

---

## Lesson 1.1: Introduction



## Lesson 1.1: Introduction

**Instructor:** Dr. GP Saggese, [gsaggese@umd.edu](mailto:gsaggese@umd.edu)



---

## 2 / 10: Invariants of a Class Lecture

### Invariants of a Class Lecture

- **Invariants of a class lecture**
  - Focus on intuition
  - Interactive Jupyter notebook tutorials
    - Tutorials done at home
    - Videos added over time
- **Class flow**
  - Alternate between slides, whiteboard, tutorials
- **Labs**
  - Review complete class project examples
  - Collaborate on class project



2 / 10

- **Invariants of a class lecture**
  - *Focus on intuition:* The goal here is to ensure that students grasp the fundamental concepts and underlying principles of the subject matter. Instead of just memorizing facts, students are encouraged to understand the ‘why’ and ‘how’ behind the concepts.
  - *Interactive Jupyter notebook tutorials:* These are hands-on coding exercises that allow students to apply what they’ve learned in a practical setting. Jupyter notebooks are particularly useful because they combine code, visualizations, and narrative text in one place.
    - \* *Tutorials done at home:* Students are expected to complete these tutorials outside of class time, allowing them to learn at their own pace and revisit challenging sections as needed.
    - \* *Videos added over time:* Supplementary videos are provided to enhance understanding. These might include walkthroughs of the tutorials or additional explanations of complex topics.
- **Class flow**
  - The class alternates between different teaching methods: slides for structured information, whiteboard for dynamic explanations and problem-solving, and tutorials for hands-on practice. This varied approach caters to different learning styles and keeps the class engaging.
- **Labs**
  - *Review complete class project examples:* Students examine full project examples to understand how individual concepts come together in a comprehensive application. This helps in visualizing the end goal of their learning.
  - *Collaborate on class project:* Students work together on a class project, fostering team-

---

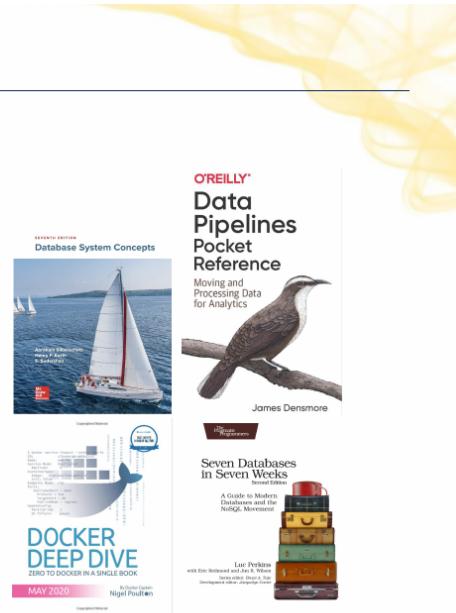
work and allowing them to learn from each other. Collaboration also mirrors real-world scenarios where teamwork is essential.

## 3 / 10: Books of the Class

### Books of the Class

- **Slides**

- Are extracted from books, technical articles, Internet
- Aim to be self-sufficient



3 / 10

- **Slides**

- **Are extracted from books, technical articles, Internet:** The slides used in this class are compiled from a variety of sources, including textbooks, scholarly articles, and online resources. This approach ensures that the information presented is comprehensive and up-to-date. By drawing from multiple sources, the slides provide a well-rounded perspective on the topics covered in the course.
- **Aim to be self-sufficient:** The slides are designed to be complete and understandable on their own. This means that even if you miss a lecture or need to review the material later, the slides should provide enough information to grasp the key concepts. They are structured to include definitions, explanations, and examples, making them a valuable resource for studying and revising the course material.

- **Images:** The images shown on the slide are likely covers or relevant pages from textbooks or articles used in the course. These visuals help to identify the primary sources of information and may also serve as a reference for further reading. By including these images, the slide emphasizes the importance of these texts in understanding the course content.

---

## 4 / 10: Learning Outcomes

### Learning Outcomes

- Model and reason about data
- Process and manipulate data
  - E.g., Python, Pandas
- Introduce a variety of data models
  - E.g., relational, NoSQL, graph DBs
  - Decide appropriate data model for different applications
- Use data management systems
  - E.g., PostgreSQL, MongoDB, HBase
  - Decide appropriate system for scenarios
- Build data processing pipelines
  - E.g., Docker, Airflow
- Build a big-data system end-to-end
  - Class project
  - Contribute to an open-source project



4 / 10

- Model and reason about data
  - This outcome focuses on understanding how to create models that represent data accurately. It involves analyzing data to draw meaningful conclusions and make informed decisions. This skill is foundational in data science and machine learning, as it helps in understanding the underlying patterns and relationships within data.
- Process and manipulate data
  - This involves using tools like *Python* and *Pandas* to clean, transform, and prepare data for analysis. These tools are essential for handling large datasets efficiently, allowing you to perform operations such as filtering, aggregating, and reshaping data.
- Introduce a variety of data models
  - Understanding different data models, such as relational databases, NoSQL, and graph databases, is crucial. Each model has its strengths and is suited for specific types of applications. For example, relational databases are great for structured data, while NoSQL is better for unstructured data.
- Use data management systems
  - This involves learning how to use systems like *PostgreSQL*, *MongoDB*, and *HBase* to store and retrieve data. Choosing the right system depends on the specific needs of your application, such as the type of data, scalability requirements, and query complexity.
- Build data processing pipelines
  - Tools like *Docker* and *Airflow* are used to automate and manage data workflows. Building pipelines helps in efficiently processing data from raw input to final output, ensuring that data is consistently and reliably transformed and analyzed.
- Build a big-data system end-to-end
  - This is a practical application of the skills learned, where you will either work on a class

---

project or contribute to an open-source project. It involves designing, implementing, and deploying a complete big-data system, providing hands-on experience in managing large-scale data processing tasks.

### Tools We Will Learn To Use

- **Programming languages**
  - Python
- **Development tools**
  - Bash/Linux OS
  - Git: data model, branching
  - GitHub: Pull Requests (PR), issues
  - Jupyter notebooks
  - Docker
- **Big data tools**
  - Extract-Transform-Load (ETL) pipelines
  - Relational DBs (PostgreSQL)
  - NoSQL DBs (HBase, MongoDB, Couchbase, Redis)
  - Graph DBs (Neo4j, GraphX, Giraph)
  - Computing framework (Hadoop, Spark, Dask)
  - Workflow manager (Airflow)
  - Cloud services (AWS)
- **Tutorials** for tools used in the class projects



5 / 10

- **Programming languages**
  - **Python:** We'll focus on Python because it's a versatile and widely-used language in data science and machine learning. It's known for its readability and has a rich ecosystem of libraries that make it ideal for data analysis and machine learning tasks.
- **Development tools**
  - **Bash/Linux OS:** Understanding the basics of Bash and Linux will help you navigate and manage files efficiently in a command-line environment, which is crucial for handling large datasets.
  - **Git:** We'll learn about Git's data model and branching to manage code versions effectively. This is essential for collaboration and maintaining a history of changes.
  - **GitHub:** We'll use GitHub for managing code repositories, focusing on Pull Requests (PR) and issues to facilitate collaboration and track project progress.
  - **Jupyter notebooks:** These are interactive tools that allow you to write and execute code in chunks, making it easier to document and share your data analysis and machine learning workflows.
  - **Docker:** Docker helps in creating consistent development environments by packaging applications and their dependencies into containers, ensuring that your code runs the same way everywhere.
- **Big data tools**
  - **Extract-Transform-Load (ETL) pipelines:** These are processes used to extract data from various sources, transform it into a suitable format, and load it into a database or data warehouse.
  - **Relational DBs (PostgreSQL):** We'll explore PostgreSQL, a powerful open-source relational database, to manage structured data using SQL.

- 
- **NoSQL DBs (HBase, MongoDB, Couchbase, Redis)**: These databases are designed to handle unstructured data and provide flexibility in data storage and retrieval.
  - **Graph DBs (Neo4j, GraphX, Giraph)**: Graph databases are used to model and query data with complex relationships, which is useful in scenarios like social networks.
  - **Computing framework (Hadoop, Spark, Dask)**: These frameworks allow for distributed processing of large datasets across clusters of computers, making data processing faster and more efficient.
  - **Workflow manager (Airflow)**: Airflow helps in scheduling and monitoring workflows, ensuring that data processing tasks are executed in the correct order.
  - **Cloud services (AWS)**: We'll explore Amazon Web Services (AWS) for scalable cloud computing resources, which are essential for handling big data projects.
- **Tutorials** for tools used in the class projects: We'll provide tutorials to help you get hands-on experience with these tools, ensuring you can apply what you learn to real-world projects.

### Todos

- Study slides and materials
- DATA605 - ELMS/Canvas site
  - Enable notifications
  - Contact info for me and TAs
    - Always keep the TAs in cc
- Check [DATA605 Schedule](#)
- Check [DATA605 GitHub repo](#)
- Get these [slides](#)
- Check [DATA605 FAQs](#)
- Setup computing environment
  - Install Linux/VMware
  - Install Docker on laptop
  - Instructions in class repo
- Bring laptop to class
- Lessons recorded
  - Still attend class, when possible



6 / 10

- **Study slides and materials**
  - It's important to review all the slides and materials provided for the course. This will help you understand the topics covered and prepare for discussions and assignments.
- **DATA605 - ELMS/Canvas site**
  - **Enable notifications:** Make sure you have notifications turned on so you don't miss any important updates or announcements.
  - **Contact info for me and TAs:** Keep the contact information for the instructor and teaching assistants handy. It's crucial to include TAs in your emails by keeping them in cc to ensure everyone is on the same page.
- **Check DATA605 Schedule**
  - The schedule is available on Google Docs. Regularly checking it will help you stay on track with deadlines and upcoming classes.
- **Check DATA605 GitHub repo**
  - The GitHub repository contains important resources and code examples. Familiarize yourself with its contents to aid your learning process.
- **Get these slides**
  - Access the slides from the GitHub link provided. They are essential for following along with lectures and understanding the course material.
- **Check DATA605 FAQs**
  - The FAQs document can answer common questions you might have about the course. It's a good first stop for any queries.
- **Setup computing environment**
  - **Install Linux/VMware:** Setting up a Linux environment or using VMware is necessary for running certain software and tools used in the course.

- 
- **Install Docker on laptop:** Docker is a tool that allows you to run applications in containers. Follow the instructions in the class repository to install it correctly.
  - **Bring laptop to class**
    - Having your laptop in class is important for participating in hands-on activities and following along with coding exercises.
  - **Lessons recorded**
    - While lessons are recorded for later viewing, attending class in person is beneficial for engaging with the material and asking questions in real-time.

### Grading

- **Quizzes**
  - 40% of grade
  - Multi-choice on previous 2 lessons
  - 20 questions in 20 minutes
  - 4-5 quizzes to encourage study during semester
- **Final Project**
  - 60% of grade
  - Comprehensive application of course concepts
  - Big data project in Python from a list of topics
  - Individual or group



7 / 10

- **Grading**

- **Quizzes**

- \* *40% of grade:* Quizzes make up a significant portion of your final grade, so it's important to take them seriously. They are designed to ensure you are keeping up with the course material.
    - \* *Multi-choice on previous 2 lessons:* Each quiz will cover content from the last two lessons, helping reinforce your understanding and retention of recent topics.
    - \* *20 questions in 20 minutes:* The quizzes are timed, with one minute per question, which means you need to be familiar with the material to answer quickly and accurately.
    - \* *4-5 quizzes to encourage study during semester:* Regular quizzes are spread throughout the semester to motivate consistent study habits and prevent last-minute cramming.

- **Final Project**

- \* *60% of grade:* The final project is the most substantial part of your grade, reflecting its importance in demonstrating your comprehensive understanding of the course.
    - \* *Comprehensive application of course concepts:* This project requires you to apply what you've learned throughout the course, showcasing your ability to integrate and utilize various concepts.
    - \* *Big data project in Python from a list of topics:* You'll choose a topic from a provided list, focusing on big data analysis using Python, which is a key skill in the field.
    - \* *Individual or group:* You have the flexibility to work alone or collaborate with peers, allowing you to choose the working style that best suits you.

### Class Projects

- **Project** is “*Build X with Y*”, where
  - X is a “use case”
  - Y is a “technology”
- **Activities**
  - Study and describe technology Y
  - Implement use case X using technology Y
  - Create Jupyter notebooks to demo your project
  - Commit code to GitHub, contribute to open-source repo
  - Write a blog entry
  - Present your project in a video
  - In-class labs + reviews
- **You choose from list** of X and Y, e.g.,
  - Data engineering
  - Emerging technologies (e.g., large language models)
  - ...
- **Each project:**
  - Individual or group ( $n < 4$ )
  - Varying difficulty levels



8 / 10

- **Class Projects:** The main goal of these projects is to combine a specific *use case* (represented by X) with a particular *technology* (represented by Y). This approach helps students apply theoretical knowledge to practical scenarios, enhancing their understanding and skills.
- **Activities:**
  - **Study and describe technology Y:** Begin by thoroughly understanding the technology you will use. This involves researching its features, capabilities, and limitations.
  - **Implement use case X using technology Y:** Apply the technology to solve a real-world problem or scenario, demonstrating how it can be effectively utilized.
  - **Create Jupyter notebooks to demo your project:** Use Jupyter notebooks to document your work, providing a clear and interactive way to showcase your project.
  - **Commit code to GitHub, contribute to open-source repo:** Share your code on GitHub, which not only helps in version control but also contributes to the open-source community.
  - **Write a blog entry:** Document your project journey, insights, and outcomes in a blog post to share your experience and learnings with a broader audience.
  - **Present your project in a video:** Create a video presentation to visually and verbally communicate your project, highlighting key aspects and results.
  - **In-class labs + reviews:** Participate in hands-on labs and peer reviews to refine your project and receive constructive feedback.
- **You choose from list** of X and Y: You have the flexibility to select from various use cases and technologies, such as data engineering or emerging technologies like large language models. This allows you to tailor the project to your interests and career goals.

---

- **Each project:**

- **Individual or group ( $n < 4$ ):** Projects can be done individually or in small groups, fostering collaboration and teamwork.
- **Varying difficulty levels:** Projects come with different levels of complexity, allowing you to challenge yourself according to your skill level and learning objectives.

---

## 9 / 10: Soft Skills to Succeed in the Workplace

### Soft Skills to Succeed in the Workplace

- **Goal:** model class project for workplace preparation
  - Work in a team
  - Design software architecture (OOP, Agile, Design Patterns)
  - Comment your code
  - Write external documentation (tutorials, manuals, how-tos)
  - Write understandable code (including for future-you)
  - Read others' code
  - Follow code conventions (PEP8, Google Code)
  - Communicate clearly (emails, Slack)
  - File a bug report
  - Reproduce a bug
  - Intuition of CS constants
  - Basic understanding of OS (virtual memory, processes)



9 / 10

- **Goal:** model class project for workplace preparation
  - *Work in a team:* Collaborating effectively with others is crucial in any workplace. It involves sharing ideas, dividing tasks, and supporting each other to achieve a common goal.
  - *Design software architecture (OOP, Agile, Design Patterns):* Understanding these concepts helps in creating scalable and maintainable software. Object-Oriented Programming (OOP) is about organizing code into objects, Agile is a flexible project management approach, and Design Patterns are standard solutions to common problems.
  - *Comment your code:* Writing comments in your code helps others (and your future self) understand the purpose and functionality of your code.
  - *Write external documentation (tutorials, manuals, how-tos):* Good documentation is essential for users to understand how to use your software effectively.
  - *Write understandable code (including for future-you):* Code should be clear and easy to read, which helps in maintaining and updating it over time.
  - *Read others' code:* Being able to understand and learn from others' code is a valuable skill that can improve your own coding practices.
  - *Follow code conventions (PEP8, Google Code):* Adhering to coding standards ensures consistency and readability across different projects and teams.
  - *Communicate clearly (emails, Slack):* Clear communication is key to avoiding misunderstandings and ensuring everyone is on the same page.
  - *File a bug report:* Knowing how to document and report bugs accurately helps in resolving issues efficiently.
  - *Reproduce a bug:* Being able to replicate a bug is the first step in diagnosing and fixing it.

- 
- *Intuition of CS constants*: Having a basic understanding of constants in computer science, like time complexity, helps in writing efficient code.
  - *Basic understanding of OS (virtual memory, processes)*: Knowing how operating systems manage resources like memory and processes can help in optimizing software performance.

---

## 10 / 10: Yours Truly

### Yours Truly

- **GP Saggese**
  - 2001-2006, PhD / Postdoc at the University of Illinois at Urbana-Champaign
  - [LinkedIn](#)
  - [gsaggese@umd.edu](mailto:gsaggese@umd.edu)
- **University of Maryland:**
  - 2023-, Lecturer for UMD DATA605: Big Data Systems
  - 2025-, Lecturer for UMD MSML610: Advanced Machine Learning
- **In the real-world**
  - Research scientist at NVIDIA, Synopsys, Teza, Engineers' Gate
  - 3x AI and fintech startup founder (ZeroSoft, June, Causify AI)
  - 20+ academic papers, 2 US patents



10 / 10

- **GP Saggese**
  - GP Saggese has a strong academic background, having completed a PhD and postdoctoral research at the University of Illinois at Urbana-Champaign between 2001 and 2006. This indicates a solid foundation in research and academia, particularly in fields related to machine learning and big data.
  - You can connect with GP Saggese on LinkedIn for professional networking or reach out via email at [gsaggese@umd.edu](mailto:gsaggese@umd.edu) for academic inquiries or collaborations.
- **University of Maryland**
  - Since 2023, GP Saggese has been a lecturer at the University of Maryland, teaching a course on Big Data Systems (UMD DATA605). This role highlights his expertise in handling and teaching about large-scale data systems.
  - Starting in 2025, he will also be teaching Advanced Machine Learning (UMD MSML610), which suggests a deep understanding of complex machine learning concepts and techniques.
- **In the real-world**
  - GP Saggese has practical experience as a research scientist at notable companies like NVIDIA, Synopsys, Teza, and Engineers' Gate. This experience bridges the gap between theoretical knowledge and practical application in industry settings.
  - He is an entrepreneur, having founded three AI and fintech startups: ZeroSoft, June, and Causify AI. This entrepreneurial experience demonstrates his ability to innovate and apply machine learning and big data concepts to real-world problems.
  - With over 20 academic papers and 2 US patents, GP Saggese has contributed significantly to the academic and technological community, showcasing his role as a thought leader in his field.

---

## Lesson 1.2: Introduction to Big Data



**Instructor:** Dr. GP Saggese, [gsaggese@umd.edu](mailto:gsaggese@umd.edu)



1 / 17

### Data Science

- **Promises of data science**
  - Give a competitive advantages
  - Make better strategic and tactical business decisions
  - Optimize business processes
- **Data science is not new**, it was called:
  - Operation research (~1970-80s)
  - Decision support, business intelligence (~1990s)
  - Predictive analytics (Early 2010s)
  - ...
- **What has changed**
  - Now learning and applying data science is *easy*
    - No need for hiring a consulting company
  - Tools are *open-source*
    - E.g., Python + pydata stack (numpy, scipy, Pandas, sklearn)
  - *Large data sets available*
  - *Cheap computing*
    - E.g., cloud computing (AWS, Google Cloud), GPUs



2 / 17

- **Promises of data science**
  - *Give a competitive advantage*: Data science can help businesses stand out by providing insights that others might not have. By analyzing data, companies can find new opportunities or improve their current operations, giving them an edge over competitors.
  - *Make better strategic and tactical business decisions*: With data science, businesses can make informed decisions based on data rather than intuition. This leads to more accurate and effective strategies and tactics.
  - *Optimize business processes*: Data science can identify inefficiencies and suggest improvements, leading to streamlined operations and cost savings.
- **Data science is not new**, it was called:
  - *Operation research (~1970-80s)*: This was an early form of data science focused on optimizing complex systems and processes.
  - *Decision support, business intelligence (~1990s)*: These terms referred to using data to support business decisions, similar to what data science does today.
  - *Predictive analytics (Early 2010s)*: This involved using data to predict future trends and behaviors, a key component of modern data science.
- **What has changed**
  - *Now learning and applying data science is easy*: In the past, businesses often needed to hire specialized consulting firms to implement data science solutions. Today, individuals can learn and apply these skills themselves.
  - *Tools are open-source*: Many powerful data science tools are freely available. For example, Python and its libraries (numpy, scipy, Pandas, sklearn) are widely used and accessible to everyone.
  - *Large data sets available*: The availability of vast amounts of data from various sources

---

has made it easier to perform comprehensive analyses.

- *Cheap computing:* The cost of computing power has decreased significantly. Cloud services like AWS and Google Cloud, along with GPUs, provide affordable and scalable computing resources.

### Motivation: Data Overload

- “*Data science is the number one catalyst for economic growth*”  
(McKinsey, 2013)
- **Explosion of data in every domain**
  - Sensing devices/networks monitor processes 24/7
    - E.g., temperature of your room, your vital signs, pollution in the air
  - Sophisticated smart-phones
    - 80%+ of the world population has a smart-phone
  - Internet and social networks make it easy to publish data
  - Internet of Things (IoT): everything is connected to the internet
    - E.g., power supply, toasters
  - Datafication turns all aspects of life into data
    - E.g., what you like/enjoy turned into a stream of your “likes”
- **Challenges**
  - How to handle the increasing amount data?
  - How to extract actionable insights and scientific knowledge from data?



3 / 17

- **Motivation: Data Overload**
  - The quote from McKinsey in 2013 highlights the importance of data science as a major driver for economic growth. This means that the ability to analyze and use data effectively can significantly boost economic activities and innovations.
- **Explosion of data in every domain**
  - *Sensing devices/networks monitor processes 24/7*: These devices continuously collect data about various aspects of our environment and lives. For example, sensors can track the temperature in your room, monitor your health through vital signs, or measure pollution levels in the air.
  - *Sophisticated smart-phones*: With over 80% of the global population owning a smartphone, these devices are a major source of data generation. They collect information through apps, GPS, and user interactions.
  - *Internet and social networks*: These platforms allow users to easily share and publish data, contributing to the vast amount of information available online.
  - *Internet of Things (IoT)*: This concept involves everyday objects being connected to the internet, enabling them to send and receive data. Examples include smart home devices like connected power supplies and toasters.
  - *Datafication*: This refers to the process of turning various aspects of life into data. For instance, your preferences and activities on social media are transformed into data streams, such as your “likes.”
- **Challenges**
  - The main challenge is managing the sheer volume of data being generated. This involves storing, processing, and analyzing data efficiently.
  - Another challenge is extracting meaningful insights and scientific knowledge from this

---

data. This means finding ways to turn raw data into useful information that can inform decisions and drive innovation.

### Scale of Data Size

- **Megabyte** =  $2^{20} \approx 10^6$  bytes
  - Typical English book
- **Gigabyte** =  $2^{30}$  bytes = 1,000 MB
  - 1/2 hour of video
  - Wikipedia (compressed, no media) is 22GB
- **Terabyte** = 1 million MB
  - Human genome: ~1 TB
  - 100,000 photos
  - \$50 for 1TB HDD, \$23/mo on AWS S3
- **Petabyte** = 1000 TB
  - 13 years of HD video
  - \$250k/year on AWS S3
- **Exabyte** = 1M TB
  - Global yearly Internet traffic in 2004
- **Zettabyte** = 1B TB =  $10^{21}$  bytes
  - Global yearly Internet traffic in 2016
  - Fill 20% of Manhattan, New York with data centers
- **Yottabytes** =  $10^{24}$  bytes
  - Yottabyte costs \$100T
  - Fill Delaware and Rhode Island with a million data centers
- **Brontobytes** =  $10^{27}$  bytes



4 / 17

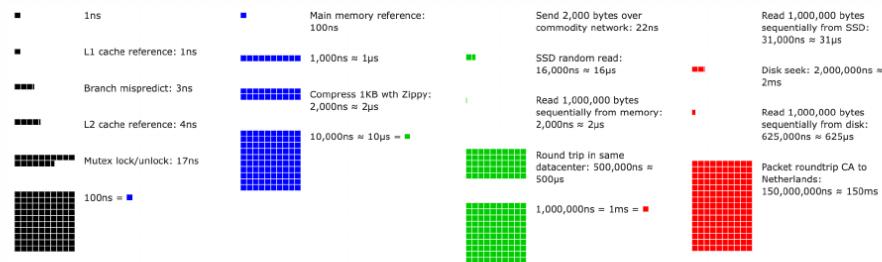
- **Megabyte:** A megabyte is a unit of digital information storage that equals approximately one million bytes. To put this into perspective, a typical English book, which contains text data, is about the size of a megabyte. This is a relatively small amount of data in today's digital world.
- **Gigabyte:** A gigabyte is 1,000 megabytes or  $2^{30}$  bytes. This size can store about half an hour of video content. For reference, the entire text of Wikipedia, when compressed and excluding media files, is around 22 gigabytes. This shows how much more data a gigabyte can hold compared to a megabyte.
- **Terabyte:** A terabyte is equivalent to one million megabytes. It can store a human genome, which is approximately 1 terabyte in size, or about 100,000 photos. In terms of cost, a 1TB hard drive is relatively affordable at around \$50, and storing this amount of data on AWS S3 cloud service costs about \$23 per month.
- **Petabyte:** A petabyte is 1,000 terabytes. This amount of storage can hold about 13 years of high-definition video. Storing a petabyte of data on AWS S3 would cost approximately \$250,000 per year, highlighting the significant expense associated with managing large-scale data.
- **Exabyte:** An exabyte is one million terabytes. In 2004, the global yearly Internet traffic reached this scale, illustrating the rapid growth of data usage over time.
- **Zettabyte:** A zettabyte equals one billion terabytes or  $10^{21}$  bytes. By 2016, global yearly Internet traffic had grown to this size. To visualize, storing a zettabyte of data would require enough data centers to fill 20% of Manhattan, New York.

- 
- **Yottabytes:** A yottabyte is  $10^{24}$  bytes. The cost of storing a yottabyte is estimated at \$100 trillion, and it would require enough data centers to fill the states of Delaware and Rhode Island.
  - **Brontobytes:** A brontobyte is  $10^{27}$  bytes, representing an even larger scale of data storage, though it is not commonly used in practical scenarios today.

## 5 / 17: Constants Everybody Should Know

### Constants Everybody Should Know

- CPU at 3GHz: 0.3 ns per instruction
- L1 cache reference/register: 1 ns
- L2 cache reference: 4 ns
- Main memory reference: 100 ns
- Read 1MB from memory: 20-100 us
- SSD random read: 16 us
- Send 1KB over network: 1 ms
- Disk seek: 2 ms
- Packet round-trip CA to Netherlands: 150 ms



5 / 17

- **CPU at 3GHz: 0.3 ns per instruction**

- This means that a CPU running at 3GHz can execute one instruction every 0.3 nanoseconds. This is incredibly fast and highlights the efficiency of modern processors. Understanding this helps us appreciate how quickly a CPU can process data and execute commands.

- **L1 cache reference/register: 1 ns**

- The L1 cache is the fastest type of memory available to the CPU, located directly on the processor chip. Accessing data from the L1 cache takes about 1 nanosecond, which is very quick and crucial for performance in executing instructions.

- **L2 cache reference: 4 ns**

- L2 cache is slightly slower than L1 but still much faster than accessing main memory. It takes about 4 nanoseconds to access data from the L2 cache, which serves as an intermediary storage to reduce the time needed to fetch data from the main memory.

- **Main memory reference: 100 ns**

- Accessing data from the main memory (RAM) is significantly slower than accessing cache memory. It takes about 100 nanoseconds, which is why having efficient cache usage is important for performance.

- **Read 1MB from memory: 20-100 us**

- Reading a megabyte of data from memory can take between 20 to 100 microseconds. This range depends on various factors like memory speed and system architecture. It's important to note the difference in scale compared to cache access times.

- **SSD random read: 16 us**

- Solid State Drives (SSDs) are much faster than traditional hard drives. A random read operation on an SSD takes about 16 microseconds, which is relatively quick and beneficial

---

for applications requiring fast data retrieval.

- **Send 1KB over network: 1 ms**

- Sending a kilobyte of data over a network typically takes about 1 millisecond. Network latency can vary based on distance and network conditions, but this gives a general idea of the time involved in data transmission.

- **Disk seek: 2 ms**

- Disk seek time refers to the time it takes for a hard drive's read/write head to move to the position on the disk where data is stored. This process takes about 2 milliseconds, which is relatively slow compared to SSDs.

- **Packet round-trip CA to Netherlands: 150 ms**

- This is the time it takes for a data packet to travel from California to the Netherlands and back. At 150 milliseconds, it highlights the latency involved in long-distance internet communications, which can impact real-time applications like video calls.

---

## 6 / 17: Big Data Applications: Marketing

### Big Data Applications: Marketing

- **Personalized marketing**
  - Target each consumer individually
  - E.g., Amazon personalizes suggestions using:
    - Shopping history
    - Search, click, browse activity
    - Other consumers and trends
    - Reviews (NLP and sentiment analysis)
- **Brands want to understand customer-product relationships**
  - Use sentiment analysis from:
    - Social media, online reviews, blogs, surveys
    - Positive, negative, neutral sentiment
  - E.g.,
    - In 2022, \$600B spent on digital marketing



6 / 17

- **Big Data Applications: Marketing**
- **Personalized marketing**
  - *Target each consumer individually:* This means using data to tailor marketing efforts to the specific preferences and behaviors of each customer. Instead of a one-size-fits-all approach, companies aim to make each customer feel like the marketing is just for them.
  - *E.g., Amazon personalizes suggestions using:* Amazon is a great example of personalized marketing. They use a variety of data points to recommend products to users.
    - \* **Shopping history:** This includes what you've bought before, which helps predict what you might want to buy next.
    - \* **Search, click, browse activity:** Every time you search for a product, click on a link, or browse through items, Amazon collects this data to understand your interests.
    - \* **Other consumers and trends:** By analyzing what similar customers are buying and current market trends, Amazon can suggest popular or trending items.
    - \* **Reviews (NLP and sentiment analysis):** Natural Language Processing (NLP) and sentiment analysis help Amazon understand the tone of reviews, whether they are positive, negative, or neutral, to better recommend products.
- **Brands want to understand customer-product relationships**
  - *Use sentiment analysis from:* Companies use sentiment analysis to gauge how customers feel about their products. This involves analyzing text data from various sources.
    - \* **Social media, online reviews, blogs, surveys:** These platforms provide a wealth of data where customers express their opinions and feelings about products.

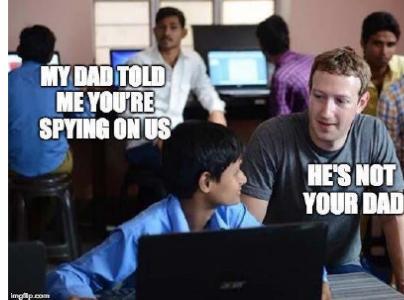
- 
- *Positive, negative, neutral sentiment:* By categorizing sentiments, brands can understand overall customer satisfaction and areas needing improvement.
  - E.g.,
    - *In 2022, \$600B spent on digital marketing:* This highlights the massive investment in digital marketing, emphasizing the importance of big data in understanding and reaching consumers effectively.

---

## 7 / 17: Big Data Applications: Advertisement

### Big Data Applications: Advertisement

- **Mobile advertisement**
  - Mobile phones are ubiquitous
  - 80% of world population has one
  - 6.5 billion smartphones
- **Integrate online and offline databases**, e.g.,
  - GPS location
  - Search history
  - Credit card transactions
- E.g.,
  - You've bought a new house
  - You google questions about house renovations
  - You watch shows about renovations
  - Your phone tracks where you are
  - Google sends you coupons for the closest Home Depot
  - "*I feel like Google is following me*"



7 / 17

- **Mobile advertisement**
  - Mobile phones are everywhere, and they play a huge role in how companies reach potential customers. With 80% of the world's population owning a mobile phone, and 6.5 billion of those being smartphones, advertisers have a massive audience to target. This means that almost everyone is reachable through their mobile device, making it a powerful tool for advertising.
- **Integrate online and offline databases**
  - Advertisers use a combination of online and offline data to create targeted ads. For example, they might use your GPS location to know where you are, your search history to understand your interests, and your credit card transactions to see what you buy. By combining these data sources, advertisers can create personalized ads that are more likely to catch your attention.
- **Example scenario**
  - Imagine you've just bought a new house. You start searching online for renovation tips and watching home improvement shows. Meanwhile, your phone tracks your location. Based on this data, Google might send you coupons for the nearest Home Depot. This targeted advertising can feel very personal, leading to the feeling that "Google is following me." This example illustrates how big data is used to create highly personalized advertising experiences.

---

## 8 / 17: Big Data Applications: Medicine

### Big Data Applications: Medicine

- **Personalized medicine**

- Patients receive treatment tailored to them for efficacy
- Genetics
- Daily activities
- Environment
- Habits

- **Biomedical data**

- **Genome sequencing**

- **Health tech**

- Personal health trackers (e.g., smart rings, phones)



8 / 17

- **Big Data Applications: Medicine**

- **Personalized medicine**

- *Personalized medicine* is a medical approach where treatments are customized for individual patients. This means that instead of a one-size-fits-all treatment, doctors use data to tailor healthcare specifically for each person.
- **Genetics:** By analyzing a patient's genetic information, doctors can predict how they might respond to certain medications or what diseases they might be at risk for.
- **Daily activities:** Information about a patient's lifestyle, such as exercise routines and diet, can help in crafting a more effective treatment plan.
- **Environment:** Environmental factors, like where a person lives and works, can influence their health and are considered in personalized treatments.
- **Habits:** Understanding a patient's habits, such as smoking or alcohol consumption, is crucial for creating a personalized healthcare plan.

- **Biomedical data**

- This refers to the vast amount of data generated in the medical field, including patient records, lab results, and imaging data. Analyzing this data helps in improving patient care and advancing medical research.

- **Genome sequencing**

- Genome sequencing involves decoding a person's DNA to understand their genetic makeup. This information is vital for identifying genetic disorders and tailoring treatments to an individual's genetic profile.

---

- **Health tech**

- Personal health trackers, like smart rings and phones, collect data on various health metrics such as heart rate, sleep patterns, and physical activity. This data can be used to monitor health in real-time and make informed decisions about lifestyle changes or medical interventions.

## Big Data Applications: Smart Cities

- **Smart cities**
  - Interconnected mesh of sensors
  - E.g., traffic sensors, camera networks, satellites
- **Goals**
  - Monitor air pollution
  - Minimize traffic congestion
  - Optimal urban services
  - Maximize energy savings



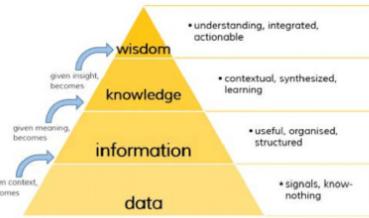
9 / 17

- **Smart cities**
  - Smart cities are urban areas that use technology and data to improve the quality of life for their residents. They rely on an *interconnected mesh of sensors* to collect and analyze data. This network of sensors can include devices like traffic sensors, camera networks, and even satellites. These tools work together to gather real-time information about various aspects of city life.
- **Goals**
  - One of the primary goals of smart cities is to *monitor air pollution*. By using sensors to track air quality, cities can identify pollution sources and take action to improve the environment.
  - Another goal is to *minimize traffic congestion*. Traffic sensors and camera networks help manage traffic flow, reduce bottlenecks, and improve commute times.
  - Smart cities aim to provide *optimal urban services*. This means using data to enhance public services like waste management, water supply, and emergency response.
  - Finally, smart cities strive to *maximize energy savings*. By analyzing energy usage patterns, cities can implement strategies to reduce consumption and promote sustainability.

## 10 / 17: Goal of Data Science

### Goal of Data Science

- **Goal:** from data to wisdom
  - Data (raw bytes)
  - Information (organized, structured)
  - Knowledge (learning)
  - Wisdom (understanding)
- **Insights enable decisions and actions**
- Combine streams of big data to **generate new data**
  - New data can be “big data” itself



10 / 17

- **Goal:** from data to wisdom
  - The journey from data to wisdom is a process of transforming raw data into something meaningful. **Data** refers to the raw bytes or unprocessed facts and figures. It's the starting point of any data science project.
  - **Information** is what you get when you organize and structure data. This step involves cleaning and formatting data so it can be analyzed.
  - **Knowledge** is derived from analyzing information. It involves learning patterns, trends, and insights from the data.
  - **Wisdom** is the ultimate goal, where you not only understand the data but can also apply it to make informed decisions and solve real-world problems.
- **Insights enable decisions and actions**
  - The insights gained from data analysis are crucial because they inform decisions and actions. Without insights, data remains just numbers and figures without practical application.
- Combine streams of big data to **generate new data**
  - In data science, combining different data streams can lead to the creation of new data sets. This new data can be considered “big data” itself, as it often involves large volumes, high velocity, and a variety of data types. This process is essential for uncovering deeper insights and creating more comprehensive models.

## 11 / 17: The Six V'S of Big Data

### The Six V'S of Big Data

- What makes “Big Data” big?
- **Volume**
  - Vast amount of data is generated
- **Variety**
  - Different forms
- **Velocity**
  - Speed of data generation
- **Veracity**
  - Biases, noise, abnormality in data
  - Uncertainty, trustworthiness
- **Valence**
  - Connectedness of data in the form of graphs
- **Value**
  - Data must be valuable
  - Benefit an organization



11 / 17

- **Volume**
  - *Volume* refers to the sheer amount of data that is being generated every second. In today’s digital world, data is being produced at an unprecedented rate from various sources like social media, sensors, and transactions. This massive volume of data is what makes it “big” and requires special tools and technologies to store, process, and analyze it effectively.
- **Variety**
  - *Variety* highlights the different forms and types of data that are available. Data can be structured, like databases, or unstructured, like videos, images, and text. This diversity in data types presents challenges in terms of storage, processing, and analysis, as different types of data require different handling techniques.
- **Velocity**
  - *Velocity* refers to the speed at which data is generated and needs to be processed. With the rise of real-time data sources such as social media feeds and IoT devices, the ability to quickly process and analyze data as it arrives is crucial for making timely decisions.
- **Veracity**
  - *Veracity* deals with the quality and trustworthiness of the data. Data can often be noisy, biased, or incomplete, which can affect the accuracy of the insights derived from it. Ensuring data veracity is essential for making reliable decisions based on data analysis.
- **Valence**
  - *Valence* is about the connectedness of data, often represented in the form of graphs. It emphasizes the relationships and interactions between different data points, which can provide deeper insights into patterns and trends.
- **Value**

- 
- *Value* is perhaps the most important aspect, as it focuses on the usefulness of the data. For data to be considered “big,” it must provide value to an organization, helping it to achieve its goals, improve operations, or gain a competitive advantage. Without value, the other characteristics of big data are meaningless.

---

## 12 / 17: The Six V's of Big Data

### The Six V's of Big Data

- **Volume**

- Exponentially increasing data
- 2.5 exabytes (1m TB) generated daily
  - 90% of data generated in last 2 years
  - Data doubles every 1.2 years
- Twitter/X: 500M tweets/day (2022)
- Google: 8.5B queries/day (2022)
- Meta: 4PB data/day (2022)
- Walmart: 2.5PB unstructured data/hour (2022)

- **Variety**

- Different data forms
  - Structured (e.g., spreadsheets, relational data)
  - Semi-structured (e.g., text, sales receipts, class notes)
  - Unstructured (e.g., photos, videos)
- Different formats (e.g., binary, CSV, XML, JSON)



12 / 17

- **Volume**

- The term *Volume* refers to the sheer amount of data being generated and collected. This is a defining characteristic of big data, as the quantities are so large that traditional data processing software cannot handle them efficiently.
- An *exabyte* is a unit of data equal to one million terabytes, and currently, about 2.5 exabytes of data are created every day. This highlights the rapid growth of data generation.
- It's noteworthy that 90% of the world's data has been created in just the last two years, illustrating the exponential growth rate. This means that the amount of data doubles approximately every 1.2 years.
- Examples of this massive data generation include platforms like Twitter/X, which sees 500 million tweets daily, and Google, which processes 8.5 billion search queries each day.
- Companies like Meta and Walmart handle enormous amounts of data daily, with Meta processing 4 petabytes and Walmart dealing with 2.5 petabytes of unstructured data every hour.

- **Variety**

- *Variety* refers to the different types of data that are being generated. Unlike traditional data, which was mostly structured, big data comes in various forms.
- Structured data is organized and easily searchable, like data in spreadsheets or databases.
- Semi-structured data includes elements of both structured and unstructured data, such as text documents, sales receipts, or class notes, which have some organizational properties but are not as rigidly formatted.
- Unstructured data is more complex and includes formats like photos and videos, which do not fit neatly into tables or databases.

- 
- Data can also come in different formats, such as binary, CSV, XML, and JSON, each requiring different methods for processing and analysis. This diversity in data types and formats presents both challenges and opportunities for data analysis.

---

## 13 / 17: The Six V's of Big Data

### The Six V's of Big Data

- **Velocity**

- Speed of data generation
  - E.g., sensors generate data streams
- Process data off-line or in real-time
- Real-time analytics: consume data as fast as generated

- **Veracity**

- Relates to data quality
- How to remove noise and bad data?
- How to fill in missing values?
- What is an outlier?
- How do you decide what data to trust?



