

# Research Computing

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
<b>2</b>	<b>Information Systems</b>	<b>3</b>
<b>3</b>	<b>Systems Analysis</b>	<b>14</b>
<b>4</b>	<b>Instrument Interfaces</b>	<b>24</b>
<b>5</b>	<b>Resource Management</b>	<b>29</b>
<b>6</b>	<b>Data Files</b>	<b>33</b>
<b>7</b>	<b>Databases</b>	<b>50</b>

# Chapter 1

## Introduction

This resource contains teaching materials for an overview course in **research computing**. This course offers a broad overview of general computing concepts designed to support research activities in a university setting.

### 1.1 Audience

The target audience is primarily college students, particularly graduate students, who conduct academic or scientific research. The information will also be useful for working professionals.

### 1.2 Course Goals

The main goal of this course is to help students improve their technical readiness for engaging in the increasingly data-centric work that they face in their degree programs, in their related research, and in their future professional and research practice. Students will become acquainted with key concepts in research computing and data management, including overviews of systems analysis techniques, data security and integrity, and database tools, among others.

### 1.3 Learning Objectives

At the end of this course students should be able to:

- Analyze requirements for management of data in different situations and projects.
- Choose appropriate technical tools and techniques to support that data management.
- Identify hazards and pitfalls in data-related projects.
- Identify factors that affect performance in the collection and preparation of data for analysis.
- Describe the core technologies most frequently employed in large-scale research data management.

### 1.4 File Contents

The course materials primarily consist of the presentation slides in Markdown format and the transcripts which go along with those slides. The transcripts are posted as wiki pages in AsciiDoc format and are also offered as PDF and EPUB eBooks. These materials may be found online at: <https://github.com/brianhigh/research-computing>.

---

## 1.5 The Research Computing Team

Thanks to the following people who have contributed to this resource (in no particular order):

*Brian High, Jim Hogan, John Yocum, Elliot Norwood, and Lianne Sheppard*

## 1.6 Copyright, License and Disclaimer

Copyright © The [Research Computing Team](#). This information is provided for educational purposes only. See [LICENSE](#) for more information. [Creative Commons Attribution 4.0 International Public License](#).

---

## Chapter 2

# Information Systems

Your research computing experience will involve using *and* developing information systems. We will take a quick look at the various components, types, and development models of these systems.

### 2.1 Information System Components

The primary components of an information system<sup>1</sup> are hardware, software, data, and *people* — the most important component of all! Why? Because systems are designed and built *by* people *for* people. If people don't use them, or they do not serve the people's needs, then they are worthless! Today we will take a closer look at how information systems are designed to help us.



Figure 2.1: Exploded view of a personal computer - Image: Gustavb, CC BY-SA 3.0 Unported

---

<sup>1</sup> Information system - Components, [Wikipedia](#), CC BY-SA 3.0

---

## 2.2 Hardware

The physical machinery of a computer system is called its **hardware**. Of course, this means the computer itself, its chassis and the parts inside it, including its core **integrated circuit** known as the **central processing unit (CPU)**, as well as its memory, called "RAM" or Random Access Memory, and any internal storage devices like hard disk drives (HDD) and solid state devices (SSD).

Accessories or [peripherals](<http://en.wikipedia.org/wiki/Peripheral>) are the devices you plug into the computer, mostly for input and output.

**Networking equipment** includes all of the devices that allow your computer to communicate with other systems. Examples are the network cables and the boxes they connect to, such as routers, switches, hubs, wireless access points, and modems.

## 2.3 Software

Software is the name for the instructions we give to computing devices to tell them what to do. Software is "soft" because the instructions are not physical entities like hardware devices. The instructions may be stored on physical media like a hard disk or USB thumbdrive, just as a cooking recipe may be written on a piece of paper or printed in a book. However, the recipe itself is just a *conceptual model* of how to perform a task. Likewise, a software program is essentially just a list of instructions (or a *logical model* that issues instructions) for the execution of a set of desired computing operations.

### 2.3.1 Application Software

As you use a computer, the **software** instructions that are executed on your behalf by the CPU, such as **programs** and **apps**, are called **application software**. Applications are the programs that serve a specific purpose for a computer **user** or are to be used for completing certain tasks, such as exploring the Internet, editing a text document, or working with data.

### 2.3.2 System Software

#### 2.3.2.1 The Operating System

Applications run within a *overall* software environment called the **operating system (OS)**.

Notable examples are the familiar **Microsoft Windows**, **OS X**, **iOS**, **Android** and **Linux** operating systems.

#### 2.3.2.2 Kernel, Drivers, and Firmware

An operating system also has a **kernel**, which is the central software program that manages the **data** exchange between the CPU and the other components within a computer. The kernel communicates with those components using **device drivers**, which are small programs that provide a software **interface** to the hardware. Devices that contain integrated circuits of their own may store software in **firmware** that allows updates through a procedure called **flashing**. The computing system will also contain **utility software** such as configuration and management tools, plus shared **software libraries** used by both applications *and* system software.

---



needs *optimally*. But our needs vary, and so we need various types of systems.



Figure 2.3: Pair Programming - Image: Ted & Ian, CC BY 2.0 Generic

---

## 2.6 Information System Types

We may use several types of information systems each day. Let's take a quick look at a few of the most common types.<sup>2</sup>

Most of us are very familiar *search information systems* like web **search engines**, such as **Google Search**, but many sites use domain-specific search engines like **PubMed**.

**Spatial information systems** in the form of **Geographic information system (GIS)** have become increasingly important in recent years. **ArcGIS** has dominated this field, with the free and open **QGIS** gaining in popularity.

**Global information systems (GLIS)** are those either developed or used in a global context. Public health examples include global health databases such as the **UNHCR Statistics & Operational Data Portals** and the WHO's **Global Health Observatory (GHO)**.

**Enterprise systems** are comprehensive organization-wide applications used for **Enterprise Resource Planning (ERP)**.

**Expert systems** support such specialty domains as diagnosis, forecasting, and delivery scheduling. They use artificial intelligence to apply knowledge and reasoning in order to solve complex problems.<sup>3</sup>

**Office automation systems** refer to systems which support the everyday business operations of an organization. **Business Process Automation (BPA)** uses these systems to improve efficiency by streamlining routine activities.

---

<sup>2</sup> *Information system - Types of information system*, **Wikipedia**, **CC BY-SA 3.0**

<sup>3</sup> For more information about expert systems for public health, see "Decision Support and Expert Systems in Public Health" by William A. Yasnoff and Perry L. Miller, *Public Health Informatics and Information Systems*, 2nd Edition, pp. 449-467 (Springer, 2014). The chapter uses *IMM/Serve* as an example. See also: *IMM/Serve: a rule-based program for childhood immunization*. P. L. Miller, S. J. Frawley, F. G. Sayward, W. A. Yasnoff, L. Duncan, D. W. Fleming Proc AMIA Annu Fall Symp. 1996 : 184-188. <http://www.ncbi.nlm.nih.gov/pmc/articles/PMC2233221/>.

---



*Personal* information systems help people manage their individual communications, **calendaring**, note-taking, diet, and fitness.

## 2.7 Software Development Process

Developers undertake the **software development process** using several different approaches. Let's take a look at a few of the most popular models.

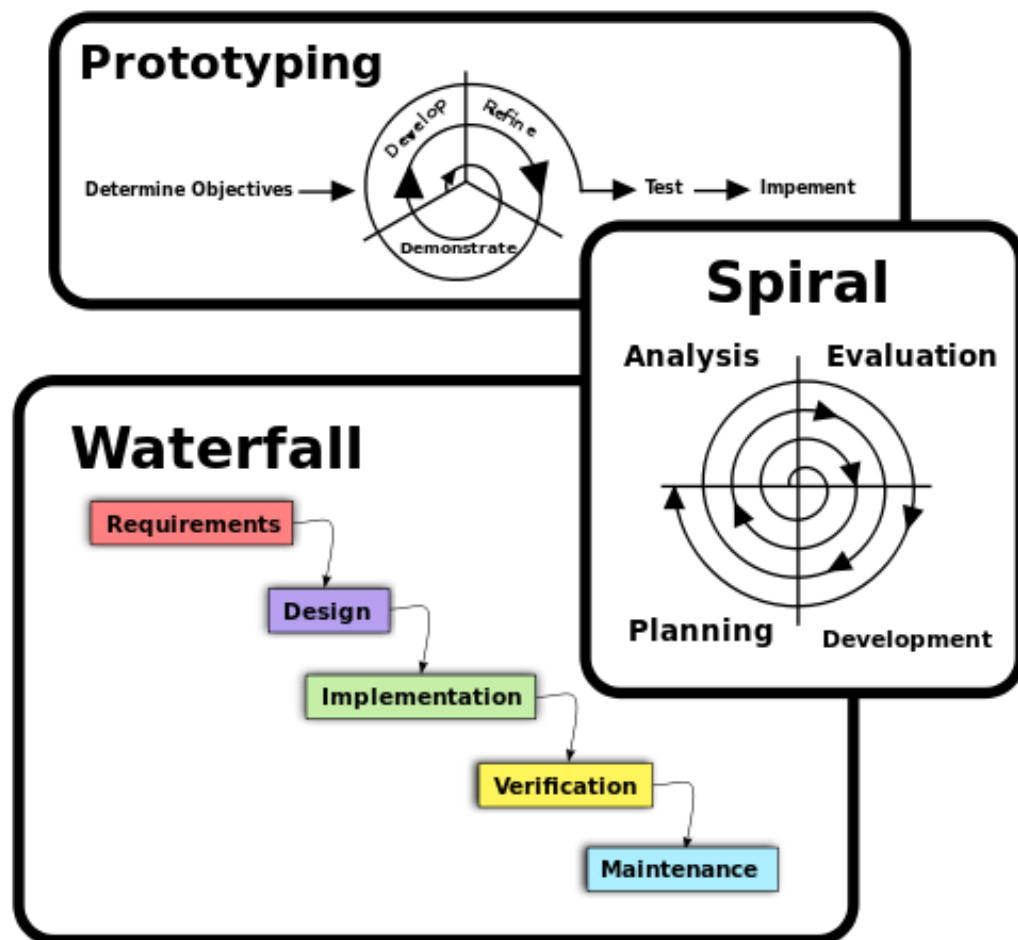


Figure 2.4: Three software development patterns mashed together - Image: Beao, Paul Smith, Public Domain

Here is a short list of common development models.<sup>4</sup> We have provided links from each of these to relevant Wikipedia pages. You are encouraged to read more about them. We'll just go through the list quickly to give you a rough idea of the differences between them.

The **Systems development life cycle (SDLC)** is the classic model. It involves lots of up-front planning and is risk averse.

<sup>4</sup> *Software development process*, [Wikipedia](#), CC BY-SA 3.0

**Waterfall development** is another and "old school" favorite, It's like the SDLC but does not offer any sort of feedback loop.

**Prototyping** is a useful technique for many models. Good when a small-scale experiment can prove an idea without risking heavy investment.

**Iterative and incremental development** might evoke the image of "baby steps" or the notion of "try, try, again". There is a central loop, between initial planning and final deployment, which repeats as needed. Like prototyping, it is a technique which can be used in other models.

Likewise, **Spiral development** is meant to address evolving requirements through cycles of repeated analysis and design, getting closer and closer to the desired product. The idea is that the *entire process* is repeated over and over until you are finally satisfied.

**Rapid application development (RAD)** focuses on development more than up-front planning.

**Agile development** is a more evolved form of RAD, with more of a focus on user engagement, and gaining wide popularity.

**Code and fix** sounds like what it is — *cowboy coding* — what most lone programmers do, and what might seem most familiar to you as a scientific researcher. This can be quick for easy projects, but can be very inefficient and expensive for larger projects, due to insufficient planning.

They are all useful methods, though, some more generally than others. The approach you take should depend upon your situation.

We'll look more closely at three of these right now.

### 2.7.1 Systems Development Life Cycle

Since information systems are so complex, it is very helpful to follow a standard development model to make sure you take care of all of the little details without missing any.



Figure 2.5: SDLC - Image: Dzonatas, CC BY-SA 3.0

For years, the standard development model was known as the SDLC, or Systems Development Life Cycle.<sup>5</sup> It works well for large, complex, expensive projects, but can be scaled down as needed. Many of its phases are used in the other models as well. Let's take a quick look at them.

#### Systems development life cycle (SDLC) phases:

- **Planning (feasibility study)**
  - There is a focus on careful *planning* before any design or coding takes place. The feasibility study explores your options and gaining approval from stakeholders.
- **Analysis**
  - *Analysis* includes a detailed study of the current system and clearly identifying requirements before designing a new system.
- **Design**
  - Once you have thoroughly defined the requirements, you can begin to model the new system.
- **Implementation**
  - *Implementation* is where the hardware assembly, software coding, testing, and deployment takes place.
- **Maintenance**
  - *Maintenance* may sound boring, but it is essential to ensure that the project is an overall success.

The main idea is that systems development is a cycle—a continual process. You need to allow for maintenance, updates, and new features. The use and upkeep of the system provides feedback which goes into planning the next version.

We will spend more time on the SDLC and its early phases in a separate module.

### 2.7.2 Waterfall Model

A related model is the **Waterfall model**.<sup>6</sup> It has basically same same steps as the SDLC, but visualizes them as cascading stair-steps instead of a circle.

---

<sup>5</sup> *Systems development life cycle*, Wikipedia, CC BY-SA 3.0

<sup>6</sup> *Waterfall model*, Wikipedia, CC BY-SA 3.0



Figure 2.6: Waterfall model - Image: Peter Kemp / Paul Smith, CC BY 3.0

It's basically similar to the SDLC, but without the feedback loop. There are cascading stair-steps, where one phase leads to another and the output of one phase becomes the input of another. It came from manufacturing where after-the-fact changes are expensive or impossible.

