: {label: "Gradient"},
  marks: [
    Plot.line(data.k.map((k, i) => ({k, value: data.qC_true[i]})),
              {x: "k", y: "value", stroke: "#9467bd", strokeWidth: 2, strokeDasharray: "5,5"
    Plot.dot(data.k.map((k, i) => ({k, value: data.zC[i]})),
             {x: "k", y: "value", fill: "#c5b0d5", fillOpacity: 0.7, r: 4})
  ]
})
```

```
//| echo: false

html`<div style="display: flex; justify-content: center; gap: 20px; margin: 10px 0; font-size
  <div style="display: flex; align-items: center; gap: 5px;">
    <svg width="30" height="3"><line x1="0" y1="1.5" x2="30" y2="1.5" stroke="#9467bd" stroke
    <span>q<sub>C</sub> (true)</span>
  </div>
  <div style="display: flex; align-items: center; gap: 5px;">
    <svg width="30" height="10"><circle cx="15" cy="5" r="4" fill="#c5b0d5" opacity="0.7"/></s
    <span>z<sub>C</sub> (measured)</span>
  </div>
</div>`
```

```
//| echo: false

// Display calculated uncertainties
html`<div style="background-color: #f0f0f0; padding: 15px; margin: 10px 0; border-radius: 5px
  <h4>Theoretical Uncertainties (Standard Deviations):</h4>
  <ul>
    <li><b>Upper sensor:</b>  <sub>U</sub> = ${sigmaU.toFixed(1)}</li>
    <li><b>Lower sensor:</b>  <sub>L</sub> = ${sigmaL.toFixed(1)}</li>
    <li><b>Average virtual sensor:</b>  <sub>A</sub> = √((0.5²·${sigmaU.toFixed(1)}²) + (0.5²
    <li><b>Gradient virtual sensor:</b>  <sub>C</sub> = √((1/${h.toFixed(2)})²·${sigmaU.toFix
```

```
  </ul>
  <p><i>Notice how:</i></p>
  <ul>
    <li>The <b>average</b> virtual sensor has <i>reduced</i> uncertainty compared to individu
    <li>The <b>gradient</b> virtual sensor has <i>amplified</i> uncertainty (coefficients = 
  </ul>
</div>`
```

Experiment with the controls:

1. Try increasing both σ_U and σ_L—watch how noise increases in all sensors
2. Set $q_U = q_L$—the gradient should be near zero, but notice the noise!
3. Decrease h (sensors closer together)—the gradient uncertainty explodes!
4. Compare the scatter in the average plot versus the individual sensor plots

This interactive example illustrates a crucial engineering trade-off: **virtual sensors that amplify measurements (large coefficients) also amplify noise**.

9.2.2.4 When Things Go Non-Linear: Breaking the Gaussian Assumption

Everything we've discussed so far relies on a critical assumption: **the virtual sensor is a linear function of the measurements**. But what happens when the relationship is non-linear?

Consider a practical example: measuring power dissipation in a resistor. If we measure current I flowing through a resistor with resistance R:

$$I_{measured} = I_{true} + n_I, \quad n_I \sim \mathcal{N}(0, \sigma_I^2)$$

The power dissipated is given by **Joule's law** (a non-linear operation with a squared term):

$$P = I^2 R$$

This raises two critical questions: 1. If I follows a Gaussian distribution, does P also follow a Gaussian distribution? 2. If not, what does the distribution of our computed power actually look like?

Let's explore both questions with a simulation:

```python
import numpy as np
import matplotlib.pyplot as plt

# Set random seed for reproducibility
np.random.seed(42)

# True values
I_true = 0.5    # amperes
R = 1        # ohms (known, no noise)
P_true = I_true**2 * R  # watts (true power dissipation)

# Noise level - relatively high to emphasize non-linear effects
sigma_I = 1   # amperes (200% relative noise)

# Generate many samples
N = 10000
I_measured = I_true + np.random.normal(0, sigma_I, N)

# Compute power using non-linear squared relationship
P_measured = I_measured**2 * R

# Compute errors
error_P = P_measured - P_true

# Create figure with two subplots
fig, axes = plt.subplots(1, 2, figsize=(14, 5))

# Left plot: Distribution of power measurements vs Gaussian
axes[0].hist(P_measured, bins=50, density=True, alpha=0.7, color='blue',
             edgecolor='black', label='Actual P distribution')

# Overlay what a Gaussian with the same mean and variance would look like
mean_P = np.mean(P_measured)
std_P = np.std(P_measured)
x_P = np.linspace(P_measured.min(), P_measured.max(), 100)
gaussian_P = (1/(std_P * np.sqrt(2*np.pi))) * np.exp(-0.5*((x_P - mean_P)/std_P)**2)
axes[0].plot(x_P, gaussian_P, 'r-', linewidth=2, label=f'Gaussian N({mean_P:.1f}, {std_P:.1f}

axes[0].axvline(P_true, color='green', linestyle='--', linewidth=2, label=f'True Power = {P_t
axes[0].set_xlabel('Measured Power (W)')
axes[0].set_ylabel('Probability Density')
axes[0].set_title('Power Distribution: I ~ Gaussian  > P ~ Gaussian\n(Non-linear: P = I²R)')
```

```
axes[0].legend()
axes[0].grid(True, alpha=0.3)

# Right plot: Distribution of errors with Gaussian overlay
axes[1].hist(error_P, bins=50, density=True, alpha=0.7, color='orange', edgecolor='black', la

# Overlay what a Gaussian with the same mean and variance would look like
mean_error = np.mean(error_P)
std_error = np.std(error_P)
x = np.linspace(error_P.min(), error_P.max(), 100)
gaussian_overlay = (1/(std_error * np.sqrt(2*np.pi))) * np.exp(-0.5*((x-mean_error)/std_erro
axes[1].plot(x, gaussian_overlay, 'r-', linewidth=2, label=f'Gaussian N({mean_error:.1f}, {s

axes[1].set_xlabel('Error in Power (W)')
axes[1].set_ylabel('Probability Density')
axes[1].set_title('Error Distribution vs Gaussian\n(Notice: NOT quite Gaussian!)')
axes[1].legend()
axes[1].grid(True, alpha=0.3)

plt.tight_layout()
plt.show()

# Print statistics
print(f"True Power: {P_true} W")
print(f"Mean Measured Power: {np.mean(P_measured):.2f} W")
print(f"Std Dev of Power: {np.std(P_measured):.2f} W")
print(f"\nError Statistics:")
print(f"  Mean error: {mean_error:.2f} W (should be ~0 for Gaussian)")
print(f"  Std dev: {std_error:.2f} W")
print(f"  Skewness: {np.mean(((error_P - mean_error)/std_error)**3):.3f} (0 for Gaussian)")
print(f"  Kurtosis: {np.mean(((error_P - mean_error)/std_error)**4):.3f} (3 for Gaussian)")
```

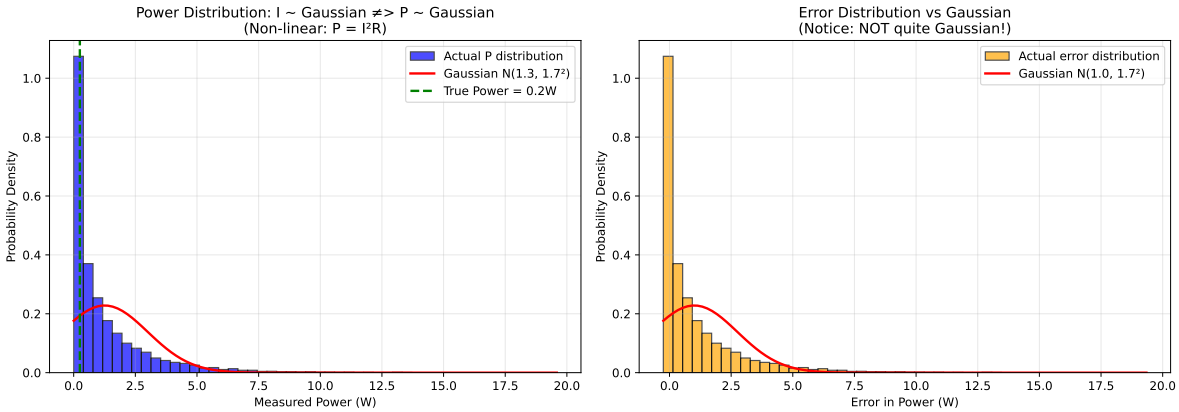

Figure 9.1: Distribution of error in a non-linear virtual sensor (power = I^2R)

```
True Power: 0.25 W
Mean Measured Power: 1.25 W
Std Dev of Power: 1.75 W

Error Statistics:
  Mean error: 1.00 W (should be ~0 for Gaussian)
  Std dev: 1.75 W
  Skewness: 2.729 (0 for Gaussian)
  Kurtosis: 13.906 (3 for Gaussian)
```

Key Observations:

1. **The power distribution itself is NOT Gaussian (left plot)**: Even though the current measurements I follow a Gaussian distribution, the computed power $P = I^2R$ does **not**. The actual distribution (blue histogram) is completely different to what a Gaussian with the same mean and variance would look like (red curve). This is a fundamental consequence: **non-linear transformations of Gaussian variables are generally not Gaussian**. The main driving force here is the fact that power measurements are never below zero, whereas the Gaussian distribution does allow for negative values.

2. **The error distribution is also NOT Gaussian (right plot)**: Similarly, the error in power estimates shows pronounced positive skew (right tail). The distribution is visibly asymmetric, and the skewness statistic deviates significantly from zero. The kurtosis also differs notably from 3 (the value for a true Gaussian).

3. **The mean error is positive (positive bias)**: Even though the input noise n_I has zero mean, the mean of the power error is positive! This is because of the **squared term**:

$$(I_{true} + n_I)^2 R = I_{true}^2 R + 2 I_{true} n_I R + n_I^2 R$$

The last term $n_I^2 R$ is always positive (since it's squared), creating a positive bias in the power estimates. Specifically, $E[n_I^2] = \sigma_I^2$, so the expected bias is $R\sigma_I^2$.

4. **The distribution is skewed right**: Large positive noise values in I get amplified more than negative noise values because of the squaring.

5. **Consequences for virtual sensors**:

   - We can't assume the virtual sensor output is Gaussian just because inputs are Gaussian
   - We can't use simple linear error propagation formulas
   - Confidence intervals based on Gaussian assumptions will be systematically wrong
   - For non-linear transformations, Monte Carlo simulation (like above) or analytical approaches like the Delta method become necessary
   - Simple averaging won't give unbiased estimates of the true power

6. **When is this a problem?**: If the noise is small relative to the signal (high signal-to-noise ratio), the non-linearity matters less. But when noise is significant or the function is highly non-linear (e.g., squares, exponentials, divisions by small numbers), the Gaussian approximation breaks down completely.

9.2.3 Data Cleansing: Dealing with Messy Real-World Data

So far, we've assumed our sensor data follows nice theoretical models. Reality is messier. Real sensor data often suffers from:

- **Incompleteness**: Missing measurements due to communication failures, power outages, or sensor downtime
- **Inconsistency**: Contradictory readings from redundant sensors or physically impossible values
- **Noise**: Not just Gaussian noise, but also spikes, drift, and other artifacts

Data cleansing (also called data cleaning or preprocessing) is the process of detecting and correcting (or removing) these issues to prepare data for analysis.

9.2.3.1 The Two Faces of Outliers

An **outlier** is a measurement that deviates significantly from the expected pattern or trend. But here's the critical question: *what does it mean?*

Sometimes outliers are the signal:

- A sudden spike in structural vibrations might indicate damage
- An unusual temperature reading might signal a fire

- Anomalous traffic patterns could indicate an accident

Sometimes outliers are noise:

- A communication error transmits garbage data
- A sensor briefly malfunctions
- Electromagnetic interference causes a spurious reading

The key is understanding your application: Are you looking for anomalies (keep the outliers!) or modeling normal behavior (remove them)?

9.2.3.2 Common Causes of Outliers

1. **Physical perturbations**: Real events that cause genuine deviations (e.g., a truck passing over a bridge causes a vibration spike)
2. **Sensor malfunctioning**: Hardware issues like degraded sensors, calibration drift, or component failure
3. **Communication errors**: Bit flips, packet loss, or transmission interference corrupting data in transit
4. **Environmental factors**: External conditions affecting sensor performance (temperature, humidity, electromagnetic fields)

9.2.3.3 Missing Values: Strategies for Handling Gaps

When data points are missing, you have several options:

Strategy	When to Use	Pros	Cons
Ignore the gap	Time-series analysis methods that handle irregular sampling	Simple, no assumptions	Reduces effective sample size
Manual filling	Critical data with small gaps where you have context	Most accurate if done carefully	Not scalable, labor-intensive
Default/flag value	When you need complete arrays but want to mark missing data	Preserves data structure	Must remember to handle flags in analysis
Mean imputation	Random, occasional gaps in stationary signals	Simple, maintains mean	Reduces variance artificially

Strategy	When to Use	Pros	Cons
Linear interpolation	Smoothly varying signals with small gaps	Reasonable for smooth trends	Poor for rapidly changing signals

Example: If you're measuring bridge deflection (a smoothly varying quantity) and miss 2 readings in a sequence of 1000, linear interpolation is reasonable. But if you're monitoring binary events (door open/closed), interpolation makes no physical sense.

9.2.3.4 Statistical Approach to Outlier Detection

A principled approach to outlier detection uses probabilistic reasoning:

1. **Model the normal behavior**: Estimate a probability distribution for typical measurements (e.g., Gaussian with mean μ and variance σ^2)
2. **Compute probabilities**: For each measurement, compute how likely it is under the model
3. **Flag low-probability points**: Measurements with very low probability are outliers

For Gaussian models, this translates to: "flag points that are too many standard deviations away from the mean."

9.2.3.5 Quick Review: Sample Statistics

Before we dive into a specific outlier detection method, let's recall how we estimate population parameters from samples.

If we have measurements $y_1, y_2, \ldots, y_N$ from our model $y_i = q + n_i$ where $n_i \sim \mathcal{N}(0, \sigma^2)$:

Sample mean (estimates q):

$$\bar{y} = \frac{1}{N} \sum_{i=1}^{N} y_i$$

Sample variance (estimates σ^2):

$$s^2 = \frac{1}{N-1} \sum_{i=1}^{N} (y_i - \bar{y})^2$$

Sample standard deviation (estimates σ):

$$s = \sqrt{s^2}$$

Note: We divide by $N-1$ (not N) in the variance formula to get an unbiased estimate (Bessel's correction).

These statistics will be the foundation of our outlier detection method.

9.2.3.6 Chauvenet's Criterion: A Practical Outlier Detection Method

One of the most straightforward outlier detection methods is **Chauvenet's criterion**, developed by the American astronomer William Chauvenet in the 1860s for cleaning astronomical observations.

9.2.3.6.1 The Basic Idea

The core principle is simple: **Reject measurements that are "too far" from the mean, where "too far" is defined in terms of standard deviations.**

But how far is "too far"? Chauvenet's insight was to use probability: if the chance of observing a point that extreme (given the Gaussian model) is less than 1/(2N), where N is the sample size, then it's likely an outlier rather than genuine variation.

9.2.3.6.2 Step-by-Step Procedure

Step 1: Compute the standardized distance

For each measurement y_i, compute how many standard deviations it is from the mean:

$$d_i = \frac{|y_i - \bar{y}|}{s}$$

where $\bar{y}$ is the sample mean and s is the sample standard deviation.

This is sometimes called the "z-score" or "standardized residual."

Step 2: Choose a threshold

Common threshold choices based on the number of standard deviations:

Threshold ($d_{threshold}$)	P(false rejection)	Typical Use
1.0	31.7%	Too lenient (would reject valid data often)
2.0	4.6%	Moderate (2-sigma rule)

Threshold ($d_{threshold}$)	P(false rejection)	Typical Use
3.0	0.3%	Conservative (3-sigma rule)
4.0	0.006%	Very conservative

The probability shown is the chance that a genuinely Gaussian point would be this far from the mean by random chance (i.e., the probability of incorrectly rejecting a good measurement).

For Chauvenet's original criterion, the threshold adapts to sample size N. A common rule: use $d_{threshold}$ such that $P(|Z| > d_{threshold}) < 1/(2N)$ where $Z \sim \mathcal{N}(0, 1)$.

Step 3: Reject or accept

- If $d_i \leq d_{threshold}$: **Accept** the measurement
- If $d_i > d_{threshold}$: **Reject** the measurement as an outlier

Step 4: Iterate

Here's the crucial part: outliers contaminate our estimates of $\bar{y}$ and s! If we have one extreme outlier, it pulls the mean toward it and inflates the standard deviation, making other outliers harder to detect.

Solution: Use an iterative approach:

1. Compute $\bar{y}$ and s from all data
2. Apply Chauvenet's criterion to identify outliers
3. Remove identified outliers
4. Recompute $\bar{y}$ and s from remaining data
5. Repeat steps 2-4 until no new outliers are found (convergence)

Typically, this converges in 2-4 iterations.

9.2.3.6.3 Example: Cleaning Temperature Data

Let's apply Chauvenet's criterion to a realistic example:

> **Note**
>
> In this example (and future ones) I am choosing to write out the simulation and experiments using Python. A good amount of the intuition about what we are trying to simulate in the example is encoded directly in the Python code (e.g., for this case, we are simulating 100 data points coming from a Normal distribution with mean 20 Celsius and standard deviation 0.5 degrees Celsius). As a result, you may need to read through the code in order to get a better understanding of the resulting plots.

```
import numpy as np
import matplotlib.pyplot as plt
from scipy import stats

# Generate realistic temperature data with outliers
np.random.seed(123)
N = 100
true_temp = 20.0  # Celsius
noise_std = 0.5

# Generate mostly good data
temperatures = true_temp + np.random.normal(0, noise_std, N)

# Add some outliers (sensor glitches)
outlier_indices = [10, 25, 63, 87]
temperatures[outlier_indices] = [35.0, -5.0, 40.0, 32.0]

def chauvenets_criterion(data, threshold=2.0, max_iterations=10):
    """
    Apply Chauvenet's criterion iteratively to remove outliers.
    Returns: cleaned data, indices of outliers, iteration history
    """
    data = np.array(data, dtype=float)
    valid_mask = np.ones(len(data), dtype=bool)
    history = []

    for iteration in range(max_iterations):
        # Compute statistics on currently valid data
        valid_data = data[valid_mask]
        mean = np.mean(valid_data)
        std = np.std(valid_data, ddof=1)

        # Compute standardized distances for ALL points
        d = np.abs(data - mean) / std

        # Find outliers (only among currently valid points)
        newly_rejected = valid_mask & (d > threshold)

        # Store history
        history.append({
            'iteration': iteration + 1,
            'mean': mean,
```

```
            'std': std,
            'n_valid': np.sum(valid_mask),
            'n_rejected_this_round': np.sum(newly_rejected)
        })

        # Check for convergence
        if np.sum(newly_rejected) == 0:
            break

        # Update valid mask
        valid_mask = valid_mask & ~newly_rejected

    outlier_indices = np.where(~valid_mask)[0]
    return data[valid_mask], outlier_indices, history

# Apply Chauvenet's criterion
cleaned_data, outlier_idx, history = chauvenets_criterion(temperatures, threshold=3.0)

# Print iteration history
print("Chauvenet's Criterion - Iteration History (threshold = 3.0 ):")
print("-" * 70)
for h in history:
    print(f"Iteration {h['iteration']}: mean={h['mean']:.3f}, std={h['std']:.3f}, "
          f"valid points={h['n_valid']}, rejected this round={h['n_rejected_this_round']}")
print("-" * 70)
print(f"Final result: {len(cleaned_data)} points retained, {len(outlier_idx)} outliers remove
print(f"Detected outlier indices: {outlier_idx}")
print(f"Actual outlier indices:   {outlier_indices}")

# Visualization
fig, axes = plt.subplots(1, 2, figsize=(14, 5))

# Plot 1: Original data
axes[0].scatter(range(N), temperatures, c='blue', alpha=0.6, label='All data')
axes[0].scatter(outlier_indices, temperatures[outlier_indices],
                c='red', s=100, marker='x', linewidths=3, label='Known outliers')
axes[0].axhline(true_temp, color='green', linestyle='--', linewidth=2, label='True temperatu
axes[0].set_xlabel('Measurement index')
axes[0].set_ylabel('Temperature (°C)')
axes[0].set_title('Original Data with Outliers')
axes[0].legend()
axes[0].grid(True, alpha=0.3)
```

```python
# Plot 2: After cleaning
valid_indices = np.arange(N)
valid_indices = np.delete(valid_indices, outlier_idx)
axes[1].scatter(valid_indices, cleaned_data, c='blue', alpha=0.6, label='Retained data')
axes[1].scatter(outlier_idx, temperatures[outlier_idx],
                c='red', s=100, marker='x', linewidths=3, label='Removed as outliers')
axes[1].axhline(true_temp, color='green', linestyle='--', linewidth=2, label='True temperatu
axes[1].set_xlabel('Measurement index')
axes[1].set_ylabel('Temperature (°C)')
axes[1].set_title('After Applying Chauvenet\'s Criterion')
axes[1].legend()
axes[1].grid(True, alpha=0.3)

plt.tight_layout()
plt.show()

# Compare statistics
print(f"\nStatistical comparison:")
print(f"Original data: mean={np.mean(temperatures):.3f}, std={np.std(temperatures, ddof=1):.3
print(f"Cleaned data:  mean={np.mean(cleaned_data):.3f}, std={np.std(cleaned_data, ddof=1):.3
print(f"True values:   mean={true_temp:.3f}, std={noise_std:.3f}")
```

```
Chauvenet's Criterion - Iteration History (threshold = 3.0 ):
----------------------------------------------------------------------
Iteration 1: mean=20.253, std=3.785, valid points=100, rejected this round=4
Iteration 2: mean=20.034, std=0.569, valid points=96, rejected this round=0
----------------------------------------------------------------------
Final result: 96 points retained, 4 outliers removed
Detected outlier indices: [10 25 63 87]
Actual outlier indices:   [10, 25, 63, 87]
```

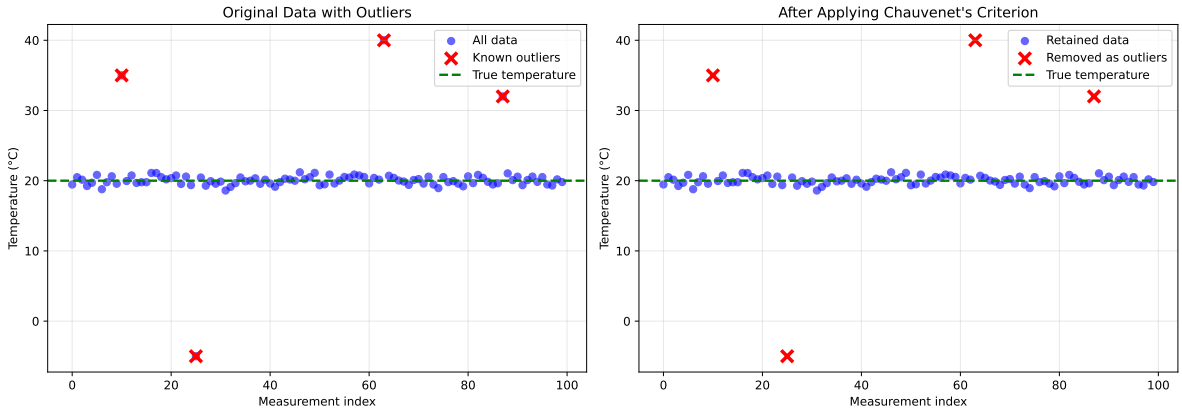

Figure 9.2: Applying Chauvenet's criterion to remove outliers from temperature data

```
Statistical comparison:
Original data: mean=20.253, std=3.785
Cleaned data:  mean=20.034, std=0.569
True values:   mean=20.000, std=0.500
```

Key Observations:

1. **Iteration matters**: The first iteration might not catch all outliers because extreme outliers inflate the standard deviation. After removing the worst outliers, the standard deviation decreases, making remaining outliers more obvious.
2. **Threshold selection**: Using $d_{threshold} = 3$ (3-sigma rule) is conservative and appropriate when you want to minimize false rejections. For noisier data or when outliers are more subtle, you might use a lower threshold like 2.5.
3. **Performance**: In this example, Chauvenet's criterion successfully identified all four artificially injected outliers, and the cleaned data has statistics much closer to the true values.
4. **Limitations**:
   - Assumes outliers are few ($<$ 10-20% of data). With many outliers, the method can fail.
   - Requires roughly Gaussian distribution of valid data
   - Can't distinguish between "bad outliers" (sensor errors) and "good outliers" (real extreme events)

9.2.3.6.4 When to Use Chauvenet's Criterion

Good for:

- Cleaning sensor data before modeling
- Quality control in data collection
- Preprocessing for regression or averaging
- Small to moderate numbers of outliers (<10-20%)

Not good for:

- Anomaly detection (when outliers are the goal)
- Highly skewed or non-Gaussian data
- Data with many outliers (>20%)
- Time series with trends (detrend first!)

9.2.4 Estimating Trends in Time Series Data

Up to now, we've focused on understanding individual measurements and their uncertainties. But when collecting time series data from sensors, we typically want to understand **patterns over time**—trends, cycles, and relationships.

9.2.4.1 From Individual Points to Patterns

The simplest approach to estimating a physical quantity q_k at time t_i would be to just use the measurement at that time: $\hat{q}_k(t_i) = y_{k,i}$. But this has serious limitations:

Problem 1: We can't predict beyond our measurements

What if we want to estimate q at a time when we didn't take a measurement? Or forecast future values? A single-point estimate gives us no predictive power.

Problem 2: We ignore structure

Most physical quantities don't jump randomly—they follow patterns:

- Bridge deflection varies smoothly with load
- Temperature follows daily and seasonal cycles
- Water levels change gradually (except during storms)

If we believe the quantity has structure (smoothness, trends, periodicities), we should use that information!

Problem 3: We don't leverage multiple measurements

If we have measurements at times $t_1, t_2, \ldots, t_N$, and we believe the quantity varies smoothly, then nearby measurements should inform our estimate at time t_i. Using only y_i wastes information from neighboring points.

9.2.4.2 The Core Idea: Model the Trend

Instead of treating each measurement independently, we'll **fit a model** that describes how the physical quantity varies over time:

$$q(t) = f(t; \theta)$$

where:

- f is a function we choose (linear, polynomial, periodic, etc.)
- θ are **parameters** we estimate from data

Then we'll find the parameters θ that best explain all our measurements $y_1, y_2, \ldots, y_N$.

This approach has powerful benefits:

- **Interpolation**: Estimate $q(t)$ at times between measurements
- **Extrapolation**: Predict $q(t)$ at future times (with caution!)
- **Noise reduction**: The model "smooths out" measurement noise
- **Compression**: Store the model parameters instead of all raw data
- **Insight**: The model reveals underlying patterns

The simplest and most widely used method for fitting such models is **linear regression**.

9.2.4.3 Linear Regression: Finding the Best-Fit Line

Linear regression is one of the most fundamental tools in data analysis. Let's build it from first principles, emphasizing the role of **residuals**—the key to understanding what regression is really doing.

9.2.4.3.1 The Setup: Simple Linear Regression

Suppose we have N measurements $(t_1, y_1), (t_2, y_2), \ldots, (t_N, y_N)$, and we believe the underlying physical quantity varies linearly with time:

$$q(t) = \beta_0 + \beta_1 t$$

where β_0 is the intercept and β_1 is the slope—these are the parameters we need to estimate.

Our measurements are noisy versions of this linear trend:

$$y_i = \beta_0 + \beta_1 t_i + n_i$$

Goal: Find the "best" values of β_0 and β_1 given our measurements.

9.2.4.3.2 Residuals: The Heart of Regression

For any choice of β_0 and β_1, we can compute the **predicted value** at time t_i:

$$\hat{y}_i = \beta_0 + \beta_1 t_i$$

The **residual** r_i is the difference between the actual measurement and our prediction:

$$r_i = y_i - \hat{y}_i = y_i - (\beta_0 + \beta_1 t_i)$$

Key intuition:

- If our model is perfect and noise-free, all residuals would be zero
- In reality, residuals capture both measurement noise and model mismatch
- **If the model structure is correct**, residuals estimate the noise
- **If the model structure is wrong**, residuals include both noise and systematic error

9.2.4.3.3 The Least Squares Criterion

How do we choose the "best" β_0 and β_1? We want to minimize the residuals, but some will be positive and some negative. A natural choice is to minimize the **sum of squared residuals** (also called the sum of squared errors, SSE):

$$\text{SSE}(\beta_0, \beta_1) = \sum_{i=1}^{N} r_i^2 = \sum_{i=1}^{N} (y_i - \beta_0 - \beta_1 t_i)^2$$

Why squares?

1. Squares are always positive (can't cancel out)
2. Squares penalize large errors more than small ones
3. Squaring leads to a smooth optimization problem with a unique solution
4. **Connects to maximum likelihood**: Under Gaussian noise, minimizing SSE is equivalent to maximum likelihood estimation

9.2.4.3.4 Deriving the Solution

To find the minimum, we take derivatives with respect to β_0 and β_1 and set them equal to zero:

$$\frac{\partial \text{SSE}}{\partial \beta_0} = -2 \sum_{i=1}^{N} (y_i - \beta_0 - \beta_1 t_i) = 0$$

$$\frac{\partial \text{SSE}}{\partial \beta_1} = -2 \sum_{i=1}^{N} t_i (y_i - \beta_0 - \beta_1 t_i) = 0$$

Simplifying the first equation:

$$\sum_{i=1}^{N} y_i = N\beta_0 + \beta_1 \sum_{i=1}^{N} t_i$$

$$\beta_0 = \bar{y} - \beta_1 \bar{t}$$

where $\bar{y} = \frac{1}{N} \sum_i y_i$ and $\bar{t} = \frac{1}{N} \sum_i t_i$.

Insight: The regression line always passes through the point $(\bar{t}, \bar{y})$—the center of the data cloud.

Substituting this into the second equation and simplifying (algebra omitted for brevity):

$$\beta_1 = \frac{\sum_{i=1}^{N}(t_i - \bar{t})(y_i - \bar{y})}{\sum_{i=1}^{N}(t_i - \bar{t})^2} = \frac{\text{Cov}(t, y)}{\text{Var}(t)}$$

These are the **least squares estimates** for the slope and intercept.

9.2.4.3.5 Example: Fitting a Trend

```
import numpy as np
import matplotlib.pyplot as plt

# Generate synthetic data with a linear trend plus noise
np.random.seed(42)
N = 30
t = np.linspace(0, 10, N)
beta_0_true = 5.0
beta_1_true = 2.0
noise_std = 3.0

y = beta_0_true + beta_1_true * t + np.random.normal(0, noise_std, N)

# Compute least squares estimates
t_bar = np.mean(t)
y_bar = np.mean(y)
beta_1 = np.sum((t - t_bar) * (y - y_bar)) / np.sum((t - t_bar)**2)
beta_0 = y_bar - beta_1 * t_bar

# Predictions and residuals
y_pred = beta_0 + beta_1 * t
residuals = y - y_pred

# Plot
fig, axes = plt.subplots(1, 2, figsize=(14, 5))

# Left plot: Data and fit
axes[0].scatter(t, y, alpha=0.6, s=50, label='Measurements', color='blue')
axes[0].plot(t, y_pred, 'r-', linewidth=2, label=f'Fit: y = {beta_0:.2f} + {beta_1:.2f}t')
axes[0].plot(t, beta_0_true + beta_1_true * t, 'g--', linewidth=2, alpha=0.7,
             label=f'True: y = {beta_0_true:.2f} + {beta_1_true:.2f}t')

# Draw residual lines
for i in range(N):
```

```
    axes[0].plot([t[i], t[i]], [y[i], y_pred[i]], 'k-', alpha=0.3, linewidth=1)

axes[0].scatter([t_bar], [y_bar], s=200, marker='X', color='red',
                edgecolors='black', linewidths=2, label='Centroid', zorder=5)
axes[0].set_xlabel('Time')
axes[0].set_ylabel('Measurement')
axes[0].set_title('Linear Regression Fit')
axes[0].legend()
axes[0].grid(True, alpha=0.3)

# Right plot: Residuals
axes[1].scatter(t, residuals, alpha=0.6, s=50, color='orange')
axes[1].axhline(0, color='red', linestyle='--', linewidth=2, label='Zero residual')
axes[1].set_xlabel('Time')
axes[1].set_ylabel('Residual (y - ŷ)')
axes[1].set_title(f'Residuals (SSE = {np.sum(residuals**2):.2f})')
axes[1].legend()
axes[1].grid(True, alpha=0.3)

plt.tight_layout()
plt.show()

print(f"True parameters:    = {beta_0_true:.3f},    = {beta_1_true:.3f}")
print(f"Estimated parameters:    = {beta_0:.3f},    = {beta_1:.3f}")
print(f"Residual standard deviation: {np.std(residuals, ddof=2):.3f}")
print(f"(Compare to true noise std: {noise_std:.3f})")
```

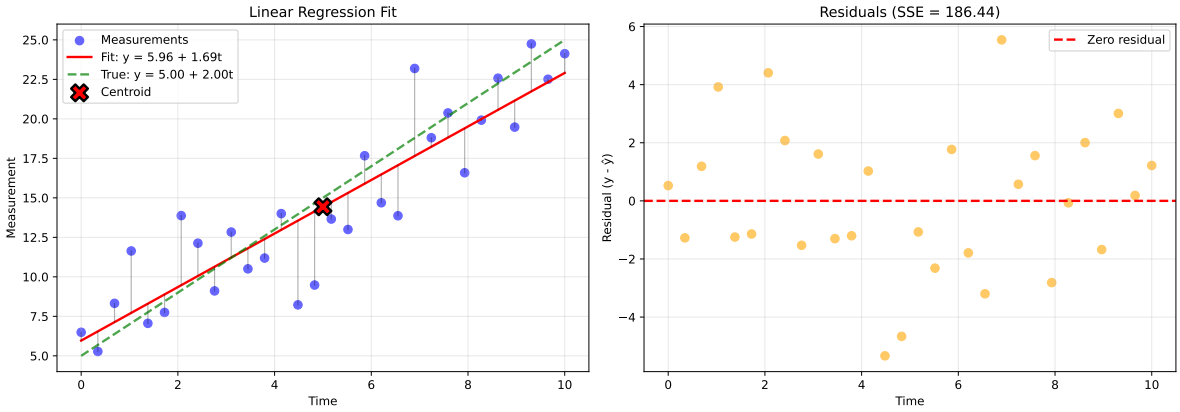

Figure 9.3: Linear regression example showing data, fit line, and residuals

```
True parameters:   = 5.000,   = 2.000
Estimated parameters:   = 5.964,   = 1.694
Residual standard deviation: 2.580
(Compare to true noise std: 3.000)
```

Key observations:

1. The fit line passes through the centroid (marked with X)
2. Residuals scatter randomly around zero (if the model is correct)
3. The residual standard deviation estimates the noise level
4. Vertical lines show each residual—these are what we minimize in sum-of-squares

9.2.4.3.6 Using Regression for Outlier Detection

Remember Chauvenet's criterion? We can combine it with regression for cleaning data with trends:

Iterative regression-based outlier removal:

1. Fit a regression model to all data
2. Compute residuals: $r_i = y_i - \hat{y}_i$
3. Compute mean and std of residuals: $\bar{r}$ and s_r
4. Remove points where $|r_i - \bar{r}|/s_r > d_{threshold}$
5. Refit the model and repeat until convergence

This is more appropriate than removing outliers based on raw y values when data has a trend, because we care about deviations from the *expected trend*, not from a constant mean.

9.2.4.3.7 Extension to Non-Linear Functions: The Power of Linearity in Parameters

Here's a remarkable fact: **linear regression can fit non-linear functions**, as long as the function is linear in the parameters.

For example, suppose we believe our quantity follows a quadratic trend:

$$q(t) = \beta_0 + \beta_1 t + \beta_2 t^2$$

This looks non-linear (there's a t^2 term!), but it's **linear in the parameters** $\beta_0, \beta_1, \beta_2$. We can still use least squares!

Matrix Form: Define "basis functions" $\phi_1(t), \phi_2(t), \ldots, \phi_M(t)$ and write:

$$q(t) = \sum_{j=1}^{M} \beta_j \phi_j(t) = \phi(t)^T \beta$$

For our quadratic example: $\phi_1(t) = 1$, $\phi_2(t) = t$, $\phi_3(t) = t^2$.

With measurements $y_1, \ldots, y_N$ at times $t_1, \ldots, t_N$, construct the **design matrix**:

$$\mathbf{X} = \begin{bmatrix} \phi_1(t_1) & \phi_2(t_1) & \cdots & \phi_M(t_1) \\ \phi_1(t_2) & \phi_2(t_2) & \cdots & \phi_M(t_2) \\ \vdots & \vdots & \ddots & \vdots \\ \phi_1(t_N) & \phi_2(t_N) & \cdots & \phi_M(t_N) \end{bmatrix}$$

And the measurement vector: $\mathbf{y} = [y_1, y_2, \ldots, y_N]^T$

The least squares solution is:

$$\hat{\beta} = (\mathbf{X}^T \mathbf{X})^{-1} \mathbf{X}^T \mathbf{y}$$

This **single formula** solves linear regression for:

- Polynomials: $\phi_j(t) = t^{j-1}$
- Fourier series: $\phi_j(t) = \sin(j\omega t), \cos(j\omega t)$
- Exponentials: $\phi_j(t) = e^{\lambda_j t}$
- Any combination you can imagine!

The key restriction: The model must be linear in the *parameters* β_j, even if it's non-linear in t.

9.2.4.3.8 Example: Fitting a Polynomial