13 / 17

- **Velocity**

- *Speed of data generation* refers to how quickly data is being produced. In today's world, data is generated at an unprecedented rate, especially from sources like sensors, social media, and online transactions.
  - \* For example, sensors in smart devices or industrial machines continuously produce data streams that need to be managed and analyzed.
- *Process data off-line or in real-time* highlights the decision organizations must make regarding how they handle incoming data. Offline processing involves storing data for later analysis, while real-time processing means analyzing data as it arrives.
- *Real-time analytics* is crucial for applications that require immediate insights, such as fraud detection or monitoring system health. It involves consuming and analyzing data as quickly as it is generated to make timely decisions.

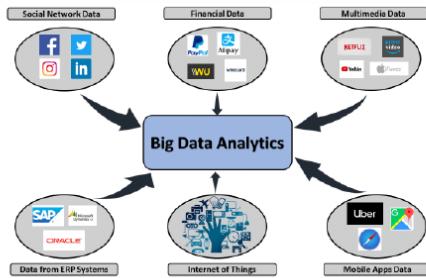
- **Veracity**

- *Relates to data quality* emphasizes the importance of ensuring that the data being used is accurate and reliable. Poor data quality can lead to incorrect insights and decisions.
- *How to remove noise and bad data?* involves identifying and eliminating irrelevant or erroneous data that can skew analysis results.
- *How to fill in missing values?* is about determining the best method to handle incomplete data, which might involve estimation or using default values.
- *What is an outlier?* refers to data points that are significantly different from others in the dataset. Identifying outliers is important as they can indicate errors or unique insights.
- *How do you decide what data to trust?* involves establishing criteria or using tools to assess the reliability of data sources, ensuring that decisions are based on credible information.

## 14 / 17: Sources of Big Data

### Sources of Big Data

- Distinguish Big Data by source
  - **Machines**
  - **People**
  - **Organizations**



- **Distinguish Big Data by source**
  - **Machines**
    - \* Machines generate a vast amount of data through sensors, logs, and automated processes. This includes data from IoT devices, industrial machines, and computer systems. Machine-generated data is typically structured and can be collected at a high velocity, making it a significant component of big data.
  - **People**
    - \* People contribute to big data through their interactions with digital platforms. This includes social media posts, online transactions, and user-generated content. This type of data is often unstructured and diverse, encompassing text, images, and videos. It provides insights into human behavior and preferences.
  - **Organizations**
    - \* Organizations produce data through their operations, such as sales records, customer databases, and financial transactions. This data is usually structured and is crucial for business analytics. Organizations also collect data from external sources to enhance their decision-making processes.
- *Context:* Understanding the sources of big data is essential for effectively managing and analyzing it. Each source has unique characteristics and challenges, influencing how data is processed and utilized.

---

## 15 / 17: Sources of Big Data: Machines

### Sources of Big Data: Machines

- **Machines generate data**
  - Real-time sensors (e.g., sensors on Boeing 787)
  - Cars
  - Website tracking
  - Personal health trackers
  - Scientific experiments
- **Pros**
  - Highly structured
- **Cons**
  - Difficult to move, computed in-place or centralized
  - Streaming, not batch



15 / 17

- Sources of Big Data: Machines

- Machines generate data

- \* Machines are a significant source of big data. They produce data through various means, such as *real-time sensors*. For example, a Boeing 787 airplane is equipped with numerous sensors that continuously collect data during flights. This data can include information about engine performance, fuel efficiency, and environmental conditions.
    - \* Cars are another example, as modern vehicles are equipped with sensors that monitor everything from tire pressure to engine diagnostics.
    - \* Websites track user interactions, generating data about user behavior, preferences, and engagement.
    - \* Personal health trackers, like fitness bands and smartwatches, collect data on physical activity, heart rate, and sleep patterns.
    - \* Scientific experiments, such as those conducted in laboratories or large-scale projects like the Large Hadron Collider, produce vast amounts of data for analysis.

- Pros

- \* The data generated by machines is *highly structured*, meaning it is organized in a specific format, making it easier to analyze and process compared to unstructured data like text or images.

- Cons

- \* One challenge with machine-generated data is that it can be *difficult to move*. Due to its large volume, it often needs to be processed in-place or within a centralized system to avoid the complexities and costs associated with data transfer.
    - \* This data is typically *streaming*, meaning it is continuously generated and needs

---

to be processed in real-time, unlike batch processing where data is collected and processed at intervals. This requires specialized tools and infrastructure to handle the constant flow of information.

## 16 / 17: Sources of Big Data: People

### Sources of Big Data: People

- **People and their activities generate data**

- Social media (Instagram, Twitter, LinkedIn)
- Video sharing (YouTube, TikTok)
- Blogging, website comments
- Internet searches
- Text messages (SMS, WhatsApp, Signal, Telegram)
- Personal documents (Google Docs, emails)



- **Pros**

- Enable personalization
- Valuable for business intelligence

- **Cons**

- Semi-structured or unstructured data
  - Text, images, movies
- Requires investment to extract value
  - Acquire → Store → Clean → Retrieve  
→ Process → Insights

UNIVERSITY OF MARYLAND  
SCIENCE SURVEILLANCE CAPITALISM  
ACADEMY

16 / 17

- **People and their activities generate data**

- Every day, people create a vast amount of data through their interactions on various platforms. **Social media** platforms like Instagram, Twitter, and LinkedIn are prime examples where users share thoughts, images, and videos, contributing to a massive pool of data. **Video sharing** sites such as YouTube and TikTok further add to this by hosting countless hours of video content. **Blogging and website comments** are other avenues where people express opinions and share information. **Internet searches** provide insights into what people are curious about or need information on. **Text messages** sent via SMS or apps like WhatsApp, Signal, and Telegram are another rich source of data. Lastly, **personal documents** such as Google Docs and emails contain a wealth of information about personal and professional communications.

- **Pros**

- The data generated by people allows for **personalization**, meaning businesses can tailor experiences and products to individual preferences. This data is also **valuable for business intelligence**, helping companies understand market trends, customer behavior, and more.

- **Cons**

- A significant challenge is that much of this data is **semi-structured or unstructured**, such as text, images, and videos, which are not easily organized into traditional databases. Extracting value from this data requires a substantial **investment** in processes to acquire, store, clean, retrieve, and process it to gain meaningful insights. Additionally, there is a concern about **surveillance capitalism**, where companies might exploit personal data for profit, raising privacy issues.

### Sources of Big Data: Organizations

- **Organizations generate data**
  - Commercial transactions
  - Credit cards
  - E-commerce
  - Banking
  - Medical records
  - Website clicks
- **Pros**
  - Highly structured
- **Cons**
  - Store every event to predict future
    - Miss opportunities
  - Stored in “data silos” with different models
    - Each department has own system
    - Additional complexity
    - Data outdated/not visible
    - Cloud computing helps (e.g., data lakes, data warehouses)



17 / 17

- **Sources of Big Data: Organizations**
  - Organizations are major producers of data, and they do this through various activities. **Commercial transactions** involve the exchange of goods and services, generating a lot of data. **Credit cards** and **e-commerce** platforms track purchases and user behavior. **Banking** institutions record financial transactions, while **medical records** contain patient information and treatment histories. **Website clicks** provide insights into user interactions and preferences.
- **Pros**
  - The data generated by organizations is often **highly structured**, meaning it is organized in a way that makes it easier to analyze and use. This structure helps in efficiently processing and extracting valuable insights.
- **Cons**
  - One downside is that organizations tend to **store every event** with the hope of predicting future trends, but this can lead to missed opportunities if not analyzed properly. Data is often kept in “**data silos**,” where each department has its own system and model. This creates **additional complexity** because data can become outdated or not easily accessible across the organization. However, **cloud computing** solutions like *data lakes* and *data warehouses* are helping to mitigate these issues by providing centralized storage and easier access to data.

---

## Lesson 1.3: Is Data Science Just Hype?



UMD DATA605: Big Data Systems

## Lesson 1.3: Is Data Science Just Hype?

**Instructor:** Dr. GP Saggese, [gsaggese@umd.edu](mailto:gsaggese@umd.edu)

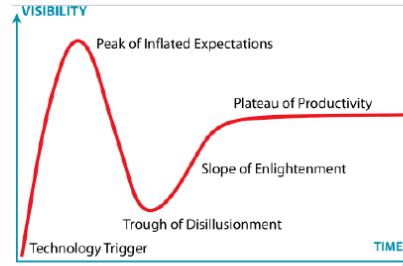


1 / 14

## 2 / 14: Is Data Science Just Hype?

### Is Data Science Just Hype?

- **Big data (or data science) is everywhere**
  - “Any process where interesting information is inferred from data”
- Data scientist called the “sexiest job” of the 21st century
  - The term has become very muddled at this point
- **Is it all hype?**



2 / 14

- **Big data (or data science) is everywhere**
  - The phrase “Any process where interesting information is inferred from data” highlights the broad scope of data science. It means that data science involves analyzing data to extract meaningful insights, which can be applied across various fields like healthcare, finance, marketing, and more. This universality is why data science is so prevalent today.
- **Data scientist called the “sexiest job” of the 21st century**
  - This statement reflects the high demand and allure of the data scientist role. The job is considered “sexy” because it combines technical skills, creativity, and problem-solving to drive business decisions and innovations. However, the term “data scientist” has become muddled, meaning that as the field has grown, the role’s definition has become less clear, encompassing a wide range of skills and responsibilities.
- **Is it all hype?**
  - This question prompts us to consider whether the excitement around data science is justified or if it’s exaggerated. While data science has transformative potential, it’s important to recognize that not all claims about its impact may be realistic. Understanding the true capabilities and limitations of data science is crucial to avoid overhyping its potential.

---

## 3 / 14: Is Data Science Just Hype?

### Is Data Science Just Hype?

- **No**
  - Extract insights and knowledge from data
  - Big data techniques revolutionize many domains
    - E.g., education, food supply, disease epidemics
- **But**
  - Similar to what statisticians have done for years
- **What is different?**
  - More data is digitally available
  - Easy-to-use programming frameworks (e.g., Hadoop) simplify analysis
  - Cloud computing (e.g., AWS) reduces costs
  - Large-scale data + simple algorithms often outperform small data + complex algorithms



3 / 14

- **Is Data Science Just Hype?**

- **No**

- \* Data science is not just a buzzword; it plays a crucial role in extracting valuable insights and knowledge from vast amounts of data. This process helps organizations and researchers make informed decisions and predictions.
    - \* The advent of big data techniques has transformed various fields by providing new ways to analyze and interpret data. For example, in education, data science can help tailor learning experiences to individual students. In the food supply chain, it can optimize logistics and reduce waste. In managing disease epidemics, it can predict outbreaks and improve response strategies.

- **But**

- \* While data science is impactful, it's important to recognize that it builds on the foundation laid by statisticians. Statisticians have been analyzing data to draw conclusions for many years, and data science extends these principles with new tools and technologies.

- **What is different?**

- \* The key difference today is the sheer volume of data that is now digitally available, which was not the case in the past. This abundance of data allows for more comprehensive analysis.
    - \* Easy-to-use programming frameworks, such as *Hadoop*, have made it simpler for people to process and analyze large datasets without needing extensive programming skills.
    - \* Cloud computing services, like *AWS*, have significantly reduced the cost and complexity of storing and processing large datasets, making data science more accessible

---

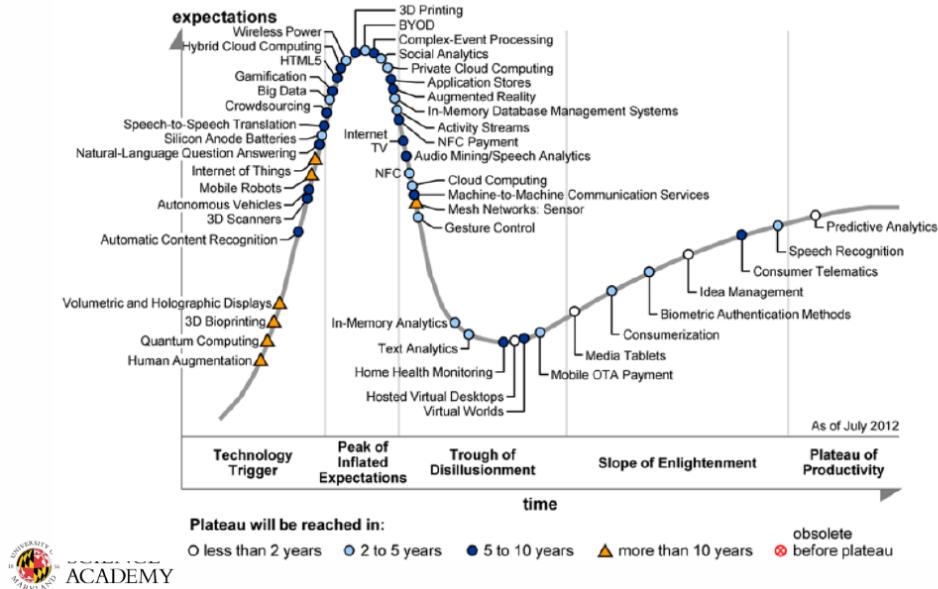
to a wider audience.

- \* Interestingly, having access to large-scale data allows for the use of simpler algorithms, which can often yield better results than using complex algorithms on smaller datasets. This is because more data can provide a clearer picture and reduce the noise in the analysis.

## 4 / 14: What Was Cool in 2012?

### What Was Cool in 2012?

- Big data, Predictive analytics



- **Big Data**

- In 2012, the term *big data* was gaining significant traction. It refers to the vast volumes of data generated every second from various sources like social media, sensors, and transactions. The challenge was not just storing this data but also processing and analyzing it to extract meaningful insights. This was a time when companies started realizing the potential of data as a valuable asset, leading to investments in technologies and infrastructure to handle big data.

- **Predictive Analytics**

- Predictive analytics involves using historical data to predict future outcomes. In 2012, this was becoming a hot topic as businesses sought to leverage data to forecast trends, customer behavior, and potential risks. The rise of machine learning algorithms played a crucial role in enhancing predictive analytics, allowing for more accurate and actionable predictions. This was particularly appealing to industries like finance, healthcare, and retail, where anticipating future trends could lead to significant competitive advantages.

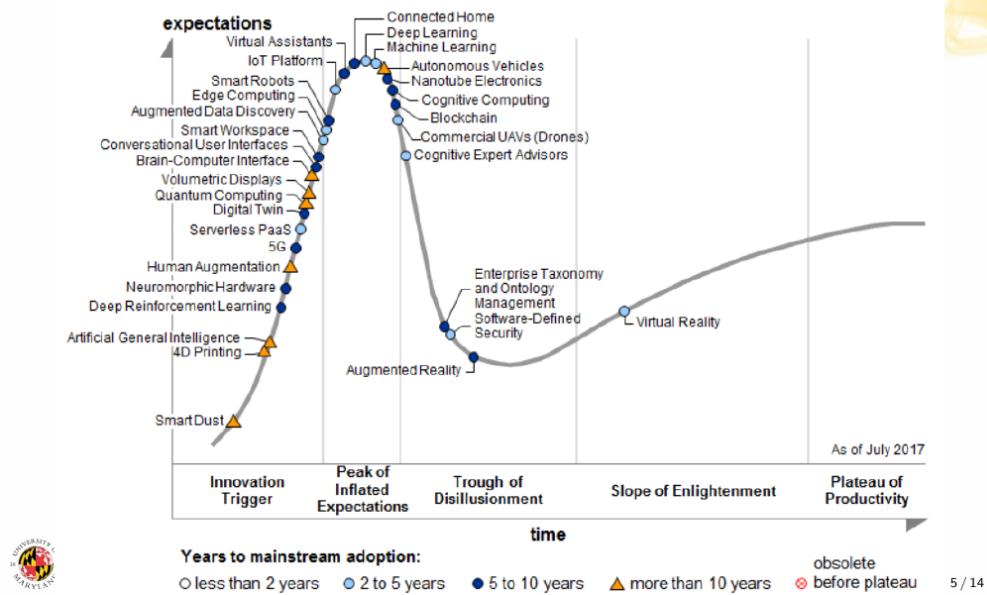
- **Context**

- The excitement around big data and predictive analytics in 2012 was driven by advancements in technology, such as improved data storage solutions and more powerful computing capabilities. This period marked the beginning of a data-driven approach in decision-making across various sectors, setting the stage for the data-centric world we live in today.

## 5 / 14: What Was Cool in 2017?

### What Was Cool in 2017?

- Deep learning, Machine learning



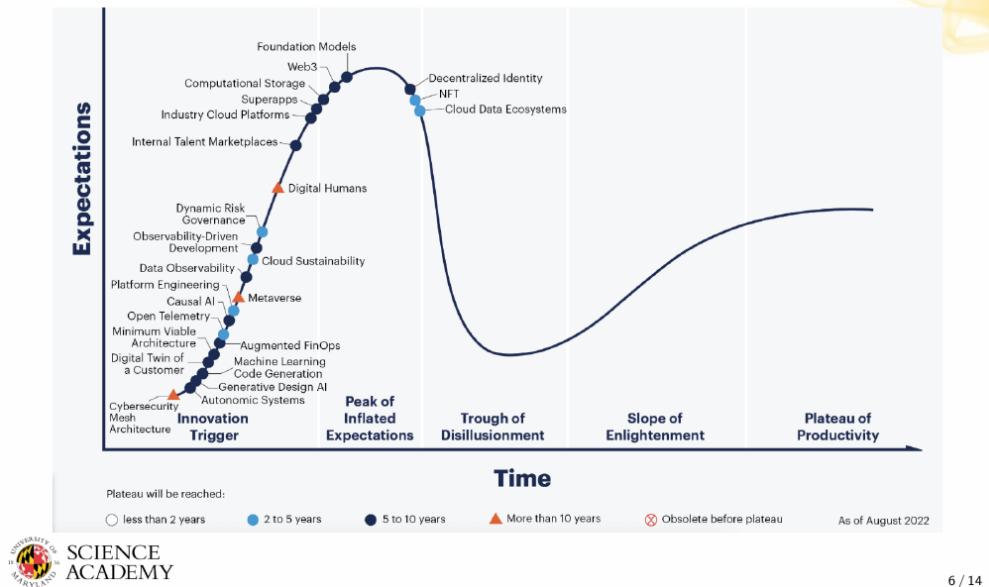
- **Deep Learning:** In 2017, deep learning was a major trend in the field of artificial intelligence. It involves using neural networks with many layers (hence “deep”) to model complex patterns in data. This approach was particularly successful in areas like image and speech recognition, where it significantly improved accuracy and performance. The excitement around deep learning was due to its ability to automatically learn features from raw data, reducing the need for manual feature engineering.
- **Machine Learning:** Machine learning, a broader field that includes deep learning, was also a hot topic in 2017. It refers to the use of algorithms and statistical models to enable computers to improve their performance on a task through experience. Machine learning was being applied across various industries, from healthcare to finance, to automate processes and gain insights from large datasets. The rise of big data and increased computational power contributed to the growing interest and advancements in machine learning during this time.

In summary, 2017 was a pivotal year for both deep learning and machine learning, as these technologies began to demonstrate their potential to transform industries and solve complex problems.

## 6 / 14: What Was Cool in 2022?

### What Was Cool in 2022?

- Causal AI



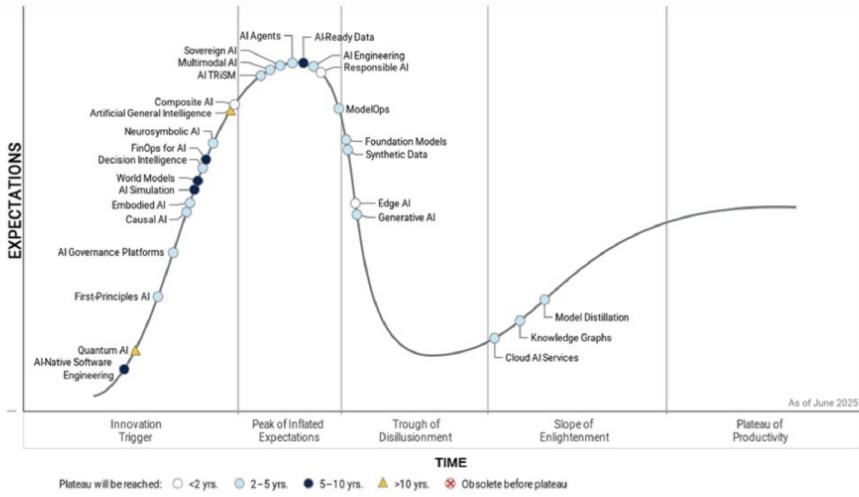
- Causal AI

- Causal AI was a significant trend in 2022, gaining attention for its ability to go beyond traditional correlation-based machine learning models. Unlike standard AI models that often identify patterns and correlations, Causal AI focuses on understanding the *cause-and-effect* relationships within data. This is crucial because knowing what causes what can lead to more accurate predictions and better decision-making.
- The importance of Causal AI lies in its potential to provide insights that are not just predictive but also prescriptive. For instance, in healthcare, understanding the causal factors of a disease can lead to more effective treatments. In business, it can help identify the true drivers of customer behavior, leading to more targeted marketing strategies.
- Causal AI uses techniques like *causal inference* and *structural causal models* to identify these relationships. This approach can help in scenarios where interventions are possible, allowing organizations to simulate the effects of potential actions before implementing them.
- The growing interest in Causal AI reflects a broader trend towards more interpretable and actionable AI systems, which are essential for building trust and ensuring ethical use of AI technologies.

## 7 / 14: What Was Cool in 2025?

### What Was Cool in 2025?

- Causal AI, Decision intelligence



7 / 14

- Causal AI:**

- Causal AI refers to a branch of artificial intelligence that focuses on understanding cause-and-effect relationships rather than just correlations. This is important because while traditional AI models can identify patterns and correlations in data, they often cannot determine if one thing causes another.
- By understanding causality, AI systems can make better predictions and decisions. For example, in healthcare, causal AI can help determine whether a treatment actually improves patient outcomes or if the observed effects are due to other factors.
- In 2025, causal AI was considered “cool” because it represented a significant advancement in AI’s ability to provide insights that are more actionable and reliable.

- Decision Intelligence:**

- Decision intelligence is an emerging field that combines data science, social science, and managerial science to improve decision-making processes. It involves using AI and machine learning to analyze data and provide recommendations that help organizations make better decisions.
- This approach is particularly valuable in complex environments where decisions need to be made quickly and with incomplete information. By leveraging decision intelligence, businesses can optimize operations, reduce risks, and improve outcomes.
- In 2025, decision intelligence gained popularity as it empowered organizations to harness the power of AI not just for automation, but for strategic decision-making, making it a “cool” trend in the tech world.

---

## 8 / 14: Key Shifts Before/After Big-Data

### Key Shifts Before/After Big-Data

- **Datasets: small, curated, clean → large, uncurated, messy**
  - Before:
    - Statistics based on small, carefully collected random samples
    - Costly and careful planning for experiments
    - Hard to do fine-grained analysis
  - Today:
    - Easily collect huge data volumes
    - Feed into algorithms
    - Strong signal overcomes noise
- **Causation → Correlation**
  - Goal: determine cause and effect
  - Causation hard to determine → focus on correlation
    - Correlation is sometimes sufficient
    - E.g., diapers and beer bought together
- **"Data-fication"**
  - = converting abstract concepts into data
  - E.g., "sitting posture" data-fied by sensors in your seat
  - Preferences data-fied into likes



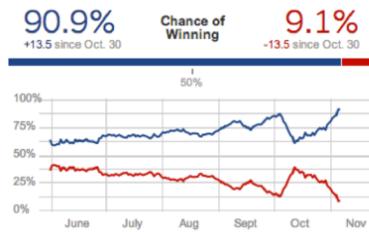
8 / 14

- **Datasets: small, curated, clean → large, uncurated, messy**
  - *Before Big Data:*
    - \* In the past, data analysis relied on small datasets that were carefully selected and cleaned. This was because collecting data was expensive and required meticulous planning.
    - \* Researchers often used random samples to make statistical inferences, but this limited the ability to perform detailed analysis.
  - *Today with Big Data:*
    - \* We can now gather vast amounts of data quickly and at a lower cost. This data is often messy and uncurated, but the sheer volume allows us to extract meaningful insights.
    - \* Algorithms can process these large datasets, and the significant amount of data helps to highlight important patterns, even if there is some noise.
- **Causation → Correlation**
  - Traditionally, the aim was to understand cause and effect relationships. However, determining causation is complex and often not feasible with large datasets.
  - With big data, the focus has shifted to identifying correlations, which can be very useful. For example, noticing that diapers and beer are often purchased together can inform marketing strategies, even if the exact cause isn't clear.
- **"Data-fication"**
  - This term refers to the process of turning abstract concepts into quantifiable data.
  - For instance, sensors can capture data about your sitting posture, transforming it into something that can be analyzed. Similarly, online preferences are converted into data points like "likes," which can be used to understand user behavior and preferences.

## 9 / 14: Examples: Election Prediction

### Examples: Election Prediction

- Nate Silver and the 2012 Elections
  - Predicted 49/50 states in 2008 US elections
  - Predicted 50/50 states in 2012 US elections
- **Reasons for accuracy**
  - Multiple data sources
  - Historical accuracy incorporation
  - Statistical models
  - Understanding correlations
  - Monte-Carlo simulations for electoral probabilities
  - Focus on probabilities
  - Effective communication



9 / 14

- **Nate Silver and the 2012 Elections**

- Nate Silver is a well-known statistician and writer who gained fame for his accurate predictions in the US elections. In 2008, he correctly predicted the outcomes in 49 out of 50 states, and in 2012, he improved his accuracy by predicting all 50 states correctly. This achievement highlighted the power of data-driven approaches in making accurate predictions.

- **Reasons for accuracy**

- **Multiple data sources:** Silver used a variety of data sources, including polls, economic indicators, and demographic data, to create a comprehensive view of the electoral landscape.
  - **Historical accuracy incorporation:** By considering historical data, Silver was able to identify patterns and trends that informed his predictions.
  - **Statistical models:** He employed sophisticated statistical models to analyze the data, which helped in understanding the dynamics of voter behavior.
  - **Understanding correlations:** Recognizing how different factors are related allowed Silver to refine his predictions and account for potential biases.
  - **Monte-Carlo simulations for electoral probabilities:** These simulations helped in estimating the likelihood of different electoral outcomes by running numerous scenarios.
  - **Focus on probabilities:** Instead of making absolute predictions, Silver focused on the probabilities of various outcomes, which provided a more nuanced understanding of the election.
  - **Effective communication:** Silver's ability to communicate complex statistical concepts in an accessible way helped the public understand the uncertainties and probabilities involved in election forecasting.

---

## 10 / 14: Examples: Google Flu Trends

### Examples: Google Flu Trends

- 5% to 20% of the US population gets the flu annually; **40k deaths**
  - Early warnings help in prevention and control
- **Google Flu Trends**
  - Provided early flu outbreak alerts via search query analysis
    - Analyzed 45 search terms
    - Used IP to determine location
  - Predicted regional flu outbreaks 1-2 weeks before CDC
  - Operated from 2008 to 2015
- **Caveat: accuracy issues**
  - Claimed 97% accuracy
  - Lower out-of-sample accuracy (overshot CDC data by 30%)
  - People search about flu without confirmed diagnosis
    - E.g., searching "fever" and "cough"



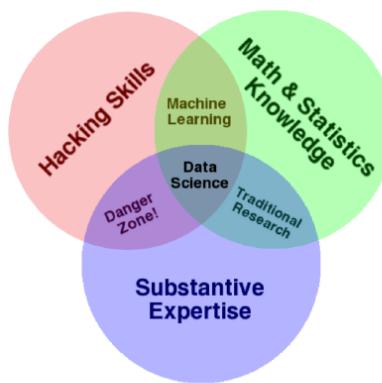
10 / 14

- **5% to 20% of the US population gets the flu annually; 40k deaths**
  - The flu is a significant health concern in the US, affecting a large portion of the population each year and resulting in approximately 40,000 deaths. This highlights the importance of early detection and prevention strategies to mitigate its impact.
- **Google Flu Trends**
  - Google Flu Trends was an innovative project that aimed to provide early warnings of flu outbreaks by analyzing search queries. By examining 45 specific search terms related to flu symptoms and using IP addresses to determine the location of searches, Google attempted to predict where flu outbreaks might occur.
  - The tool was able to predict regional flu outbreaks 1-2 weeks before the Centers for Disease Control and Prevention (CDC), offering potentially valuable lead time for public health responses.
  - The project ran from 2008 to 2015, showcasing the potential of big data and machine learning in public health surveillance.
- **Caveat: accuracy issues**
  - Although Google Flu Trends claimed a high accuracy rate of 97%, it faced challenges with accuracy, particularly when applied to new data (out-of-sample). It sometimes overestimated flu cases by as much as 30% compared to CDC data.
  - One reason for this discrepancy is that people often search for flu-related symptoms like "fever" and "cough" without having a confirmed flu diagnosis. This behavior can lead to overestimations in flu predictions, as not all searches correlate directly with actual flu cases.

## 11 / 14: Data Scientist

### Data Scientist

- Ambiguous, ill-defined term
- From Drew Conway's Venn Diagram



11 / 14

- **Ambiguous, ill-defined term**

- The term *data scientist* is often used in various contexts, leading to some confusion about what it precisely means. It can encompass a wide range of skills and responsibilities, making it a bit tricky to pin down.
- In general, a data scientist is someone who uses data to solve problems. This can involve analyzing data, building models, and communicating insights to help make decisions.
- The role can vary significantly between organizations. Some might focus more on statistical analysis, while others might emphasize machine learning or data engineering.

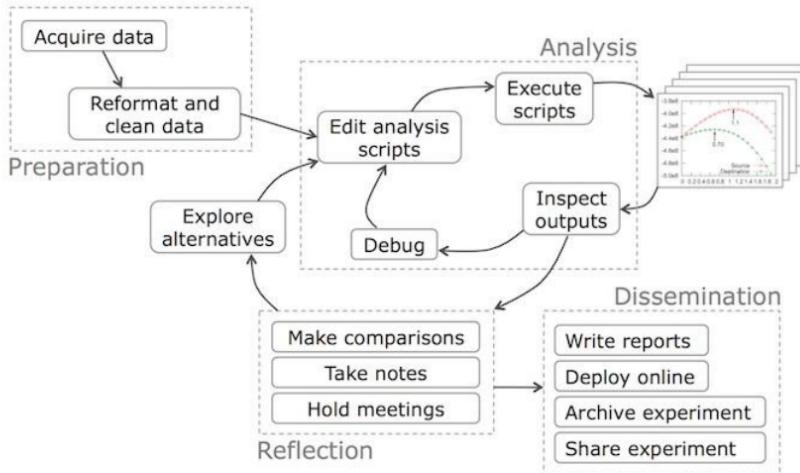
- **From Drew Conway's Venn Diagram**

- Drew Conway's Venn Diagram is a popular way to visualize the skill set of a data scientist. It highlights the intersection of three main areas: *hacking skills*, *math and statistics knowledge*, and *substantive expertise*.
- *Hacking skills* refer to the ability to manipulate and analyze data using programming languages like Python or R.
- *Math and statistics knowledge* is crucial for understanding data patterns and building predictive models.
- *Substantive expertise* involves having domain knowledge to apply data science effectively in a specific field, such as finance, healthcare, or marketing.
- The diagram helps illustrate why the role of a data scientist can be so varied and why it requires a diverse skill set.

## 12 / 14: Typical Data Scientist Workflow

### Typical Data Scientist Workflow

- From Data Science Workflow



12 / 14

- Typical Data Scientist Workflow
  - From Data Science Workflow

In this slide, we are looking at a visual representation of a *typical data scientist's workflow*. This workflow is a series of steps that data scientists follow to extract insights and value from data. Let's break down the key components:

- Data Collection:** This is the first step where data scientists gather raw data from various sources. This could include databases, APIs, or even web scraping. The goal is to collect relevant data that will be used for analysis.
- Data Cleaning:** Once the data is collected, it often needs cleaning. This involves removing duplicates, handling missing values, and correcting errors. Clean data is crucial for accurate analysis.
- Data Exploration:** In this phase, data scientists explore the data to understand its structure and characteristics. They use statistical methods and visualization tools to identify patterns and trends.
- Model Building:** After understanding the data, the next step is to build predictive models. This involves selecting appropriate algorithms and training them on the data to make predictions or classifications.
- Model Evaluation:** Once a model is built, it needs to be evaluated to ensure its accuracy and reliability. This involves testing the model on new data and using metrics to assess its performance.
- Deployment:** The final step is deploying the model into a production environment where it

---

can be used to make real-time decisions or predictions.

This workflow is iterative, meaning data scientists often revisit previous steps to refine their models and improve results. Understanding this workflow is essential for anyone looking to work in data science, as it provides a structured approach to solving complex data problems.

## 13 / 14: Where Data Scientist Spends Most Time

### Where Data Scientist Spends Most Time

- 80-90% of the work is data cleaning and wrangling
- “Janitor Work” in Data Science

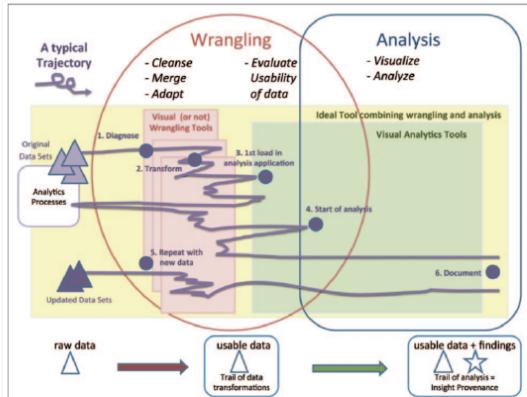


Figure 1. The iterative process of wrangling and analysis. One or more initial data sets may be used and new versions may come later. The wrangling and analysis phases overlap. While wrangling tools tend to be separated from the visual analysis tools, the ideal system would provide integrated tools (light yellow). The purple line illustrates a typical iterative process with multiple back and forth steps. Much wrangling may need to take place before the data can be loaded within visualization and analysis tools, which typically immediately reveals new problems with the data. Wrangling might take place at all the stages of analysis as users sort out interesting insights from dirty data, or new data become available or needed. At the bottom we illustrate how the data evolves from raw data to usable data that leads to new insights.



13 / 14

- **80-90% of the work is data cleaning and wrangling**
  - Data scientists spend a significant portion of their time on tasks related to preparing data for analysis. This involves cleaning and organizing raw data to make it usable.
  - Data cleaning includes removing errors, handling missing values, and ensuring consistency in data formats.
  - Data wrangling involves transforming and mapping data from one “raw” form into another format to make it more appropriate and valuable for a variety of downstream purposes, such as analytics.
  - This step is crucial because the quality of the data directly impacts the quality of the insights derived from it.
- **“Janitor Work” in Data Science**
  - The term “janitor work” is often used to describe the less glamorous, yet essential, tasks of cleaning and organizing data.
  - While it might sound mundane, this work is critical because it lays the foundation for any successful data analysis or machine learning project.
  - Without proper data cleaning and wrangling, the results of data analysis can be misleading or incorrect.
  - This highlights the importance of attention to detail and thoroughness in the early stages of data science projects.

*The image on the slide likely illustrates the proportion of time spent on these tasks, emphasizing their significance in the data science workflow.*

---

## 14 / 14: What a Data Scientist Should Know

### What a Data Scientist Should Know

- **Data grappling skills** ← DATA605
  - Move and manipulate data with programming
  - Scripting languages (e.g., Python)
  - Data storage tools: relational databases, key-value stores
  - Programming frameworks: SQL, Hadoop, Spark
- **Data visualization experience**
  - Draw informative data visuals
  - Tools: D3.js, plotting libraries
  - Know what to draw
- **Knowledge of statistics**
  - Error-bars, confidence intervals
  - Python libraries, Matlab, R
- **Experience with forecasting and prediction**
  - Basic machine learning techniques
- **Communication skills** ← DATA605
  - Tell the story, communicate findings



14 / 14

- **Data grappling skills** ← DATA605
  - *Move and manipulate data with programming:* This means being able to handle data efficiently using code. It's about cleaning, transforming, and preparing data for analysis.
  - *Scripting languages (e.g., Python):* Python is a popular language for data science due to its simplicity and powerful libraries. Knowing how to write scripts in Python is crucial for automating data tasks.
  - *Data storage tools: relational databases, key-value stores:* Understanding how to store and retrieve data is essential. Relational databases like MySQL or PostgreSQL and key-value stores like Redis are common tools.
  - *Programming frameworks: SQL, Hadoop, Spark:* SQL is used for querying databases, while Hadoop and Spark are frameworks for processing large datasets. These tools help manage and analyze big data efficiently.
- **Data visualization experience**
  - *Draw informative data visuals:* Creating visuals that clearly communicate data insights is key. This involves choosing the right type of chart or graph to represent the data.
  - *Tools: D3.js, plotting libraries:* D3.js is a JavaScript library for creating dynamic, interactive data visualizations. Other plotting libraries like Matplotlib or Seaborn in Python are also widely used.
  - *Know what to draw:* It's important to understand which visual representation best suits the data and the message you want to convey.
- **Knowledge of statistics**
  - *Error-bars, confidence intervals:* These are statistical tools used to express the uncertainty in data. Understanding them helps in making informed decisions based on data analysis.

- 
- *Python libraries, Matlab, R*: These are tools and languages commonly used for statistical analysis. Each has its strengths, and knowing how to use them can enhance data analysis capabilities.
  - **Experience with forecasting and prediction**
    - *Basic machine learning techniques*: This involves using algorithms to make predictions based on data. Understanding the basics of machine learning is crucial for building predictive models.
  - **Communication skills** ← DATA605
    - *Tell the story, communicate findings*: Being able to explain data insights in a clear and compelling way is vital. This involves not just presenting data but also telling a story that highlights the key findings and their implications.

---

## Lesson 1.4: Data Models



UMD DATA605: Big Data Systems

**Instructor:** Dr. GP Saggese, [gsaggese@umd.edu](mailto:gsaggese@umd.edu)



1 / 17

### Data Models

- **Data modeling**
  - Represents and captures structure and properties of real-world entities
  - Abstraction: real-world → **representation**
- **Data model**
  - Describes how data is *represented* (e.g., relational, key-value) and *accessed* (e.g., insert operations, query)
  - Schema in a DB describes a specific data collection using a data model
- **Why need data model?**
  - Know data structure to write general-purpose code
  - Share data across programs, organizations, systems
  - Integrate information from multiple sources
  - Preprocess data for efficient access (e.g., building an index)

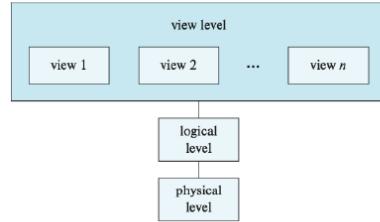


2 / 17

- **Data modeling**
  - *Data modeling* is the process of creating a visual representation of the structure and properties of real-world entities. Think of it as creating a blueprint that helps us understand and organize data. This process involves taking complex real-world scenarios and simplifying them into a format that computers can work with, which is known as *abstraction*. Essentially, we are translating the real world into a digital format that can be easily managed and manipulated.
- **Data model**
  - A *data model* is a framework that defines how data is structured and accessed. For example, in a relational data model, data is organized into tables, while in a key-value model, data is stored as pairs. The data model also dictates how data can be inserted, queried, and updated. A *schema* in a database is a specific instance of a data model that describes the organization of a particular data collection.
- **Why need data model?**
  - Having a data model is crucial because it provides a clear structure for data, which allows developers to write code that can handle various data scenarios. It also facilitates data sharing across different programs, organizations, and systems, ensuring consistency and compatibility. Moreover, data models enable the integration of information from multiple sources, making it easier to combine and analyze data. Additionally, they help in preprocessing data for efficient access, such as by building an index, which speeds up data retrieval and improves performance.

### Multiple Layers of Data Modeling

- **Physical layer**
  - How is the data physically stored
  - How to represent complex data structures (e.g., B-trees for indexing)
- **Logical layer**
  - Entities
  - Attributes
  - Type of information stored
  - Relationships among the above
- **Views**
  - Restrict information flow
  - Security and/or ease-of-use



3 / 17

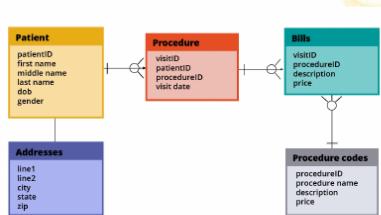
- **Physical layer**
  - This layer deals with the actual storage of data on hardware. It involves understanding how data is organized on physical media like hard drives or SSDs. For example, using *B-trees* is a method to efficiently index and retrieve data, which is crucial for performance in large databases.
  - The focus here is on optimizing storage and retrieval processes, ensuring that data can be accessed quickly and efficiently.
- **Logical layer**
  - This layer is about the abstract representation of data. It defines the *entities* (like tables in a database) and their *attributes* (the columns in those tables).
  - It also describes the *type of information* stored, such as integers, strings, or dates, and the *relationships* between different entities, like how a customer might be linked to their orders.
  - This layer is crucial for designing databases that accurately reflect the real-world scenarios they are meant to model.
- **Views**
  - Views are a way to present data to users in a specific format. They can be used to *restrict information flow*, ensuring that users only see the data they are authorized to access.
  - This is important for *security* and can also make the system easier to use by simplifying complex data structures into more understandable formats for end-users.