### 2.7.3 Agile Model

The **Agile model** is a newer, but very popular, especially among smaller teams within budding organizations. Hallmarks of this model include methods such as pair programming, test-driven development, and frequent product releases.<sup>7</sup>

---

<sup>7</sup> *Agile software development*, [Wikipedia](#), CC BY-SA 3.0



Figure 2.7: Agile Software Development methodology - Image: VersionOne, Inc., CC BY-SA 3.0

Smaller teams that can meet regularly, ideally face-to-face. Working in pairs, with one person coding and other helping "over the shoulder". After you identify use cases, then you write tests and then build the system to pass the tests. By developing an automated test and build system, releases can be pushed out quickly and more often.

## 2.7.4 Transparency

Information systems vary in the **openness** of their implementations, in terms of both **interoperability** standards and specific design details.<sup>8</sup>

<sup>8</sup> *Openness*, Wikipedia, CC BY-SA 3.0

You can have *open* systems (and **standards**, **source**), where the technical specifications are publicly available.<sup>9</sup> Different organizations may implement them in their own way, yet still maintain **interoperability** with other implementations.

Or systems may be *closed*, or **proprietary**, where an organization keeps the details to itself, making it more difficult for competitors to inter-operate. While this may provide a competitive advantage for the producer it contributes to what is called **vendor lock-in**, where a consumer becomes dependent on the vendor, unable to switch to another due to the high costs and disruption.

These interoperability aspects will include **file formats**, **communications protocols**, **security** and **encryption**.

All of those are important when you are collaborating, sharing data and files with others, who might be using different platforms.

By using transparent systems, you not only increase your ease of communication and collaboration, you also contribute to openness in a broader, social context.

Information **transparency** supports **openness** in:

- **Government**
- **Research**
- **Education**
- **Courseware**
- **Content**
- **Culture**

We have provided links to several popular movements which are working to increase openness and transparency in various aspects of society. You are encouraged to spend some time learning about these trends.

So, if you want the benefits of openness in your work and more freedom to make changes, consider building your information infrastructure with open technologies.

### 2.7.5 Transparency Example: This Course

As an example, we have assembled a transparent information system to create and support this course.

We have developed the course transparently, using an open content review process where students, staff and faculty look at the materials and evaluate them to determine whether or not they best meet the course goals.

We have an open content license, the Creative Commons Attribution Share-Alike **CC BY-SA 4.0 International** license.

We have open development where our source is freely and publicly available on **GitHub**).

We are using open file formats (**Markdown**, **HTML**, **CSS**, **PNG**, **AsciiDoc**, **PDF**), open source tools tools (**RStudio**, **Git**, **Redmine**, **Canvas**, **Linux**, **Bash**) and open communications protocol standards (**HTTP/HTTPS**).

As you take part in this course, and provide feedback which will go toward improving it, we thank *you* for contributing!

---

<sup>9</sup> Software standard, **Wikipedia**, **CC BY-SA 3.0**

## 2.8 Wrap-Up

We hope that this brief overview of Information Systems has given you a more clear picture of the what they are and how they are built.

For more information, please read the related sections in the [Computing Basics Wiki](#), particularly, the pages on [hardware](#) and [software](#).

In the next module, we will take a closer look at [requirements gathering](#) and [systems analysis](#), two of the most important topics of this course.

---

## Chapter 3

# Systems Analysis

Investing time and money into a computer or information system without a clear course of action can be expensive and wasteful. By taking some time to fully consider the issue at hand and pursue a disciplined approach to finding a practical solution, you greatly increase your odds of success and decrease costs. We call this process *Systems Analysis*.

### 3.1 Systems Development Life Cycle

Systems analysis is an important part of an overall approach to *systems development*.

The life of an information system follows a cycle. The classic development model is called the Systems Development Life Cycle, or SDLC.<sup>1</sup>



Figure 3.1: SDLC - Image: Dzonatas, CC BY-SA 3.0

---

<sup>1</sup> *Systems development life cycle - Phases*, [Wikipedia](#), CC BY-SA 3.0

---



### 3.1.1 Planning Phase

The **Planning** phase defines the primary issue (*problem* or *goal*) and performs a **feasibility study**. Here, you clarify the project scope, compare your best options, and come up with a plan.

### 3.1.2 Analysis Phase

The **Analysis** phase focusses on the issue, defined previously, and studies its role in the current (or proposed) system. The system is explored, piece by piece, in light of the project goals, to determine system requirements.

### 3.1.3 Design Phase

In the **Design** phase, a detailed model of the proposed system is created. Various components or modules address each of the requirements identified earlier.

### 3.1.4 Implementation Phase

During the **Implementation** phase, a working system is built from the design and put into use.

### 3.1.5 Maintenance Phase

The **Maintenance** phase includes ongoing updates and evaluation. As changes are needed, the cycle repeats with more planning, analysis, and so on. We will take a closer look at the **systems analysis** phase next.

## 3.2 What will you need?

Essentially, in **Systems analysis** we answer the question, "*What will you need* to reach your goal?" In other words, "**What are your requirements?**"

A **primary goal** of this course is to help you develop your skills in **requirements analysis**.

**Systems analysis** helps you **clarify your project needs** and **plan ahead** in order to *obtain and allocate* **critical resources**.

The main idea here is ...

If you don't *know* what you *need*, how can you *ask* for it?

## 3.3 Systems Analysis

After completing an initial **feasibility study** to "determine if creating a new or improved system is a viable solution", proposing the project, and gaining approval from **stakeholders**, you may then conduct a **systems analysis**.

**Systems analysis** will involve "**breaking down** the system in different pieces to analyze the situation, **analyzing project goals**, breaking down what needs to be created and attempting to **engage users** so that **definite requirements** can be defined."<sup>2</sup>

---

<sup>2</sup> *Systems development life cycle - System investigation*, Wikipedia, CC BY-SA 3.0

### 3.4 Systems Analysis Definition

So, what, exactly, *is* "Systems Analysis"?

- "A *system* is a set of interacting or interdependent components"<sup>3</sup>
- *Analysis* means "to take apart"<sup>4</sup>

Putting these two ideas together, we have:

**Systems analysis** is a problem solving technique that decomposes a system into its component pieces for the purpose of the studying how well those component parts work and interact to accomplish their purpose.<sup>5</sup>

— Lonnie D. Bentley *Systems Analysis and Design for the Global Enterprise*

We perform this analysis by working through a series of **five phases**.

### 3.5 Systems Analysis Phases

Systems Analysis<sup>6</sup> has its own series of phases. We will look at each of these one by one.

---

<sup>3</sup> *System*, [Wikipedia](#), CC BY-SA 3.0

<sup>4</sup> *Systems analysis*, [Wikipedia](#), CC BY-SA 3.0

<sup>5</sup> *Systems Analysis and Design for the Global Enterprise 7th ed.*, by Lonnie D. Bentley, as quoted by [Wikipedia](#)

<sup>6</sup> *Systems analysis - Information technology*, [Wikipedia](#), CC BY-SA 3.0

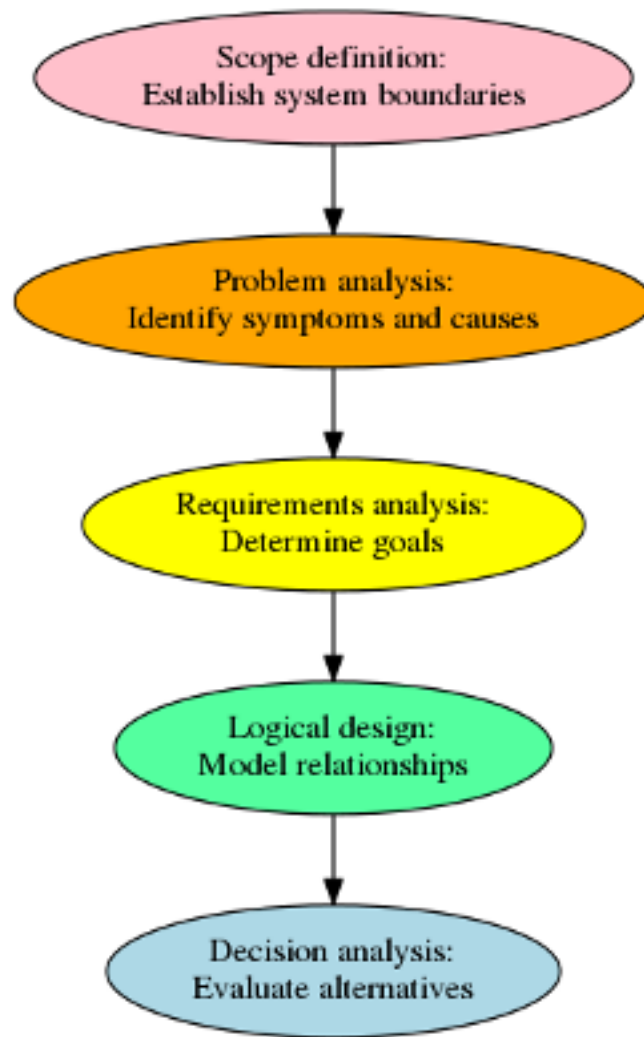


Figure 3.2: Systems Analysis Phases - Image: Brian High, CC-BY 4.0 International

During **Scope definition**, we establish our system boundaries. In our **Problem analysis** phase, we identify the symptoms and causes. Our **Requirements analysis** determines our system goals. The **Logical design** phase models relationships within the system. And our **Decision analysis** evaluates the various alternatives.

### 3.6 Scope Definition

So, do we mean by *scope*?<sup>7</sup>

**Scope** involves *getting information* required to start a project, and the *features* the product would need to have in order to meet its *stakeholders requirements*.

— Wikipedia *Scope (project management)*

---

<sup>7</sup> *Scope (project management)*, Wikipedia, CC BY-SA 3.0

Put another way, *project* scope defines the *work to be done* whereas *product* scope is concerned with the *desired features and functions*.

By being careful to define scope *early on*, we can be more watchful for *scope creep*.<sup>8</sup>

**Scope creep** is [...] the *incremental expansion* of the scope of a project [...], while nevertheless *failing to adjust* the schedule and budget.

— Wikipedia *Scope (project\_management)*

### 3.7 Problem Analysis

Problem analysis is critical. When we say, *problem*, we can also think *goal*, *research question*, or *issue*. If you don't get this right, you can waste a lot of time and money solving the *wrong problem* or address a *non-issue*.

We can summarize the key points of problem analysis<sup>9</sup> as:

- Define and clarify the problem (or issue)
  - *What exactly are we trying to solve?*
- Determine the problem's importance
  - *How much does it matter?*
- Assess the feasibility of solving the problem
  - *Do we have the resources we need?*
- Consider any negative impacts (unintended consequences)
  - *What could go wrong?*
- Prioritize problems to solve (bottlenecks? low-hanging fruit?)
  - *What are the most critical issues?*
- Answer: *what, why, who, when, where, and how much?*
  - *Drill down into the problem with all kinds of questions to expose dependencies.*
- Find **causes** (especially **root cause**) and **symptoms** (effects)
  - *What is really going on here?*

---

<sup>8</sup> *Scope (project management)*, Wikipedia, CC BY-SA 3.0

<sup>9</sup> Jenette Nagy, *Defining and Analyzing the Problem*, Analyzing Community Problems and Designing and Adapting Community Interventions, Kansas University, CC BY-NC-SA 3.0 US

### 3.8 Root Cause Analysis

There are several methods of root cause analysis, but one simple one is "Ask Why Five Times" ...



Figure 3.3: Root Cause Analysis Tree Diagram - Image: KellyLawless, CC BY-SA 3.0 Unported

... where you keep asking "Why? But, why? But, Why?" over and over again until there is a clear root cause that you can actually do something about.

A real example is mentioned in [Ask Why 5 Times, Business Analysis Guidebook/Root Cause Analysis \(wiki-books.org\)](#), where the US National Park Service was able to slow the rate of deterioration of the Jefferson memorial simply by changing the lighting schedule.<sup>10</sup>

This method may also be used in [Requirements Analysis](#).

### 3.9 Requirements Analysis

- [Elicit, Analyze, and Record \(EAR\)](#):

---

<sup>10</sup> Ask Why 5 Times, Business Analysis Guidebook/Root Cause Analysis, [Wikibooks](#)

- System and project **requirements**

EAR Analysis is all about *listening*.

Gather requirements by interviewing stakeholders. Observe how people currently do their work or use the system. Make sure the requirements are detailed, clear, unambiguous, and comprehensive. Document the requirements as a list, in diagrams, or narratives.<sup>11</sup>

As an example, in Spring 2014, we held a meeting with our graduate students and asked them about what sort of computing needs they had. We had a discussion, *listened* to what they had to say, summarized the main concerns, and documented them.

- Further elucidate **Measurable goals**

By continuing to ask questions like "Why? ... Why? ... Why? ...", get more specific until the goals become quantifiable. Measured goals are easier to meet since you can measure your progress in achieving them.

Mission objectives determine the goals. Compare the stated goals against mission objectives to produce a small set of critical, measured goals.

- Output: **Requirements specification**

This document will be used in the *logical design* phase to model the relationships within the system. So, it should be clear and thorough. The more specific and unambiguous this specification is, the easier it will be to design the system to meet the requirements.

One way to be more clear is to organize the requirements by type.

### 3.9.1 Requirements Categories

You may categorize requirements according to several important types:

**Types of requirements:**

Operational requirements are the **utility, effectiveness, and deployment** needs, with respect to practical use of the system within the overall operation. These are the **customer Requirements**.

**Functional** requirements are specific things the system must **do**.

**Non-functional** requirements are specific things the system must **provide**.

**Architectural** requirements define how the system must be **structured**, in other words, how the components must **interrelate**.

Behavioral requirements describe how **users** and other systems will **interact** with the system and how the **system** will respond.

Performance requirements define **how well** the systems needs to do things, **measured** in terms of quantity, quality, coverage, timeliness or readiness.

While there are other ways to group requirements, these are the most significant categories.