```
# Generate data with a non-linear trend
np.random.seed(123)
N = 30
t = np.linspace(0, 5, N)
y_true = 2 + 3*t - 0.5*t**2  # Quadratic trend
y = y_true + np.random.normal(0, 2, N)

# Fit polynomials of degree 1, 2, and 3
degrees = [1, 2, 3]
colors = ['red', 'green', 'blue']
```

```
plt.figure(figsize=(10, 6))
plt.scatter(t, y, alpha=0.6, s=50, label='Measurements', color='black', zorder=5)

t_smooth = np.linspace(0, 5, 200)

for degree, color in zip(degrees, colors):
    # Construct design matrix
    X = np.column_stack([t**j for j in range(degree + 1)])
    X_smooth = np.column_stack([t_smooth**j for j in range(degree + 1)])

    # Least squares solution
    beta = np.linalg.solve(X.T @ X, X.T @ y)

    # Predictions
    y_pred = X @ beta
    y_smooth = X_smooth @ beta

    # Plot
    plt.plot(t_smooth, y_smooth, color=color, linewidth=2,
             label=f'Degree {degree} (SSE={np.sum((y - y_pred)**2):.1f})')

plt.xlabel('Time')
plt.ylabel('Measurement')
plt.title('Polynomial Regression: Different Model Complexities')
plt.legend()
plt.grid(True, alpha=0.3)
plt.tight_layout()
plt.show()
```

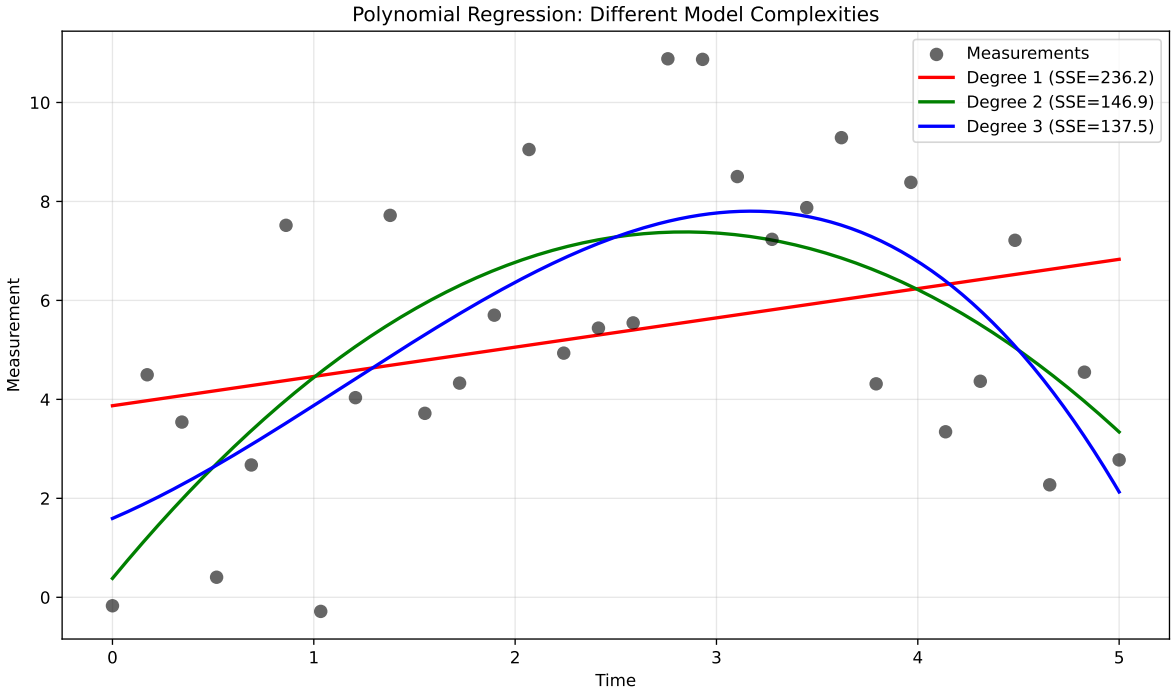

Figure 9.4: Polynomial regression example

Important lessons:

1. **Degree 1** (linear) underfits—it can't capture the curvature
2. **Degree 2** (quadratic) fits well—it matches the true model
3. **Degree 3** (cubic) fits slightly better (lower SSE) but starts to wiggle unnecessarily

This illustrates the **bias-variance tradeoff**:

- Too simple models have high bias (systematic error)
- Too complex models have high variance (overfitting to noise)
- The "just right" model matches the true complexity

9.2.4.3.9 Summary: The Power of Linear Regression

Linear regression is remarkably versatile:

- **Simple and interpretable**: Clear parameters with physical meaning
- **Computationally efficient**: Closed-form solution, no iterative optimization
- **Extensible**: Works for polynomials, Fourier series, and many non-linear basis functions
- **Foundation for advanced methods**: Underlies many machine learning algorithms

Key takeaways:

1. Residuals are the difference between data and model predictions
2. Good models have residuals that look like random noise
3. Least squares minimizes the sum of squared residuals
4. Linear regression works for non-linear functions (as long as they're linear in parameters)
5. Can be combined with outlier detection for robust fitting

10 Module 4: Entity Relationship Diagrams and Set Theory

10.1 Module Overview

10.1.1 Learning Objectives

By the end of this module, students will be able to:

- Understand the structure and visual components of Entity Relationship Diagrams, and the motivation for using them
- Be able to distinguish between attributes, entity sets and relationships
- Be able to identify weak entity sets and how to display them
- Understand the need and use of roles and sub-classes in ER diagrams
- Recognize the value of set theory for describing and working with data records
- Be able to define sets in your own words, along with derivative terms such as sub-sets and super-sets
- Be able to apply (and recognize the notation for) the basic set operations of *intersection*, *union*, *set difference*, *complement*, *cartesian product*, *cardinality*
- Recognize the value of visual diagramming approaches to sets such as Venn diagrams, and identify equivalences between set theory expressions and their corresponding Venn diagrams
- Understand and be able to apply commutative, associative, distributive properties of set operations. Similarly, understand what are complimentary sets, and disjoint sets

10.1.2 Topics Covered

- Entity Relationship Diagrams (ER Diagrams)
  - Components: attributes, entity sets, relationships
  - Weak Entity Sets
  - Roles and Subclasses in ER diagrams
- Set Theory
  - Definition of sets, subsets, and supersets

- Basic operations: intersection, union, set difference, complement, cartesian product, cardinality
- Venn Diagrams
- Properties: commutative, associative, distributive, complementary sets, disjoint sets

10.1.3 Project Milestones

Come up with 3 ideas for your final project, regardless of whether or not you are already assigned to a team.

10.2 Lecture Notes

10.2.1 From Time Series to Data Structures

In Module 3, we focused on individual time series—sequences of measurements from sensors over time. We learned how to clean data, propagate uncertainty, detect outliers, and fit trends using linear regression. These techniques are essential for understanding what each sensor tells us.

But real engineering systems don't consist of isolated sensors. Consider an environmental monitoring system for a watershed: you might have stream flow sensors at multiple locations, weather stations measuring precipitation and temperature, water quality sensors measuring pH and turbidity, and infrastructure sensors monitoring dams and bridges. Each sensor produces its own time series, but the real power comes from understanding the **relationships** among these measurements.

This raises new questions:

- How do we describe the structure of our data collection system itself?
- What entities (sensors, locations, infrastructure assets) exist in our system?
- How are these entities related to each other?
- What properties (attributes) does each entity have?

When systems become complex, with dozens or hundreds of sensors, multiple types of infrastructure, and intricate relationships, we need **structured methods for organizing and visualizing** this information. We can't just keep track of it all in our heads or in ad-hoc spreadsheets.

Entity Relationship (ER) Diagrams provide a visual, intuitive way to model the structure of complex data systems **before** we implement them in databases. Think of ER diagrams as the architectural blueprint for your data—they help you think through what information you

need to store and how different pieces relate to each other, long before you write a single line of database code.

10.2.2 Entity Relationship Diagrams

10.2.2.1 What Are Entity Relationship Diagrams?

Entity Relationship Diagrams (ER Diagrams) are graphical models that help us answer two fundamental questions about the data our system needs:

1. **What information must the database hold?** (What entities and their properties exist?)
2. **What are the relationships among components of that information?** (How do entities connect to each other?)

At their core, ER diagrams are visual representations where:

- **Boxes (rectangles)** represent collections of similar data elements (called **entity sets**)
- **Arrows and diamonds** represent connections and relationships between these elements
- **Ovals** represent properties (attributes) of the elements

ER diagrams serve as a crucial **design tool**—they're typically the first step in database design. You sketch out an ER diagram to understand your data structure, then later translate it into a more rigorous model (like a relational database schema). The diagram helps you communicate with stakeholders, identify missing information, and catch design flaws early when they're easy to fix.

Why are ER diagrams important?

- **Communication**: They provide a common visual language for engineers, database designers, and domain experts to discuss data structure
- **Design validation**: They help catch structural problems (missing entities, incorrect relationships) before implementation
- **Documentation**: They serve as living documentation of your system's data architecture
- **Foundation for implementation**: They directly inform the design of relational databases and other data structures

10.2.2.2 A Simple Example: Infrastructure Monitoring System

Let's start with a concrete example from civil engineering: a monitoring system for a set of bridges in a city.

The scenario: The city maintains several bridges and has installed sensors on each to monitor structural health. Each bridge has multiple sensors measuring different things (vibration, strain, deflection). We need to organize data about:

- The bridges themselves (location, construction year, type)
- The sensors (type, installation date, measurement units)
- The measurements collected over time
- Which sensors are installed on which bridges

Here's a simple ER diagram for this system:

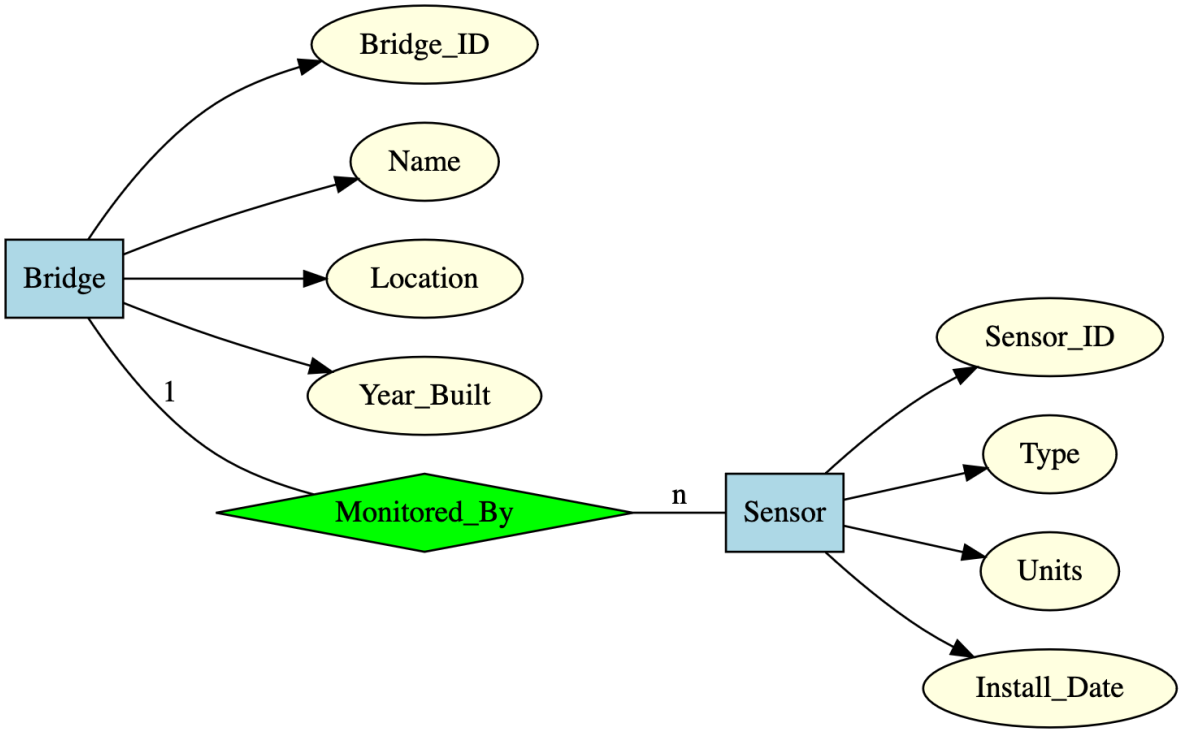

Figure 10.1: Simple ER diagram for a bridge monitoring system

Reading this diagram:

- **Bridge** and **Sensor** (blue boxes) are **entity sets**—collections of bridges and sensors
- The ovals (yellow) attached to each box are **attributes**—properties that describe each entity
- The diamond (green) labeled "Monitored_By" represents a **relationship** between bridges and sensors
- The labels "1" and "n" indicate **multiplicity**: one bridge can be monitored by many (n) sensors

This simple diagram already tells us a lot: we can see at a glance what information we're tracking and how bridges and sensors are related. As we develop our understanding, we'll add more detail and handle more complex scenarios.

10.2.2.3 Basic Components of ER Diagrams

Now let's examine each component of ER diagrams in detail, building up from the simplest concepts to more complex ones.

10.2.2.3.1 Entity Sets

An **entity** is an abstract object or "thing" that exists in the domain we're modeling. Entities have distinct identity and properties that distinguish them from other objects.

Examples of entities:

- A specific bridge (e.g., the Roberto Clemente Bridge in Pittsburgh)
- A particular sensor (e.g., strain gauge #A-2301 installed on the bridge deck)
- A monitoring station (e.g., USGS stream gauge 03049500 on the Allegheny River)
- A maintenance event (e.g., the inspection performed on June 15, 2024)

An **entity set** is a collection (or group) of similar entities that share the same type and properties. Entity sets form the basis for database tables—each entity in the set will eventually become a row in a table.

Examples of entity sets:

- **Bridge**: The collection of all bridges in the city's infrastructure system
- **Sensor**: The collection of all sensors deployed across the monitoring network
- **MonitoringStation**: All environmental monitoring stations in a watershed
- **MaintenanceEvent**: All recorded maintenance activities

Notation: Entity sets are represented as **rectangles** in ER diagrams.

Key insight: Think of entity sets as categories or types. Just as "Student" is a category containing individual students, "Bridge" is a category containing individual bridges. Each individual bridge is an entity; the collection of all bridges is the entity set.

10.2.2.3.2 Attributes

Attributes are properties or characteristics that describe the entities in an entity set. Each entity in a set has values for these attributes, and these values provide detailed information about that specific entity.

Notation: Attributes are represented as **ovals** connected to their entity set.

Example: The Bridge entity set might have these attributes:

- `Bridge_ID`: A unique identifier for the bridge
- `Name`: The bridge's name (e.g., "Fort Pitt Bridge")
- `Location`: Geographic location or address
- `Year_Built`: Construction year
- `Bridge_Type`: Type of bridge (suspension, truss, arch, etc.)
- `Material`: Construction materials used
- `Length`: Span length in meters
- `Max_Load`: Maximum load capacity in tons

Types of Attributes:

ER diagrams distinguish several special types of attributes:

1. Key Attributes (Primary Keys)

Some attributes uniquely identify each entity in the set—no two entities can have the same value for these attributes. These are called **key attributes** or **primary keys**.

- In our Bridge example, `Bridge_ID` would be a key attribute
- Each bridge has a unique ID; you can't have two different bridges with the same ID
- Notation: Underline the attribute name in the oval (e.g., Bridge_ID)

2. Multi-valued Attributes

Some attributes can have multiple values for a single entity. For example:

- A bridge might be constructed from multiple materials (steel, concrete, and asphalt)
- A sensor might measure multiple quantities simultaneously
- A building might have multiple phone numbers

Notation: Multi-valued attributes are shown as **double-lined ellipses** (an ellipse within an ellipse).

Example: If the `Material` attribute of Bridge can hold multiple values, we'd draw it as:

3. Derived Attributes

Some attributes can be **computed** or **derived** from other existing attributes rather than stored directly. For example:

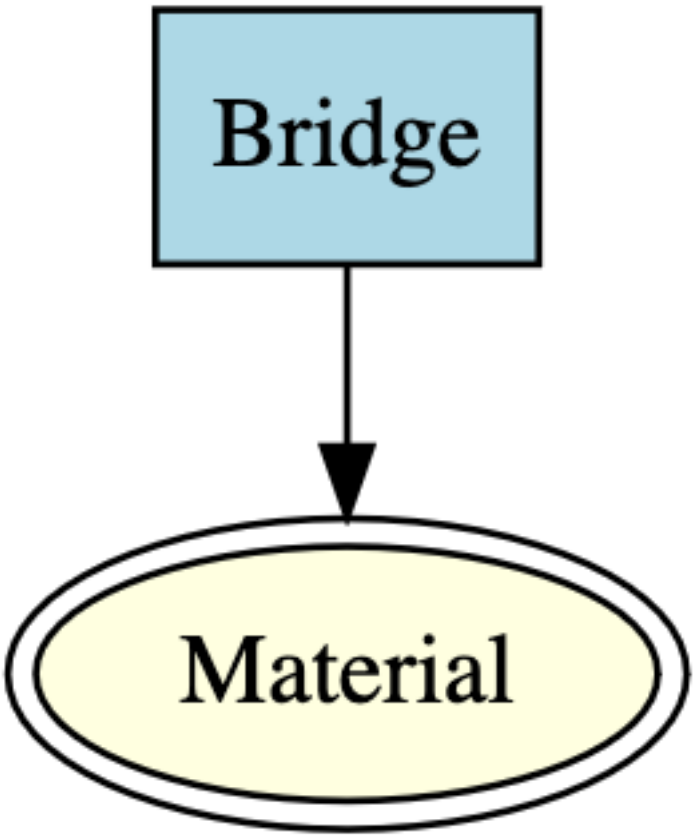

Figure 10.2: Multi-valued attribute example

- `Age` of a bridge can be derived from `Year_Built` and the current year
- `Average_Daily_Flow` might be derived from individual flow measurements
- `Total_Sensors` for a bridge could be counted from the relationship to the Sensor entity set

Notation: Derived attributes are shown as **dashed-line ellipses**.

Why distinguish derived attributes? It helps in database design—you typically don't store derived values (they're redundant and can become inconsistent), but you do want to document them in your ER diagram to show what information users can obtain.

Example diagram showing different attribute types:

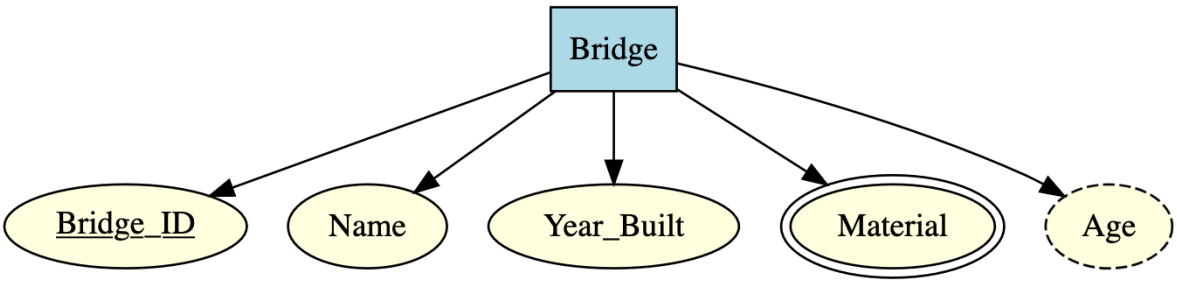

Figure 10.3: Different types of attributes in an ER diagram

In this diagram:

- `Bridge_ID` is underlined (key attribute)
- `Material` has double lines (multi-valued)
- `Age` has dashed lines (derived from `Year_Built`)

10.2.2.3.3 Relationships

Entity sets by themselves only tell us what things exist in our system. The real power of ER diagrams comes from modeling **relationships**—the connections among entity sets.

Definition: A **relationship** is a connection or association among two or more entity sets. Relationships capture how entities interact with, depend on, or relate to each other.

Notation: Relationships are represented as **diamonds** in ER diagrams, with lines connecting the diamond to the participating entity sets.

Examples from infrastructure monitoring:

- A Bridge is **Monitored_By** Sensors
- A Sensor **Records** Measurements
- A MaintenanceEvent **Performed_On** a Bridge

- An Engineer **Inspects** a Bridge

Multiplicity (Cardinality) of Relationships

A critical aspect of relationships is their **multiplicity** or **cardinality**—how many entities from each set can participate in the relationship. Understanding multiplicity is essential for correct database design.

For **binary relationships** (relationships between two entity sets), there are four common patterns:

1. One-to-One (1:1)

Each entity in set A is associated with at most one entity in set B, and vice versa.

Example: In a city management system, each `Building` has exactly one `TaxRecord`, and each `TaxRecord` belongs to exactly one `Building`.

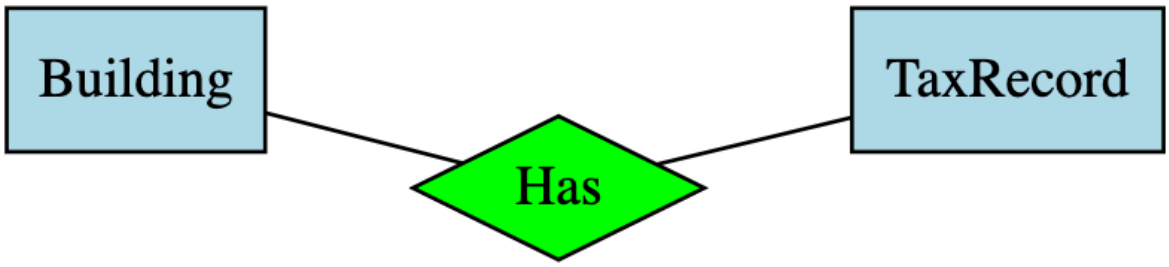

Figure 10.4: One-to-one relationship example

Notation: Arrow pointing to **both** entity sets, or labels "1" on both sides.

2. One-to-Many (1:n)

Each entity in set A can be associated with multiple entities in set B, but each entity in B is associated with at most one entity in A.

Example: One `Bridge` is monitored by many `Sensors`, but each `Sensor` is installed on only one `Bridge`.

Figure 10.5: One-to-many relationship example

Notation: Arrow points toward the "one" side, or labels "1" and "n" on the respective sides.

3. Many-to-One (n:1)

This is the reverse of one-to-many: multiple entities in set A can be associated with a single entity in set B.

Example: Many `Sensors` are manufactured by one `Manufacturer`, but each `Manufacturer` produces many sensors.

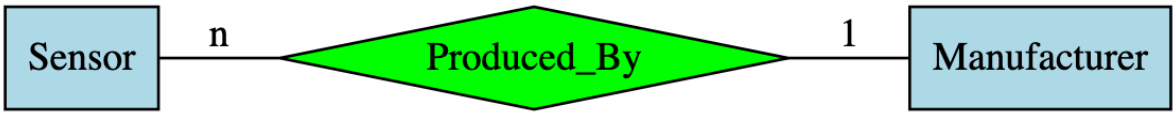

Figure 10.6: Many-to-one relationship example

Note: Many-to-one and one-to-many are essentially the same relationship viewed from different directions. The arrow points toward the "one" side.

4. Many-to-Many (n:m or m:n)

Each entity in set A can be associated with multiple entities in set B, and each entity in B can be associated with multiple entities in A.

Example: Multiple `Engineers` can inspect a given `Bridge`, and each `Engineer` can inspect multiple `Bridges`.

Figure 10.7: Many-to-many relationship example

Notation: No arrows, or labels "n" and "m" (or "n" on both sides) indicating multiple on each side.

Relationship Attributes

Relationships themselves can have attributes! These are properties that belong to the connection between entities, not to the entities themselves.

Example: Consider the `Inspects` relationship between `Engineer` and `Bridge`. We might want to record:

- `Inspection_Date`: When the inspection occurred
- `Condition_Rating`: The condition score assigned during that inspection
- `Notes`: Comments from that specific inspection

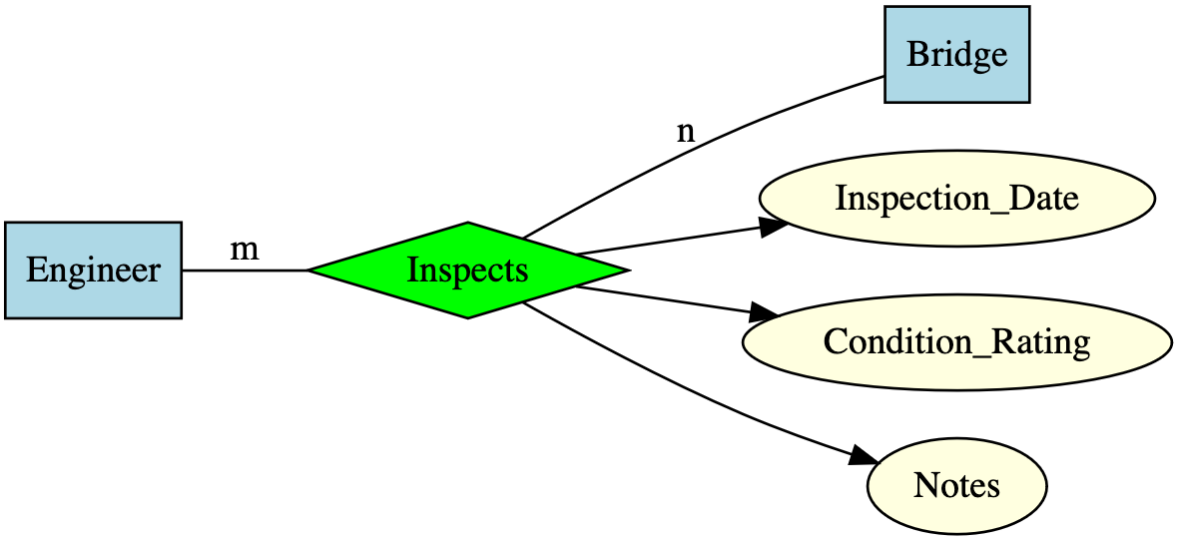

Figure 10.8: Relationship with attributes

These attributes don't belong to the Engineer alone (one engineer performs many inspections on different dates), nor to the Bridge alone (a bridge is inspected multiple times). They belong to the **relationship**—the specific event of this engineer inspecting this bridge.

Key insight: Relationship attributes are especially common in many-to-many relationships, where they capture information about each specific connection between entities.

10.2.2.4 Weak Entity Sets

Sometimes, an entity set cannot uniquely identify its entities using only its own attributes. Instead, the identity of each entity depends partly on attributes from another related entity set. This leads to the concept of **weak entity sets**.

Definition: A **weak entity set** is an entity set whose key (unique identifier) is composed of attributes, some or all of which belong to another entity set. The weak entity set depends on another entity set (called its **identifying** or **owner** entity set) for its existence and identification.

Motivation and Common Scenarios:

The principal source of weak entity sets occurs when entity sets fall into a **hierarchy of containment or ownership**. If entity set E contains subunits that belong to entities in set F, the names or identifiers of E entities may not be unique across the entire system—they're only unique within the context of their parent F entity.

Example 1: Rooms in Buildings

Consider a university campus database with `Building` and `Room` entity sets.

- Each building has a unique `Building_ID`
- Rooms are identified by `Room_Number` within a building
- But `Room_Number` alone isn't globally unique—many buildings have a "Room 101"
- A room's complete identity requires **both** its room number **and** the building it's in

In this case, `Room` is a weak entity set:

- It depends on `Building` (the identifying entity set)
- Its key is a combination of its own `Room_Number` and the `Building_ID` from the related Building entity

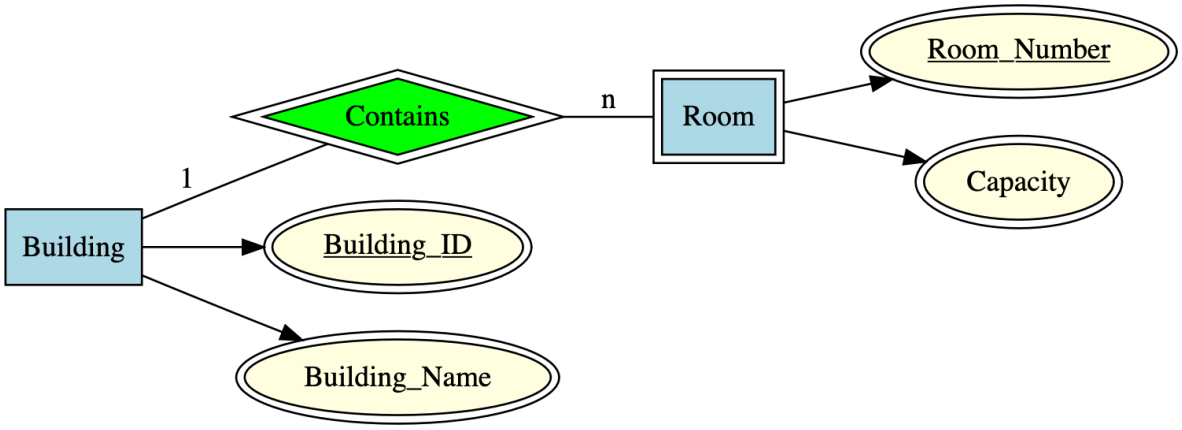

Figure 10.9: Weak entity set example: Rooms depend on Buildings

Notation:

- **Weak entity sets** are shown as **rectangles with double borders** (double-lined box)
- The **identifying relationship** (the many-one relationship connecting the weak entity to its owner) is shown as a **diamond with double borders**
- The partial key attribute (e.g., `Room_Number`) is underlined in the weak entity

Example 2: Sensor Readings in a Monitoring System

Consider our bridge monitoring system. Suppose each sensor generates readings with sequential reading numbers:

- `Sensor` is a regular (strong) entity with unique `Sensor_ID`
- `Reading` entities are identified by a `Sequence_Number` (1, 2, 3, …) within each sensor
- But `Sequence_Number` alone doesn't uniquely identify a reading globally—Sensor A's reading #5 is different from Sensor B's reading #5
- Each reading's complete identity requires the `Sensor_ID` plus its own `Sequence_Number`

Therefore, `Reading` is a weak entity set depending on `Sensor`:

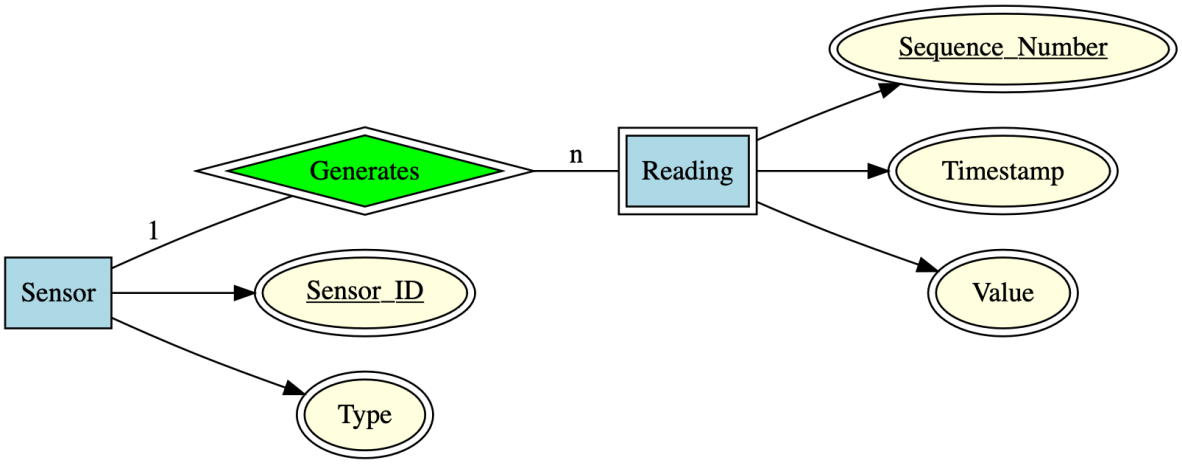

Figure 10.10: Weak entity set example: Readings depend on Sensors

Why distinguish weak entity sets?

1. **Conceptual clarity**: Makes explicit the dependency between entities
2. **Database design**: Affects how you create tables and foreign keys
3. **Integrity constraints**: Weak entities cannot exist without their owner (if you delete a Building, all its Rooms must be deleted too—this is called a cascading delete)
4. **Key structure**: Documents that the complete key is composite (spans multiple entity sets)

Key characteristics of weak entity sets:

- They have a **partial key** (discriminator)—an attribute that is unique only within the context of the owner entity
- They participate in an **identifying relationship** with their owner entity set
- The relationship is always **many-to-one** from the weak entity to the owner
- Weak entities cannot exist independently—they're existentially dependent on their owner

10.2.2.5 Roles in Relationships

Sometimes, the same entity set participates in a relationship **multiple times**, playing different roles in each participation. When this happens, we need to explicitly label the roles to distinguish them.

When are roles needed?

Roles are necessary when:

1. The same entity set appears two or more times in a single relationship
2. We need to clarify which "side" of the relationship each participation represents
3. The meaning of the relationship depends on distinguishing the different ways the entity participates

Example 1: Road Network

Consider a transportation database modeling road segments connecting intersections.

- Both endpoints of a road segment are intersections
- So the `Intersection` entity set appears twice in the `Connects` relationship
- We need to distinguish the "start" intersection from the "end" intersection

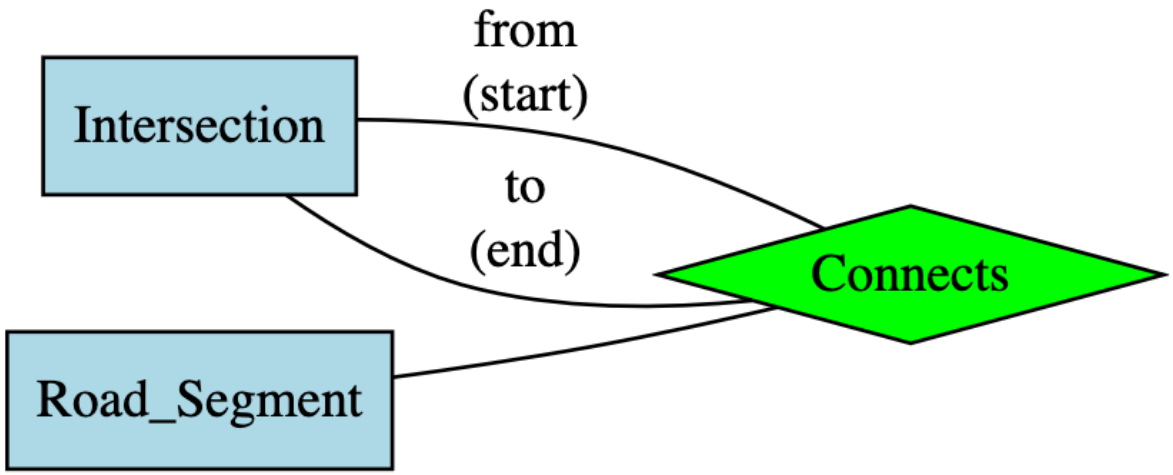

Figure 10.11: Roles in relationships: Road segments connect intersections

Here, the labels **"from"** and **"to"** are **roles** that clarify which intersection is the start point and which is the end point of the road segment.

Notation: We label the edges (lines) between the entity set and the relationship diamond with the role names.

Key insight: Roles are essential for clarity when an entity set relates to itself. Without role labels, it would be ambiguous which entity plays which part in the relationship.

10.2.2.6 Subclasses and ISA Hierarchies

Often, an entity set contains certain entities that have **special properties** not shared by all members of the set. Some entities are more specialized versions of the general entity type, with additional attributes or relationships.

Motivation: In these cases, it's useful to define **subclasses** (also called subtypes or specializations)—special-case entity sets that inherit properties from a parent entity set but add their own specific attributes and relationships.

Definition: An **ISA hierarchy** (read as "is-a") models inheritance where a subclass entity "is a" specialized version of the parent entity.

Example 1: Types of Bridges

Consider our bridge monitoring system. All bridges share common properties (location, year built, material), but different bridge types have type-specific attributes:

- **Suspension bridges** have cable properties (cable diameter, tower height, anchorage type)
- **Truss bridges** have truss configuration details (truss type, number of panels)
- **Arch bridges** have arch span and rise characteristics

Rather than adding all these specialized attributes to every bridge (leaving most empty), we create subclasses:

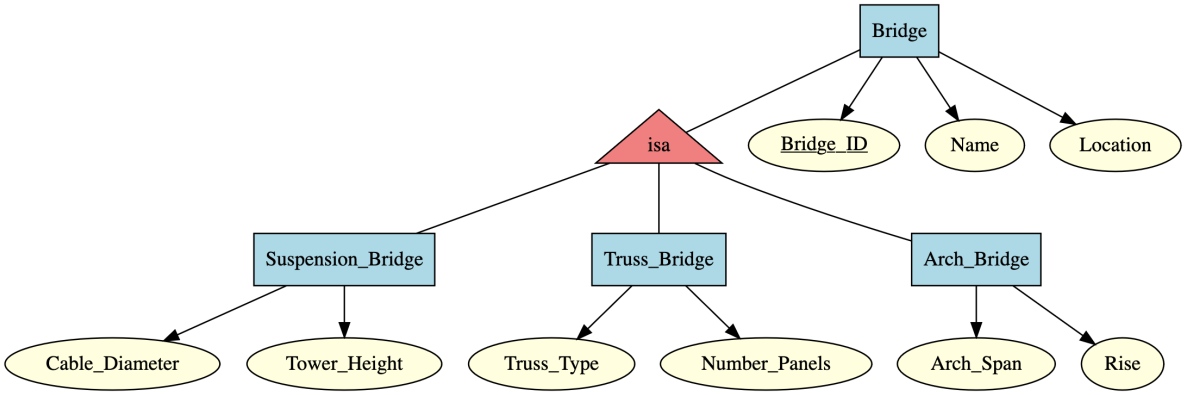

Figure 10.12: ISA hierarchy: Specialized bridge types

Reading this diagram:

- `Suspension_Bridge`, `Truss_Bridge`, and `Arch_Bridge` are **subclasses** of `Bridge`
- The **triangle** labeled "isa" represents the inheritance relationship
- Each suspension bridge **is a** bridge, so it has all Bridge attributes (ID, Name, Location) **plus** its specific attributes (Cable_Diameter, Tower_Height)
- Similarly for truss and arch bridges

Notation:

- Subclasses are connected to their parent (superclass) through a **triangle** labeled "isa"
- The triangle points toward the subclasses

- All ISA relationships are **one-to-one** (each subclass entity corresponds to exactly one parent entity)
- We don't draw arrows on ISA relationships because the one-to-one nature is implicit

Properties of ISA Hierarchies:

1. **Inheritance**: Subclasses inherit all attributes and relationships from their parent
2. **Specialization**: Subclasses add their own specific attributes and relationships
3. **Substitutability**: Anywhere a parent entity can appear, a subclass entity can appear (but not vice versa)
4. **Mutual exclusivity** (often): An entity typically belongs to only one subclass (a bridge is either suspension or truss, not both)—though this isn't always required

Why use subclasses?

- **Clarity**: Makes explicit which attributes apply to which entity types
- **Efficiency**: Avoids storing null values for attributes that don't apply
- **Extensibility**: Easy to add new specialized types without modifying the parent
- **Type-specific relationships**: Subclasses can participate in relationships unique to their type

Real-world analogy: Think of biological taxonomy. A "Mammal" is a subclass of "Animal"—it inherits all animal properties (eats, moves, reproduces) but adds mammal-specific features (warm-blooded, has fur, nurses young). Similarly, "Dog" is a subclass of "Mammal," further specializing the type.

10.2.2.7 Putting It All Together: A Complete ER Diagram

Now let's integrate all the concepts we've learned into a comprehensive ER diagram for a realistic infrastructure monitoring system. This example will demonstrate entity sets, attributes (including key, multi-valued, and derived), relationships with different multiplicities, weak entity sets, roles, and subclasses.

Scenario: A city operates a structural health monitoring system for bridges. The system tracks:

- Bridges of different types (suspension, arch, truss)
- Sensors installed on bridges
- Engineers who maintain and inspect bridges
- Inspection events with condition ratings
- Sensor readings over time

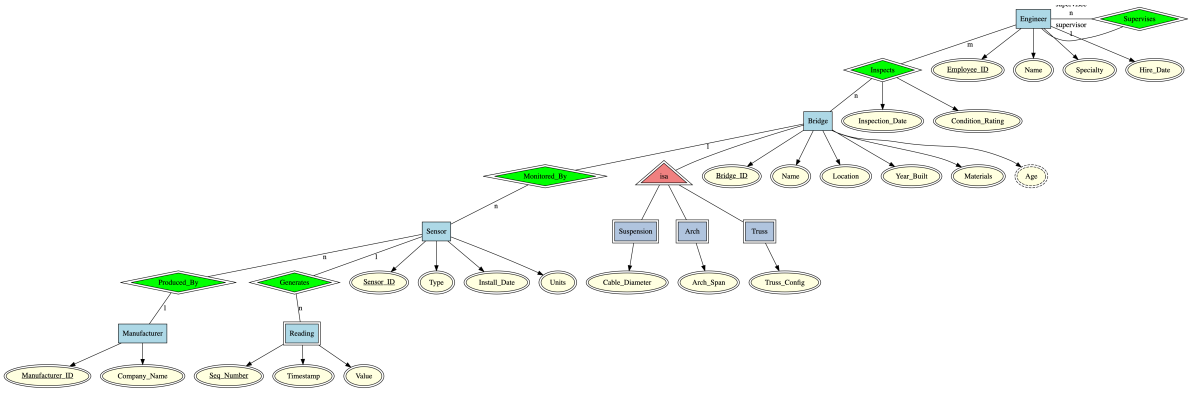

Figure 10.13: Comprehensive ER diagram for bridge monitoring system

Here's the complete ER diagram:

Reading this comprehensive diagram:

Entity Sets and Attributes:

- **Bridge**: Has `Bridge_ID` (key), `Name`, `Location`, `Year_Built`, `Materials` (multi-valued - bridges can use multiple materials), and `Age` (derived from `Year_Built`)
- **Engineer**: Has `Employee_ID` (key), `Name`, `Specialty`, `Hire_Date`
- **Sensor**: Has `Sensor_ID` (key), `Type`, `Install_Date`, `Units`
- **Manufacturer**: Has `Manufacturer_ID` (key), `Company_Name`
- **Reading** (weak entity): Has `Seq_Number` (partial key), `Timestamp`, `Value`

Relationships:

1. **Inspects** (many-to-many): Engineers inspect bridges, with attributes `Inspection_Date` and `Condition_Rating` describing each inspection event
2. **Monitored_By** (one-to-many): Each bridge is monitored by many sensors
3. **Produced_By** (many-to-one): Many sensors are produced by one manufacturer
4. **Generates** (one-to-many, identifying): Each sensor generates many readings; Reading is a weak entity depending on Sensor
5. **Supervises** (self-relationship with roles): Engineers supervise other engineers, with roles "supervisor" and "supervisee"

Subclasses (ISA Hierarchy):

- **Suspension**, **Arch**, and **Truss** are specialized types of bridges
- Each has type-specific attributes (cable diameter, arch span, truss configuration)
- They inherit all Bridge attributes

This diagram captures the complete structure of the monitoring system in a single visual representation. From this ER diagram, a database designer could:

- Identify all the tables needed
- Determine primary and foreign keys
- Understand entity dependencies and referential integrity constraints
- Plan for specialized storage of subclass-specific data

Design insights from this diagram:

1. **Readings are weak entities** because they only make sense in the context of a specific sensor
2. **Materials is multi-valued** because bridges often use multiple construction materials
3. **Age is derived** to avoid redundant storage (can compute from `Year_Built`)
4. **Inspection attributes** belong to the relationship, not to engineers or bridges alone
5. **The supervision hierarchy** uses roles to clarify the asymmetric relationship among engineers
6. **Bridge subclasses** allow type-specific attributes without cluttering the base Bridge entity

10.2.3 Set Theory for Data Management

10.2.3.1 Why Set Theory Matters for Databases

We've just learned how ER diagrams help us **design** the structure of a database—they tell us what entities and relationships we need. But once we've implemented a database, how do we actually **work with** the data?

This is where **set theory** becomes essential. Set theory provides the mathematical foundation for:

1. **Understanding what data is**: Every table in a database is fundamentally a set of records (rows). The entity set "Bridge" in our ER diagram becomes a set of bridge records in the database.
2. **Querying data**: When you ask a database "find all bridges built after 2000," you're asking for a **subset** of the bridge records. When you ask "find sensors that are on bridges AND manufactured by Company X," you're computing an **intersection** of two sets.
3. **Combining data**: Database operations like JOIN combine data from multiple tables using set operations. Understanding set theory helps you understand what these operations actually do.
4. **Reasoning about data**: Set theory gives us precise language to describe collections of data and relationships between collections.

The connection: Think of it this way:

- **ER diagrams** describe the **structure** (the blueprint)
- **Set theory** describes the **data itself** and operations on it (the mathematical foundation)
- **Databases** are the **implementation** (the actual system)

As we'll see, operations you perform on database records—filtering, combining, selecting—are fundamentally set operations. Understanding set theory helps you think clearly about what your queries do and how to construct them correctly.

10.2.3.2 Sets: The Basics

10.2.3.2.1 Defining Sets

A **set** is a collection of distinct objects, considered as a single entity. The objects in a set are called **elements** or **members** of the set.

Key properties of sets:

1. **Unordered**: The order of elements doesn't matter. {A, B, C} is the same set as {C, A, B}
2. **Distinct**: Each element appears only once. There's no such thing as "two copies" of the same element in a set
3. **Well-defined**: It must be clear whether any given object is in the set or not

Notation:

- **Curly braces** denote sets: $S = \{a, b, c\}$
- **Capital letters** typically denote sets: A, B, S, T
- **Lowercase letters** typically denote elements: a, b, x, y
- **Membership**: $x \in S$ means "x is an element of S" (x is in S)
- **Non-membership**: $x \notin S$ means "x is not an element of S"

Examples from engineering data:

Example 1: The set of bridges in our monitoring system:

$$B = \{\text{Roberto Clemente Bridge, Fort Pitt Bridge, Fort Duquesne Bridge}\}$$

Example 2: The set of sensor types:

$$T = \{\text{accelerometer, strain\_gauge, temperature, displacement}\}$$

Example 3: The set of inspection years for a particular bridge:

$$Y = \{2018, 2019, 2020, 2021, 2022, 2023, 2024\}$$

Set-builder notation:

Often, we define sets by specifying a property that elements must satisfy, rather than listing all elements. This is called **set-builder notation**:

$$S = \{x \mid \text{condition on } x\}$$

Read as: "S is the set of all x such that x satisfies this condition"

Engineering examples:

- All bridges built after 2000: $B_{new} = \{b \mid b \in Bridges \text{ and } b.year\_built > 2000\}$
- All sensors with readings exceeding a threshold: $S_{high} = \{s \mid s \in Sensors \text{ and } s.max\_reading > 100\}$
- All engineers with civil specialization: $E_{civil} = \{e \mid e \in Engineers \text{ and } e.specialty = \text{"Civil"}\}$

Special sets:

- **Empty set** (denoted $\emptyset$ or $\{\}$): The set with no elements. For example, if no bridges were built in 1950, then $\{b \mid b.year\_built = 1950\} = \emptyset$
- **Universal set** (denoted U or Ω): The set of all elements under consideration in a given context. For our bridge database, the universal set might be all infrastructure assets in the city.

10.2.3.2.2 Subsets and Supersets

Sets can be related to each other through containment.

Definition: Set A is a **subset** of set B (written $A \subseteq B$) if every element of A is also an element of B.

Mathematically: $A \subseteq B$ if and only if $\forall x : (x \in A \Rightarrow x \in B)$

(Read as: "for all x, if x is in A, then x is in B")

Definition: Set B is a **superset** of set A (written $B \supseteq A$) if $A \subseteq B$. In other words, B contains all elements of A (and possibly more).

Proper subsets: A is a **proper subset** of B (written $A \subset B$ or $A \subsetneq B$) if $A \subseteq B$ **and** $A \neq B$ (meaning B has at least one element not in A).

Engineering examples:

Example 1: Let S be all sensors, and S_{temp} be all temperature sensors. Then:

$$S_{temp} \subset S$$

Temperature sensors are a proper subset of all sensors.

Example 2: Let B be all bridges, and B_{2020} be bridges inspected in 2020:

$$B_{2020} \subseteq B$$

The bridges inspected in 2020 form a subset of all bridges.

Example 3: Consider these sets: - E = all engineers in the database - E_{civil} = engineers with civil engineering specialty - E_{senior} = engineers with more than 10 years experience - E_{lead} = senior civil engineers

Then: $E_{lead} \subseteq E_{senior}$, $E_{lead} \subseteq E_{civil}$, and both $E_{senior} \subseteq E$ and $E_{civil} \subseteq E$

Important properties:

1. Every set is a subset of itself: $A \subseteq A$
2. The empty set is a subset of every set: $\emptyset \subseteq A$ for any set A
3. Subset relation is transitive: if $A \subseteq B$ and $B \subseteq C$, then $A \subseteq C$

10.2.3.2.3 Set Membership and Cardinality

Set Membership:

We've already seen the membership notation:

- $x \in A$: "x is an element of A" or "x belongs to A"
- $x \notin A$: "x is not an element of A"

Example: If $B = \{\text{Bridge A, Bridge B, Bridge C}\}$, then:

- Bridge A $\in B$ (Bridge A is in the set)
- Bridge D $\notin B$ (Bridge D is not in the set)

Important distinction: Be careful not to confuse membership ($\in$) with the subset relation ($\subseteq$):

- Bridge A $\in B$ (an individual bridge is an *element* of the bridge set)
- $\{\text{Bridge A}\} \subseteq B$ (a set containing one bridge is a *subset* of the bridge set)

Cardinality:

The **cardinality** of a set is the number of elements it contains. We denote it as $|A|$ or $\#A$ or sometimes $card(A)$.

Examples:

- If $B = \{\text{Bridge A, Bridge B, Bridge C}\}$, then $|B| = 3$

- If $T = \{\text{accelerometer, strain\_gauge, temperature, displacement}\}$, then $|T| = 4$
- The empty set has cardinality zero: $|\emptyset| = 0$

Engineering application: Cardinality is useful for counting:

- How many sensors are installed? $|Sensors|$
- How many bridges were inspected in 2024? $|\{b \mid b.last\_inspection = 2024\}|$
- How many readings has a sensor generated? $|\{r \mid r.sensor\_id = \text{"S001"}\}|$

Properties:

1. Cardinality is always non-negative: $|A| \geq 0$
2. If $A \subseteq B$, then $|A| \leq |B|$ (subsets can't be larger than their supersets)
3. If $A \subset B$ (proper subset), then $|A| < |B|$

10.2.3.3 Operations on Sets

Just as we can perform arithmetic operations on numbers (addition, multiplication), we can perform operations on sets to create new sets. These operations are fundamental to database queries and data manipulation.

10.2.3.3.1 Union

The **union** of two sets contains all elements that are in either set (or both).

Definition: The union of sets A and B, denoted $A \cup B$, is:

$$A \cup B = \{x \mid x \in A \text{ or } x \in B\}$$

Read as: "A union B is the set of all x such that x is in A or x is in B (or both)"

Key point: The "or" here is **inclusive**—an element is in the union if it's in A, or in B, or in both.

Example 1: Simple sets

Let $A = \{1, 2, 3\}$ and $B = \{3, 4, 5\}$. Then:

$$A \cup B = \{1, 2, 3, 4, 5\}$$

Note that 3 appears in both sets, but it only appears once in the union (sets contain distinct elements).

Example 2: Engineering application

Let B_{steel} be the set of steel bridges and $B_{concrete}$ be the set of concrete bridges:

$$B_{steel} \cup B_{concrete} = \{\text{all bridges made of steel or concrete}\}$$

This represents all bridges built from either material.

Example 3: Sensor data

Let S_{2023} be sensors installed in 2023 and S_{2024} be sensors installed in 2024:

$$S_{2023} \cup S_{2024} = \{\text{all sensors installed in 2023 or 2024}\}$$

Properties of union:

1. **Commutative**: $A \cup B = B \cup A$ (order doesn't matter)
2. **Associative**: $(A \cup B) \cup C = A \cup (B \cup C)$
3. **Identity element**: $A \cup \emptyset = A$ (union with empty set does nothing)
4. **Idempotent**: $A \cup A = A$ (unioning a set with itself gives the same set)

10.2.3.3.2 Intersection

The **intersection** of two sets contains only elements that are in **both** sets.

Definition: The intersection of sets A and B, denoted $A \cap B$, is:

$$A \cap B = \{x \mid x \in A \text{ and } x \in B\}$$

Read as: "A intersection B is the set of all x such that x is in both A and B"

Example 1: Simple sets

Let $A = \{1, 2, 3, 4\}$ and $B = \{3, 4, 5, 6\}$. Then:

$$A \cap B = \{3, 4\}$$

Only elements in both sets appear in the intersection.

Example 2: Engineering application

Let B_{old} be bridges built before 1950 and B_{steel} be steel bridges:

$$B_{old} \cap B_{steel} = \{\text{bridges that are both old AND made of steel}\}$$

This represents old steel bridges.

Example 3: Database query

Let E_{civil} be engineers with civil specialty and E_{senior} be engineers with >10 years experience:

$$E_{civil} \cap E_{senior} = \{\text{senior civil engineers}\}$$

This is exactly how database queries with multiple conditions work—you're finding the intersection of multiple constraint sets.

Properties of intersection:

1. **Commutative**: $A \cap B = B \cap A$
2. **Associative**: $(A \cap B) \cap C = A \cap (B \cap C)$
3. **Identity element**: $A \cap U = A$ where U is the universal set
4. **Zero element**: $A \cap \emptyset = \emptyset$ (intersection with empty set is always empty)
5. **Idempotent**: $A \cap A = A$

10.2.3.3.3 Set Difference

The **set difference** (or **relative complement**) contains elements that are in the first set but **not** in the second set.

Definition: The set difference of A and B, denoted $A \setminus B$ or $A - B$, is:

$$A \setminus B = \{x \mid x \in A \text{ and } x \notin B\}$$

Read as: "A minus B is the set of all x such that x is in A but not in B"

Important: Set difference is **not commutative**—$A \setminus B$ is generally different from $B \setminus A$.

Example 1: Simple sets

Let $A = \{1, 2, 3, 4\}$ and $B = \{3, 4, 5, 6\}$. Then:

$$A \setminus B = \{1, 2\}$$

$$B \setminus A = \{5, 6\}$$

Example 2: Engineering application

Let B be all bridges and $B_{inspected}$ be bridges inspected this year:

$$B \setminus B_{inspected} = \{\text{bridges NOT yet inspected this year}\}$$

This gives you the bridges that still need inspection—a practical query!

Example 3: Sensor maintenance

Let S be all sensors and S_{faulty} be sensors reporting errors:

$$S \setminus S_{faulty} = \{\text{sensors that are working correctly}\}$$

Properties of set difference:

1. **Not commutative**: $A \setminus B \neq B \setminus A$ (in general)
2. $A \setminus \emptyset = A$ (removing nothing changes nothing)
3. $A \setminus A = \emptyset$ (removing all elements leaves the empty set)
4. $A \setminus B = \emptyset$ if and only if $A \subseteq B$

10.2.3.3.4 Complement

The **complement** of a set contains all elements in the universal set that are **not** in the given set.

Definition: The complement of set A, denoted A^c or $\overline{A}$ or A', is:

$$A^c = \{x \mid x \in U \text{ and } x \notin A\}$$

where U is the universal set (the set of all elements under consideration).

Equivalently: $A^c = U \setminus A$

Important: The complement depends on what you define as your universal set.

Example 1: Simple example

Let $U = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$ and $A = \{2, 4, 6, 8, 10\}$ (even numbers). Then:

$$A^c = \{1, 3, 5, 7, 9\}$$

(odd numbers)

Example 2: Engineering application

Let U be all bridges in the city and B_{old} be bridges built before 1950:

$$B_{old}^c = \{\text{bridges built in 1950 or later}\}$$

Example 3: Database application

Let U be all sensors and S_{active} be sensors currently transmitting data:

$$S_{active}^c = \{\text{sensors that are inactive or offline}\}$$

Properties of complement:

1. **Double complement**: $(A^c)^c = A$ (complement of complement gives back the original)
2. **Complement of universal set**: $U^c = \emptyset$
3. **Complement of empty set**: $\emptyset^c = U$
4. **De Morgan's Laws** (very important):
    - $(A \cup B)^c = A^c \cap B^c$
    - $(A \cap B)^c = A^c \cup B^c$

De Morgan's Laws in plain language:

- "NOT (A or B)" is the same as "(NOT A) and (NOT B)"
- "NOT (A and B)" is the same as "(NOT A) or (NOT B)"

These are fundamental to logic and database queries!

10.2.3.3.5 Cartesian Product

The **Cartesian product** creates a set of **ordered pairs** by combining elements from two sets. This operation is fundamental to understanding how database tables relate to each other.

Definition: The Cartesian product of sets A and B, denoted $A \times B$, is:

$$A \times B = \{(a, b) \mid a \in A \text{ and } b \in B\}$$

Read as: "A cross B is the set of all ordered pairs (a,b) where a is in A and b is in B"

Important: The result is a set of **ordered pairs**—(a,b) is different from (b,a).

Example 1: Simple sets

Let $A = \{1, 2\}$ and $B = \{x, y\}$. Then:

$$A \times B = \{(1, x), (1, y), (2, x), (2, y)\}$$

Notice: $|A \times B| = |A| \times |B| = 2 \times 2 = 4$

Example 2: Engineers and Bridges

Let $E = \{\text{Alice, Bob}\}$ (engineers) and $B = \{\text{Bridge A, Bridge B}\}$ (bridges). Then:

$$E \times B = \{(\text{Alice, Bridge A}), (\text{Alice, Bridge B}), (\text{Bob, Bridge A}), (\text{Bob, Bridge B})\}$$

This represents all possible pairings of engineers with bridges—exactly what you get when you JOIN two database tables without any filtering condition!

Example 3: Sensor locations and types

Let $L = \{\text{North, South, East, West}\}$ (locations) and $T = \{\text{temperature, strain}\}$ (sensor types). Then:

$$L \times T = \{(\text{North, temperature}), (\text{North, strain}), (\text{South, temperature}), ...\}$$

This gives all possible combinations of locations and sensor types—8 possibilities total.

Connection to Database Tables:

The Cartesian product is **exactly what happens** when you JOIN two tables without any conditions:

```
SELECT * FROM Engineers, Bridges;
```

This query produces $|Engineers| \times |Bridges|$ rows—every engineer paired with every bridge. That's why we usually add WHERE conditions to filter the Cartesian product down to meaningful relationships!

Properties of Cartesian product:

1. **Not commutative**: $A \times B \neq B \times A$ (order matters in ordered pairs)
2. **Cardinality**: $|A \times B| = |A| \times |B|$
3. **Associative** (in a sense): $(A \times B) \times C \cong A \times (B \times C) \cong A \times B \times C$ (can extend to triples, etc.)
4. $A \times \emptyset = \emptyset$ (product with empty set is empty)

Extended Cartesian product:

We can extend to multiple sets: $A_1 \times A_2 \times \cdots \times A_n$ produces n-tuples (ordered sequences of n elements).

Database connection: A database table with n columns can be viewed as a subset of a Cartesian product:

$$\text{Bridge table} \subseteq \text{IDs} \times \text{Names} \times \text{Locations} \times \text{Years} \times ...$$

Not all possible combinations appear—only valid, meaningful bridge records. But the structure is fundamentally a Cartesian product filtered by constraints.

10.2.3.4 Visualizing Sets with Venn Diagrams

Venn diagrams are visual representations of sets and their relationships using overlapping circles (or other closed curves). They provide an intuitive way to understand set operations and relationships.

Basic structure:

- Each set is represented by a circle or closed curve
- The universal set U is typically represented by a rectangle containing all circles
- Elements are points inside the circles
- Overlapping regions represent intersections

How to read Venn diagrams:

The position of a point tells you which set(s) it belongs to:

- Inside circle A only $\rightarrow$ element is in $A \setminus B$
- Inside circle B only $\rightarrow$ element is in $B \setminus A$
- In the overlap of A and B $\rightarrow$ element is in $A \cap B$
- Outside both circles (but in rectangle) $\rightarrow$ element is in $(A \cup B)^c$

Example: Bridge classification

Consider bridges classified by two properties: material (steel vs. not steel) and age (old vs. new).

Let: - S = set of steel bridges - O = set of old bridges (built before 1950)

The Venn diagram divides all bridges into four regions:

1. $S \cap O$: Old steel bridges
2. $S \cap O^c$: New steel bridges
3. $S^c \cap O$: Old non-steel bridges
4. $S^c \cap O^c$: New non-steel bridges

Translating between notation and diagrams:

Practice moving back and forth between symbolic notation and visual representation:

Set Expression	Description	Visual Region
$A \cap B$	Elements in both A and B	Overlap of circles
$A \cup B$	Elements in either A or B (or both)	Both circles combined
$A \setminus B$	Elements in A but not B	Part of A circle not overlapping B

Set Expression	Description	Visual Region
$(A \cup B)^c$	Elements in neither A nor B	Outside both circles
$A^c \cap B^c$	Elements in neither A nor B (same as above)	Outside both circles

Complex expressions:

Venn diagrams help verify identities like De Morgan's Laws:

- $(A \cup B)^c$ (complement of union) and $A^c \cap B^c$ (intersection of complements) shade the same region
- $(A \cap B)^c$ (complement of intersection) and $A^c \cup B^c$ (union of complements) shade the same region

Three-set Venn diagrams:

With three sets, you get more complex overlapping patterns—up to 8 distinct regions representing all possible combinations of membership:

- In none of the sets
- In only A, only B, or only C
- In A and B but not C, A and C but not B, or B and C but not A
- In all three sets: $A \cap B \cap C$

Engineering application:

Consider monitoring system alerts with three conditions:

- H = high strain readings
- V = high vibration readings
- T = temperature anomalies

Engineers might want to query different regions:

- $H \cap V \cap T$: Bridges with all three alert types (critical priority)
- $H \cup V \cup T$: Bridges with any alert type (all monitored bridges)
- $H \cap V \cap T^c$: High strain and vibration but normal temperature
- $(H \cup V \cup T)^c$: Bridges with no alerts (all clear)

Venn diagrams make these queries visually intuitive.

10.2.3.5 Properties of Set Operations

Understanding the algebraic properties of set operations helps you manipulate set expressions, simplify queries, and reason about data.

10.2.3.5.1 Commutative Property

An operation is **commutative** if the order of operands doesn't matter.

Commutative operations:

1. **Union**: $A \cup B = B \cup A$
    - Example: (Steel bridges) (Old bridges) = (Old bridges) (Steel bridges)
2. **Intersection**: $A \cap B = B \cap A$
    - Example: (Civil engineers) (Senior staff) = (Senior staff) (Civil engineers)

Non-commutative operations:

1. **Set difference**: $A \setminus B \neq B \setminus A$ (generally)
    - Example: (All sensors) (Faulty sensors) (Faulty sensors) (All sensors)
    - The first gives working sensors; the second is empty!
2. **Cartesian product**: $A \times B \neq B \times A$
    - Example: Engineers × Bridges Bridges × Engineers
    - The ordered pairs (e, b) and (b, e) are different

10.2.3.5.2 Associative Property

An operation is **associative** if grouping doesn't matter when combining three or more sets.

Associative operations:

1. **Union**: $(A \cup B) \cup C = A \cup (B \cup C)$
    - You can write $A \cup B \cup C$ without ambiguity
    - Example: (Steel bridges) (Concrete bridges) (Wood bridges) groups any way
2. **Intersection**: $(A \cap B) \cap C = A \cap (B \cap C)$
    - You can write $A \cap B \cap C$ without ambiguity
    - Example: Finding engineers who are civil AND senior AND experienced in bridges

Why this matters: Associativity means you can process multiple conditions in any order—useful for query optimization in databases.

10.2.3.5.3 Distributive Property

Distributive laws describe how one operation distributes over another.

Key distributive properties:

1. **Intersection distributes over union**:

$$A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$$

   Example: Steel bridges that are (old or long) = (Steel bridges that are old) or (Steel bridges that are long)

2. **Union distributes over intersection**:

$$A \cup (B \cap C) = (A \cup B) \cap (A \cup C)$$

   Example: Sensors that are (temperature OR (strain AND high-precision)) = (temperature OR strain) AND (temperature OR high-precision)

Why this matters: These laws allow you to rewrite queries in equivalent forms, which can be more efficient to compute or clearer to understand.

Absorption laws (related):

- $A \cup (A \cap B) = A$
- $A \cap (A \cup B) = A$

These simplify redundant conditions.

10.2.3.5.4 Special Sets and Relationships

Empty Set ($\emptyset$):

Properties:

- $A \cup \emptyset = A$ (union identity)
- $A \cap \emptyset = \emptyset$ (intersection with empty is empty)
- $A \setminus \emptyset = A$
- $\emptyset \subseteq A$ for all A (empty set is subset of everything)

Universal Set (U):

Properties:

- $A \cup U = U$
- $A \cap U = A$ (intersection identity)

- $A \subseteq U$ for all A in the context
- $U^c = \emptyset$

Complementary Sets:

Two sets A and B are **complementary** if:

- $A \cup B = U$ (together they cover everything)
- $A \cap B = \emptyset$ (they don't overlap)
- $B = A^c$ (B is the complement of A)

Example: Active sensors and inactive sensors are complementary—every sensor is either active or inactive, but not both.

Disjoint Sets:

Two sets A and B are **disjoint** (or **mutually exclusive**) if:

$$A \cap B = \emptyset$$

They have no elements in common.

Examples:

- Suspension bridges and truss bridges are disjoint (a bridge can't be both types)
- Sensors installed in 2020 and sensors installed in 2021 are disjoint
- Temperature sensors and strain gauges might be disjoint (if types don't overlap)

Properties of disjoint sets:

- If A and B are disjoint: $|A \cup B| = |A| + |B|$
- This is the **addition principle** of counting

Partition:

A collection of sets $A_1, A_2, \ldots, A_n$ is a **partition** of set S if:

1. They're mutually disjoint: $A_i \cap A_j = \emptyset$ for $i \neq j$
2. Their union is S: $A_1 \cup A_2 \cup \cdots \cup A_n = S$
3. None are empty: $A_i \neq \emptyset$

Example: Bridge types (suspension, truss, arch, beam) partition the set of all bridges—every bridge is exactly one type.

Why partitions matter: They're fundamental to classification systems and database table designs. Entity subclasses in ER diagrams often represent partitions.

10.2.3.6 Connecting Sets to ER Diagrams

Now we can bring everything together: **ER diagrams describe structure, set theory provides the mathematical foundation, and database queries perform set operations**.

10.2.3.6.1 Entity Sets as Sets

Every **entity set** in an ER diagram corresponds to a **set** in set theory:

- The Bridge entity set → the set of all bridge records
- The Sensor entity set → the set of all sensor records
- Each entity (a specific bridge) → an element of the set

Attributes define subsets:

- Bridges built after 2000 → $\{b \in Bridges \mid b.year\_built > 2000\}$ (a subset)
- Steel bridges → $\{b \in Bridges \mid \text{"steel"} \in b.materials\}$ (a subset)

10.2.3.6.2 Relationships as Set Operations

One-to-many relationships can be expressed using sets:

For the relationship "Bridge is monitored by Sensors":

- For each bridge b, there's a set of sensors: $Sensors(b) = \{s \in Sensors \mid s.bridge\_id = b.id\}$
- Finding all sensors on any bridge in set B_{subset}:

$$\bigcup_{b \in B_{subset}} Sensors(b)$$

Many-to-many relationships are subsets of Cartesian products:

The "Engineer Inspects Bridge" relationship is a subset of $Engineers \times Bridges$:

$$Inspects \subseteq Engineers \times Bridges$$

Not every $(engineer, bridge)$ pair is in the relationship—only those where that engineer actually inspected that bridge.

10.2.3.6.3 Queries as Set Operations

WHERE clause = intersection:

```
SELECT * FROM Bridges
WHERE material = 'steel' AND year_built > 2000;
```

This computes: $B_{steel} \cap B_{new}$

OR conditions = union:

```
SELECT * FROM Bridges
WHERE material = 'steel' OR material = 'concrete';
```

This computes: $B_{steel} \cup B_{concrete}$

NOT conditions = complement or difference:

```
SELECT * FROM Bridges
WHERE bridge_id NOT IN (SELECT bridge_id FROM Inspections WHERE year = 2024);
```

This computes: $Bridges \setminus Bridges_{inspected\_2024}$

JOIN = Cartesian product + filter:

```
SELECT * FROM Engineers JOIN Bridges ON Engineers.specialty = Bridges.type;
```

This computes: $\{(e, b) \in Engineers \times Bridges \mid e.specialty = b.type\}$

(A filtered subset of the Cartesian product)

Aggregation = cardinality:

```
SELECT COUNT(*) FROM Sensors WHERE type = 'temperature';
```

This computes: $|S_{temperature}|$

10.2.3.6.4 Design Implications

Understanding the set-theoretic foundation helps with:

1. **Query optimization**: Recognizing equivalent set expressions
   - $A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$ suggests different query plans
2. **Understanding results**: Knowing what data you'll get
   - Cartesian product without JOIN condition explodes data size

- UNION removes duplicates (set union), UNION ALL doesn't

3. **Constraint specification**: Defining data integrity rules
   - Disjoint subclasses → partition constraint
   - Foreign keys → subset constraint (child IDs must be in parent IDs)
4. **Reasoning about coverage**: Do your queries cover all cases?
   - Have you partitioned the search space?
   - Are conditions mutually exclusive or overlapping?

Example: Suppose you want to ensure all bridges are categorized by age. You might verify:

$$(Bridges_{old} \cup Bridges_{new} = Bridges) \wedge (Bridges_{old} \cap Bridges_{new} = \emptyset)$$

This confirms old and new bridges form a partition—complete coverage, no overlap.

11 Module 5: The Relational Model

11.1 Module Overview

11.1.1 Learning Objectives

By the end of this module, students will be able to:

- Understand the motivation behind the development of the relational model, and be able to provide a definition of it in your own words
- Describe the high-level connection between set theory (from Module 4) and the relational model
- Identify and relate the following concepts: relations, attributes, tuples and types; as well as connect them to their corresponding analogous terms: tables, columns, rows and domain
- Describe the meaning of *key* attributes and understand the difference between a schema and an instance of a database
- Understand the steps of relational database creation: design (using a DDL), initial data load, and query execution
- Identify the function of a query and how natural language queries for relational databases can be formalized with relational algebra
- Understand the link between relational algebra and SQL, as well as the fact that queries return relations
- Apply and reason through the application of the following relational algebra operators and their compositions: *select*, *project*, *cross product*, *natural join*, *theta join*, *union*, *difference*, *intersection*, *rename*
- Translate E/R diagrams to relational models of databases

11.1.2 Topics Covered

- **Introduction to the Relational Model**
  - Historical context and motivation
  - Definition of the relational model
  - Connection to set theory from Module 4
- **Relational Model Components**

  - Relations, attributes, tuples, types
  - Analogous terms: tables, columns, rows, domains
  - Key attributes and constraints
  - Schema vs. instance

- **Relational Database Creation Process**
  - Database design using DDL (Data Definition Language)
  - Initial data loading
  - Query execution

- **Queries and Query Languages**
  - Purpose and function of queries
  - Formalizing natural language queries with relational algebra
  - Link between relational algebra and SQL
  - Query results as relations

- **Relational Algebra Operators**
  - Select (σ)
  - Project (π)
  - Cross Product ($\times$)
  - Natural Join ($\bowtie$)
  - Theta Join ($\bowtie_\theta$)
  - Union ($\cup$)
  - Difference ($-$)
  - Intersection ($\cap$)
  - Rename (ρ)
  - Composing operators for complex queries

- **From ER Diagrams to Relational Models**
  - Converting entity sets to relations
  - Translating relationships (1:1, 1:n, n:m)
  - Handling weak entity sets
  - Representing ISA hierarchies
  - Complete example: Bridge monitoring system from Module 4

11.1.3 Project Milestones

Meet with your teammates to discuss final project ideas and select a winning one.

11.2 Lecture Notes

11.2.1 From ER Diagrams to Actual Databases

In Module 4, we learned two fundamental concepts: Entity Relationship (ER) diagrams for visualizing database structure, and set theory for understanding the mathematical foundations of data. ER diagrams helped us answer questions like "What entities exist in our system?" and "How are they related?" Set theory gave us the mathematical language to describe collections of data and operations on them.

But ER diagrams are **design tools**—they help us think through and communicate what our database should look like. They don't actually store data or answer queries. To build a working database system, we need to translate our ER diagram design into a concrete **implementation**. This is where the **relational model** comes in.

The relational model provides the foundation for nearly all modern database systems. It gives us:

- A precise way to **represent** data (as relations/tables)
- A formal language to **query** data (relational algebra)
- Rules for **translating** ER diagrams into actual database structures

An important insight: Any database design—and the requirements for it—ultimately comes from **envisioned use cases**. In other words, understanding how users will interact with your database (what questions they'll ask, what queries they'll run) is the most important starting point for design. You might have a perfect ER diagram, but if it doesn't support the queries your users need to run efficiently, it's not serving its purpose.

As we progress through this module, keep in mind this design principle: **the structure of your database should be driven by the queries you anticipate running**. The relational model gives us both the structure (how to organize data in tables) and the query language (relational algebra) to make this happen.

11.2.2 Introduction to the Relational Model

The relational model is used by almost all major commercial database systems today—from PostgreSQL and MySQL to Oracle and Microsoft SQL Server. Despite being over 40 years old, it remains the dominant approach for organizing and querying data.

At its core, a **Database Management System (DBMS)** provides efficient, reliable, convenient, and safe multi-user storage of and access to massive amounts of persistent data. The relational model is the theoretical foundation that makes this possible, combined with high-level query languages that are both simple to use and expressive enough to handle complex data needs.

11.2.2.1 What is the Relational Model?

The relational model organizes data using a few simple but powerful concepts:

- **Database** = a set of named **relations** (also called **tables**)
- Each **relation** has a set of named **attributes** (also called **columns**)
- Each **tuple** (also called **row**) contains a value for each attribute
- Each attribute has a **type** (also called **domain**) that constrains what values it can hold

Additionally:

- A **key** is an attribute (or set of attributes) whose value uniquely identifies each tuple in the relation
- **NULL** is a special value representing "unknown" or "undefined"

Two critical concepts distinguish the structure from the data itself:

- **Schema** = the structural description of relations in the database (the set of relations, their attributes, types, and constraints)
- **Instance** = the actual contents of the database at a given point in time (the actual data)

The schema is like a blueprint—it describes what the database looks like. The instance is the current state—the actual rows of data stored at this moment.

11.2.2.2 Historical Context and Motivation

The relational model was introduced by Edgar F. Codd in 1970 in his seminal paper "A Relational Model of Data for Large Shared Data Banks." Before Codd's work, databases were navigational—you had to write code that explicitly navigated through data structures using pointers and links, similar to traversing linked lists or trees in programming.

> **Note**
>
> I actually had not gone to the paper itself before, but in confirming what I got in these notes from Claude, I went to the source (so that I could include the link to the URL) today. The foresight of this 55 year old paper is quite remarkable. With the relational algebra we went over today, you might be able to understand the whole paper.

Codd's revolutionary insight was that data could be organized as simple tables (relations) and queried using high-level, declarative languages based on mathematical logic. Instead of telling the computer *how* to find data (navigate this pointer, follow that link), you simply describe *what* you want, and the DBMS figures out how to retrieve it efficiently.

This abstraction dramatically simplified database programming and made databases accessible to non-programmers. It's why SQL (Structured Query Language), which implements Codd's relational algebra, became ubiquitous.

11.2.2.3 Connection to Set Theory

The relational model is fundamentally grounded in **set theory**, which we studied in Module 4. This connection isn't coincidental—Codd built the relational model on the mathematical foundation of sets.

Here's how the concepts map:

- A **relation** is a set of tuples (no duplicate rows, unordered)
- **Relational algebra operations** (select, project, join, union, etc.) are set operations
- The **Cartesian product** from set theory becomes the cross product in relational algebra
- **Union**, **intersection**, and **difference** work exactly as they do in set theory

This mathematical foundation gives the relational model:

1. **Precision**: Operations are well-defined mathematically
2. **Compositionality**: Query results are relations, so you can nest queries
3. **Optimization**: Mathematical properties allow DBMSs to rewrite queries for efficiency

Throughout this module, you'll see how the set operations you learned in Module 4 translate directly into relational algebra operators for querying databases.

11.2.3 Relational Model Components

Now let's examine the core components of the relational model in detail. We'll use examples from the bridge monitoring system we designed in Module 4 to make these concepts concrete.

11.2.3.1 Relations, Tuples, and Attributes

At the mathematical core, a **relation** is a set of tuples that share the same structure. Each tuple is an ordered collection of values, and each value corresponds to an **attribute** of the relation.

Formal definition: If we have attributes $A_1, A_2, \ldots, A_n$ with corresponding domains (sets of allowed values) $D_1, D_2, \ldots, D_n$, then a relation R is a subset of the Cartesian product:

$$R \subseteq D_1 \times D_2 \times \cdots \times D_n$$

In simpler terms: a relation is a set of n-tuples, where each tuple has the same structure—values for the same set of attributes.

Example: Consider a Bridge relation with attributes (Bridge_ID, Name, Year_Built, Length). A tuple in this relation might be:

$$(B001, \text{"Roberto Clemente Bridge"}, 1928, 244)$$

This tuple represents one bridge in our monitoring system.

Key properties of relations:

1. **Tuples are unordered**: There's no "first" or "last" tuple (they're elements of a set)
2. **No duplicate tuples**: Each tuple in a relation must be unique (set property)
3. **Attributes are atomic**: Each attribute value is indivisible (no nested structures)

11.2.3.2 Tables, Rows, and Columns: The Common Terminology

While "relation," "tuple," and "attribute" are the formal mathematical terms, database practitioners use more intuitive terminology:

Mathematical Term	Database Term	What It Means
Relation	Table	A collection of related data
Tuple	Row	A single record in the table
Attribute	Column	A property or field of the records

Example: Here's a Bridge table visualized:

Bridge_ID	Name	Year_Built	Length
B001	Roberto Clemente Bridge	1928	244
B002	Fort Pitt Bridge	1959	1201
B003	Fort Duquesne Bridge	1969	1201

In this table:

- Each **row** is a tuple representing one bridge
- Each **column** is an attribute (property) of bridges
- The entire **table** is the Bridge relation

Throughout this course, we'll use both sets of terms interchangeably, but remember they refer to the same concepts.

11.2.3.3 Domains and Types

Every attribute has a **domain** (or **type**)—the set of all possible values that attribute can take.

Examples of domains:

- `Bridge_ID`: String of format "B" followed by 3 digits (e.g., "B001", "B002")
- `Year_Built`: Integer between 1800 and current year
- `Length`: Positive real number (meters)
- `Sensor_Type`: Enumeration from {"accelerometer", "strain_gauge", "temperature", "displacement"}

Domains serve several purposes:

1. **Data integrity**: Prevent nonsensical values (e.g., negative lengths, future construction years)
2. **Storage optimization**: DBMS can use appropriate storage formats (integers vs. strings vs. floats)
3. **Query validation**: Catch errors at query time (e.g., comparing incompatible types)

In practice, database systems provide built-in types (INTEGER, VARCHAR, DATE, FLOAT, etc.) and allow you to define constraints that further restrict domains.

11.2.3.4 Keys and Constraints

A **key** is an attribute (or set of attributes) that uniquely identifies each tuple in a relation. No two tuples can have the same values for all key attributes.

Example: In our Bridge relation, `Bridge_ID` is a key—each bridge has a unique ID:

Bridge_ID (KEY)	Name	Year_Built	Length
B001	Roberto Clemente Bridge	1928	244
B002	Fort Pitt Bridge	1959	1201
B003	Fort Duquesne Bridge	1969	1201

Notice that two bridges can have the same `Length` or even the same `Name`, but they must have different `Bridge_ID` values.

11.2.3.5 Schema vs. Instance

These two concepts are fundamental to understanding databases:

Schema = The structure or blueprint of the database

The schema defines:

- What relations exist
- What attributes each relation has
- The types/domains of each attribute
- What keys and constraints apply

The schema is **relatively stable**—it changes only when you redesign the database structure.

Instance = The actual data in the database at a specific moment

The instance contains:

- The actual tuples (rows) in each relation
- The specific values stored

The instance **changes frequently**—every time you insert, update, or delete data.

Example:

Bridge Schema:

```
Bridge(Bridge_ID: STRING, Name: STRING, Year_Built: INTEGER, Length: REAL)
  PRIMARY KEY: Bridge_ID
  CONSTRAINT: Year_Built > 1800
```

Bridge Instance (October 2024):

Bridge_ID	Name	Year_Built	Length
B001	Roberto Clemente Bridge	1928	244
B002	Fort Pitt Bridge	1959	1201
B003	Fort Duquesne Bridge	1969	1201

Bridge Instance (November 2024, after adding a new bridge):

Bridge_ID	Name	Year_Built	Length
B001	Roberto Clemente Bridge	1928	244
B002	Fort Pitt Bridge	1959	1201

Bridge_ID	Name	Year_Built	Length
B003	Fort Duquesne Bridge	1969	1201
B004	Smithfield Street Bridge	1883	361

The schema remained the same—only the instance changed.

Key insight: When we design databases (including when we translate ER diagrams to relational schemas), we're defining the **schema**. When users interact with databases (querying, inserting, updating data), they're working with the **instance**.

11.2.4 Relational Database Creation Process

Once you've designed your database schema (often starting from an ER diagram), you need to actually create and populate the database. This process typically involves three main steps: defining the structure using DDL, loading initial data, and executing queries. Let's examine each step.

11.2.4.1 Database Design with DDL

DDL (Data Definition Language) is the subset of database commands used to define and modify the structure of your database—creating tables, specifying attributes, defining keys and constraints.

Here's an example of creating our Bridge table using SQL DDL syntax (note: while we're focusing on relational algebra in this module, seeing a bit of DDL helps understand the connection between abstract schemas and concrete databases):

```
CREATE TABLE Bridge (
    Bridge_ID VARCHAR(10) PRIMARY KEY,
    Name VARCHAR(100) NOT NULL,
    Year_Built INTEGER CHECK (Year_Built > 1800)
);
```

This DDL statement:

- Creates a table named `Bridge`
- Defines four attributes with their types
- Specifies `Bridge_ID` as the primary key
- Adds NOT NULL constraint (Name must have a value)

Some Key DDL operations:

- `CREATE TABLE`: Define a new relation
- `ALTER TABLE`: Modify an existing table structure
- `DROP TABLE`: Delete a table

11.2.4.2 Data Loading

After defining the schema, you need to populate the database with actual data. This is called **data loading** or **data insertion**.

Example: Loading bridges into our system:

```
INSERT INTO Bridge (Bridge_ID, Name, Year_Built, Length)
VALUES ('B001', 'Roberto Clemente Bridge', 1928, 244);

INSERT INTO Bridge (Bridge_ID, Name, Year_Built, Length)
VALUES ('B002', 'Fort Pitt Bridge', 1959, 1201);

INSERT INTO Bridge (Bridge_ID, Name, Year_Built, Length)
VALUES ('B003', 'Fort Duquesne Bridge', 1969, 1201);
```

For large datasets, DBMSs typically provide bulk loading utilities that can read data from CSV files or other formats:

```
COPY Bridge FROM '/path/to/bridges.csv'
WITH (FORMAT CSV, HEADER TRUE);
```

Data loading considerations:

- **Constraint validation**: The DBMS checks all constraints during insertion (keys, but also other constraints we haven't seen yet such as foreign keys, CHECK constraints)
- **Transaction handling**: Large data loads are often wrapped in transactions for consistency
- **Performance**: Bulk loading is much faster than individual inserts

11.2.4.3 Query Execution

Once your database has structure (schema) and data (instance), users can **query** it—asking questions and retrieving information.

Example natural language query: "Find all bridges built before 1950"

This would be expressed in SQL as:

```
SELECT * FROM Bridge
WHERE Year_Built < 1950;
```

And would return:

Bridge_ID	Name	Year_Built	Length
B001	Roberto Clemente Bridge	1928	244
B004	Smithfield Street Bridge	1883	361

What happens during query execution?

1. **Parsing**: The DBMS parses the query to check syntax
2. **Validation**: Checks that tables and columns exist, types match, etc.
3. **Optimization**: The DBMS creates an execution plan (how to actually retrieve the data efficiently)
4. **Execution**: The plan is executed, accessing the stored data
5. **Result**: The result is returned as a new relation (table)

Key insight: The relational model's power comes from **declarative queries**—you specify *what* you want, not *how* to get it. The DBMS handles the "how" through query optimization. This is fundamentally different from navigational databases where you had to specify the exact path through data structures.

In the next sections, we'll learn **relational algebra**—the formal mathematical language that underlies these queries and allows us to reason about what queries compute.

11.2.5 Queries and Query Languages

Now that we understand how databases are structured and created, let's focus on how we actually *use* them—by asking questions and retrieving information through **queries**.

A **query** is a request for information from a database. Some queries are easy to pose ("Find all bridges"), while others are more complex ("Find all sensors on bridges built before 1950 that have recorded readings above threshold in the past month"). Similarly, some queries are easy for a DBMS to execute efficiently, while others require significant computation.

An important note: the term "query language" is somewhat misleading—these languages are also used to **modify** data (inserting, updating, deleting), not just retrieve it. But historically, they're called query languages because retrieval was their primary purpose.

A fundamental property: In the relational model, queries return **relations**. This property is called **closure** or **compositionality**—the result of a query is itself a relation, which means you can use it as input to another query. This enables building complex queries from simpler ones.

11.2.5.1 What is a Query?

At the most basic level, a query expresses an information need:

- **Natural language**: "Show me all bridges built before 1950"
- **Formal language (relational algebra)**: $\sigma_{\text{Year\_Built}<1950}(\text{Bridge})$
- **SQL**: `SELECT * FROM Bridge WHERE Year_Built < 1950`

All three express the same information need, but with different levels of precision and formality.

Queries typically involve:

- **Selection**: Choosing which rows satisfy certain conditions
- **Projection**: Choosing which columns to include in results
- **Combination**: Joining data from multiple tables
- **Aggregation**: Computing summary statistics (count, average, etc.)

The power of declarative query languages is that you specify *what* you want, not *how* to get it. The DBMS figures out an efficient execution strategy.

11.2.5.2 Relational Algebra as a Query Language

Relational algebra is the formal mathematical language for expressing queries on relational databases. It consists of a set of operators that take one or more relations as input and produce a new relation as output.

Relational algebra serves several important purposes:

1. **Formal foundation**: Provides a precise mathematical definition of what queries compute
2. **Query optimization**: DBMSs use relational algebra internally to reason about and optimize queries
3. **Conceptual tool**: Helps us understand what operations are possible and how they compose

The core relational algebra operators are:

- **Select** (σ): Picks certain rows based on a condition
- **Project** (π): Picks certain columns
- **Cross Product** ($\times$): Combines all rows from two tables
- **Natural Join** ($\bowtie$): Combines rows from two tables based on common attributes
- **Union** ($\cup$): Combines rows from two tables (with same schema)
- **Difference** ($-$): Rows in first table but not in second
- **Intersection** ($\cap$): Rows in both tables

- **Rename** (ρ): Changes relation or attribute names

Notice how these connect to the set operations from Module 4! Union, difference, and intersection work exactly as they did with sets. The relational-specific operators (select, project, join) extend set theory for working with structured data.

We'll explore each operator in detail in the next section.

11.2.5.3 The Link to SQL

SQL (Structured Query Language) is the practical implementation of relational algebra used by virtually all commercial database systems. While relational algebra is the mathematical foundation, SQL is what you actually write when working with databases.

Correspondence between relational algebra and SQL:

Relational Algebra	SQL Equivalent
$\sigma_{\text{condition}}(R)$	`SELECT * FROM R WHERE condition`
$\pi_{\text{A,B}}(R)$	`SELECT A, B FROM R`
$R \times S$	`SELECT * FROM R, S` (or `R CROSS JOIN S`)
$R \bowtie S$	`SELECT * FROM R NATURAL JOIN S`
$R \cup S$	`SELECT * FROM R UNION SELECT * FROM S`

Key differences:

1. **Duplicates**: Relational algebra is based on sets (no duplicates). SQL is based on multisets/bags (allows duplicates unless you use `DISTINCT`)
2. **NULL values**: SQL includes NULL for unknown/undefined values; pure relational algebra doesn't
3. **Syntax**: SQL is more verbose but arguably more readable for complex queries
4. **Additional features**: SQL includes many features beyond basic relational algebra (aggregation, grouping, subqueries, etc.)

In this module, we're focusing on **relational algebra** because it provides the clean mathematical foundation. Understanding relational algebra makes learning SQL straightforward—you'll already understand what operations mean, and SQL syntax is just a different notation.

Example: Let's see the same query in both notations.

Query: "Find the names of bridges built before 1950"

Relational Algebra:

$$\pi_{\text{Name}}(\sigma_{\text{Year\_Built}<1950}(\text{Bridge}))$$

SQL:

```
SELECT Name
FROM Bridge
WHERE Year_Built < 1950;
```

Both express: first filter bridges to those built before 1950 (select/WHERE), then extract just the Name column (project/SELECT Name).

11.2.6 Relational Algebra Operators

Now we'll examine each relational algebra operator in detail. For each operator, we'll provide a formal definition, explain what it does intuitively, and show examples using our bridge monitoring system.

Throughout this section, we'll use these sample relations:

Bridge:

Bridge_ID	Name	Year_Built	Length
B001	Roberto Clemente Bridge	1928	244
B002	Fort Pitt Bridge	1959	1201
B003	Fort Duquesne Bridge	1969	1201
B004	Smithfield Street Bridge	1883	361

Sensor:

Sensor_ID	Type	Bridge_ID	Install_Date
S001	accelerometer	B001	2020
S002	strain_gauge	B001	2020
S003	temperature	B002	2019
S004	accelerometer	B003	2021

11.2.6.1 Select (σ)

The **select** operator picks certain **rows** from a relation based on a condition. It filters the relation to include only tuples that satisfy the specified predicate.

Notation: $\sigma_{\text{condition}}(R)$

Definition: $\sigma_{\text{condition}}(R) = \{t \mid t \in R \text{ and condition}(t) \text{ is true}\}$

In words: Select returns all tuples t from relation R where the condition evaluates to true.

Example 1: Find all bridges built before 1950

$$\sigma_{\text{Year\_Built}<1950}(\text{Bridge})$$

Result:

Bridge_ID	Name	Year_Built	Length
B001	Roberto Clemente Bridge	1928	244
B004	Smithfield Street Bridge	1883	361

Example 2: Find all accelerometer sensors

$$\sigma_{\text{Type='accelerometer'}}(\text{Sensor})$$

Result:

Sensor_ID	Type	Bridge_ID	Install_Date
S001	accelerometer	B001	2020
S004	accelerometer	B003	2021

Key points:

- Select operates on rows (horizontal filtering)
- The condition can involve comparisons ($=$, $<$, $>$, $\leq$, $\geq$, $\neq$), logical operators ($\wedge$, $\vee$, $\neg$), and arithmetic
- The result schema is identical to the input schema—only rows are filtered

11.2.6.2 Project (π)

The **project** operator picks certain **columns** from a relation. It extracts a subset of attributes while discarding the others.

Notation: $\pi_{\text{A}_1,\text{A}_2,\ldots,\text{A}_k}(R)$

Definition: $\pi_{\text{A}_1,\ldots,\text{A}_k}(R) = \{(t.\text{A}_1, \ldots, t.\text{A}_k) \mid t \in R\}$

In words: Project returns tuples containing only the specified attributes from R.

Example 1: Get just the names and years of bridges

$$\pi_{\text{Name, Year\_Built}}(\text{Bridge})$$

Result:

Name	Year_Built
Roberto Clemente Bridge	1928
Fort Pitt Bridge	1959
Fort Duquesne Bridge	1969
Smithfield Street Bridge	1883

Example 2: Get sensor types

$$\pi_{\text{Type}}(\text{Sensor})$$

Result:

Type
accelerometer
strain_gauge
temperature

Key points:

- Project operates on columns (vertical filtering)
- **Duplicates are always eliminated** in relational algebra (since results are sets). Notice in Example 2 we only get one "accelerometer" even though there are two accelerometer sensors
- SQL behaves differently: it keeps duplicates unless you use `DISTINCT`. In SQL, `SELECT Type FROM Sensor` would return "accelerometer" twice, while `SELECT DISTINCT Type FROM Sensor` matches the relational algebra behavior
- The result schema contains only the projected attributes

11.2.6.3 Cross Product ($\times$)

The **cross product** (also called **Cartesian product**) combines two relations by pairing each tuple from the first relation with every tuple from the second relation.

Notation: $R \times S$

Definition: $R \times S = \{(r, s) \mid r \in R \text{ and } s \in S\}$

In words: The cross product returns all possible pairs of tuples, one from R and one from S.

Example: Consider smaller relations for clarity:

SmallBridge:

Bridge_ID	Name
B001	Roberto Clemente
B002	Fort Pitt

SmallSensor:

Sensor_ID	Type
S001	accelerometer
S002	strain_gauge

$$\text{SmallBridge} \times \text{SmallSensor}$$

Result:

Bridge_ID	Name	Sensor_ID	Type
B001	Roberto Clemente	S001	accelerometer
B001	Roberto Clemente	S002	strain_gauge
B002	Fort Pitt	S001	accelerometer
B002	Fort Pitt	S002	strain_gauge

Key points:

- If R has m tuples and S has n tuples, then $R \times S$ has $m \times n$ tuples
- Cross product by itself is rarely useful—it generates all possible combinations, including nonsensical ones (e.g., Bridge B001 paired with a sensor actually on Bridge B002)
- It becomes useful when combined with select to filter for meaningful pairings (this is essentially what a join does)
- The result schema includes all attributes from both relations

11.2.6.4 Natural Join ($\bowtie$)

The **natural join** combines two relations by matching tuples that have equal values for all attributes with the same name. It's like a cross product followed by filtering for equality on common attributes, then eliminating duplicate columns.

Notation: $R \bowtie S$

Definition: Natural join automatically:

1. Finds all attributes that appear in both R and S
2. Keeps only tuple pairs where these common attributes have equal values
3. Eliminates one copy of the duplicate attributes

Example: Join Bridge and Sensor on their common attribute Bridge_ID

$$\text{Bridge} \bowtie \text{Sensor}$$

This implicitly enforces `Bridge.Bridge_ID = Sensor.Bridge_ID` and removes the duplicate Bridge_ID column.

Result:

Bridge_ID	Name	Year_Built	Length	Sensor_ID	Type	Install_Date
B001	Roberto Clemente Bridge	1928	244	S001	accelerometer	2020
B001	Roberto Clemente Bridge	1928	244	S002	strain_gauge	2020
B002	Fort Pitt Bridge	1959	1201	S003	temperature	2019
B003	Fort Duquesne Bridge	1969	1201	S004	accelerometer	2021

Notice Bridge B004 doesn't appear (no sensors are installed on it).

Formal definition using other operators:

Natural join can be defined in terms of cross product, select, and project. If R and S share attributes $A_1, \ldots, A_k$:

$$R \bowtie S = \pi_{\text{all attributes, removing duplicates}}(\sigma_{R.A_1=S.A_1 \wedge \cdots \wedge R.A_k=S.A_k}(R \times S))$$

Key points:

- Automatically finds common attributes—no need to specify join condition
- Very convenient when relations are designed with consistent naming
- Eliminates one copy of join attributes (otherwise you'd have Bridge_ID appearing twice)
- Result includes only tuples with matching values—unmatched tuples are excluded

11.2.6.5 Theta Join ($\bowtie_\theta$)

The **theta join** (or just "join") is the most general form of join. It combines tuples from two relations based on an arbitrary condition θ.

Notation: $R \bowtie_\theta S$

Definition: $R \bowtie_\theta S = \sigma_\theta(R \times S)$

In words: Theta join is a cross product followed by a select with condition θ.

Example 1: Find pairs of bridges and sensors where the sensor was installed after the bridge was built

$$\text{Bridge} \bowtie_{\text{Year\_Built}<\text{Install\_Date}} \text{Sensor}$$

This keeps pairs where the bridge's construction year is before the sensor's installation year.

Result:

Bridge_ID	Name	Year_Built	Length	Sensor_ID	Type	Bridge_ID	Install_Date
B001	Roberto Clemente Bridge	1928	244	S001	accelerometer	B001	2020
B001	Roberto Clemente Bridge	1928	244	S002	strain_gauge	B001	2020
B001	Roberto Clemente Bridge	1928	244	S003	temperature	B002	2019
...	...	...	...	...	...	...	...

(Note: This would include all combinations where Year_Built < Install_Date, which includes many nonsensical pairings)

Example 2: A more practical theta join—finding which sensors are on which bridges:

$$\text{Bridge} \bowtie_{\text{Bridge.Bridge\_ID}=\text{Sensor.Bridge\_ID}} \text{Sensor}$$

This is actually equivalent to natural join when the condition is equality on same-named attributes.

Key points:

- Theta join is the **basic operation implemented in DBMSs**—the term "join" in database systems often means theta join
- The condition θ can be any boolean expression involving attributes from both relations
- Natural join is a special case of theta join (where θ is equality on all common attributes)
- Unlike natural join, theta join does **not** eliminate duplicate columns

11.2.6.6 Union ($\cup$)

The **union** operator combines tuples from two relations **vertically**, producing a single relation containing all tuples that appear in either input relation (or both).

Notation: $R \cup S$

Definition: $R \cup S = \{t \mid t \in R \text{ or } t \in S\}$

Requirement: R and S must be **union-compatible**—they must have the same number of attributes, and corresponding attributes must have compatible types. Formally, R and S must have the same schema.

Example: Suppose we have two separate monitoring systems with their own Bridge relations:

PittsburghBridges:

Bridge_ID	Name
B001	Roberto Clemente
B002	Fort Pitt

AlleghenyBridges:

Bridge_ID	Name
B002	Fort Pitt
B003	Fort Duquesne

$$\text{PittsburghBridges} \cup \text{AlleghenyBridges}$$

Result:

Bridge_ID	Name
B001	Roberto Clemente

Bridge_ID	Name
B002	Fort Pitt
B003	Fort Duquesne

Notice Fort Pitt (B002) appears only once—duplicates are eliminated (set property).

Key points:

- Union combines tuples **vertically** (adding rows)
- Both relations must have identical schemas
- Duplicates are eliminated (set-based operation)
- This is the same union operation from set theory in Module 4

11.2.6.7 Difference (−)

The **difference** operator returns tuples that are in the first relation but **not** in the second relation.

Notation: $R - S$

Definition: $R - S = \{t \mid t \in R \text{ and } t \notin S\}$

Requirement: Like union, R and S must be union-compatible (same schema).

Example: Using the same bridge relations:

$$\text{PittsburghBridges} - \text{AlleghenyBridges}$$

Result:

Bridge_ID	Name
B001	Roberto Clemente

Only Roberto Clemente appears because it's in PittsburghBridges but not in Allegheny-Bridges.

Example 2: Find bridges with no sensors

$$\pi_{\text{Bridge\_ID}}(\text{Bridge}) - \pi_{\text{Bridge\_ID}}(\text{Sensor})$$

This gets all bridge IDs from Bridge, then removes those that appear in Sensor, leaving only bridges without sensors.

Result:

Bridge_ID
B004

Key points:

- Difference is **not commutative**: $R - S \neq S - R$ (in general)
- Both relations must have identical schemas
- Useful for finding what's in one relation but not another
- Same as set difference from Module 4

11.2.6.8 Intersection (∩)

The **intersection** operator returns tuples that appear in **both** input relations.

Notation: $R \cap S$

Definition: $R \cap S = \{t \mid t \in R \text{ and } t \in S\}$

Requirement: R and S must be union-compatible (same schema).

Example:

$$\text{PittsburghBridges} \cap \text{AlleghenyBridges}$$

Result:

Bridge_ID	Name
B002	Fort Pitt

Only Fort Pitt appears because it's in both relations.

Key points:

- Intersection finds common elements
- Both relations must have identical schemas
- Can be expressed using difference: $R \cap S = R - (R - S)$
- Same as set intersection from Module 4

11.2.6.9 Rename (ρ)

The **rename** operator changes the name of a relation and/or its attributes. This is useful for disambiguation and for making schemas compatible for set operations.

Notation:

- $\rho_S(R)$ — rename relation R to S
- $\rho_{S(A_1,\ldots,A_n)}(R)$ — rename relation to S and attributes to $A_1, \ldots, A_n$

Use cases:

1. Unifying schemas for set operations

Suppose we have:

OldBridges(ID, BridgeName) and **NewBridges(Identifier, Name)**

They have different schemas, but we want to take their union. We can rename to make them compatible:

$$\rho_{\text{Bridges(Bridge\_ID, Name)}}(\text{OldBridges}) \cup \rho_{\text{Bridges(Bridge\_ID, Name)}}(\text{NewBridges})$$

2. Disambiguation in self-joins

A **self-join** is when a relation is joined with itself. We need to rename one copy to distinguish them.

Example: Find pairs of bridges built in the same year

$$\sigma_{\text{B1.Year\_Built=B2.Year\_Built} \wedge \text{B1.Bridge\_ID<B2.Bridge\_ID}}(\rho_{\text{B1}}(\text{Bridge}) \times \rho_{\text{B2}}(\text{Bridge}))$$

Here we rename Bridge to B1 and B2 so we can refer to them separately.

Alternate notation—Assignment statements:

Instead of deeply nested expressions, we can use assignment statements (a sequence of instructions):

```
B1 := Bridge
B2 := Bridge
SameYear :=  (B1.Year_Built = B2.Year_Built AND B1.Bridge_ID < B2.Bridge_ID)(B1 × B2)
Result :=  (B1.Name, B2.Name, B1.Year_Built)(SameYear)
```

This is more readable for complex queries—each step is named and can be referenced later.

Key points:

- Rename is essential for self-joins (comparing a relation to itself)
- Useful for making schemas compatible for set operations
- Assignment notation provides an algorithmic way to express complex queries

11.2.6.10 Composing Operators for Complex Queries

The real power of relational algebra comes from **composing** operators—building complex queries by combining simpler operations. Because each operator returns a relation, the output of one operation can be the input to another.

Example: "Find the names of bridges that have accelerometer sensors"

We need to:

1. Filter sensors to just accelerometers
2. Join with bridges to get bridge information
3. Project to get just bridge names

Step-by-step:

1. AccelSensors := $\sigma_{\text{Type='accelerometer'}}(\text{Sensor})$
2. BridgesWithAccel := Bridge $\bowtie$ AccelSensors
3. Result := $\pi_{\text{Name}}(\text{BridgesWithAccel})$

As a single expression:

$$\pi_{\text{Name}}(\text{Bridge} \bowtie \sigma_{\text{Type='accelerometer'}}(\text{Sensor}))$$

Result:

Name
Roberto Clemente Bridge
Fort Duquesne Bridge

Example 2: "Find bridges built before 1950 that are longer than 300 meters"

$$\sigma_{\text{Length}>300}(\sigma_{\text{Year\_Built}<1950}(\text{Bridge}))$$

Or equivalently, using a compound condition:

$$\sigma_{\text{Year\_Built}<1950 \wedge \text{Length}>300}(\text{Bridge})$$

Result:

Bridge_ID	Name	Year_Built	Length
B004	Smithfield Street Bridge	1883	361

Key insight: Relational algebra is **compositional**—you can build arbitrarily complex queries by chaining operators together. This is the foundation for how DBMSs process queries: they break down complex SQL queries into sequences of relational algebra operations, then optimize the order and method of execution.

11.2.7 From ER Diagrams to Relational Models

Now we bring everything together: translating the ER diagrams from Module 4 into actual relational schemas that can be implemented in a database. This translation process is mechanical—there are specific rules for converting each ER diagram component into relational tables.

11.2.7.1 Translation Rules: Entity Sets to Relations

The most basic translation rule is straightforward: **each entity set becomes a relation (table)**.

Rule: For each entity set E with attributes $A_1, A_2, \ldots, A_n$:

- Create a relation E with attributes $A_1, A_2, \ldots, A_n$
- The key attribute(s) of the entity set become the primary key of the relation

Example: The Bridge entity set from Module 4:

ER Diagram: Bridge entity with attributes (Bridge_ID, Name, Year_Built, Material, Length), where Bridge_ID is the key.

Relational Schema:

```
Bridge(Bridge_ID, Name, Year_Built, Material, Length)
  PRIMARY KEY: Bridge_ID
```

As a table:

Bridge_ID	Name	Year_Built	Material	Length
B001	Roberto Clemente Bridge	1928	steel	244
B002	Fort Pitt Bridge	1959	steel	1201

Example 2: The Sensor entity set:

ER Diagram: Sensor entity with attributes (Sensor_ID, Type, Install_Date), where Sensor_ID is the key.

Relational Schema:

```
Sensor(Sensor_ID, Type, Install_Date)
  PRIMARY KEY: Sensor_ID
```

Key points:

- Each entity becomes a row in the relation
- Entity attributes become relation attributes
- Key attributes become primary keys
- Multi-valued attributes require special handling (we'll see this shortly)

11.2.7.2 Handling Relationships

Relationships in ER diagrams connect entity sets. How we translate them depends on the **multiplicity** (1:1, 1:n, or n:m).

> **Note**
>
> Notice that in class (and in the slides) I only idscussed the n:m case, recommending that we create a new table for the relationship always. These notes also consider the special cases of 1:1 and 1:n relationships. While you can still create a new table for those relationships, it may be more efficient to avoid that and instead use foreign keys (or merge the tables) in some situations as discussed by Claude here.

11.2.7.2.1 One-to-One Relationships

For a **one-to-one** (1:1) relationship between entity sets E and F:

Option 1: Add a foreign key to one of the relations

- Choose either E or F (typically the one participating fully in the relationship)
- Add F's primary key as a foreign key attribute in E (or vice versa)

Example: Suppose each Bridge has exactly one TaxRecord and each TaxRecord belongs to exactly one Bridge.

ER Diagram: Bridge (1) — Has_Tax_Record — (1) TaxRecord

Relational Schema:

```
Bridge(Bridge_ID, Name, Year_Built, Length)
  PRIMARY KEY: Bridge_ID

TaxRecord(Tax_ID, Assessment_Value, Last_Updated, Bridge_ID)
  PRIMARY KEY: Tax_ID
  FOREIGN KEY: Bridge_ID REFERENCES Bridge(Bridge_ID)
```

We added `Bridge_ID` as a foreign key in TaxRecord.

Option 2: Merge into a single relation

- If the relationship is total (every entity participates), you could combine both entity sets into one table
- This is less common but can simplify queries

11.2.7.2.2 One-to-Many Relationships

For a **one-to-many** (1:n) relationship from E to F (one E relates to many Fs):

Rule: Add the primary key of the "one" side as a foreign key in the "many" side.

Example: One Bridge is monitored by many Sensors (from Module 4).

ER Diagram: Bridge (1) — Monitored_By — (n) Sensor

Relational Schema:

```
Bridge(Bridge_ID, Name, Year_Built, Length)
  PRIMARY KEY: Bridge_ID

Sensor(Sensor_ID, Type, Install_Date, Bridge_ID)
  PRIMARY KEY: Sensor_ID
  FOREIGN KEY: Bridge_ID REFERENCES Bridge(Bridge_ID)
```

Each sensor has a `Bridge_ID` foreign key indicating which bridge it's on.

As tables:

Bridge:

Bridge_ID	Name	Year_Built	Length
B001	Roberto Clemente Bridge	1928	244
B002	Fort Pitt Bridge	1959	1201

Sensor:

Sensor_ID	Type	Install_Date	Bridge_ID
S001	accelerometer	2020	B001
S002	strain_gauge	2020	B001
S003	temperature	2019	B002

Notice multiple sensors can have the same Bridge_ID (many-to-one from Sensor's perspective).

11.2.7.2.3 Many-to-Many Relationships

For a **many-to-many** (n:m) relationship between entity sets E and F:

Rule: Create a new relation (called a **junction table** or **linking table**) with:

- Primary keys from both E and F as foreign keys
- The combination of these foreign keys forms the primary key
- Any attributes of the relationship itself

Example: Engineers inspect Bridges, and each engineer can inspect multiple bridges, while each bridge can be inspected by multiple engineers.

ER Diagram: Engineer (n) — Inspects — (m) Bridge

The relationship Inspects has attributes: Inspection_Date, Condition_Rating

Relational Schema:

```
Engineer(Employee_ID, Name, Specialty)
  PRIMARY KEY: Employee_ID

Bridge(Bridge_ID, Name, Year_Built, Length)
  PRIMARY KEY: Bridge_ID

Inspects(Employee_ID, Bridge_ID, Inspection_Date, Condition_Rating)
  PRIMARY KEY: (Employee_ID, Bridge_ID, Inspection_Date)
  FOREIGN KEY: Employee_ID REFERENCES Engineer(Employee_ID)
  FOREIGN KEY: Bridge_ID REFERENCES Bridge(Bridge_ID)
```

As tables:

Inspects:

Employee_ID	Bridge_ID	Inspection_Date	Condition_Rating
E001	B001	2024-01-15	Good
E001	B002	2024-01-20	Fair
E002	B001	2024-02-10	Good

This captures that Engineer E001 inspected both B001 and B002, and Bridge B001 was inspected by both E001 and E002.

Key insight: Many-to-many relationships **cannot** be captured with just foreign keys in the entity tables—you must create a junction table.

11.2.7.3 Weak Entity Sets in Relational Models

Weak entity sets depend on another entity set for their identity. When translating to relations:

Rule:

- Create a relation for the weak entity set
- Include all its own attributes
- Include the primary key of the identifying (owner) entity set as a foreign key
- The primary key of the weak entity is a **composite key**: its own partial key plus the owner's primary key

Example: Sensor Readings depend on Sensors (from Module 4).

ER Diagram: Sensor (strong) — Generates — (weak) Reading

Reading has partial key `Sequence_Number` (unique within each sensor, but not globally).

Relational Schema:

```
Sensor(Sensor_ID, Type, Bridge_ID)
  PRIMARY KEY: Sensor_ID

Reading(Sensor_ID, Sequence_Number, Timestamp, Value)
  PRIMARY KEY: (Sensor_ID, Sequence_Number)
  FOREIGN KEY: Sensor_ID REFERENCES Sensor(Sensor_ID)
```

As a table:

Reading:

Sensor_ID	Sequence_Number	Timestamp	Value
S001	1	2024-01-01 08:00	0.05
S001	2	2024-01-01 08:10	0.06
S002	1	2024-01-01 08:00	120.5
S002	2	2024-01-01 08:10	121.0

Notice Sequence_Number alone doesn't uniquely identify a reading—you need both Sensor_ID and Sequence_Number.

11.2.7.4 ISA Hierarchies and Inheritance

ISA hierarchies (subclasses) have multiple translation approaches. The choice depends on your query patterns and storage efficiency needs.

Approach 1: Single Table (ER-Style) - Create one table for the superclass with all possible attributes - Add a `Type` column to distinguish subclasses - Use NULL for attributes that don't apply to a given subclass

Approach 2: Table Per Subclass - Create separate tables for each subclass - Each subclass table contains all attributes (inherited + specific) - No table for the superclass

Approach 3: Table Per Hierarchy (Hybrid) - Create one table for the superclass with shared attributes - Create separate tables for each subclass with subclass-specific attributes only - Subclass tables reference the superclass via foreign key

Example: Bridge has subclasses Suspension_Bridge, Arch_Bridge, Truss_Bridge (from Module 4).

ER Diagram:

- Bridge (superclass) with attributes (Bridge_ID, Name, Year_Built, Length)
- Suspension_Bridge with specific attributes (Cable_Diameter, Tower_Height)
- Arch_Bridge with specific attributes (Arch_Span, Rise)

Approach 1 - Single Table:

```
Bridge(Bridge_ID, Name, Year_Built, Length, Bridge_Type,
       Cable_Diameter, Tower_Height, Arch_Span, Rise)
  PRIMARY KEY: Bridge_ID
```

Bridge_ID	Name	Year_Built	Length	Bridge_Type	Cable_Diameter	Tower_Height	Arch_Span	Rise
B001	Roberto Clemente	1928	244	suspension	0.5	45	NULL	NULL
B002	Fort Pitt	1959	1201	arch	NULL	NULL	300	60

Approach 3 - Table Per Hierarchy:

```
Bridge(Bridge_ID, Name, Year_Built, Length)
  PRIMARY KEY: Bridge_ID

Suspension_Bridge(Bridge_ID, Cable_Diameter, Tower_Height)
  PRIMARY KEY: Bridge_ID
  FOREIGN KEY: Bridge_ID REFERENCES Bridge(Bridge_ID)