## 4 / 17: Data Models: Logical Layer

### Data Models: Logical Layer

- **Modeling constructs**

- Concepts to represent data structure
- E.g.,
  - Entity types
  - Entity attributes
  - Relationships between entities
  - Relationships between attributes



- **Integrity constraints**

- Ensure data integrity
  - Avoid errors and inconsistencies
  - E.g., field can't be empty, must be an integer

- **Manipulation constructs**

- E.g., insert, update, delete data



4 / 17

- **Modeling constructs**

- These are the tools and concepts used to define how data is structured within a database. They help in organizing data in a way that makes it easy to understand and manipulate.
- *Entity types* refer to the different categories or classes of data, like “Customer” or “Product.”
- *Entity attributes* are the properties or details about an entity, such as a customer’s name or a product’s price.
- *Relationships between entities* describe how different entities are connected, like a customer placing an order.
- *Relationships between attributes* define how attributes within an entity relate to each other, ensuring consistency and logical connections.

- **Integrity constraints**

- These are rules that ensure the accuracy and consistency of data within the database.
- They help prevent errors, such as entering incorrect data types or leaving important fields empty.
- For example, a constraint might require that a phone number field must contain only numbers and not be left blank.

- **Manipulation constructs**

- These are the operations that allow users to interact with the data, such as adding new data (insert), changing existing data (update), or removing data (delete).
- These constructs are essential for maintaining and updating the database as new information becomes available or as old information becomes obsolete.

### Data Independence

- **Logical data independence**
  - Change data representation without altering programs
  - E.g., API abstracting backend
- **Physical data independence**
  - Change data layout on disk without altering programs
    - Index data
    - Partition/distribute/replicate data
    - Compress data
    - Sort data



5 / 17

- **Data Independence**

- **Logical data independence**

- \* This concept refers to the ability to change the way data is represented without needing to modify the programs that use this data. For example, if you have an application that accesses a database through an API, you can change how the data is structured or stored in the backend without needing to change the application itself. This is important because it allows developers to improve or optimize the data model without disrupting the applications that rely on it.

- **Physical data independence**

- \* This involves changing how data is physically stored on disk without affecting the programs that access it. Techniques like indexing, partitioning, distributing, replicating, compressing, and sorting data can be used to optimize performance or storage efficiency. For instance, you might decide to index certain columns to speed up queries or compress data to save space. These changes can be made without altering the application code, which means that applications remain stable and unaffected by these optimizations. This flexibility is crucial for maintaining and scaling large systems efficiently.

## Examples of Data Models

- **Some examples of data models**
  - Relational model (SQL)
  - Entity-relationship (ER) model
  - XML
  - Object-oriented (OO)
  - Object-relational
  - RDF
  - Property graph
- **Serialization formats as data models**
  - CSV
  - Parquet
  - JSON
  - Protocol Buffer
  - Avro/Thrift
  - Python Pickle



6 / 17

- **Some examples of data models**
  - **Relational model (SQL)**: This is one of the most common data models used in databases. It organizes data into tables (or relations) with rows and columns. Each table represents a different entity, and SQL (Structured Query Language) is used to manage and query the data.
  - **Entity-relationship (ER) model**: This model is used to visually represent the data and its relationships. It uses entities (things we want to store information about) and relationships (how these entities are related) to design a database.
  - **XML**: Extensible Markup Language is a flexible text format used to store and transport data. It is both human-readable and machine-readable, making it useful for data interchange between systems.
  - **Object-oriented (OO)**: This model organizes data as objects, similar to how object-oriented programming languages like Java or C++ work. It allows for more complex data structures and relationships.
  - **Object-relational**: This combines features of both relational and object-oriented models, allowing for complex data types and relationships while still using a relational database.
  - **RDF**: Resource Description Framework is used to represent information about resources on the web. It is a standard model for data interchange on the web, often used in semantic web applications.
  - **Property graph**: This model represents data as nodes, edges, and properties. It is used in graph databases, which are useful for applications that involve complex relationships, like social networks.
- **Serialization formats as data models**

- 
- **CSV:** Comma-Separated Values is a simple format for storing tabular data in plain text. Each line of the file is a data record, and each record consists of fields separated by commas.
  - **Parquet:** This is a columnar storage file format optimized for use with big data processing frameworks like Apache Hadoop and Apache Spark. It is efficient for both storage and query performance.
  - **JSON:** JavaScript Object Notation is a lightweight data interchange format that is easy for humans to read and write and easy for machines to parse and generate. It is widely used in web applications.
  - **Protocol Buffer:** Developed by Google, this is a method for serializing structured data. It is more efficient than XML and JSON in terms of size and speed, making it suitable for communication protocols and data storage.
  - **Avro/Thrift:** These are serialization frameworks that provide rich data structures and a compact binary format. They are often used in distributed systems for data exchange.
  - **Python Pickle:** This is a module in Python used to serialize and deserialize Python objects. It is useful for saving the state of an object to a file or sending it over a network.

### Good Data Models

- **Good data model** should be:
  - Expressive
    - Capture real-world data
  - Easy to use
  - Perform well
- **Trade-off between characteristics**
  - E.g., more powerful models
    - Represent more datasets
    - Harder to use/query
    - Less efficient (e.g., more memory, time)
- **Evolution of data models** captures data structure
  - Structured data → Relational DBs
  - Semi-structured web data → XML, JSON
  - Unstructured data → NoSQL DBs



7 / 17

- **Good data model** should be:
  - **Expressive:** A good data model needs to be able to accurately represent the complexities of real-world data. This means it should be capable of capturing all the necessary details and relationships within the data, allowing for a comprehensive understanding and analysis.
  - **Easy to use:** It's important that the data model is user-friendly. This means that both technical and non-technical users should be able to interact with it without excessive difficulty. An easy-to-use model facilitates better data management and accessibility.
  - **Perform well:** Performance is key in data models. They should be efficient in terms of speed and resource usage, ensuring that data can be processed and retrieved quickly without consuming excessive computational resources.
- **Trade-off between characteristics:**
  - There is often a balance to be struck between the expressiveness, usability, and performance of a data model. For example, more powerful models can represent a wider variety of datasets, which is beneficial for capturing complex data. However, these models can be more challenging to use and query, and they may require more memory and processing time, which can impact performance.
- **Evolution of data models** captures data structure:
  - Over time, data models have evolved to better handle different types of data structures. Initially, structured data was managed using Relational Databases (Relational DBs), which are well-suited for organized data with clear relationships.
  - As the web introduced more semi-structured data, formats like XML and JSON became popular for their flexibility in representing data that doesn't fit neatly into tables.
  - With the rise of unstructured data, such as text and multimedia, NoSQL databases

---

emerged to provide more scalable and flexible solutions for storing and querying this type of data.

---

## 8 / 17: A Brief History of Databases (Early 1960s)

### A Brief History of Databases (Early 1960s)

- **1960s: Early beginning**
  - Computers become attractive technology
  - Enterprises adopt computers
  - Applications use own data stores
    - Each application has its own format
    - Data unavailable to other programs
- **Database:** term for “shared data banks” by multiple applications
  - Define data format
  - Store as “data dictionary” (schema)
  - Implement “database management” software to access data
- **Issues**
  - How to write data dictionaries?
  - How to access data?
  - Who controls the data?
    - E.g., integrity, security, privacy concerns



8 / 17

- A Brief History of Databases (Early 1960s)
- **1960s: Early beginning**
  - During the 1960s, computers started to become a popular and attractive technology for businesses. This was a time when companies began to see the potential of using computers to improve their operations.
  - As a result, many enterprises started adopting computers to help with various tasks. However, each application or program that a company used had its own way of storing data. This meant that the data was often in a unique format specific to that application.
  - Because of this, the data used by one application was not easily accessible or usable by other programs. This created a challenge for businesses that wanted to integrate and share data across different applications.
- **Database:** term for “shared data banks” by multiple applications
  - The concept of a “database” emerged as a solution to these challenges. A database was envisioned as a shared data bank that multiple applications could use.
  - To make this work, it was necessary to define a standard format for the data. This standard format was known as a “data dictionary” or schema, which described how the data was organized.
  - Additionally, software known as “database management” systems (DBMS) was developed to help access and manage the data stored in these databases.
- **Issues**
  - There were several issues that needed to be addressed with the introduction of databases.

---

One of the main questions was how to write and maintain these data dictionaries or schemas.

- Another challenge was figuring out the best way to access the data stored in the databases. This involved developing methods and tools to efficiently retrieve and manipulate the data.
- Control over the data was also a significant concern. Questions arose about who should have the authority to manage the data, ensuring its integrity, security, and privacy. These concerns were crucial as they impacted how data was protected and who could access it.

### A Brief History of Databases (1960s)

- **1960s, Hierarchical and Network Model**
  - Connect records of different types
  - Example: connect accounts with customers
  - Network model aimed for generality and flexibility
- IBM's IMS Hierarchical Database (1966)
  - Designed for Apollo space program
  - Predates hard disks
  - Used by over 95% of top Fortune 1000 companies
  - Processes 50 billion transactions daily, manages 15 million GBs of data
- **Cons**
  - Exposed too much internal data (structures/pointers)
  - Leaky abstraction



9 / 17

- A Brief History of Databases (1960s)
- **1960s, Hierarchical and Network Model**
  - In the 1960s, databases were designed to connect different types of records, much like a family tree or a network of roads. This was known as the hierarchical and network model.
  - For example, in a bank, you could connect customer records with their account records, allowing for a structured way to manage relationships between data.
  - The network model was developed to be more general and flexible, allowing for more complex relationships between data, which was a significant advancement at the time.
- **IBM's IMS Hierarchical Database (1966)**
  - IBM developed the IMS (Information Management System) database in 1966 specifically for the Apollo space program, which required a robust system to manage vast amounts of data.
  - This database was created before the invention of hard disks, showcasing its pioneering nature in data management.
  - IMS became incredibly popular, with over 95% of the top Fortune 1000 companies using it, processing an enormous volume of transactions and managing vast amounts of data daily.
- **Cons**
  - One major downside of these early database models was that they exposed too much of the internal workings, such as data structures and pointers, to the users.

- 
- This exposure led to what is known as a “leaky abstraction,” where the complexity of the system was not hidden, making it difficult for users to interact with the database without understanding its intricate details.

## 10 / 17: Relational, Hierarchical, Network Model

### Relational, Hierarchical, Network Model

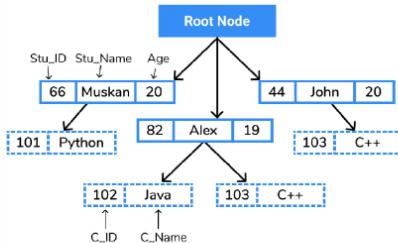
#### • Relational model

- Data as tuples in relations
- SQL

Customer ID	Tax ID	Name	Address	[More fields...]
1234567890	555-551222	Ramesh	323 Southern Avenue	...
2223344556	555-552323	Adam	1200 Main Street	...
3334445563	555-553333	Shweta	871 Rani Jhansi Road	...
4232342432	555-532553	Sarfaraz	123 Maulana Azad Sarani	...

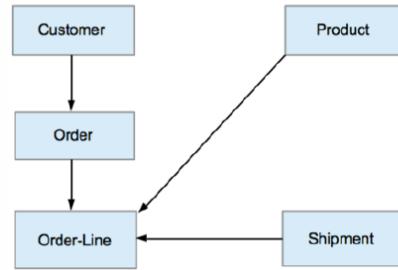
#### • Hierarchical model

- Tree-like structure
  - One parent, many children
  - Connected through links
- XML DBs resurgence in 1990s



#### • Network model

- Graph organization
  - Multiple parents and children
- Graph DBs resurgence in 2010s



#### • Relational model

- This model organizes data into tables, which are technically called *relations*. Each table consists of rows, known as *tuples*, and columns, which represent different data attributes. This model is widely used because it allows for easy data manipulation and querying using a language called SQL (Structured Query Language). SQL is a powerful tool for managing and retrieving data from relational databases, making this model very popular in various applications.

#### • Hierarchical model

- In this model, data is organized in a tree-like structure. Each record has a single parent and can have multiple children, similar to a family tree. This structure is connected through links, making it efficient for certain types of data retrieval. The hierarchical model saw a resurgence in the 1990s with the rise of XML databases, which use a similar tree structure to store and manage data.

#### • Network model

- The network model uses a graph-based organization, allowing for more complex relationships between data. Unlike the hierarchical model, a record in the network model can have multiple parents and children, providing greater flexibility. This model became popular again in the 2010s with the emergence of graph databases, which are particularly useful for applications involving complex relationships, such as social networks or recommendation systems.

### A Brief History of Databases (1970s)

- **1970s: Relational model**
  - Set theory, first-order predicate logic
    - Ted Codd developed the Relational Model
  - Elegant, formal model
    - Provided data independence
    - Users didn't worry about data storage, processing
  - High-level query language
    - SQL based on relational algebra
  - Notion of normal forms
    - Reason about data and relations
    - Remove redundancies
- **Influential projects**
  - INGRES (UC Berkeley), System R (IBM)
  - Ignored IMS compatibility
- **Debates:**
  - Relational Model vs Network Model proponents



11 / 17

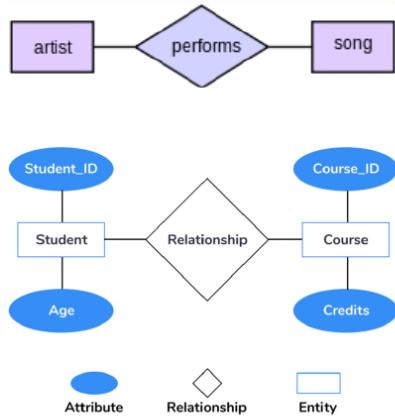
- **1970s: Relational model**
  - **Set theory, first-order predicate logic**
    - \* In the 1970s, Ted Codd introduced the Relational Model, which was a groundbreaking way to organize and manage data. It was based on *set theory* and *first-order predicate logic*, which are mathematical concepts that help in structuring data logically.
  - **Elegant, formal model**
    - \* This model was considered elegant because it provided a clear and formal way to handle data. One of its key benefits was *data independence*, meaning users didn't need to worry about how data was stored or processed. They could focus on what data they needed, not how to get it.
  - **High-level query language**
    - \* SQL, or Structured Query Language, was developed based on relational algebra, allowing users to interact with databases using a high-level language that was easier to understand and use.
  - **Notion of normal forms**
    - \* Normal forms were introduced to help organize data efficiently. They provided a way to think about data and its relationships, aiming to eliminate redundancies and ensure data integrity.
- **Influential projects**
  - Projects like INGRES from UC Berkeley and System R from IBM were pivotal in demonstrating the practical applications of the Relational Model. These projects ignored compatibility with existing systems like IMS, focusing instead on the new relational approach.
- **Debates:**

- 
- During this time, there was a significant debate between proponents of the Relational Model and those who supported the Network Model. The Network Model was another way to organize data, but it was more complex and less flexible compared to the relational approach. This debate highlighted the shift towards more user-friendly and efficient database systems.

## 12 / 17: Entity-Relationship Model

### Entity-Relationship Model

- **Entity-Relationship Model**
  - Proposed in 1976 by Peter Chen
- Describes knowledge as:
  - **Entities:** Physical or logical objects, "Nouns"
  - **Relationships:** Connections between entities, "Verbs"
- Map ER model to relational DB
  - Entities, relationships → tables



12 / 17

- **Entity-Relationship Model**
  - The Entity-Relationship (ER) Model was introduced by Peter Chen in 1976. This model is a way to visually represent the data and its structure in a database. It helps in organizing and understanding the data by breaking it down into entities and relationships.
- **Describes knowledge as:**
  - **Entities:** These are the main components of the ER model. Think of entities as *nouns*; they represent physical or logical objects. For example, in a university database, entities could be "Student," "Course," or "Professor."
  - **Relationships:** These describe how entities are connected to each other, acting as *verbs*. For instance, a "Student" *enrolls in* a "Course," or a "Professor" *teaches* a "Course."
- **Map ER model to relational DB**
  - The ER model is a foundational step in designing a relational database. Entities and relationships are translated into tables. Each entity becomes a table, and relationships often become additional tables or foreign keys that link tables together. This mapping is crucial for creating a structured and efficient database that reflects the real-world scenario it is meant to model.

### A Brief History of Databases (1980s)

- **1980s: Relational model acceptance**
  - SQL standard due to IBM's backing
  - Enhanced relational model
    - Set-valued attributes, aggregation
- **Late 80's**
  - Object-oriented DBs
    - Store objects, not tables
    - Overcome *impedance mismatch* between languages and databases
  - Object-relational DBs
    - User-defined types
    - Combine object-oriented benefits with relational model
  - No expressive difference from pure relational model



13 / 17

- A Brief History of Databases (1980s)
- **1980s: Relational model acceptance**
  - During the 1980s, the relational database model gained widespread acceptance. This was largely due to the support from IBM, a major player in the tech industry. IBM's backing helped establish SQL (Structured Query Language) as the standard language for managing and querying relational databases. This standardization was crucial because it allowed different database systems to communicate using a common language.
  - The relational model was enhanced during this time to include more complex features. For example, set-valued attributes allowed for more flexible data structures, and aggregation functions enabled more sophisticated data analysis directly within the database.
- **Late 80's**
  - In the late 1980s, object-oriented databases emerged. These databases were designed to store data as objects rather than in traditional tables. This approach aimed to address the *impedance mismatch* problem, which is the disconnect between how data is represented in programming languages (as objects) and how it is stored in relational databases (as tables).
  - Object-relational databases were developed to combine the benefits of object-oriented databases with the relational model. They introduced features like user-defined types, allowing for more customized data structures while maintaining the relational framework.
  - Despite these advancements, there was no significant expressive difference between object-relational databases and the pure relational model. This means that while the new models offered additional features, they did not fundamentally change the way

---

data could be expressed or queried compared to traditional relational databases.

---

## 14 / 17: Object-Oriented

### Object-Oriented

- OOP is a data model
  - Object behavior described through data (fields) and code (methods)
- **Composition**
  - has-a relationships
  - E.g., Employee class has an Address class
- **Inheritance**
  - is-a relationships
  - E.g., Employee class derives from Person class
- **Polymorphism**
  - Code executed depends on the class of the object
  - One interface, many implementations
  - E.g., draw() method on a Circle vs Square object, both descending from Shape class
- **Encapsulation**
  - E.g., private vs public fields/members
  - Prevents external code from accessing inner workings of an object



14 / 17

- **Object-Oriented**
  - Object-Oriented Programming (OOP) is a way to model data and behavior in programming. It uses *objects* to represent real-world entities. Each object contains data, known as fields, and code, known as methods, which define its behavior. This approach helps in organizing complex programs by breaking them down into smaller, manageable parts.
- **Composition**
  - Composition is about creating complex types by combining objects. It describes a **has-a** relationship, meaning one object contains or is composed of another. For example, an **Employee** class might have an **Address** class as part of its structure. This allows for building more complex data structures by reusing existing classes.
- **Inheritance**
  - Inheritance is a mechanism where a new class, known as a subclass, is derived from an existing class, known as a superclass. This creates an **is-a** relationship. For instance, an **Employee** class might inherit from a **Person** class, meaning an employee *is a* person. This allows the subclass to inherit fields and methods from the superclass, promoting code reuse.
- **Polymorphism**
  - Polymorphism allows objects to be treated as instances of their parent class, even though they might be instances of a derived class. This means the same operation can behave differently on different classes. For example, a **draw()** method might be implemented differently in a **Circle** class and a **Square** class, both of which are derived from a **Shape** class. This concept allows for flexibility and the ability to extend code easily.
- **Encapsulation**
  - Encapsulation is about restricting access to certain components of an object, typically

---

by using private and public fields or members. This means that the internal state of an object is hidden from the outside, and can only be accessed or modified through specific methods. This helps in protecting the integrity of the data and prevents external code from interfering with the internal workings of an object.

---

## 15 / 17: A Brief History of Databases (1990s)

### A Brief History of Databases (1990s)

- **Late 90's-today**
- Web/Internet emerges
- XML: eXtensible Markup Language
  - For *semi-structured* data
  - Tree-like structure
  - Flexible schema

```
<?xml version="1.0" encoding="UTF-8"?>
<CATALOG>
  <CD>
    <TITLE>Empire Burlesque</TITLE>
    <ARTIST>Bob Dylan</ARTIST>
    <COUNTRY>USA</COUNTRY>
    <COMPANY>Columbia</COMPANY>
    <PRICE>10.90</PRICE>
    <YEAR>1985</YEAR>
  </CD>
  <CD>
    <TITLE>Hide your heart</TITLE>
    <ARTIST>Bonnie Tyler</ARTIST>
    <COUNTRY>UK</COUNTRY>
    <COMPANY>CBS Records</COMPANY>
    <PRICE>9.90</PRICE>
    <YEAR>1988</YEAR>
  </CD>
...
...
```



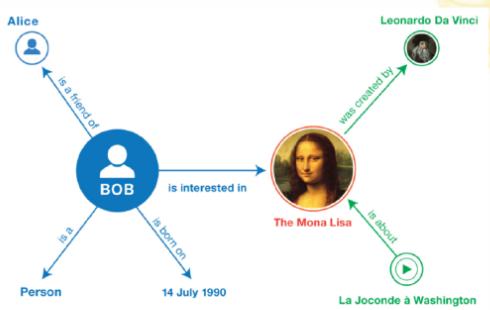
15 / 17

- **Late 90's-today**
  - During the late 1990s, the internet began to become widely accessible, leading to a significant shift in how data was stored and accessed. This era marked the beginning of the digital age, where information could be shared globally at unprecedented speeds. The rise of the web and internet technologies necessitated new ways to handle data that was not strictly structured.
- **Web/Internet emerges**
  - The emergence of the web and the internet revolutionized how businesses and individuals interacted with data. It allowed for the creation of dynamic websites and online services that could serve millions of users. This period saw a move from traditional databases to more flexible systems that could handle the diverse and growing amount of data generated online.
- **XML: eXtensible Markup Language**
  - XML was introduced as a way to manage *semi-structured* data, which is data that doesn't fit neatly into tables like traditional databases. It uses a tree-like structure, making it ideal for representing complex data hierarchies. XML's flexible schema means that it can adapt to different types of data without requiring a fixed structure, which was crucial for the evolving needs of web applications.
- **XML Example**
  - The example provided shows a simple XML document representing a catalog of CDs. Each CD entry includes details like the title, artist, country, company, price, and year. This format allows for easy data exchange between systems, as XML is both human-readable and machine-readable. The flexibility of XML made it a popular choice for data interchange on the web during this time.

## 16 / 17: Resource Description Framework

### Resource Description Framework

- Aka RDF
- **(subject, predicate, object) triple**
- Example of RDF triple
  - Subject=sky
  - Predicate=has-the-color
  - Object=blue
- Maps to a labeled, directed multi-graph
  - More general than a tree
- **Stored in:**
  - Relational DBs
  - Dedicated “triple-stores” DBs

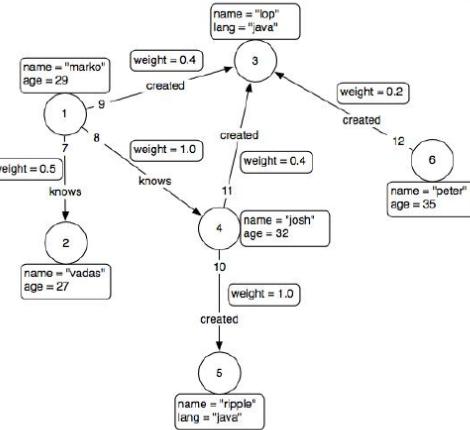


16 / 17

- **Resource Description Framework (RDF)**
  - RDF is a framework used to represent information about resources on the web. It's a standard model for data interchange, which means it helps different systems understand and use data consistently.
- **(subject, predicate, object) triple**
  - RDF uses a simple structure called a “triple” to describe data. Each triple consists of three parts: a *subject*, a *predicate*, and an *object*. This structure is similar to a simple sentence where the subject is the thing being described, the predicate is the property or relationship, and the object is the value or another resource.
- **Example of RDF triple**
  - In the example given, the *subject* is “sky,” the *predicate* is “has-the-color,” and the *object* is “blue.” This triple states that the sky has the color blue.
- **Maps to a labeled, directed multi-graph**
  - RDF triples can be visualized as a graph where nodes represent subjects and objects, and edges represent predicates. This graph is more flexible than a tree because it allows multiple connections between nodes, making it suitable for complex data relationships.
- **Stored in:**
  - RDF data can be stored in traditional relational databases or specialized databases known as “triple-stores,” which are optimized for handling RDF triples. These storage methods help manage and query RDF data efficiently.

## Property Graph Model

- **Graph:**
  - Vertices and edges
  - Properties for each edge and vertex
- **Stored in:**
  - Relational DBs
  - Graph DBs



- **Graph:**
  - *Vertices and edges:* In the property graph model, a graph is made up of two main components: vertices (also known as nodes) and edges. Vertices represent entities or objects, while edges represent the relationships or connections between these entities. For example, in a social network, vertices could represent people, and edges could represent friendships.
  - *Properties for each edge and vertex:* Each vertex and edge can have properties, which are key-value pairs that store additional information. For instance, a vertex representing a person might have properties like “name” and “age,” while an edge representing a friendship might have a property like “since” to indicate when the friendship started. This allows for rich, detailed data representation.
- **Stored in:**
  - *Relational DBs:* Although relational databases are traditionally used for structured data in tables, they can also store graph data. However, this often requires complex queries and joins to represent relationships, which can be less efficient.
  - *Graph DBs:* Graph databases are specifically designed to store and query graph data. They naturally represent vertices and edges, making it easier and faster to perform operations like finding connections or traversing paths. Examples include Neo4j and Amazon Neptune.

---

## Lesson 2.1: Git



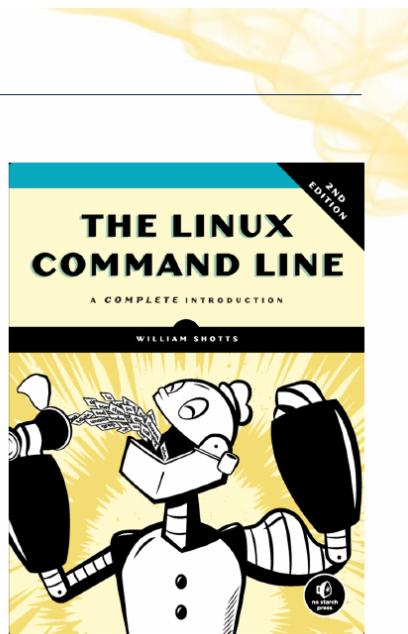
### Lesson 2.1: Git

**Instructor:** Dr. GP Saggese, [gsaggese@umd.edu](mailto:gsaggese@umd.edu)



### Bash / Linux: Resources

- **How Linux works**
  - Processes
  - File ownership and permissions
  - Virtual memory
  - How to administer a Linux box as root
- **Easy**
  - [Command-Line for Beginners](#)
  - E.g., `find`, `xargs`, `chmod`, `chown`, symbolic, and hard links
- **Mastery**
  - [The Linux Command Line](#)



2 / 27

- **How Linux works**
  - *Processes*: In Linux, a process is an instance of a running program. Understanding processes is crucial because they are the basic units of execution in Linux. You can manage processes using commands like `ps`, `top`, and `kill`.
  - *File ownership and permissions*: Linux uses a permission system to control who can read, write, or execute a file. Each file has an owner and a group, and permissions are set for the owner, group, and others. Commands like `chmod` and `chown` are used to change these settings.
  - *Virtual memory*: This is a memory management capability that provides an “idealized abstraction of the storage resources” that are actually available on a given machine. It allows for more efficient and secure use of memory.
  - *How to administer a Linux box as root*: The root user has full control over the system. Administering as root involves tasks like installing software, managing users, and configuring system settings. It’s important to be cautious when operating as root to avoid unintentional system changes.
- **Easy**
  - *Command-Line for Beginners*: This resource is a great starting point for those new to Linux. It covers basic commands and concepts, helping users become comfortable with the command line interface.
  - *E.g., find, xargs, chmod, chown, symbolic, and hard links*: These are fundamental commands and concepts in Linux. `find` is used to search for files, `xargs` is used to build and execute command lines from standard input, `chmod` and `chown` are used to change file permissions and ownership, and symbolic and hard links are methods of creating pointers to files.

---

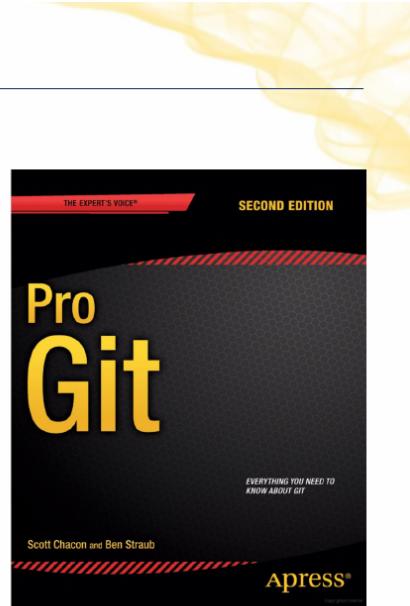
- **Mastery**

- *The Linux Command Line*: This resource is aimed at users who want to deepen their understanding of Linux. It covers more advanced topics and provides a comprehensive guide to mastering the command line, which is essential for efficient system administration and automation.

## 3 / 27: Git Resources

### Git Resources

- Concepts in the slides
- Tutorial: [Tutorial Git](#)
- We will use Git during the project
- Mastery: [Pro Git](#) (free)
- Web resources:
  - <https://githowto.com>
  - [dangitgit.com](https://dangitgit.com) (without swearing)
  - [Oh Sh\\*t, Git!?! \(with swearing\)](https://ohshitgit.com)
- Playgrounds
  - <https://learngitbranching.js.org>



3 / 27

- **Concepts in the slides:** This bullet point suggests that the slide contains key concepts related to Git, a version control system. Understanding these concepts is crucial for managing code and collaborating on projects effectively.
- **Tutorial: Tutorial Git:** This link directs you to a Git tutorial, which is a practical guide to help you get started with Git. It's a hands-on resource that will walk you through the basics of using Git, making it easier to follow along and practice.
- **We will use Git during the project:** This point emphasizes the importance of Git in your coursework or project. It indicates that Git will be a tool you'll need to use, so gaining familiarity with it is essential for successful project completion.
- **Mastery: Pro Git (free):** This is a link to a comprehensive book on Git, available for free. It's an excellent resource for those who want to deepen their understanding and become proficient in using Git.
- **Web resources:**
  - <https://githowto.com>: A website offering step-by-step Git tutorials, ideal for beginners.

- **dangitgit.com (without swearing)**: A humorous resource that provides solutions to common Git problems without using offensive language.
- **Oh Sh\*t, Git!?! (with swearing)**: Similar to the previous resource but includes swearing, offering a light-hearted approach to solving Git issues.

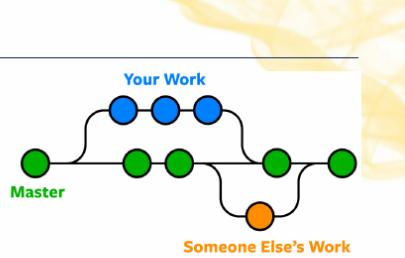
- Playgrounds:

- <https://learngitbranching.js.org>: An interactive platform where you can practice Git commands and concepts in a visual and engaging way. It's a great tool for experimenting with Git without affecting real projects.

## 4 / 27: Git Branching

### Git Branching

- **Branching**
  - Diverge from the main development line
- **Why branch?**
  - Work without affecting the main code
  - Avoid changes in the main branch
  - Merge code downstream for updates
  - Merge code upstream after completion
- **Git branching is lightweight**
  - Instantaneous
  - A branch is a pointer to a commit
  - Git stores data as snapshots, not file differences
- **Git workflows branch and merge often**
  - Multiple times a day
  - Surprising for users of centralized VCS
    - E.g., branch before lunch
  - Branches are cheap
    - Use them to isolate and organize work



- **Branching**
  - *Branching* in Git allows developers to create a separate line of development. This means you can work on new features or bug fixes without interfering with the main codebase. It's like creating a parallel universe where you can experiment freely.
- **Why branch?**
  - Branching lets you work independently without affecting the main code. This is crucial for maintaining stability in the main branch, especially in collaborative environments.
  - By keeping changes isolated, you avoid introducing errors or incomplete features into the main branch.
  - Once your work is ready, you can merge your changes downstream to update your branch with the latest code or upstream to integrate your completed work back into the main branch.
- **Git branching is lightweight**
  - Creating a branch in Git is quick and doesn't require much space. It's essentially just a

pointer to a specific commit in the project's history.

- Unlike some other version control systems, Git stores data as snapshots of the entire project at a given time, rather than just the differences between files. This makes branching and switching between branches very efficient.

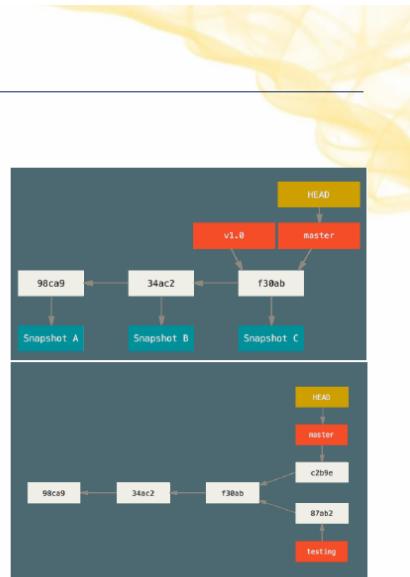
- **Git workflows branch and merge often**

- In Git, it's common to create and merge branches multiple times a day. This might be surprising for those used to centralized version control systems, where branching can be more cumbersome.
- Because branches are easy and inexpensive to create, they are used frequently to keep work organized and isolated. For example, you might create a branch for a new feature before lunch and merge it back into the main branch after testing it in the afternoon.

## 5 / 27: Git Branching

### Git Branching

- **master (or main)** is a normal branch
  - Pointer to the last commit
  - Moves forward with each commit
- **HEAD**
  - Pointer to the current branch
  - E.g., `master`, `testing`
  - `git checkout <BRANCH>` moves across branches
- **git branch testing**
  - Creates a new pointer `testing`
  - Points to the current commit
  - Pointer is movable
- Divergent history
  - Work progresses in two “split” branches



- **master (or main) is a normal branch**

- In Git, the `master` or `main` branch is the default branch where the main development happens. It's essentially a *pointer* to the latest commit in the project. Every time you make a new commit, this pointer moves forward to include the new changes. This is why it's called a “branch”—it represents a line of development.

- **HEAD**

- `HEAD` is a special pointer in Git that tells you which branch you are currently working on. For example, if you're working on the `master` branch, `HEAD` will point to `master`. When you switch branches using `git checkout <BRANCH>`, `HEAD` moves to point to the new branch, allowing you to work on different lines of development.

- **git branch testing**

- This command creates a new branch called `testing`. It acts as a new pointer that starts

at the current commit you're on. This new branch is independent and can move forward as you make new commits. This is useful for experimenting or developing new features without affecting the main branch.

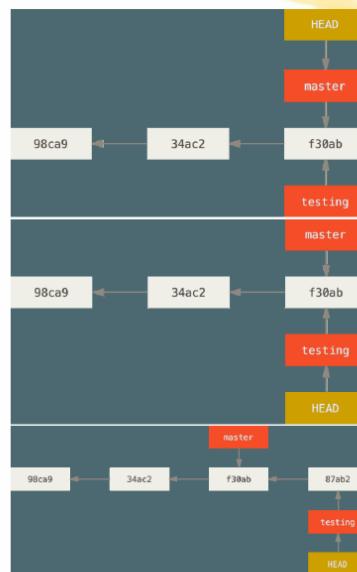
- **Divergent history**

- When you have multiple branches, like `master` and `testing`, they can develop independently. This is known as a divergent history, where each branch can have its own set of commits. This allows for parallel development, where different features or fixes can be worked on simultaneously without interfering with each other.

## 6 / 27: Git Checkout

### Git Checkout

- `git checkout` switches branches
  - Moves `HEAD` pointer to the new branch
  - Changes files in the working directory to match the branch pointer
- E.g., two branches, `master` and `testing`
  - You are on `master`
  - `git checkout testing`
  - Pointer moves, working directory changes
  - Keep working and commit on `testing`
  - Pointer to `testing` moves forward



- **git checkout switches branches**

- The `git checkout` command is used to switch between different branches in a Git repository. This is a fundamental operation when working with Git, as it allows you to move between different lines of development.
- When you use `git checkout`, it moves the `HEAD pointer` to the branch you want to switch to. The `HEAD` is a reference to the current branch you are working on.
- It also updates the files in your working directory to match the state of the branch you have switched to. This means that the files you see and work with will reflect the latest commit on that branch.

- **Example with two branches, `master` and `testing`**

- Imagine you have two branches in your repository: `master` and `testing`. You start on the `master` branch.
- By executing `git checkout testing`, you switch from the `master` branch to the `testing` branch.
- This action moves the `pointer` from `master` to `testing`, and the files in your working

directory are updated to reflect the state of the `testing` branch.

- You can continue to work on the `testing` branch, making changes and committing them. Each commit you make will move the pointer of the `testing` branch forward, capturing your progress.

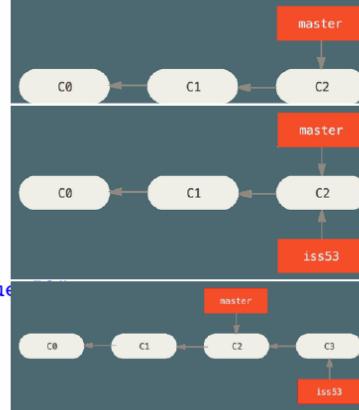
- **Images**

- The images likely illustrate the process of switching branches and how the HEAD pointer and working directory change. They provide a visual representation of the concepts discussed, making it easier to understand how `git checkout` operates in practice.

## 7 / 27: Git Branching and Merging

### Git Branching and Merging

- Tutorials
  - [Work on main](#)
  - [Hot fix](#)
- Start from a project with some commits
- Branch to work on a new feature “Issue 53”  
`> git checkout -b iss53`  
`work ... work ... work`  
`> git commit -m "Add feature for Issue 53"`



- **Git Branching and Merging:** This slide is about using Git, a version control system, to manage changes in a project. It focuses on branching and merging, which are key concepts in Git that help in organizing and integrating different lines of work.
- **Tutorials:** The slide provides links to tutorials that explain how to work on the main branch and how to apply a hot fix. These tutorials are useful for understanding basic Git operations and handling urgent bug fixes without disrupting the main workflow.
- **Start from a project with some commits:** Before branching, it's important to have a project with existing commits. This ensures that you have a stable base to work from and that your changes are built on top of a solid foundation.
- **Branch to work on a new feature “Issue 53”:** The slide demonstrates how to create a new branch for a specific feature or issue. Using the command `git checkout -b iss53`, you create a branch named “iss53” to work on a new feature. This allows you to make changes without affecting the main codebase. After completing the work, you commit the changes with a message describing the feature added.

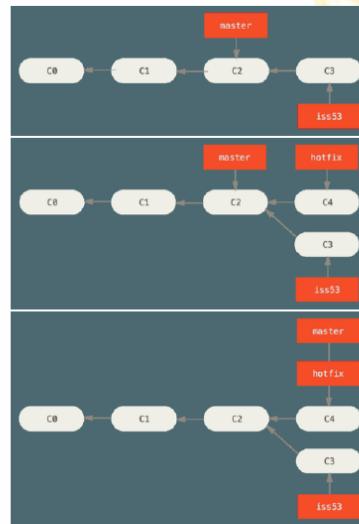
- **Images:** The images likely illustrate the branching and merging process visually, showing how branches diverge from the main line of development and how they are eventually merged back. This visual representation helps in understanding the flow of changes and how different branches relate to each other.

## 8 / 27: Git Branching and Merging

### Git Branching and Merging

- **Need a hotfix to master**

```
> git checkout master
> git checkout -b hotfix
fix ... fix ...
> git commit -am "Hot fix"
> git checkout master
> git merge hotfix
```
- **Fast forward**
  - Now there is no divergent history between `master` and `iss53`

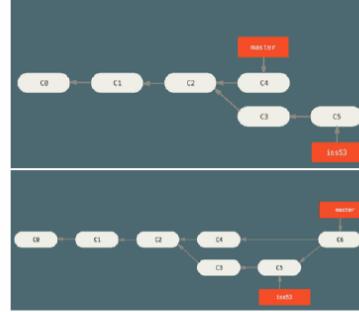


- **Need a hotfix to master**
  - This section explains how to apply a quick fix, known as a *hotfix*, to the `master` branch in Git.
  - The process begins by switching to the `master` branch using `git checkout master`.
  - A new branch named `hotfix` is created with `git checkout -b hotfix`. This allows you to make changes without affecting the `master` branch directly.
  - After making the necessary fixes, you commit the changes with `git commit -am "Hot fix"`. The `-am` flag is used to add and commit changes with a message in one step.
  - Finally, you merge the `hotfix` branch back into `master` using `git merge hotfix`, integrating the changes.
- **Fast forward**
  - This concept refers to a type of merge in Git where the `master` branch is updated to match another branch, like `iss53`, without creating a new commit.
  - A fast forward merge occurs when there is no divergent history between the branches, meaning one branch is directly ahead of the other.
  - This is a simple and efficient way to update branches when no conflicts exist, keeping the commit history clean and linear.