---

<sup>11</sup> *Requirements analysis*, [Wikipedia](#), CC BY-SA 3.0

### 3.10 Requirements Modeling: Example

Another way to make requirements clear is through *requirements modeling*. Here is an example of modeling behavioral requirements with a **Use Case Diagram**.

#### Survey Data System

The behavioral requirements are stated in **role goal** format.

- **Researcher** *uploads survey*.
- **Subject** *takes survey*.
- **Subject** *uploads results*.
- **Researcher** *downloads results*.

Gramatically speaking, the role is the *subject* and the goal is stated as a *verb* and its *object*. Each of these requirements forms the basis of a *use case*.

#### 3.10.1 Use Case Diagram

The **Use Case Diagram** is a standard means of showing these requirements pictorially.

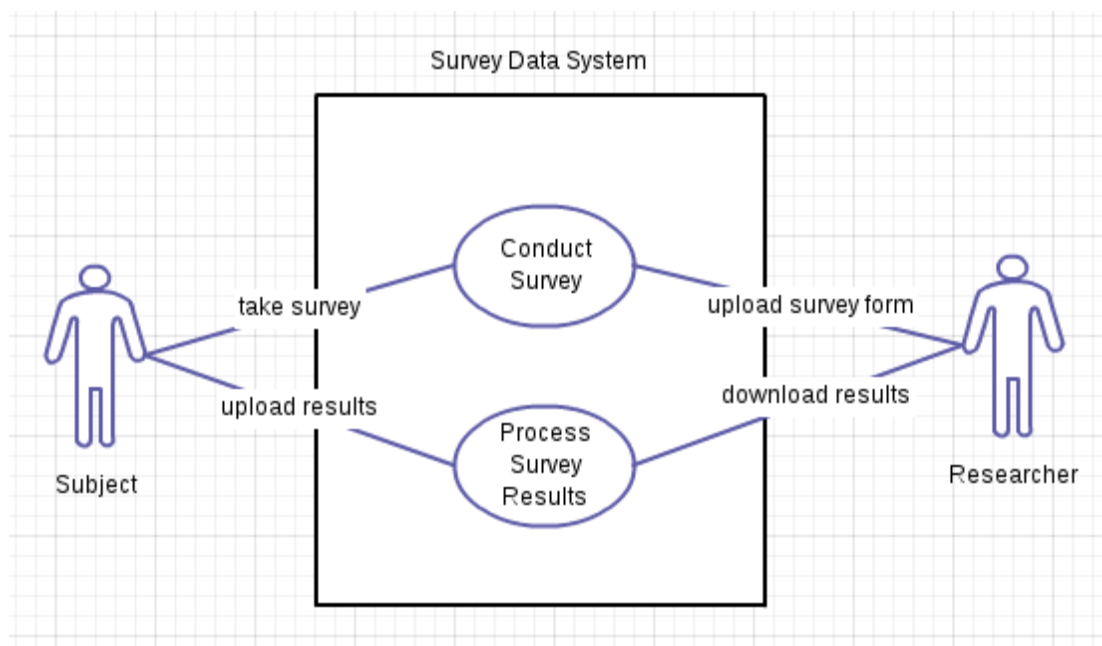


Figure 3.4: Research Survey Data System - Image: Brian High, CC0 1.0

Here, human *actors* are depicted as stick figures. These are the **roles**. The *goal* is the line connecting the role to a system activity or function, shown as an oval. In this example, the *system* is drawn as a box with the actors outside the box. The reason is that the system's *product scope* was defined to be the electronic data system. The human actors interact with that system. The information flowing to and from an actor, as well as data flows within a system, will be modeled in the *logical design*.

### 3.11 Logical Design

Once system behaviours have been modeled from the perspective of user interaction, we can begin to model the internal workings of the system with the *logical design*.<sup>12</sup> The *logical* part means that this is an *abstract representation* of the system. Therefore, the system is modeled in terms of abstractions such as *data flows*, *entities*, and *relationships*. We model how the system will satisfy function requirements such as *inputs and outputs*. To make the abstract more concrete, we make use of *Graphical modeling* techniques to produce graphics such as the **Data Flow Diagram (DFD)** and **Entity Relationship Diagram (ERD)**.

#### 3.11.1 Example Entity Relationship Diagram (ERD)

For example, let's consider a systems which records the playlists of musical performances at a local pub. You will want to store which artist performs which song. The relationship between an artist and a song can be shown in a simple **Entity Relationship Diagram**. We start with the behavioral *use case* written as **Artist performs Song**. Then, using a standard set of symbols, we can produce this diagram.

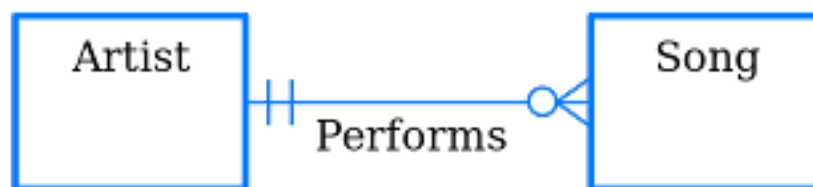


Figure 3.5: ERD artist performs song - Image: Bignose, Public Domain

The boxes represent *entities* and the line connecting them is the relationship. Here, the symbols indicate that there is a one-to-many relationship. This diagram is saying that exactly one artist performs one or more songs, (or doesn't perform any).

Since this "system" describes only solo performers, you would want to model it differently to include groups of artists that perform more than one song. You would want a many-to-many relationship as well as other entities to represent the musical groups. Since there will be many performances, you will also want an entity to represent the performances.

The value of this type of diagram is that complex relationships can be mapped to a great level of detail. In fact, they can be used to automatically generate database schemas.

### 3.12 Logical Design: Diagrams

Several examples of these diagrams can be found in the **Systems Analysis and Design** tutorial.

This **tutorial**<sup>13</sup> (**PDF**, **HTML**, **MP4**) provides several examples from a fictitious public health research study.

You will find several other real-world examples from actual public health research projects in that course repository.

<sup>12</sup> *Systems design - Logical design*, Wikipedia, CC BY-SA 3.0

<sup>13</sup> See also: **Data Management**, UW Canvas.



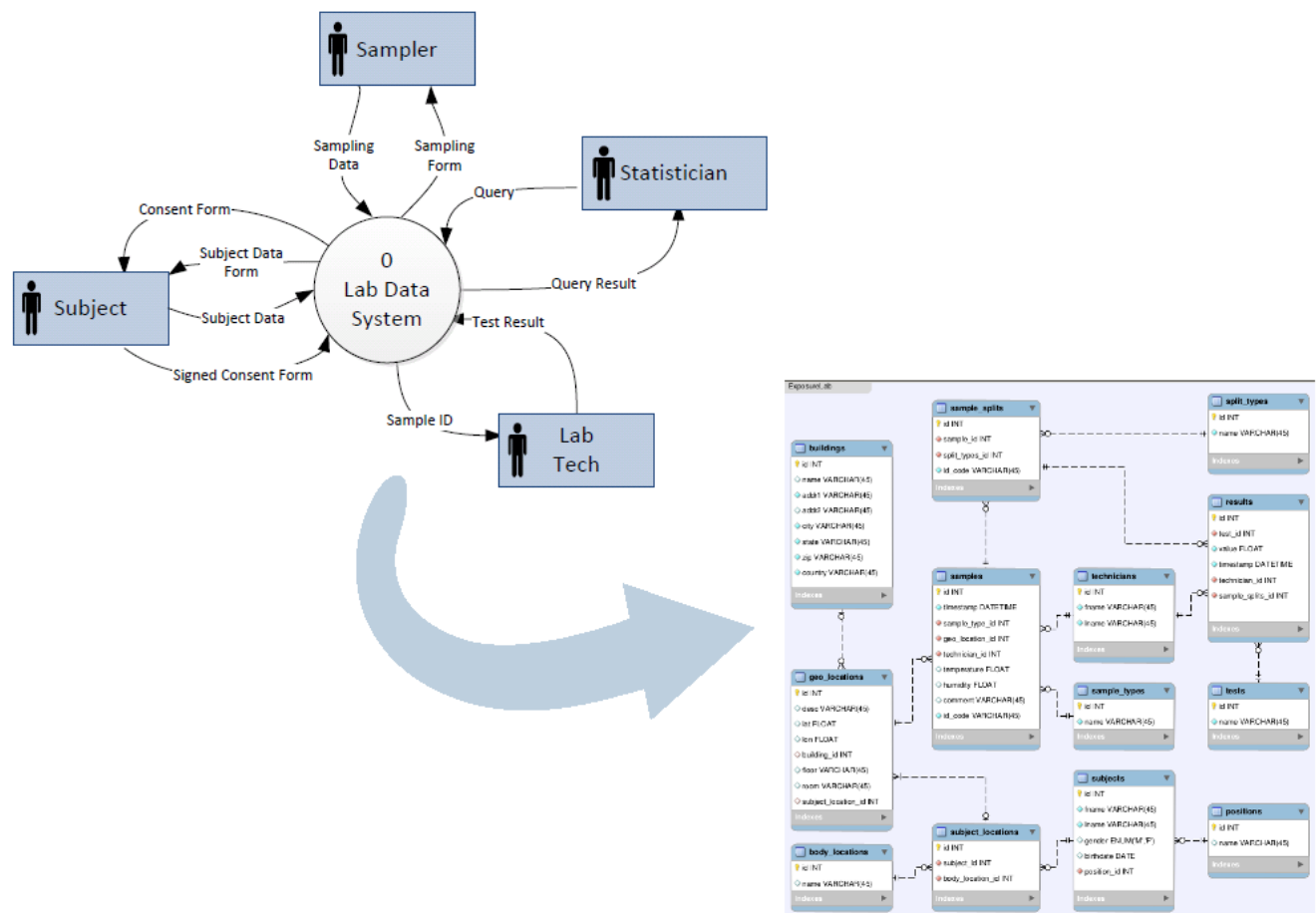


Figure 3.6: DFD and ERD - Image: Brian High, CC0 1.0

### 3.13 Decision Analysis

These models and diagrams will help you make a very important decision. Before you go further in the system design process, you need to decide whether to buy a system or build one. Or you may consider building a system of components, some of which you might buy and others might be custom built. You will want to come up with a few of the best alternatives and present them to your stakeholders. In your case that might be your lab manager, principal investigator, or funding agency.

Your presentation will also include a **Decision analysis** to weigh the pros and cons of the various options against the requirements in order to help the stakeholders with their decision. You should conclude your analysis with a recommendation of your top choice and explain why this choice is the most compelling. Then you will want to get a decision, and the approval to continue, before you invest any more time on further analysis.<sup>14</sup>

The system development life cycle (SDLC) continues on to other phases, which we do not have time to cover here. However, we hope that this glimpse at the systems analysis phase has demonstrated the value of this approach in gathering and clarifying requirements that can be used to design and build an information system.

<sup>14</sup> *Decision analysis*, Wikipedia, CC BY-SA 3.0

## Chapter 4

# Instrument Interfaces

Data is waiting to become information. Through instrument interfaces, and data acquisition software you can transform raw data into useful information.

### 4.1 Input/Output (I/O)

Computers and devices are equipped with a variety of inputs, and outputs. Inputs and outputs have both physical and virtual components. On the physical side, is the various ports, connectors, and cables utilized. On the virtual side, are the various protocols that operate over the physical interface. That said, physical interfaces aren't limited to cabled connections. Wireless both radio frequency and optical are growing in popularity. Regardless of the physical medium most are digital, however some, such as audio, are analog.

### 4.2 Peripherals

Peripherals are not limited to monitors and keyboards. They also include sensors, and complex instruments. Each of these devices connects via some interface which could be wired or wireless. Be aware though, that a peripheral may be available in more than one interface type or may be equipped with multiple interface options. For example, monitors often equipped with both analog and digital inputs.

### 4.3 Instrument Interface Technologies

An input/output card expands a computer's I/O options. These can range from a basic USB card, to a GPIB controller. Something to keep in mind, laptop computers tend to be more limited in their I/O expansion options. If you need more than USB and Ethernet capability, a desktop computer is recommended.

---



Figure 4.1: Chassis Plans 8011 Digital I/O Card: Lippincott, CC BY 3.0

A bus technology allows multiple devices to share a common medium. With bus technologies, you can share a single port on your computer with multiple devices via a bus specific hub or in some cases via daisy chaining devices.

Hewlett-Packard Interface Bus evolved into the **General Purpose Interface Bus**. It's a standardized interface bus used by a wide array of scientific instruments. One nice feature of GPIB, is the ease of converting it into another interface technology such as Ethernet or USB.



Figure 4.2: IEEE-488 Connectors: 1-1111, Public Domain

**Ethernet** is the most common network interface technology. So common, that most computers built today include Ethernet on-board. Many modern instruments have native Ethernet support, enabling them to be accessed directly from a desk, instead of via a computer located next to the instrument.

**Serial** and **Parallel** are both "legacy" interfaces. Their popularity has been in a decline over the past decade, largely being replaced with USB and Ethernet. Due to their decline, they aren't typically seen on modern computers.

**USB** is a fairly universal bus technology. It's used to connect everything from keyboards to GPIB interface controllers. It's found on every computer made in the past 10 years, making it a safe choice when compatibility is needed.

## 4.4 Remote Data Acquisition

Sensors exist for most everything, from temperature and pressure to particulate detection.

Hackable devices have become popular in the past few years. They are designed to make development of custom devices and sensor packages more accessible to a wider user base. In the past, where a commercially built sensor package would be used, a hackable system may be a suitable replacement.