Arch_Bridge(Bridge_ID, Arch_Span, Rise)
  PRIMARY KEY: Bridge_ID
  FOREIGN KEY: Bridge_ID REFERENCES Bridge(Bridge_ID)
```

Tradeoffs:

- **Single Table**: Simple queries for common attributes, but many NULLs and wasted space
- **Table Per Subclass**: No NULLs, but queries for common attributes require UNIONs
- **Table Per Hierarchy**: Balanced approach, but requires JOINs to get complete entity

11.2.7.5 Complete Example: Bridge Monitoring System

Let's translate the complete ER diagram from Module 4:

Entity Sets:

- Bridge (Bridge_ID, Name, Location, Year_Built, Materials)
- Engineer (Employee_ID, Name, Specialty, Hire_Date)
- Sensor (Sensor_ID, Type, Install_Date)
- Manufacturer (Manufacturer_ID, Company_Name)

Relationships:

- Bridge (1) — Monitored_By — (n) Sensor
- Sensor (n) — Produced_By — (1) Manufacturer

- Engineer (n) — Inspects — (m) Bridge (with attributes Inspection_Date, Condition_Rating)
- Engineer (n) — Supervises — (n) Engineer (self-relationship)

Weak Entity:

- Reading (depends on Sensor) with partial key Seq_Number

Complete Relational Schema:

```
-- Entity sets become tables
Bridge(Bridge_ID, Name, Location, Year_Built, Materials, Length)
  PRIMARY KEY: Bridge_ID

Engineer(Employee_ID, Name, Specialty, Hire_Date)
  PRIMARY KEY: Employee_ID

Sensor(Sensor_ID, Type, Install_Date, Bridge_ID, Manufacturer_ID)
  PRIMARY KEY: Sensor_ID
  FOREIGN KEY: Bridge_ID REFERENCES Bridge(Bridge_ID)
  FOREIGN KEY: Manufacturer_ID REFERENCES Manufacturer(Manufacturer_ID)

Manufacturer(Manufacturer_ID, Company_Name)
  PRIMARY KEY: Manufacturer_ID

-- Weak entity
Reading(Sensor_ID, Seq_Number, Timestamp, Value)
  PRIMARY KEY: (Sensor_ID, Seq_Number)
  FOREIGN KEY: Sensor_ID REFERENCES Sensor(Sensor_ID)

-- Many-to-many relationships become junction tables
Inspects(Employee_ID, Bridge_ID, Inspection_Date, Condition_Rating)
  PRIMARY KEY: (Employee_ID, Bridge_ID, Inspection_Date)
  FOREIGN KEY: Employee_ID REFERENCES Engineer(Employee_ID)
  FOREIGN KEY: Bridge_ID REFERENCES Bridge(Bridge_ID)

Supervises(Supervisor_ID, Supervisee_ID)
  PRIMARY KEY: (Supervisor_ID, Supervisee_ID)
  FOREIGN KEY: Supervisor_ID REFERENCES Engineer(Employee_ID)
  FOREIGN KEY: Supervisee_ID REFERENCES Engineer(Employee_ID)
```

Key observations:

1. Each entity set became a table

2. One-to-many relationships (Bridge-Sensor, Manufacturer-Sensor) are captured by foreign keys in the "many" side
3. Many-to-many relationships (Engineer-Bridge Inspects, Engineer-Engineer Supervises) required junction tables
4. The weak entity Reading has a composite primary key
5. Self-relationships like Supervises are handled like any other many-to-many relationship

This relational schema can now be implemented in any relational DBMS using DDL (as we saw in the Database Creation Process section), populated with data, and queried using relational algebra or SQL.

12 Module 6: The Structured Query Language (SQL)

12.1 Module Overview

12.1.1 Learning Objectives

By the end of this module, students will be able to:

- Understand what SQL is and where it comes from, as well as its prevalence in industry and computing infrastructure today
- Distinguish between the Data Description Language (DDL) and Data Manipulation Language (DML) groups of statements
- Formulate queries using the following SQL statements, functions and clauses, as well as understand their relational algebra counterparts (where applicable):
  - `SELECT`, `SELECT ... WHERE`, `SELECT ... GROUP BY`, `SELECT ... JOIN`
  - Nested selects (`SELECT ... SELECT`)
  - Aggregate functions: `SUM()`, `MAX()`, `AVG()`, `COUNT()`
  - DDL statements: `CREATE TABLE`, `ALTER`, `DROP`
  - DML statements: `INSERT`, `DELETE`, `UPDATE`
- Install and issue queries to a local DBMS such as MySQL (with MySQL Workbench as a client), or SQLite with its CLI interface
- Translate relational algebra expressions to equivalent SQL queries and vice versa
- Write relatively more complex SQL queries involving joins, aggregations, and subqueries

12.1.2 Topics Covered

- **Introduction to SQL**
  - Historical context and development
  - SQL's role in modern computing infrastructure
  - SQL standards and variations across database systems
  - The relationship between SQL and relational algebra
- **SQL Statement Categories**

  - Data Definition Language (DDL): Defining database structure
  - Data Manipulation Language (DML): Querying and modifying data

- **Basic SQL Queries**
  - `SELECT` statement structure
  - Filtering with `WHERE` clauses
  - Sorting results with `ORDER BY`
  - Eliminating duplicates with `DISTINCT`
  - Limiting results

- **SQL Operators and Functions**
  - Comparison operators (=, <, >, <=, >=, <>)
  - Logical operators (AND, OR, NOT)
  - Pattern matching with `LIKE`

- **Aggregate Functions and Grouping**
  - `COUNT()`, `SUM()`, `AVG()`, `MIN()`, `MAX()`
  - `GROUP BY` clause
  - Understanding the order of SQL clause execution

- **Joins in SQL**
  - Cross joins (Cartesian product)
  - Inner joins (theta joins and natural joins)
  - Outer joins (LEFT, RIGHT, FULL)
  - Self-joins

- **Nested Queries (Subqueries)**
  - Subqueries in WHERE clauses
  - Subqueries in FROM clauses (derived tables)

- **Data Definition with DDL**
  - `CREATE TABLE` with constraints
  - Data types and domains
  - Primary keys and foreign keys
  - `ALTER TABLE` to modify structure
  - `DROP TABLE` to delete tables

- **Data Modification with DML**
  - `INSERT` statements (single row and multiple rows)
  - `UPDATE` with conditions
  - `DELETE` with conditions
  - Transaction concepts (brief introduction)

- **Working with Local Database Systems**
  - Installing and configuring MySQL or SQLite
  - Using database clients (MySQL Workbench, SQLite CLI)

12.1.3 Project Milestones

Submit a set of 5 slides for your group's final project proposal presentation:

1. Title of the project, and its members
2. Motivation for your project
3. Specific database problem you are aiming to solve and expected goals
4. Proposed solution (and how it draws from concepts learned in this course)
5. Remaining questions for your team about how to complete the project

12.2 Lecture Notes

12.2.1 From Relational Algebra to SQL

In Module 5, we studied the relational model and relational algebra—the mathematical foundation for working with relational databases. We learned operators like select (σ), project (π), join ($\bowtie$), union ($\cup$), and others that allow us to express queries precisely.

While relational algebra provides the theoretical framework, it's not a practical language for working with real database systems. This is where **SQL (Structured Query Language)** comes in.

SQL is the practical implementation of relational algebra. Nearly every concept from Module 5's relational algebra has a direct counterpart in SQL:

Relational Algebra	SQL Equivalent
$\sigma_{\text{condition}}(R)$	`SELECT * FROM R WHERE condition`
$\pi_{\text{A,B}}(R)$	`SELECT A, B FROM R`
$R \times S$	`SELECT * FROM R CROSS JOIN S`
$R \bowtie S$	`SELECT * FROM R NATURAL JOIN S`
$R \cup S$	`SELECT * FROM R UNION SELECT * FROM S`

However, SQL extends beyond pure relational algebra with additional features:

- **Aggregate functions** (COUNT, SUM, AVG, MAX, MIN) for computing statistics
- **Grouping operations** (GROUP BY) for organizing data

- **NULL values** for representing missing or unknown data
- **Sorting** (ORDER BY) for controlling output order
- **Data modification** statements (INSERT, UPDATE, DELETE)
- **Schema definition** statements (CREATE TABLE, ALTER TABLE, DROP TABLE)

SQL also differs from relational algebra in one important way: while relational algebra treats relations as **sets** (no duplicates), SQL treats tables as **multisets** or **bags** (duplicates allowed by default). You must explicitly use `DISTINCT` to eliminate duplicates.

Throughout this module, we'll see how SQL translates the abstract concepts from Module 5 into executable commands that work with real database systems.

12.2.2 Introduction to SQL

12.2.2.1 What is SQL?

SQL (Structured Query Language) is a standardized programming language for managing and querying relational databases. It provides commands to:

- Define database structures (tables, constraints, relationships)
- Query data (retrieve information from tables)
- Modify data (insert, update, delete records)
- Control access (grant and revoke permissions)

SQL is both powerful and relatively simple to learn. Unlike procedural programming languages where you specify *how* to accomplish a task, SQL is **declarative**—you specify *what* you want, and the database management system (DBMS) figures out how to execute it efficiently.

12.2.2.2 Historical Context

SQL was developed in the early 1970s at IBM by Donald Chamberlin and Raymond Boyce. It was originally called **SEQUEL** (Structured English Query Language) and was designed for IBM's System R database, one of the first implementations of Edgar Codd's relational model.

The language was later renamed SQL, and in 1986, it became an ANSI (American National Standards Institute) standard. Major revisions followed:

- **SQL-86** (1986): First standard
- **SQL-92** (1992): Major expansion, widely implemented
- **SQL:1999** (1999): Added triggers, recursive queries
- **SQL:2003** (2003): Added window functions, XML support
- **SQL:2016** (2016): JSON support, pattern matching

Despite standardization, different database vendors have implemented SQL with variations. The core features remain consistent, but advanced features and syntax details can differ between systems like MySQL, PostgreSQL, Oracle, SQL Server, and SQLite.

12.2.2.3 SQL's Role in Modern Computing

SQL is ubiquitous in modern computing infrastructure. It's used in:

- **Web applications**: Nearly all web apps use SQL databases for data persistence
- **Business intelligence**: Data warehouses and analytics platforms rely on SQL
- **Financial systems**: Banks and financial institutions use SQL extensively
- **Scientific computing**: Research data management and analysis
- **Mobile apps**: SQLite is embedded in iOS and Android applications
- **Government systems**: Census data, tax systems, public records

According to various industry surveys, SQL consistently ranks as one of the most in-demand skills for data-related jobs. Learning SQL provides a foundation for working with structured data across virtually any domain.

12.2.2.4 SQL and Database Management Systems

SQL is a language standard, but you execute SQL commands through a **Database Management System (DBMS)**—software that manages the storage, retrieval, and manipulation of data. Common SQL-based DBMS include:

- **MySQL**: Open-source, widely used for web applications
- **PostgreSQL**: Open-source, advanced features, ACID compliant
- **SQLite**: Lightweight, serverless, embedded in applications
- **Oracle Database**: Enterprise-grade, commercial
- **Microsoft SQL Server**: Enterprise-grade, Windows-focused
- **MariaDB**: Open-source MySQL fork

For this course, we'll focus on MySQL and SQLite as they're freely available and well-suited for learning.

12.2.3 SQL Statement Categories: DDL vs DML

SQL statements fall into several categories based on their purpose. The two most important for this course are:

12.2.3.1 Data Definition Language (DDL)

DDL statements define and modify the structure of databases—they create, alter, and delete tables and other database objects.

Key DDL statements:

- `CREATE TABLE`: Define a new table with columns and constraints
- `ALTER TABLE`: Modify an existing table structure (add/remove columns, change constraints)
- `DROP TABLE`: Delete a table and all its data

Example:

```
CREATE TABLE Bridge (
    Bridge_ID VARCHAR(10) PRIMARY KEY,
    Name VARCHAR(100),
    Year_Built INTEGER,
    Length REAL
);
```

DDL statements affect the **schema** (structure) of the database, not the data itself.

12.2.3.2 Data Manipulation Language (DML)

DML statements work with the data stored in tables—they query, insert, update, and delete records.

Key DML statements:

- `SELECT`: Query data from tables (the most frequently used SQL statement)
- `INSERT`: Add new records to a table
- `UPDATE`: Modify existing records
- `DELETE`: Remove records from a table

Example:

```
SELECT Name, Year_Built
FROM Bridge
WHERE Year_Built < 1950;
```

DML statements affect the **instance** (data) of the database, not the structure.

12.2.3.3 Other SQL Categories (Brief Overview)

For completeness, two other categories exist but are beyond this course's scope:

- **Data Control Language (DCL)**: Manages permissions and access control (`GRANT`, `REVOKE`)
- **Transaction Control Language (TCL)**: Manages transactions (`BEGIN`, `COMMIT`, `ROLLBACK`)

We'll briefly mention transactions in the DML section, but the focus of this module is on DDL and DML.

12.2.4 Basic SQL Queries

The `SELECT` statement is the foundation of SQL queries. It retrieves data from one or more tables based on specified criteria.

12.2.4.1 Basic SELECT Structure

The simplest form of a SELECT query has this structure:

```
SELECT column1, column2, ...
FROM table_name;
```

Example: Retrieve all bridge names and years built:

```
SELECT Name, Year_Built
FROM Bridge;
```

To retrieve all columns, use the asterisk (*):

```
SELECT *
FROM Bridge;
```

This corresponds to the relational algebra expression: Bridge (selecting all attributes).

12.2.4.2 Filtering Rows with WHERE

The `WHERE` clause filters rows based on a condition. This corresponds to the relational algebra **select** operator (σ).

```
SELECT column1, column2, ...
FROM table_name
WHERE condition;
```

Example: Find bridges built before 1950:

```
SELECT Name, Year_Built
FROM Bridge
WHERE Year_Built < 1950;
```

This translates to: $\pi_{\text{Name, Year\_Built}}(\sigma_{\text{Year\_Built}<1950}(\text{Bridge}))$

12.2.4.3 Sorting Results with ORDER BY

The `ORDER BY` clause sorts the result set by one or more columns. Sorting doesn't exist in pure relational algebra (since relations are unordered sets), but it's essential for presenting results.

```
SELECT column1, column2, ...
FROM table_name
ORDER BY column1 [ASC|DESC], column2 [ASC|DESC], ...;
```

- **ASC**: Ascending order (default)
- **DESC**: Descending order

Example: List bridges by year built, oldest first:

```
SELECT Name, Year_Built
FROM Bridge
ORDER BY Year_Built ASC;
```

Example: List bridges by length, longest first:

```
SELECT Name, Length
FROM Bridge
ORDER BY Length DESC;
```

12.2.4.4 Eliminating Duplicates with DISTINCT

The `DISTINCT` keyword eliminates duplicate rows from results. This makes SQL behave like relational algebra (which treats relations as sets).

```
SELECT DISTINCT column1, column2, ...
FROM table_name;
```

Example: Find all unique sensor types:

```
SELECT DISTINCT Type
FROM Sensor;
```

Without `DISTINCT`, if multiple sensors have the same type, that type would appear multiple times in the result.

12.2.4.5 Limiting Results

Many SQL systems provide a way to limit the number of rows returned. The syntax varies by DBMS:

SQLite and MySQL:

```
SELECT column1, column2, ...
FROM table_name
LIMIT n;
```

Example: Get the 5 oldest bridges:

```
SELECT Name, Year_Built
FROM Bridge
ORDER BY Year_Built ASC
LIMIT 5;
```

Note: Different database systems use different syntax for this feature (`TOP` in SQL Server, `FETCH FIRST` in standard SQL).

12.2.4.6 Combining Clauses

You can combine these clauses in a single query. SQL processes them in a specific logical order:

1. **FROM**: Identify the table
2. **WHERE**: Filter rows
3. **SELECT**: Choose columns
4. **DISTINCT**: Eliminate duplicates
5. **ORDER BY**: Sort results
6. **LIMIT**: Restrict number of rows

Example: Find accelerometer sensors, show their IDs and types, sorted by ID:

```
SELECT DISTINCT Sensor_ID, Type
FROM Sensor
WHERE Type = 'accelerometer'
ORDER BY Sensor_ID
LIMIT 10;
```

12.2.5 SQL Operators and Conditions

SQL provides various operators for constructing conditions in `WHERE` clauses. These allow you to precisely filter data based on complex criteria.

12.2.5.1 Comparison Operators

Comparison operators test relationships between values:

Operator	Meaning	Example
`=`	Equal to	`Year_Built = 1950`
`<>` or `!=`	Not equal to	`Type <> 'accelerometer'`
`<`	Less than	`Length < 300`
`>`	Greater than	`Year_Built > 2000`
`<=`	Less than or equal	`Length <= 500`
`>=`	Greater than or equal	`Year_Built >= 1900`

Example: Find bridges longer than 1000 meters:

```sql
SELECT Name, Length
FROM Bridge
WHERE Length > 1000;
```

12.2.5.2 Logical Operators

Logical operators combine multiple conditions:

- **AND**: Both conditions must be true
- **OR**: At least one condition must be true
- **NOT**: Negates a condition

Example: Find bridges built between 1900 and 1950:

```sql
SELECT Name, Year_Built
FROM Bridge
WHERE Year_Built >= 1900 AND Year_Built <= 1950;
```

Example: Find sensors that are either accelerometers or strain gauges:

```sql
SELECT Sensor_ID, Type
FROM Sensor
WHERE Type = 'accelerometer' OR Type = 'strain_gauge';
```

Example: Find bridges not built in the 20th century:

```sql
SELECT Name, Year_Built
FROM Bridge
WHERE NOT (Year_Built >= 1900 AND Year_Built < 2000);
```

12.2.5.3 Pattern Matching with LIKE

The `LIKE` operator performs pattern matching on text. Two wildcards are available:

- **%**: Matches any sequence of characters (including empty)
- **_**: Matches exactly one character

Example: Find bridges with “Bridge” in their name:

```sql
SELECT Name
FROM Bridge
WHERE Name LIKE '%Bridge%';
```

Example: Find sensors with IDs starting with 'S00':

```sql
SELECT Sensor_ID, Type
FROM Sensor
WHERE Sensor_ID LIKE 'S00_';
```

Note: Pattern matching is case-insensitive in some database systems (like MySQL by default) but case-sensitive in others (like PostgreSQL).

12.2.5.4 NULL Values

SQL uses `NULL` to represent missing or unknown data. Testing for NULL requires special operators:

- `IS NULL`: Tests if a value is NULL
- `IS NOT NULL`: Tests if a value is not NULL

Example: Find sensors without an installation date:

```sql
SELECT Sensor_ID, Type
FROM Sensor
WHERE Install_Date IS NULL;
```

Important: You cannot use `= NULL` or `<> NULL`. The expressions `value = NULL` and `value <> NULL` always evaluate to unknown (not true or false), so they won't work as intended.

12.2.5.5 Operator Precedence

When combining operators, SQL follows precedence rules (highest to lowest):

1. Arithmetic operators (`*`, `/`, `+`, `-`)
2. Comparison operators (`=`, `<`, `>`, etc.)
3. `NOT`
4. `AND`
5. `OR`

Use parentheses to override precedence and make expressions clearer:

```
SELECT Name
FROM Bridge
WHERE (Year_Built < 1900 OR Year_Built > 2000) AND Length > 500;
```

12.2.6 Aggregate Functions and Grouping

Aggregate functions compute summary statistics across multiple rows. They extend beyond pure relational algebra, providing powerful analytical capabilities.

12.2.6.1 Common Aggregate Functions

SQL provides several built-in aggregate functions:

Function	Purpose	Example
`COUNT(*)`	Count all rows	`COUNT(*)`
`COUNT(column)`	Count non-NULL values in column	`COUNT(Sensor_ID)`
`SUM(column)`	Sum of values	`SUM(Length)`
`AVG(column)`	Average of values	`AVG(Year_Built)`
`MIN(column)`	Minimum value	`MIN(Year_Built)`
`MAX(column)`	Maximum value	`MAX(Length)`

Example: Count the total number of bridges:

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

Example: Find the average bridge length:

```
SELECT AVG(Length)
FROM Bridge;
```

Example: Find the oldest bridge construction year:

```
SELECT MIN(Year_Built)
FROM Bridge;
```

12.2.6.2 Grouping with GROUP BY

The `GROUP BY` clause partitions rows into groups based on column values, then applies aggregate functions to each group separately.

```
SELECT column1, aggregate_function(column2)
FROM table_name
GROUP BY column1;
```

Example: Count how many sensors are on each bridge:

```
SELECT Bridge_ID, COUNT(*) AS Sensor_Count
FROM Sensor
GROUP BY Bridge_ID;
```

This groups sensors by Bridge_ID, then counts how many sensors are in each group.

Example: Find the average length of bridges by decade:

```
SELECT (Year_Built / 10) * 10 AS Decade, AVG(Length) AS Avg_Length
FROM Bridge
GROUP BY (Year_Built / 10) * 10;
```

12.2.6.3 Rules for GROUP BY

When using `GROUP BY`:

1. **SELECT clause** can only include:
    - Columns listed in `GROUP BY`
    - Aggregate functions
    - Constants
2. You **cannot** select non-grouped, non-aggregated columns

Invalid example:

```
-- This will error: Name is not in GROUP BY
SELECT Bridge_ID, Name, COUNT(*)
FROM Sensor
GROUP BY Bridge_ID;
```

Valid version:

```sql
SELECT Bridge_ID, COUNT(*)
FROM Sensor
GROUP BY Bridge_ID;
```

12.2.6.4 Understanding SQL Clause Execution Order

SQL processes clauses in a specific logical order (not the order you write them):

1. **FROM**: Identify tables
2. **WHERE**: Filter rows
3. **GROUP BY**: Group rows
4. **HAVING**: Filter groups
5. **SELECT**: Choose columns and apply functions
6. **ORDER BY**: Sort results
7. **LIMIT**: Restrict number of rows

This order explains why:

- `WHERE` filters individual rows *before* grouping
- You can't use aggregate functions in `WHERE` (aggregates happen after WHERE)
- `HAVING` filters groups *after* `GROUP BY` (so you can use aggregates there)

Example: Find bridges with multiple sensors (at least 2):

```sql
SELECT Bridge_ID, COUNT(*) AS Sensor_Count
FROM Sensor
GROUP BY Bridge_ID
HAVING COUNT(*) >= 2;
```

The `HAVING` clause filters groups *after* grouping, keeping only groups where the count is at least 2.

Example: Find the average length of bridges built after 1900, grouped by decade, showing only decades with average length > 500:

```sql
SELECT (Year_Built / 10) * 10 AS Decade, AVG(Length) AS Avg_Length
FROM Bridge
WHERE Year_Built > 1900
GROUP BY (Year_Built / 10) * 10
HAVING AVG(Length) > 500
ORDER BY Decade;
```

Execution order: 1. FROM Bridge 2. WHERE Year_Built > 1900 (filter individual bridges) 3. GROUP BY decade (group remaining bridges) 4. HAVING AVG(Length) > 500 (filter groups) 5. SELECT Decade, AVG(Length) (compute results) 6. ORDER BY Decade (sort output)

12.2.7 Joins in SQL

Joins combine data from multiple tables. They translate directly from Module 5's relational algebra join operators.

12.2.7.1 Cross Join (Cartesian Product)

A **cross join** produces all possible combinations of rows from two tables. This corresponds to the relational algebra cross product ($R \times S$).

```
SELECT *
FROM Table1 CROSS JOIN Table2;
```

Alternative syntax (comma notation):

```
SELECT *
FROM Table1, Table2;
```

Example: Combine all bridges with all sensor types (not useful, but demonstrates the concept):

```
SELECT Bridge.Name, Sensor.Type
FROM Bridge CROSS JOIN Sensor;
```

Cross joins by themselves are rarely useful because they produce many meaningless combinations. They become useful when combined with WHERE conditions (which creates a theta join).

12.2.7.2 Inner Join (Theta Join)

An **inner join** combines rows from two tables based on a condition. This corresponds to the relational algebra theta join ($R \bowtie_\theta S$).

Syntax:

```
SELECT columns
FROM Table1
INNER JOIN Table2 ON condition;
```

Example: Find which sensors are on which bridges:

```
SELECT Bridge.Name, Sensor.Sensor_ID, Sensor.Type
FROM Bridge
INNER JOIN Sensor ON Bridge.Bridge_ID = Sensor.Bridge_ID;
```

This joins Bridge and Sensor tables where the Bridge_ID values match, giving us sensor information along with the bridge name.

Note: The `INNER` keyword is optional—`JOIN` alone means inner join.

12.2.7.3 Natural Join

A **natural join** automatically joins tables on columns with the same name. This corresponds to relational algebra's natural join ($R \bowtie S$).

```
SELECT *
FROM Bridge NATURAL JOIN Sensor;
```

This automatically joins on `Bridge_ID` (the common column name).

Caution: Natural joins can be dangerous if tables have multiple columns with the same name—the join will use all of them, which may not be what you want. Explicit `JOIN...ON` is generally preferred for clarity.

12.2.7.4 Outer Joins

Outer joins include rows that don't have matching values in the other table, filling in NULL for missing data.

LEFT OUTER JOIN: Include all rows from the left table, even if there's no match in the right table.

```
SELECT Bridge.Name, Sensor.Sensor_ID
FROM Bridge
LEFT JOIN Sensor ON Bridge.Bridge_ID = Sensor.Bridge_ID;
```

This returns all bridges, even those without sensors (Sensor_ID will be NULL for bridges with no sensors).

RIGHT OUTER JOIN: Include all rows from the right table, even if there's no match in the left table.

```
SELECT Bridge.Name, Sensor.Sensor_ID
FROM Bridge
RIGHT JOIN Sensor ON Bridge.Bridge_ID = Sensor.Bridge_ID;
```

This returns all sensors, even if their Bridge_ID doesn't exist in the Bridge table.

FULL OUTER JOIN: Include all rows from both tables, filling with NULL where there's no match.

```
SELECT Bridge.Name, Sensor.Sensor_ID
FROM Bridge
FULL OUTER JOIN Sensor ON Bridge.Bridge_ID = Sensor.Bridge_ID;
```

Note: SQLite doesn't support FULL OUTER JOIN directly. You can simulate it using UNION of LEFT and RIGHT joins.

12.2.7.5 Self-Joins

A **self-join** joins a table to itself. This requires aliasing the table with different names.

Example: If engineers can supervise other engineers, find pairs of supervisor-supervisee:

```
SELECT E1.Name AS Supervisor, E2.Name AS Supervisee
FROM Engineer E1
INNER JOIN Engineer E2 ON E1.Employee_ID = E2.Supervisor_ID;
```

Here we alias Engineer as E1 and E2 to distinguish the two copies.

12.2.7.6 Joining Multiple Tables

You can chain multiple joins to combine data from three or more tables.

Example: Find sensor readings along with bridge names:

```sql
SELECT Bridge.Name, Sensor.Type, Reading.Value, Reading.Timestamp
FROM Bridge
INNER JOIN Sensor ON Bridge.Bridge_ID = Sensor.Bridge_ID
INNER JOIN Reading ON Sensor.Sensor_ID = Reading.Sensor_ID;
```

This joins three tables: Bridge → Sensor → Reading, connecting bridges to their sensors to the readings from those sensors.

12.2.7.7 Connection to Relational Algebra

SQL Join	Relational Algebra	Meaning
`CROSS JOIN`	$R \times S$	All combinations
`JOIN...ON`	$R \bowtie_\theta S$	Theta join with condition
`NATURAL JOIN`	$R \bowtie S$	Natural join on common attributes
`LEFT JOIN`	No direct equivalent	Include unmatched left rows
`RIGHT JOIN`	No direct equivalent	Include unmatched right rows

Outer joins extend beyond pure relational algebra, providing additional flexibility for handling incomplete data.

12.2.8 Nested Queries and Subqueries

A **subquery** (or **nested query**) is a SELECT statement embedded within another SQL statement. Subqueries allow you to break complex queries into logical steps.

12.2.8.1 Subqueries in WHERE Clauses

The most common use of subqueries is in WHERE clauses to filter based on results from another query.

Example: Find bridges that have at least one sensor installed:

```sql
SELECT Name
FROM Bridge
WHERE Bridge_ID IN (SELECT Bridge_ID FROM Sensor);
```

The inner query `SELECT Bridge_ID FROM Sensor` returns all bridge IDs that appear in the Sensor table. The outer query then selects bridges whose ID is in that list.

Example: Find the oldest bridge:

```
SELECT Name, Year_Built
FROM Bridge
WHERE Year_Built = (SELECT MIN(Year_Built) FROM Bridge);
```

The subquery finds the minimum year, then the outer query selects bridges built in that year.

12.2.8.2 Subqueries in FROM Clauses

You can use a subquery in the FROM clause, treating its result as a temporary table (called a **derived table**).

Example: Find bridges with above-average length:

```
SELECT B.Name, B.Length
FROM Bridge B, (SELECT AVG(Length) AS AvgLen FROM Bridge) AS Avg
WHERE B.Length > Avg.AvgLen;
```

The subquery computes the average length, then the outer query compares each bridge's length to that average.

Alternative using WHERE subquery:

```
SELECT Name, Length
FROM Bridge
WHERE Length > (SELECT AVG(Length) FROM Bridge);
```

Both approaches work; choose based on readability and performance.

12.2.8.3 Subquery Operators

SQL provides special operators for working with subqueries:

IN: Check if a value appears in the subquery results

```
SELECT Name
FROM Bridge
WHERE Bridge_ID IN (SELECT Bridge_ID FROM Sensor WHERE Type = 'accelerometer');
```

Find bridges that have accelerometer sensors.

EXISTS: Check if the subquery returns any rows

```
SELECT Name
FROM Bridge B
WHERE EXISTS (SELECT * FROM Sensor S WHERE S.Bridge_ID = B.Bridge_ID);
```

Find bridges that have at least one sensor. EXISTS is often more efficient than IN for large datasets.

NOT IN / NOT EXISTS: Negate the above

```
SELECT Name
FROM Bridge
WHERE Bridge_ID NOT IN (SELECT Bridge_ID FROM Sensor);
```

Find bridges without any sensors.

12.2.8.4 When to Use Joins vs. Subqueries

Many queries can be written with either joins or subqueries. Guidelines:

- **Use joins** when you need columns from multiple tables in the result
- **Use subqueries** when you only need data from one table but need to filter based on another
- **Use EXISTS** for checking existence (often faster than IN for large datasets)

Example with JOIN:

```
-- Using join: need columns from both tables
SELECT Bridge.Name, Sensor.Type
FROM Bridge
INNER JOIN Sensor ON Bridge.Bridge_ID = Sensor.Bridge_ID;
```

Example with subquery:

```
-- Using subquery: only need Bridge data
SELECT Name
FROM Bridge
WHERE Bridge_ID IN (SELECT Bridge_ID FROM Sensor);
```

12.2.9 Data Definition Language (DDL)

DDL statements define the structure of databases. They create, modify, and delete tables and other database objects.

12.2.9.1 CREATE TABLE

The `CREATE TABLE` statement defines a new table with its columns, data types, and constraints.

Basic syntax:

```
CREATE TABLE table_name (
    column1 datatype constraints,
    column2 datatype constraints,
    ...
    table_constraints
);
```

Example: Create the Bridge table:

```
CREATE TABLE Bridge (
    Bridge_ID VARCHAR(10) PRIMARY KEY,
    Name VARCHAR(100) NOT NULL,
    Year_Built INTEGER,
    Length REAL CHECK (Length > 0)
);
```

12.2.9.2 Common Data Types

SQL provides various data types for different kinds of data:

Numeric types:

- `INTEGER` or `INT`: Whole numbers
- `REAL` or `FLOAT`: Floating-point numbers
- `DECIMAL(p,s)` or `NUMERIC(p,s)`: Fixed-precision decimals (p = precision, s = scale)

String types:

- `VARCHAR(n)`: Variable-length string, max n characters
- `CHAR(n)`: Fixed-length string, exactly n characters

- `TEXT`: Variable-length string, no specified maximum

Date/Time types:

- `DATE`: Date (year, month, day)
- `TIME`: Time (hour, minute, second)
- `TIMESTAMP` or `DATETIME`: Date and time combined

Boolean:

- `BOOLEAN`: True/false values (not supported in all systems)

Note: Data type names and availability vary by database system. The types listed here are common across most systems.

12.2.9.3 Constraints

Constraints enforce rules on data to maintain integrity.

PRIMARY KEY: Uniquely identifies each row

```
Bridge_ID VARCHAR(10) PRIMARY KEY
```

FOREIGN KEY: References a primary key in another table

```
CREATE TABLE Sensor (
    Sensor_ID VARCHAR(10) PRIMARY KEY,
    Bridge_ID VARCHAR(10),
    FOREIGN KEY (Bridge_ID) REFERENCES Bridge(Bridge_ID)
);
```

NOT NULL: Column cannot contain NULL values

```
Name VARCHAR(100) NOT NULL
```

UNIQUE: All values in column must be distinct

```
Email VARCHAR(100) UNIQUE
```

CHECK: Enforces a condition on column values

```
Year_Built INTEGER CHECK (Year_Built > 1800 AND Year_Built <= 2100)
```

DEFAULT: Provides default value when none specified

```
Install_Date DATE DEFAULT CURRENT_DATE
```

Example: Create Sensor table with multiple constraints:

```
CREATE TABLE Sensor (
    Sensor_ID VARCHAR(10) PRIMARY KEY,
    Type VARCHAR(50) NOT NULL,
    Bridge_ID VARCHAR(10) NOT NULL,
    Install_Date DATE,
    FOREIGN KEY (Bridge_ID) REFERENCES Bridge(Bridge_ID)
);
```

12.2.9.4 ALTER TABLE

The `ALTER TABLE` statement modifies an existing table structure.

Add a column:

```
ALTER TABLE Bridge
ADD COLUMN Material VARCHAR(50);
```

Drop a column:

```
ALTER TABLE Bridge
DROP COLUMN Material;
```

Modify a column (syntax varies by database system):

```
-- MySQL syntax
ALTER TABLE Bridge
MODIFY COLUMN Name VARCHAR(200);
```

Add a constraint:

```
ALTER TABLE Sensor
ADD CONSTRAINT check_type CHECK (Type IN ('accelerometer', 'strain_gauge', 'temperature'));
```

12.2.9.5 DROP TABLE

The `DROP TABLE` statement deletes a table and all its data permanently.

```
DROP TABLE table_name;
```

Example:

```
DROP TABLE Sensor;
```

Warning: This operation is irreversible. The table and all its data are permanently deleted.

Note: You cannot drop a table if other tables have foreign keys referencing it (unless you use CASCADE in some database systems).

12.2.9.6 Complete Example: Bridge Monitoring Database

Here's how to create the bridge monitoring database schema:

```
CREATE TABLE Bridge (
    Bridge_ID VARCHAR(10) PRIMARY KEY,
    Name VARCHAR(100) NOT NULL,
    Year_Built INTEGER CHECK (Year_Built > 1800),
    Length REAL CHECK (Length > 0)
);

CREATE TABLE Sensor (
    Sensor_ID VARCHAR(10) PRIMARY KEY,
    Type VARCHAR(50) NOT NULL,
    Bridge_ID VARCHAR(10) NOT NULL,
    Install_Date DATE,
    FOREIGN KEY (Bridge_ID) REFERENCES Bridge(Bridge_ID)
);

CREATE TABLE Reading (
    Sensor_ID VARCHAR(10),
    Sequence_Number INTEGER,
```

```
    Timestamp TIMESTAMP NOT NULL,
    Value REAL,
    PRIMARY KEY (Sensor_ID, Sequence_Number),
    FOREIGN KEY (Sensor_ID) REFERENCES Sensor(Sensor_ID)
);
```

This creates three tables with appropriate primary keys, foreign keys, and constraints.

12.2.10 Data Manipulation Language (DML)

DML statements modify the data stored in tables. They insert new records, update existing records, and delete records.

12.2.10.1 INSERT Statement

The `INSERT` statement adds new rows to a table.

Insert a single row (specify columns explicitly):

```
INSERT INTO table_name (column1, column2, column3)
VALUES (value1, value2, value3);
```

Example: Add a bridge:

```
INSERT INTO Bridge (Bridge_ID, Name, Year_Built, Length)
VALUES ('B001', 'Roberto Clemente Bridge', 1928, 244);
```

Insert without specifying columns (values must match all columns in order):

```
INSERT INTO Bridge
VALUES ('B002', 'Fort Pitt Bridge', 1959, 1201);
```

Insert multiple rows at once:

```
INSERT INTO Sensor (Sensor_ID, Type, Bridge_ID, Install_Date)
VALUES
    ('S001', 'accelerometer', 'B001', '2020-01-15'),
    ('S002', 'strain_gauge', 'B001', '2020-01-15'),
    ('S003', 'temperature', 'B002', '2019-12-10');
```

Insert from a SELECT query:

```
INSERT INTO Archive_Bridge
SELECT * FROM Bridge
WHERE Year_Built < 1900;
```

This copies all bridges built before 1900 into an archive table.

12.2.10.2 UPDATE Statement

The `UPDATE` statement modifies existing rows.

Basic syntax:

```
UPDATE table_name
SET column1 = value1, column2 = value2, ...
WHERE condition;
```

Example: Update a bridge's length:

```
UPDATE Bridge
SET Length = 245
WHERE Bridge_ID = 'B001';
```

Update multiple columns:

```
UPDATE Bridge
SET Name = 'Clemente Bridge', Length = 245
WHERE Bridge_ID = 'B001';
```

Update based on a condition:

```
UPDATE Sensor
SET Install_Date = '2020-01-01'
WHERE Install_Date IS NULL;
```

Warning: If you omit the WHERE clause, **all rows** will be updated:

```
-- DANGER: This updates every bridge
UPDATE Bridge
SET Length = 0;
```

Always include a WHERE clause unless you intend to update all rows.

12.2.10.3 DELETE Statement

The `DELETE` statement removes rows from a table.

Basic syntax:

```
DELETE FROM table_name
WHERE condition;
```

Example: Delete a specific sensor:

```
DELETE FROM Sensor
WHERE Sensor_ID = 'S003';
```

Delete based on a condition:

```
DELETE FROM Reading
WHERE Timestamp < '2020-01-01';
```

Warning: If you omit the WHERE clause, **all rows** will be deleted:

```
-- DANGER: This deletes all sensors
DELETE FROM Sensor;
```

The table structure remains, but all data is removed.

Note: You cannot delete rows if other tables have foreign key constraints referencing them (unless CASCADE DELETE is enabled).

12.2.10.4 Transactions (Brief Introduction)

A **transaction** is a sequence of SQL operations that execute as a single unit. Either all operations succeed, or all are rolled back.

Basic transaction commands:

- `BEGIN` or `START TRANSACTION`: Start a transaction
- `COMMIT`: Save all changes permanently
- `ROLLBACK`: Undo all changes since BEGIN

Example:

```sql
BEGIN;

INSERT INTO Bridge (Bridge_ID, Name, Year_Built, Length)
VALUES ('B005', 'New Bridge', 2024, 500);

UPDATE Sensor
SET Bridge_ID = 'B005'
WHERE Sensor_ID = 'S010';

COMMIT;
```

If any statement fails, you can `ROLLBACK` to undo all changes:

```sql
BEGIN;

DELETE FROM Bridge WHERE Bridge_ID = 'B001';
-- Oops, wrong bridge!