## 9 / 27: Git Branching and Merging

### Git Branching and Merging

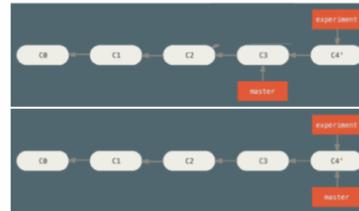
- Keep working on `iss53`
  - > `git checkout iss53`
  - `work ... work ... work`
    - The branch keeps diverging
- At some point you are done with `iss53`
  - You want to merge your work back to `master`
  - Go to the target branch
    - > `git checkout master`
    - > `git merge iss53`
- Git can't fast forward
- Git creates a new snapshot with the 3-way “merge commit” (i.e., a commit with more than one parent)
- Delete the branch
  - > `git branch -d iss53`



- **Keep working on `iss53`**
  - This step involves switching to the branch named `iss53` using the command `git checkout iss53`. This is where you continue to make changes and improvements related to the issue or feature you’re working on. The phrase “`work ... work ... work`” implies that you are actively developing or fixing something on this branch.
  - **The branch keeps diverging:** As you work on `iss53`, the branch may diverge from the `master` branch if other changes are made to `master` in the meantime. This means that the two branches are becoming different from each other.
- **At some point you are done with `iss53`**
  - Once you have completed your work on `iss53`, you will want to integrate these changes back into the main branch, typically `master`.
  - **Go to the target branch:** You switch to the `master` branch using `git checkout master` and then merge the changes from `iss53` into `master` with `git merge iss53`.
- **Git can’t fast forward**
  - Sometimes, Git cannot simply move the `master` branch pointer forward to include the changes from `iss53` because the branches have diverged. In such cases, Git performs a 3-way merge.
- **Git creates a new snapshot with the 3-way “merge commit”**
  - A 3-way merge involves creating a new commit that has more than one parent, effectively combining the histories of both branches. This new commit is called a “merge commit.”
- **Delete the branch**
  - After successfully merging `iss53` into `master`, you can delete the `iss53` branch using `git branch -d iss53`. This is a way to clean up your repository by removing branches that are no longer needed.

### Fast Forward Merge

- Merge a commit Y with a commit X that can be reached by following the history of commit Y
- There is no divergent history to merge
  - Git simply moves the branch pointer forward from X to Y
- **Mental model:** a branch is just a pointer that indicates where the tip of the branch is
- E.g., C4' is reachable from C3
  - > `git checkout master`
  - > `git merge experiment`
- Git moves the pointer of master to C4'



10 / 27

- **Fast Forward Merge:** This is a type of merge in Git where you combine two commits, Y and X, but X is already part of the history of Y. This means that Y is just an extension of X without any separate paths or branches that need to be reconciled.
- **No Divergent History:** In this scenario, there are no conflicting changes or separate lines of development. Git can simply update the branch pointer to the latest commit, Y, without creating a new merge commit. This is efficient and keeps the commit history clean.
- **Mental Model:** Think of a branch as a simple pointer that marks the latest commit in that branch. When you perform a fast forward merge, you're essentially just moving this pointer forward to include the new commits.
- **Example:** If you have a branch `master` and another branch `experiment` with a new commit C4', and C4' is directly reachable from C3, you can merge `experiment` into `master` using:
  - > `git checkout master`
  - > `git merge experiment`

This command moves the `master` branch pointer to C4', effectively incorporating the changes without creating a new merge commit.

- **Visual Representation:** The images likely illustrate how the branch pointer moves from C3 to C4', showing the simplicity and efficiency of a fast forward merge. This visual aid helps in understanding how Git handles such merges seamlessly.

## 11 / 27: Merging Conflicts

### Merging Conflicts

- Tutorial:
  - [Merging conflicts](#)
- Sometimes **Git can't merge**, e.g.,
  - The same file has been modified by both branches
  - One file was modified by one branch and deleted by another
- **Git:**
  - Does not create a merge commit
  - Pauses to let you resolve the conflict
  - Adds conflict resolution markers
- **User merges manually**
  - Edit the files using `git mergetool`
  - Use `git add` to mark as resolved
  - Use `git commit` to finalize the merge
  - Use PyCharm or VS Code for assistance

```
<<<<< HEAD:index.html
<div id="footer">contact : email.support@github.com</div>
=====
<div id="footer">
please contact us at support@github.com
</div>
>>>> iss53:index.html

$ git merge iss53
Auto-merging index.html
CONFLICT (content): Merge conflict in index.html
Automatic merge failed; fix conflicts and then commit the result.

$ git status
On branch master
You have unmerged paths.
  (fix conflicts and run "git commit")

Unmerged paths:
  (use "git add <file>..." to mark resolution)

    both modified:   index.html

no changes added to commit (use "git add" and/or "git commit -a")

$ git status
On branch master
All conflicts fixed but you are still merging.
  (use "git commit" to conclude merge)

Changes to be committed:

modified:   index.html
```



11 / 27

- **Merging Conflicts:** This slide is about handling situations where Git, a version control system, encounters conflicts during the merging process. Merging is when you combine changes from different branches of a project. Sometimes, Git can't automatically merge changes, leading to conflicts.
- **Tutorial:** A link is provided to a tutorial on merging conflicts. This is a helpful resource for understanding how to handle these situations step-by-step.
- **When Git Can't Merge:** Conflicts occur when:
  - Both branches have modified the same file. This means changes have been made to the same lines or sections of code, and Git doesn't know which changes to keep.
  - One branch has modified a file while another branch has deleted it. Git can't decide whether to keep the modifications or accept the deletion.
- **Git's Behavior During Conflicts:**
  - Git won't automatically create a merge commit when it detects conflicts. Instead, it pauses the process.
  - It adds conflict resolution markers in the files, which are special lines that show where the conflicts are.
- **User's Role in Resolving Conflicts:**
  - You need to manually edit the files to resolve conflicts. Tools like `git mergetool` can help with this.
  - Once you've resolved the conflicts, use `git add` to mark the files as resolved.
  - Finally, use `git commit` to complete the merge process.

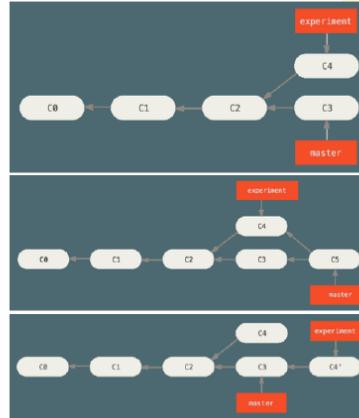
- Integrated development environments (IDEs) like PyCharm or VS Code can provide additional support and make the process easier by offering visual tools to resolve conflicts.

This slide emphasizes the importance of understanding how to handle merge conflicts, as they are a common part of collaborative software development.

## 12 / 27: Git Rebasing

### Git Rebasing

- In Git, there are **two ways of merging divergent history**
- E.g., consider **master** and **experiment** have a common ancestor C2
  - **Merge**
    - Go to the target branch  
 > `git checkout master`  
 > `git merge experiment`
    - Create a new snapshot C5 and commit
  - **Rebase**
    - Go to the branch to rebase  
 > `git checkout experiment`  
 > `git rebase master`
    - Rebase algorithm:
      - Get all the changes committed in the branch (C4) where we are on (**experiment**) since the common ancestor (C2)
      - Sync to the branch that we are rebasing onto (**master** at C3)
      - Apply the changes C4
      - Only the current branch is affected
      - Finally, fast forward **experiment**



12 / 27

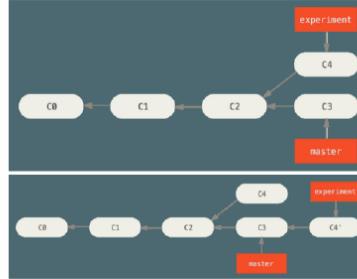
- **Two Ways of Merging Divergent History in Git**
  - In Git, when you have two branches that have diverged from a common point, there are two main strategies to bring them back together: *merge* and *rebase*. Both methods aim to integrate changes from one branch into another, but they do so in different ways.
- **Merge**
  - To merge, you first switch to the target branch where you want to integrate changes. For example, if you want to merge changes from **experiment** into **master**, you would first check out **master**.
  - The command `git merge experiment` combines the histories of the two branches. This creates a new commit, often called a “merge commit” (e.g., C5), which has two parent commits, preserving the history of both branches.
- **Rebase**
  - Rebasing involves changing the base of your branch to a different commit. You start by checking out the branch you want to rebase, such as **experiment**.
  - The rebase process involves taking all the changes made in **experiment** since the common ancestor (C2) and applying them on top of the **master** branch at its current state (C3).
  - This effectively rewrites the history of **experiment**, making it appear as if the changes were made after **master**'s latest commit. Only the **experiment** branch is altered, and it results in a cleaner, linear history.

- After rebasing, the `experiment` branch is fast-forwarded to reflect these changes.

## 13 / 27: Uses of Rebase

### Uses of Rebase

- **Rebasing makes for a cleaner history**
  - The history looks like all the work happened in series
  - Although in reality, it happened in parallel to the development in the `master` branch
- **Rebasing to contribute to a project**
  - Developer
    - You are contributing to a project that you don't maintain
    - You work on your branch
    - When you are ready to integrate your work, rebase your work onto `origin/master`
  - The maintainer
    - Does not have to do any integration work
    - Does just a fast forward or a clean apply (no conflicts)



- **Rebasing makes for a cleaner history**
  - When you rebase, it rearranges the commit history to make it look like all the changes were made one after the other, in a straight line. This is called a linear history.
  - In reality, multiple developers might have been working on different features or fixes at the same time, but rebasing makes it look like these changes were made sequentially. This can make the history easier to read and understand.
- **Rebasing to contribute to a project**
  - *Developer's role:* If you're contributing to a project that someone else maintains, you typically work on your own branch. Once your work is ready to be added to the main project, you rebase your branch onto the latest version of the `origin/master` branch. This ensures your changes are up-to-date with the main project.
  - *Maintainer's role:* The project maintainer benefits because they don't have to manually integrate your changes. They can simply fast forward or apply your changes cleanly, without dealing with conflicts. This makes the process smoother and less error-prone.

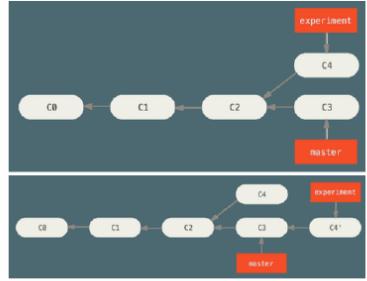
## 14 / 27: Golden Rule of Rebasing

### Golden Rule of Rebasing

- **Remember:** rebasing means abandoning existing commits and creating new ones that are similar but different

- **Problem**

- You push commits to a remote
- Others pull commits and base work on them
- You rewrite commits with `git rebase`
- You push again with `git push --force`
- Collaborators must re-merge work



- **Solution**

- Strict: "*Do not ever rebase commits outside your repository*"
- Loose: "*Rebase your branch if only you use it, even if pushed to a server*"



14 / 27

- **Golden Rule of Rebasing**

- **Remember:** When you rebase, you are essentially taking your existing commits and creating new versions of them. These new commits are similar to the old ones but are technically different. This means that the history of your project changes, which can have significant implications when working with others.

- **Problem**

- Imagine you have pushed your commits to a remote repository, and your teammates have pulled these commits to work on them.
- If you decide to rewrite your commit history using `git rebase`, you are altering the commit history that your teammates are also using.
- When you push these changes back to the remote repository using `git push --force`, it can create confusion and extra work for your collaborators, as they now have to re-merge their work with the new commit history.

- **Solution**

- A strict approach is to *never rebase commits that have been shared outside your personal repository*. This avoids any potential conflicts or confusion for your collaborators.
- A more flexible approach allows for rebasing if you are the only one working on a branch, even if it has been pushed to a server. This is because no one else is relying on that commit history, so there is no risk of disrupting others' work.

- **Visual Aids**

- The images likely illustrate the process of rebasing and the potential issues that can arise when it is not done carefully. They serve as a visual reminder of the importance of following the golden rule of rebasing.

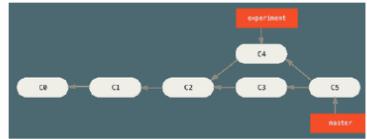
## 15 / 27: Rebase vs Merge: Philosophical Considerations

### Rebase vs Merge: Philosophical Considerations

- Deciding **Rebase-vs-merge** depends on the answer to the question:
  - What does the commit history of a repo mean?*

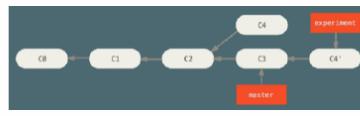
#### 1. History is the record of what actually happened

- "History should not be tampered with, even if messy!"*
- Use `git merge`



#### 2. History represents how a project should have been made

- You should tell the history in the way that is best for future readers"*
- Use `git rebase` and `filter-branch`



15 / 27

#### • Rebase vs Merge: Philosophical Considerations

- The choice between **Rebase** and **Merge** is fundamentally about how you view the commit history of a repository. This decision is not just technical but also philosophical, as it reflects your perspective on the importance and purpose of commit history.

#### • History is the record of what actually happened

- This viewpoint suggests that the commit history should be an accurate reflection of all the changes that have occurred, even if it appears messy or complex. The idea is that history should remain untouched to preserve the authenticity of the development process.
- In this case, using `git merge` is recommended. Merging keeps all the commits intact, showing exactly how the project evolved over time, including all branches and merges.

#### • History represents how a project should have been made

- From this perspective, the commit history is seen as a narrative that should be crafted to be as clear and understandable as possible for future readers. This means organizing and simplifying the history to make it more logical and easier to follow.
- Here, using `git rebase` and `filter-branch` is suggested. Rebasing allows you to rewrite commit history, making it linear and cleaner, which can help in understanding the progression of the project without the clutter of all the branching and merging details.

In summary, the choice between rebase and merge depends on whether you value an accurate historical record or a clean, understandable narrative for future developers.

---

## 16 / 27: Rebase vs Merge: Philosophical Considerations

### Rebase vs Merge: Philosophical Considerations

- **Many man-centuries have been wasted** discussing rebase-vs-merge at the watercooler
    - Total waste of time! Tell people to get back to work!
  - When you contribute to a project often people decide for you based on their preference
  - **Best of the merge-vs-rebase approaches**
    - Rebasing changes you've made in your local repo
      - Even if you have pushed but you know the branch is yours
      - Use `git pull --rebase` to clean up the history of your work
      - If the branch is shared with others then you need to definitively `git merge`
    - Only `git merge` to master to preserve the history of how something was built
  - **Personally**
    - I like to squash-and-merge branches to `master`
    - Rarely are my commits "complete"; they are just checkpoints
-  SCIENCE ACADEMY
- 16 / 27
- **Many man-centuries have been wasted** discussing rebase-vs-merge at the watercooler
    - This point humorously highlights how much time developers spend debating whether to use rebase or merge in version control systems like Git. The suggestion to "get back to work" implies that these discussions can be unproductive and that the choice between rebase and merge might not be as critical as some make it out to be.
  - When you contribute to a project often people decide for you based on their preference
    - In many projects, the decision to use rebase or merge is often made by the project maintainers or team leads. This means that individual contributors might not have a say in the matter and should follow the established workflow to maintain consistency.
  - **Best of the merge-vs-rebase approaches**
    - *Rebase changes you've made in your local repo:* Rebasing is useful for keeping your local changes up-to-date with the main branch. It helps in cleaning up the commit history, making it linear and easier to understand.
      - \* Even if you have pushed but you know the branch is yours: If you are the only one working on a branch, rebasing after pushing is generally safe.
      - \* Use `git pull --rebase` to clean up the history of your work: This command helps in integrating changes from the main branch into your branch without creating unnecessary merge commits.
      - \* If the branch is shared with others then you need to definitively `git merge`: When working on a shared branch, merging is safer to avoid disrupting others' work.
    - Only `git merge` to master to preserve the history of how something was built: Merging into the main branch (often called master) keeps a record of how features were developed and integrated, which can be important for understanding the project's history.
  - **Personally**

- I like to squash-and-merge branches to `master`: Squashing combines all commits from a branch into a single commit before merging, which simplifies the commit history.
- Rarely are my commits “complete”; they are just checkpoints: This suggests that the speaker uses commits as a way to save progress rather than as final, polished changes. Squashing helps in presenting a cleaner history when merging into the main branch.

## 17 / 27: Remote Branches

### Remote Branches

- **Remote branches** are pointers to branches in remote repositories

```
> git remote -v
origin  git@github.com:gpsaggesse/umd_classes.git (fetch)
origin  git@github.com:gpsaggesse/umd_classes.git (push)
```

- **Tracking branches**

- Local references representing the state of the remote repository
- E.g., `master` tracks `origin/master`
- You can't change the remote branch (e.g., `origin/master`)
- You can change the tracking branch (e.g., `master`)
- Git updates tracking branches when you do `git fetch origin` (or `git pull`)

- To share code in a local branch you need to push it to a remote

```
> git push origin serverfix
```

- To work on it

```
> git checkout -b serverfix origin/serverfix
```



17 / 27

- **Remote branches** are essentially bookmarks that point to the state of branches in a remote repository. They help you keep track of what others are working on and the latest updates from the remote repository. When you run the command `git remote -v`, it shows you the URLs of the remote repositories you have configured, both for fetching and pushing changes. This is crucial for collaboration, as it ensures everyone is working with the most current version of the code.
- **Tracking branches** are local branches that have a direct relationship with a remote branch. For example, your local `master` branch might track `origin/master`, meaning it reflects the state of the `master` branch on the remote named `origin`. You can't directly modify a remote branch like `origin/master`; instead, you make changes to your local tracking branch and then push those changes to the remote. When you run `git fetch origin` or `git pull`, Git updates your tracking branches to reflect the latest changes from the remote repository.
- To share your work with others, you need to push your local branch to the remote repository. For instance, if you have a local branch named `serverfix`, you would use the command `git push origin serverfix` to upload it to the remote repository. This makes your changes available to others who have access to the repository.
- If you want to start working on a branch that exists on the remote but not locally, you can

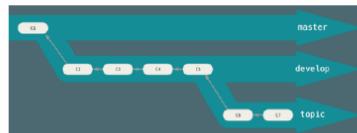
---

create a new local branch that tracks the remote branch. The command `git checkout -b serverfix origin/serverfix` creates a new local branch named `serverfix` that tracks the `serverfix` branch on the remote repository. This allows you to work on the branch locally and later push your changes back to the remote.

## 18 / 27: Git Workflows

### Git Workflows

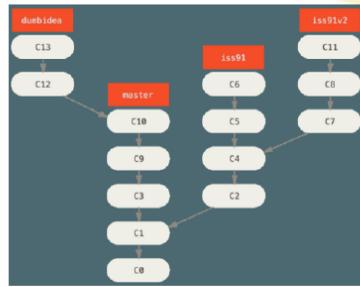
- **Git workflows** = ways of working and collaborating using Git
- **Long-running branches** = branches at different levels of stability that are always open
  - master is always ready to be released
  - develop branch to develop in
  - topic/feature branches
  - When branches are "stable enough," they are merged up



- **Git workflows:** These are structured methods for using Git, a version control system, to manage and collaborate on projects. Git workflows help teams organize their work, manage changes, and ensure that everyone is on the same page. They provide a framework for how code is developed, tested, and deployed.
- **Long-running branches:** These are branches in a Git repository that are continuously maintained and updated. They serve different purposes and have varying levels of stability:
  - **master branch:** This is the main branch that is always ready for release. It contains the most stable version of the project.
  - **develop branch:** This branch is used for ongoing development. It is where new features and fixes are integrated before they are considered stable enough to be merged into the `master` branch.
  - **Topic/feature branches:** These are temporary branches created to work on specific features or fixes. Once the work is complete and stable, these branches are merged into the `develop` branch.
  - The process of merging branches ensures that only stable and tested code is integrated into the main branches, maintaining the integrity of the project.

## Git Workflows

- **Topic branches** = short-lived branches for a single feature
  - E.g., hotfix, wip-XYZ
  - Easy to review
  - Siloed from the rest
  - This is typical of Git since other VCS support for branches is not good enough
  - E.g.,
    - You start iss91, then you cancel some stuff, and go to iss91v2
    - Somebody starts a dumbidea branch and merges it to master (!)
    - You squash-and-merge your iss91v2



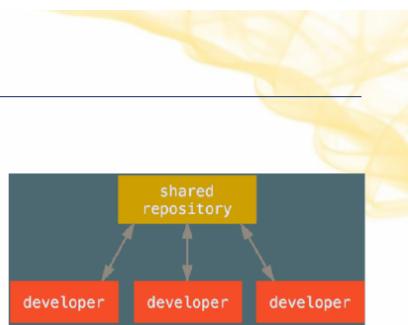
19 / 27

- **Topic branches** are short-lived branches created for working on a single feature or fix. They are temporary and focused, allowing developers to work on specific tasks without affecting the main codebase.
  - Examples of topic branches include `hotfix` for urgent bug fixes and `wip-XYZ` for work-in-progress features.
  - These branches are *easy to review* because they contain changes related to a single feature or fix, making it simpler for team members to understand and provide feedback.
  - Topic branches are *siloed from the rest* of the codebase, meaning they are isolated and do not interfere with the main or other branches until changes are ready to be integrated.
  - Git excels at handling branches, unlike some other version control systems (VCS) where branch support might not be as robust.
  - For example, you might start working on a branch named `iss91`, realize some changes need to be canceled, and then continue on a new branch `iss91v2`.
  - Sometimes, someone might create a branch called `dumbidea` and mistakenly merge it into the `master` branch, which can lead to issues if not reviewed properly.
  - Once your work on `iss91v2` is complete, you can use a *squash-and-merge* strategy to combine all changes into a single commit before merging it into the main branch, keeping the history clean and concise.

## Centralized Workflow

- **Centralized workflow in centralized VCS**

- Developers:
  - Check out the code from the central repo on their computer
  - Modify the code locally
  - Push it back to the central hub (assuming no conflicts with the latest copy, otherwise they need to merge)



- **Centralized workflow in Git**

- Developers:
  - Have push (i.e., write) access to the central repo
  - Need to fetch and then merge
  - Cannot push code that will overwrite each other's code (only fast-forward changes)

- **Centralized workflow in centralized VCS**

- *Developers:*

- \* **Check out the code from the central repo on their computer:** This means that developers download the latest version of the code from a central repository to their local machine. This is the starting point for any changes they want to make.
    - \* **Modify the code locally:** Developers work on their own computers to make changes to the code. This allows them to test and develop features independently.
    - \* **Push it back to the central hub:** Once changes are made, developers upload their modified code back to the central repository. If someone else has changed the code in the meantime, developers may need to merge those changes with their own before pushing.

- **Centralized workflow in Git**

- *Developers:*

- \* **Have push (i.e., write) access to the central repo:** In Git, developers can directly update the central repository, but they need permission to do so.
    - \* **Need to fetch and then merge:** Before pushing changes, developers must fetch the latest updates from the central repository and merge them with their local changes. This ensures that their work is up-to-date with the central version.
    - \* **Cannot push code that will overwrite each other's code:** Git prevents developers from overwriting each other's work. They can only push changes that are compatible with the current state of the central repository, often requiring a fast-forward merge. This helps maintain the integrity of the codebase.

## 21 / 27: Forking Workflows

### Forking Workflows

- Typically, developers don't have permissions to update branches directly on a project
  - Read-write permissions for core contributors
  - Read-only for everybody else
- **Solution**
  - “Forking” a repo
  - External contributors:
    - Clone the repo and create a branch with their work
    - Create a writable fork of the project
    - Push branches to the fork
    - Prepare a PR (Pull Request) with their work
  - Project maintainer:
    - Reviews PRs
    - Accepts PRs
    - Integrates PRs
  - In practice, it's the project maintainer who pulls the code when it's ready, instead of external contributors pushing the code
- **Aka "GitHub workflow"**
  - The “innovation” was forking (Fork me on GitHub!)
  - GitHub was acquired by Microsoft for 7.5 billion USD



21 / 27

#### • Forking Workflows

- In many open-source projects, not all developers have the ability to make changes directly to the project's codebase. This is because only core contributors, who are trusted members of the project, have read-write permissions. Everyone else typically has read-only access, meaning they can view the code but cannot make changes directly.

#### • Solution

- The concept of “forking” a repository is a common solution to this problem. When developers want to contribute to a project but don't have direct access, they can create a “fork,” which is essentially a personal copy of the repository.

#### – External contributors:

- \* They start by cloning the repository, which means they download a copy of the code to their local machine.
- \* They then create a branch in their fork where they can make changes and add new features.
- \* Once their work is complete, they push these changes to their forked repository.
- \* Finally, they submit a Pull Request (PR) to the original project, proposing that their changes be merged into the main codebase.

#### – Project maintainer:

- \* The maintainer of the project reviews the submitted PRs to ensure the changes are beneficial and do not introduce issues.
- \* If the PR is satisfactory, the maintainer accepts it and integrates the changes into the main project.
- \* This process ensures that only vetted and approved code is added to the project, maintaining its quality and integrity.

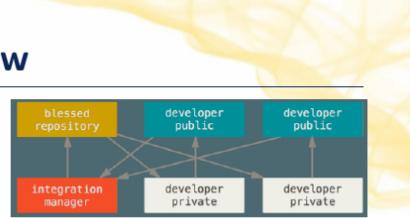
- Aka “GitHub workflow”

- This workflow is often referred to as the “GitHub workflow” because GitHub popularized the concept of forking repositories. The phrase “Fork me on GitHub!” became synonymous with encouraging contributions to open-source projects.
- GitHub’s success and influence in the software development community were significant factors in its acquisition by Microsoft for a substantial sum of 7.5 billion USD, highlighting the importance and value of collaborative coding platforms.

## 22 / 27: Integration-Manager Workflow

### Integration-Manager Workflow

- This is the classical model for open-source development
  - E.g., Linux, GitHub (forking) workflow



#### 1. One repo is the official project

- Only the project maintainer pushes to the public repo
- E.g., causify-ai/csfy

#### 2. Each contributor

- Has read access to everyone else's public repo
- Forks the project into a private copy
  - Write access to their own public repo
  - E.g., gpsaggese/csfy
- Makes changes
- Pushes changes to their own public copy
- Sends a pull request to the maintainer asking to merge changes

#### 3. The maintainer

- Adds the contributor's repo as a remote
- Merges the changes into a local branch
- Tests changes locally
- Pushes the branch to the official repo



SCIENCE  
ACADEMY

22 / 27

- Integration-Manager Workflow

This workflow is a traditional model used in open-source development. It's a structured way to manage contributions from multiple developers, ensuring that the main project remains stable and organized. This model is commonly seen in projects like Linux and on platforms like GitHub, where the concept of “forking” is prevalent.

- One repo is the official project

In this workflow, there is a single repository that is considered the “official” version of the project. Only the project maintainer has the authority to push changes directly to this public repository. For example, in a project named `causify-ai/csfy`, this would be the main repository everyone refers to.

- Each contributor

Contributors have the ability to read from all public repositories, which means they can see what others are working on. They create a “fork” of the project, which is essentially a personal copy where they have write access. For instance, a contributor might fork the project to `gpsaggese/csfy`. They make their changes in this fork and then push these changes to

their own public repository. Once satisfied, they send a “pull request” to the maintainer, requesting that their changes be merged into the official project.

- **The maintainer**

The maintainer plays a crucial role in this workflow. They add the contributor’s repository as a “remote,” which allows them to pull in the changes. They then merge these changes into a local branch on their machine, where they can test and verify that everything works as expected. Once the changes are confirmed to be stable and beneficial, the maintainer pushes the updated branch to the official repository, thus integrating the contributor’s work into the main project.

## 23 / 27: Git log

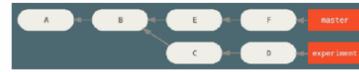
### Git log

- `git log` reports info about commits

- **refs** are references to:

- HEAD (commit you are working on, next commit)
- origin/master (remote branch)
- experiment (local branch)
- d921970 (commit)

- ^ after a reference resolves to the parent of that commit
  - HEAD^ = commit before HEAD, i.e., last commit
  - ^2 means the second parent of a merge commit
  - A merge commit has multiple parents



- **Git log**

- The `git log` command is a powerful tool in Git that provides detailed information about the history of commits in a repository. It allows you to see what changes have been made, who made them, and when they were made. This is crucial for tracking the evolution of a project and understanding the context of changes.

- **Refs (References)**

- **Refs** are pointers to specific commits or branches in your Git repository. They help you navigate through the history of your project.
- **HEAD**: This is the current commit you are working on. It represents the latest state of your working directory and is where your next commit will be based.
- **origin/master**: This refers to the remote branch, typically the main branch on the remote repository. It helps you keep track of changes that have been pushed to the central repository.
- **experiment**: This is an example of a local branch. Local branches are used to develop

features or fixes independently before merging them into the main branch.

- **d921970**: This is an example of a specific commit identifier (SHA-1 hash). Each commit in Git has a unique identifier that allows you to reference it directly.

- **Commit Parent References**

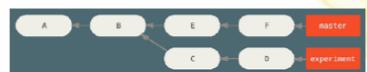
- The `~` symbol is used to navigate through the commit history by referring to parent commits.
- **HEAD<sup>~</sup>**: This refers to the commit immediately before the current HEAD. It's useful for looking at the previous state of the project.
- `~2`: This is used in the context of merge commits, which have more than one parent. It refers to the second parent of a merge commit, allowing you to see the other branch that was merged.
- Understanding parent commits is essential for resolving conflicts and understanding how different branches have been integrated.

## 24 / 27: Dot notation

### Dot notation

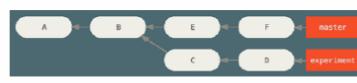
- **Double-dot notation**

- `1..2` = commits that are reachable from 2 but not from 1
- Like a “difference”
- `git log master..experiment` → D,C
- `git log experiment..master` → F,E



- **Triple-dot notation**

- `1...2` = commits that are reachable from either branch but not from both
- Like “union excluding intersection”
- `git log master...experiment` → F,E,D,C



- **Double-dot notation**

- This concept is used in Git to compare two branches or commits. When you see `1..2`, it means you are looking at all the commits that can be reached from commit 2 but not from commit 1. Think of it as finding the “difference” between two points in your project’s history.
- For example, if you run `git log master..experiment`, Git will show you the commits that are in the `experiment` branch but not in the `master` branch. In this case, it would list commits D and C.
- Conversely, `git log experiment..master` will show you the commits that are in `master` but not in `experiment`, which would be F and E.

---

- **Triple-dot notation**

- This notation is slightly different. When you see `1...2`, it refers to all the commits that are reachable from either of the two branches but not from both. It's like taking a "union" of the two sets of commits and then excluding the ones that are common to both.
- For instance, `git log master...experiment` will show you all the commits that are unique to either `master` or `experiment`, which would be F, E, D, and C. This helps you see what changes are unique to each branch without the overlap.

## 25 / 27: Advanced Git

### Advanced Git

- **Stashing**

- Copy the state of your working directory (e.g., modified and staged files)
- Save it in a stack
- Apply it later

- **Cherry-picking**

- Apply a single commit from one branch onto another

- **rerere**

- = "Reuse Recorded Resolution"
- Git caches how to solve certain conflicts

- **Submodules / subtrees**

- Projects including other Git projects



25 / 27

- **Stashing**

- *Stashing* is a useful feature in Git that allows you to temporarily save changes in your working directory. This is particularly helpful when you need to switch branches but aren't ready to commit your current changes. By stashing, you create a snapshot of your modified and staged files and store it in a stack. You can then apply these changes later when you're ready to continue working on them. This helps maintain a clean working directory and prevents incomplete changes from being committed.

- **Cherry-picking**

- *Cherry-picking* is a powerful tool in Git that lets you apply a specific commit from one branch to another. This is useful when you want to incorporate a particular change without merging entire branches. For example, if a bug fix is made in a feature branch, you can cherry-pick that commit to include it in the main branch without merging all other changes from the feature branch.

- **rerere**

- The term *rerere* stands for "Reuse Recorded Resolution." This feature in Git helps auto-

---

mate the resolution of merge conflicts. When you resolve a conflict, Git can remember how you resolved it. If the same conflict occurs again, Git can automatically apply the previously recorded resolution, saving time and reducing repetitive work.

- **Submodules / subtrees**

- *Submodules* and *subtrees* are methods for including one Git repository within another. This is useful for managing dependencies or incorporating external projects. Submodules link to a specific commit of another repository, while subtrees allow you to merge the entire history of the included project. Both methods help manage complex projects with multiple components, ensuring that each part can be developed and maintained independently.

## 26 / 27: Advanced Git

### Advanced Git

- **bisect**

- `git bisect` helps identify the commit that introduced a bug
  - Bug appears at the top of the tree
  - Unknown revision where it started
  - Script returns 0 if good, non-zero if bad
  - `git bisect` finds the revision where the script changes from good to bad

- **filter-branch**

- Rewrite repository history in a scriptable way
  - E.g., change email, remove sensitive file
  - Check out each version, run a command, commit the result

- **Hooks**

- Run scripts before committing, merging, etc

- **bisect**

- *What it does:* `git bisect` is a powerful tool that helps you find the specific commit where a bug was introduced in your code.
- *How it works:* Imagine your code is like a tree, and the bug is at the top. You know the bug exists now, but you're not sure when it started. `git bisect` allows you to mark the current version as “bad” and a previous version as “good.” It then helps you narrow down the exact commit where the bug first appeared by checking out different commits and running a test script. If the script returns 0, the commit is good; if it returns a non-zero value, the commit is bad. This process continues until the problematic commit is identified.

- **filter-branch**

- *Purpose:* This command is used to rewrite the history of a Git repository in a way that can be automated with scripts.

- *Use cases:* You might use `filter-branch` if you need to change the author email in past commits or if you need to remove a file that contains sensitive information from the entire history. It works by checking out each commit, applying a command to it, and then committing the changes back.

- **Hooks**

- *Functionality:* Hooks are scripts that Git can run automatically at certain points in your workflow, such as before you commit changes or merge branches.
- *Why they're useful:* They can help enforce rules or automate tasks. For example, you might use a pre-commit hook to run tests or a linter to ensure code quality before changes are committed. This helps maintain consistency and quality in your codebase.

## 27 / 27: GitHub

### GitHub

- GitHub acquired by MSFT for \$7.5 billion
- **GitHub: largest host for Git repositories**
  - Git hosting (100M+ open source projects)
  - Pull Requests (PRs), forks
  - Issue tracking
  - Code review
  - Collaboration
  - Wiki
  - Actions (CI/CD)
- **"Forking a project"**
  - Open-source communities
    - Negative connotation
    - Modify and create a competing project
  - GitHub parlance
    - Copy a project to contribute without push/write access



- **GitHub acquired by MSFT for \$7.5 billion**
  - In 2018, Microsoft acquired GitHub, a major platform for software development, for \$7.5 billion. This acquisition highlighted the importance of GitHub in the tech industry as a central hub for developers to collaborate and share code.
- **GitHub: largest host for Git repositories**
  - GitHub is the largest platform for hosting Git repositories, with over 100 million open-source projects. It provides essential tools for developers, such as:
    - \* **Git hosting:** Allows developers to store and manage their code.
    - \* **Pull Requests (PRs) and forks:** Facilitate collaboration by enabling developers to propose changes and work on copies of projects.
    - \* **Issue tracking:** Helps manage bugs and feature requests.
    - \* **Code review:** Allows peers to review code changes for quality and consistency.
    - \* **Collaboration:** Encourages teamwork and communication among developers.

- 
- \* **Wiki**: Provides documentation and information about projects.
  - \* **Actions (CI/CD)**: Automates workflows for continuous integration and continuous deployment, streamlining the development process.
  - “**Forking a project**”
    - In open-source communities, “forking” can have a negative connotation, as it sometimes means creating a competing project by modifying the original. However, in GitHub’s context, forking is a positive action. It allows users to copy a project to contribute improvements or features without needing direct write access to the original repository. This process is crucial for open-source collaboration, enabling widespread participation and innovation.

---

## Lesson 2.1: Git



UMD DATA605 - Big Data Systems

1 / 27

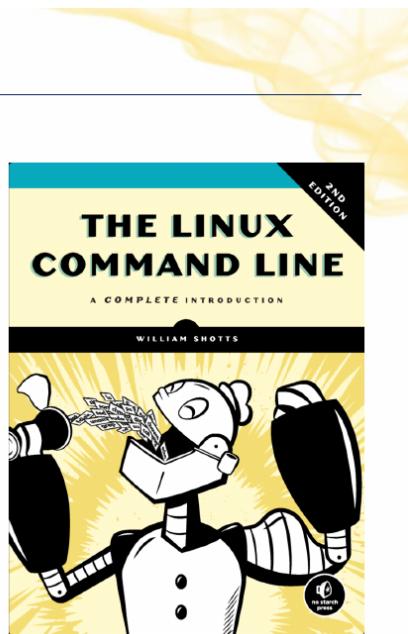
## Lesson 2.1: Git

Instructor: Dr. GP Saggese, [gsaggese@umd.edu](mailto:gsaggese@umd.edu)



### Bash / Linux: Resources

- **How Linux works**
  - Processes
  - File ownership and permissions
  - Virtual memory
  - How to administer a Linux box as root
- **Easy**
  - [Command-Line for Beginners](#)
  - E.g., `find`, `xargs`, `chmod`, `chown`, symbolic, and hard links
- **Mastery**
  - [The Linux Command Line](#)



2 / 27

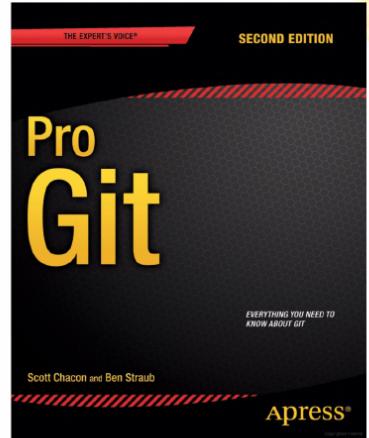
- **How Linux works**
  - *Processes*: In Linux, a process is an instance of a running program. Understanding processes is crucial because they are the basic units of execution in Linux. You can manage processes using commands like `ps`, `top`, and `kill`.
  - *File ownership and permissions*: Every file and directory in Linux has an owner and a set of permissions that determine who can read, write, or execute it. This is important for security and managing access to files.
  - *Virtual memory*: Linux uses virtual memory to extend the physical memory of the system. It allows the system to use disk space as additional RAM, which is essential for running large applications or multiple programs simultaneously.
  - *How to administer a Linux box as root*: The root user has full control over the system. Administering as root involves tasks like installing software, managing users, and configuring system settings. It's important to be cautious when operating as root to avoid accidental system damage.
- **Easy**
  - *Command-Line for Beginners*: This resource is a great starting point for those new to Linux. It covers basic commands and concepts, helping users become comfortable with the command line.
  - *E.g., find, xargs, chmod, chown, symbolic, and hard links*: These are fundamental commands and concepts in Linux. `find` helps locate files, `xargs` is used to build and execute command lines, `chmod` and `chown` are for changing file permissions and ownership, and symbolic/hard links are ways to reference files.
- **Mastery**
  - *The Linux Command Line*: This resource is for those who want to deepen their under-

---

standing of Linux. It covers advanced topics and provides a comprehensive guide to mastering the command line, making it ideal for users who want to become proficient in Linux.

## Git Resources

- Concepts in the slides
- Tutorial: [Tutorial Git](#)
- We will use Git during the project
- Mastery: [Pro Git](#) (free)
- Web resources:
  - <https://githowto.com>
  - [dangitgit.com](https://dangitgit.com) (without swearing)
  - [Oh Sh\\*t, Git!?!?](https://ohshitgit.com) (with swearing)
- Playgrounds
  - <https://learngitbranching.js.org>



3 / 27

- **Concepts in the slides:** This bullet point suggests that the slide presentation includes key concepts related to Git, a version control system. Understanding these concepts is crucial for managing code and collaborating with others in software development projects.
- **Tutorial:** The link provided (Tutorial Git) is a resource for a Git tutorial. This tutorial is likely designed to help beginners get started with Git, covering basic commands and workflows.
- **We will use Git during the project:** This indicates that Git will be an essential tool for the upcoming project. Students should familiarize themselves with Git to effectively manage their code and collaborate with team members.
- **Mastery:** The link to Pro Git offers a free, comprehensive guide to mastering Git. This resource is ideal for those who want to deepen their understanding and become proficient in using Git.
- **Web resources:**
  - githowto.com provides step-by-step instructions for learning Git.
  - dangitgit.com offers solutions to common Git problems without using profanity.
  - Oh Sh\*t, Git!?!? humorously addresses Git issues with a bit of swearing, making it a fun and relatable resource.
- **Playgrounds:** learngitbranching.js.org is an interactive platform where users can practice Git commands and branching strategies in a visual and engaging way. This is particularly useful for hands-on learning and experimentation.

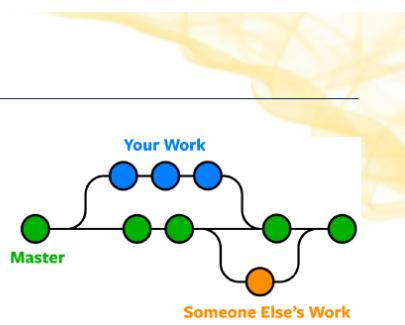
The images on the right side of the slide likely provide visual aids or examples related to the Git

---

concepts discussed, although they are not visible in this text format.