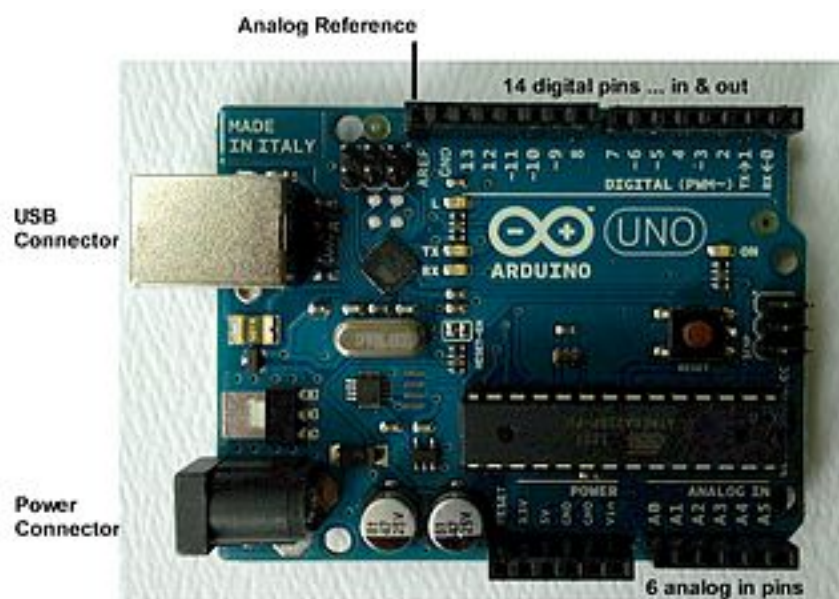


Figure 4.3: Arduino Uno: 1sfoerster, CC BY SA 3.0

GPS enables you to quickly and accurately determine location, altitude, and time. By integrating GPS into your field data acquisition, you may be able to partially automate location tracking of sensors and samples.

Mobile devices such as smartphones and tablets can be used for field data acquisition. They can be used to scan barcodes on sample containers and record the location (when equipped with GPS), or used to log notes during an interview.

## 4.5 Instrument Data Acquisition Software

Instruments require acquisition software. It translates the raw data from the instrument into a useful file format, or provides some additional processing capability. Keep in mind, some software and formats are limited to specific brands of instruments. So, be sure the software you wish to use will work with your equipment and other tools.

**LabVIEW** is a popular package due to its unique development language, called G. What sets G apart from others, is the graphical development model. Instead of writing pages of code, you graphically build your acquisition system by dragging and dropping objects on the screen.

**ChemStation** from Agilent is one of many chromatography packages. Like many instrument acquisition packages, it's a vendor specific software package limited to use with Agilent equipment.

Torrent Suite is a modern sequence analysis package. It's available with an open source license, making it free software. Open source alternatives exist for a variety of proprietary software.

## 4.6 Legacy Systems



Figure 4.4: Intertec Superbrain: Brighterorange, CC BY SA 3.0

---

Earlier we introduced the idea of "legacy interfaces" like serial and parallel, which are showing up on fewer and fewer computers these days. Which introduces a problem with both hardware and software used to interface with an instrument.

Let's take for example a mid 90s vintage HPLC, which is likely to use GPIB to talk with the computer. While the basic GPIB interface is supported on a modern system, the instrument may have other requirements. Such as requiring manufacturer specific software that depends on their own interface card. And, the interface card uses the ISA bus, a bus that hasn't been used in computers in almost 10 years.

---

So, why is this a problem? As the computer ages, it's reliability goes down. In addition, it's compatibility with peripherals and software becomes worse over time. Thus, while the instrument may still do it's job, you may have difficulty transferring data from the computer to other systems, printing, etc. These issues may first appear as minor annoyances, but can quickly evolve into a work stoppage.

You can help to mitigate this situation, by reviewing the requirements for the instrument. For instruments that have a very long life, it's best to avoid ones that depend on proprietary hardware interfaces. So, look for units that use USB or Ethernet. Review the software support policy, most manufacturers require you to purchase updates over time. But, make sure they have plans to support the instrument long term. In addition, you may want to see if any third-party software vendors support your instrument, as they have a greater financial interest in supporting older equipment.

## 4.7 Instrument Overload

Up until now, we've covered how instruments interface with a computer, software that interacts with the instrument, and the pitfalls of legacy systems. Now, we must turn our attention to the overall resource needs of instruments.

In a lab environment, it's typical to have 1, maybe 2 instruments connected to a single computer. However, in the interest of saving money, and space some attempt to connect far more than that to a single computer. If the instruments aren't being used simultaneously, you can probably get away with it. But, if you do wish to interact with many at once, you're likely to run into issues.

One group actually attempted to connect a couple dozen instruments in a mobile environment to a single laptop computer. From a pure software perspective, their tool of choice, LabVIEW, is designed to interact with multiple devices at once. But, as the number instruments goes up, the amount of CPU, Memory, and Disk activity also goes up. That said, their main issue wasn't with any of those resources, it was with interface technology. Each instrument connected via USB, which is a bus technology. And, like most bus technologies, the more devices on the bus, the slower it gets.

So, when developing a plan involving instruments that require a computer interface, you must consider more than just the physical connection, and the software support. You must also consider if the interface technology can really support that many active instruments at a time. In this example, the lab group ended up purchasing a second computer to divide up the work load.

## 4.8 Summary

To review. Peripherals are typically external input and output devices, such as keyboards, and monitors. There are a number of interface technologies on the market, the latest instruments can be purchased with Ethernet built-in, while most older devices will rely on GPIB. As for data acquisition, in the field you can use mobile devices like tablets, and GPS to aid your work. And, in the lab, use software like LabVIEW to process data from an instrument.

When purchasing lab instruments or working with older models keep a few things in mind. Older instruments may not work properly with modern computers. In some cases, the manufacturer may have a paid option to make it work, such as a software or interface upgrade. However, that option may be costly, so it's best to plan ahead for that scenario, rather than be stuck trying to keep an old computer alive.

Finally, make sure your instrument needs, don't exceed your computer and interface limitations. If you exceed those limits, you could face issues with reliable recording of data to possible hardware damage. Ultimately, planning ahead is likely to save time, and money.

---

## Chapter 5

# Resource Management

Information can be processed most efficiently when the appropriate resources are allocated, and utilized in an effective manner. In order to aid you, we'll go over some methods of determining your resource needs, and techniques to best utilize the resources available to you.

To some this process is a dark art. At its core, is the process of capacity planning. Capacity planning involves determining the amount CPU, RAM, and Disk your information processing needs. To achieve efficient resource utilization, you may need to optimize your work flow, and potentially leverage technologies like parallel processing. In order to ensure you are effectively using your computing capacity, you'll need to monitor your resource utilization at points throughout your work.

### 5.1 System Resources

The CPU or Central Processing Unit is the heart of your computer. As the name implies virtually all data processing is handled within the CPU. In simplest terms, a CPU's performance capability is measured in two ways. It's clock speed, that is the frequency the CPU operates at, which is measured in megahertz or gigahertz. And, the number of cores it has. Each core can perform a single operation (or calculation) at a time. So, the more cores you have, the more calculations that can be performed simultaneously.

RAM or Random Access Memory, is very fast, but short term memory. It's job is to hold the data that the CPU is actively working with. If your needs call for large amounts of RAM, you're in luck. It's relatively easy to upgrade, and fairly cheap.

Disks are used to store data for the long term. But, not all disks are created equally. There are two main types of disk, Solid State Drives, and Hard Disk Drives. An SSD is many times faster than a hard disk, but that comes with a much heftier price tag for large amounts of storage. Hard Disks on the other hand, have been around for years. They are cheap even for storing several terabytes of data.

With disks, there are 3 basic ways to utilize them. The most common is a stand alone drive, either internal or external to your computer. The next most common is network storage, where the disks live somewhere else on the network. Network storage is good for large capacity, but rarely good for high performance. The third, is a disk array, which is several disks working together usually in a redundant fashion so data is retained in the event of a disk failing.

### 5.2 Optimization

In order to optimize your data, and work-flow, you need to identify what resources you are using, identify bottlenecks, and eliminate them.

---

Every operating system has tools to track resource usage. On Windows, the Performance Monitor (example shown below) is the most helpful. It gives a breakdown of RAM, CPU, Disk, and Network utilization by application. On a Mac, the *Activity Monitor* is the most user friendly method of tracking resource usage. And, in the newest versions of OS X, it has a color coding scheme to help identify if you are hitting the limits. On Linux, you've got many choices, but `htop` is a solid choice for CPU and RAM monitoring. For disk activity, you'll want to use `iostat`.

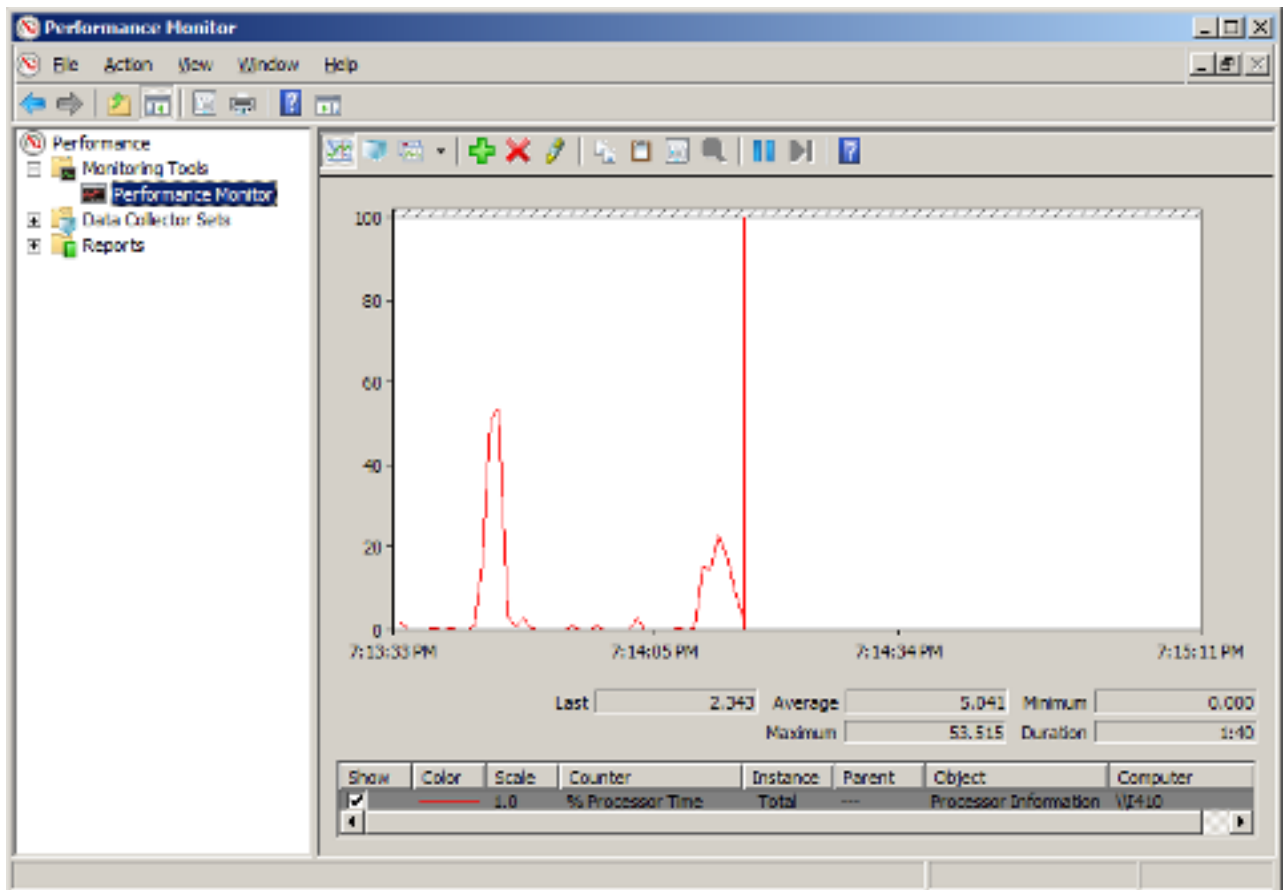


Figure 5.1: Windows 7 Performance Monitor: Microsoft

Once you have determined your usage, you can try to identify bottlenecks. A bottleneck could be caused by your available resources, or your software. If you aren't maxing out the CPU, Memory, and Disk, then the bottleneck is likely within the software itself. However, if you are maxing out a particular resource, then increasing the available resources should help. For example, if your system has a single hard disk drive, and it's being maxed out, replacing it with a solid state drive should speed things up.

### 5.3 Memory Utilization

Random Access Memory or RAM is a finite resource. The amount you can install into a system is limited by the CPU, motherboard, and sometimes the operating system you are running.

In order for a system to perform efficiently, you must use your available RAM in an effective manner. Which means you need to ensure that each application or process is allocated enough memory. While at the same time you must



avoid exceeding the available amount. When you exceed that limit, your computer will begin to swap or thrash. The result being poor performance, and a possible loss of data.

Allocating memory is made a bit more difficult due to the behaviour of some applications. Software such as R, MATLAB, and Excel by default do all of their data processing in RAM. Which means, you need enough memory to fit the raw data set, and any changes being made to it.

If you plan on working with large data sets, you have a few options. The first option is to purchase enough memory to do all your work in RAM. However, this option can be costly, and doesn't really scale well. Your second option is to break up the data into smaller pieces, and do your calculations in sections. The third option, is to use a plugin or add-on for your software that lets you work from disk. This option however isn't available for all software.

## 5.4 Case Study: Climate Change Project

To help provide context, we will look at a real research project conducted by the University of Washington.

Let's start the scenario with a support request made by a research assistant to the departmental IT staff. Here is the actual request.

```
I'm running a SAS program that involves building a large dataset from the meteorology files. Each meteorology file is 864 KB, and I'm merging together about 5,000 of these separate meteorology files into one giant dataset. So far I've been running this program since Wednesday, and it's still running. I'm thinking that perhaps my computer does not have enough processing memory to run this task efficiently. Also, I'm running this off of my C:/drive, so the slow processing time is not due to sending information over the network. I think that if I continue to let my program run over the weekend, it should definitely be done by Monday. However, I'm concerned because I will need to run this program again for another set of about 5,000 files later.
```

This provides sufficient detail to get an idea that this is a resource management problem. Either more memory needs to be installed, or the SAS program needs to be modified to use less memory.