ROLLBACK;  -- Undoes the deletion
```

Transactions ensure data consistency, especially when multiple related operations must succeed together.

12.2.11 Setting Up and Using a Local DBMS

To practice SQL, you need a database management system installed on your computer. This section covers installation for MySQL and SQLite on macOS.

12.2.11.1 Option 1: SQLite (Recommended for Simplicity)

SQLite is a lightweight, serverless database engine. It's ideal for learning SQL because it requires minimal setup.

Installation on macOS:

SQLite comes pre-installed on macOS. To verify:

```sql
sqlite3 --version
```

If you see a version number, you're ready to go.

Using SQLite:

1. **Create a database** (or open existing):

```
sqlite3 mydatabase.db
```

2. **Execute SQL commands** at the SQLite prompt:

```
sqlite> CREATE TABLE Test (id INTEGER, name TEXT);
sqlite> INSERT INTO Test VALUES (1, 'Hello');
sqlite> SELECT * FROM Test;
```

3. **Exit SQLite**:

```
sqlite> .quit
```

Useful SQLite commands (start with a dot):

- `.tables` — List all tables
- `.schema tablename` — Show table structure
- `.quit` — Exit SQLite
- `.help` — Show all commands

SQLite Dialect Notes:

- SQLite uses dynamic typing (less strict than other systems)
- No RIGHT JOIN or FULL OUTER JOIN (use workarounds with UNION)
- Some ALTER TABLE operations are limited
- DATE/TIME handling differs slightly from other systems

For most of this course, these differences won't matter. Standard SQL works fine in SQLite.

12.2.11.2 Option 2: MySQL (Industry Standard)

MySQL is a full-featured, client-server database system widely used in industry.

Installation on macOS:

1. **Install via Homebrew** (recommended):

```
brew install mysql
```

2. **Start MySQL server**:

```
brew services start mysql
```

3. **Secure the installation** (optional but recommended):

```
mysql_secure_installation
```

4. **Connect to MySQL**:

```
mysql -u root -p
```

(Enter password when prompted, or just press Enter if no password set)

Installing MySQL Workbench (GUI client):

1. Download MySQL Workbench from mysql.com/downloads
2. Install the .dmg file
3. Launch MySQL Workbench
4. Create a connection to localhost (usually connects automatically)

Using MySQL Command Line:

Once connected to MySQL:

```
mysql> CREATE DATABASE mydb;
mysql> USE mydb;
mysql> CREATE TABLE Test (id INT, name VARCHAR(50));
mysql> INSERT INTO Test VALUES (1, 'Hello');
mysql> SELECT * FROM Test;
```

Useful MySQL commands:

- `SHOW DATABASES;` — List all databases
- `USE database_name;` — Switch to a database
- `SHOW TABLES;` — List tables in current database
- `DESCRIBE tablename;` — Show table structure
- `EXIT;` — Close connection

12.2.11.3 Choosing Between SQLite and MySQL

Use SQLite if:

- You want the simplest setup
- You're working on a single computer
- File-based databases suit your needs

Use MySQL if:

- You want industry-standard experience
- You prefer a GUI tool (MySQL Workbench)
- You plan to learn database administration

Both are excellent choices for this course. The SQL syntax is nearly identical for the topics we cover.

12.2.11.4 Next Steps

Once you have a DBMS installed, you're ready to create databases and practice SQL. Your instructor will provide scripts to set up example databases for exercises and assignments.

12.2.12 SQL vs Relational Algebra

This section provides a side-by-side comparison of relational algebra expressions (from Module 5) and their SQL equivalents. Understanding these correspondences helps you translate between the mathematical foundation and practical implementation.

12.2.12.1 Basic Operations

Operation	Relational Algebra	SQL
Select all	Bridge	`SELECT * FROM Bridge;`
Project columns	$\pi_{\text{Name, Year\_Built}}(\text{Bridge})$	`SELECT Name, Year_Built FROM Bridge;`
Filter rows	$\sigma_{\text{Year\_Built}<1950}(\text{Bridge})$	`SELECT * FROM Bridge WHERE Year_Built < 1950;`
Combined	$\pi_{\text{Name}}(\sigma_{\text{Length}>500}(\text{Bridge}))$	`SELECT Name FROM Bridge WHERE Length > 500;`

12.2.12.2 Set Operations

Operation	Relational Algebra	SQL
Union	$R \cup S$	`SELECT * FROM R UNION SELECT * FROM S;`
Intersection	$R \cap S$	`SELECT * FROM R INTERSECT SELECT * FROM S;`*
Difference	$R - S$	`SELECT * FROM R EXCEPT SELECT * FROM S;`*

*Note: SQLite doesn't support INTERSECT or EXCEPT. Use IN/NOT IN with subqueries as workarounds.

12.2.12.3 Joins

Operation	Relational Algebra	SQL
Cross product	Bridge $\times$ Sensor	`SELECT * FROM Bridge CROSS JOIN Sensor;`
Theta join	Bridge $\bowtie_{\text{Bridge.Bridge\_ID=Sensor.Bridge\_ID}}$ Sensor	`SELECT * FROM Bridge JOIN Sensor ON Bridge.Bridge_ID = Sensor.Bridge_ID;`

Operation	Relational Algebra	SQL
Natural join	Bridge ⋈ Sensor	`SELECT * FROM Bridge NATURAL JOIN Sensor;`

12.2.12.4 Complex Queries

Example 1: Find names of bridges with accelerometer sensors

Relational Algebra:

$$\pi_{\text{Name}}(\text{Bridge} \bowtie (\sigma_{\text{Type='accelerometer'}}(\text{Sensor})))$$

SQL:

```sql
SELECT Bridge.Name
FROM Bridge
JOIN Sensor ON Bridge.Bridge_ID = Sensor.Bridge_ID
WHERE Sensor.Type = 'accelerometer';
```

Example 2: Find bridges without any sensors

Relational Algebra:

$$\pi_{\text{Bridge\_ID}}(\text{Bridge}) - \pi_{\text{Bridge\_ID}}(\text{Sensor})$$

SQL:

```sql
SELECT Bridge_ID
FROM Bridge
WHERE Bridge_ID NOT IN (SELECT Bridge_ID FROM Sensor);
```

12.2.12.5 Key Differences

Beyond syntax, SQL and relational algebra differ in important ways:

Aspect	Relational Algebra	SQL
Duplicates	Relations are sets (no duplicates)	Tables allow duplicates (use DISTINCT to eliminate)
NULL values	Not part of theory	NULL represents missing/unknown data
Ordering	Relations are unordered sets	ORDER BY controls output order
Aggregation	Not in pure relational algebra	COUNT, SUM, AVG, etc.
Grouping	Not in pure relational algebra	GROUP BY clause

12.2.12.6 SQL Extensions Beyond Relational Algebra

SQL provides features that extend beyond pure relational algebra:

- **Aggregate functions**: COUNT, SUM, AVG, MIN, MAX
- **Grouping**: GROUP BY and HAVING
- **Sorting**: ORDER BY
- **NULL handling**: IS NULL, IS NOT NULL
- **String operations**: LIKE for pattern matching
- **Date/time operations**: Date arithmetic and formatting
- **Subqueries**: Nested SELECT statements
- **Data modification**: INSERT, UPDATE, DELETE
- **Schema definition**: CREATE, ALTER, DROP

These extensions make SQL more practical and powerful than pure relational algebra, while maintaining the same fundamental concepts.

12.2.12.7 Translation Practice

When translating between relational algebra and SQL:

1. **Identify the operations**: What relational algebra operators are used?
2. **Map to SQL clauses**:
    - $\sigma \rightarrow$ WHERE
    - $\pi \rightarrow$ SELECT columns
    - $\bowtie \rightarrow$ JOIN
    - $\cup, -, \cap \rightarrow$ UNION, EXCEPT, INTERSECT
3. **Apply SQL clause order**: FROM, WHERE, GROUP BY, HAVING, SELECT, ORDER BY
4. **Handle SQL-specific features**: Consider duplicates, NULLs, ordering

With practice, translating between these notations becomes second nature, allowing you to think mathematically about queries while writing practical SQL code.

13 Module 7: Database Design

13.1 Module Overview

13.1.1 Learning Objectives

By the end of this module, students will be able to:

- Explain how relational databases are designed and connect the theory behind it to earlier concepts such as Entity Relationship Diagrams and Relational Algebra
- Explain how the properties of the data lead to certain dependencies that allow for it to be structured, and understand how different structures lead to different design choices named "normal forms"
- Decompose a relation up to its Boyce-Codd Normal Form (BCNF), when possible
- Derive functional dependencies given properties of the data, and draw these dependencies in a diagram
- Understand the concept of closures and keys of relations

13.1.2 Topics Covered

- Principles of database design
- Properties and normal forms
- Functional dependencies
- Keys and closures

13.1.3 Project Milestones

Meet with your team to agree on a set of milestones and deliverables that are needed to complete the final project. This will be required as part of Assignment 4.

13.2 Lecture Notes

13.2.1 Why do we need to design a database?

In Modules 5 and 6, we learned the relational model and SQL—giving us the tools to create databases and query them. We even saw mechanical rules for translating ER diagrams into relational schemas. So we're done, right? We can design databases!

Not quite. While you can *create* a database from an ER diagram, the question remains: **Is your design any good?**

Different designs can store the same information, but some designs lead to:

- **Wasted storage** from redundant data
- **Inconsistencies** when data is updated incorrectly
- **Lost information** when rows are deleted
- **Maintenance nightmares** as the database evolves

Database design theory helps us create schemas that avoid these problems. The good news is there's elegant mathematical theory—based on functional dependencies and normal forms—that guides us toward better designs, often automatically.

13.2.1.1 A Poorly Designed Database

Let's see what can go wrong. Suppose we want to track our bridge monitoring system, and we decide to store everything in one big table:

BridgeSensor (poorly designed):

Bridge_ID	Bridge_Name	Year_Built	Length	Sensor_ID	Sensor_Type	Install_Date	Manufacturer	Mfr_Country
B001	Roberto Clemente Bridge	1928	244	S001	accelerometer	2020-01-15	Acme Sensors	USA
B001	Roberto Clemente Bridge	1928	244	S002	strain_gauge	2020-01-15	Acme Sensors	USA
B001	Roberto Clemente Bridge	1928	244	S003	temperature	2020-02-10	TempTech	Germany
B002	Fort Pitt Bridge	1959	1201	S004	accelerometer	2019-12-01	Acme Sensors	USA

Bridge_ID	Bridge_Name	Year_Built	Length	Sensor_ID	Sensor_Type	Install_Date	Manufacturer	Mfr_Country
B002	Fort Pitt Bridge	1959	1201	S005	strain_gauge	2019-12-01	SensorCorp	Japan

This design works—we can store data and query it. But look closely at the problems:

Problem 1: Redundancy

Notice how "Roberto Clemente Bridge", "1928", and "244" appear three times—once for each sensor on bridge B001. Similarly, "Acme Sensors" and "USA" appear three times. We're storing the same information repeatedly, wasting space.

Problem 2: Update Anomalies

Suppose we discover the Roberto Clemente Bridge is actually 245 meters long, not 244. We must update *three* rows. If we forget to update one row, our database becomes inconsistent—some rows say 244, others say 245. Which is correct?

Problem 3: Deletion Anomalies

What if we remove all sensors from the Fort Pitt Bridge (maybe for maintenance)? We'd delete both rows with Bridge_ID = B002. But now we've lost all information about that bridge—we no longer know it exists, when it was built, or its length. The sensor data deletion accidentally deleted bridge data too.

Problem 4: Insertion Anomalies

Suppose we build a new bridge but haven't installed sensors yet. We can't add a row to BridgeSensor because we don't have sensor information to fill in. We're forced to either wait until sensors are installed, or insert NULL values for sensor columns, or create dummy sensor entries—all unsatisfying workarounds.

13.2.1.2 A Better Design: Decomposition

Now let's decompose this into separate tables following proper design principles:

Bridge:

Bridge_ID	Bridge_Name	Year_Built	Length
B001	Roberto Clemente Bridge	1928	244
B002	Fort Pitt Bridge	1959	1201
B003	Fort Duquesne Bridge	1969	1201

Sensor:

Sensor_ID	Sensor_Type	Install_Date	Bridge_ID	Manufacturer_ID
S001	accelerometer	2020-01-15	B001	M001
S002	strain_gauge	2020-01-15	B001	M001
S003	temperature	2020-02-10	B001	M002
S004	accelerometer	2019-12-01	B002	M001
S005	strain_gauge	2019-12-01	B002	M003

Manufacturer:

Manufacturer_ID	Manufacturer	Mfr_Country
M001	Acme Sensors	USA
M002	TempTech	Germany
M003	SensorCorp	Japan

How decomposition solves the problems:

1. **No redundancy**: Each bridge's information appears exactly once. Each manufacturer's information appears exactly once.
2. **No update anomalies**: To change a bridge's length, we update exactly one row in the Bridge table. Impossible to create inconsistency.
3. **No deletion anomalies**: Removing all sensors from a bridge (deleting rows from Sensor table) doesn't affect the Bridge table. Bridge information persists.
4. **No insertion anomalies**: We can add a new bridge to the Bridge table without having any sensors. We can add a new manufacturer without having sensors from them yet.

13.2.1.3 Design Theory to the Rescue

The decomposition above looks obviously better. But how do we systematically arrive at such designs? How do we know we're done decomposing? Could we decompose further?

This is where **database design theory** comes in:

- **Functional dependencies** capture properties of the data (e.g., "Bridge_ID determines Bridge_Name")
- **Normal forms** (1NF, 2NF, 3NF, BCNF) provide target structures that eliminate anomalies

- **Decomposition algorithms** automatically break down mega-relations into well-designed tables

The rest of this module develops this theory systematically. We'll learn how to: 1. Identify functional dependencies from data properties 2. Recognize when a schema violates normal forms 3. Decompose relations to achieve higher normal forms (usually BCNF) 4. Verify that our decomposition preserves all information

By the end, you'll have principled techniques for designing databases that are efficient, consistent, and maintainable.

13.2.2 Database design theory

How do we systematically design good databases? Database design theory provides a principled approach based on **decomposition**.

13.2.2.1 The Decomposition Approach

The core idea is simple but powerful:

1. **Start with a "mega-relation"** containing all attributes in a single table
2. **Identify properties of the data** (functional dependencies)
3. **Decompose systematically** into smaller relations based on these properties
4. **Achieve a normal form** that guarantees no anomalies and no information loss

This process can be done algorithmically—given a set of functional dependencies, we can automatically decompose a relation into a well-designed schema.

13.2.2.2 The Decomposition Process

Here's the general workflow:

INPUT:

- A relation R with all attributes (the "mega-relation")
- Functional dependencies describing properties of the data (e.g., "Bridge_ID determines Bridge_Name")
- Optionally: multivalued dependencies for more complex cases

PROCEDURE:

- Analyze functional dependencies to identify problems (violations of normal forms)
- Systematically decompose R into smaller relations $R_1, R_2, \ldots, R_n$

- Each decomposition step removes one source of anomalies

OUTPUT:

- A set of relations in a target **normal form** (typically BCNF or 3NF)
- Guarantees:
  - **No anomalies**: No redundancy, no update/deletion/insertion anomalies
  - **Lossless decomposition**: We can reconstruct the original information by joining the decomposed relations
  - **Dependency preservation** (when possible): All original constraints are still enforceable

13.2.2.3 Normal Forms: Quality Levels for Designs

Normal forms are increasingly strict criteria for database design. Think of them as quality levels:

- **First Normal Form (1NF)**: Eliminate repeating groups and ensure atomic values
- **Second Normal Form (2NF)**: Eliminate partial dependencies on composite keys
- **Third Normal Form (3NF)**: Eliminate transitive dependencies
- **Boyce-Codd Normal Form (BCNF)**: Every determinant must be a candidate key

Each normal form eliminates certain types of anomalies. Higher normal forms are "better" but sometimes require tradeoffs. BCNF is the sweet spot for most applications—it eliminates all anomalies arising from functional dependencies.

13.2.2.4 Why This Matters

The beautiful thing about this theory is that it's **automatic and systematic**:

- You don't need to guess what tables to create
- You don't rely on intuition about what "feels right"
- The math tells you exactly how to decompose

This doesn't mean design is mechanical—you still need to:

- Understand your domain well enough to identify functional dependencies
- Sometimes make tradeoffs between normal forms and query performance
- Recognize when denormalization makes sense for specific use cases

But the theory gives you a solid foundation and a principled starting point.

13.2.2.5 Roadmap for This Module

To understand and apply this theory, we'll proceed as follows:

1. **Functional dependencies**: Learn what they are, how to identify them, and rules for reasoning about them
2. **Keys and closures**: Understand how functional dependencies relate to keys
3. **Normal forms**: Study 1NF, 2NF, 3NF, and BCNF in detail with examples
4. **Decomposition algorithms**: See how to systematically achieve normal forms

Let's start with functional dependencies—the foundation of everything else.

13.2.2.6 Functional Dependencies

A **functional dependency** (FD) is a constraint that describes a relationship between attributes. It captures the idea that knowing the value of certain attributes allows us to uniquely determine the value of other attributes.

Formal definition: Given a relation R, a functional dependency $\bar{A} \to \bar{B}$ holds if:

Whenever two tuples in R agree on all attributes in $\bar{A}$, they must also agree on all attributes in $\bar{B}$.

In other words, the values of attributes in $\bar{A}$ **determine** or **functionally determine** the values of attributes in $\bar{B}$.

Notation:

- $\bar{A}$ denotes a set of attributes $\{A_1, A_2, \ldots, A_n\}$
- $\bar{A} \to \bar{B}$ means "$\bar{A}$ functionally determines $\bar{B}$"
- We say "$\bar{A}$ is a **determinant**" and "$\bar{B}$ depends on $\bar{A}$"

13.2.2.7 Examples from the Bridge Monitoring System

Let's identify functional dependencies in our BridgeSensor mega-relation:

BridgeSensor(Bridge_ID, Bridge_Name, Year_Built, Length, Sensor_ID, Sensor_Type, Install_Date, Manufacturer, Mfr_Country)

FD 1: Bridge_ID → Bridge_Name, Year_Built, Length

If two rows have the same Bridge_ID, they must have the same Bridge_Name, Year_Built, and Length.

This makes sense: a bridge ID uniquely identifies a bridge, so all properties of that bridge are determined by the ID.

FD 2: Sensor_ID → Sensor_Type, Install_Date, Bridge_ID, Manufacturer

If two rows have the same Sensor_ID, they must have the same Sensor_Type, Install_Date, Bridge_ID, and Manufacturer.

A sensor ID uniquely identifies a sensor and all its properties, including which bridge it's on and who manufactured it.

FD 3: Manufacturer → Mfr_Country

If two rows have the same Manufacturer, they must have the same Mfr_Country.

A manufacturer's name determines where they're located. (We're assuming manufacturer names are unique—if not, we'd need a Manufacturer_ID.)

Non-example: We do NOT have Year_Built → Bridge_Name

Multiple bridges can be built in the same year, so knowing the year doesn't tell us which bridge. This is not a functional dependency.

13.2.2.8 Visualizing Functional Dependencies

We can draw functional dependency diagrams to show the structure. For BridgeSensor:

```
Bridge_ID → Bridge_Name, Year_Built, Length
Sensor_ID → Sensor_Type, Install_Date, Bridge_ID, Manufacturer
Manufacturer → Mfr_Country
```

These dependencies reveal the hidden structure in our data. Notice that Sensor_ID determines Bridge_ID, and Bridge_ID in turn determines bridge properties. By transitivity, Sensor_ID determines all attributes in the relation—making it a candidate key!

13.2.2.9 Types of Functional Dependencies

Given a functional dependency $\bar{A} \to \bar{B}$:

Trivial FD: $\bar{B} \subseteq \bar{A}$

Every attribute on the right side is also on the left side.

Example: {Bridge_ID, Bridge_Name} → Bridge_ID

This is always true (trivially) because if two tuples agree on Bridge_ID and Bridge_Name, they obviously agree on Bridge_ID alone. Trivial FDs are not interesting—they don't tell us anything useful.

Nontrivial FD: $\bar{B} \not\subseteq \bar{A}$

At least one attribute on the right side is not on the left side.

Example: Bridge_ID → {Bridge_Name, Bridge_ID}

This is nontrivial because Bridge_Name is not in the left side.

Completely nontrivial FD: $\bar{A} \cap \bar{B} = \emptyset$

No attribute appears on both sides.

Example: Bridge_ID → Bridge_Name, Year_Built

This is completely nontrivial—the sets are disjoint.

Why this matters: Completely nontrivial FDs are the most interesting for database design. They reveal true dependencies between different attributes.

13.2.2.10 Rules for Manipulating Functional Dependencies

Functional dependencies obey certain algebraic rules. These rules let us derive new FDs from existing ones.

Splitting Rule (Right Side):

If $\bar{A} \to \{B_1, B_2, \ldots, B_m\}$, then we can split this into multiple FDs:

$$\bar{A} \to B_1, \quad \bar{A} \to B_2, \quad \ldots, \quad \bar{A} \to B_m$$

Example: From Bridge_ID → {Bridge_Name, Year_Built, Length}, we get:

- Bridge_ID → Bridge_Name
- Bridge_ID → Year_Built
- Bridge_ID → Length

Combining Rule (Right Side):

Conversely, if we have multiple FDs with the same left side, we can combine them:

If $\bar{A} \to B_1$ and $\bar{A} \to B_2$, … and $\bar{A} \to B_m$, then:

$$\bar{A} \to \{B_1, B_2, \ldots, B_m\}$$

Important: We **cannot** split or combine the left side arbitrarily. The FD $\{A_1, A_2\} \to B$ is NOT equivalent to $A_1 \to B$ and $A_2 \to B$.

13.2.2.11 Functional Dependencies and Keys

Keys are intimately connected to functional dependencies. In fact, **a key is a set of attributes whose functional dependency extends to all attributes in the relation**.

Definition: A set of attributes $\bar{K}$ is a **key** for relation R if:

1. $\bar{K} \rightarrow$ all attributes of R (the key determines everything)
2. No proper subset of $\bar{K}$ has property 1 (minimality—can't remove any attribute from $\bar{K}$ and still have a key)

The first property means that knowing the values of key attributes uniquely identifies a tuple—no two tuples can have the same key values. The second property means the key contains no redundant attributes.

Example: In our BridgeSensor relation:

{Sensor_ID} $\rightarrow$ all attributes

This is a key because:

- Sensor_ID determines all nine attributes in the relation (directly and via transitivity through Bridge_ID and Manufacturer)
- It's a single attribute, so it's automatically minimal

Non-example: {Bridge_ID, Sensor_ID} also determines all attributes, but it's **not a key** because it's not minimal—we can remove Bridge_ID and still have a key (just Sensor_ID).

Multiple Candidate Keys:

A relation can have multiple keys. Each is called a **candidate key**. We typically choose one as the **primary key** for implementation.

Example: Consider a simpler relation:

Student(Student_ID, Email, Name, GPA)

With functional dependencies:

- Student_ID $\rightarrow$ Email, Name, GPA
- Email $\rightarrow$ Student_ID, Name, GPA

Both Student_ID and Email are candidate keys—either one uniquely identifies a student. We'd choose one (say Student_ID) as the primary key.

13.2.2.12 Inference Rules for Functional Dependencies

Beyond splitting and combining, there are several other important rules for deriving new functional dependencies from existing ones. These rules are called **Armstrong's Axioms** and form a complete system for reasoning about FDs.

Reflexivity (Trivial FD Rule):

If $\bar{B} \subseteq \bar{A}$, then $\bar{A} \to \bar{B}$

Any set of attributes determines its subsets.

Example: {Bridge_ID, Bridge_Name} → Bridge_ID (trivially true)

This captures trivial FDs—they're always true but not informative.

Augmentation:

If $\bar{A} \to \bar{B}$, then $\{\bar{A}, \bar{C}\} \to \{\bar{B}, \bar{C}\}$ for any set $\bar{C}$

If $\bar{A}$ determines $\bar{B}$, then adding attributes to both sides preserves the dependency.

Example: From Bridge_ID → Bridge_Name, we get:

$$\{\text{Bridge\_ID, Sensor\_ID}\} \to \{\text{Bridge\_Name, Sensor\_ID}\}$$

Transitivity:

If $\bar{A} \to \bar{B}$ and $\bar{B} \to \bar{C}$, then $\bar{A} \to \bar{C}$

Dependencies chain together.

Example: Suppose we have:

- Sensor_ID → Bridge_ID
- Bridge_ID → Year_Built

Then by transitivity: Sensor_ID → Year_Built

This makes intuitive sense: if a sensor ID determines which bridge, and the bridge determines when it was built, then the sensor ID determines when the bridge was built.

Derived Rules:

From Armstrong's Axioms, we can derive additional useful rules:

Union: If $\bar{A} \to \bar{B}$ and $\bar{A} \to \bar{C}$, then $\bar{A} \to \{\bar{B}, \bar{C}\}$

Decomposition: If $\bar{A} \to \{\bar{B}, \bar{C}\}$, then $\bar{A} \to \bar{B}$ and $\bar{A} \to \bar{C}$

Pseudotransitivity: If $\bar{A} \to \bar{B}$ and $\{\bar{B}, \bar{C}\} \to \bar{D}$, then $\{\bar{A}, \bar{C}\} \to \bar{D}$

These rules are all you need to derive any FD that logically follows from a given set of FDs. This completeness property is mathematically elegant—Armstrong's Axioms are both **sound** (only derive true FDs) and **complete** (can derive all true FDs).

13.2.2.13 Closures

The **closure** of a set of attributes is a powerful tool for answering questions about functional dependencies. It tells us all attributes that can be determined from a given set.

Definition: Given a set of functional dependencies F and a set of attributes $\bar{A}$, the **closure of** $\bar{A}$ (denoted $\bar{A}^+$) is:

The set of all attributes B such that $\bar{A} \to B$ can be inferred from F using Armstrong's Axioms.

In other words, $\bar{A}^+$ contains all attributes functionally determined by $\bar{A}$.

Computing the Closure:

Here's the algorithm for computing $\bar{A}^+$ given a set of FDs:

1. Start with $\bar{A}^+ = \bar{A}$ (the closure includes the starting attributes)
2. Repeat until no more attributes can be added:
    - For each FD $\bar{X} \to \bar{Y}$ in F:
        - If $\bar{X} \subseteq \bar{A}^+$ (all attributes in $\bar{X}$ are already in the closure)
        - Then add $\bar{Y}$ to $\bar{A}^+$ (we can now determine $\bar{Y}$ too)
3. Return $\bar{A}^+$

Example: Consider the BridgeSensor relation with these FDs:

- F_1: Bridge_ID → Bridge_Name, Year_Built, Length
- F_2: Sensor_ID → Sensor_Type, Install_Date, Bridge_ID, Manufacturer
- F_3: Manufacturer → Mfr_Country

Compute: {Sensor_ID}$^+$

Step 1: Start with {Sensor_ID}

Step 2: Apply F_2 since Sensor_ID is in our closure:

- Add Sensor_Type, Install_Date, Bridge_ID, Manufacturer
- Now: {Sensor_ID, Sensor_Type, Install_Date, Bridge_ID, Manufacturer}

Step 3: Apply F_1 since Bridge_ID is now in our closure:

- Add Bridge_Name, Year_Built, Length
- Now: {Sensor_ID, Sensor_Type, Install_Date, Bridge_ID, Manufacturer, Bridge_Name, Year_Built, L

Step 4: Apply F_3 since Manufacturer is another attribute with an associated FD that we added to our closure in Step 2:

- Add Mfr_Country
- Now: {Sensor_ID, Sensor_Type, Install_Date, Bridge_ID, Manufacturer, Bridge_Name, Year_Built, L

Step 5: No more FDs apply. Done!

Result: $\{\text{Sensor\_ID}\}^+ = \{\text{Sensor\_ID, Sensor\_Type, Install\_Date, Bridge\_ID, Manufacturer, Bridge\_Name,}$

This tells us: knowing the Sensor_ID, we can determine ALL attributes in the relation—not just sensor properties, but also which bridge it's on, all that bridge's properties, the manufacturer, and the manufacturer's country (all via transitivity). This confirms that Sensor_ID is a superkey (and in fact, a key).

Another Example: $\{\text{Bridge\_ID}\}^+$

Step 1: Start with {Bridge_ID}

Step 2: Apply F_1:

- Add Bridge_Name, Year_Built, Length
- Now: {Bridge_ID, Bridge_Name, Year_Built, Length}

Step 3: Can't apply F_2 (don't have Sensor_ID) or F_3 (don't have Manufacturer)

Result: $\{\text{Bridge\_ID}\}^+ = \{\text{Bridge\_ID, Bridge\_Name, Year\_Built, Length}\}$

Bridge_ID only determines bridge properties, not sensor or manufacturer properties.

13.2.2.14 Closures and Keys

Closures provide a simple test for whether a set of attributes is a key.

Key Test Using Closure:

A set of attributes $\bar{A}$ is a **superkey** for relation R if and only if $\bar{A}^+$ contains all attributes of R.

A superkey is a **key** if no proper subset of $\bar{A}$ is also a superkey (minimality condition).

Example: Is {Sensor_ID} a key for BridgeSensor?

From our earlier calculation: $\{\text{Sensor\_ID}\}^+ = \{\text{Sensor\_ID, Sensor\_Type, Install\_Date, Bridge\_ID, Manufactu}$

This is all 9 attributes! Therefore, {Sensor_ID} **is** a superkey.

Is it a key? Since it's a single attribute, it's automatically minimal—we can't remove any attributes from it. Therefore, {Sensor_ID} is **a key** for BridgeSensor.

Example: Is {Bridge_ID, Sensor_ID} a key?

Let's compute the closure:

Start: {Bridge_ID, Sensor_ID}

Apply F_1 (Bridge_ID → Bridge_Name, Year_Built, Length):

- Add: Bridge_Name, Year_Built, Length

Apply F_2 (Sensor_ID → Sensor_Type, Install_Date, Bridge_ID, Manufacturer):

- Add: Sensor_Type, Install_Date, Manufacturer (Bridge_ID already there)

Apply F_3 (Manufacturer → Mfr_Country):

- Add: Mfr_Country

Result: All 9 attributes! So {Bridge_ID, Sensor_ID} is a **superkey**.

But is it a key? Check if any proper subset is also a superkey:

$\{\text{Bridge\_ID}\}^+ = \{\text{Bridge\_ID, Bridge\_Name, Year\_Built, Length}\}$ — not all attributes

$\{\text{Sensor\_ID}\}^+$ results in **all attributes** as we have seen before.

Since {Sensor_ID} is a superkey and a proper subset of {Bridge_ID, Sensor_ID}, the composite is **not** a key (it violates minimality). The key for BridgeSensor is just Sensor_ID—each row represents one sensor, and sensor IDs are unique.

Finding All Keys Systematically:

How do we find all candidate keys for a relation?

Naive approach: Test every subset of attributes. For n attributes, that's 2^n subsets—exponential and impractical for large relations.

Better approach: Use the minimality property

1. Start by testing single attributes: If any attribute A has $\{A\}^+ =$ all attributes, then $\{A\}$ is a key
2. Test pairs of attributes: $\{A, B\}$
3. Test triples: $\{A, B, C\}$
4. And so on…

Key optimization: If $\bar{C}$ is a key, then any superset $\{\bar{C}, D\}$ is a superkey but **not** a key (violates minimality). So once we find a key of size k, we don't need to test any supersets of that key.

Example: For BridgeSensor with 9 attributes, we'd test:

- 9 single-attribute sets
- If none are keys, test $\binom{9}{2} = 36$ pairs

- And so on…

In practice, domain knowledge often tells us what the keys are. But closure computation gives us a mechanical procedure when we're unsure.

13.2.2.15 Specifying Functional Dependencies for a Relation

When designing a database, we need to specify which functional dependencies hold. But there are infinitely many true FDs (including all the trivial ones). How do we choose which ones to write down?

Logical Implication:

We say a set of FDs S_2 **follows from** (or is **implied by**) a set S_1 if:

Every relation instance that satisfies all FDs in S_1 also satisfies all FDs in S_2.

In other words, S_1 logically entails S_2.

Example:

Given: $S_1 = \{\text{Sensor\_ID} \rightarrow \text{Bridge\_ID}, \text{Bridge\_ID} \rightarrow \text{Year\_Built}\}$

Does S_1 imply Sensor_ID $\rightarrow$ Year_Built?

Yes! By transitivity. Any relation satisfying S_1 must also satisfy Sensor_ID $\rightarrow$ Year_Built.

Testing If an FD Follows:

To test whether $\bar{A} \rightarrow \bar{B}$ follows from a set S of FDs:

1. Compute $\bar{A}^+$ using the FDs in S
2. If $\bar{B} \subseteq \bar{A}^+$, then $\bar{A} \rightarrow \bar{B}$ follows from S
3. Otherwise, it doesn't

Example:

Given FDs:

- Bridge_ID $\rightarrow$ Bridge_Name, Year_Built, Length
- Sensor_ID $\rightarrow$ Bridge_ID, Sensor_Type

Does Sensor_ID $\rightarrow$ Year_Built follow?

Compute $\{\text{Sensor\_ID}\}^+$:

- Start: Sensor_ID
- Apply second FD: Add Bridge_ID, Sensor_Type
- Apply first FD: Add Bridge_Name, Year_Built, Length
- Result: {Sensor_ID, Bridge_ID, Sensor_Type, Bridge_Name, Year_Built, Length}

Since Year_Built is in the closure, **yes**, Sensor_ID → Year_Built follows.

Minimal Basis (Canonical Cover):

Our goal when specifying FDs is to find a **minimal set** that captures all the dependencies without redundancy.

A set of FDs S is **minimal** (or a **canonical cover**) if:

1. All FDs are **completely nontrivial** (right side doesn't overlap with left side)
2. No FD in S can be derived from the others (no redundancy)
3. No attribute in any left side is redundant (can't make any left side smaller)

Example of redundant FDs:

Consider:

- A → B
- B → C
- A → C

The third FD is redundant—it follows from the first two by transitivity. A minimal set would omit it.

Why minimality matters:

- **Clarity**: Easier to understand the essential dependencies
- **Efficiency**: Fewer constraints to check when inserting/updating data
- **Design**: Minimal FD sets clearly reveal the structure needed for normalization

In practice, you identify FDs based on domain knowledge (understanding what real-world properties constrain the data), then optionally simplify to a minimal basis. The minimal basis is what you'll use when decomposing relations.

13.2.3 Normal Forms

Now that we understand functional dependencies, we can study **normal forms**—quality criteria for database design. Each normal form eliminates certain types of anomalies by restricting what functional dependencies are allowed.

Normal forms form a hierarchy:

$$1\text{NF} \subset 2\text{NF} \subset 3\text{NF} \subset \text{BCNF}$$

Every relation in 2NF is also in 1NF. Every relation in 3NF is also in 2NF. And so on. Higher normal forms are "stricter" and eliminate more anomalies.

13.2.3.1 Why Normalization Matters

Unnormalized databases suffer from several problems:

1. **Anomalies**: Redundancy leads to update, deletion, and insertion anomalies (as we saw earlier)
2. **Inefficient updates**: Programs must update multiple copies of the same data
3. **Complex indexing**: Indexing large, redundant tables is cumbersome
4. **Wasted storage**: Repeated data consumes unnecessary space
5. **Maintenance nightmares**: Changes to schema or constraints require updating many places

Normalization systematically eliminates these problems by decomposing relations according to their functional dependencies.

Important notes:

- Normalization should be part of the design process, not an afterthought
- The normalization process may reveal additional entities and relationships to add to your ER diagram
- ER modeling and normalization should be used together—ER diagrams help identify entities and relationships, normalization ensures the resulting schema is well-designed

13.2.3.2 The Running Example

We'll use our BridgeSensor mega-relation from earlier:

BridgeSensor(Bridge_ID, Bridge_Name, Year_Built, Length, Sensor_ID, Sensor_Type, Install_Date, Manufacturer, Mfr_Country)

With functional dependencies:

- F_1: Bridge_ID → Bridge_Name, Year_Built, Length
- F_2: Sensor_ID → Sensor_Type, Install_Date, Bridge_ID, Manufacturer
- F_3: Manufacturer → Mfr_Country

Key: {Sensor_ID} (as we discovered via closure)

Sample data:

Bridge_ID	Bridge_Name	Year_Built	Length	Sensor_ID	Sensor_Type	Install_Date	Manufacturer	Mfr_Country
B001	Roberto Clemente Bridge	1928	244	S001	accelerometer	2020-01-15	Acme Sensors	USA

Bridge_ID	Bridge_Name	Year_Built	Length	Sensor_ID	Sensor_Type	Install_Date	Manufacturer	Mfr_Country
B001	Roberto Clemente Bridge	1928	244	S002	strain_gauge	2020-01-15	Acme Sensors	USA
B001	Roberto Clemente Bridge	1928	244	S003	temperature	2020-02-10	TempTech	Germany
B002	Fort Pitt Bridge	1959	1201	S004	accelerometer	2019-12-01	Acme Sensors	USA

We'll progressively normalize this relation through 1NF, 2NF, 3NF, and finally BCNF.

13.2.3.3 First Normal Form (1NF)

Definition: A relation is in **First Normal Form (1NF)** if:

1. All attribute values are **atomic** (indivisible)
2. Each attribute contains only a single value from its domain (no sets, lists, or repeating groups)
3. There is a **primary key** that uniquely identifies each row

What 1NF eliminates: Repeating groups and multi-valued attributes.

Simple Example of 1NF Violation:

Consider a poorly designed Employee relation:

Employee_ID	Name	Phone_Numbers
E001	Alice	412-555-0100, 412-555-0101
E002	Bob	412-555-0200

The Phone_Numbers column violates 1NF—it contains a list (multiple phone numbers in one cell).

To fix: Create separate rows for each phone number:

Employee_ID	Name	Phone_Number
E001	Alice	412-555-0100
E001	Alice	412-555-0101
E002	Bob	412-555-0200

Employee_ID	Name	Phone_Number

Now each cell contains a single atomic value.

Our BridgeSensor Example:

Our BridgeSensor relation **is already in 1NF**:

- All values are atomic (single values, not lists)
- Primary key exists: Sensor_ID

So we can move directly to analyzing higher normal forms. The real action begins with 2NF, where we address **partial dependencies**.

13.2.3.4 Second Normal Form (2NF)

Definition: A relation is in **Second Normal Form (2NF)** if:

1. It is in 1NF, AND
2. Every non-key attribute is **fully functionally dependent** on the entire primary key (no partial dependencies)

What is a partial dependency?

A **partial dependency** occurs when a non-key attribute depends on only *part* of a composite primary key, not the whole key.

If the primary key is a single attribute, partial dependencies are impossible—the relation is automatically in 2NF (assuming it's in 1NF).

Modified BridgeSensor for 2NF Demonstration:

To demonstrate 2NF violations, let's adjust our example slightly. Suppose sensor IDs are only unique *within* each bridge (Sensor 1 on Bridge A is different from Sensor 1 on Bridge B). Then:

Key: {Bridge_ID, Sensor_ID} (composite key)

BridgeSensor(Bridge_ID, Bridge_Name, Year_Built, Length, Sensor_ID, Sensor_Type, Install_Date, Manufacturer, Mfr_Country)

Functional dependencies:

- Bridge_ID → Bridge_Name, Year_Built, Length ← **Partial!** (depends only on part of key)
- {Bridge_ID, Sensor_ID} → Sensor_Type, Install_Date, Manufacturer

- Manufacturer → Mfr_Country

Problem: Bridge_Name, Year_Built, and Length depend only on Bridge_ID, not on the full key {Bridge_ID, Sensor_ID}. This is a **partial dependency**, violating 2NF.

Anomalies from 2NF violation:

Looking at the data:

Bridge_ID	Bridge_Name	Year_Built	Length	Sensor_ID	Sensor_Type	Install_Date	Manufacturer	Mfr_Country
B001	Roberto Clemente Bridge	1928	244	1	accelerometer	2020-01-15	Acme Sensors	USA
B001	Roberto Clemente Bridge	1928	244	2	strain_gauge	2020-01-15	Acme Sensors	USA
B001	Roberto Clemente Bridge	1928	244	3	temperature	2020-02-10	TempTech	Germany

Bridge information repeats for each sensor—classic redundancy from partial dependency.

Decomposition to 2NF:

Remove attributes that depend on only part of the key:

Bridge(Bridge_ID, Bridge_Name, Year_Built, Length)

- Primary key: Bridge_ID
- No partial dependencies (key is single attribute)

BridgeSensor(Bridge_ID, Sensor_ID, Sensor_Type, Install_Date, Manufacturer, Mfr_Country)

- Primary key: {Bridge_ID, Sensor_ID}
- All non-key attributes depend on the *full* key

Now the tables:

Bridge:

Bridge_ID	Bridge_Name	Year_Built	Length
B001	Roberto Clemente Bridge	1928	244
B002	Fort Pitt Bridge	1959	1201

BridgeSensor:

Bridge_ID	Sensor_ID	Sensor_Type	Install_Date	Manufacturer	Mfr_Country
B001	1	accelerometer	2020-01-15	Acme Sensors	USA
B001	2	strain_gauge	2020-01-15	Acme Sensors	USA
B001	3	temperature	2020-02-10	TempTech	Germany
B002	1	accelerometer	2019-12-01	Acme Sensors	USA

Benefits of 2NF:

- **Eliminated redundancy**: Bridge information appears only once
- **No update anomalies**: Changing a bridge's length requires updating one row
- **No deletion anomalies**: Removing all sensors from a bridge doesn't delete bridge information
- **No insertion anomalies**: Can add a bridge without sensors

Note: Relations with single-attribute primary keys (like Bridge above) are automatically in 2NF if they're in 1NF, since partial dependencies are impossible.

Next problem: BridgeSensor is now in 2NF, but still has issues. Notice Manufacturer appears multiple times, and Mfr_Country is redundant. This is a **transitive dependency**, which 3NF addresses.

13.2.3.5 Third Normal Form (3NF)

Definition: A relation is in **Third Normal Form (3NF)** if:

1. It is in 2NF, AND
2. No non-key attribute is **transitively dependent** on the primary key

What is a transitive dependency?

A **transitive dependency** occurs when: - Key $\rightarrow A \rightarrow B$

In other words, a non-key attribute A determines another non-key attribute B. The key determines B indirectly through A (by transitivity).

Example from BridgeSensor (currently in 2NF):

BridgeSensor(Bridge_ID, Sensor_ID, Sensor_Type, Install_Date, Manufacturer, Mfr_Country)

Functional dependencies:

- {Bridge_ID, Sensor_ID} $\rightarrow$ Manufacturer (key determines Manufacturer)

- Manufacturer → Mfr_Country (non-key determines non-key)

By transitivity: {Bridge_ID, Sensor_ID} → Manufacturer → Mfr_Country

This is a **transitive dependency**—the key determines Mfr_Country indirectly through Manufacturer. Violates 3NF!

Problem with transitive dependencies:

Look at the data:

Bridge_ID	Sensor_ID	Sensor_Type	Install_Date	Manufacturer	Mfr_Country
B001	1	accelerometer	2020-01-15	Acme Sensors	USA
B001	2	strain_gauge	2020-01-15	Acme Sensors	USA
B001	3	temperature	2020-02-10	TempTech	Germany
B002	1	accelerometer	2019-12-01	Acme Sensors	USA

Notice "Acme Sensors → USA" appears three times. If Acme Sensors moved to Canada:

- We'd need to update multiple rows (update anomaly)
- If we update inconsistently, we have contradictory data

Decomposition to 3NF:

Identify the transitive dependency and split it out:

1. **Identify determinants**: Manufacturer is a determinant (determines Mfr_Country)
2. **Create new relation**: Make a relation with Manufacturer as the key
3. **Remove dependent attributes**: Remove Mfr_Country from BridgeSensor

Sensor(Bridge_ID, Sensor_ID, Sensor_Type, Install_Date, Manufacturer)

- Primary key: {Bridge_ID, Sensor_ID}
- Foreign key: Manufacturer references Manufacturer table
- No transitive dependencies!

Manufacturer(Manufacturer, Mfr_Country)

- Primary key: Manufacturer
- Simple relation, no transitive dependencies

Now the tables:

Sensor:

Bridge_ID	Sensor_ID	Sensor_Type	Install_Date	Manufacturer
B001	1	accelerometer	2020-01-15	Acme Sensors
B001	2	strain_gauge	2020-01-15	Acme Sensors
B001	3	temperature	2020-02-10	TempTech
B002	1	accelerometer	2019-12-01	Acme Sensors

Manufacturer:

Manufacturer	Mfr_Country
Acme Sensors	USA
TempTech	Germany

Benefits of 3NF:

- **No redundancy from transitive dependencies**: Each manufacturer's country appears once
- **No update anomalies**: Changing Acme Sensors' country requires one update
- **No deletion anomalies**: Deleting all sensors from a manufacturer doesn't delete manufacturer info
- **Better data integrity**: Can't have inconsistent countries for the same manufacturer

Summary so far:

We now have three relations, all in 3NF:

1. **Bridge**(Bridge_ID, Bridge_Name, Year_Built, Length)
2. **Sensor**(Bridge_ID, Sensor_ID, Sensor_Type, Install_Date, Manufacturer)
3. **Manufacturer**(Manufacturer, Mfr_Country)

Are we done? For most practical purposes, **yes**—3NF is often good enough. But there's one more normal form worth knowing: BCNF, which handles a subtle edge case.

13.2.3.6 Boyce-Codd Normal Form (BCNF)

Definition: A relation is in **Boyce-Codd Normal Form (BCNF)** if:

For every nontrivial functional dependency $\bar{A} \to \bar{B}$, $\bar{A}$ is a **superkey** (contains a candidate key).

In simpler terms: **Every determinant must be a candidate key.**

BCNF is a stricter version of 3NF. While 3NF says "no non-key attribute determines another non-key attribute," BCNF says "ONLY keys can be determinants."

Relationship to 3NF:

- Every relation in BCNF is also in 3NF
- Most relations in 3NF are also in BCNF
- BCNF violations occur in special cases, typically when there are **overlapping candidate keys**

When does BCNF differ from 3NF?

BCNF catches cases where a non-key attribute determines a key attribute (or part of a key). This doesn't violate 3NF (which only restricts non-key → non-key dependencies), but it still causes anomalies.

Example: Bridge Inspection Schedule

Consider a relation tracking bridge inspections:

Inspection(Inspector, Bridge, Bridge_Type, Inspection_Date)

Business rules:

1. Each inspector specializes in exactly one bridge type (arch, suspension, truss)
2. Each bridge has exactly one type
3. Each inspector inspects each bridge at most once

Functional dependencies:

- Inspector → Bridge_Type (each inspector specializes in one type)
- Bridge → Bridge_Type (each bridge has one type)
- {Inspector, Bridge} → Inspection_Date (key determines date)

Keys: {Inspector, Bridge} is the only candidate key

Sample data:

Inspector	Bridge	Bridge_Type	Inspection_Date
Alice	B001	suspension	2024-01-15
Alice	B002	suspension	2024-02-10
Bob	B003	arch	2024-01-20
Bob	B004	arch	2024-03-05
Carol	B005	truss	2024-02-01

Is this in 3NF?

Check for transitive dependencies (non-key → non-key):

- Bridge_Type is determined by Inspector (both are part of different candidate keys or non-key)
- But wait—Inspector and Bridge are both part of the primary key!
- No non-key attribute determines another non-key attribute

Yes, this is in 3NF.

Is this in BCNF?

Check if every determinant is a superkey:

- Inspector → Bridge_Type, but Inspector alone is NOT a superkey
- Inspector is a determinant but not a candidate key!

No, this violates BCNF.

Problem—Anomalies despite being in 3NF:

Look at the data again:

Inspector	Bridge	Bridge_Type	Inspection_Date
Alice	B001	suspension	2024-01-15
Alice	B002	suspension	2024-02-10

Bridge_Type "suspension" is repeated for every bridge Alice inspects.

- **Update anomaly**: If Alice switches specializations from suspension to arch, we must update every row where she appears
- **Insertion anomaly**: We can't record that a new inspector specializes in truss bridges until they inspect a bridge
- **Deletion anomaly**: If Bob stops inspecting B003 and B004, we lose the information that Bob specializes in arch bridges

Decomposition to BCNF:

Split based on the problematic FD Inspector → Bridge_Type:

InspectorSpecialty(Inspector, Bridge_Type)

- Primary key: Inspector
- FD: Inspector → Bridge_Type (determinant IS a key)

BridgeInspection(Inspector, Bridge, Inspection_Date)

- Primary key: {Inspector, Bridge}
- FD: {Inspector, Bridge} → Inspection_Date (determinant IS a key)

Now the tables:

InspectorSpecialty:

Inspector	Bridge_Type
Alice	suspension
Bob	arch
Carol	truss

BridgeInspection:

Inspector	Bridge	Inspection_Date
Alice	B001	2024-01-15
Alice	B002	2024-02-10
Bob	B003	2024-01-20
Bob	B004	2024-03-05
Carol	B005	2024-02-01

Both relations are now in BCNF. Every determinant is a candidate key.

Benefits of BCNF:

- **No redundancy**: Each inspector's specialty appears once
- **No anomalies**: All the update/insertion/deletion anomalies are eliminated
- **Stronger guarantee**: BCNF is the "gold standard" for functional dependency-based normal forms

Note: We also need the FD Bridge → Bridge_Type to be enforced. We'd need:

BridgeType(Bridge, Bridge_Type)

Making our final BCNF schema:

1. **InspectorSpecialty**(Inspector, Bridge_Type)
2. **BridgeInspection**(Inspector, Bridge, Inspection_Date)
3. **BridgeType**(Bridge, Bridge_Type)

When to stop at 3NF vs. push to BCNF:

- **BCNF is the theoretical ideal**: No anomalies from functional dependencies
- **3NF is sometimes preferred** when achieving BCNF would:
  - Require too many joins for common queries (performance cost)

– Lose dependency preservation (can't enforce all original FDs with just the decomposed tables)

For most applications, BCNF is the target. But there are legitimate cases where 3NF is good enough.

13.2.3.7 Higher Normal Forms

BCNF handles all anomalies arising from functional dependencies. But there are other types of dependencies that cause anomalies, leading to even higher normal forms.

Fourth Normal Form (4NF):

4NF addresses **multivalued dependencies** (MVDs)—situations where one attribute determines a set of values for another attribute, independently of other attributes.

Example: Suppose an instructor teaches multiple courses and has multiple phone numbers, independently:

Instructor	Course	Phone
Dr. Smith	DB Design	555-0100
Dr. Smith	DB Design	555-0101
Dr. Smith	Data Mining	555-0100
Dr. Smith	Data Mining	555-0101

Every combination appears, creating redundancy. The multivalued dependency is:

$$\text{Instructor} \twoheadrightarrow \text{Course}$$

$$\text{Instructor} \twoheadrightarrow \text{Phone}$$

To achieve 4NF, split into two relations:

- **InstructorCourse**(Instructor, Course)
- **InstructorPhone**(Instructor, Phone)

Fifth Normal Form (5NF) and Beyond:

5NF (also called **Projection-Join Normal Form**) deals with join dependencies—complex cases where information can only be reconstructed by joining three or more tables.

Higher normal forms (6NF, Domain-Key Normal Form) exist but are primarily of theoretical interest. They're rarely used in practice because:

1. BCNF eliminates virtually all practical anomalies in most databases
2. Higher normal forms can require excessive joins, hurting query performance
3. The additional complexity rarely justifies the marginal benefit

Practical Advice:

Don't assume the highest level of normalization is always the most desirable.

- **BCNF is the sweet spot** for most applications—strong guarantees without excessive complexity
- **Higher normal forms** require more joins, which can slow query performance
- **Denormalization** (intentionally violating normal forms) is sometimes appropriate for:
  - Read-heavy applications where query speed matters more than update efficiency
  - Data warehouses optimized for analytics
  - Caching frequently accessed aggregated data

The key is understanding the tradeoffs and making informed design decisions based on your specific requirements.

14 Module 8: Data Representation and Compression

14.1 Module Overview

14.1.1 Learning Objectives

By the end of this module, students will be able to:

- Explain why different choices of how to represent (or compress) data/information offer tradeoffs that we should be aware of
- Predict the size of a simple ASCII text file based on its content
- Describe the differences between lossless and lossy compression schemes
- Understand the benefits of frequency domain representations for certain types of data
- Apply Huffman coding to an arbitrary text string
- Be aware of the compression algorithms used for JPEG and in other domains

14.1.2 Topics Covered

- Data Representation: understanding tradeoffs in representation choices
- Filesystems and ASCII encoding
- Types of data compression (lossless vs. lossy)
- Frequency Domain Representation of Data
- Huffman coding
- JPEG and other compression applications
- LLMs as compression algorithms?

14.1.3 Project Milestones

Schedule a meeting with the instructor for any clarifications needed before you go on Thanksgiving break.

14.2 Lecture Notes

14.2.1 Why Talk About Data Compression?

The database systems we've been studying are largely the "at rest" destination for data in a longer pipeline. Roughly speaking, data is first acquired, then transmitted, and finally stored. At each stage, **size matters**:

- The more data we transmit, the higher the bandwidth costs and the longer the transfer times
- The more data we store, the higher the storage costs and the slower our queries

Data compression allows us to represent, transmit, and store data in a more compact form. We measure file sizes in **bits** (or bytes, where 1 byte = 8 bits), and we quantify compression efficiency using the **compression ratio**—the ratio of the original size to the compressed size.

But before we can compress data, we need to understand how data is represented in the first place. Let's start with something familiar: text.

14.2.1.1 How Text Becomes Bits: Character Encoding

When you save a text file, each character must be converted to a sequence of bits. The mapping from characters to bit sequences is called a **character encoding**. Different encodings make different tradeoffs:

ASCII (American Standard Code for Information Interchange):

- Uses exactly 7 bits per character (often stored as 8 bits for convenience)
- Can represent 128 characters: uppercase and lowercase English letters, digits, punctuation, and control characters
- Simple and compact for English text, but cannot represent characters from other languages

UTF-8 (Unicode Transformation Format - 8 bit):

- Variable-length encoding: 1 to 4 bytes per character
- ASCII characters use just 1 byte (backward compatible with ASCII)
- Extended characters (accents, non-Latin scripts, emoji) use 2-4 bytes
- The dominant encoding on the web today

UTF-16:

- Uses 2 or 4 bytes per character
- More efficient for texts with many non-ASCII characters (like Chinese or Japanese)
- Less efficient for English text compared to UTF-8

14.2.1.2 A Concrete Example

Consider the text: **"Life, liberty and the pursuit of happiness."**

This string has 43 characters (including spaces and punctuation). How much storage does it require?

Encoding	Bytes per Character	Total Size
ASCII	1 byte (for these characters)	43 bytes = 344 bits
UTF-8	1 byte (all ASCII characters)	43 bytes = 344 bits
UTF-16	2 bytes minimum	86 bytes = 688 bits
UTF-32	4 bytes fixed	172 bytes = 1,376 bits

For this English text, ASCII and UTF-8 are equally efficient. But if we included an emoji like , UTF-8 would need 4 extra bytes for that single character, while ASCII couldn't represent it at all.

14.2.1.3 Try It Yourself

Use the interactive tool below to explore how different text strings are encoded. Try entering text with special characters or emoji to see how the encoding affects file size:

```
//| echo: false
viewof text_input = Inputs.textarea({
  label: "Enter your text:",
  value: "Life, liberty and the pursuit of happiness.",
  rows: 2,
  width: "100%"
})
```

```
//| echo: false
viewof encoding_select = Inputs.select(
  ["ASCII", "UTF-8", "UTF-16", "UTF-32"],
  {label: "Select encoding:", value: "UTF-8"}
)
```

```
//| echo: false
function calculateSize(text, encoding) {
  try {
    let bytes;
```

```
    if (encoding === "ASCII") {
      // Check if all characters are ASCII (code points < 128)
      for (let i = 0; i < text.length; i++) {
        if (text.charCodeAt(i) > 127) {
          return { success: false };
        }
      }
      bytes = text.length; // 1 byte per character for ASCII
    } else if (encoding === "UTF-8") {
      // Use TextEncoder for UTF-8
      const encoder = new TextEncoder();
      bytes = encoder.encode(text).length;
    } else if (encoding === "UTF-16") {
      // UTF-16: 2 bytes per BMP character, 4 bytes for supplementary
      bytes = 0;
      for (let i = 0; i < text.length; i++) {
        const code = text.codePointAt(i);
        if (code > 0xFFFF) {
          bytes += 4;
          i++; // Skip the next code unit (surrogate pair)
        } else {
          bytes += 2;
        }
      }
    } else if (encoding === "UTF-32") {
      // UTF-32: 4 bytes per character (code point)
      bytes = [...text].length * 4; // Spread to handle surrogate pairs correctly
    }
    return { success: true, bytes: bytes, bits: bytes * 8 };
  } catch (e) {
    return { success: false };
  }
}

result = calculateSize(text_input, encoding_select)
charCount = [...text_input].length // Correct character count handling surrogate pairs
```

```
//| echo: false
html`<div style="background: #f8f9fa; padding: 1em; border-radius: 5px; margin-top: 1em;">
${result.success
  ? html`<strong>Results for ${encoding_select} encoding:</strong>
<ul>
```

```
<li>Characters: <strong>${charCount}</strong></li>
<li>Bytes: <strong>${result.bytes}</strong></li>
<li>Bits: <strong>${result.bits}</strong></li>
<li>Average bits per character: <strong>${(result.bits / charCount).toFixed(2)}</strong></li
</ul>`
  : html`<strong>Error:</strong> The text contains characters that cannot be encoded in ${enc
}
</div>`
```

This simple example illustrates a key insight: **the same information can be represented in different ways, with different storage costs**. This is the foundation of data compression—finding more efficient representations.

14.2.2 Change of Basis to Uncover Structure

Character encoding showed us that the same text can be represented with different numbers of bits. But there's a deeper idea: sometimes changing your **perspective** on data reveals structure that wasn't obvious before—structure that enables compression.

Consider a musical note. In the **time domain**, we represent it as a sequence of air pressure measurements over time—thousands of samples per second. But in the **frequency domain**, that same note might be described by just a few numbers: a fundamental frequency (the pitch) and a handful of harmonics with their amplitudes. The same information, radically different representations.

The mathematical tool that converts between these perspectives is the **Fourier Transform** (or its discrete version, the DFT/FFT). Here's the key insight:

- Time-domain and frequency-domain representations contain **exactly the same information**
- They use the **same number of values** (no compression yet!)
- But frequency-domain representations often reveal that only a few components carry most of the signal's energy

This is why frequency-domain analysis underlies many compression algorithms: once you see that most coefficients are near zero, you can throw them away.

14.2.2.1 Example: Accelerometer Data from a Building

Let's look at real data. In Assignment 1, we analyzed accelerometer recordings from a building during an earthquake. The data was recorded at 100 Hz (one sample every 0.01 seconds) for 50 seconds—that's 5,000 samples per sensor.

Here's what one sensor's data looks like in the time domain versus the frequency domain:

```
import numpy as np
import matplotlib.pyplot as plt

# Load the seismic response data
data = np.loadtxt('../../assignments/1/SeismicResponse.txt')
sensor_data = data[:, 0]  # First accelerometer (a1)

# Time domain parameters
dt = 0.01  # sampling period in seconds
n_samples = len(sensor_data)
time = np.arange(n_samples) * dt

# Compute FFT
fft_result = np.fft.fft(sensor_data)
frequencies = np.fft.fftfreq(n_samples, dt)

# Only take positive frequencies (signal is real)
pos_mask = frequencies >= 0
pos_frequencies = frequencies[pos_mask]
pos_amplitude = np.abs(fft_result[pos_mask]) / n_samples * 2  # Normalize

# Create side-by-side plots
fig, axes = plt.subplots(1, 2, figsize=(12, 4))

# Time domain plot
axes[0].plot(time, sensor_data, linewidth=0.5)
axes[0].set_xlabel('Time (s)')
axes[0].set_ylabel('Acceleration (m/s²)')
axes[0].set_title('Time Domain: 5,000 samples')
axes[0].grid(True, alpha=0.3)

# Frequency domain plot
axes[1].plot(pos_frequencies, pos_amplitude, linewidth=0.5)
axes[1].set_xlabel('Frequency (Hz)')
axes[1].set_ylabel('Amplitude (m/s²)')
axes[1].set_title('Frequency Domain: also 5,000 values')
axes[1].set_xlim(0, 20)  # Focus on low frequencies where building modes are
axes[1].grid(True, alpha=0.3)

plt.tight_layout()
plt.show()
```

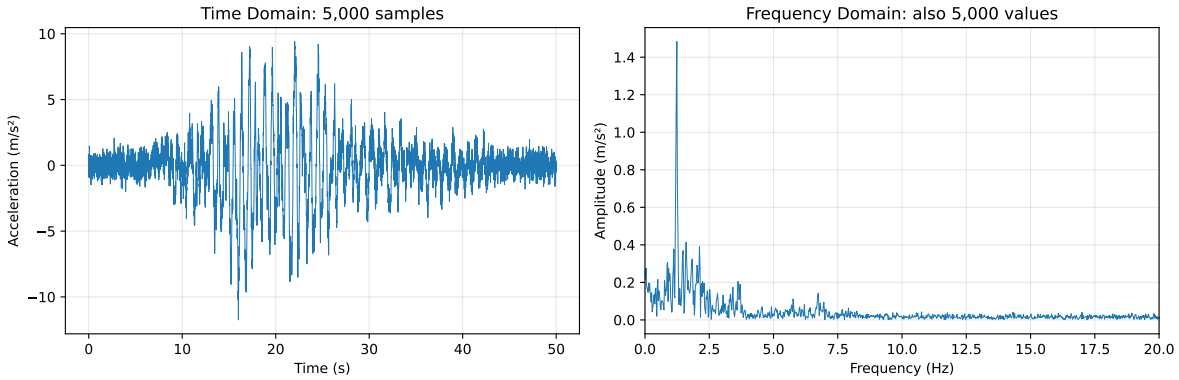

Figure 14.1: Comparison of time-domain and frequency-domain representations of accelerometer data

Notice something important in the frequency plot: **most of the energy is concentrated in just a few frequency bands** (the peaks). This is typical for structural vibration data—buildings have specific resonant frequencies (natural modes) that dominate the response.

14.2.2.2 From Representation to Compression

This observation suggests a compression strategy:

1. Transform the signal to the frequency domain (using FFT)
2. Identify the dominant peaks (frequencies with significant amplitude)
3. Store only those frequency-amplitude pairs, discarding the rest
4. To reconstruct, use the inverse FFT with only the kept components

For the signal above, instead of storing 5,000 time samples, we might store just 10-20 frequency-amplitude pairs—a compression ratio of 250:1 or better, with minimal loss of information relevant to structural analysis.

This approach works well when:

- The signal is composed of a few dominant frequencies (like building vibrations, musical notes, or periodic processes)
- The discarded components aren't important for your application (e.g., high-frequency noise in structural monitoring)

14.2.3 Feature Extraction: A Different Kind of Compression

Sometimes you don't need to preserve the full signal at all—you just need to extract **specific features** that answer your questions.

Consider sensor networks monitoring infrastructure. Most of the time, nothing interesting happens. Sensors might record data continuously, but transmitting and storing all of it would be expensive and wasteful. Instead, many systems use **event-triggered** approaches:

- Record data continuously in a local buffer
- Compute summary statistics (mean, standard deviation, peak value) over short windows
- Only transmit/store data when something unusual happens

For example, accelerometers monitoring seismic activity might:

1. Compute the standard deviation of acceleration over 1-second windows
2. If $\sigma > 0.1$ m/s^2, flag an event and transmit the raw data
3. Otherwise, just store the summary statistics

This is **feature extraction**: we derive quantities that capture the essential characteristics of the data. The compression is extreme—instead of thousands of samples, we might store just a few numbers per time window. But this only works when we know in advance what features matter for our application.

The tradeoff is clear:

Approach	Compression	Reconstructability	Use Case
Lossless (e.g., zip)	Moderate	Perfect	Archival, legal records
Frequency-domain	High	Approximate	Audio, images, signals
Feature extraction	Very high	None (features only)	Monitoring, anomaly detection

14.2.4 Huffman Coding: Lossless Compression for Text

The previous sections discussed lossy compression—throwing away information we don't need. But sometimes we need **lossless** compression: we want smaller files, but we must be able to reconstruct the original exactly. Huffman coding is a classic algorithm that achieves this for text (and other symbol sequences).

Let's work through it with a concrete example: compressing the string **"life liberty the pursuit of happiness"** (37 characters).

14.2.4.1 Attempt 1: Fixed-Length Encoding

The simplest approach is to assign each unique character a fixed-length binary code. First, let's identify the unique characters:

`l, i, f, e, (space), b, r, t, y, h, p, u, s, o, a, n`

That's **16 unique characters**. To represent 16 different values, we need $\lceil \log_2(16) \rceil = 4$ bits per character.

Character	Code	Character	Code
l	0000	h	1000
i	0001	p	1001
f	0010	u	1010
e	0011	s	1011
(space)	0100	o	1100
b	0101	a	1101
r	0110	n	1110
t	0111	y	1111

Total size: 37 characters $\times$ 4 bits = **148 bits**

This is already better than ASCII (which would use $37 \times 8 = 296$ bits), but can we do better?

14.2.4.2 Attempt 2: Variable-Length Encoding

Look at the character frequencies in our string:

Character	Count	Character	Count
(space)	5	f	2
e	4	h	2
i	4	l	2
p	3	r	2
s	3	u	2
t	3	a	1
		b	1
		n	1
		o	1
		y	1

Space appears 5 times, but 'y' appears only once. What if we used **shorter codes for frequent characters** and longer codes for rare ones? If space used only 2 bits instead of 4, we'd save $5 \times 2 = 10$ bits just from that one character!

But there's a catch: with variable-length codes, how do we know where one code ends and the next begins? Consider if we assigned:

- space = 0
- e = 01

Then the sequence "01" could mean "space, something starting with 1" or just "e". This ambiguity breaks decoding.

The solution is to use **prefix-free codes**: no code can be the beginning (prefix) of another code. For example:

- 0 and 01 are NOT both allowed (0 is a prefix of 01)
- 10 and 01 ARE both allowed (neither is a prefix of the other)

14.2.4.3 The Huffman Algorithm

David Huffman discovered an elegant algorithm that constructs the optimal prefix-free code for any character frequency distribution. Here's how it works:

1. **Create a leaf node** for each character, labeled with its frequency
2. **Build a tree** by repeatedly:
    - Find the two nodes with the smallest frequencies
    - Create a new parent node with frequency = sum of the two children
    - Remove the two children from the list, add the parent
3. **Repeat** until only one node remains (the root)
4. **Assign codes** by traversing the tree: left = 0, right = 1

Let's apply this to our string. Starting with nodes sorted by frequency:

```
a:1, b:1, n:1, o:1, y:1, f:2, h:2, l:2, r:2, u:2, p:3, s:3, t:3, e:4, i:4, _:5
```

(using _ for space)

Building the tree step by step:

1. Combine a(1) + b(1) → node(2)
2. Combine n(1) + o(1) → node(2)
3. Combine y(1) + node(2) from step 1 → node(3)

4. Continue combining the two smallest nodes at each step…
5. Eventually all merge into a single root with frequency 37

The resulting Huffman tree produces these codes:

Character	Huffman Code	Bits	Count	Total Bits
(space)	101	3	5	15
e	001	3	4	12
i	000	3	4	12
p	1101	4	3	12
s	1110	4	3	12
t	1100	4	3	12
f	0100	4	2	8
h	0110	4	2	8
r	0101	4	2	8
u	0111	4	2	8
l	11111	5	2	10
a	10011	5	1	5
b	10000	5	1	5
n	11110	5	1	5
o	10010	5	1	5
y	10001	5	1	5

Total Huffman size: **142 bits**

(Note: The exact codes depend on tie-breaking choices during tree construction. Different valid Huffman trees may produce slightly different codes, but all achieve the same optimal total bit count.)

14.2.4.4 Comparing Results

Encoding	Bits	Compression vs ASCII
ASCII (8 bits/char)	296	—
Fixed 4-bit encoding	148	50%
Huffman encoding	142	52%

Huffman coding saved **4%** compared to fixed-length encoding, and **52%** compared to ASCII!

The savings come from the principle: **frequent characters get short codes, rare characters get long codes**. The average code length is minimized.

14.2.4.5 Important Properties of Huffman Codes

1. **Prefix-free**: No code is a prefix of another, so decoding is unambiguous
2. **Optimal**: For a given frequency distribution, no prefix-free code uses fewer bits on average
3. **Requires the codebook**: To decode, you need either the Huffman tree or the code table

The need to store/transmit the codebook is overhead that reduces the benefit for short messages. Huffman coding shines when:

- The message is long (codebook overhead is amortized)
- Character frequencies are skewed (some characters much more common than others)
- Lossless reconstruction is required

14.2.5 Other Compression Algorithms and Applications

We've covered the key concepts: character encoding, frequency-domain transformations, feature extraction, and Huffman coding. These ideas appear throughout the field of data compression. Here are a few more algorithms worth knowing about:

14.2.5.1 Run-Length Encoding (RLE)

Run-length encoding is perhaps the simplest compression algorithm. Instead of storing each symbol individually, RLE counts consecutive repetitions:

Original: `BBWWWWBBBBBBWBBBBBBBBWW` (23 characters)

RLE encoded: `2B4W6B1W8B2W` (12 characters)

The encoding reads: "2 B's, then 4 W's, then 6 B's, then 1 W, then 8 B's, then 2 W's."

RLE works well when data has long runs of identical values—which is common in:

- **Binary images** (fax documents, simple graphics): Large regions of black or white
- **Sparse data**: Sensor readings that are mostly zero with occasional spikes
- **Database columns**: Sorted data where adjacent rows often have the same value

RLE performs poorly on data without repetition. For the string "abcdefgh", RLE would produce "1a1b1c1d1e1f1g1h"—actually longer than the original!

14.2.5.2 JPEG Image Compression

JPEG is the dominant format for photographs, achieving 10:1 compression or better with minimal visible quality loss. It combines several techniques we've discussed:

1. **Color space conversion**: RGB colors are converted to YCbCr (luminance + chrominance). Human vision is more sensitive to brightness than color, so chrominance channels can be compressed more aggressively.
2. **Block processing**: The image is divided into 8×8 pixel blocks.
3. **Discrete Cosine Transform (DCT)**: Each block is transformed to the frequency domain (similar to FFT, but using cosines). Low-frequency components represent smooth gradients; high-frequency components represent sharp edges and fine detail.
4. **Quantization** (the lossy step): DCT coefficients are divided by a quantization matrix and rounded. This discards high-frequency detail that humans don't easily perceive. Higher compression = more aggressive quantization = more visible artifacts.
5. **Entropy coding**: The quantized coefficients are compressed using RLE (many coefficients become zero after quantization) and Huffman coding.

The "quality" setting in JPEG controls the quantization step—lower quality means more coefficients are rounded to zero, smaller files, but more visible blocking artifacts.

14.2.5.3 A Provocative Thought: LLMs as Compression?

Here's an interesting perspective: language models like GPT or Claude can be viewed as compression algorithms.

Consider: a well-trained language model can predict the next word in a sentence with high accuracy. If you can predict what comes next, you don't need to store it—you only need to store the *surprises* (the deviations from prediction).

This is exactly how modern compression algorithms like **arithmetic coding** work: they assign shorter codes to more predictable symbols. A language model that accurately predicts "the" after "in" can compress that sequence very efficiently.

In fact, researchers have shown that large language models achieve state-of-the-art compression ratios on text—they're implicitly learning a very sophisticated model of language structure that enables extreme compression.

This connection between prediction and compression is deep: **compression is prediction, and prediction is compression**. The better you understand your data's structure, the more efficiently you can represent it.

15 Module 9: Database Design Practicum

15.1 Module Overview

15.1.1 Learning Objectives

By the end of this module, students will be able to:

- Be able to take a description of a dataset and convert it into an E/R model of it.
- Be able to translate the E/R diagram to a relational database schema.
- Be able to write an SQL script to implement the schema in SQLite.
- Be able to populate an existing databse with data contained in CSV files using SQL statements.
- Be able to create, read, update and delete (CRUD) data in the SQLite database using SQL.
- Understand that SQL statements (and the database schema under it) can be abstracted into higher-level structures such as a RESTful API, or a web application.
- Understand that higher levels of abstraction are possible, particularly through the use of LLMs and their ability to parse natural language.

15.1.2 Topics Covered

- ER Diagrams in Practice
- The Relational Model
- Implementing a Relational Database
- Data Ingestion
- CRUD Using SQL
- CRUD using a Web Framework
- CRUD using Natural Language

15.1.3 Project Milestones

Define the structure of your final report, and set milestones for completing each section.

15.2 Lecture Notes

15.2.1 The Problem: Connecting Thermal Comfort and Energy Use

Throughout this module, we will work through a complete database design and implementation problem: **understanding the relationship between thermal comfort and energy consumption in a residential building**.

15.2.1.1 Problem Statement

Imagine you want to answer the question: *What is the energy use required to keep occupants comfortable, and how does it change by season of the year?*

To answer this, you have access to multiple data sources:

1. **Ecobee Thermostat**: Provides temperature readings, humidity, HVAC mode (heating, cooling, fan-only), and runtime data for different zones in the house
2. **eGauge Power Meter**: Monitors electrical consumption across all circuits in the electrical panel, including HVAC equipment
3. **Smart Watches**: Each occupant wears a device that:
    - Tracks skin temperature continuously
    - Periodically prompts for thermal comfort ratings using the PMV (Predicted Mean Vote) scale (-3 to +3, where -3 is cold, 0 is neutral, and +3 is hot)
    - Detects which room the occupant is currently in
4. **Weather Data**: Outside temperature, humidity, and other environmental conditions from a nearby weather station

15.2.1.2 The Question We Want to Answer

We want to determine the energy consumption required to maintain thermal comfort. For this analysis, we'll define "comfortable" as having a mean PMV rating across all occupants between -0.5 and +0.5.

Specifically, we want to understand: - How much energy is consumed during periods when occupants are comfortable vs. uncomfortable? - How does this relationship vary by season? - What thermostat settings and HVAC modes are associated with comfort at different times of year?

15.2.1.3 What We'll Build

To answer these questions, we will:

1. Design an Entity-Relationship (E/R) diagram capturing all relevant entities and their relationships
2. Transform this E/R diagram into a relational database schema
3. Implement the schema in SQLite using SQL DDL statements
4. Create sample data representing one year of measurements
5. Populate the database by ingesting data from CSV files
6. Write SQL queries to perform CRUD operations and answer our analytical questions
7. Build a simple Flask web application that provides a RESTful API for database access
8. Demonstrate how natural language interfaces (via LLMs) can simplify database interactions

This hands-on example will tie together all the concepts from previous modules and demonstrate how database design supports real-world data analysis in civil and environmental engineering.

15.2.2 ER Diagrams in Practice

Creating an Entity-Relationship (E/R) diagram is the first step in database design. The goal is to identify the **entities** (things we want to store information about), their **attributes** (properties of those things), and the **relationships** between them.

15.2.2.1 Step 1: Identify Entities from the Problem Description

Looking back at our problem statement, we're collecting data from multiple sources. Each distinct "thing" we're measuring or tracking is a candidate entity:

1. **Occupants** - The people living in the house who report comfort levels
2. **Rooms** - The physical spaces where comfort is measured
3. **Comfort Ratings** - Individual comfort assessments from occupants
4. **HVAC Readings** - Thermostat and heating/cooling system measurements
5. **Power Readings** - Electrical consumption measurements
6. **Weather Readings** - Outside environmental conditions

Each of these will become a table in our database.

15.2.2.2 Step 2: Identify Attributes

For each entity, we need to determine what information to store. Let's work through each one:

Occupant:

- `occupant_id` (primary key): Unique identifier for each person
- `occupant_name`: The person's name

Room:

- `room_id` (primary key): Unique identifier for each room
- `room_name`: The room's name (e.g., "Living Room", "Kitchen")

ComfortRating:

- `comfort_id` (primary key): Unique identifier for each rating
- `pmv_rating`: The PMV (Predicted Mean Vote) score from -3 to +3
- `skin_temp`: Skin temperature in Celsius
- `comfort_timestamp`: When the rating was recorded
- Foreign keys (relationships): `occupant_id`, `room_id`

HVACReading:

- `hvac_id` (primary key): Unique identifier for each reading
- `heating_runtime`: Minutes of heating in the measurement period
- `cooling_runtime`: Minutes of cooling in the measurement period
- `fan_state`: Whether the fan is on or off
- `temp_setpoint`: Target temperature setting
- `thermostat_temp`: Actual measured temperature
- `hvac_timestamp`: When the reading was recorded

PowerReading:

- `power_id` (primary key): Unique identifier for each reading
- `power_consumption`: Power consumption in kilowatts
- `power_timestamp`: When the reading was recorded

WeatherReading:

- `weather_id` (primary key): Unique identifier for each reading
- `outside_temp`: Outside temperature in Fahrenheit
- `outside_humidity`: Outside relative humidity percentage
- `weather_timestamp`: When the reading was recorded

15.2.2.3 Step 3: Identify Relationships

Now we determine how entities connect to each other:

1. **Occupant → ComfortRating**: One-to-Many
    - Each occupant can submit many comfort ratings over time
    - Each comfort rating belongs to exactly one occupant
2. **Room → ComfortRating**: One-to-Many
    - Each room can have many comfort ratings (from different occupants at different times)
    - Each comfort rating is associated with exactly one room

15.2.2.4 Step 4: Design Decisions

There is no single correct way of desigining this E/R diagram. Every design is just a set of choices we make and each of them has pros and cons. For example, notice some choices we did not make:

- **No explicit time hierarchy entities** (Hour, Day, Season, Year): Instead, we use timestamps and can extract temporal information using SQL date functions when needed. This is simpler and more flexible.
- **No direct relationships between readings**: HVAC, Power, and Weather readings are independent time series. We'll join them based on timestamps when analyzing the data.

But it can be argued that this was a pragmatic design choice: we're optimizing for **simplicity** and **query flexibility** rather than enforcing rigid temporal structures in the schema.

15.2.2.5 The Complete ER Diagram

Here's our simplified ER diagram:

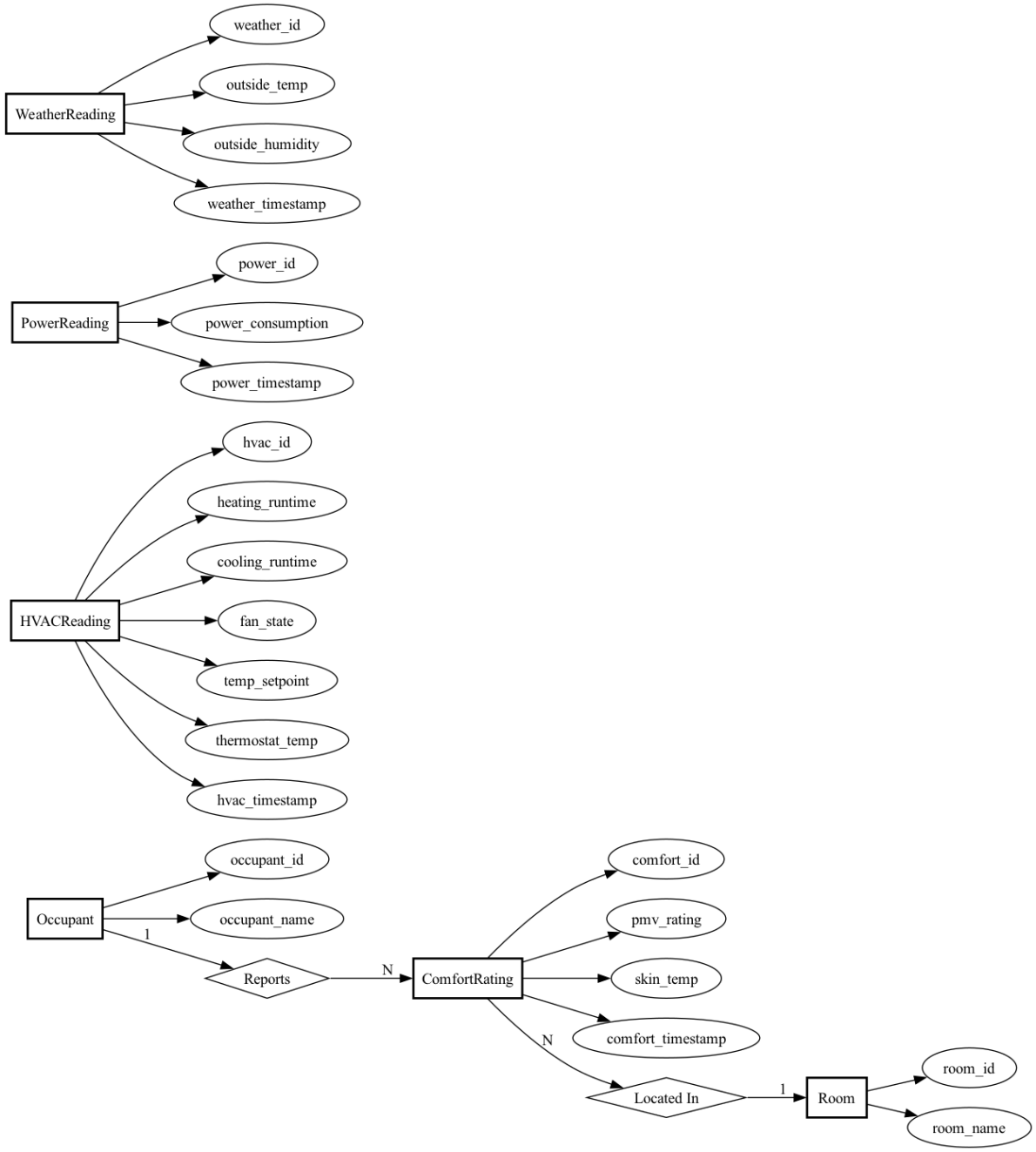

Figure 15.1: Simplified ER Diagram

Key features of this design:

- **Clean separation of concerns**: Each entity represents one type of measurement or

dimension
- **Timestamp-based**: All time-series data uses timestamps rather than complex time hierarchies
- **Foreign keys enforce integrity**: Comfort ratings must reference valid occupants and rooms
- **Ready for time-based joins**: The timestamp fields let us correlate readings across tables using SQL date/time functions

This design gives us the flexibility to answer questions like:

- "What was the average comfort level when the HVAC was in heating mode?"
- "How does power consumption correlate with outside temperature by season?"
- "Which rooms have the most comfort complaints?"

All without needing to pre-define how we'll slice time (by hour, day, week, season, etc.).

15.2.3 The Relational Model

Now that we have our ER diagram, let's translate it into a relational schema. You've seen the theory in previous modules—here's how we apply it to our thermal comfort problem.

15.2.3.1 Translating the ER Diagram

Following the standard translation rules, each entity becomes a table and relationships become foreign keys. Here's our complete schema:

Dimension Tables:

```
-- Occupant table
CREATE TABLE Occupant (
    occupant_id INTEGER PRIMARY KEY,
    occupant_name TEXT NOT NULL
);

-- Room table
CREATE TABLE Room (
    room_id INTEGER PRIMARY KEY,
    room_name TEXT NOT NULL
);
```

Fact Tables:

```sql
-- ComfortRating table
CREATE TABLE ComfortRating (
    comfort_id INTEGER PRIMARY KEY,
    occupant_id INTEGER NOT NULL,
    room_id INTEGER NOT NULL,
    pmv_rating REAL NOT NULL,
    skin_temp REAL NOT NULL,
    comfort_timestamp DATETIME NOT NULL,
    FOREIGN KEY (occupant_id) REFERENCES Occupant(occupant_id),
    FOREIGN KEY (room_id) REFERENCES Room(room_id)
);

-- HVACReading table
CREATE TABLE HVACReading (
    hvac_id INTEGER PRIMARY KEY,
    heating_runtime REAL NOT NULL,
    cooling_runtime REAL NOT NULL,
    fan_state TEXT NOT NULL,
    temp_setpoint REAL NOT NULL,
    thermostat_temp REAL NOT NULL,
    hvac_timestamp DATETIME NOT NULL
);

-- PowerReading table
CREATE TABLE PowerReading (
    power_id INTEGER PRIMARY KEY,
    power_consumption REAL NOT NULL,
    power_timestamp DATETIME NOT NULL
);

-- WeatherReading table
CREATE TABLE WeatherReading (
    weather_id INTEGER PRIMARY KEY,
    outside_temp REAL NOT NULL,
    outside_humidity REAL NOT NULL,
    weather_timestamp DATETIME NOT NULL
);
```

15.2.3.2 What Happened to the Relationships?

You learned that relationships can be translated into tables with foreign keys to the participating entities. Our ER diagram shows two relationships:

- **Reports**: Occupant → ComfortRating (1:N)
- **LocatedIn**: Room → ComfortRating (1:N)

We *could* have created separate `Reports` and `LocatedIn` tables, each with foreign keys. But for **one-to-many relationships**, there's a more efficient approach: just add the foreign key directly to the "many" side.

Instead of:

```
-- Less efficient approach
CREATE TABLE Reports (
    occupant_id INTEGER,
    comfort_id INTEGER,
    FOREIGN KEY (occupant_id) REFERENCES Occupant(occupant_id),
    FOREIGN KEY (comfort_id) REFERENCES ComfortRating(comfort_id)
);
```

We do:

```
-- More efficient approach
CREATE TABLE ComfortRating (
    comfort_id INTEGER PRIMARY KEY,
    occupant_id INTEGER NOT NULL,  -- Foreign key embedded here
    ...
);
```

This avoids an extra table and extra joins. Each `ComfortRating` row directly stores which occupant reported it and which room it's for.

When would you create a separate relationship table? For **many-to-many relationships**. For example, if we tracked "which occupants are authorized to adjust settings in which rooms" (many occupants, many rooms, many-to-many), we'd need a separate `RoomAccess` table with `(occupant_id, room_id)` pairs.

15.2.3.3 Design Choices Worth Noting

Why separate dimension and fact tables? This follows the star schema pattern. Dimensions (Occupant, Room) are relatively static, while facts (readings, ratings) are high-volume time series. This separation makes queries efficient and updates logical.

Why no time hierarchy tables? We considered adding Hour, Day, Season, and Year tables. But for this application, using timestamps and SQL date functions is simpler and more flexible. We can extract temporal groupings in queries without pre-defining the hierarchy in the schema.

Normalization? The schema is in 3NF: no redundancy, no transitive dependencies. For example, `occupant_name` lives only in the `Occupant` table, not duplicated in every `ComfortRating` row.

Foreign keys in fact tables only? Notice that `ComfortRating` has foreign keys to `Occupant` and `Room` because there's a semantic relationship: comfort ratings are reported by occupants in specific rooms. But `HVACReading`, `PowerReading`, and `WeatherReading` have no foreign keys—they're independent time series we'll correlate via timestamps in queries.

15.2.3.4 What This Schema Lets Us Do

This design supports the analytical queries we need:

- Join comfort ratings with HVAC/power/weather data based on matching timestamps
- Filter by occupant or room using the foreign key relationships
- Extract temporal patterns (by hour, day, season) using SQL date functions on timestamps
- Aggregate measurements across different dimensions

In the next sections, we'll implement this schema in SQLite and populate it with a year's worth of synthetic data.

15.2.4 Implementing a Relational Database

With our schema designed, it's time to create the actual database. This section covers executing the DDL statements in SQLite and understanding SQLite-specific implementation details.

15.2.4.1 Creating the Database

We've saved our schema in a file called `schema.sql`. To create the database:

```
sqlite3 comfort-energy.db < schema.sql
```

This creates a file called `comfort-energy.db` with all six tables. The file starts at 32KB (SQLite's minimum file size) and will grow as we add data.

15.2.4.2 SQLite-Specific Implementation Details

INTEGER PRIMARY KEY Auto-increment:

In SQLite, `INTEGER PRIMARY KEY` is special—it's an alias for the built-in `ROWID`. This means:

- SQLite automatically assigns sequential IDs if you don't provide them
- No need for `AUTOINCREMENT` keyword (and you shouldn't use it—it's slower and rarely necessary)
- IDs are guaranteed unique and stable

Data Type Storage:

SQLite uses dynamic typing, but our column type declarations provide storage hints:

- `INTEGER`: Stored as 1, 2, 3, 4, 6, or 8 bytes depending on value magnitude
- `REAL`: 8-byte IEEE floating point (equivalent to `DOUBLE` in other databases)
- `TEXT`: Variable-length UTF-8 or UTF-16 encoded string
- `DATETIME`: Not a real type—SQLite stores it as `TEXT` in ISO 8601 format (`YYYY-MM-DD HH:MM:SS`)

Foreign Key Enforcement:

This is critical: SQLite supports foreign key constraints, but they're **disabled by default** for backward compatibility. You must enable them for each database connection:

```
PRAGMA foreign_keys = ON;
```

Without this, SQLite will silently accept invalid foreign key values (e.g., a `ComfortRating` referencing a non-existent `occupant_id`). Always enable this pragma when working with databases that use referential integrity.

15.2.4.3 Verifying the Database Structure

Let's verify the database was created correctly. First, list all tables:

```
sqlite3 comfort-energy.db ".tables"
```

Output:

```
ComfortRating  HVACReading    Occupant       PowerReading   Room           WeatherReading
```

All six tables exist. Now check the full schema:

```
sqlite3 comfort-energy.db ".schema"
```

This outputs the complete DDL, confirming our table definitions match the design. The database structure is now ready for data.

15.2.5 Data Ingestion

With an empty database structure in place, we need to populate it with data. In practice, this data would come from real sensors (ecobee, eGauge, smart watches, weather APIs). For this module, we've generated one year of realistic synthetic data.

15.2.5.1 Generating Synthetic Data

We created a Python script (`generate_data.py`) that produces CSV files with realistic patterns:

- **4 occupants**: Alice Johnson, Bob Martinez, Carol Chen, David Kim
- **6 rooms**: Living Room, Kitchen, Master Bedroom, Bedroom 2, Office, Basement
- **Weather readings**: 8,784 rows (hourly for 366 days in 2024)
- **HVAC readings**: 105,408 rows (every 5 minutes)
- **Power readings**: 105,408 rows (every 5 minutes, correlated with HVAC)
- **Comfort ratings**: 5,096 rows (2-5 random surveys per occupant per day)

The synthetic data includes realistic patterns:

- Seasonal temperature variations (Pittsburgh climate)
- HVAC mode switching based on outside temperature and setpoint
- Power consumption correlated with HVAC runtime (heating ~3.5kW, cooling ~4.0kW when active)

- PMV ratings mostly near 0 (comfortable) with occasional outliers (20% uncomfortable)
- Skin temperature correlated with comfort ratings

Here's a sample of the comfort ratings CSV:

```
comfort_id,occupant_id,room_id,pmv_rating,skin_temp,comfort_timestamp
1,1,3,0.14,33.33,2024-01-01 17:53:00
2,1,5,-0.03,33.1,2024-01-01 17:51:00
3,2,3,-2.42,31.97,2024-01-01 21:08:00
```

15.2.5.2 Loading Data with SQLite's .import Command

SQLite provides a `.import` command for bulk loading CSV files. Our `database-loader.sql` script automates this:

```
-- Enable foreign key constraints
PRAGMA foreign_keys = ON;

-- Set CSV mode
.mode csv

-- Import each table
.import occupants.csv Occupant
.import rooms.csv Room
.import weather_readings.csv WeatherReading
.import hvac_readings.csv HVACReading
.import power_readings.csv PowerReading
.import comfort_ratings.csv ComfortRating

-- Remove header rows that got imported as data
DELETE FROM WeatherReading WHERE weather_id = 'weather_id';
DELETE FROM HVACReading WHERE hvac_id = 'hvac_id';
DELETE FROM PowerReading WHERE power_id = 'power_id';
DELETE FROM ComfortRating WHERE comfort_id = 'comfort_id';
```

Important details:

1. **Foreign key order matters**: We import `Occupant` and `Room` first because `ComfortRating` has foreign keys referencing them. With `PRAGMA foreign_keys = ON`, SQLite would reject comfort ratings referencing non-existent occupants.
2. **Header row handling**: SQLite's `.import` command treats the first row as data, not a header. We delete rows where the ID equals the column name string to remove these.

3. **CSV mode**: `.mode csv` tells SQLite to expect comma-separated values with proper quote handling.

Run the import:

```
sqlite3 comfort-energy.db < database-loader.sql
```

15.2.5.3 Verifying the Data Load

After importing, verify the row counts:

```
SELECT 'Occupants:' as table_name, COUNT(*) as count FROM Occupant
UNION ALL SELECT 'Rooms:', COUNT(*) FROM Room
UNION ALL SELECT 'ComfortRatings:', COUNT(*) FROM ComfortRating
UNION ALL SELECT 'HVACReadings:', COUNT(*) FROM HVACReading
UNION ALL SELECT 'PowerReadings:', COUNT(*) FROM PowerReading
UNION ALL SELECT 'WeatherReadings:', COUNT(*) FROM WeatherReading;
```

Output:

```
Occupants:|4
Rooms:|6
ComfortRatings:|5096
HVACReadings:|105408
PowerReadings:|105408
WeatherReadings:|8784
```

Let's spot-check the data looks correct:

```
SELECT * FROM Occupant;
```

Output:

```
1|Alice Johnson
2|Bob Martinez
3|Carol Chen
4|David Kim
```

And a sample comfort rating:

```
SELECT * FROM ComfortRating LIMIT 3;
```

Output:

```
1|1|3|0.14|33.33|2024-01-01 17:53:00
2|1|5|-0.03|33.1|2024-01-01 17:51:00
3|2|3|-2.42|31.97|2024-01-01 21:08:00
```

The data is loaded correctly. We now have a year's worth of thermal comfort and energy data ready for analysis.

15.2.6 CRUD Using SQL

With our database populated, we can now perform Create, Read, Update, and Delete (CRUD) operations using SQL. This section demonstrates both basic CRUD operations and analytical queries that answer our research question about thermal comfort and energy use.

15.2.6.1 CREATE: Inserting New Data

Add a new room to the database:

```
INSERT INTO Room (room_name) VALUES ('Guest Room');
```

SQLite automatically assigns `room_id = 7` (the next available ID). Verify:

```
SELECT * FROM Room;
```

Output:

```
1|Living Room
2|Kitchen
3|Master Bedroom
4|Bedroom 2
5|Office
6|Basement
7|Guest Room
```

Add a new comfort rating:

```
INSERT INTO ComfortRating (occupant_id, room_id, pmv_rating, skin_temp, comfort_timestamp)
VALUES (1, 3, -0.5, 32.8, '2024-12-01 14:30:00');
```

This records that Alice Johnson (occupant 1) felt slightly cool (PMV = -0.5) in the Master Bedroom at 2:30 PM on December 1st.

15.2.6.2 READ: Querying Data

Simple query: Find all comfort ratings for a specific occupant:

```
SELECT * FROM ComfortRating
WHERE occupant_id = 1
LIMIT 5;
```

Analytical query 1: Energy use by season

This answers part of our main question: how does energy consumption vary by season?

```
SELECT
    CASE
        WHEN CAST(strftime('%m', power_timestamp) AS INTEGER) IN (12, 1, 2) THEN 'Winter'
        WHEN CAST(strftime('%m', power_timestamp) AS INTEGER) IN (3, 4, 5) THEN 'Spring'
        WHEN CAST(strftime('%m', power_timestamp) AS INTEGER) IN (6, 7, 8) THEN 'Summer'
        ELSE 'Fall'
    END as season,
    ROUND(AVG(power_consumption), 2) as avg_power_kw,
    ROUND(SUM(power_consumption * 5.0 / 60.0), 2) as total_kwh
FROM PowerReading
GROUP BY season
ORDER BY
    CASE season
        WHEN 'Winter' THEN 1
        WHEN 'Spring' THEN 2
        WHEN 'Summer' THEN 3
        ELSE 4
    END;
```

Output:

```
Winter|4.30|9384.88
Spring|3.41|7532.38
Summer|2.30|5089.12
Fall|3.40|7428.09
```

Key insight: Winter energy use is 84% higher than summer, primarily due to heating loads in Pittsburgh's cold climate.

Analytical query 2: Comfort levels by room

Which rooms have the most comfort complaints?

```
SELECT
    r.room_name,
    COUNT(*) as total_ratings,
    ROUND(AVG(cr.pmv_rating), 2) as avg_pmv,
    SUM(CASE WHEN cr.pmv_rating < -0.5 OR cr.pmv_rating > 0.5 THEN 1 ELSE 0 END) as uncomfor
    ROUND(100.0 * SUM(CASE WHEN cr.pmv_rating < -0.5 OR cr.pmv_rating > 0.5 THEN 1 ELSE 0 EN
FROM ComfortRating cr
JOIN Room r ON cr.room_id = r.room_id
GROUP BY r.room_name
ORDER BY pct_uncomfortable DESC;
```

Output:

```
Living Room|841|0.03|253|30.1
Basement|845|-0.01|247|29.2
Bedroom 2|842|0.01|230|27.3
Office|874|0.01|235|26.9
Kitchen|839|0.0|221|26.3
Master Bedroom|855|0.02|210|24.6
```

Key insight: The Living Room has the highest discomfort rate (30.1%), while the Master Bedroom is most comfortable (24.6% uncomfortable). This might indicate HVAC zoning issues or different usage patterns.

Analytical query 3: Overall comfort statistics

```
SELECT
    ROUND(AVG(pmv_rating), 3) as overall_avg_pmv,
    ROUND(100.0 * SUM(CASE WHEN pmv_rating BETWEEN -0.5 AND 0.5 THEN 1 ELSE 0 END) / COUNT(*
FROM ComfortRating;
```

Output:

```
0.012|72.6
```

Key insight: On average, PMV is nearly neutral (0.012), and occupants are comfortable 72.6% of the time. This aligns with our data generation assumptions (80% comfortable with some variation).

Analytical query 4: Daily energy consumption

```
SELECT
    DATE(power_timestamp) as date,
    ROUND(SUM(power_consumption * 5.0 / 60.0), 2) as daily_kwh
FROM PowerReading
GROUP BY DATE(power_timestamp)
ORDER BY date
LIMIT 10;
```

Output:

```
2024-01-01|103.51
2024-01-02|103.1
2024-01-03|103.52
2024-01-04|104.16
2024-01-05|101.55
2024-01-06|103.55
2024-01-07|102.29
2024-01-08|100.89
2024-01-09|103.59
2024-01-10|102.95
```

Daily winter consumption averages ~103 kWh. Summer days (not shown) average ~70 kWh.

15.2.6.3 UPDATE: Modifying Existing Data

Rename a room (maybe "Office" becomes "Home Office"):

```
UPDATE Room
SET room_name = 'Home Office'
WHERE room_id = 5;
```

Verify the change:

```
SELECT * FROM Room WHERE room_id = 5;
```

Output:

```
5|Home Office
```

Correct an erroneous comfort rating:

```
UPDATE ComfortRating
SET pmv_rating = 0.0
WHERE comfort_id = 100 AND pmv_rating > 2.0;
```

This might be used if a sensor malfunction reported unrealistic values.

15.2.6.4 DELETE: Removing Data

Delete the guest room we added earlier:

```
DELETE FROM Room WHERE room_id = 7;
```

Verify it's gone:

```
SELECT COUNT(*) as room_count FROM Room;
```

Output:

```
6
```

Delete comfort ratings from a specific day (perhaps sensor was broken):

```
DELETE FROM ComfortRating
WHERE DATE(comfort_timestamp) = '2024-01-15';
```

Important: Be careful with DELETE. Foreign key constraints protect you from deleting referenced data. For example, trying to delete an occupant who has comfort ratings would fail (if we had `ON DELETE RESTRICT` set), or would cascade delete all their ratings (if we had `ON DELETE CASCADE`). Our current schema doesn't specify, so SQLite allows the delete but leaves orphaned comfort ratings—which violates referential integrity.

15.2.6.5 Answering the Main Question

Combining these queries, we can now answer: **What is the energy use required to keep occupants comfortable, and how does it change by season?**

From our analysis:

- **Winter comfort** requires ~103 kWh/day (~9,385 kWh over winter months)
- **Summer comfort** requires ~70 kWh/day (~5,089 kWh over summer months)
- **Overall**, occupants are comfortable 72.6% of the time with mean PMV near neutral
- **Spatial variation**: Comfort varies by room, with Living Room being least comfortable

The higher winter energy use reflects Pittsburgh's heating-dominated climate. More sophisticated queries could correlate specific comfort periods with power consumption to isolate the "cost of comfort" more precisely.

15.2.7 CRUD using a Web Framework

In the previous section, we interacted with our database directly through SQL commands. While this is powerful, it's not practical for end users who need a more intuitive interface. Web frameworks let us abstract SQL operations behind user-friendly forms and pages.

We'll use **Flask**, a lightweight Python web framework, to build a simple web interface for our thermal comfort database. The goal is to show how the exact SQL queries we wrote earlier can be triggered through a web browser.

15.2.7.1 Conceptual Overview: From SQL to Web Interface

The transformation is straightforward:

SQL Operation	Web Interface
`SELECT * FROM Room`	Web page displaying table of rooms
`INSERT INTO Room VALUES (...)`	HTML form with "Add Room" button
`UPDATE Room SET ...`	Edit form with room name input
`DELETE FROM Room WHERE ...`	Delete button with confirmation page

The web framework acts as a **thin layer** between the browser and the database:

1. User clicks a link or submits a form in their browser
2. Flask receives the HTTP request
3. Flask executes the corresponding SQL query
4. Flask formats the results as HTML and sends it back to the browser

15.2.7.2 Architecture

Our Flask application consists of:

`app.py` - Main application file (~200 lines)

- Database connection logic
- Route handlers (functions that respond to URLs)
- Each route executes SQL and renders a template

`templates/` - HTML templates (11 files)

- `base.html` - Common layout and navigation
- CRUD pages: `rooms.html`, `add_room.html`, `edit_room.html`, etc.
- Analytics pages: `energy_by_season.html`, `comfort_by_room.html`, etc.

Each page displays the SQL query it executes, creating a direct learning connection to the previous section.

15.2.7.3 Key Flask Patterns

Pattern 1: Displaying Query Results

```
@app.route('/rooms')
def list_rooms():
    """READ: List all rooms"""
    conn = get_db_connection()
    rooms = conn.execute('SELECT * FROM Room ORDER BY room_id').fetchall()
    conn.close()
    return render_template('rooms.html', rooms=rooms)
```

This maps the URL `/rooms` to a function that: 1. Connects to the database 2. Executes the same SELECT query we used before 3. Passes results to an HTML template 4. Returns the rendered page to the browser

Pattern 2: Processing Form Submissions

```
@app.route('/rooms/add', methods=['GET', 'POST'])
def add_room():
    """CREATE: Add a new room"""
    if request.method == 'POST':
        room_name = request.form['room_name']
        conn = get_db_connection()
```

```
        conn.execute('INSERT INTO Room (room_name) VALUES (?)', (room_name,))
        conn.commit()
        conn.close()
        return redirect(url_for('list_rooms'))
    return render_template('add_room.html')
```

This handles both: - **GET request**: Show the form (when user visits `/rooms/add`) - **POST request**: Process the form data and execute INSERT (when user submits)

The `?` placeholder prevents SQL injection attacks - Flask safely escapes the user input.

Pattern 3: Analytical Queries

The same analytical queries from the previous section become web pages:

```
@app.route('/analytics/energy-by-season')
def energy_by_season():
    """Analytical query: Energy consumption by season"""
    conn = get_db_connection()
    results = conn.execute('''
        SELECT
            CASE
                WHEN CAST(strftime('%m', power_timestamp) AS INTEGER) IN (12, 1, 2) THEN 'Wi
                ...
            END as season,
            ROUND(AVG(power_consumption), 2) as avg_power_kw,
            ROUND(SUM(power_consumption * 5.0 / 60.0), 2) as total_kwh
        FROM PowerReading
        GROUP BY season
        ...
    ''').fetchall()
    conn.close()
    return render_template('energy_by_season.html', results=results)
```

This is the **exact same SQL** from the previous section, just wrapped in a Flask route.

15.2.7.4 What's Implemented

The complete application includes:

CRUD Operations: - Rooms: Full CRUD (create, read, update, delete) - Comfort Ratings: Create and read with foreign key dropdowns

Analytical Queries: - Energy consumption by season - Comfort levels by room - Overall comfort statistics - Daily energy consumption

Features: - Delete operations require confirmation (as you requested) - Each page shows the SQL query being executed - Foreign key constraints are enforced - Simple, clean interface with minimal styling

15.2.7.5 Running the Application

The complete source code is available here.

To run it locally:

1. **Unzip the file you downloaded:**
   ```
   unzip comfort-app.zip
   ```
2. **Install Flask** (if not already installed):
   ```
   pip install flask
   ```
3. **Run the application:**
   ```
   python3 app.py
   ```
4. **Open your browser** to `http://127.0.0.1:5000/`

You'll see a homepage with links to all CRUD operations and analytical queries. Try:

- Adding a new room
- Viewing comfort ratings (with occupant and room names joined)
- Exploring the seasonal energy analysis
- Deleting a room (note the confirmation step)

15.2.7.6 Key Takeaways

1. **Web frameworks abstract SQL** - The browser becomes your database interface
2. **Same queries, different interface** - Every SQL query from the previous section has a web equivalent
3. **Thin abstraction layer** - Flask adds minimal code between HTTP requests and SQL execution
4. **User-friendly** - End users don't need to know SQL to interact with the data

This pattern scales: production applications use the same concepts, just with more sophisticated frameworks (Django, Ruby on Rails, ASP.NET) and security features. But the core principle remains: **web pages are just SQL queries in disguise**.

15.2.8 CRUD using Natural Language

We've now seen three ways to interact with our database:

1. **SQL** - Direct queries (precise, but requires SQL knowledge)
2. **Web forms** - HTML interfaces (user-friendly, but requires building and maintaining web pages)
3. **Natural language** - The next frontier

15.2.8.1 The Natural Language Abstraction

Large Language Models (LLMs) have introduced a new level of abstraction: **asking questions in plain English and getting answers from your database**. Instead of writing SQL or clicking through forms, you can simply ask:

> "What was the average energy consumption in winter months when occupants were comfortable?"

The LLM translates this to SQL, executes it, and returns results in natural language. This pattern is emerging rapidly across database tools and could fundamentally change how non-technical users interact with data.

15.2.8.2 How It Works

The general pattern:

1. **User asks a question** in natural language
2. **LLM analyzes the question** and the database schema
3. **LLM generates SQL** that answers the question
4. **System executes SQL** on the database
5. **LLM formats results** in human-readable form

For example:

```
User: "Which room has the most comfort complaints?"

LLM generates:
SELECT r.room_name, COUNT(*) as complaints
FROM ComfortRating cr
JOIN Room r ON cr.room_id = r.room_id
WHERE cr.pmv_rating < -0.5 OR cr.pmv_rating > 0.5
GROUP BY r.room_name
```

```
ORDER BY complaints DESC
LIMIT 1

System returns: "Living Room with 253 complaints"
```

15.2.8.3 Benefits

1. Democratizes Data Access

- Non-technical users can query databases without learning SQL
- Reduces dependency on data analysts for simple questions
- Faster iteration on exploratory data analysis

2. Flexibility

- No need to pre-build every possible query as a web page
- Users can ask questions you never anticipated
- Natural follow-up questions work seamlessly

3. Contextual Understanding

- LLMs can (sometimes?) interpret ambiguous queries using schema context
- Can suggest clarifying questions when queries are unclear
- Can explain results in domain-specific terms

15.2.8.4 Pitfalls and Limitations

1. Accuracy Concerns

- LLMs can generate incorrect SQL, especially for complex queries
- Subtle logical errors may go unnoticed by non-technical users
- Always verify critical results against known benchmarks

2. Security Risks

- Potential for SQL injection if not properly sandboxed
- Users might inadvertently access sensitive data
- Need strict permission controls and query validation

3. Black Box Problem

- Users may not understand how answers were derived
- Difficult to debug when results seem wrong
- Loss of SQL literacy over time as users rely on natural language

4. Performance Issues

- LLMs may generate inefficient queries
- No guarantee of query optimization
- Repeated similar questions don't benefit from caching (unless explicitly implemented)

5. Schema Dependency

- Requires clear table and column names for good results
- Ambiguous schema designs confuse the LLM
- Works best with well-documented, normalized databases

15.2.8.5 A Practical Example: sqlite-utils-ask

One implementation of this pattern is sqlite-utils-ask by Simon Willison. This tool uses OpenAI's GPT models to answer questions about SQLite databases in natural language.

Installation:

```
# Install sqlite-utils with the ask plugin
pip install sqlite-utils
sqlite-utils install sqlite-utils-ask

# Set your OpenAI API key
export OPENAI_API_KEY="your-api-key-here"
```

Note: You'll need to have your own API key for this to work.

Basic Usage:

Ask questions about our thermal comfort database:

```
# Ask a question
sqlite-utils ask comfort-energy.db "What was the average PMV rating in the Living Room?"
```

The tool will:

1. Examine the database schema
2. Generate appropriate SQL
3. Execute the query
4. Return results in natural language

Example Questions to Try:

```
# Simple aggregation
sqlite-utils ask comfort-energy.db "How many comfort ratings were recorded?"

# Temporal analysis
sqlite-utils ask comfort-energy.db "What was the total energy consumption in January 2024?"

# Join query
sqlite-utils ask comfort-energy.db "Which occupant reported feeling uncomfortable most often

# Complex analytical query
sqlite-utils ask comfort-energy.db "Compare average power consumption between summer and win
```

Seeing the Generated SQL:

Add the `--sql` flag to see what SQL was generated:

```
sqlite-utils ask comfort-energy.db "Which room is most comfortable?" --sql
```

This is valuable for:

- Learning SQL from natural language examples
- Verifying the query logic
- Understanding how the LLM interpreted your question

15.2.8.6 When to Use Each Interface

Interface	Best For	Avoid When
SQL	Precise queries, performance-critical operations, production systems	Non-technical users, exploratory analysis
Web Forms	Predefined workflows, data entry, public-facing applications	Ad-hoc analysis, rapidly changing requirements
Natural Language	Exploratory analysis, one-off questions, democratizing data access	Mission-critical queries, high-security contexts, complex transactions

15.2.8.7 The Future

Natural language database interfaces are still evolving, but the trajectory is clear:

- **Hybrid approaches** combining natural language with traditional SQL for complex queries
- **Query validation** where LLMs explain their SQL before execution
- **Multi-turn conversations** that refine queries through dialogue
- **Automatic visualization** of results based on query intent

As these tools mature, the gap between "thinking of a question" and "getting an answer" will shrink dramatically. However, understanding the underlying SQL and database design remains crucial—both for building robust systems and for validating results when they matter.

15.2.8.8 Key Takeaway

Each abstraction layer trades **power and precision** for **accessibility and ease of use**:

```
SQL → Web Forms → Natural Language
↑                                 ↑
Precise, fast, experts only      Accessible, flexible, everyone
```

The best data systems often support multiple interfaces, letting users choose the right tool for their task and skill level.