### Git Branching

- **Branching**
  - Diverge from main development line
- **Why branch?**
  - Work without affecting main code
  - Avoid changes in main branch
  - Merge code downstream for updates
  - Merge code upstream after completion
- **Git branching is lightweight**
  - Instantaneous
  - Branch is a pointer to a commit
  - Git stores data as snapshots, not file differences
- **Git workflows branch and merge often**
  - Multiple times a day
  - Surprising for users of distributed VCS
    - E.g., branch before lunch
  - Branches are cheap
    - Use them to isolate and organize work



4 / 27

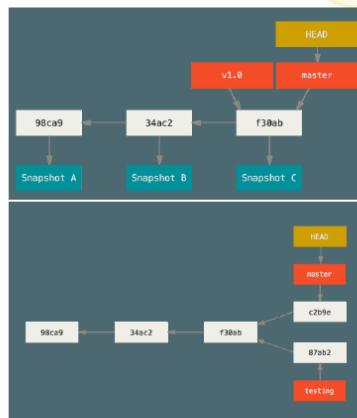
- **Branching**
  - Branching in Git allows developers to create a separate line of development. This means you can work on new features or bug fixes without interfering with the main codebase. It's like creating a parallel universe where you can experiment freely.
- **Why branch?**
  - Branching lets you work independently without affecting the main code. This is crucial for maintaining stability in the main branch, especially in collaborative projects.
  - By keeping changes isolated, you avoid introducing errors into the main branch. This is particularly important in large projects where multiple developers are working simultaneously.
  - Once your work is ready, you can merge your changes downstream to update your branch with the latest code from the main branch. This ensures your work is up-to-date.
  - After completing your work, you can merge your changes upstream into the main branch, integrating your new features or fixes.
- **Git branching is lightweight**
  - Creating a branch in Git is quick and doesn't require much space. It's essentially a pointer to a specific commit, making it efficient.
  - Unlike some other systems, Git stores data as snapshots of the entire project, not just the differences between files. This makes branching and merging fast and reliable.
- **Git workflows branch and merge often**
  - In Git, it's common to branch and merge multiple times a day. This might be surprising for those used to other version control systems, but it's a testament to Git's efficiency.
  - You can create a branch for even small tasks, like before taking a lunch break, to keep your work organized and isolated.

- 
- Since branches are inexpensive to create, they are a great tool for managing different tasks and experiments without cluttering the main codebase.

## 5 / 27: Git Branching

### Git Branching

- **master (or main)** is a normal branch
  - Pointer to the last commit
  - Moves forward with each commit
- **HEAD**
  - Pointer to the local branch
  - E.g., `master`, `testing`
  - `git checkout <BRANCH>` moves across branches
- **git branch testing**
  - Create a new pointer `testing`
  - Points to the current commit
  - Pointer is movable
- Divergent history
  - Work progresses in two “split” branches



5 / 27

- **master (or main) is a normal branch**
  - In Git, the `master` or `main` branch is the default branch where the main line of development occurs. It's essentially a pointer that keeps track of the last commit in the sequence. As you make new commits, this pointer moves forward to include the latest changes, ensuring that the branch always reflects the most recent state of your project.
- **HEAD**
  - HEAD is a special pointer in Git that represents your current working location in the repository. It usually points to the latest commit on the branch you are working on, such as `master` or `testing`. When you use the command `git checkout <BRANCH>`, you are telling Git to move `HEAD` to point to a different branch, effectively switching your working context to that branch.
- **git branch testing**
  - This command creates a new branch named `testing`. It acts as a new pointer that starts at the current commit where `HEAD` is located. This new branch is independent and can be moved forward with new commits, allowing you to experiment or develop features without affecting the `master` branch.
- **Divergent history**
  - When you have multiple branches, such as `master` and `testing`, they can develop independently. This means that each branch can have its own sequence of commits, leading to a “split” or divergent history. This is useful for working on different features or versions of a project simultaneously without interference.

### Git Checkout

- `git checkout` switches branch
  - Move HEAD pointer to new branch
  - Change files in working dir to match branch pointer
- E.g., two branches, `master` and `testing`
  - You are on `master`
  - `git checkout testing`
  - Pointer moves, working dir changes
  - Keep working and commit on `testing`
  - Pointer to `testing` moves forward



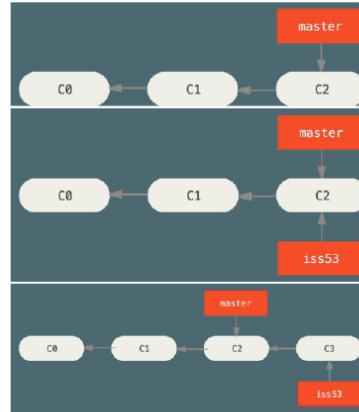
6 / 27

- **git checkout switches branch**
  - The `git checkout` command is used to switch between different branches in a Git repository. This is a fundamental operation in Git that allows you to move between different lines of development.
  - When you use `git checkout` to switch branches, it moves the *HEAD pointer* to the branch you want to work on. The `HEAD` is a reference to the current branch you are working on.
  - It also changes the files in your working directory to match the state of the branch you have switched to. This means that the files you see and work with will be the ones from the branch you have checked out.
- **Example with two branches, master and testing**
  - Imagine you have two branches in your repository: `master` and `testing`. You start on the `master` branch.
  - By executing `git checkout testing`, you switch from the `master` branch to the `testing` branch.
  - This action moves the *pointer* to the `testing` branch, and the files in your working directory are updated to reflect the state of the `testing` branch.
  - You can continue to work on the `testing` branch, making changes and committing them. Each commit will move the pointer of the `testing` branch forward, capturing the new state of your work.
- **Images**
  - The images likely illustrate the process of switching branches and how the `HEAD` pointer and working directory change as a result. These visuals can help you understand the concept of branch switching and how it affects your work in Git.

## 7 / 27: Git Branching and Merging

### Git Branching and Merging

- Tutorials
  - [Work on main](#)
  - [Hot fix](#)
- Start from a project with some commits
- Branch to work on a new feature “Issue 53”  
`> git checkout -b iss53  
work ... work ... work  
> git commit`

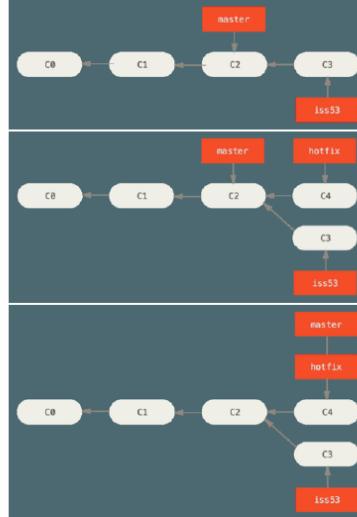


7 / 27

- **Git Branching and Merging:** This slide is about managing different versions of a project using Git, a popular version control system. Branching and merging are key concepts in Git that help developers work on different features or fixes simultaneously without interfering with the main project.
- **Tutorials:** The slide provides links to tutorials that guide you through working on the main branch and applying hot fixes. These tutorials are useful for understanding how to manage your code effectively and ensure that changes are integrated smoothly.
- **Start from a project with some commits:** Before branching, it's important to have a project with existing commits. This means you have a history of changes that have been made to the project, which is essential for tracking progress and understanding the context of your work.
- **Branch to work on a new feature “Issue 53”:** The slide demonstrates how to create a new branch to work on a specific feature or issue. By using the command `git checkout -b iss53`, you create and switch to a new branch named “iss53”. This allows you to work on the feature independently, making changes and committing them without affecting the main branch. This is crucial for maintaining a clean and organized workflow, especially in collaborative environments.
- **Images:** The images likely illustrate the process of branching and merging visually, helping to reinforce the concepts discussed. Visual aids can be very helpful in understanding how branches diverge and merge back into the main project.

### Git Branching and Merging

- **Need a hotfix to master**  
> `git checkout master`  
> `git checkout -b hotfix`  
fix ... fix ... fix  
> `git commit -am "Hot fix"`  
> `git checkout master`  
> `git merge hotfix`
- **Fast forward**
  - Now there is a divergent history between `master` and `iss53`

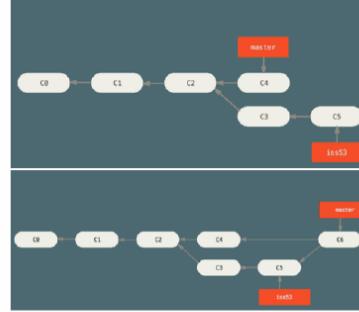


- **Need a hotfix to master**
  - Sometimes, you might encounter a situation where an urgent fix is needed in the main branch of your project, often called `master`. This is known as a *hotfix*.
  - The process begins by switching to the `master` branch using the command `git checkout master`.
  - Next, you create a new branch specifically for the hotfix with `git checkout -b hotfix`. This allows you to work on the fix without affecting the main branch until it's ready.
  - After making the necessary changes, you commit them with a message, for example, `git commit -am "Hot fix"`.
  - Finally, you merge the hotfix back into the `master` branch using `git merge hotfix`. This incorporates the changes into the main branch, resolving the issue.
- **Fast forward**
  - In Git, a *fast forward* occurs when you merge branches and there is no divergent history between them. However, in this case, there is a divergent history between `master` and another branch named `iss53`.
  - This means that changes have been made in both branches that need to be reconciled. Understanding how to handle these situations is crucial for maintaining a clean and functional project history.

## 9 / 27: Git Branching and Merging

### Git Branching and Merging

- Keep working on `iss53`
  - > `git checkout iss53`
  - `work ... work ... work`
    - The branch keeps diverging
- At some point you are done with `iss53`
  - You want to merge your work back to `master`
  - Go to the target branch
    - > `git checkout master`
    - > `git merge iss53`
- Git can't fast forward
- Git creates a new snapshot with the 3-way “merge commit” (i.e., a commit with more than one parent)
- Delete the branch
  - > `git branch -d iss53`



- **Keep working on `iss53`**
  - This step involves switching to the branch named `iss53` using the command `git checkout iss53`. Once on this branch, you continue making changes and committing them. This is a common practice when working on a specific feature or issue, allowing you to isolate your work from the main codebase.
  - **The branch keeps diverging:** As you work on `iss53`, the branch may diverge from the `master` branch if other changes are made to `master` in the meantime. This means the two branches have different histories.
- **At some point you are done with `iss53`**
  - Once your work on `iss53` is complete, you need to integrate these changes back into the main branch, typically `master`. This is done by first switching to the `master` branch using `git checkout master`.
  - **Go to the target branch:** After switching to `master`, you merge the changes from `iss53` into `master` using `git merge iss53`.
- **Git can't fast forward**
  - If Git cannot perform a fast-forward merge, it means that the `master` branch has progressed since `iss53` was created. In such cases, Git creates a new commit called a “merge commit” to combine the histories of both branches.
- **Git creates a new snapshot with the 3-way “merge commit”**
  - A merge commit is a special type of commit that has more than one parent. It represents the point where two branches have been combined, preserving the history of both branches.
- **Delete the branch**
  - After successfully merging `iss53` into `master`, the branch `iss53` is no longer needed. You

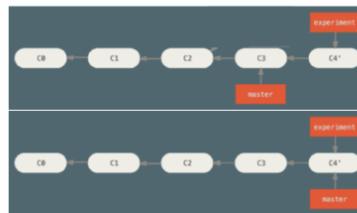
---

can delete it using `git branch -d iss53` to keep your branch list clean and organized. This step is important for maintaining a tidy repository and avoiding clutter from unused branches.

## 10 / 27: Fast Forward Merge

### Fast Forward Merge

- **Fast forward merge**
  - Merge a commit X with a commit Y that can be reached by following the history of commit X
- There is not divergent history to merge
  - Git simply moves the branch pointer forward from X to Y
- **Mental model:** a branch is just a pointer that says where the tip of the branch is
- E.g., C4' is reachable from C3
  - > `git checkout master`
  - > `git merge experiment`
- Git moves the pointer of master to C4'



10 / 27

- **Fast forward merge**
  - When we talk about a *fast forward merge*, we're referring to a situation where you want to merge two commits, X and Y. In this case, commit Y is directly reachable from commit X by simply following the history. This means there are no other changes or branches that have diverged between these two commits.
- **There is not divergent history to merge**
  - In a fast forward merge, Git doesn't have to do any complex merging work. Instead, it just updates the branch pointer to the latest commit. So, if you were on commit X and you want to merge with commit Y, Git will just move the branch pointer from X to Y.
- **Mental model:** a branch is just a pointer that says where the tip of the branch is
  - Think of a branch in Git as a simple pointer. It points to the latest commit in that branch. For example, if you have a branch with commits C1, C2, C3, and C4', and you want to merge C3 with C4', Git will just move the branch pointer from C3 to C4'.
- **Example command**
  - The command `git checkout master` followed by `git merge experiment` is an example of how you might perform a fast forward merge. Here, `master` is the branch you're on, and `experiment` is the branch you want to merge into `master`. If `experiment` is ahead of `master` with no divergent changes, Git will simply move the `master` pointer forward to the latest commit on `experiment`.
- **Visual aids**
  - The images provided likely illustrate the concept of a fast forward merge by showing how the branch pointer moves from one commit to another without any branching or merging conflicts. This visual representation helps in understanding how straightforward a fast forward merge is compared to other types of merges.

### Merging Conflicts

- Tutorial:
  - [Merging conflicts](#)
- Sometimes **Git can't merge**, e.g.,
  - The same file has been modified by both branches
  - One file was modified by one branch and deleted by another
- **Git:**
  - Does not create a merge commit
  - Pauses to let you resolve the conflict
  - Adds conflict resolution markers
- **User merges manually**
  - Edit the files `git mergetool`
  - `git add` to mark as resolved
  - `git commit`
  - Use PyCharm or VS Code

```
<<<<< HEAD:index.html
<div id="footer">contact : email.support@github.com</div>
=====
<div id="footer">
please contact us at support@github.com
</div>
>>>>> iss53:index.html

$ git merge iss53
Auto-merging index.html
CONFLICT (content): Merge conflict in index.html
Automatic merge failed; fix conflicts and then commit the result.

$ git status
On branch master
You have unmerged paths.
  (fix conflicts and run "git commit")

Unmerged paths:
  (use "git add <file>..." to mark resolution)
    both modified:   index.html

no changes added to commit (use "git add" and/or "git commit -a")

$ git status
On branch master
All conflicts fixed but you are still merging.
  (use "git commit" to conclude merge)

Changes to be committed:

modified:   index.html
```



11 / 27

- **Merging Conflicts:** This slide is about handling situations where Git, a version control system, encounters conflicts while trying to merge changes from different branches. Merging conflicts occur when Git cannot automatically combine changes because the same file has been altered in different ways in separate branches, or when one branch modifies a file that another branch deletes.
- **Git's Behavior:**
  - When Git encounters a conflict, it does not automatically create a merge commit. Instead, it pauses the process to allow the user to manually resolve the conflict.
  - Git adds conflict resolution markers in the files to highlight the conflicting sections. These markers help users identify the parts of the code that need attention.
- **User's Role in Conflict Resolution:**
  - Users must manually edit the files to resolve conflicts. This can be done using tools like `git mergetool`, which assists in comparing and editing the conflicting files.
  - After resolving the conflicts, users need to use `git add` to mark the conflicts as resolved, followed by `git commit` to finalize the merge.
  - Integrated development environments (IDEs) like PyCharm or VS Code can be used to simplify the process of resolving merge conflicts, providing a more visual and user-friendly interface.

The images on the right likely illustrate examples of merge conflicts and how they appear in different tools, helping to visualize the process described.

### Git Rebasing

- In Git there are **two ways of merging divergent history**
  - E.g., master and experiment have a common ancestor C2

#### Merge

- Go to the target branch
 

```
> git checkout master
```

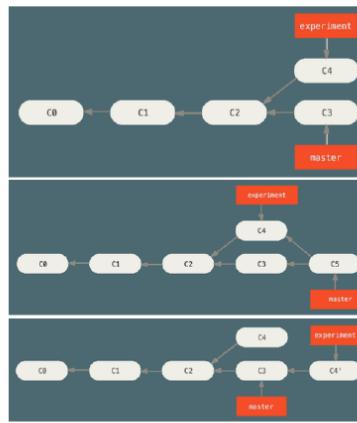
`> git merge experiment`
- Create a new snapshot C5 and commit

#### Rebase

- Go to the branch to rebase
 

```
> git checkout experiment
```

`> git rebase master`
- Rebase algo:
  - Get all the changes committed in the branch (C4) where we are on (`experiment`) since the common ancestor (C2)
  - Sync to the branch that we are rebasing onto (`master` at C3)
  - Apply the changes C4
  - Only current branch is affected
  - Finally fast forward experiment



12 / 27

#### Git Rebasing

- In Git, there are **two ways of merging divergent history**. This means when you have two branches that have developed separately from a common starting point, you can bring them back together using either merging or rebasing. For example, if you have a `master` branch and an `experiment` branch that both originated from a common commit C2, you can use these methods to combine their histories.

#### Merge

- To merge, you first **go to the target branch** where you want to combine the changes. In this case, you switch to the `master` branch using the command:

```
> git checkout master
> git merge experiment
```

- This process creates a new commit, C5, which represents the combined history of both branches. This new commit is a snapshot that includes all changes from both branches.

#### Rebase

- Rebasing involves **going to the branch you want to rebase**. Here, you switch to the `experiment` branch:

```
> git checkout experiment
> git rebase master
```

- The rebase algorithm works by taking all the changes made in the `experiment` branch since the common ancestor C2 (in this case, changes in C4), and then applying them on top of the `master` branch at C3. This effectively moves the base of the `experiment` branch to the tip of the `master` branch.

- Only the current branch (`experiment`) is affected by this operation, and it results in a



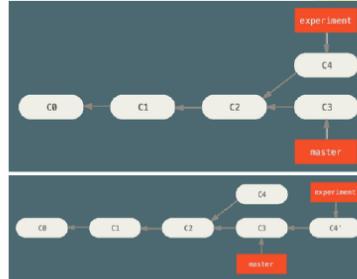
---

cleaner project history by avoiding unnecessary merge commits.

- Finally, the `experiment` branch is fast-forwarded, meaning it now appears as if it was developed directly from the latest commit on `master`.

### Uses of Rebase

- **Rebasing makes for a cleaner history**
  - The history looks like all the work happened in series
  - Although in reality it happened in parallel to the development in master
- **Rebasing to contribute to a project**
  - Developer
    - You are contributing to a project that you don't maintain
    - You work on your branch
    - When you are ready to integrate your work, rebase your work onto origin/master
  - The maintainer
    - Does not have to do any integration work
    - Does just a fast forward or a clean apply (no conflicts)



13 / 27

- **Rebasing makes for a cleaner history**
  - When you rebase, it rearranges the commit history so that it appears as if all the work was done one after the other, in a straight line. This is different from what actually happens, where multiple developers might be working on different features at the same time. By making the history linear, it becomes easier to follow and understand the sequence of changes.
  - *In reality*, development often happens in parallel, with different branches being worked on simultaneously. Rebasing helps to present this parallel work in a more organized and sequential manner.
- **Rebasing to contribute to a project**
  - *Developer*: If you're contributing to a project that you don't own or maintain, you typically work on your own branch. Once your work is ready to be added to the main project, you rebase your branch onto the latest version of the main branch (often called origin/master). This ensures your changes are up-to-date with the latest project developments.
  - *The maintainer*: By rebasing before you submit your changes, you make it easier for the project maintainer. They can simply fast-forward the main branch to include your changes or apply them cleanly without having to resolve conflicts. This streamlines the integration process and reduces the workload for the maintainer.

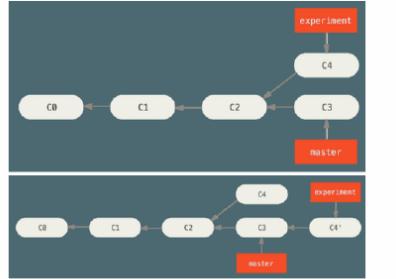
## 14 / 27: Golden Rule of Rebasing

### Golden Rule of Rebasing

- **Remember:** rebasing means abandoning existing commits and creating new ones that are similar but different

- **Problem**

- You push commits to a remote
- Others pull commits and base work on them
- You rewrite commits with `git rebase`
- You push again with `git push --force`
- Collaborators must re-merge work



- **Solution**

- Strict: "*Do not ever rebase commits outside your repository*"
- Loose: "*Rebase your branch if only you use it, even if pushed to a server*"



14 / 27

- **Golden Rule of Rebasing**

- **Remember:** When you rebase, you are essentially taking existing commits and creating new ones that are similar but not identical. This means the history of your project changes, which can lead to complications if not handled carefully.

- **Problem**

- **You push commits to a remote:** This means you've shared your work with others, and they might start using it as a base for their own work.
- **Others pull commits and base work on them:** Your collaborators download your changes and start building their own work on top of it.
- **You rewrite commits with `git rebase`:** Rebasing changes the commit history, which can make the history look cleaner but also changes the commit IDs.
- **You push again with `git push --force`:** This action forces the new commit history onto the remote, overwriting the old history.
- **Collaborators must re-merge work:** Because the commit history has changed, your collaborators will have to adjust their work to fit the new history, which can be time-consuming and error-prone.

- **Solution**

- **Strict:** "*Do not ever rebase commits outside your repository*": This means you should avoid rebasing commits that have already been shared with others to prevent disrupting their work.
- **Loose:** "*Rebase your branch if only you use it, even if pushed to a server*": If you are the only one working on a branch, it's generally safe to rebase, even if the branch is on a remote server, because no one else is affected by the changes.

The images likely illustrate the difference between a clean commit history with rebasing and a

---

more complex one without it, emphasizing the importance of understanding when and how to use rebasing effectively.

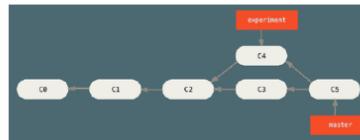
## 15 / 27: Rebase vs Merge: Philosophical Considerations

### Rebase vs Merge: Philosophical Considerations

- Deciding **Rebase-vs-merge** depends on the answer to the question:
  - What does the commit history of a repo mean?*

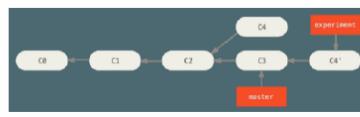
#### 1. History is the record of what actually happened

- "History should not be tampered with, even if messy!"*
- Use `git merge`



#### 2. History represents how a project should have been made

- "You should tell the history in the way that is best for future readers"*
- Use `git rebase` and `filter-branch`



15 / 27

#### • Rebase vs Merge: Philosophical Considerations

- When working with Git, a version control system, you often face the decision of whether to use **rebase** or **merge**. This decision is not just technical but also philosophical, as it relates to how you perceive the commit history of a repository.
- The key question to consider is: *What does the commit history of a repo mean?*

#### • History is the record of what actually happened

- This perspective values the authenticity of the commit history. It suggests that the history should reflect the actual sequence of events, even if it appears messy or complex.
- If you believe in preserving the true sequence of events, you should use `git merge`. This approach keeps all the original commits intact and shows how the project evolved over time, including all branches and merges.

#### • History represents how a project should have been made

- This viewpoint emphasizes clarity and readability for future developers. It suggests that the commit history should be organized in a way that makes it easy to understand the project's development process.
- If you prefer a clean and linear history, you should use `git rebase` and possibly `filter-branch`. This approach allows you to rewrite the commit history to present a more streamlined and coherent narrative, which can be beneficial for new team members or when reviewing the project's evolution.

#### • Conclusion

- The choice between rebase and merge is not just about technical differences but also about how you want to present the history of your project. Consider your team's needs and the importance of clarity versus authenticity when making this decision.

### Rebase vs Merge: Philosophical Considerations

- Many man-centuries have been wasted discussing rebase-vs-merge at the watercooler
  - Total waste of time! Tell people to get back to work!
- When you contribute to a project often people decide for you based on their preference
- **Best of the merge-vs-rebase approaches**
  - Rebase changes you've made in your local repo
    - Even if you have pushed but you know the branch is yours
    - Use `git pull --rebase` to clean up the history of your work
    - If the branch is shared with others then you need to definitively `git merge`
  - Only `git merge` to master to preserve the history of how something was built
- **Personally**
  - I like to squash-and-merge branches to `master`
  - Rarely my commits are "complete", are just checkpoints



16 / 27

- **Rebase vs Merge: Philosophical Considerations**
  - The debate between using *rebase* or *merge* in version control systems like Git is a common topic among developers. It's often discussed at length, sometimes humorously referred to as a waste of time, as it can distract from actual work. The key takeaway is that while these discussions can be engaging, they shouldn't overshadow productivity.
- **Project Contribution Decisions**
  - When contributing to a project, the choice between rebase and merge is often made by the project maintainers based on their preferences. This means that as a contributor, you might not always have the freedom to choose your preferred method.
- **Best of the merge-vs-rebase approaches**
  - **Rebase:** This is useful for cleaning up your local commit history. If you're working on a branch that is solely yours, you can rebase even after pushing. Using `git pull --rebase` helps keep your work history tidy.
  - **Merge:** When working on a shared branch, merging is safer to avoid conflicts. Merging into the master branch is recommended to maintain a clear history of how the project evolved.
- **Personally**
  - The speaker prefers to use a *squash-and-merge* approach when integrating branches into the master. This method combines all changes into a single commit, which is useful when individual commits are more like checkpoints rather than complete features. This approach helps keep the master branch clean and organized.

### Remote Branches

- **Remote branches** are pointers to branches in remote repos

```
git remote -v
origin  git@github.com:gpsaggesse/umd_classes.git (fetch)
origin  git@github.com:gpsaggesse/umd_classes.git (push)
```

- **Tracking branches**

- Local references representing the state of the remote repo
- E.g., `master` tracks `origin/master`
- You can't change the remote branch (e.g., `origin/master`)
- You can change tracking branch (e.g., `master`)
- Git updates tracking branches when you do `git fetch origin` (or `git pull`)

- To share code in a local branch you need to push it to a remote

```
> git push origin serverfix
```

- To work on it

```
> git checkout -b serverfix origin/serverfix
```

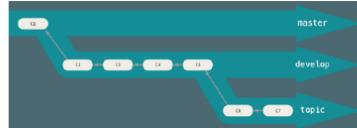


17 / 27

- **Remote branches** are essentially bookmarks that point to the branches in a remote repository. They help you keep track of the state of branches in a repository that is hosted somewhere else, like on GitHub. When you run the command `git remote -v`, it shows you the URLs of the remote repositories you have set up, both for fetching and pushing changes. This is crucial for collaboration, as it allows multiple people to work on the same project from different locations.
- **Tracking branches** are local branches that are set up to track the state of a branch in a remote repository. For example, your local `master` branch might track `origin/master`, which is the master branch on the remote repository. While you can't directly change a remote branch like `origin/master`, you can make changes to your local tracking branch, such as `master`. When you run `git fetch origin` or `git pull`, Git updates your tracking branches to reflect the latest state of the remote branches.
- To share your work from a local branch with others, you need to push it to a remote repository. For instance, if you have a local branch named `serverfix`, you can share it by executing `git push origin serverfix`. This command sends your local changes to the remote repository, making them accessible to others.
- If you want to start working on a branch that exists on the remote repository but not locally, you can create a new local branch that tracks the remote branch. For example, using `git checkout -b serverfix origin/serverfix` creates a new local branch named `serverfix` that tracks the `serverfix` branch on the remote repository. This allows you to work on the branch locally and later push your changes back to the remote.

## Git Workflows

- **Git workflows** = ways of working and collaborating using Git
- **Long-running branches** = branches at different level of stabilities, that are always open
  - master is always ready to be released
  - develop branch to develop in
  - topic / feature branches
  - When branches are “stable enough” they are merged up

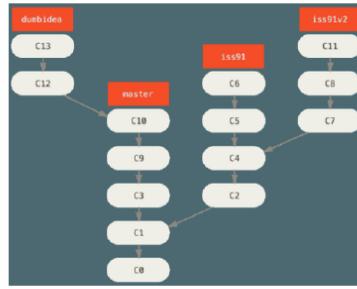


18 / 27

- **Git workflows:** These are structured methods for using Git, a version control system, to manage and collaborate on projects. Git workflows help teams organize their work, manage changes, and ensure that everyone is on the same page. They provide a framework for how code is developed, reviewed, and integrated into the main project.
- **Long-running branches:** These are branches in a Git repository that remain open for extended periods. They serve different purposes and have varying levels of stability:
  - **master branch:** This is the main branch that is always ready for release. It contains the most stable version of the project.
  - **develop branch:** This branch is used for ongoing development. It acts as an integration branch for features and fixes before they are considered stable enough for the **master**.
  - **Topic/feature branches:** These are short-lived branches created for specific features or fixes. Developers work on these branches independently and merge them into the **develop** branch once they are stable.
  - The process of merging branches ensures that only stable and tested code is integrated into the main branches, maintaining the project’s overall stability.

### Git Workflows

- **Topic branches** = short-lived branches for a single feature
  - E.g., hotfix, wip-XYZ
  - Easy to review
  - Silo-ed from the rest
  - This is typical of Git since other VCS support for branches is not good enough
  - E.g.,
    - You start iss91, then you cancel some stuff, and go to iss91v2
    - Somebody starts dumbidea branch and merge to master (!)
    - You squash-and-merge your iss91v2



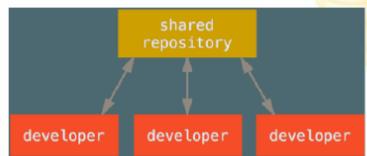
19 / 27

- **Topic branches** are a key concept in Git workflows. They are *short-lived branches* created specifically for working on a single feature or fix. This allows developers to work independently on different tasks without interfering with the main codebase.
  - For example, you might create a branch named `hotfix` to quickly address a bug or `wip-XYZ` for a work-in-progress feature. These branches are temporary and usually deleted after their changes are merged into the main branch.
  - **Easy to review:** Because topic branches are focused on a single feature or fix, they contain fewer changes, making it easier for others to review the code.
  - **Silo-ed from the rest:** These branches are isolated from the main codebase, reducing the risk of introducing errors into the main branch until the feature is complete and tested.
  - Git excels at handling branches compared to other version control systems (VCS), which may not support branching as effectively.
  - For instance, you might start working on a branch `iss91`, realize some changes are needed, and then create a new branch `iss91v2` to continue your work. Meanwhile, someone else might create a `dumbidea` branch and mistakenly merge it into `master`, highlighting the importance of careful branch management. Finally, you might use a *squash-and-merge* strategy to combine your `iss91v2` changes into the main branch, keeping the commit history clean.

## Centralized Workflow

- **Centralized workflow in centralized VCS**

- Developers:
  - Check out the code from the central repo on their computer
  - Modify the code locally
  - Push it back to the central hub (assuming no conflicts with latest copy, otherwise they need to merge)



- **Centralized workflow in Git**

- Developers:
  - Have push (i.e., write) access to the central repo
  - Need to fetch and then merge
  - Cannot push code that will overwrite each other code (only fast-forward changes)

- **Centralized workflow in centralized VCS**

- *Developers:*

- \* **Check out the code from the central repo on their computer:** In a centralized version control system (VCS), there is a single central repository where all the code is stored. Developers need to check out or download the code from this central location to their local machines to work on it.
    - \* **Modify the code locally:** Once the code is on their local machine, developers can make changes and improvements as needed.
    - \* **Push it back to the central hub:** After making changes, developers push their modified code back to the central repository. If someone else has made changes to the same part of the code, developers must resolve these conflicts before pushing.

- **Centralized workflow in Git**

- *Developers:*

- \* **Have push (i.e., write) access to the central repo:** In Git, developers need permission to push changes to the central repository. This ensures that only authorized users can update the main codebase.
    - \* **Need to fetch and then merge:** Before pushing changes, developers must fetch the latest version of the code from the central repository and merge it with their local changes. This helps prevent conflicts and ensures that the codebase is up-to-date.
    - \* **Cannot push code that will overwrite each other code (only fast-forward changes):** Git prevents developers from pushing changes that would overwrite others' work. Only changes that can be fast-forwarded, meaning they don't conflict with the current state of the repository, can be pushed directly. This helps maintain the integrity of the codebase.

### Forking Workflows

- Typically devs don't have permissions to update directly branches on a project
  - Read-write permissions for core contributors
  - Read-only for anybody else
- **Solution**
  - “Forking” a repo
  - External contributors
    - Clone the repo and create a branch with the work
    - Create a writable fork of the project
    - Push branches to fork
    - Prepare a PR with their work
  - Project maintainer
    - Reviews PRs
    - Accepts PRs
    - Integrates PRs
  - In practice it's the project maintainer that pulls the code when it's ready, instead of external contributors pushing the code
- **Aka “GitHub workflow”**
  - “Innovation” was forking (Fork me on GitHub!)
  - GitHub acquired by Microsoft for 7.5b USD



21 / 27

- **Forking Workflows**

- In many open-source projects, developers usually don't have the permission to directly update the main branches of a project. This is to maintain the integrity and stability of the codebase.
    - \* *Core contributors* have read-write permissions, meaning they can make changes directly to the project.
    - \* Everyone else has read-only access, which means they can view the code but cannot make changes directly.

- **Solution**

- The concept of “forking” a repository allows external contributors to work on a project without having direct write access.
    - \* External contributors can clone the repository and create their own branch to work on.
    - \* They create a writable fork, which is essentially a personal copy of the project where they can make changes.
    - \* Once their work is ready, they push their changes to their fork and prepare a Pull Request (PR) to propose their changes to the original project.
  - The project maintainer plays a crucial role in this workflow.
    - \* They review the PRs submitted by external contributors.
    - \* If the changes are satisfactory, they accept and integrate the PRs into the main project.
    - \* Typically, the maintainer pulls the code into the main project, rather than the contributor pushing it directly.

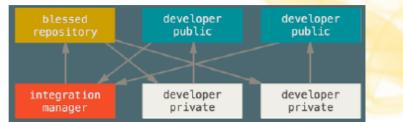
- **Aka “GitHub workflow”**

- 
- This workflow is often referred to as the “GitHub workflow” because GitHub popularized the concept of forking with their platform.
  - The ability to fork repositories and collaborate in this way was a significant innovation, leading to the phrase “Fork me on GitHub!”
  - GitHub’s success and influence in the software development community were highlighted by its acquisition by Microsoft for \$7.5 billion USD. This underscores the importance and value of collaborative coding platforms.

## 22 / 27: Integration-Manager Workflow

### Integration-Manager Workflow

- This is the classical model for open-source development
  - E.g., Linux, GitHub (forking) workflow



1. **One repo is the official project**
  - Only the project maintainer pushes to the public repo
  - E.g., causify-ai/csfy
2. **Each contributor**
  - Has read access to everyone else's public repo
  - Forks the project into a private copy
    - Write access to their own public repo
    - E.g., gpsaggese/csfy
  - Makes changes
  - Pushes changes to his own public copy
  - Sends email to maintainer asking to pull changes (pull request)
3. **The maintainer**
  - Adds contributor repo as a remote
  - Merges the changes into a local branch
  - Tests changes locally
  - Pushes branch to the official repo



22 / 27

#### • Integration-Manager Workflow

This workflow is a traditional model used in open-source development. It's commonly seen in projects like Linux and on platforms like GitHub, where the concept of forking is prevalent. This model helps manage contributions from multiple developers efficiently.

#### • One repo is the official project

In this workflow, there is a single official repository for the project. Only the project maintainer has the authority to push changes directly to this public repository. For example, in a project named `causify-ai/csfy`, the maintainer would be the only one updating the official repo.

#### • Each contributor

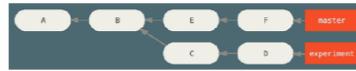
Contributors have read access to all public repositories, allowing them to see what others are working on. They fork the official project, creating their own private copy where they have write access. For instance, a contributor might fork the project to `gpsaggese/csfy`. They make changes in their forked copy and push these changes to their public repo. Once satisfied, they send a pull request to the maintainer, asking for their changes to be incorporated into the official project.

#### • The maintainer

The maintainer plays a crucial role in this workflow. They add the contributor's repository as a remote to their local setup. After merging the proposed changes into a local branch, they test these changes to ensure they work correctly. Once verified, the maintainer pushes the updated branch to the official repository, integrating the contributor's work into the project. This process ensures that all changes are reviewed and tested before becoming part of the official codebase.

### Git log

- `git log` reports info about commits
- **refs** are references to:
  - HEAD (commit you are working on, next commit)
  - origin/master (remote branch)
  - experiment (local branch)
  - d921970 (commit)
- ^ after a reference resolves to the parent of that commit
  - `HEAD^` = commit before HEAD, i.e., last commit
  - `^2` means `^`
  - A merge commit has multiple parents



23 / 27

- **Git log**
  - The `git log` command is a tool used in Git to display a history of commits. This is useful for tracking changes and understanding the evolution of a project over time. Each commit in the log contains information such as the author, date, and a message describing the changes made.
- **Refs (References)**
  - **Refs** are pointers to specific commits or branches in a Git repository. They help you navigate through different points in the project's history.
  - **HEAD**: This is a special reference that points to the current commit you are working on. It represents the latest state of your working directory and is where your next commit will be based.
  - **origin/master**: This refers to the remote branch named **master**. It is the version of the branch stored on a remote server, often used to collaborate with others.
  - **experiment**: This is an example of a local branch. Local branches are used to work on different features or fixes without affecting the main codebase.
  - **d921970**: This is an example of a specific commit identified by its unique hash. Each commit in Git has a unique identifier, which allows you to reference it directly.
- **Caret (^) Notation**
  - The caret symbol (^) is used to navigate through commit history. It helps you identify parent commits.
  - `HEAD^` refers to the commit immediately before the current `HEAD`. This is useful for looking at the previous state of the project.
  - `^2` is shorthand for `^`, which means the second parent of a commit. This is particularly relevant for merge commits, which have multiple parent commits due to the merging of

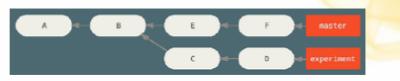
---

branches.

### Dot notation

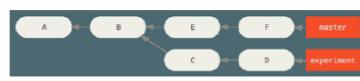
- **Double-dot notation**

- $1..2$  = commits that are reachable from 2 but not from 1
- Like a “difference”
- `git log master..experiment` → D, C
- `git log experiment..master` → F, E



- **Triple-dot notation**

- $1...2$  = commits that are reachable from either branch but not from both
- Like “union excluding intersection”
- `git log master...experiment` → F, E, D, C



24 / 27

- **Double-dot notation**

- The double-dot notation is a way to compare two branches or commits in Git. When you see something like  $1..2$ , it means you’re looking at the commits that are reachable from the second commit (or branch) but not from the first. Think of it like finding the *difference* between two sets.
- For example, if you run `git log master..experiment`, Git will show you the commits that are in the `experiment` branch but not in the `master` branch. In this case, it would show commits D and C.
- Conversely, `git log experiment..master` will show you the commits that are in `master` but not in `experiment`, which are F and E.

- **Triple-dot notation**

- The triple-dot notation is slightly different. When you see  $1...2$ , it refers to the commits that are reachable from either of the two branches but not from both. It’s like taking the *union* of the two sets and then excluding the *intersection*.
- For instance, `git log master...experiment` will list all the commits that are unique to either `master` or `experiment`, but not those that are common to both. In this example, it would show commits F, E, D, and C.

### Advanced Git

- **Stashing**
  - Copy state of your working dir (e.g., modified and staged files)
  - Save it in a stack
  - Apply later
- **Cherry-picking**
  - Apply a single commit from one branch onto another
- **rerere**
  - = “Reuse Recorded Resolution”
  - Git caches how to solve certain conflicts
- **Submodules / subtrees**
  - Project including other Git projects



25 / 27

- **Advanced Git**
- **Stashing**
  - *Stashing* is a handy feature in Git that allows you to temporarily save changes in your working directory. This includes both modified and staged files. Imagine you’re in the middle of working on a feature, but suddenly need to switch to another branch to fix a bug. Instead of committing incomplete work, you can stash it. This saves your changes in a stack-like structure, allowing you to apply them later when you’re ready to continue.
- **Cherry-picking**
  - *Cherry-picking* is a useful tool when you want to apply a specific commit from one branch to another. This is particularly helpful when you have a commit with a bug fix or a feature that you want to include in another branch without merging the entire branch. It allows for precise control over what changes are incorporated into your branch.
- **rerere**
  - The term *rerere* stands for “Reuse Recorded Resolution.” This feature in Git helps you manage merge conflicts more efficiently. When you resolve a conflict, Git can remember how you resolved it. If the same conflict arises again, Git can automatically apply the previous resolution, saving you time and effort.
- **Submodules / subtrees**
  - These are methods for including one Git project within another. *Submodules* allow you to keep a Git repository as a subdirectory of another Git repository. This is useful

---

for managing dependencies. *Subtrees* offer a more integrated approach, allowing you to merge and split repositories more seamlessly. Both methods help manage complex projects that rely on multiple repositories.

### Advanced Git

- **bisect**
  - `git bisect` helps identifying the commit that introduced a bug
    - Bug appears at top of tree
    - Unknown revision where it started
    - Script returns 0 if good, non-0 if bad
    - `git bisect` finds revision where script changes from good to bad
- **filter-branch**
  - Rewrite repo history in a script-able way
    - E.g., change email, remove sensitive file
  - Check out each version, run command, commit result
- **Hooks**
  - Run scripts before commit, merging,



26 / 27

- **bisect**
  - *Purpose:* `git bisect` is a powerful tool used to pinpoint the exact commit that introduced a bug in your codebase.
    - \* Imagine you have a bug in your project, and you know it exists in the latest version, but you're not sure when it first appeared. `git bisect` helps you find that specific commit.
    - \* You start by marking the current version as “bad” (where the bug is present) and a previous version as “good” (where the bug was absent).
    - \* You can use a script that returns 0 if the code is good and a non-zero value if it’s bad. This script automates the process of checking each commit.
    - \* `git bisect` will then automatically check out different commits and run your script to determine where the code changes from good to bad, effectively narrowing down the problematic commit.
- **filter-branch**
  - *Purpose:* This command allows you to rewrite the history of your repository in a programmable way.
    - \* It’s useful for making changes across many commits, such as altering author information or removing sensitive data from the history.
    - \* The process involves checking out each commit, applying your changes (like running a command), and then committing the result back into the history.
    - \* This is a powerful tool but should be used with caution, as rewriting history can affect collaborators and shared repositories.
- **Hooks**
  - *Purpose:* Hooks are scripts that Git can run automatically at certain points in your

---

workflow.