Let's calculate the memory needed to load the entire dataset into RAM:

```
5000 * 864 KB = 4320000 KB
```

4320000 KB is over 4 GB of RAM. The computer running the analysis did not have that much RAM installed. So, the main reason this was running slowly was that the hard disk was used as "swap space" when the RAM had been consumed. This "swapping out" of memory is extremely slow.

But was it necessary to load the entire dataset into RAM? We'll look into that question in another section.

## 5.5 CPU Utilization

With modern CPUs having multiple cores, parallel processing is the only effective way to utilize all of the CPU power available. In order to utilize it, your data and work-flow may need adjustment. With parallel processing, your data is divided into pieces, and calculations are done on several pieces simultaneously.

Thankfully, there are some well developed tools and techniques to help with this. One of the more common is **MapReduce** which was popularized by Apache's Hadoop. MapReduce is a framework for processing large volumes

of data in parallel. Some lesser seen tools include **GNU Parallel** which is a tool used to run and manage command-line tools in a parallel fashion.

As an example, climate data can be processed in a parallel fashion. The data can be divided up by area, and then computation performed on a per area basis. Thus, instead of doing calculations for one ZIP code at a time, you could process data for 4, 8, or more areas at once.

## 5.6 Summary

In summary, there are three main components to resource management:

- Capacity planning: Identification and allocation of necessary resources.
- Utilization monitoring: Verifying you are using the resources you've allocated.
- Bottleneck resolution: Identification, and correction of performance bottlenecks.

## Chapter 6

# Data Files

### 6.1 Data Files Overview

- Data File Types: **Binary** vs. **plain text**
- **Filename extensions** and **File type associations**
- **ASCII**, **Unicode**, and **UTF-8**
- **Structured**, **Semi-structured**, and **Unstructured** data
- **XML** and **JSON**
- **Delimited** and **Multi-Line** file formats
- **Data file layout** and **Tidy Data**

### 6.2 Data File Types: Binary vs. Plain Text

There are essentially two main categories of digital file types, *binary* and *plain text*.

If a file is binary, the file just contains "zeros and ones". While this is technically true of any digital file stored within a binary computer system, the contents of a binary file does not conform to any standard **character encoding** system. The format may be highly efficient for storage or processing, but is essentially *opaque*, in that by simply looking at a binary file's contents, you can't really know what the format is or how to read it.<sup>1</sup> Examples of binary files are database files, multimedia files, and compressed files (such as `zip` files).

Plain text files, on the other hand, are composed of *characters*. Typically they are **ASCII** or **Unicode** characters represented by one or more bytes, where a byte is (generally) 8 bits. A bit can be considered either *zero* (off) or *one* (on). Plain text file formats are usually open and standard. Examples are web pages (**HTML**) as well as **XML**, and **CSV** (comma separated value) data files.

---

<sup>1</sup> Actually, byte sequences called **magic numbers** or **file signatures** may be used to identify file formats, but their use is not completely standardised or universal.

## 6.3 File Name Extensions

Filenames generally have an **extension**, which is the part at the end ("suffix") of the filename, consisting of the last dot (.) and the characters that follow it.<sup>2</sup>

Examples of binary filename extensions for images are `.png` and `.jpeg`. To launch "executable" programs on Windows systems you will often launch an `.exe` file. The `.dmg` ("disk image") filename extension is used on OS X. Common extensions for binary data files are `.xls` and `.sas7bdat`.

Plain text file formats for data files include `.csv`, `.tsv`, `.txt`, `.xml`, and `.json`, among others. Program source code is usually stored in plain text files, with extensions such as `.R`, `.py`, `.pl`, `.c`, `.sh`, `.bat`, and `.do`.

The extension is used to determine which "default application" should open it. Within the operating system, the extension is mapped to default applications. Mappings such as these are called **file type associations**.<sup>3</sup>

## 6.4 Viewing Binary and Plain Text Files

When viewing the *raw* contents of files, whether they are binary or text files, we will often make use of a *hexadecimal dump*.

**Hexadecimal** is a base-16 number system with digits 0-F:

```
0 1 2 3 4 5 6 7 8 9 A B C D E F
```

Whereas binary has two possibilities, 0 and 1, hexadecimal has 16, including the ten decimal digits plus the letters a-f.

Let's "dump" files in "hex" with **hexdump**...

```
$ hexdump -C -n 64 filename
```

Where the options we are using in this example are:

- `-C` = display in hex and ASCII
- `-n 64` = show the first 64 characters
- `filename` = name of file to view

In this example, we will view the first 64 bytes of an **SVG** image file. The file format stores information about the image in text, even though the file is displayed as a graphical image. Our filename is `pie.svg`.

```
$ hexdump -C -n 64 pie.svg
```

```
00000000  3c 3f 78 6d 6c 20 76 65 72 73 69 6f 6e 3d 22 31 |<?xml version="1|
00000010  2e 30 22 20 65 6e 63 6f 64 69 6e 67 3d 22 75 74 |.0" encoding="ut|
00000020  66 2d 38 22 3f 3e 0a 3c 21 44 4f 43 54 59 50 45 |f-8"?>.<!DOCTYPE|
00000030  20 73 76 67 20 50 55 42 4c 49 43 20 22 2d 2f 2f | svg PUBLIC "-//|
00000040
```

<sup>2</sup> *Filename extension*, [Wikipedia](#), CC BY-SA 3.0

<sup>3</sup> *file association*, [Wikipedia](#), CC BY-SA 3.0

We see a column of numbers on the left which show the character numbers in hexadecimal. Each line shows the hexadecimal number for each of 16 characters, in the center of the output, and on the right is the text equivalent (in "ASCII") of those character numbers.

We can see that this is an "XML" document with a version number, and the character encoding is shown as "UTF-8". The document type ("DOCTYPE") is "svg". All of this is contained in XML tags, similar in structure to HTML (the language of most web pages). The hexadecimal numbers correlate to the ASCII characters because the first 128 characters of the UTF-8 encoding scheme are the same as the ASCII character set. We go into more detail on this matter later in this chapter.

Even if the file was not a text file, and the ASCII printout looked like random characters, we would still be able to look at the hexadecimal dump to learn about the file.

For example, here is the **PNG** (binary) version of that same image. We will use the same syntax with hexdump, but look inside the `pie.png` file.

```
$ hexdump -C -n 64 pie.png
```

```
00000000  89 50 4e 47 0d 0a 1a 0a 00 00 00 0d 49 48 44 52 |.PNG.....IHDR|
00000010  00 00 01 2c 00 00 02 26 10 04 00 00 00 13 97 a3 |...,...&.....|
00000020  46 00 00 00 04 67 41 4d 41 00 00 b1 8f 0b fc 61 |F....gAMA.....a|
00000030  05 00 00 00 20 63 48 52 4d 00 00 7a 26 00 00 80 |....cHRM..z&...|
00000040
```

We have the same format of output. On the right, we see that the file is identified<sup>4</sup> as a PNG file, as shown in the first few ASCII characters, but all other ASCII characters appear random (meaningless). Dots are shown for "non-printing" characters. Since the file is binary, and not encoded as characters, the ASCII which has been interpreted by hexdump is not very useful for learning anything more about the image. We will just have to open the image in a graphics viewer to see what it is. Although both image files would display the same, you can see that there is a big difference between the contents of plain text and binary file formats.

## 6.5 Character Encodings

We will now take a closer look at the most popular character encodings for text files.

- **ASCII** (7-bit): The best-known standard for text.
- **Extended ASCII** (8-bit): The extra bit allows for a few more special symbols.
- **Unicode** (1-4 bytes): The current standard.
- Note: 8 **bits** per **byte**

### 6.5.1 ASCII

Some key points to know about ASCII are:

- "American Standard Code for Information Interchange"<sup>5</sup>
- **ASCII** standard first published in 1963

<sup>4</sup> The first few bytes of a file are often used to identify the file type.

<sup>5</sup> *ASCII*, [Wikipedia](#), [CC BY-SA 3.0](#)

- Current version of US ASCII is [ANSI X3.4-1986](#)
- ASCII was internationalized as [ISO 646:1983](#)
- 7-bit character set with 128 characters ( $2^7 = 128$ )

## 6.5.2 ASCII Table

The `ascii` command prints all 128 ASCII characters.

```
$ ascii
```

```
Usage: ascii [-dxohv] [-t] [char-alias...]
  -t = one-line output  -d = Decimal table  -o = octal table  -x = hex table
  -h = This help screen -v = version information
Prints all aliases of an ASCII character. Args may be chars, C \-escapes,
English names, ^-escapes, ASCII mnemonics, or numerics in decimal/octal/hex.
```

Dec	Hex	Dec	Hex	Dec	Hex	Dec	Hex	Dec	Hex	Dec	Hex	Dec	Hex	Dec	Hex								
0	00	NUL	16	10	DLE	32	20	48	30	0	64	40	@	80	50	P	96	60	`	112	70	p	
1	01	SOH	17	11	DC1	33	21	!	49	31	1	65	41	A	81	51	Q	97	61	a	113	71	q
2	02	STX	18	12	DC2	34	22	"	50	32	2	66	42	B	82	52	R	98	62	b	114	72	r
3	03	ETX	19	13	DC3	35	23	#	51	33	3	67	43	C	83	53	S	99	63	c	115	73	s
4	04	EOT	20	14	DC4	36	24	\$	52	34	4	68	44	D	84	54	T	100	64	d	116	74	t
5	05	ENQ	21	15	NAK	37	25	%	53	35	5	69	45	E	85	55	U	101	65	e	117	75	u
6	06	ACK	22	16	SYN	38	26	&	54	36	6	70	46	F	86	56	V	102	66	f	118	76	v
7	07	BEL	23	17	ETB	39	27	'	55	37	7	71	47	G	87	57	W	103	67	g	119	77	w
8	08	BS	24	18	CAN	40	28	(	56	38	8	72	48	H	88	58	X	104	68	h	120	78	x
9	09	HT	25	19	EM	41	29	)	57	39	9	73	49	I	89	59	Y	105	69	i	121	79	y
10	0A	LF	26	1A	SUB	42	2A	*	58	3A	:	74	4A	J	90	5A	Z	106	6A	j	122	7A	z
11	0B	VT	27	1B	ESC	43	2B	+	59	3B	;	75	4B	K	91	5B	[	107	6B	k	123	7B	{
12	0C	FF	28	1C	FS	44	2C	,	60	3C	<	76	4C	L	92	5C	\	108	6C	l	124	7C	
13	0D	CR	29	1D	GS	45	2D	-	61	3D	=	77	4D	M	93	5D	]	109	6D	m	125	7D	}
14	0E	SO	30	1E	RS	46	2E	.	62	3E	>	78	4E	N	94	5E	^	110	6E	n	126	7E	~
15	0F	SI	31	1F	US	47	2F	/	63	3F	?	79	4F	O	95	5F	_	111	6F	o	127	7F	DEL

You will see that there is a header showing the command usage followed by an ASCII table listing. The listing is arranged in 8 sets of columns, with each set showing the decimal (Dec) and hexadecimal (Hex) value for each character. Starting from zero (0), the first 32 characters (and the 128th) are the so-called "non-printing" characters, so those are shown with 2-3 letter codes describing the character. The 33rd character is the "Space" so nothing is shown. All other characters are symbols which appear on the standard [US keyboard](#). The punctuation characters and decimal digits are followed by capital letters, more punctuation, lower-case letters, more punctuation, and finally ending with "DEL" (Delete), the 128th character (numbered 127, or 7F in hexadecimal). To have more characters, we would need more bits in our encoding standard, which we will look into next.

## 6.5.3 Extended ASCII

- [ISO-8859-1](#) is an 8-bit extension with 191 characters
- ISO-8859-1 ("ISO Latin 1") was first published in 1987

A0	A1	A2	A3	A4	A5	A6	A7	A8	A9	AA	AB	AC	AD	AE	AF
B0	B1	B2	B3	B4	B5	B6	B7	B8	B9	BA	BB	BC	BD	BE	BF
C0	C1	C2	C3	C4	C5	C6	C7	C8	C9	CA	CB	CC	CD	CE	CF
D0	D1	D2	D3	D4	D5	D6	D7	D8	D9	DA	DB	DC	DD	DE	DF
E0	E1	E2	E3	E4	E5	E6	E7	E8	E9	EA	EB	EC	ED	EE	EF
F0	F1	F2	F3	F4	F5	F6	F7	F8	F9	FA	FB	FC	FD	FE	FF

Figure 6.1: ISO-8859-1 (Latin1) - Image: Roman Czyborra

### 6.5.3.1 Windows Latin 1 (Windows-1252)

- ISO-8859-1 was extended to [Windows-1252](#)
- Windows-1252 is sometimes (incorrectly) called "ANSI"<sup>6</sup>

80	81	82	83	84	85	86	87	88	89	8A	8B	8C	8D	8E	8F
90	91	92	93	94	95	96	97	98	99	9A	9B	9C	9D	9E	9F
A0	A1	A2	A3	A4	A5	A6	A7	A8	A9	AA	AB	AC	AD	AE	AF
B0	B1	B2	B3	B4	B5	B6	B7	B8	B9	BA	BB	BC	BD	BE	BF
C0	C1	C2	C3	C4	C5	C6	C7	C8	C9	CA	CB	CC	CD	CE	CF
D0	D1	D2	D3	D4	D5	D6	D7	D8	D9	DA	DB	DC	DD	DE	DF
E0	E1	E2	E3	E4	E5	E6	E7	E8	E9	EA	EB	EC	ED	EE	EF
F0	F1	F2	F3	F4	F5	F6	F7	F8	F9	FA	FB	FC	FD	FE	FF

Figure 6.2: Windows-1252 (WinLatin1) - Image: Roman Czyborra

### 6.5.4 Why Character Encoding Matters

If a file is created using one character encoding, but is viewed using another, the characters are likely to display incorrectly. The resulting garbled text is sometimes called [mojibake](#).