- \* They can be set to run before actions like committing or merging, allowing you to enforce rules or automate tasks.
- \* For example, you might use a pre-commit hook to check code style or run tests before allowing a commit to proceed.
- \* Hooks help maintain code quality and consistency across a team by automating checks and processes.

## GitHub

- GitHub acquired by MSFT for 7.5b
- **GitHub: largest host for Git repos**
  - Git hosting (100m+ open source projects)
  - PRs, forks
  - Issue tracking
  - Code review
  - Collaboration
  - Wiki
  - Actions (CI / CD)
- **“Forking a project”**
  - Open-source communities
    - Negative connotation
    - Modify and create a competing project
  - GitHub parlance
    - Copy a project to contribute without push/write access



27 / 27

- **GitHub acquired by MSFT for 7.5b**
  - In 2018, Microsoft acquired GitHub for \$7.5 billion. This acquisition was significant because GitHub is a major platform for developers, and Microsoft's purchase indicated their commitment to supporting open-source software and developer communities.
- **GitHub: largest host for Git repos**
  - GitHub is the largest platform for hosting Git repositories, with over 100 million open-source projects. It provides a space for developers to store and manage their code.
  - **Git hosting:** GitHub allows developers to host their code repositories, making it easier to manage and share code.
  - **PRs, forks:** Pull requests (PRs) and forks are essential features for collaboration, allowing developers to propose changes and work on copies of projects.
  - **Issue tracking:** GitHub offers tools to track bugs and feature requests, helping teams manage their projects efficiently.
  - **Code review:** Developers can review each other's code, ensuring quality and consistency.
  - **Collaboration:** GitHub fosters teamwork by providing tools for developers to work together on projects.
  - **Wiki:** Each project can have its own wiki, which is useful for documentation and sharing information.
  - **Actions (CI / CD):** GitHub Actions enable continuous integration and continuous deployment (CI/CD), automating the testing and deployment of code.
- **“Forking a project”**
  - In open-source communities, *forking* can have a negative connotation, as it sometimes means creating a competing project by modifying an existing one.

- 
- In GitHub parlance, forking is a positive action where a developer copies a project to contribute to it, especially when they don’t have push or write access to the original repository. This allows for collaboration and improvement of the project.

---

## Lesson 2.2: Data Pipelines



## Lesson 2.2: Data Pipelines

**Instructor:** Dr. GP Saggese, [gsaggese@umd.edu](mailto:gsaggese@umd.edu)

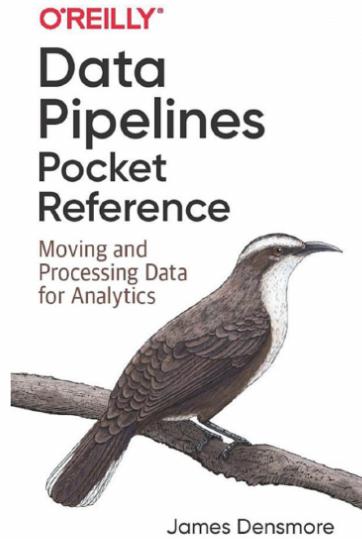


---

## 2 / 21: Data Pipelines: Resources

### Data Pipelines: Resources

- Concepts in the slides
- Class project
- Mastery
  - [Data Pipelines Pocket Reference](#)



2 / 21

- **Concepts in the slides**

- This bullet point suggests that the slides themselves contain important concepts related to data pipelines. Data pipelines are essential in managing and processing large volumes of data efficiently. They automate the flow of data from one system to another, ensuring that data is clean, organized, and ready for analysis. Understanding these concepts is crucial for anyone working with big data or machine learning, as they form the backbone of data-driven decision-making.

- **Class project**

- The mention of a class project indicates that students will have a practical opportunity to apply what they've learned about data pipelines. This hands-on experience is invaluable because it allows students to see how theoretical concepts are implemented in real-world scenarios. Working on a project helps solidify understanding and provides a chance to troubleshoot and solve problems that may arise in data pipeline creation and management.

- **Mastery**

- The reference to the *Data Pipelines Pocket Reference* book suggests a resource for students to deepen their understanding and achieve mastery in the subject. This book likely offers detailed explanations, examples, and best practices for building and managing data pipelines. It's a useful tool for students who want to go beyond the basics and gain a comprehensive understanding of the topic.

### Data as a Product

- **Many services today "sell" data**
  - Services are typically powered by data and machine learning, e.g.,
    - Personalized search engine (Google)
    - Sentiment analysis on user-generated data (Facebook)
    - E-commerce + recommendation engine (Amazon)
    - Streaming data (Netflix, Spotify)
- **Several steps are required to generate data products**
  - Data ingestion
  - Data pre-processing
    - Cleaning, tokenization, feature computation
  - Model training
  - Model deployment
    - MLOps
  - Model monitoring
    - Is model working?
    - Is model getting slower?
    - Are model performance getting worse?
  - Collect feedback from deployment
    - E.g., recommendations vs what users bought
    - Ingest data from production for future versions of the model



3 / 21

- 
- **Model deployment:** Once trained, models need to be deployed into a production environment where they can be used in real-time applications. This involves MLOps, which is the practice of managing the lifecycle of machine learning models.
  - **Model monitoring:** After deployment, it's important to continuously monitor the model to ensure it is performing well. This includes checking if the model is functioning correctly, if it's slowing down, or if its performance is degrading over time.
  - **Collect feedback from deployment:** Gathering feedback is essential for improving models. For example, comparing recommendations with actual user purchases helps refine future versions of the model. This feedback loop involves ingesting new data from the production environment to enhance the model's accuracy and relevance.

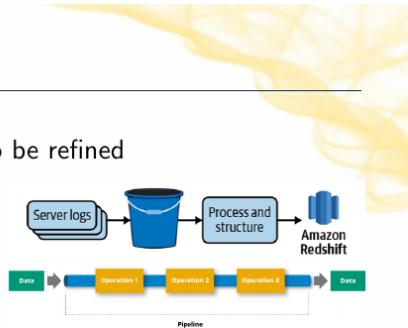
## 4 / 21: Data Pipelines

### Data Pipelines

- “Data is the new oil” . . . but oil needs to be refined

- **Data pipelines**

- Processes that move and transform data
- **Goal:** derive new value from data through analytics, reporting, machine learning



- **Data needs to be:**

- Collected
- Pre-processed / cleaned
- Validated
- Processed
- Combined

- **Data ingestion**

- Simplest data pipeline
- Extract data (e.g., from REST API)
- Load data into DB (e.g., SQL table)



4 / 21

- **Data Pipelines**

- Think of data pipelines as the systems that help us move and change data from one place to another. Just like oil needs to be refined to be useful, data needs to be processed to be valuable.
- **Goal:** The main aim of data pipelines is to help us get new insights from data. This can be through analytics, creating reports, or using machine learning to make predictions or decisions.

- **Data needs to be:**

- **Collected:** Gathering data from various sources, like sensors, databases, or user inputs.
- **Pre-processed / cleaned:** Removing errors or inconsistencies in the data to make sure it's accurate and ready for analysis.
- **Validated:** Checking that the data is correct and meets the required standards.
- **Processed:** Transforming the data into a format that is useful for analysis or reporting.
- **Combined:** Merging data from different sources to provide a complete picture.

- **Data Ingestion**

- This is the simplest form of a data pipeline. It involves taking data from a source, like a REST API, and putting it into a database, such as a SQL table. This is the first step in making data ready for further processing and analysis.

## 5 / 21: Roles in Building Data Pipelines

### Roles in Building Data Pipelines

- **Data engineers**

- Build and maintain data pipelines
- Tools:
  - Python / Java / Go / No-code
  - SQL / NoSQL stores
  - Hadoop / MapReduce / Spark
  - Cloud computing

- **Data scientists**

- Build predictive models
- Tools:
  - Python / R / Julia
  - Hadoop / MapReduce / Spark
  - Cloud computing

- **Data analysts**

- E.g., marketing, MBAs, sales, ...
- Build metrics and dashboards
- Tools:
  - Excel spreadsheets
  - GUI tools (e.g., Tableaux)
  - Desktop



5 / 21

- **Data engineers**

- *Role:* Data engineers are responsible for creating and maintaining the systems that allow data to be collected, stored, and processed efficiently. They ensure that data pipelines are robust and scalable, which is crucial for handling large volumes of data.

- *Tools:*

- \* **Programming Languages:** They often use languages like Python, Java, and Go for scripting and automation. These languages are versatile and widely used in the industry.
- \* **Data Storage:** SQL and NoSQL databases are essential for storing structured and unstructured data, respectively.
- \* **Big Data Frameworks:** Hadoop, MapReduce, and Spark are popular for processing large datasets. These frameworks allow for distributed computing, which is necessary for handling big data.
- \* **Cloud Computing:** Cloud platforms provide scalable resources and services, making it easier to manage data infrastructure without the need for physical hardware.

- **Data scientists**

- *Role:* Data scientists focus on analyzing data and building models that can predict future trends or behaviors. Their work often involves statistical analysis and machine learning.

- *Tools:*

- \* **Programming Languages:** Python, R, and Julia are commonly used for data analysis and modeling due to their rich libraries and ease of use.
- \* **Big Data Frameworks:** Like data engineers, data scientists also use Hadoop, MapReduce, and Spark to handle large datasets.
- \* **Cloud Computing:** Cloud services provide the computational power needed for

---

complex data analysis and model training.

- **Data analysts**

- *Role:* Data analysts interpret data and create reports that help businesses make informed decisions. They often work closely with business teams to understand their data needs.
- *Tools:*
  - \* **Excel:** A staple tool for data analysis, Excel is used for its simplicity and powerful data manipulation capabilities.
  - \* **GUI Tools:** Tools like Tableau allow analysts to create interactive dashboards and visualizations, making it easier to communicate insights.
  - \* **Desktop:** Many data analysis tasks are performed on desktop computers, using software that provides a user-friendly interface for data exploration.

---

## 6 / 21: Practical problems in Data Pipeline Organization

### Practical problems in Data Pipeline Organization

- Who is responsible for the data?
- Issues with scaling
  - Performance
  - Memory
  - Disk
- Build-vs-buy
  - Which tools?
  - Open-source vs proprietary?
- Architecture
- Service level agreement (SLA)
- Talk to stakeholders on a regular basis



6 / 21

- Who is responsible for the data?
  - It's crucial to identify who owns the data within an organization. This person or team is responsible for ensuring data quality, security, and compliance. Without clear ownership, data management can become chaotic, leading to errors and inefficiencies.
- Issues with scaling
  - *Performance*: As data volume grows, systems may slow down. It's important to optimize processes to maintain speed.
  - *Memory*: Large datasets require more memory. Efficient data handling and processing techniques are necessary to prevent system overloads.
  - *Disk*: Storage capacity can become a bottleneck. Planning for scalable storage solutions is essential to accommodate growing data needs.
- Build-vs-buy
  - *Which tools?*: Deciding on the right tools is critical. Consider factors like ease of use, integration capabilities, and community support.
  - *Open-source vs proprietary?*: Open-source tools are often cost-effective and flexible, while proprietary solutions may offer better support and features. Weigh the pros and cons based on your organization's needs.
- Architecture
  - *Who is in charge of it?*: Assigning a person or team to oversee the architecture ensures consistency and alignment with business goals.
  - *Conventions*: Establishing standards and best practices helps maintain a coherent and efficient data pipeline.
  - *Documentation*: Proper documentation is vital for understanding and maintaining the data pipeline, especially as team members change.

- 
- **Service level agreement (SLA)**
    - SLAs define the expected performance and availability of data services. They help set clear expectations and accountability between data providers and users.
  - **Talk to stakeholders on a regular basis**
    - Regular communication with stakeholders ensures that the data pipeline aligns with business objectives and adapts to changing needs. It also helps in identifying potential issues early and fosters collaboration.

### Data Ingestion

- **Data ingestion**
  - = extract data from one source and load it into another store
- **Data sources / sinks**
  - DBs
    - E.g., Postgres, MongoDB
  - REST API
    - Abstraction layer on top of DBs
  - Network file system / cloud
    - E.g., CSV files, Parquet files
  - Data warehouses
  - Data lakes
- **Source ownership**
  - An organization can use 10-1000s of data sources
  - Internal
    - E.g., DB storing shopping carts for a e-commerce site
  - 3rd-parties
    - E.g., Google analytics tracking website usage



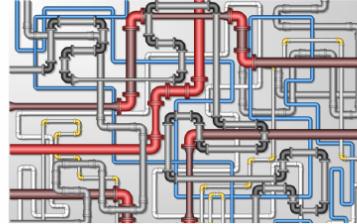
7 / 21

- **Data ingestion**
  - *Data ingestion* is the process of taking data from one place and moving it to another. This is a crucial step in data processing because it allows data to be collected from various sources and stored in a centralized location where it can be analyzed or used for other purposes.
- **Data sources / sinks**
  - **DBs:** Databases like Postgres and MongoDB are common sources and destinations for data. They store structured data and are often used in applications that require quick access to data.
  - **REST API:** This is a way to access data over the internet. It acts as a bridge between databases and applications, allowing data to be retrieved or sent using HTTP requests.
  - **Network file system / cloud:** Data can also be stored in files, such as CSV or Parquet files, which can be located on a network or in the cloud. These files are often used for large datasets that need to be processed in bulk.
  - **Data warehouses:** These are large storage systems designed to hold vast amounts of data from different sources, optimized for query and analysis.
  - **Data lakes:** Unlike data warehouses, data lakes store raw data in its native format, which can include structured, semi-structured, and unstructured data.
- **Source ownership**
  - Organizations often deal with a wide range of data sources, sometimes numbering in the thousands. These sources can be internal, such as a database that keeps track of shopping carts for an e-commerce site, or external, like third-party services such as Google Analytics, which provides insights into website usage. Understanding who owns the data source is important for managing access and ensuring data quality.

## 8 / 21: Data Pipeline Paradigms

### Data Pipeline Paradigms

- There are **several styles of building data pipelines**
- **Multiple phases**
  - Extract
  - Load
  - Transform
- **Phases arranged in different ways**  
depending on philosophy about data / roles
  - ETL
  - ELT
  - EtLT



8 / 21

- **Several styles of building data pipelines:** In the world of data processing, a data pipeline is a series of steps that data goes through from its raw form to a usable format. There are different ways to design these pipelines, and each style has its own advantages and use cases. Understanding these styles helps in choosing the right approach for specific data needs.
- **Multiple phases:** Data pipelines typically involve three main phases:
  - **Extract:** This is the first step where data is collected from various sources. These sources can be databases, APIs, or even files.
  - **Load:** After extraction, the data is loaded into a storage system. This could be a data warehouse, a data lake, or any other storage solution.
  - **Transform:** In this phase, the data is cleaned, organized, and converted into a format that is suitable for analysis or further processing.
- **Phases arranged in different ways:** The order and emphasis of these phases can vary based on the philosophy or the roles involved in the data processing:
  - **ETL (Extract, Transform, Load):** This traditional approach involves transforming data before loading it into the storage system. It's useful when data needs to be cleaned and structured before storage.
  - **ELT (Extract, Load, Transform):** In this modern approach, data is loaded in its raw form and transformed later. This is beneficial when dealing with large volumes of data, as it allows for more flexible and scalable processing.
  - **EtLT (Extract, then Load and Transform):** This is a hybrid approach where some transformation occurs before loading, and additional transformations happen afterward. It combines the benefits of both ETL and ELT, providing a balanced approach.

---

## 9 / 21: ETL Paradigm: Phases

### ETL Paradigm: Phases

- **Extract**

- Gather data from various data sources, e.g.,
  - Internal / external data warehouse
  - REST API
  - Data downloading from API
  - Web scraping

- **Transform**

- Raw data is combined and formatted to become useful for analysis step

- **Load**

- Move data into the final destination, e.g.,
  - Data warehouse
  - Data lake

- **Data ingestion pipeline = E + L**

- Move data from one point to another
- Format the data
- Make a copy
- Have different tools to operate on the data



9 / 21

- **Extract**

- The first phase of the ETL process involves gathering data from various sources. This is crucial because data can reside in multiple places, such as internal or external data warehouses, which are large storage systems for data. Additionally, data can be collected from REST APIs, which are interfaces that allow different software applications to communicate with each other. Data can also be downloaded directly from APIs or collected through web scraping, which involves extracting data from websites. The goal here is to gather all the necessary data that will be used in later stages.

- **Transform**

- Once the data is extracted, it needs to be transformed. This means taking the raw data and combining it in a way that makes it useful for analysis. This step often involves cleaning the data, removing duplicates, and converting it into a format that can be easily analyzed. The transformation phase is essential because raw data is often messy and not immediately useful for making decisions or gaining insights.

- **Load**

- After transforming the data, the next step is to load it into its final destination. This could be a data warehouse, which is used for storing large amounts of structured data, or a data lake, which can store both structured and unstructured data. The loading phase ensures that the data is stored in a place where it can be accessed and analyzed by data analysts and other stakeholders.

- **Data ingestion pipeline = E + L**

- The data ingestion pipeline focuses on the extraction and loading phases of the ETL process. It involves moving data from one point to another, ensuring that it is formatted correctly, and often involves making a copy of the data. Different tools are used to

---

operate on the data during this process, ensuring that it is efficiently and accurately transferred to its destination. This pipeline is crucial for maintaining a steady flow of data into systems where it can be used for analysis and decision-making.

---

## 10 / 21: ETL Paradigm: Example

### ETL Paradigm: Example

- **Extract**
  - Buy-vs-build data ingestion tools
    - Vendor lock-in
- **Transform**
  - Data conversion (e.g., parsing timestamp)
  - Create new columns from multiple source columns
    - E.g., year, month, day → yyyy/mm/dd
  - Aggregate / filter through business logic
    - Try not to filter, better to add tags / mark data
  - Anonymize data
- **Load**
  - Organize data in a format optimized for data analysis
    - E.g., load data in relational DB
  - Finally data modeling



10 / 21

- **ETL Paradigm: Example**

- **Extract**

- \* This step involves gathering data from various sources. You can either buy or build your own data ingestion tools. *Buying* tools can be quicker and easier but may lead to **vendor lock-in**, meaning you become dependent on a specific vendor's technology and may face challenges if you want to switch tools later.

- **Transform**

- \* This is where the raw data is cleaned and converted into a usable format. For example, you might need to convert timestamps into a standard format.
    - \* You can also create new columns by combining data from multiple sources, such as converting separate year, month, and day columns into a single yyyy/mm/dd format.
    - \* Aggregating or filtering data is done based on business logic. However, it's often better to tag or mark data instead of filtering it out, as this retains more information for future analysis.
    - \* Anonymizing data is crucial for privacy, ensuring that personal information is protected during analysis.

- **Load**

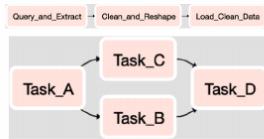
- \* The final step is to load the transformed data into a system where it can be easily accessed and analyzed. This often involves organizing the data in a format optimized for analysis, such as a relational database.
    - \* After loading, data modeling can be performed to structure the data in a way that supports business needs and decision-making processes.

---

## 11 / 21: Workflow Orchestration

### Workflow Orchestration

- Companies have **many data pipelines** (10-1000s)
- Orchestration tools**, e.g.,
  - Apache Airflow (from AirBnB)
  - Luigi (from Spotify)
  - AWS Glue
  - Kubeflow
- Schedule and manage** flow of tasks according to their dependencies
  - Pipeline and jobs are represented through DAGs
- Monitor, retry, and send alarms



11 / 21

- Workflow Orchestration** is crucial for managing complex data processes within companies. Many organizations have **many data pipelines**—ranging from tens to thousands—that need to be efficiently managed and executed. These pipelines are sequences of data processing steps that transform raw data into valuable insights.
- Orchestration tools** are software solutions that help automate, schedule, and manage these data pipelines. Some popular tools include:
  - Apache Airflow**: Developed by AirBnB, it's widely used for its flexibility and scalability.
  - Luigi**: Created by Spotify, it's known for its simplicity and ease of use.
  - AWS Glue**: A fully managed service by Amazon Web Services that simplifies the process of building data pipelines.
  - Kubeflow**: Designed for Kubernetes, it focuses on machine learning workflows.
- These tools help **schedule and manage** the flow of tasks by understanding their dependencies. They use Directed Acyclic Graphs (DAGs) to represent pipelines and jobs, ensuring tasks are executed in the correct order.
- Additionally, orchestration tools can **monitor** the execution of tasks, **retry** failed tasks, and **send alarms** to notify users of any issues. This ensures that data processes run smoothly and any problems are quickly addressed.

## 12 / 21: ELT Paradigm

### ELT Paradigm

- **ETL** has been the standard approach for long time

- Extract → Transform → Load
- **Cons**

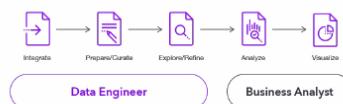
- Need to understand the data at ingestion time
- Need to know how the data will be used

- Today **ELT** is becoming the pattern of choice

- Extract → Load → Transform

- **Pro:**

- No need to know how the data will be used
- Separate data engineers and data scientists / analysts
- Data engineers focus on data ingestion (E + L)
- Data scientists focus on transform (T)



- ETL → ELT **enabled by new technologies**

- Large storage to save all the raw data (cloud computing)
- Distributed data storage and querying (e.g., HDFS)



12 / 21

- **ETL** has been the standard approach for a long time

- *Extract → Transform → Load:* This traditional method involves extracting data from various sources, transforming it into a suitable format, and then loading it into a data warehouse or database.

- **Cons:**

- \* *Need to understand the data at ingestion time:* This means that before you even start processing the data, you need to have a clear understanding of its structure and how it will be used, which can be limiting.
- \* *Need to know how the data will be used:* This requires predicting future data needs, which can be challenging and may lead to inefficiencies if the data usage changes.

- Today **ELT** is becoming the pattern of choice

- *Extract → Load → Transform:* This modern approach involves loading raw data into a storage system first and then transforming it as needed.

- **Pro:**

- \* *No need to know how the data will be used:* This flexibility allows for data to be stored in its raw form and transformed later based on actual needs.
- \* *Separate data engineers and data scientists/analysts:* This division of labor allows data engineers to focus on the technical aspects of data ingestion (E + L), while data scientists can concentrate on analyzing and transforming the data (T).

- ETL → ELT **enabled by new technologies**

- *Large storage to save all the raw data (cloud computing):* Cloud services provide scalable storage solutions that make it feasible to store vast amounts of raw data.

- *Distributed data storage and querying (e.g., HDFS):* Technologies like Hadoop Distributed File System (HDFS) allow for efficient storage and retrieval of large datasets

- 
- across multiple machines.
  - *Columnar DBs*: These databases store data in columns rather than rows, optimizing them for read-heavy operations typical in analytics.
  - *Data compression*: This reduces the storage footprint and speeds up data processing by minimizing the amount of data that needs to be read from disk.

---

## 13 / 21: Row-based vs Columnar DBs

### Row-based vs Columnar DBs

- **Row-based DBs**

- E.g., MySQL, Postgres
- Optimized for reading / writing rows
- Read / write small amounts of data frequently

OrderId	CustomerId	ShippingCountry	OrderTotal
1	1258	US	55.25
2	5698	AUS	125.36
3	2265	US	776.95
4	8954	CA	32.16

- **Columnar DBs**

- E.g., Amazon Redshift, Snowflake
- Read / write large amounts of data infrequently
- Analytics requires a few columns
- Better data compression

Block 1	1, 1258, US, 55.25
Block 2	2, 5698, AUS, 125.36
Block 3	3, 2265, US, 776.95
Block 4	4, 8954, CA, 32.16



13 / 21

- **Row-based DBs**

- *Examples include* MySQL and Postgres. These are traditional databases that store data in rows. This means that all the data for a single record is stored together.
- They are *optimized for reading and writing rows*, which makes them ideal for applications where you need to access complete records frequently, such as transactional systems.
- These databases are best suited for scenarios where you need to read or write small amounts of data frequently. This is because accessing a full row is efficient, but reading specific columns from many rows can be slower.

- **Columnar DBs**

- *Examples include* Amazon Redshift and Snowflake. These databases store data in columns rather than rows, which means that all the data for a single column is stored together.
- They are designed to *read and write large amounts of data infrequently*, making them ideal for analytical queries where you need to process large datasets.
- In analytics, you often need only a few columns from a large dataset. Columnar databases excel in these scenarios because they can quickly access the needed columns without reading entire rows.
- They also offer *better data compression* because similar data types are stored together, which can significantly reduce storage requirements and improve query performance.

## EtLT

- **ETL**
  - Extract → Transform → Load
- **ELT**
  - Extract → Load → Transform
  - Transformation / data modeling ("T") according to business logic
- **EtLT**
  - Sometimes transformations with limited scope ("t") are needed
    - De-duplicate records
    - Parse URLs into individual components
    - Obfuscate sensitive data (for legal or security reasons)
  - Then implement rest of "LT" pipeline



14 / 21

- **ETL**
  - **Extract → Transform → Load:** This is a traditional data processing approach where data is first extracted from various sources. After extraction, the data is transformed into a suitable format or structure, often involving cleaning, aggregating, or enriching the data. Finally, the transformed data is loaded into a target system, such as a data warehouse, where it can be used for analysis or reporting.
- **ELT**
  - **Extract → Load → Transform:** In this approach, data is extracted and immediately loaded into a storage system, like a data lake or cloud storage. The transformation occurs after loading, allowing for more flexibility and scalability. This method is particularly useful when dealing with large volumes of data, as it leverages the processing power of modern storage systems to perform transformations according to specific business logic.
- **EtLT**
  - **Sometimes transformations with limited scope ("t") are needed:** In some cases, minor transformations are necessary before loading data. These transformations might include:
    - \* **De-duplicate records:** Removing duplicate entries to ensure data quality.
    - \* **Parse URLs into individual components:** Breaking down URLs into parts like protocol, domain, and path for easier analysis.
    - \* **Obfuscate sensitive data:** Masking or encrypting sensitive information to comply with legal or security requirements.
  - **Then implement rest of "LT" pipeline:** After these initial transformations, the data is loaded, and further transformations are applied as needed, following the ELT process. This hybrid approach balances the need for immediate data cleaning with the flexibility

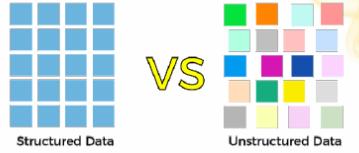
---

of post-load transformations.

## 15 / 21: Structure in Data (or Lack Thereof)

### Structure in Data (or Lack Thereof)

- **Structured data:** there is a schema
  - Relational DB
  - CSV
  - DataFrame
  - Parquet
- **Semi-structured:** subsets of data have different schema
  - Logs
  - HTML pages
  - XML
  - Nested JSON
  - NoSQL data
- **Unstructured:** no schema
  - Text
  - Pictures
  - Movies
  - Blobs of data



15 / 21

- **Structured data:** This type of data is organized in a highly predictable way, often following a specific schema or format.
  - *Relational DB:* Databases that store data in tables with rows and columns, like a spreadsheet. Each table has a defined structure.
  - *CSV:* Stands for Comma-Separated Values, a simple file format used to store tabular data, where each line is a data record.
  - *DataFrame:* A data structure used in programming languages like Python (Pandas) and R, which organizes data into a 2D table.
  - *Parquet:* A columnar storage file format optimized for use with big data processing frameworks like Apache Hadoop and Apache Spark.
- **Semi-structured:** This data type doesn't fit neatly into tables but still has some organizational properties.
  - *Logs:* Records of events or transactions, often with a timestamp and other metadata, but not always in a uniform format.
  - *HTML pages:* Web pages that have a structured format using tags, but the content within can vary greatly.
  - *XML:* A markup language that defines rules for encoding documents in a format that is both human-readable and machine-readable.
  - *Nested JSON:* A lightweight data interchange format that is easy for humans to read and write, and for machines to parse and generate, often with nested structures.
  - *NoSQL data:* Databases that store data in a format other than tabular relations, often used for large sets of distributed data.
- **Unstructured:** This data lacks a predefined format or organization, making it more challenging to process and analyze.

- 
- *Text*: Free-form text data, such as emails, social media posts, or articles.
  - *Pictures*: Image files that contain visual data without a specific structure.
  - *Movies*: Video files that include a sequence of images and sound, lacking a structured format.
  - *Blobs of data*: Binary Large Objects, which can be any type of data stored as a single entity in a database, often unstructured.

This slide highlights the different types of data structures, emphasizing the importance of understanding the format and organization of data when working with machine learning and big data. Each type requires different tools and techniques for processing and analysis.

### Data Cleaning

- **Data cleanliness**

- Quality of source data varies greatly
- Data is typically messy
  - Duplicated records
  - Incomplete or missing records (nans)
  - Inconsistent formats (e.g., phone with / without dashes)
  - Mislabeled or unlabeled data



- **When to clean it?**

- As soon as possible!
- As late as possible!
- In different stages
- → Pipeline style: ETL vs ELT vs EtLT

- **Heuristics when dealing with data**

- Hope for the best, assume the worst
- Validate data early and often
- Don't trust anything
- Be defensive



16 / 21

- **Data cleanliness**

- The quality of data you receive can vary widely. Some datasets might be well-organized, while others can be quite messy. This messiness can manifest in several ways:

- \* **Duplicated records:** Sometimes, the same data entry appears multiple times, which can skew analysis results.
- \* **Incomplete or missing records:** Often, datasets have gaps where information is missing, represented as 'nans' (not-a-number).
- \* **Inconsistent formats:** Data might be recorded in different formats, such as phone numbers with or without dashes, making it hard to standardize.
- \* **Mislabeled or unlabeled data:** Data might be incorrectly labeled or lack labels entirely, complicating analysis.

- **When to clean it?**

- The timing of data cleaning is crucial and can be approached in several ways:
  - \* *As soon as possible:* Cleaning early can prevent errors from propagating through your analysis.
  - \* *As late as possible:* Sometimes, it's better to wait until you have a clearer picture of what data is essential.
  - \* *In different stages:* Cleaning can be an ongoing process, occurring at various points in your workflow.
  - \* *Pipeline style:* Different approaches like ETL (Extract, Transform, Load), ELT (Extract, Load, Transform), and EtLT (Extract, transform, Load, Transform) offer structured ways to handle data cleaning.

- **Heuristics when dealing with data**

- When working with data, it's important to adopt a cautious mindset:

- 
- \* *Hope for the best, assume the worst:* Be optimistic but prepared for potential issues.
  - \* *Validate data early and often:* Regular checks can catch errors before they become problematic.
  - \* *Don't trust anything:* Always verify data integrity and accuracy.
  - \* *Be defensive:* Anticipate and guard against potential data issues to ensure reliable analysis.

### OLAP vs OLTP Workloads

- There are two classes of data workloads
- **OLTP**
  - On-Line Transactional Processing
  - Execute large numbers of transactions by a large number of processes in real-time
  - **Lots of concurrent small read / write transactions**
  - E.g., online banking, e-commerce, travel reservations
- **OLAP**
  - On-Line Analytical Processing
  - Perform multi-dimensional analysis on large volumes of data
  - **Few large read or write transactions**
  - E.g., data mining, business intelligence

**OLAP**



**OLTP**



- **Two Classes of Data Workloads**

- In the world of data processing, there are two main types of workloads: OLTP and OLAP. These are essential for different purposes and understanding them helps in choosing the right system for your needs.

- **OLTP (On-Line Transactional Processing)**

- This type of workload is designed to handle a large number of transactions. Think of it as the backbone of systems that require real-time processing.
  - **Lots of concurrent small read/write transactions:** OLTP systems are optimized for handling many small transactions simultaneously. This is crucial for applications where speed and efficiency are key.
  - *Examples:* Online banking, e-commerce platforms, and travel reservation systems are typical use cases. These systems need to process numerous transactions quickly and accurately.

- **OLAP (On-Line Analytical Processing)**

- OLAP is focused on analyzing large volumes of data. It's about understanding data trends and patterns rather than processing transactions.
  - **Few large read or write transactions:** Unlike OLTP, OLAP systems deal with fewer but much larger transactions. This is because they are designed to perform complex queries and data analysis.
  - *Examples:* Data mining and business intelligence applications rely on OLAP to provide insights and support decision-making processes. These systems help businesses understand their data better and make informed decisions.

---

## 18 / 21: Challenges with Data Pipelines

### Challenges with Data Pipelines

- High-volume vs low-volume
  - Lots of small reads / writes
  - A few large reads / writes
- Batch vs streaming
  - Real-time constraints
- API rate limits / throttling
- Connection time-outs
- Slow downloads
- Incremental mode vs catch-up



18 / 21

- Challenges with Data Pipelines

- High-volume vs low-volume

- \* Data pipelines often have to handle different types of data loads. *High-volume* refers to situations where there are many small data transactions, like numerous small reads and writes. This can be challenging because it requires the system to efficiently manage a large number of operations. On the other hand, *low-volume* involves fewer but larger transactions, which can be easier to manage but require the system to handle large data chunks at once.

- Batch vs streaming

- \* Data can be processed in two main ways: *batch* processing and *streaming*. Batch processing involves collecting data over a period and processing it all at once, which is suitable for tasks without real-time constraints. *Streaming* processes data in real-time as it arrives, which is crucial for applications needing immediate insights or actions.

- API rate limits / throttling

- \* Many data sources impose *API rate limits*, restricting the number of requests you can make in a given time frame. This is known as throttling and can be a significant challenge when building data pipelines, as it requires careful planning to avoid exceeding these limits and potentially losing access to critical data.

- Connection time-outs

- \* Data pipelines often rely on network connections to access data. *Connection time-outs* occur when a connection takes too long to establish or maintain, leading to interruptions in data flow. This can be particularly problematic for time-sensitive applications.

- 
- **Slow downloads**
    - \* Sometimes, data downloads can be slow due to network issues or server limitations. This can delay the entire data processing pipeline, especially if the pipeline depends on timely data retrieval to function correctly.
  - **Incremental mode vs catch-up**
    - \* Data pipelines can operate in *incremental mode*, where only new or changed data is processed, or in *catch-up* mode, where the pipeline processes all data to bring the system up to date. Choosing the right mode depends on the specific needs of the application and can impact the efficiency and speed of data processing.

## 19 / 21: Data Warehouse vs Data Lake

### Data Warehouse vs Data Lake

#### • Data warehouse

- = DB storing data from different systems in a structured way
- Corresponds to ETL data pipeline style
- E.g., a large Postgres instance with many DBs and tables
- E.g.,
  - AWS Athena, RDS
  - Google BigQuery



#### • Data lake

- Data stored semi-structured or unstructured
- Corresponds to ELT data pipeline style
- E.g., AWS S3 bucket storing blog posts, flat files, JSON objects, images



19 / 21

#### • Data warehouse

- A *data warehouse* is essentially a large database that stores data from various sources in a highly organized and structured manner. This means that the data is cleaned, transformed, and formatted to fit into predefined tables and schemas.
- It follows the ETL (Extract, Transform, Load) data pipeline style. In this process, data is first extracted from different sources, then transformed into a suitable format, and finally loaded into the data warehouse.
- An example of a data warehouse could be a large Postgres instance that contains multiple databases and tables, each designed to hold specific types of data.
- Popular services that provide data warehousing capabilities include AWS Athena and RDS, as well as Google BigQuery. These platforms offer robust solutions for managing and querying large volumes of structured data efficiently.

#### • Data lake

- A *data lake* is a storage system that holds data in its raw form, which can be semi-structured or unstructured. This means that data is stored as-is, without any transformation or structuring, allowing for a more flexible and scalable storage solution.
- It aligns with the ELT (Extract, Load, Transform) data pipeline style. Here, data is first extracted and loaded into the data lake, and transformation occurs later, often when the data is needed for analysis.
- An example of a data lake could be an AWS S3 bucket that stores a variety of data types such as blog posts, flat files, JSON objects, and images. This flexibility allows organizations to store vast amounts of diverse data types in a single repository, making it easier to perform big data analytics.

### Data Lake: Pros and Cons

- **Data lake**
  - Stores semi-structured or unstructured data
- **Pros**
  - Cheaper cloud storage
  - Easier changes to types or properties
    - E.g., JSON documents
  - Data scientists
    - Initially unsure how to access and use data
    - Want to explore raw data
- **Cons**
  - Not optimized for querying like structured data warehouse
    - Tools query data lake similar to SQL
    - E.g., AWS Athena, Redshift Spectrum



20 / 21

- **Data lake**
  - A *data lake* is a storage system that holds a vast amount of raw data in its native format, which can be either semi-structured or unstructured. This means that the data is stored as-is, without any transformation or structuring, allowing for flexibility in how it can be used later.
- **Pros**
  - **Cheaper cloud storage:** Data lakes are cost-effective because they use cloud storage solutions that are generally cheaper than traditional data warehouses. This makes them an attractive option for storing large volumes of data.
  - **Easier changes to types or properties:** Since data lakes store data in its raw form, it's easier to make changes to data types or properties. For example, JSON documents can be stored without needing to define a schema upfront, allowing for flexibility in data management.
  - **Data scientists:** Data lakes are particularly beneficial for data scientists who may not know exactly what data they need at the outset. They can explore and experiment with the raw data to uncover insights and patterns without being constrained by predefined structures.
- **Cons**
  - **Not optimized for querying like structured data warehouse:** Data lakes are not designed for fast querying like traditional data warehouses. While tools exist to query data lakes using SQL-like syntax (such as AWS Athena and Redshift Spectrum), these queries may not be as efficient or fast as those on structured data warehouses. This can be a drawback for users who need quick access to specific data insights.

---

## 21 / 21: Advantages of Cloud Computing

### Advantages of Cloud Computing

- Ease of building and deploying:
  - Data pipelines
  - Data warehouses
  - Data lakes
- Managed services
  - No need for admin and deploy
  - Highly scalable DBs
    - E.g., Amazon Redshift, Google BigQuery, Snowflake
- Rent-vs-buy
  - Easy to scale up and out
  - Easy to upgrade
  - Better cash-flow
- Cost of storage and compute is continuously dropping
  - Economies of scale
- Cons
  - The flexibility has a cost (2x-3x more expensive than owning)
  - Vendor lock-in



21 / 21

- Ease of building and deploying:
  - Cloud computing makes it simpler to create and launch *data pipelines*, which are systems that move data from one place to another, often transforming it along the way. This is crucial for processing large amounts of data efficiently.
  - *Data warehouses* are centralized repositories for storing and analyzing large volumes of structured data. Cloud services provide tools to set these up quickly without needing to manage physical hardware.
  - *Data lakes* are storage systems that hold vast amounts of raw data in its native format. Cloud platforms offer the flexibility to store and process this data easily.
- Managed services:
  - Cloud providers offer *managed services*, meaning they handle the maintenance and deployment of infrastructure, freeing users from these tasks.
  - They provide *highly scalable databases*, which can grow with your needs without requiring manual intervention. Examples include Amazon Redshift, Google BigQuery, and Snowflake, which are designed to handle large-scale data processing efficiently.
- Rent-vs-buy:
  - Cloud computing allows businesses to *scale up and out* easily, meaning they can increase their resources as needed without significant upfront investment.
  - It also simplifies *upgrading* systems, as cloud providers regularly update their services.
  - This model improves *cash-flow* since businesses pay for what they use rather than investing heavily in infrastructure upfront.
- Cost of storage and compute is continuously dropping:
  - The *economies of scale* achieved by cloud providers mean that as they grow, they can offer storage and computing power at lower costs, benefiting users.

- 
- **Cons:**
    - While cloud computing offers flexibility, it can be *2x-3x more expensive* than owning and managing your own infrastructure.
    - There is a risk of *vendor lock-in*, where switching providers becomes difficult due to reliance on a specific provider's services and tools.

---

## Lesson 3.2: Docker



### Lesson 3.2: Docker

**Instructor:** Dr. GP Saggese, [gsaggese@umd.edu](mailto:gsaggese@umd.edu)



1 / 8

---

## 2 / 8: Containerizing an App

### Containerizing an App

- **Containerizing an app** means creating a container with your app inside
- Develop application code with dependencies
  - Install dependencies
    - Inside a container
    - Inside a virtual env
- Create a Dockerfile describing:
  - App
  - Dependencies
  - How to run it
- Build image with `docker image build`
- (Optional) Push image to Docker image registry
- Run/test container from image
- Distribute app as a container (no installation required)



2 / 8

- **Containerizing an app** involves packaging your application and all its dependencies into a single, lightweight container. This makes it easy to run the app consistently across different environments.
- **Develop application code with dependencies:**
  - When developing your app, you need to ensure all necessary libraries and tools are included.
  - *Install dependencies* can be done in two main ways:
    - \* **Inside a container:** This ensures that the app and its dependencies are bundled together, making it portable.
    - \* **Inside a virtual environment:** This is another method to manage dependencies, but it is more common in traditional development setups.
- **Create a Dockerfile describing:**
  - The **App:** Specify the application code and any necessary configurations.
  - **Dependencies:** List all the libraries and tools your app needs to run.
  - **How to run it:** Provide instructions on how to start the application within the container.
- **Build image with `docker image build`:** This command compiles the Dockerfile into a Docker image, which is a snapshot of your app and its environment.
- **(Optional) Push image to Docker image registry:** By uploading your image to a registry, you make it accessible to others or to different environments, facilitating easy sharing and deployment.

- 
- **Run/test container from image:** Once the image is built, you can run it to test if the app works as expected in the containerized environment.
  - **Distribute app as a container (no installation required):** Containers simplify app distribution because they encapsulate everything needed to run the app, eliminating the need for users to install dependencies separately.

## 3 / 8: Building a Container

### Building a Container

- **Dockerfile**
  - Describe how to create a container
- **Build context**
  - `docker build -t web:latest .` where `.` is the build context
  - Send directory containing the application to Docker engine to build the application
  - Typically the Dockerfile is in the root directory of the build context



3 / 8

- **Building a Container**
  - **Dockerfile**
    - \* The Dockerfile is a script that contains a series of instructions on how to build a Docker container. Think of it as a recipe that tells Docker what base image to use, what software to install, and what commands to run. This file is crucial because it defines the environment and configuration of your containerized application. By writing a Dockerfile, you ensure that your application can run consistently across different environments.
  - **Build context**
    - \* The build context is the set of files that Docker needs to build the container. When you run the command `docker build -t web:latest ..`, the `..` specifies the current directory as the build context. This means Docker will use all the files in this directory to build the container. It's important to ensure that the Dockerfile is located in the root directory of the build context so Docker can find it easily. This setup allows Docker to package your application and its dependencies into a container efficiently.

---

## 4 / 8: Dockerfile: Example

### Dockerfile: Example

```
FROM python:3.8-slim-buster
LABEL maintainer="gsaggese@umd.edu"

WORKDIR /app

COPY requirements.txt requirements.txt
RUN pip3 install -r requirements.txt

COPY . .

CMD ["python3", "-m", "flask", "run", "--host=0.0.0.0"]
```



4 / 8

- **FROM python:3.8-slim-buster:** This line specifies the base image for our Docker container. We're using a lightweight version of Python 3.8, which is based on Debian Buster. This choice helps keep the container size small while providing the necessary Python environment.
- **LABEL maintainer="gsaggese@umd.edu":** Here, we're adding metadata to the image. The `maintainer` label indicates who is responsible for maintaining the Dockerfile. This can be useful for others who might need to contact the maintainer for questions or issues.
- **WORKDIR /app:** This command sets the working directory inside the container to `/app`. Any subsequent commands will be run in this directory. It helps organize the file structure within the container.
- **COPY requirements.txt requirements.txt:** This line copies the `requirements.txt` file from the host machine into the container. This file typically contains a list of Python packages that the application depends on.
- **RUN pip3 install -r requirements.txt:** After copying the `requirements.txt` file, this command installs the required Python packages listed in it. Using `pip3` ensures that the packages are installed for Python 3.
- **COPY . .:** This command copies all files from the current directory on the host machine into the working directory of the container. This includes the application code and any other necessary files.
- **CMD ["python3", "-m", "flask", "run", "--host=0.0.0.0"]:** This is the command that will be executed when the container starts. It runs a Flask application, which is a popular

---

web framework for Python. The `--host=0.0.0.0` option makes the server accessible from outside the container, which is important for testing and deployment.

## 5 / 8: Docker: Commands

### Docker: Commands

- Show all the available images

```
> docker images
REPOSITORY          TAG      IMAGE ID      CREATED        SIZE
counter_app-web-fe  latest   4bf6439418a1  17 minutes ago  54.7MB
...
```

- Show a particular image

```
> docker images counter_app_web-fe
counter_app-web-fe    latest      4bf6439418a1      17 minutes ago      54.7MB
...
```

- Delete an image

```
> docker rmi ...
```

- Show the running containers

```
> docker container ls
CONTAINER ID        IMAGE               COMMAND             CREATED            STATUS              PORTS               NAMES
505541bcf8b5        counter_app-web-fe   "python app.py"   7 minutes ago     Up 7 minutes       0.0.0.0:5001->5000/tcp   counter_app
c1889540cf2         redis:alpine        "docker-entrypoint.sh" 7 minutes ago     Up 7 minutes       6379/tcp           counter_app
```



5 / 8

- Show all the available images

- The command `docker images` is used to list all Docker images stored on your system. Each image is a snapshot of a container's file system and can be used to create running containers. The output includes details such as the repository name, tag, image ID, creation date, and size. This helps you manage and keep track of the images you have downloaded or created.

- Show a particular image

- To view details of a specific image, use `docker images` followed by the image name. This command filters the list to show only the specified image, making it easier to find information about a particular image when you have many stored on your system.

- Delete an image

- The command `docker rmi` is used to remove one or more Docker images from your system. This is useful for freeing up disk space or removing outdated images. You need to specify the image ID or name to delete it. Be cautious, as deleting an image will prevent you from creating new containers from it unless you re-download or rebuild it.

- Show the running containers

- Use `docker container ls` to list all currently running containers. This command provides details such as container ID, image name, command being executed, creation time, status, ports, and container names. This information is crucial for monitoring and managing active containers, ensuring they are running as expected and troubleshooting any issues.

---

## 6 / 8: Docker: Commands

### Docker: Commands

- Show running containers

```
> docker container ls
CONTAINER ID   IMAGE          COMMAND           CREATED          STATUS
PORTS          NAMES
281d654f6b8d   counter_app-web-fe   "python app.py"   5 minutes ago   Up 5 minutes
0.0.0.0:5001->5000/tcp  counter_app-web-fe-1

de55ae4104da   redis:alpine      "docker-entrypoint.s..." 5 minutes ago   Up 5 minutes
6379/tcp       counter_app-redis-1
```

- Show volumes and networks

```
> docker volume ls
DRIVER    VOLUME NAME
local     counter_app_counter-vol

> docker network ls
NETWORK ID   NAME          DRIVER    SCOPE
b4c1976d7c27  bridge        bridge    local
33ff702253b3  counter-app_counter-net bridge    local
```



6 / 8

- Show running containers

- The command `docker container ls` is used to list all the currently running Docker containers on your system. This is a useful command when you want to see which applications are actively running in your Docker environment.
- The output provides several details:
  - \* **CONTAINER ID:** A unique identifier for each running container.
  - \* **IMAGE:** The Docker image from which the container was created.
  - \* **COMMAND:** The command that is being executed inside the container.
  - \* **CREATED:** How long ago the container was started.
  - \* **STATUS:** The current state of the container, such as “Up” and the duration.
  - \* **PORTS:** Information about port mappings between the host and the container.
  - \* **NAMES:** The name assigned to the container, which can be used to reference it in other commands.

- Show volumes and networks

- The command `docker volume ls` lists all the Docker volumes on your system. Volumes are used to persist data generated by and used by Docker containers.
  - \* **DRIVER:** The type of driver used for the volume, typically “local”.
  - \* **VOLUME NAME:** The name of the volume, which can be used to reference it in other commands.
- The command `docker network ls` lists all the Docker networks. Networks allow containers to communicate with each other and with the outside world.
  - \* **NETWORK ID:** A unique identifier for each network.
  - \* **NAME:** The name of the network.
  - \* **DRIVER:** The type of network driver, such as “bridge”, which is the default net-

---

work driver.

- \* **SCOPE:** Indicates whether the network is local or global.

## 7 / 8: Docker: Delete State

### Docker: Delete State

- Commands:

```
> docker container ls  
> docker container rm $(docker container ls -q)  
  
> docker images  
> docker rmi $(docker images -q)  
  
> docker volume ls  
> docker volume rm $(docker volume ls -q)  
  
> docker network ls  
> docker network rm $(docker network ls -q)
```



7 / 8

- Docker: Delete State

- This slide is about removing different types of Docker resources. Docker is a tool that helps developers create, deploy, and run applications in containers. Sometimes, you need to clean up these resources to free up space or start fresh.

- Commands:

- List and Remove Containers:

- \* **docker container ls:** This command lists all running containers. Containers are like lightweight virtual machines that run your applications.
    - \* **docker container rm \$(docker container ls -q):** This command removes all running containers. The **-q** option lists only the container IDs, which are then passed to the **rm** command to delete them.

- List and Remove Images:

- \* **docker images:** This command lists all Docker images. Images are the blueprints for containers, containing everything needed to run an application.
    - \* **docker rmi \$(docker images -q):** This command removes all images. Like with containers, the **-q** option lists only the image IDs for removal.

- List and Remove Volumes:

- \* **docker volume ls:** This command lists all Docker volumes. Volumes are used to persist data generated by and used by Docker containers.
    - \* **docker volume rm \$(docker volume ls -q):** This command removes all volumes. Removing volumes will delete any data stored in them, so use this command with

---

caution.

- **List and Remove Networks:**

- \* `docker network ls`: This command lists all Docker networks. Networks allow containers to communicate with each other.
- \* `docker network rm $(docker network ls -q)`: This command removes all networks. Be careful when removing networks, as it can disrupt communication between containers.

## 8 / 8: Docker Tutorial

### Docker Tutorial

- [Docker tutorial](#)



8 / 8

- **Docker Tutorial**

- This slide introduces a tutorial on Docker, which is a platform used to develop, ship, and run applications inside containers. Containers are lightweight, portable, and ensure that software runs consistently across different computing environments.

- **Docker tutorial**

- The link provided directs you to a detailed tutorial on Docker. This tutorial is likely to cover the basics of Docker, including how to install it, create Docker images, and run containers. It might also delve into more advanced topics such as Docker Compose, which is used for defining and running multi-container Docker applications.
- Understanding Docker is crucial for anyone involved in software development or deployment, as it simplifies the process of managing application dependencies and environments. This tutorial is a valuable resource for beginners and those looking to deepen their understanding of containerization.
- *Key takeaway:* Docker helps in creating a consistent environment for applications, making it easier to develop, test, and deploy software across different platforms.

---

## Lesson 4.1: Relational DBs



### Lesson 4.1: Relational DBs

**Instructor:** Dr. GP Saggese, [gsaggese@umd.edu](mailto:gsaggese@umd.edu)



1 / 13

---

## 2 / 13: Relational Model: Overview

### Relational Model: Overview

- Introduced by Ted Codd (late 60's, early 70's)
- **First prototypes**
  - Ingres Project at Berkeley (1970-1985)
    - Ingres (INteractive Graphics REtrieval System)
    - → PostgreSQL (=Post Ingres)
  - IBM System R (1970s) → Oracle, IBM DB2
- **Contributions from relational data model**
  - Formal semantics for data operations
  - Data independence: separation of logical and physical data models
  - Declarative query languages (e.g., SQL)
  - Query optimization
- **Key to commercial success**



2 / 13

- **Relational Model: Overview**
  - The relational model was introduced by Ted Codd in the late 1960s and early 1970s. This model revolutionized how databases were structured and accessed, laying the groundwork for modern database systems.
- **First prototypes**
  - The Ingres Project at Berkeley, which ran from 1970 to 1985, was one of the first implementations of the relational model. Ingres stands for INteractive Graphics REtrieval System and was a pioneering project that eventually led to the development of PostgreSQL, a widely used database system today.
  - IBM System R was another early implementation in the 1970s. It was a research project that significantly influenced the development of commercial database systems like Oracle and IBM DB2.
- **Contributions from relational data model**
  - The relational model provided formal semantics for data operations, which means it defined a clear and mathematical way to handle data.
  - It introduced the concept of data independence, allowing the separation of logical data models (how data is organized) from physical data models (how data is stored), making databases more flexible and easier to manage.
  - Declarative query languages, such as SQL, emerged from this model, enabling users to specify what data they want without detailing how to retrieve it.
  - Query optimization became possible, allowing databases to efficiently execute queries by determining the best way to access and process data.
- **Key to commercial success**
  - The relational model's ability to provide a structured and efficient way to manage data

was crucial to its widespread adoption in commercial applications. Its principles continue to underpin many modern database systems.

### 3 / 13: Relational Model: Key Definitions

#### Relational Model: Key Definitions

- Relational DB is a collection of **tables / relations**
  - Unique name and schema for each table
  - E.g., `instructor` and `course` relations
- **Row / tuple / record:** Represents a relationship among values
- **Element:** Corresponds to a **column / field / attribute**
  - Atomic elements (e.g., phone number as a single object)
  - `NULL` for unknown or non-existent values
- **Schema of a relation**
  - List of attributes and their domains
  - Like type definition in programming languages
  - E.g., domain of `salary` is integers  $\geq 0$
- **Instance of relation**
  - Specific instantiation with actual values
  - Changes over time

ID	name	dept_name	salary
10101	Srinivasan	Comp. Sci.	65000
12121	Wu	Finance	90000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
32323	El Said	History	60000
33434	Perry	Physics	87000
45565	Katz	Comp. Sci.	75000
58583	Calisher	History	62000
76543	Singh	Finance	80000
76766	Crick	Biology	72000
83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000

instructor relation

course_id	title	dept_name	credits
BIO-101	Intro. to Biology	Biology	4
BIO-301	Genetics	Biology	4
BIO-399	Computational Biology	Biology	3
CS-101	Intro. to Computer Science	Comp. Sci.	4
CS-190	Game Design	Comp. Sci.	4
CS-315	Robotics	Comp. Sci.	3
CS-319	Image Processing	Comp. Sci.	3
CS-347	Database System Concepts	Comp. Sci.	3
EE-181	Intro. to Digital Systems	Elec. Eng.	3
FIN-201	Investment Banking	Finance	3
HIS-351	World History	History	3
MU-199	Music Video Production	Music	3
PHY-101	Physical Principles	Physics	4

course relation



3 / 13

- **Relational DB is a collection of tables / relations:** In a relational database, data is organized into tables, also known as relations. Each table has a unique name and a defined structure, called a schema. For example, you might have tables named `instructor` and `course`, each with its own set of columns and data types.
- **Row / tuple / record:** Each row in a table represents a single, complete set of related data. Think of it as a single entry in a table that connects different pieces of information, like a row in the `instructor` table that includes an instructor's ID, name, and department.
- **Element:** This refers to the individual data points within a row, corresponding to columns or fields. Each element is atomic, meaning it holds a single piece of data, like a phone number. If data is missing or unknown, it is represented by `NULL`.
- **Schema of a relation:** The schema defines the structure of a table, listing all the attributes (columns) and their data types or domains. It's similar to a type definition in programming, ensuring that data fits expected formats, like ensuring a `salary` is always a non-negative integer.
- **Instance of relation:** This is a snapshot of the table at a particular moment, filled with actual data. As data is added, removed, or updated, the instance of the relation changes over time, reflecting the current state of the database.

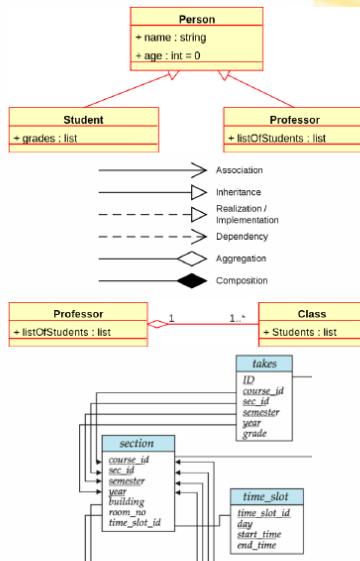
The images on the right visually represent the `instructor` and `course` tables, showing how data is organized in a relational database.

## 4 / 13: UML Class Diagram

### UML Class Diagram

- **UML class diagram**

- UML = Unified Modeling Language
- Used in OOP and DB design



- **In OOP design**

- Diagram showing classes, attributes, methods, and relationships

- **In DB design**

- Each box is a table / relation
- Columns / fields / attributes are listed inside the box
- Primary keys are underlined
- Foreign key constraints are represented by arrows



4 / 13

- **UML class diagram**

- UML stands for *Unified Modeling Language*, which is a standardized way to visualize the design of a system.
- It is commonly used in *Object-Oriented Programming (OOP)* and database (DB) design to help developers and designers understand the structure and relationships within a system.

- **In OOP design**

- A UML class diagram is a visual representation that shows the classes in a system, along with their attributes (properties) and methods (functions).
- It also illustrates the relationships between different classes, such as inheritance, association, and dependency, which helps in understanding how different parts of the system interact with each other.

- **In DB design**

- In the context of database design, each box in a UML class diagram represents a table or a relation.
- Inside each box, the columns or fields of the table are listed, which are also known as attributes.
- Primary keys, which uniquely identify each record in a table, are underlined to distinguish them from other attributes.
- Foreign key constraints, which establish relationships between tables, are depicted by arrows pointing from one table to another, indicating how tables are linked.

This slide provides a foundational understanding of how UML class diagrams are used in both software and database design, emphasizing their role in visualizing and organizing complex systems.

## 5 / 13: Example: University DB

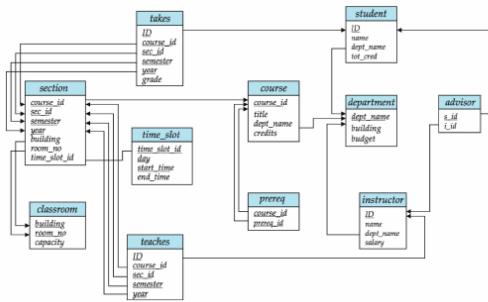
### Example: University DB

- **UML diagram of a DB and schemas representing a University**

- Each box is a table / relation
- Column / fields / attributes are listed inside the box
- Primary keys are underlined fields
- Foreign key constraints are arrows between boxes

- **Analysis of the diagram**

- ER model
- Entities
  - student
  - department
  - ...
- Relationships
  - takes
  - teaches
  - ...



5 / 13

- **UML diagram of a DB and schemas representing a University**

- The diagram is a visual representation of a database using UML (Unified Modeling Language). It helps us understand how data is organized within a university's database.
- **Each box is a table / relation:** In the diagram, each box represents a table in the database. A table is a collection of related data entries, similar to a spreadsheet.
- **Column / fields / attributes are listed inside the box:** Inside each box, you'll see a list of fields or attributes. These are the individual pieces of data stored in the table, like a student's name or ID.
- **Primary keys are underlined fields:** A primary key is a unique identifier for each record in a table. It's underlined in the diagram to show its importance.
- **Foreign key constraints are arrows between boxes:** Arrows indicate relationships between tables. A foreign key in one table points to a primary key in another, linking the data.

- **Analysis of the diagram**

- **ER model:** The diagram is based on an Entity-Relationship (ER) model, which is a way to visually represent data and its relationships.
- **Entities:** These are the main objects or concepts in the database. For example:
  - \* **student:** Represents students in the university.
  - \* **department:** Represents different departments within the university.
  - \* ...: There are likely more entities, each representing a different aspect of the university.
- **Relationships:** These show how entities are connected. For example:
  - \* **takes:** Represents the relationship between students and the courses they enroll in.
  - \* **teaches:** Represents the relationship between instructors and the courses they

teach.

- \* ...: Additional relationships help define how entities interact with each other.

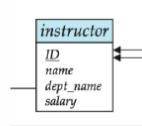
## 6 / 13: Primary Key

### Primary Key

- $R$  is a set of attributes of a relation  $r$ 
  - E.g., ID, name, dept\_name, salary are attributes of instructor
- $K$  is a superkey of  $R$  if values for  $K$  identify a unique tuple of each relation  $r(R)$ 
  - E.g., (ID) and (ID, name) are superkeys of instructor
  - (name) is not a superkey of instructor
- **Primary key:** minimal set of attributes that uniquely identify each row
  - Typically small and immutable
  - Would SSN be a primary key? Yes and no
- **Primary key constraint:** rows can't have the same primary key

ID	name	dept_name	salary
10101	Srinivasan	Comp. Sci.	65000
12121	Wu	Finance	90000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
32343	El Said	History	60000
33456	Gold	Physics	87000
45565	Katz	Comp. Sci.	75000
58583	Califieri	History	62000
76543	Singh	Finance	80000
76766	Crick	Biology	72000
83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000

instructor relation



6 / 13

- **$R$  is a set of attributes of a relation  $r$ :** In databases, a relation is like a table, and attributes are the columns in that table. For example, in an **instructor** table, the columns might be **ID**, **name**, **dept\_name**, and **salary**. These columns hold specific types of data for each instructor.
- **$K$  is a superkey of  $R$ :** A superkey is a set of one or more columns that can uniquely identify any row in the table. For instance, in the **instructor** table, the **ID** column alone can uniquely identify each instructor, making it a superkey. Even a combination like **(ID, name)** is a superkey, but it's not minimal. However, just **name** isn't a superkey because multiple instructors could have the same name.
- **Primary key:** This is the smallest set of columns that can uniquely identify each row in the table. It's important for a primary key to be small and not change over time. For example, a Social Security Number (SSN) could be a primary key because it's unique to each person, but it might not be ideal due to privacy concerns.
- **Primary key constraint:** This rule ensures that no two rows in the table can have the same primary key value. This is crucial for maintaining the uniqueness of each row in the database, preventing duplicate entries.

## 7 / 13: Question: What Are Primary Keys?

### Question: What Are Primary Keys?

- Marital status
  - Married(person1\_ssn, person2\_ssn, date\_married, date\_divorced)
- Bank account
  - Account(cust\_ssn, account\_number, cust\_name, balance, cust\_address)
- Research assistantship at UMD
  - RA(student\_id, project\_id, supervisor\_id, appt\_time, appt\_start\_date, appt\_end\_date)
- Information typically found on Wikipedia
  - Person(Name, Born, Died, Citizenship, Education, ...)
- Info about US President on Wikipedia
  - President(name, start\_date, end\_date, vice\_president, preceded\_by, succeeded\_by)
- Tour de France: historical rider participation information
  - Rider(Name, Born, Team-name, Coach, Sponsor, Year)



7 / 13

- Question: What Are Primary Keys?

- Marital status

- Marital status
- \* This example shows a table that might be used to track marital status. The primary key here could be a combination of *person1\_ssn* and *person2\_ssn*, as these two together uniquely identify a marriage record. The *date\_married* and *date\_divorced* provide additional context but are not unique identifiers.

- Bank account

- Bank account
- \* In this table, the primary key is likely the *account\_number*, as it uniquely identifies each bank account. Other fields like *cust\_ssn* and *cust\_name* provide additional information but are not unique by themselves.

- Research assistantship at UMD

- Research assistantship at UMD
- \* Here, the primary key could be a combination of *student\_id* and *project\_id*, as this combination uniquely identifies each research assistantship. The other fields provide details about the appointment.

- Information typically found on Wikipedia

- Information typically found on Wikipedia
- \* For a general person entry, a primary key might not be explicitly defined, but a unique identifier could be a combination of *Name* and *Born* date, as these together can uniquely identify a person.

- Info about US President on Wikipedia

- Info about US President on Wikipedia
- \* The primary key could be a combination of *name* and *start\_date*, as these together uniquely identify a presidential term. Other fields provide context about the presidency.

- Tour de France: historical rider participation information

- Tour de France: historical rider participation information
- \* In this table, a primary key could be a combination of *Name* and *Year*, as these

together uniquely identify a rider's participation in a specific year. Other fields provide additional details about the rider's participation.

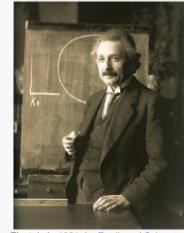
- **Image Context**

- The image likely provides a visual representation of how primary keys are used in databases to uniquely identify records. It might show examples of tables with highlighted primary keys to reinforce the concept.

## 8 / 13: Answer: What Are Primary Keys?

### Answer: What Are Primary Keys?

- Marital status
  - Married(**person1\_ssn**, **person2\_ssn**, **date\_married**, **date\_divorced**)
- Bank account
  - Account(**cust\_ssn**, **account\_number**, **cust\_name**, **balance**, **cust\_address**)
- Research assistantship at UMD
  - RA(**student\_id**, **project\_id**, **supervisor\_id**, **appt\_time**, **appt\_start\_date**, **appt\_end\_date**)
- Information typically found on Wikipedia
  - Person(**Name**, **Born**, **Died**, **Citizenship**, **Education**, ...)
- Info about US President on Wikipedia
  - President(**name**, **start\_date**, **end\_date**, **vice\_president**, **preceded\_by**, **succeeded\_by**)
- Tour de France: historical rider participation information
  - Rider(**Name**, **Born**, **Team-name**, Coach, Sponsor, **Year**)



Albert Einstein

Einstein in 1921, by Ferdinand Schmutzler

**Born** 14 March 1879  
Ulm, Germany

**Died** 18 April 1955 (aged 76)  
Princeton, New Jersey, U.S.

**Citizenship** Full list [show]

**Education** Federal polytechnic school in Zurich (Federal teaching diploma, 1900)  
University of Zurich (PhD, 1905)

**Known for** General relativity  
Special relativity  
Photoelectric effect  
 $E=mc^2$  (Mass-energy equivalence)  
 $E=h\nu$  (Planck-Einstein relation)  
Theory of Brownian motion



8 / 13

- **Marital Status**

- In a database table for marital status, the primary key is a combination of **person1\_ssn**, **person2\_ssn**, and **date\_married**. This means that each marriage record is uniquely identified by the social security numbers of the two individuals involved and the date they got married. This combination ensures that each marriage is distinct, even if the same individuals marry multiple times.

- **Bank Account**

- For bank accounts, the primary key is the **account\_number**. This is a unique identifier for each bank account, ensuring that no two accounts have the same number. It helps in efficiently managing and accessing account details like customer name, balance, and address.

- **Research Assistantship at UMD**

- In the context of research assistantships, the primary keys are **student\_id** and **project\_id**. This combination uniquely identifies each assistantship position, linking a student to a specific project. Additionally, **appt\_start\_date** is also a key, which might be used to track the duration of the assistantship.

- **Information Typically Found on Wikipedia**

- For a person's information on Wikipedia, the primary key is a combination of **Name**, **Born**, **Died**, **Citizenship**, **Education**, .... This set of attributes uniquely identifies a person, considering that names alone might not be unique.
- **Info About US President on Wikipedia**
  - The primary keys for US Presidents are **name** and **start\_date**. This combination ensures that each presidential term is uniquely identified, even if a president serves non-consecutive terms.
- **Tour de France: Historical Rider Participation Information**
  - For historical rider participation in the Tour de France, the primary keys are **Name**, **Born**, **Team-name**, and **Year**. This combination uniquely identifies a rider's participation in a specific year, accounting for changes in teams or multiple participations over different years.

In summary, primary keys are crucial in databases as they uniquely identify each record, ensuring data integrity and efficient data retrieval.

## 9 / 13: Foreign Key

### Foreign Key

- **Foreign key** = primary key of another relation
  - E.g., (ID) from student in takes, advisor
  - takes is the “referencing relation”, has the foreign key
  - student is the “referenced relation”, has the primary key
  - Shown by an arrow from referencing → referenced
- **Foreign key constraint:** for each row, the primary key tuple must exist
  - Aka referential integrity constraint
  - If (student101, DATA605) is in takes, there must be student101 in student
  - The key referenced as a foreign key must exist as a primary key



9 / 13

- **Foreign key:** This is a concept in databases where a column (or a set of columns) in one table is used to refer to the primary key of another table.
  - For example, if you have a column (ID) in a table called **takes**, it might be used to refer to the ID in another table called **student**. This means that the ID in **takes** is a foreign key.
  - The table **takes** is known as the “referencing relation” because it contains the foreign key.
  - The table **student** is the “referenced relation” because it contains the primary key that the foreign key points to.

- This relationship is often depicted with an arrow pointing from the referencing table to the referenced table.
- **Foreign key constraint:** This is a rule that ensures data integrity between tables.
  - Also known as the referential integrity constraint, it requires that for every foreign key value in the referencing table, there must be a corresponding primary key value in the referenced table.
  - For instance, if there is an entry (`student101`, `DATA605`) in the `takes` table, there must be a corresponding `student101` entry in the `student` table.
  - This ensures that the foreign key always points to a valid, existing primary key in the referenced table.

## 10 / 13: Relational Algebra: 1/4

### Relational Algebra: 1/4

- **Relation:** set of tuples
- **Relational algebra:** operations on relations producing a new relation
  - Unary: selection, projection, rename
  - Binary: union, set difference, intersection, Cartesian product, join
- **Selection  $\Sigma$ :** select tuples satisfying a predicate
  - E.g., select `instructor` tuples where `dept_name = "Physics"`
- **Projection  $\pi$ :** return tuples with subset of attributes
  - E.g., project `instructor` tuples with `(name, salary)`
- **Set operations:** union, intersection, set difference
  - Must be compatible (same attributes)

<i>ID</i>	<i>name</i>	<i>dept_name</i>	<i>salary</i>
10101	Srinivasan	Comp. Sci.	65000
12121	Wu	Finance	90000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
32343	El Said	History	60000
33456	Gold	Physics	87000
45565	Katz	Comp. Sci.	75000
58583	Califeri	History	62000
76543	Singh	Finance	80000
76766	Crick	Biology	72000
83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000

<i>ID</i>	<i>name</i>	<i>dept_name</i>	<i>salary</i>
22222	Einstein	Physics	95000
33456	Gold	Physics	87000

$\sigma_{dept.name = "Physics"}(instructor)$

<i>ID</i>	<i>name</i>	<i>salary</i>
10101	Srinivasan	65000
12121	Wu	90000
15151	Mozart	40000
22222	Einstein	95000
32343	El Said	60000
33456	Gold	87000
45565	Katz	75000
58583	Califeri	62000
76543	Singh	80000
76766	Crick	72000
83821	Brandt	92000
98345	Kim	80000

$\Pi_{ID, name, salary}(instructor)$     10 / 13

- **Relation:** In the context of databases, a *relation* is essentially a table. Each table consists of rows, known as *tuples*, and columns, which are the attributes of the data. Think of a relation as a structured way to store data where each row represents a unique data entry.
- **Relational algebra:** This is a set of operations used to manipulate and query data stored in relations. The result of these operations is always a new relation. Relational algebra is foundational for understanding how queries are processed in databases.
  - **Unary operations:** These operations work on a single relation.
    - \* *Selection ( $\Sigma$ )* filters rows based on a condition.
    - \* *Projection ( $\pi$ )* selects specific columns from the relation.
    - \* *Rename* changes the name of the relation or its attributes.
  - **Binary operations:** These require two relations.
    - \* *Union* combines rows from two relations.



- \* *Set difference* finds rows in one relation but not the other.
  - \* *Intersection* finds common rows between two relations.
  - \* *Cartesian product* pairs each row of one relation with every row of another.
  - \* *Join* combines rows from two relations based on a related attribute.
- **Selection  $\Sigma$** : This operation is used to filter data. For example, if you want to find all instructors in the Physics department, you would use a selection operation to filter out only those rows where the department name is “Physics”.
  - **Projection  $\pi$** : This operation is used to narrow down the columns of interest. For instance, if you only need the names and salaries of instructors, you would project these two attributes from the instructor relation.
  - **Set operations**: These operations allow you to combine or compare two relations. It’s important that the relations involved have the same attributes to ensure compatibility. This is similar to how you might combine or compare sets in mathematics.

## 11 / 13: Relational Algebra: 2/4

### Relational Algebra: 2/4

- **Cartesian product**: combine two relations into a new one
  - `instructor = (ID, name, dept_name, salary)`
  - `teaches = (ID, course_id, sec_id, semester, year)`
- E.g., `instructor x teaches` gives `(instructor.ID, instructor.name, instructor.dept_name, teaches.ID, ..., )`

<i>instructor</i> relation	<i>teaches</i> relation	<i>instructor x teaches</i>															
<i>ID</i>	<i>name</i>	<i>dept_name</i>	<i>salary</i>	<i>ID</i>	<i>course_id</i>	<i>sec_id</i>	<i>semester</i>	<i>year</i>	<i>ID</i>	<i>name</i>	<i>dept_name</i>	<i>salary</i>	<i>teaches.ID</i>	<i>course_id</i>	<i>sec_id</i>	<i>semester</i>	<i>year</i>
10101 Srinivasan Comp. Sci. 65000	10101 CS-101 1 Fall 2017	10101 Srinivasan Comp. Sci. 65000 10101 CS-101 1 Fall 2017															
12121 Wu Finance 90000	10101 CS-315 1 Spring 2018	10101 Srinivasan Comp. Sci. 65000 10101 CS-315 1 Spring 2018															
15151 Mozart Music 40000	10101 CS-347 1 Fall 2017	10101 Srinivasan Comp. Sci. 65000 10101 CS-347 1 Fall 2017															
22222 Einstein Physics 95000	12121 FIN-201 1 Spring 2018	10101 Srinivasan Comp. Sci. 65000 12121 FIN-201 1 Spring 2018															
32343 El Said History 60000	15151 MU-199 1 Spring 2018	10101 Srinivasan Comp. Sci. 65000 15151 MU-199 1 Spring 2018															
33456 Gold Physics 87000	22222 PHY-101 1 Fall 2017	10101 Srinivasan Comp. Sci. 65000 22222 PHY-101 1 Fall 2017															
45565 Katz Comp. Sci. 75000	22222 HIS-351 1 Spring 2018	12121 Wu Finance 90000 10101 CS-101 1 Fall 2017															
58583 Califori History 62000	32343 HIS-351 1 Spring 2018	12121 Wu Finance 90000 10101 CS-315 1 Spring 2018															
76543 Singh Finance 80000	45565 CS-101 1 Spring 2018	12121 Wu Finance 90000 10101 CS-347 1 Fall 2017															
76766 Crick Biology 72000	45565 CS-319 1 Spring 2018	12121 Wu Finance 90000 12121 FIN-201 1 Spring 2018															
83821 Brandt Comp. Sci. 92000	76766 BIO-101 1 Summer 2017	12121 Wu Finance 90000 15151 MU-199 1 Spring 2018															
98345 Kim Elec. Eng. 80000	76766 BIO-301 1 Summer 2018	12121 Wu Finance 90000 22222 PHY-101 1 Fall 2017															
				... ...	... ...	... ...	... ...	... ...	... ...	... ...	... ...	... ...	... ...	... ...	... ...	... ...	... ...
				15151 Mozart Music 40000 10101 CS-101 1 Fall 2017	15151 Mozart Music 40000 10101 CS-315 1 Spring 2018	15151 Mozart Music 40000 10101 CS-347 1 Fall 2017	15151 Mozart Music 40000 12121 FIN-201 1 Spring 2018	15151 Mozart Music 40000 15151 MU-199 1 Spring 2018	15151 Mozart Music 40000 22222 PHY-101 1 Fall 2017								
				22222 Einstein Physics 95000 10101 CS-101 1 Fall 2017	22222 Einstein Physics 95000 10101 CS-315 1 Spring 2018	22222 Einstein Physics 95000 10101 CS-347 1 Fall 2017	22222 Einstein Physics 95000 12121 FIN-201 1 Spring 2018	22222 Einstein Physics 95000 15151 MU-199 1 Spring 2018	22222 Einstein Physics 95000 22222 PHY-101 1 Fall 2017								



- **Cartesian product**: This operation is a fundamental concept in relational algebra, which is a part of database theory. It allows us to combine two tables (or relations) into a single new table. The new table contains all possible combinations of rows from the original tables.
  - In this example, we have two relations: `instructor` and `teaches`. The `instructor` relation includes columns for `ID`, `name`, `dept_name`, and `salary`. The `teaches` relation includes columns for `ID`, `course_id`, `sec_id`, `semester`, and `year`.
- When we perform a Cartesian product of `instructor` and `teaches`, we create a new relation that includes every possible pairing of rows from these two tables. This means that for

each row in the `instructor` table, it is paired with every row in the `teaches` table. The resulting table will have columns from both tables, such as `instructor.ID`, `instructor.name`, `instructor.dept_name`, `teaches.ID`, and so on.

- **Visual aids:** The images provided in the slide show the `instructor` and `teaches` relations separately, and then the result of their Cartesian product. This visual representation helps in understanding how the Cartesian product operation combines the data from both tables into a larger set of data.
- *Important note:* While the Cartesian product is a powerful tool, it can result in very large tables, especially if the original tables have many rows. This is because the number of rows in the resulting table is the product of the number of rows in the original tables.

## 12 / 13: Relational Algebra: 3/4

### Relational Algebra: 3/4

- **Join:** composition of two operations
  - Cartesian product
  - Selection based on equality between two fields
  - E.g., `instructor`  $\times$  `teaches` when `instructor.ID` = `teaches.ID`

<code>ID</code>	<code>name</code>	<code>dept_name</code>	<code>salary</code>	<code>ID</code>	<code>course_id</code>	<code>sec_id</code>	<code>semester</code>	<code>year</code>	<code>instructor.ID</code>	<code>name</code>	<code>dept_name</code>	<code>salary</code>	<code>teaches.ID</code>	<code>course_id</code>	<code>sec_id</code>	<code>semester</code>	<code>year</code>
10101	Srinivasan	Comp. Sci.	65000	10101	CS-101	1	Fall	2017	10101	Srinivasan	Comp. Sci.	65000	10101	CS-315	1	Fall	2017
12121	Wu	Finance	90000	10101	CS-315	1	Spring	2018	10101	Srinivasan	Comp. Sci.	65000	10101	CS-347	1	Spring	2018
15151	Mozart	Music	40000	10101	CS-347	1	Fall	2017	12121	Wu	Finance	90000	12121	FIN-201	1	Spring	2018
22222	Einstein	Physics	95000	12121	FIN-201	1	Spring	2018	15151	Mozart	Music	40000	15151	MU-199	1	Spring	2018
22348	El Said	History	60000	15151	MU-199	1	Spring	2018	22222	Einstein	Physics	95000	22222	PHY-101	1	Fall	2017
33456	Gold	Physics	87000	22222	PHY-101	1	Fall	2017	32343	El Said	History	60000	32343	HIS-351	1	Spring	2018
45565	Katz	Comp. Sci.	75000	32343	HIS-351	1	Spring	2018	45565	Katz	Comp. Sci.	75000	45565	CS-101	1	Spring	2018
58583	Califeri	History	62000	45565	CS-101	1	Spring	2018	45565	Katz	Comp. Sci.	75000	45565	CS-319	1	Spring	2018
76543	Singh	Finance	80000	45565	CS-101	1	Spring	2018	76766	Crick	Biology	72000	76766	BIO-101	1	Summer	2017
76766	Crick	Biology	72000	45565	CS-319	1	Spring	2018	83821	Brandt	Comp. Sci.	92000	83821	CS-190	1	Spring	2017
83821	Bradt	Comp. Sci.	92000	76766	BIO-101	1	Summer	2017	83821	Brandt	Comp. Sci.	92000	83821	CS-190	2	Spring	2017
98345	Kim	Elec. Eng.	80000	83821	CS-190	1	Spring	2017	83821	Brandt	Comp. Sci.	92000	83821	CS-319	2	Spring	2018
<b>instructor relation</b>				76766	BIO-301	1	Summer	2018	98345	Kim	Elec. Eng.	80000	98345	EE-181	1	Spring	2017
<b>teaches relation</b>				$\sigma_{instructor.ID=teaches.ID}(instructor \times teaches)$													



12 / 13

- **Join:** composition of two operations
  - The *join* operation in relational algebra is a fundamental concept used to combine two tables based on a related column. It is essentially a combination of two operations: the Cartesian product and selection.
  - **Cartesian product:** This operation takes two tables and pairs every row from the first table with every row from the second table. While this creates a large number of combinations, it is not very useful on its own because it doesn't consider any relationships between the data.
  - **Selection based on equality between two fields:** After the Cartesian product, the selection operation is applied to filter the results. This selection is based on a condition, typically an equality between two fields from the different tables. For example, if we have an `instructor` table and a `teaches` table, we might join them on the condition that

`instructor.ID = teaches.ID`. This means we only keep the rows where the instructor's ID matches the ID in the teaches table, effectively linking instructors to the courses they teach.

- The images in the slide likely show examples of the `instructor` and `teaches` relations, as well as the result of the join operation, illustrating how the data is combined based on the specified condition.

## 13 / 13: Relational Algebra: 4/4

### Relational Algebra: 4/4

- **Query:** combination of relational algebra operations
  - E.g., “*find course\_id from table section for fall 2017*”
- **Assignment:** assign parts of relational algebra to temporary relation variables
  - Write a query as a sequential program
  - E.g., “*find course\_id for classes in both fall 2017 and spring 2018*”
- **Equivalent queries:** two queries giving the same result on any DB instance
  - Some formulations are more efficient

$$\Pi_{course\_id}(\sigma_{semester='Fall' \wedge year=2017}(section))$$

$$\begin{aligned}courses\_fall\_2017 &\leftarrow \Pi_{course\_id}(\sigma_{semester='Fall' \wedge year=2017}(section)) \\courses\_spring\_2018 &\leftarrow \Pi_{course\_id}(\sigma_{semester='Spring' \wedge year=2018}(section)) \\courses\_fall\_2017 \cap courses\_spring\_2018\end{aligned}$$



13 / 13

- **Query:** In relational algebra, a query is essentially a combination of operations that allow us to retrieve specific data from a database. For example, if you want to find the `course_id` from a table named `section` for the fall of 2017, you would use a series of operations to filter and select the relevant data. This is similar to asking a question to the database and getting the answer in the form of data.
- **Assignment:** Sometimes, it's useful to break down complex queries into smaller parts. This is where assignment comes in. You can assign parts of your query to temporary variables, making it easier to manage and understand. Think of it like writing a step-by-step program where each step builds on the previous one. For instance, if you want to find `course_id` for classes in both fall 2017 and spring 2018, you might first find each separately and then combine the results.
- **Equivalent queries:** In relational algebra, different queries can sometimes produce the same result. These are known as equivalent queries. However, not all equivalent queries are created equal—some are more efficient than others. This means they can retrieve the same data faster or with less computational effort, which is important for optimizing database performance.