<sup>6</sup> [Windows-1252](#), [Wikipedia](#), CC BY-SA 3.0

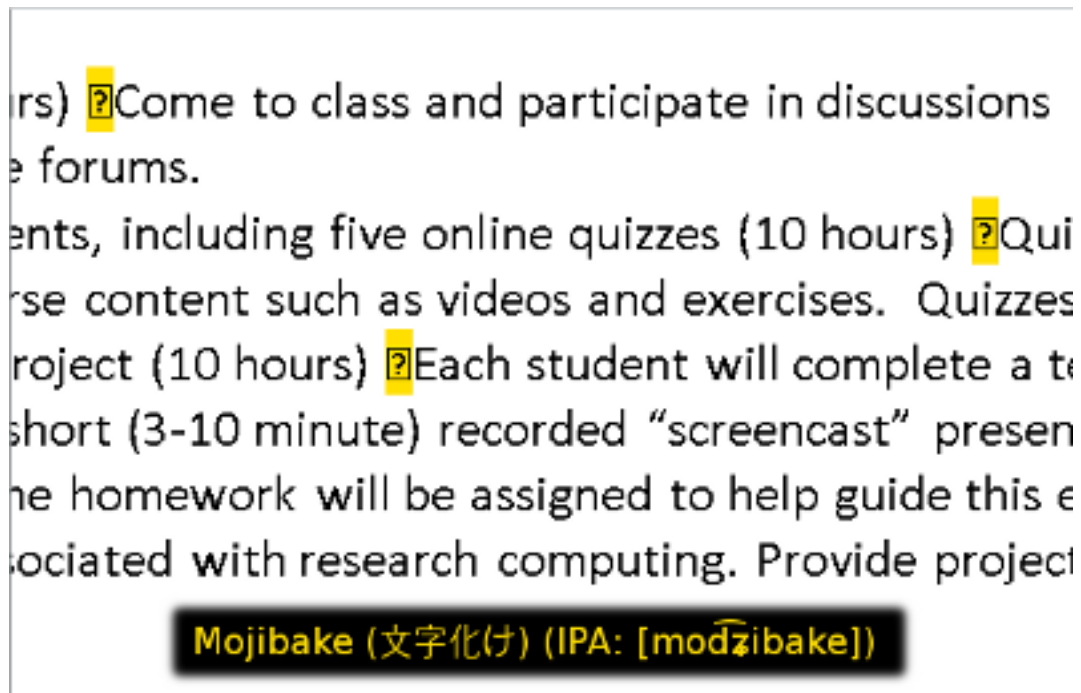


Figure 6.3: Mojibake example in MS-Word

We can see how differences in character encodings can matter with a few simple examples. Let's first generate a table of characters with Python.

### Example 6.1 Printing the Windows-1252 character set with Python

The following Python script will show the printable characters of the Windows-1252 character set when run on a Windows system using a graphical Python interpreter such as *IDLE* or *PyScripter*.

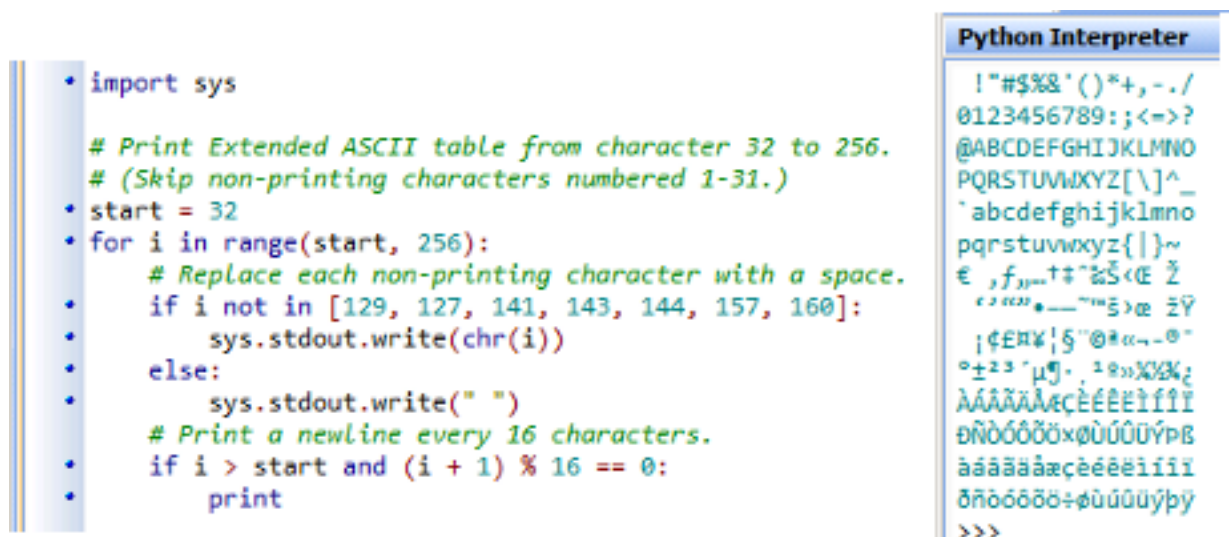


Figure 6.4: Windows-1252 Table Python Script



Here is the full code listing for that Python script.

### asciitable.py

```
# If run on a Windows system in a graphical environment such as
# IDLE's Python Shell, by default, this will print the Windows
# Latin 1 character set, a.k.a. Windows-1252 (WinLatin1).

import sys

# Print Extended ASCII table from character 32 to 256.
# (Skip non-printing characters numbered 1-31.)
start = 32
for i in range(start, 256):
    # Replace each non-printing character with a space.
    if i not in [129, 127, 141, 143, 144, 157, 160]:
        sys.stdout.write(chr(i))
    else:
        sys.stdout.write(" ")
    # Print a newline every 16 characters.
    if i > start and (i + 1) % 16 == 0:
        print
```

### Example 6.2 Changing the Character Encoding within your application

We can see the characters properly in a non-Windows environment if we specifically set the character encoding in the application.

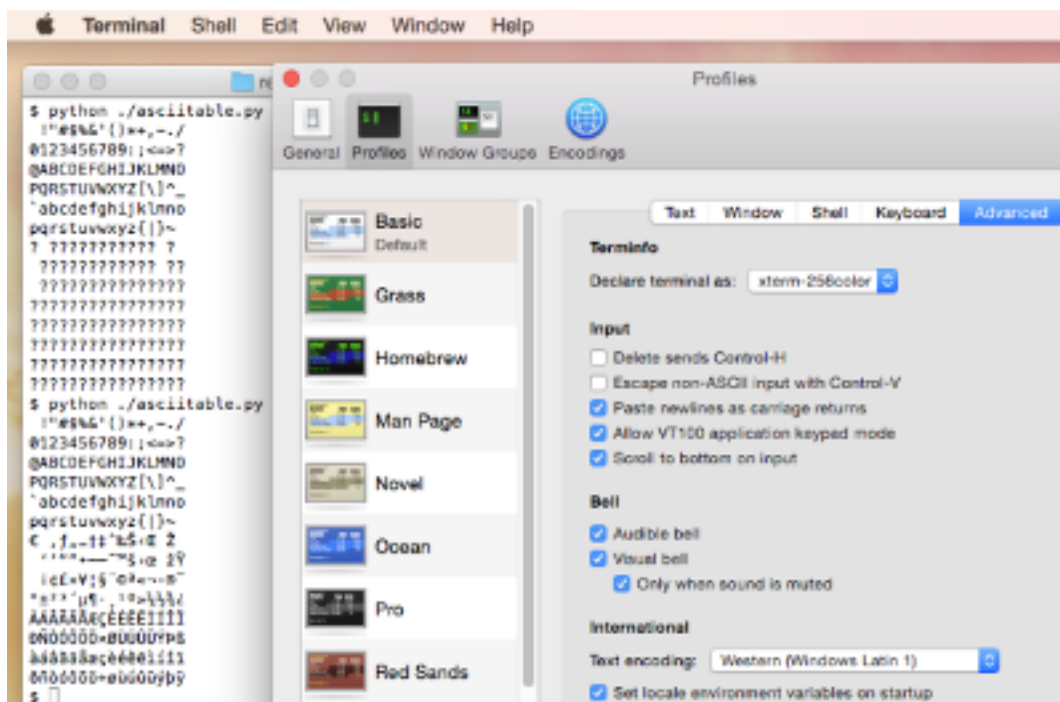


Figure 6.5: Changing Character Encoding in Mac OS X Terminal



```
$ file asciitable.txt
asciitable.txt: Non-ISO extended-ASCII text, with CRLF, NEL line terminators
```

While this tells us a little about the text format, we still don't know the specific encoding standard used.

As you can see, dealing with various character encodings on different computing systems can be tricky.<sup>8</sup> Is there a universal character encoding standard? Yes, *Unicode*! In the next section, we'll see how we can convert our file to Unicode.<sup>9</sup>

### 6.5.5 Unicode

**Unicode** provides an **internationalized** character encoding **standard**, to "encompass the characters of all the world's living languages".<sup>10</sup>

- Like ASCII, but supports over 110,000 characters
- Unicode standard was published in 1991
- Most commonly used **encodings** are UTF-8 and UTF-16<sup>11</sup>

You can browse the Unicode **code charts** to get an idea of the many character sets available.

#### 6.5.5.1 Unicode Symbol Example: the Micro Sign

##### Example 6.5 Encoding the Micro Sign

The character  $\mu$ , with Unicode<sup>12</sup> **name** "MICRO SIGN" is encoded:

Encodings	Decimal	Hex
Unicode	181	U+00B5
Extended ASCII	181	B5
HTML <b>numeric</b> character reference	&#181;	&#xB5;
HTML <b>named</b> character entity	&micro;	

##### Example 6.6 Typing the Micro Sign

How do you type the  $\mu$  character into your computer?

Use these character codes:

Name	Decimal	Hex
MICRO SIGN	181	00B5

<sup>8</sup> You will find many cases of this issue discussed on help forums such as [stackoverflow](#). There are modules in programming languages such as [Ruby](#) and [Python](#) which help address these problems. Applications like MS-Word also allow you to set or **convert** encodings.

<sup>9</sup> For a more thorough treatment of dealing with character encoding problems, see "Bad Data Lurking in Plain Text" by Josh Levy, PhD, which is the fourth chapter of Q. Ethan McCallum's *Bad Data Handbook* (O'Reilly Media, Inc., 2012).

<sup>10</sup> Joe Becker, [Unicode 88](#)

<sup>11</sup> *Unicode*, [Wikipedia](#), CC BY-SA 3.0

<sup>12</sup> *Unicode*, [Wikipedia](#), CC BY-SA 3.0

With these operating systems:<sup>13</sup>

- **Windows:** [Alt]**decimal** (using numeric keypad) ... *or* ... **hex**[Alt][x] (does not require numeric keypad)
- **OS X:** for  $\mu$ , you can simply use [Opt][m] ... *or* ... [Command][Ctrl][Space] ... Search by **name** ... *or* ... use **Unicode Hex Input** (Input Source) and **hex**
- **Linux:** [Shift][Ctrl]**hex**

### 6.5.5.2 Some Other Useful Symbols

Table 6.1: HTML Entities for Common Math Symbol Characters

Character Name	Char.	Entity	Num. Entity	Hex. Entity
DEGREE SYMBOL	°	&deg;	&#176;	&#xB0;
MICRO MU SYMBOL	$\mu$	&micro;	&#181;	&#xB5;
LOWER CASE SIGMA	$\sigma$	&sigma;	&#963;	&#x3C3;
N-ARY SUMMATION	$\Sigma$	&sum;	&#8721;	&#x2211;
GREEK SMALL LETTER PI	$\pi$	&pi;	&#960;	&#x3C0;
GREEK SMALL LETTER ALPHA	$\alpha$	&alpha;	&#945;	&#x3B1;
GREEK SMALL LETTER BETA	$\beta$	&beta;	&#946;	&#x3B2;
GREEK SMALL LETTER GAMMA	$\gamma$	&gamma;	&#947;	&#x3B3;
INCREMENT	$\Delta$	&Delta;	&#8710;	&#x2206;
GREEK SMALL LETTER EPSILON	$\epsilon$	&epsilon;	&#949;	&#x3B5;
INFINITY	$\infty$	&infin;	&#8734;	&#x221E;
PLUS OR MINUS	$\pm$	&plusmn;	&#177;	&#xB1;
NOT EQUALS	$\neq$	&ne;	&#8800;	&#x2260;
ALMOST EQUAL	$\asymp$	&asymp;	&#8776;	&#x2248;
GREATER THAN OR EQUAL TO	$\geq$	&ge;	&#8805;	&#x2265;
LESS THAN OR EQUAL TO	$\leq$	&le;	&#8804;	&#x2264;
DIVISION SIGN	$\div$	&divide;	&#247;	&#xF7;
SUPERSCRPT TWO	$^2$	&sup2;	&#178;	&#xB2;
SUPERSCRPT THREE	$^3$	&sup3;	&#179;	&#xB3;

For example, in Windows, you can use the "Num. Entity" column for [Alt] codes such as [Alt]946 for  $\beta$  (beta).

### 6.5.6 UTF-8: Encoding the Unicode Code Space

- **UTF-8** (1993) is a variable-length 8-bit character encoding
- A UTF-8 character will use one to four 8-bit bytes
- ASCII characters are the first 128 characters of UTF-8
- Use of UTF-8 surpassed ASCII on the Web in Dec. 2007

<sup>13</sup> Unicode input, [Wikipedia](#), CC BY-SA 3.0

- UTF-8 is the default encoding for HTML5 and JSON

UTF-8 and UTF-16 are the standard encodings for Unicode text in HTML documents, with UTF-8 as the preferred and most used encoding.<sup>14</sup>

— Wikipedia *UTF-8*

#### 6.5.6.1 Character Encoding Conversion Example

We can convert a file encoded as Windows-1252 into UTF-8 with `iconv`.<sup>15</sup>

---

**Example 6.7** Converting a Windows-1252 file into UTF-8

---

```
$ iconv -f windows-1252 -t utf-8 asciitable.txt > asciitable2.txt
$ file asciitable2.txt
asciitable2.txt: UTF-8 Unicode text
```

---

As you can see, you can use `file` to verify that this is a Unicode file encoded as UTF-8.

---

**Tip**

"Normalize" text datafiles to a common, universal encoding format like UTF-8 to ensure characters are displayed with the intended symbols.

---

## 6.6 Data Structure

- **Structured**: Formal and rigorous design
  - Example: **Relational database**
- **Semi-structured**: **Self-describing**, validatable
  - **Markup** using **tags** or **key-value pairs**
  - Examples: **XML** and **JSON**
- **Unstructured**:
  - Multimedia and text document files
  - Any internal structure, if present, is assumed or unreliable
  - Example: email "body" ("header" is semi-structured)
  - May have "implied" structure, like "delimited text"

---

<sup>14</sup> UTF-8, Wikipedia, CC BY-SA 3.0

<sup>15</sup> `iconv` is another tool originally developed for Unix, Linux and OS X systems, though Windows versions are available and can be found with an Internet search.

## 6.7 XML

- Self-describing<sup>16</sup>
- Structured
- Standard
- Parsable with libraries
- Examples:
  - XHTML
  - KML
  - XLSX
  - p-list
  - SVG



Figure 6.7: XML - Image: Dreftymac, CC BY 2.5

## 6.8 JSON

- JavaScript Object Notation
- Open format (ISO and ECMA standards)

---

<sup>16</sup> XML, Wikipedia, CC BY-SA 3.0

---

- Human-readable text
- For transmitting data objects
- Attribute–value pairs
- Often used in **Ajax** web applications

---

**Example 6.8** JSON data structure for a person (John Smith)<sup>17</sup>

```
{
  "firstName": "John",
  "lastName": "Smith",
  "isAlive": true,
  "age": 25,
  "height_cm": 167.6,
  "address": {
    "streetAddress": "21 2nd Street",
    "city": "New York",
    "state": "NY",
    "postalCode": "10021-3100"
  },
  "phoneNumbers": [
    {
      "type": "home",
      "number": "212 555-1234"
    },
    {
      "type": "office",
      "number": "646 555-4567"
    }
  ],
  "children": [],
  "spouse": null
}
```

---

## 6.9 Delimited Text Files

Files formatted with **delimiter separated values** use:

- Comma (e.g., "CSV")
- Tab (e.g., "TSV")
- Pipe (vertical bar: |)

... or other single character as a separator between values.

The records (rows) are separated by **line-ending** characters (newlines):

- Carriage-return (CR)
  - Line-feed (LF)
  - Carriage-return, Line-feed (CRLF)
-

## 6.10 Fixed-Width Text Files

- Text files arranged in neatly formatted columns
- Space filled with varying numbers of spaces or tabs
- Easier to look at, but a little harder to parse
- Lines are separated with newlines

	mpg	cyl	disp
Mazda RX4	21.0	6	160
Mazda RX4 Wag	21.0	6	160
Datsun 710	22.8	4	108
Hornet 4 Drive	21.4	6	258
Hornet Sportabout	18.7	8	360
Valiant	18.1	6	225

(Data from `mtcars`, *The R Datasets Package*, R Core Team.)<sup>18</sup>

## 6.11 Multi-line Text Files

Some popular genomics file formats use multi-line records.

- FASTA<sup>19</sup>

>gi|5524211|gb|AAD44166.1| cytochrome b [Elephas maximus maximus]  
LCLYTHIGRNIYYGSYLYSETWNTGIMLLLLITMATAFMGYVLPWQMSFWGATVITNLFSAIPYIGTNLV  
EWIWGGFSVDKATLNRFFAFHFILPFTMVALAGVHLTFLHETGSNNPLGLTSDSDKIPFHPYYTIKDFLG  
LLILILLLLLLALLSPDMLGDPDNHMPADPLNTPLHIKPEWYFLFAYAILRSVPNKLGGVVALFLSIVIL  
GLMPFLHTSKHRSMMLRPLSQALFWTLTMDLLTLTWIGSQPVEYPYTIIGQMASILYFSIILAFPLIAGX  
IENY

- FASTQ<sup>20</sup>

```
@SRR001666.1 071112_SLXA-EAS1_s_7:5:1:817:345 length=36  
GGGTGATGGCCGCTGCCGATGGCGTCAAATCCCACC  
+SRR001666.1 071112_SLXA-EAS1_s_7:5:1:817:345 length=36  
IIIIIIIIIIIIIIIIIIIIIIIIIIIIIIII9IG9IC
```

## 6.12 Data File Layout: Tidy Data

## Structure data files as simple "columns and rows" ...

<sup>18</sup> Extracted from the 1974 *Motor Trend* US magazine. Source: Henderson and Velleman (1981), Building multiple regression models interactively. *Biometrics*, 37, 391–411.

<sup>19</sup> *FASTA format*, [Wikipedia](#), CC BY-SA 3.0

<sup>20</sup> *FASTQ format*, [Wikipedia](#), CC BY-SA 3.0



subID	height	weight
1	58	115
2	59	117
3	60	120

... to make them easier to import and analyze.


(Data from [women](#), *The R Datasets Package*, R Core Team.)<sup>21</sup>

The basic tenets of *tidy data* are:<sup>22</sup>

- Each variable forms a column.
- Each observation forms a row.
- Each type of observational unit forms a table

## 6.13 Data File Layout: Tidy Data

Is this spreadsheet<sup>23</sup> *tidy data* or not? Why or why not?

 <b>World Health Organization</b> <b>Organisation Mondiale de la Santé</b> Department of Measurement and Health Information December 2004		Table 1. Estimated total deaths ('000), by cause and WHO Member State, 2002 (a)						
GBD code	GBD cause (b)	Afghanistan	Albania	Algeria	Andorra	Angola	Antigua and Barbuda	Argentina
	<i>Data sources - level of evidence</i>							
	All cause mortality (c)	Level 4b	Level 2b	Level 3b	Level 3b	Level 4b	Level 3b	Level 1a
	Cause-specific mortality (d)	Level 4	Level 2b	Level 4	Level 4	Level 4	Level 2b	Level 2a
	Population ('000) (e)	22,930	3,141	31,266	69	13,184	73	37,981
W000	All Causes	484.5	22.1	173.3	0.6	306.6	0.6	281.4
	Lower uncertainty bound (f)	347.6	19.3	158.0	0.5	255.3	0.5	276.6
	Upper uncertainty bound (f)	695.8	28.0	188.4	0.6	434.8	0.6	286.6
W001	I. Communicable, maternal, perinatal and	317.2	1.8	56.9	0.0	230.3	0.1	36.2
W002	A. Infectious and parasitic diseases	169.3	0.2	30.2	0.0	143.0	0.0	15.6
W003	1. Tuberculosis	21.1	0.0	0.6	0.0	4.8	-	0.8
W004	2. STDs excluding HIV	0.3	0.0	2.1	0.0	2.3	0.0	0.0
W005	a. Syphilis	0.1	0.0	2.0	0.0	2.3	0.0	0.0
W006	b. Chlamydia	0.1	-	0.0	-	0.0	-	-
W007	c. Gonorrhoea	0.0	-	0.0	-	0.0	-	-
W009	3. HIV/AIDS	0.0	0.0	0.3	0.0	21.1	0.0	1.8
W010	4. Diarrhoeal diseases	41.2	0.0	8.1	0.0	48.8	0.0	0.4
W011	5. Childhood-cluster diseases	15.5	0.0	4.1	0.0	12.4	0.0	0.0
W012	a. Pertussis	6.6	-	0.6	-	4.2	-	0.0
W013	b. Poliomyelitis	0.0	-	-	0.0	-	-	-
W014	c. Diphtheria	0.0	-	0.0	-	0.1	-	-
W015	d. Measles	2.2	-	3.4	-	6.2	-	-
W016	e. Tetanus	6.7	0.0	0.0	0.0	1.9	0.0	0.0
W017	6. Meningitis	8.3	0.0	0.2	0.0	0.6	0.0	0.4
W018	7. Hepatitis B (g)	1.0	0.0	0.5	0.0	0.8	0.0	0.1
W019	8. Hepatitis C (g)	0.5	0.0	0.2	0.0	0.4	0.0	0.1

## 6.14 Wide and Long: Which Table is Tidier?

<sup>21</sup> *The World Almanac and Book of Facts*, 1975. Reference: McNeil, D. R. (1977) *Interactive Data Analysis*. Wiley.

<sup>22</sup> Hadley Wickham, *Tidy Data*

<sup>23</sup> WHO

Table 6.2: Iris data in "wide" format

Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Species
5.1	3.5	1.4	0.2	setosa
5.7	2.8	4.1	1.3	versicolor

Table 6.3: Iris data in "long" format

Flower.Id	Species	Flower.Part	Length	Width
1	setosa	Petal	1.4	0.2
1	setosa	Sepal	5.1	3.5
100	versicolor	Petal	4.1	1.3
100	versicolor	Sepal	5.7	2.8

(Data from [iris](#), *The R Datasets Package*, R Core Team.)<sup>24</sup> ,

## 6.15 Why does tidiness matter?

Now we can "facet" a plot by Species and Flower.Part.

```
ggplot(data=iris, aes(x=Width, y=Length)) +  
  geom_point() + facet_grid(Species ~ Flower.Part, scale="free") +  
  geom_smooth(method="lm") + theme_bw(base_size=16)
```

---

<sup>24</sup> Anderson, Edgar (1935). The irises of the Gaspé Peninsula, *Bulletin of the American Iris Society*, 59, 2–5. Reference: Fisher, R. A. (1936) The use of multiple measurements in taxonomic problems. *Annals of Eugenics*, 7, Part II, 179–188.

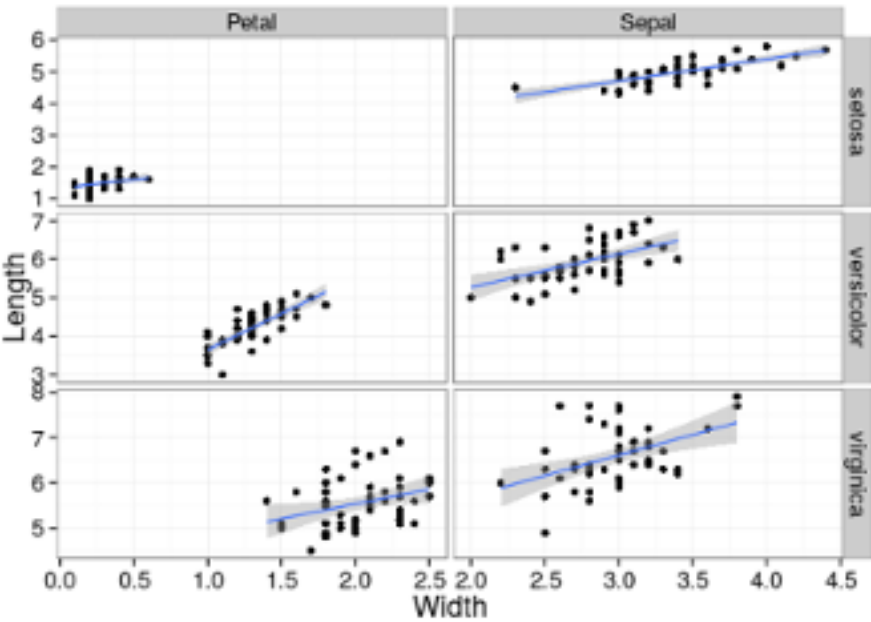


Figure 6.8: Plot using tidy iris data

## Chapter 7

# Databases

Research produces data, and information that need to be stored for future review or processing. Databases are one such solution, offering an integrated system of information storage, and management. There are a variety of database systems on the market, each with their own pros and cons.

Databases come in all shapes and sizes. In general, there are three basic types. The *flat file*, which is much like a spreadsheet. *Relational*, which is like a series of spreadsheets linked by common identifiers. And finally, the family of *NoSQL* databases, which have various storage models for different use cases.

Much like the variety of database types, there are a variety of query languages. A query language is how you interact with the database. With it, you can insert new data, update records, etc. We'll go over some specific languages like Structured Query Language later in the section.

- Types
  - Flat
  - Relational
  - NoSQL (various storage models)
- Query Languages
  - SQL
  - CQL
  - Etc.

### 7.1 Key Terms

Before we can discuss the details of each database type, and their related query languages, we have to get some key terms out of the way. These terms will come up again later, in addition to being found in numerous database books, and articles.

**CRUD** represents the four most basic database operations. Create refers to creating new entries, or adding new data to the database. Read, as the name implies is retrieving data from the database. Update is simply changing the contents of an existing database record. And, Delete, is the removal of a record.

**ACID** is a method of ensuring that data is stored reliably. Atomicity or an atomic operation, means either the transaction or query is completed fully, or not completed at all. In other words, if the query fails, the database must not be

changed. Consistency means that database must remain in a valid state. If the database has any data validation rules, all data must be in compliance with them. Isolation is important in multi-user databases. With Isolation, each users transactions must not interfere with each other, so that each user has a consistent view of the database. Durability means the data has been committed to disk. So, that in the event of a power failure, crash, etc. your data is safe.

**Normalization** is all about removing redundant data. For example instead of storing a customer's contact information with their order. The order would refer to a customer ID number, which refers to their contact information. Thus, multiple orders can refer to the same data.

**CAP Theorem** is mostly relevant to distributed NoSQL databases. That is databases which are spread across many servers. It states that you can not deliver a consistent database view, ensure that every query is responded to, and tolerate a failure in the system at the same time. In other words, you must compromise in one of those areas, to guarantee the other two.

**Database Schema** is a formal description of the database structure. It defines the tables, data types, and other rules governing the database.

## 7.2 Flat File

### Flat File Model

	Route No.	Miles	Activity
Record 1	I-95	12	Overlay
Record 2	I-495	05	Patching
Record 3	SR-301	33	Crack seal

Figure 7.1: Flat File Model: US Department of Transportation, Public Domain

**Flat file** databases are extremely simple. They are simply a single table of values, just like a spreadsheet. The actual data format varies, there are proprietary formats like Excel, but open standards like CSV or comma separated values, are widely seen.

Due to their simple nature, they lack a number of features found in more capable database systems. For example, they lack indexing, which means that searching a large flat file could require considerably more time than a relational database. They also lack concurrent write access, since they aren't server based. Finally, updates can be extremely slow, as they require rewriting the entire file to disk after making a change.

All these limitations aside, there are some benefits. Flat files can be very lightweight in resource usage, at least when they are small in size. For databases using CSV or similar formats, you can even edit the database using a basic text editor.

Despite the limitations, there are situations where a flat file is a reasonable choice. For example, recording sensor data from a temperature and pressure sensor. This data tends to consist of timestamped rows with sensor readings, with each file named after the sensor.

That said, it's generally best to not go with a flat file database for storage. If at some point you find that your needs change, converting to a more capable format, could become a rather daunting task.

- Lightweight to a point
- Limited scalability
- No built in concurrent access
- No indexing
- Lacks built in consistency checking

## 7.3 Relational

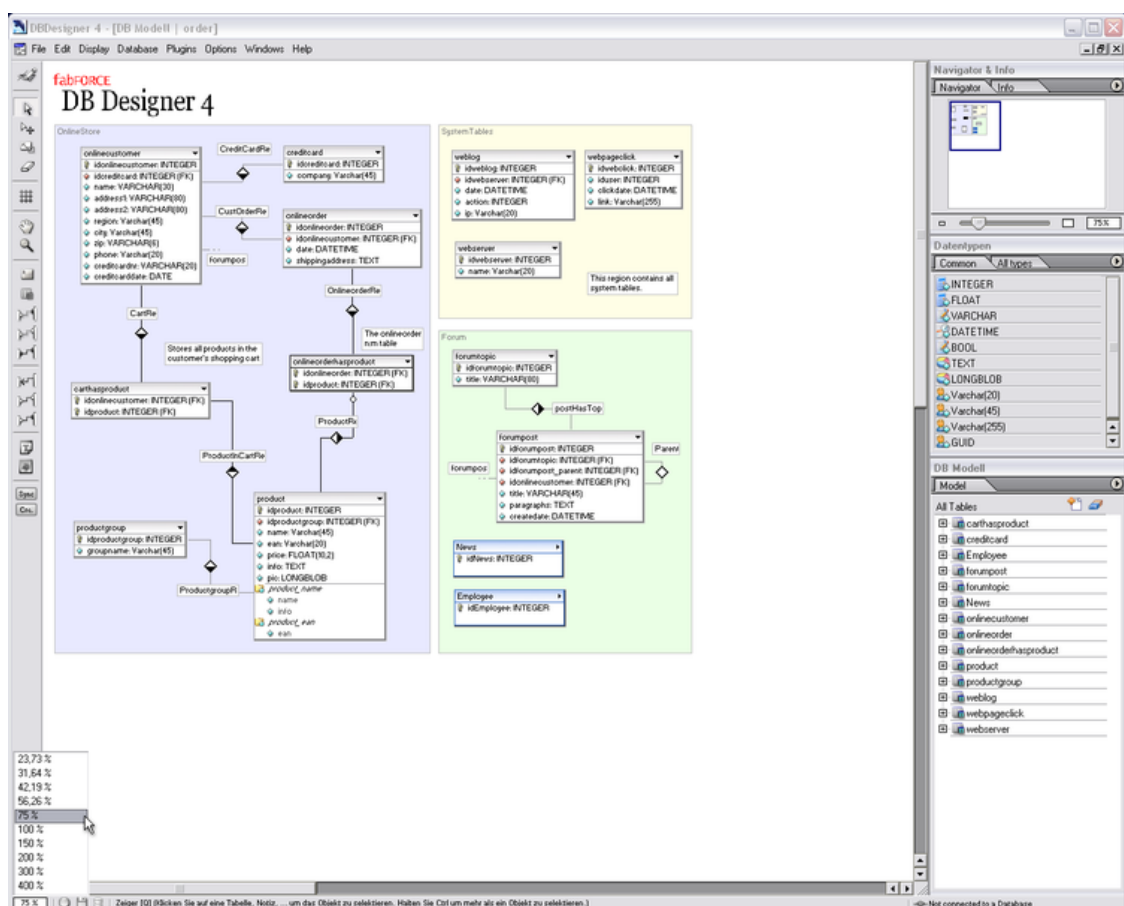


Figure 7.2: fabFORCE.net DBDesigner4: MichaelGZinner, CC BY SA 3.0

**Relational** databases, as their name implies, relate data from one table to another. These relations reduce the duplication of data that is common to multiple records. For example, in an ordering system, you could store your customers within one table, and link their orders to them via a customer identification number.

Relational databases are basically the jack of all trades in the database world. They are highly flexible enabling them to meet the widest number of use cases, such as recording survey results. But, that flexibility comes at a price. You have to tell the database about the data you plan to store. In other words, a relational database requires a schema to define the structure of your data, and how it relates.

Virtually all of them support **ODBC or Open Database Connectivity**, which is a standardized interface between an operating system or programming language and the database driver. Through ODBC, you can plug your database into a huge number of applications, such as *R* or even *Excel*.

There are variety of relational database engines on the market, some open source, others proprietary. The most common open source database is *MySQL*, which is now owned by Oracle. For those with very small needs, there is *SQLite* which is commonly embedded into other applications like Firefox and Android. Even Microsoft has an offering, in the form of Microsoft *SQL Server*.

---

**Example 7.1** Does climate impact human health?

A database can be used to store any kind of data. One group had the challenge of analyzing a large volume of climate data to compare with health records. The health care data was location coded, in a similar fashion to the climate data that was being used. Through the use of *MySQL*, they were able to relate health care records by location with the climate data. Thus enabling them to query the data, and look for patterns.

---

So, which should you choose? My advice is to pick an open source option, like *MySQL* or *PostgreSQL*. They have low minimum resources making them easy to setup on a laptop for testing purposes. Documentation is widely available. And, more importantly, their free nature means, a lot of folks have taken the time to integrate them into a huge array of tools.

That said, different engines make sacrifices to make gains in other areas. This commonly occurs in the area of **ACID** compliance. A database may forgo strict data validation, or consistent disk writing to speed up query response time. If you value your data over speed, be sure to pick an **ACID** compliant database.

In short, if you aren't sure what kind of database to use, an **ACID** compliant relational database is a good start. It's typically easier to migrate from a relational database to other types, as opposed to converting to one.

- General purpose
- Wide application support through ODBC
- Less duplicate data

## 7.4 NoSQL

**NoSQL or "Not Only SQL"** databases serve a variety of niches. Each implementation is geared towards a limited set of use cases, in other words, they don't try to be general purpose databases. NoSQL databases are most commonly associated with **Big Data**, and **High Performance Computing**. By Big Data, we're talking about databases in the multi terabyte and larger range. But, not only are they big, they generally require high throughput either from thousands of concurrent requests or simply doing a query across a few billion rows of data in a short period of time.

Within the NoSQL arena, there are a lot of options. A couple commonly used are Apache's *Cassandra*, which functions similarly to a traditional database. It has tables, which consist of rows and columns. However, the method you query across multiple tables is a bit different than a relational database. *mongoDB* on the other hand, is known as

---

a document store. A document being an object containing a series of values. Where a value could be as simple as a name, to an entire PDF. In addition, objects can be related to each other through their unique IDs.

Much like there are a number of NoSQL databases, each excelling in their own ways, there are a variety of **storage models**. A storage model is how the data is actually stored within the database. Each storage method is best suited to select use cases, or applications. You'll want to consult the documentation for each database to see what it's most suited for. You may also want to look to sites like **BioStar** to see what others in your field are using to solve their problems.

With all this variation in data storage methods, there has to be a catch. While many relational databases offer ACID compliance, NoSQL databases rarely do. They make many sacrifices in order to scale massively. The guiding rule with NoSQL is the CAP Theorem which basically says you can't ensure every server shows the exact same data, handle every query, and not lose access to something in the event of a crash.

---

**Example 7.2** Collecting data from remote sensors.

While NoSQL implies Big Data, it doesn't require it. NoSQL databases lend themselves to solving a variety of problems. One group was tasked with performing remote data collection using Internet connected sensors. Each of these sensors sends data back in 30 second intervals to KairosDB. KairosDB is a time-series database, designed for tracking sensor data over large periods of time, and from a large number of sensors. Being a purpose specific database, it's easily integrated with tools used for trend analysis.

---

So, which NoSQL option to go with? That isn't an easy question to answer. Since each database system varies from the next, you have to closely evaluate your needs. You need to consider what questions you are asking from your data. And, also factor in, what is your data. For example, is it a huge table of values, or millions of documents. Finally, look at the other tools you are using. They may include support for one or maybe two options. In which case, you are either stuck with those choices, or reconsider the tools being used.

## 7.5 Query Languages

A database isn't very useful if you don't have a way to enter, and retrieve data from it. A query language enables you to enter data, select data based on criteria, change existing data, and even remove it.

The most prevalent is SQL or *Structured Query Language* which is a standardized language implemented by most relational databases. However, it's not alone in that arena. Some databases such as *Oracle*, and PostgreSQL have their own add-ons called *Procedural Languages* which enable you to leverage scripting languages like *Perl* within the database engine itself.

Outside of the relational database world, the languages differ from one database engine to the next. For example, Apache's Cassandra offers *Cassandra Query Language* which has an SQL inspired syntax.

## 7.6 Structured Query Language

**SQL or Structured Query Language** is the most widely used database query language. With SQL, you can query for data, create new databases, etc. At the core, SQL is an ANSI standardized language, enabling multiple database vendors to implement it. That said, SQL itself doesn't meet everyone's needs, resulting in proprietary extensions to it.

When possible, it's best to avoid the use of proprietary extensions. Otherwise, in the future should you need to change your database back-end, for example switching from MySQL to PostgreSQL. You may find yourself rewriting a lot of you queries, and adjusting your database schema.

---



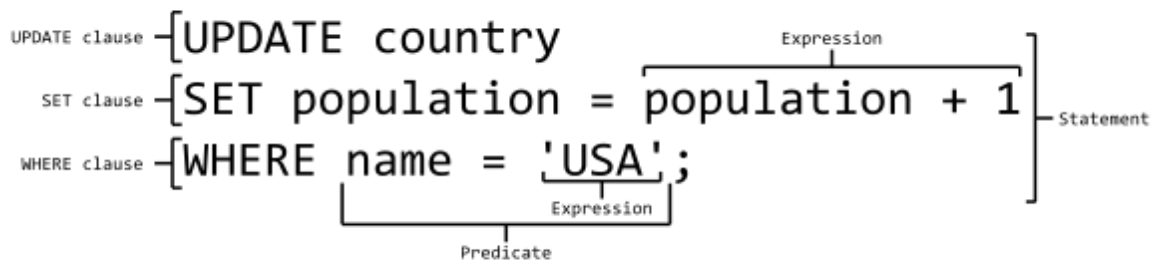


Figure 7.3: SQL Statement Anatomy: Ferdna, CC BY SA 3.0 MIGRATED

So, what does SQL look like? It has a lot of commands, though there are 4 basic commands which you'll want to get most comfortable with. The SELECT command, which is used to select or retrieve data from the database. The INSERT command, which as the name states, is used to insert new data. UPDATE, which is used to change values within the database. And, DELETE, which removes records from the database.

- ANSI Standard
- Proprietary Extensions
- Widely Supported

## 7.7 Other Query Languages

Among the numerous non-standard query languages, here are a few examples. Apache's Cassandra implements their Cassandra Query Language, which is SQL inspired. Oracle, Microsoft, and others offer **Procedural Languages** which embed some kind of scripting language within the database. Procedural Languages enable you to complex operations using things like loops and arrays within the database engine itself. By embedding the language into the engine, you gain performance from not having to transmit data across the network. As well, gaining capabilities like having some code executed upon a database insert or update.

In short, it's best to use a standardized language like SQL whenever possible. However, you shouldn't hinder yourself when the database engine offers an option that will improve performance or reduce data processing time.

- CQL: Cassandra
- PL/SQL: Oracle
- Transact-SQL: Microsoft SQL Server

## 7.8 Summary

We've gone over quite a bit of basics on databases. So, let's review the key points to remember.

CRUD (or Create, Read, Update, Delete) are the basic operations performed on a database. In SQL these operations are represented with INSERT, SELECT, UPDATE, and DELETE.

ACID (or Atomicity, Consistency, Isolation, Durability) compliance controls how data is stored. It ensures that all queries are atomic, e.g. all or nothing. That data stored is in full compliance with any validation rules. And, that database changes are really stored on disk.

Normalization is the process of reducing duplication in a database by relating records, instead of repeating the same information in each record.

Schema is the design and specifications for the database. It defines tables, content types, etc.

CAP Theorem applies mostly to NoSQL. It states that all database servers in a group can't show the exact same data, respond to every query, and tolerate a node failure without compromising in one of those areas.

We covered the three basic types of databases, flat, relational, and NoSQL. With flat files being generally frowned upon unless you have specific reasons for them. NoSQL typically only required when working with big data. And, relational being the most general purpose database you can use.

We've also covered ODBC, or Open Database Connectivity. Which is a standardized scheme for connecting applications, and languages with databases. In the Java world, the equivalent is JDBC or Java Database Connectivity. Regardless of language, these interfaces are your friend.

Finally, we covered some query languages. SQL being the most common query language, makes it a valuable tool to know how to use. If you plan to stick with a single database engine for a lot of projects, it may benefit you to become familiar with it's specific language. For instance, if working with Oracle a lot, you'll probably want to know PL/SQL.

---