---

## Lesson 4.2: SQL



UMD DATA605 - Big Data Systems



### Lesson 4.2: SQL

**Instructor:** Dr. GP Saggese, [gsaggese@umd.edu](mailto:gsaggese@umd.edu)



1 / 32

### SQL Overview

- **Relational algebra:** mathematical language to manipulate *relations*
- **SQL:** language to describe and transform data in a relational DB
  - Originally Sequel
  - Changed to Structured Query Language
- **SQL statements grouped by goal**
  - Data definition language (DDL)
    - Define schema (tables, attributes, indices)
    - Specify integrity constraints (primary key, foreign key, not null)
  - Data modification language (DML)
    - Modify data in tables
    - Insert, Update, Delete
  - Query data (DQL)
  - Control transactions
    - Specify beginning and end, control isolation level
  - Define views
  - Authorization
    - Specify access and security constraints



2 / 32

- **Relational algebra:** This is a *mathematical framework* used to work with data organized in tables, known as relations. It provides a set of operations that allow you to manipulate and retrieve data in a structured way. Think of it as the theoretical foundation for how databases manage and process data.
- **SQL:** This stands for Structured Query Language, a powerful tool used to interact with relational databases. Originally called “Sequel,” SQL allows users to describe, manipulate, and query data stored in databases. It’s the standard language for managing and retrieving data in relational database systems.
- **SQL statements grouped by goal:**
  - **Data definition language (DDL):** This part of SQL is used to define and manage the structure of the database. It includes creating tables, defining columns, and setting up rules like primary keys (unique identifiers for records) and foreign keys (links between tables).
  - **Data modification language (DML):** These commands are used to change the data within the database. You can add new data (Insert), change existing data (Update), or remove data (Delete).
  - **Query data (DQL):** This involves retrieving data from the database, typically using the SELECT statement to specify what data you want to see.
  - **Control transactions:** Transactions are sequences of operations performed as a single unit. SQL allows you to manage these transactions, ensuring data integrity and consistency, by marking the start and end of transactions and setting isolation levels to control how transactions interact.
  - **Define views:** Views are virtual tables created by a query. They allow you to simplify

complex queries and present data in a specific format without altering the actual tables.

- **Authorization:** This involves setting permissions to control who can access or modify data, ensuring security and privacy within the database.

## 3 / 32: SQL Overview

### SQL Overview

- Data description language (DDL)

```
CREATE TABLE <name> (<field> <domain>, ... )
```

- Data modification language (DML)

```
INSERT INTO <name> (<field names>) VALUES (<field values>)
DELETE FROM <name> WHERE <condition>
UPDATE <name> SET <field name> = <value> WHERE <condition>
```

- Query language

```
SELECT <fields> FROM <name> WHERE <condition>
```



3 / 32

- **SQL Overview**

- **Data description language (DDL)**

- \* DDL is used to define and manage database structures. The `CREATE TABLE` command is a fundamental part of DDL. It allows you to create a new table in the database by specifying the table's name and its fields (or columns) along with their data types, known as domains. For example, you might create a table for storing customer information with fields like `CustomerID`, `Name`, and `Email`.

- **Data modification language (DML)**

- \* DML is used for managing data within the database. The `INSERT INTO` command adds new records to a table. You specify the table name, the fields you want to populate, and the corresponding values. The `DELETE FROM` command removes records from a table based on a specified condition, allowing you to manage data by removing unnecessary or outdated entries. The `UPDATE` command modifies existing records, setting new values for specified fields where certain conditions are met. This is useful for keeping data current and accurate.

- **Query language**

- \* The `SELECT` statement is the core of SQL's query language, allowing you to retrieve data from one or more tables. You specify the fields you want to retrieve, the table name, and any conditions that filter the results. This is essential for analyzing and reporting data, as it enables you to extract specific information from large datasets

efficiently.

## 4 / 32: Create Table

### Create Table

```
CREATE TABLE r
  (A_1 D_1,
   A_2 D_2,
   ...,
   A_n D_n,
   IntegrityConstraint_1,
   IntegrityConstraint_n);
```

where:

- *r* is name of *table* (aka *relation*)
- *A<sub>i</sub>* name of *attribute* (aka *field*, *column*)
- *D<sub>i</sub>* domain of attribute *A<sub>i</sub>*

- **Constraints**

- SQL prevents changes violating integrity constraints
- Primary key
  - Must be non-null and unique
  - PRIMARY KEY (A<sub>j1</sub>, A<sub>j2</sub>, ..., A<sub>jn</sub>)
- Foreign key
  - Attribute values must match primary key values in relation *s*
  - FOREIGN KEY (A<sub>k1</sub>, A<sub>k2</sub>, ..., A<sub>kn</sub>) REFERENCES *s*
- Not null
  - Null value not allowed for attribute
  - A<sub>i</sub> D<sub>i</sub> NOT NULL



4 / 32

- **Create Table Syntax**

- The CREATE TABLE statement is used to define a new table in a database. This is a fundamental operation in SQL, which stands for Structured Query Language, used for managing and manipulating relational databases.
- The syntax includes specifying the table name (*r*) and defining its structure with attributes (A<sub>1</sub>, A<sub>2</sub>, ..., A<sub>n</sub>) and their respective data types (D<sub>1</sub>, D<sub>2</sub>, ..., D<sub>n</sub>).

- **Table and Attributes**

- **r**: This is the name of the table, also known as a *relation* in database terminology. It is important to choose a meaningful name that reflects the data stored in the table.
- **A<sub>i</sub>**: These are the names of the attributes, which are also referred to as *fields* or *columns*. Each attribute represents a specific piece of data within the table.
- **D<sub>i</sub>**: This represents the domain or data type of the attribute A<sub>i</sub>. Common data types include integers, strings, and dates, which define what kind of data can be stored in each column.

- **Constraints**

- Constraints are rules applied to table columns to ensure data integrity and accuracy.
- **Primary Key**
  - \* A primary key is a unique identifier for each record in a table. It must be non-null and unique, ensuring that each entry can be uniquely identified.
  - \* The syntax PRIMARY KEY (A<sub>j1</sub>, A<sub>j2</sub>, ..., A<sub>jn</sub>) specifies which attribute(s) serve as the primary key.
- **Foreign Key**

- \* A foreign key is used to link two tables together. It ensures that the values in one table match values in another table's primary key, maintaining referential integrity.
- \* The syntax FOREIGN KEY (A\_k1, A\_k2, ..., A\_kn) REFERENCES s indicates that the attributes must match primary key values in another table s.
- **Not Null**
  - \* This constraint ensures that a column cannot have a null value, meaning every entry must have a value for this attribute.
  - \* The syntax A\_i NOT NULL is used to enforce this rule on a specific attribute.

## 5 / 32: Select

### Select

```
SELECT A_1, A_2, ..., A_n
    FROM r_1, r_2, ..., r_m
   WHERE P;
```

- **SELECT:** select the attributes to list (i.e., projection)
- **FROM:** list of tables to be accessed
  - Define a Cartesian product of the tables
  - The query is going to be optimized to avoid to enumerate tuples that will be eliminated
- **WHERE:** predicate involving attributes of the relations in the **FROM** clause (i.e., selection)
- In **SELECT** or **WHERE** clauses, might need to use the table names as prefix to qualify the attribute name
  - E.g., `instructor.ID` vs `teaches.ID`
- A **SELECT** statement can be expressed in terms of relational algebra
  - Cartesian product → selection → projection
  - Difference: SQL allows duplicate values, relational algebra works with mathematical sets



5 / 32

- **SELECT:** This keyword is used to specify which columns or attributes you want to retrieve from the database. It's like choosing specific data points from a larger dataset. This process is known as *projection* because you're projecting only certain columns from the entire table.
- **FROM:** Here, you list the tables from which you want to retrieve data. When you mention multiple tables, SQL initially considers all possible combinations of rows from these tables, known as a Cartesian product. However, don't worry about inefficiency; SQL optimizes the query to avoid unnecessary combinations that won't be used.
- **WHERE:** This clause is used to filter the data based on specific conditions. It acts like a sieve, allowing only the rows that meet the criteria to pass through. This is known as *selection*.
- **Table Name Prefixes:** Sometimes, different tables might have columns with the same name. To avoid confusion, you can prefix the column name with the table name, like `instructor.ID` or `teaches.ID`.
- **Relational Algebra Connection:** A **SELECT** statement can be broken down into relational

---

algebra operations: Cartesian product, selection, and projection. However, a key difference is that SQL can return duplicate rows, while relational algebra typically deals with sets, which don't allow duplicates.

## 6 / 32: Null Values

### Null Values

- An arithmetic operation with NULL yields NULL
- Comparison with NULL
  - $1 < \text{NULL}$
  - $\text{NOT}(1 < \text{NULL})$
  - SQL yields UNKNOWN when comparing with NULL value
  - There are 3 logical values: True, False, Unknown
- Boolean operators
  - Can be extended according to common sense, e.g.,
  - True AND UNKNOWN = UNKNOWN
  - False AND Unknown = False
- In a WHERE clause, if the result is UNKNOWN it's not included



6 / 32

- **Null Values**
  - When you perform any arithmetic operation with a NULL value, the result will always be NULL. This is because NULL represents an unknown or missing value, so any calculation involving it remains indeterminate.
- **Comparison with NULL**
  - If you try to compare a number, like 1, with NULL, the result is not straightforward. For example,  $1 < \text{NULL}$  doesn't return True or False; instead, it returns UNKNOWN.
  - Similarly, negating the comparison, such as  $\text{NOT}(1 < \text{NULL})$ , also results in UNKNOWN.
  - In SQL, when you compare anything with NULL, the outcome is UNKNOWN because NULL is not a value but a placeholder for missing information.
  - SQL logic includes three possible outcomes for comparisons: True, False, and Unknown.
- **Boolean operators**
  - Boolean logic in SQL can be extended to handle UNKNOWN values. For instance, if you have True AND UNKNOWN, the result is UNKNOWN because the unknown part makes the whole expression uncertain.
  - Conversely, False AND UNKNOWN results in False because the False value dominates the outcome regardless of the unknown part.
- **In a WHERE clause**
  - When using a WHERE clause in SQL, if the condition evaluates to UNKNOWN, the row is not included in the result set. This is because SQL only includes rows where the condition

is definitively True.

## 7 / 32: Group by Query

### Group by Query

- The attributes in GROUP BY are used to form groups
  - Tuples with the same value on all attributes are placed in one group
- Any attribute that is not in the GROUP BY can appear in the SELECT clause only as argument of aggregate function

```
SELECT dept_name, AVG(salary)
      FROM instructor
     GROUP BY dept_name;

-- Error.
SELECT dept_name, salary
      FROM instructor
     GROUP BY dept_name;
```

- salary is not in GROUP BY so it must be in an aggregate function

ID	name	dept.name	salary
76766	Crick	Biology	72000
45565	Katz	Comp. Sci.	75000
10101	Srinivasan	Comp. Sci.	65000
83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000
12121	Wu	Finance	90000
76543	Singh	Finance	80000
32343	El Said	History	60000
58583	Califieri	History	62000
15151	Mozart	Music	40000
33456	Gold	Physics	87000
22222	Einstein	Physics	95000

dept.name	avg_salary
Biology	72000
Comp. Sci.	77333
Elec. Eng.	80000
Finance	85000
History	61000
Music	40000
Physics	91000



7 / 32

- Group by Query:** This slide explains how the GROUP BY clause works in SQL queries. The GROUP BY clause is used to arrange identical data into groups. This is particularly useful when you want to perform aggregate calculations on subsets of data.
- Attributes in GROUP BY:** When you use GROUP BY, the attributes you specify are used to form groups. All rows with the same values in these attributes are grouped together. For example, if you group by dept\_name, all instructors from the same department will be grouped together.
- Attributes in SELECT Clause:** If an attribute is not included in the GROUP BY clause, it can only appear in the SELECT clause if it is used with an aggregate function like AVG, SUM, COUNT, etc. This is because SQL needs to know how to handle the non-grouped data.
- Example Query:** The first SQL example shows a correct usage where dept\_name is grouped, and AVG(salary) is calculated for each department. This is a valid query because salary is used within an aggregate function.
- Error Example:** The second SQL example demonstrates an error. Here, salary is included in the SELECT clause without being part of an aggregate function or the GROUP BY clause. This is not allowed because SQL cannot determine which salary value to display for each department group.
- Important Note:** Always ensure that any attribute not in the GROUP BY clause is used with an aggregate function in the SELECT clause to avoid errors. This is a common mistake when writing SQL queries involving grouping and aggregation.

## Having

- State a condition that applies to groups instead of tuples (like WHERE)
- Any attribute in the HAVING clause must appear in the GROUP BY clause
- E.g., find departments with avg salary of instructors > 42k

```
SELECT dept_name, AVG(salary) AS avg_salary
      FROM instructor
      GROUP BY dept_name
      HAVING AVG(salary) > 42000;
```

- How does it work
  - FROM is evaluated to create a relation
  - (optional) WHERE is used to filter
  - GROUP BY collects tuples into groups
  - (optional) HAVING is applied to each group and groups are filtered
  - SELECT generates tuples of the results, applying aggregate functions to get a single result for each group



8 / 32

- **Having**

- The HAVING clause is used to filter data at the group level, unlike the WHERE clause, which filters individual rows or tuples. This means that HAVING is applied after the data has been grouped.
- It's important to note that any attribute used in the HAVING clause must also be included in the GROUP BY clause. This ensures that the filtering condition is applied to the correct group of data.
- For example, if you want to find departments where the average salary of instructors is greater than \$42,000, you would use the HAVING clause to filter these groups after calculating the average salary for each department.

- **How does it work**

- The SQL query execution starts with the FROM clause, which identifies the tables involved and creates a relation from them.
- If a WHERE clause is present, it filters the rows based on specified conditions before any grouping occurs.
- The GROUP BY clause then collects these filtered rows into groups based on one or more columns.
- The HAVING clause is applied to these groups, allowing you to filter out entire groups based on aggregate conditions.
- Finally, the SELECT clause generates the output, applying any aggregate functions to produce a single result for each group, such as calculating averages or sums.

### Nested Subqueries

- SQL allows using the result of a query in another query
  - Use a subquery returning one attribute (scalar subquery) where a value is used
  - Use the result of a query for set membership in the WHERE clause
  - Use the result of a query in a FROM clause ::::columns ::::{.column width=60%}

```
SELECT tmp.dept_name, tmp.avg_salary
FROM (
    SELECT dept_name,
        AVG(salary) AS avg_salary
    FROM instructor
    GROUP BY dept_name) AS tmp
WHERE avg_salary > 42000
:::: :::: {.column width=40%}
```

dept_name	avg_salary
-----------	------------

Finance	85000.00000000000
---------	-------------------



9 / 32

- Nested Subqueries

- In SQL, you can use the result of one query as part of another query. This is known as a *nested subquery*. It allows for more complex data retrieval by building on simpler queries.
- **Scalar Subquery:** This is a subquery that returns a single value. You can use it in places where you would normally use a single value, like in a comparison in the WHERE clause.
- **Set Membership:** You can use a subquery to check if a value is part of a set of values. This is often done using the IN keyword in the WHERE clause.
- **FROM Clause:** You can use a subquery to create a temporary table that can be used in the FROM clause of another query. This is useful for organizing complex queries and breaking them into manageable parts.

- Example Explanation

- The SQL example provided demonstrates using a subquery in the FROM clause. The subquery calculates the average salary for each department from the `instructor` table and groups the results by `dept_name`.
- The outer query then selects departments where the average salary is greater than 42,000. The subquery is aliased as `tmp`, which acts like a temporary table for the outer query to use.
- This approach is useful for filtering data based on aggregated results, such as finding departments with higher-than-average salaries.

## With

- WITH clause allows to define a temporary relation containing the results of a subquery
- It can be equivalent to a nested subqueries, but clearer
- E.g., find department with the maximum budget ::::columns ::::{.column width=80%}

```
WITH max_budget(value) as (
    SELECT MAX(budget) FROM department)
SELECT department.dept_name, budget
    FROM department, max_budget
    WHERE department.budget = max_budget.value
:::: :::: {.column width=20%}
```

**dept\_name      budget**

Finance 120000.00



10 / 32

- **WITH Clause:** The WITH clause in SQL is used to create a temporary table or relation that holds the results of a subquery. This can make complex queries easier to read and manage by breaking them down into simpler parts. Instead of writing a long, nested subquery, you can define it once with WITH and then reference it in your main query.
- **Clarity Over Nested Subqueries:** Using the WITH clause can be more readable than using nested subqueries. It allows you to name the result of a subquery, making your SQL code more understandable and maintainable.
- **Example - Finding Maximum Budget:** In the example provided, the WITH clause is used to find the department with the maximum budget. The subquery `SELECT MAX(budget)` `FROM department` calculates the maximum budget and stores it in a temporary relation called `max_budget`. The main query then selects the department name and budget where the department's budget matches this maximum value.
- **Code Explanation:** The SQL code first defines `max_budget` using the WITH clause. It then selects the department name and budget from the `department` table where the budget equals the maximum budget found. This approach simplifies the query by separating the logic into distinct parts.
- **Visual Aid:** The image on the right likely provides a visual representation of the query's logic or the database schema, helping to further clarify how the WITH clause is applied in this context.

### Insert

- To insert data into a relation we can specify tuples to insert
  - Tuples

```
INSERT INTO course VALUES ('DATA-605', 'Big data systems', '(  
    INSERT INTO course(course_id, title, dept_name, credits)  
        VALUES ('DATA-605', 'Big data systems', 'Comp. Sci.', 3)
```

- Query whose results is a set of tuples

```
INSERT INTO instructor  
    (SELECT ID, name, dept_name, 18000  
     FROM student  
     WHERE dept_name = 'Music' AND tot_cred > 144)
```

- Nested queries are evaluated and then inserted so this doesn't create infinite loops

```
INSERT INTO student (SELECT * FROM student)
```

- Many DB have bulk loader utilities to insert a large set of tuples into a relation, reading from formatted text files This is much faster than `INSERT` statements



11 / 32

- **Insert Data into a Relation**

- When we want to add new data to a database table, we use the `INSERT` command. This command allows us to specify the exact data, or *tuples*, we want to add.

- **Tuples**

- \* A tuple is essentially a single row of data. In the example provided, the SQL command is adding a new course to the 'course' table. The command specifies the course ID, title, department name, and credits. This is a straightforward way to add a single row of data.

- \* The second example shows how you can specify the column names explicitly. This is useful when you want to ensure that the data is inserted into the correct columns, especially if the table has many columns.

- **Query Result as a Set of Tuples**

- \* Sometimes, instead of inserting data manually, you might want to insert data that is the result of a query. The example shows how to insert data into the 'instructor' table by selecting certain students from the 'student' table who meet specific criteria (e.g., students from the Music department with more than 144 credits).

- **Nested Queries**

- \* Nested queries are queries within queries. The example shows a nested query that selects all students and attempts to insert them back into the 'student' table. However, databases are designed to handle such operations without creating infinite loops, ensuring that the operation completes successfully.

- **Bulk Loader Utilities**

- \* For large datasets, using individual `INSERT` statements can be inefficient. Many databases offer bulk loader utilities that can read data from formatted text files and

---

insert it into the database much faster. This is particularly useful when dealing with big data, where the volume of data is too large for manual insertion.

## 12 / 32: Update

### Update

- SQL can change a value in a tuple without changing all the other values

- E.g., increase salary of all instructors by 5%

```
UPDATE instructor SET salary = salary * 1.05
```

- E.g., conditionally

```
UPDATE instructor SET salary = salary * 1.05
WHERE salary < 70000
```

- Nesting is allowed

```
UPDATE instructor SET salary = salary * 1.05
WHERE salary < (SELECT AVG(salary) FROM instructor)
```



12 / 32

- **Update**

In databases, the *UPDATE* command is used to modify existing data within a table. This is particularly useful when you need to change specific values without altering the entire dataset. For instance, if you want to adjust the salary of instructors, you can do so without affecting other attributes like their names or departments.

- **SQL can change a value in a tuple without changing all the other values**

This means you can target specific columns for updates. In our example, we focus on the salary column, leaving other columns untouched.

- **E.g., increase salary of all instructors by 5%**

The SQL command `UPDATE instructor SET salary = salary * 1.05` demonstrates how to increase every instructor's salary by 5%. This operation multiplies each salary by 1.05, effectively giving a raise across the board.

- **E.g., conditionally**

Sometimes, you only want to update certain rows based on a condition. The command `UPDATE instructor SET salary = salary * 1.05 WHERE salary < 70000` shows how to apply the raise only to instructors earning less than \$70,000. This conditional update ensures that only specific entries are modified.

- **Nesting is allowed**

SQL allows for more complex conditions using subqueries. For example, `UPDATE`

---

`instructor SET salary = salary * 1.05 WHERE salary < (SELECT AVG(salary) FROM instructor)` updates salaries for instructors earning below the average salary. This nested query first calculates the average salary and then applies the update conditionally, showcasing SQL's flexibility in handling complex data operations.

## 13 / 32: Delete

### Delete

- One can delete tuples using a query returning entire rows of a table

```
DELETE FROM r WHERE p
```

where:

- r is a relation
- P is a predicate

- Remove all tuples (but not the table)

```
DELETE FROM instructor
```



13 / 32

- Delete

- The `DELETE` statement in SQL is used to remove data from a table. It allows you to delete specific rows based on a condition or even all rows if no condition is specified.

- **One can delete tuples using a query returning entire rows of a table**

- \* The basic syntax for deleting rows is `DELETE FROM r WHERE p`. Here, r represents the table (or relation) from which you want to delete data, and p is the condition (or predicate) that specifies which rows should be deleted.
    - \* For example, if you have a table of students and you want to delete all students who have graduated, you would specify a condition that identifies those students.

- **Remove all tuples (but not the table)**

- \* If you want to delete all rows from a table but keep the table structure intact, you can use the `DELETE FROM instructor` command without a `WHERE` clause. This will remove all data from the `instructor` table, but the table itself will remain in the database.

- \* This is useful when you want to clear out old data but plan to reuse the table for new data in the future.



### SQL Tutorial

sql_basics.ipynb
sql_joins.ipynb
sql_nulls_and_unknown.ipynb

- SQL tutorial dir
- Readme\*\* \*\*
  - Explains how to run the tutorial
- Three notebooks in tutorial\_university
- **How to learn from a tutorial**
  - Reset the notebook
  - Execute each cell one at the time
  - Ideally create a new file and retype (!) everything
  - Understand what each cell does
  - Look at the output
  - Change the code
  - Play with it
  - Build your mental model



14 / 32

- **SQL Tutorial Directory**
  - This is the main folder where all the materials for the SQL tutorial are stored. It acts as the starting point for anyone looking to learn SQL through this tutorial.
- **Readme**
  - The Readme file is crucial as it provides instructions on how to run the tutorial. It's the first document you should read to understand the setup and execution process.
- **Three Notebooks in tutorial\_university**
  - These notebooks are part of the tutorial and contain practical exercises and examples. They are designed to help you learn SQL by doing, rather than just reading.
- **How to Learn from a Tutorial**
  - **Reset the Notebook:** Start fresh to ensure you don't have any leftover data or errors from previous sessions.
  - **Execute Each Cell One at a Time:** This helps you focus on understanding each part of the code.
  - **Retype Everything:** By creating a new file and retyping the code, you reinforce learning through practice.
  - **Understand Each Cell:** Take the time to comprehend what each piece of code is doing.
  - **Look at the Output:** Observing the results helps you connect the code with its effects.
  - **Change the Code:** Experimenting with the code helps deepen your understanding.
  - **Play with It:** Engaging with the code in a playful manner can lead to new insights and a better grasp of concepts.
  - **Build Your Mental Model:** Developing a mental framework of how SQL works will aid in solving problems and applying knowledge in real-world scenarios.

- **Movie Database Example (Optional)**

- This is an additional resource that provides a practical example of using SQL with a movie database. It's optional but can be a valuable exercise for applying what you've learned.

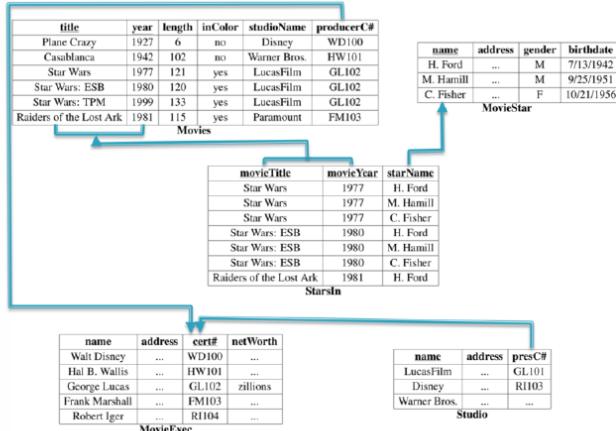
## 15 / 32: Example Schema for SQL Queries

### Example Schema for SQL Queries

```

Movie(title, year, length, inColor, studioName, producerC#)
StarsIn(movieTitle, movieYear, starName)
MovieStar(name, address, gender, birthdate)
MovieExec(name, address, cert#, netWorth)
Studio(name, address, presC#)

```



15 / 32

- **Example Schema for SQL Queries:** This slide presents a schema, which is essentially a blueprint or structure for organizing data in a database. It defines how data is stored, the relationships between different data entities, and the constraints on the data.
- **Movie Table:** This table stores information about movies. It includes fields like *title*, *year*, *length*, *inColor* (indicating if the movie is in color), *studioName* (the studio that produced the movie), and *producerC#* (a unique identifier for the producer).
- **StarsIn Table:** This table captures the relationship between movies and the stars who acted in them. It includes *movieTitle*, *movieYear*, and *starName*. This helps in linking actors to the movies they have starred in.
- **MovieStar Table:** This table contains details about movie stars, including their *name*, *address*, *gender*, and *birthdate*. It helps in storing personal information about actors.
- **MovieExec Table:** This table is for movie executives, capturing their *name*, *address*, *cert#* (a unique certification number), and *netWorth*. It provides insights into the people managing the movie industry.
- **Studio Table:** This table holds information about movie studios, including *name*, *address*, and *presC#* (a unique identifier for the studio president). It helps in organizing data about the companies producing films.

- **Context:** Understanding this schema is crucial for writing SQL queries, as it defines the structure of the data you will be querying. Each table represents a different aspect of the movie industry, and the relationships between them allow for complex queries to extract meaningful insights.

## 16 / 32: SQL: Data Definition

### SQL: Data Definition

- CREATE TABLE

```
CREATE TABLE movieExec (
    name char(30),
    address char(100),
    cert# integer primary key,
    networth integer);

CREATE TABLE movie (
    title char(100),
    year integer,
    length integer,
    inColor smallint,
    studioName char(20),
    producerC# integer references
        movieExec(cert#) );
```

- Must define `movieExec` before `movie`. Why?



16 / 32

- **SQL: Data Definition**

- **CREATE TABLE**

- \* The `CREATE TABLE` statement is used to define a new table in a database. In this example, two tables are being created: `movieExec` and `movie`.
- \* **movieExec Table:** This table includes columns for `name`, `address`, `cert#`, and `networth`. The `cert#` column is designated as the primary key, which means it uniquely identifies each record in the table.
- \* **movie Table:** This table includes columns for `title`, `year`, `length`, `inColor`, `studioName`, and `producerC#`. The `producerC#` column is a foreign key that references the `cert#` column in the `movieExec` table. This establishes a relationship between the two tables, linking each movie to a specific movie executive.

- **Must define `movieExec` before `movie`. Why?**

- \* The `movieExec` table must be defined before the `movie` table because the `movie` table contains a foreign key (`producerC#`) that references the `cert#` column in the `movieExec` table. In SQL, a table must exist before another table can reference it. This ensures that the database knows about the structure and constraints of the referenced table, allowing it to enforce referential integrity.

---

## 17 / 32: SQL: Data Manipulation

### SQL: Data Manipulation

- INSERT

```
INSERT INTO StarsIn values('King Kong', 2005, 'Naomi Watts')
INSERT INTO StarsIn(starName, movieTitle, movieYear)
    values('Naomi Watts', 'King Kong', 2005);
```

- DELETE

```
DELETE FROM movies WHERE movieYear < 1980;
```

- Syntax is fine, but this command will be rejected. Why?

```
DELETE FROM movies
    WHERE length < (SELECT avg(length) FROM movies);
```

- Problem: as we delete tuples, the average length changes
- Solution:
  - First, compute avg length and find all tuples to delete
  - Next, delete all tuples found above (without recomputing avg or retesting the tuples)



17 / 32

- INSERT

- The `INSERT` command is used to add new records to a table in a database. In the first example, we are adding a new entry to the `StarsIn` table with the values ‘King Kong’, 2005, and ‘Naomi Watts’. This assumes the table’s columns are in the order of movie title, year, and star name.
- The second example shows a more explicit way to insert data by specifying the column names. This is useful when you want to ensure that the data is inserted into the correct columns, especially if the table structure changes or if you are not inserting values for all columns.

- DELETE

- The `DELETE` command removes records from a table. The first example attempts to delete all movies released before 1980. However, the slide notes that this command will be rejected. This could be due to constraints like foreign keys or permissions that prevent deletion.
- The second example aims to delete movies with a length less than the average length of all movies. The problem here is that as movies are deleted, the average length recalculates, potentially altering which movies should be deleted. The solution involves calculating the average length first, identifying all movies to delete, and then deleting them in one go without recalculating the average. This ensures consistency and avoids logical errors.

## 18 / 32: SQL: Data Manipulation

### SQL: Data Manipulation

- UPDATE
  - Increase all movieExec netWorth's over 100,000 USD by 6%, all other accounts receive 5%
  - Write two update statements:

```
UPDATE movieExec SET netWorth = netWorth * 1.06
    WHERE netWorth > 100000;
UPDATE movieExec SET netWorth = netWorth * 1.05
    WHERE netWorth <= 100000;
```
  - The order is important
  - Can be done better using the case statement

```
UPDATE movieExec SET netWorth =
CASE
    WHEN netWorth > 100000 THEN netWorth * 1.06
    WHEN netWorth <= 100000 THEN netWorth * 1.05
END;
```



18 / 32

- **SQL: Data Manipulation**

- **UPDATE:** The UPDATE statement in SQL is used to modify existing records in a table. In this context, we are focusing on updating the `netWorth` of movie executives based on certain conditions.
- **Increase all movieExec netWorth's over 100,000 USD by 6%, all other accounts receive 5%:** This task involves adjusting the `netWorth` of movie executives. If an executive's `netWorth` is over 100,000 USD, it should be increased by 6%. If it is 100,000 USD or less, it should be increased by 5%.
- **Write two update statements:** The slide provides two separate SQL statements to achieve this task. The first statement updates the `netWorth` for those with more than 100,000 USD, and the second statement updates the rest.

```
UPDATE movieExec SET netWorth = netWorth * 1.06
    WHERE netWorth > 100000;
```

```
UPDATE movieExec SET netWorth = netWorth * 1.05
    WHERE netWorth <= 100000;
```

- **The order is important:** The order of these statements is crucial because executing them in the wrong order could lead to incorrect results. If the second statement is executed first, it would incorrectly apply a 5% increase to all records, including those that should receive a 6% increase.
- **Can be done better using the case statement:** The slide suggests a more efficient approach using a CASE statement within a single UPDATE command. This method is more concise and reduces the risk of errors associated with executing multiple statements in the correct order.

```

UPDATE movieExec SET netWorth =
CASE
    WHEN netWorth > 100000 THEN netWorth * 1.06
    WHEN netWorth <= 100000 THEN netWorth * 1.05
END;

```

- *Important Point:* Using a CASE statement not only simplifies the code but also ensures that all updates are handled in one atomic operation, reducing the potential for mistakes and improving maintainability.

## 19 / 32: SQL Single Table Queries

### SQL Single Table Queries

- Movies produced by Disney in 1990: note the *rename*

```

SELECT m.title, m.year
  FROM movie m
 WHERE m.studioname = 'disney' AND m.year = 1990;

```

- The SELECT clause can contain expressions

```
SELECT title || ' (' || to_char(year) || ')' AS titleyear
```

```
SELECT 2014 - year
```

- The WHERE clause support a large number of different predicates and combinations thereof

```
year BETWEEN 1990 and 1995
```

```
title LIKE 'star wars%'
```

```
title LIKE 'star wars _'
```



19 / 32

- **Movies produced by Disney in 1990: note the *rename***

– The SQL query provided is designed to retrieve a list of movies produced by Disney in the year 1990. The SELECT statement specifies that we want to see the title and year of each movie. The FROM clause indicates that this data is being pulled from a table named movie, which is given an alias m for convenience. The WHERE clause filters the results to only include movies where the studioname is ‘disney’ and the year is 1990. This is a basic example of a single table query that uses filtering to narrow down results.

- **The SELECT clause can contain expressions**

– SQL allows for expressions within the SELECT clause to manipulate or format the data being retrieved. For example, the expression title || ' (' || to\_char(year) || ')' AS titleyear concatenates the movie title with its year in parentheses, creating a new column named titleyear. This is useful for creating more readable outputs directly from the query.

– Another example is SELECT 2014 - year, which calculates the age of the movie by subtracting its release year from 2014. This demonstrates how SQL can be used to

perform calculations directly within the query.

- The WHERE clause supports a large number of different predicates and combinations thereof
  - The WHERE clause is powerful and flexible, allowing for various conditions to filter data. For instance, `year BETWEEN 1990 AND 1995` filters movies released within this range of years. This is a straightforward way to specify a range condition.
  - The LIKE operator is used for pattern matching. For example, `title LIKE 'star wars%'` finds titles that start with “star wars”, while `title LIKE 'star wars _'` looks for titles that start with “star wars” followed by exactly one additional character. These examples show how LIKE can be used to search for specific patterns in text data.

## 20 / 32: Single Table Queries

### Single Table Queries

- Find distinct movies sorted by title

```
SELECT DISTINCT title
  FROM movie
 WHERE studioname = 'disney' AND year = 1990
 ORDER by title;
```

- Average length of a movie

```
SELECT year, avg(length)
  FROM movie
 GROUP BY year;
```

- **GROUP BY:** is a very important concept that shows up in many data processing platforms

- What it does:
  - Partition the tuples by the group attributes (`year` in this case)
  - Do something (`compute avg` in this case) for each group
  - Number of resulting tuples == number of groups



20 / 32

- Find distinct movies sorted by title

- This SQL query is designed to retrieve a list of unique movie titles from a database. It specifically looks for movies produced by Disney in the year 1990.
- The `SELECT DISTINCT` clause ensures that each movie title appears only once in the results, even if there are duplicates in the database.
- The `WHERE` clause filters the results to include only those movies where the studio name is ‘disney’ and the release year is 1990.
- Finally, the `ORDER BY title` clause sorts the resulting list of movie titles alphabetically, making it easier to read and analyze.

- Average length of a movie

- This query calculates the average length of movies for each year.
- The `SELECT` statement retrieves the year and the average length of movies released in that year.

- The `GROUP BY year` clause is crucial here as it organizes the data into groups based on the year each movie was released.
- The `avg(length)` function computes the average movie length for each group of movies released in the same year.
- **GROUP BY:** is a very important concept that shows up in many data processing platforms
  - **What it does:**
    - \* It partitions the data into groups based on specified attributes, which is `year` in this example.
    - \* For each group, it performs a specified operation, such as computing the average length of movies.
    - \* The number of resulting rows in the output corresponds to the number of unique groups formed, which in this case is the number of distinct years in the dataset.

## 21 / 32: Single Table Queries

### Single Table Queries

- Find movie with the maximum length

```
SELECT title, year
  FROM movie
 WHERE movie.length = (SELECT max(length) FROM movie);
```

- The smaller “subquery” is called a “nested subquery”
- Find movies with at most 5 stars: an example of a correlated subquery

```
SELECT *
  FROM movies m
 WHERE 5 >= (SELECT count(*)
    FROM starsIn si
   WHERE si.title = m.title AND
         si.year = m.year);
```

- The “inner” subquery counts the number of actors for that movie.



21 / 32

- Single Table Queries
- Find movie with the maximum length

```
SELECT title, year
  FROM movie
 WHERE movie.length = (SELECT max(length) FROM movie);
```

- This query is designed to find the movie with the longest duration in the database.
- The main part of the query selects the `title` and `year` from the `movie` table.
- The condition `WHERE movie.length = (SELECT max(length) FROM movie)` uses a *nested subquery* to determine the maximum length of any movie in the table.

- A *nested subquery* is a query within another query. Here, it calculates the maximum length and uses that value to filter the main query.
- This approach is efficient for finding a single record that meets a specific criterion, like the longest movie.
- Find movies with at most 5 stars: an example of a correlated subquery

```
SELECT *
  FROM movies m
 WHERE 5 >= (SELECT count(*)
               FROM starsIn si
              WHERE si.title = m.title AND
                    si.year = m.year);
```

- This query retrieves movies that have five or fewer actors associated with them.
- The main query selects all columns from the `movies` table.
- The condition `5 >= (SELECT count(*) FROM starsIn si WHERE si.title = m.title AND si.year = m.year)` uses a *correlated subquery*.
- A *correlated subquery* is a subquery that depends on the outer query for its values. Here, it counts the number of actors (`starsIn`) for each movie.
- This technique is useful for filtering results based on related data, such as counting related records in another table.

## 22 / 32: Single Table Queries

### Single Table Queries

- Rank movies by their length

```
SELECT title, year,
       (SELECT count(*)
        FROM movies m2
       WHERE m1.length <= m2.length) AS rank
      FROM movies m1;
```

- Key insight: A movie is ranked 5th if there are exactly 4 movies with longer length.
- Most database systems support some sort of a `rank` keyword for doing this
- The above query doesn't work in presence of ties, etc.

- Set operations

```
SELECT name
      FROM movieExec
    union/intersect/minus
      SELECT name FROM
            movieStar
```



22 / 32

- Single Table Queries
  - Rank movies by their length

- \* The SQL query provided is designed to rank movies based on their length. It does this by comparing each movie's length to all other movies in the database.
- \* The subquery `(SELECT count(*) FROM movies m2 WHERE m1.length <= m2.length)` counts how many movies have a length greater than or equal to the current movie (`m1`). This count effectively gives the rank of the movie.
- \* **Key Insight:** If a movie is ranked 5th, it means there are exactly 4 movies that are longer than it.
- \* Many modern databases have built-in functions like `RANK()` that simplify this process and handle ties more gracefully.
- \* The provided query does not handle ties well, meaning if two movies have the same length, they might not be ranked correctly.
- **Set Operations**
  - The SQL snippet demonstrates how to use set operations to compare two lists of names from different tables.
  - **Union:** Combines results from both tables, removing duplicates, to show all unique names.
  - **Intersect:** Finds common names that appear in both `movieExec` and `movieStar`.
  - **Minus (or Except in some databases):** Shows names that are in `movieExec` but not in `movieStar`.
  - These operations are useful for comparing datasets and finding relationships or differences between them.

## 23 / 32: Single Table Queries

### Single Table Queries

- Set Comparisons

```

SELECT *
  FROM movies
 WHERE year IN [1990, 1995, 2000];

SELECT *
  FROM movies
 WHERE year NOT IN (
    SELECT EXTRACT(year from birthdate)
      FROM MovieStar
 );

```

- **Single Table Queries**
  - *Set Comparisons*

- \* The first SQL query demonstrates how to retrieve all records from the `movies` table where the `year` column matches any of the specified values: 1990, 1995, or 2000. This is achieved using the `IN` keyword, which is a convenient way to filter results based on a list of values. It's particularly useful when you want to check if a column's value belongs to a specific set of options.
- \* The second SQL query is a bit more complex. It selects all records from the `movies` table where the `year` is *not* in a set of years extracted from the `birthdate` column of the `MovieStar` table. The `EXTRACT(year from birthdate)` function is used to pull out the year part from the `birthdate`. This query is an example of a subquery, where the inner query provides a list of years that are then used by the outer query to filter out movies released in those years. This is useful for excluding certain records based on a dynamic set of values derived from another table.

## 24 / 32: Multi-Table Queries

### Multi-Table Queries

- Key:

- Do a join to get an appropriate table
- Use the constructs for single-table queries
- You will get used to doing all at once

- Examples:

```
SELECT title, year, me.name AS producerName
    FROM movies m, movieexec me
   WHERE m.producerC# = me.cert#;
```



24 / 32

- Multi-Table Queries

- Key:

- \* *Do a join to get an appropriate table:* When working with databases, data is often spread across multiple tables. To extract meaningful information, you need to combine these tables. This process is called a “join.” It allows you to link tables based on a related column, such as an ID or a key.
- \* *Use the constructs for single-table queries:* Once you've joined the tables, you can use the same SQL commands you would use for a single table. This means you can filter, sort, and select data as if it were all in one table.
- \* *You will get used to doing all at once:* Initially, working with multi-table queries might seem complex. However, with practice, you'll become comfortable with writ-

---

ing these queries and performing joins as a routine part of your data analysis.

- **Examples:**

- The provided SQL query demonstrates a multi-table query. It selects the `title` and `year` of movies from the `movies` table and the `name` of the producer from the `movieexec` table. The `WHERE` clause specifies that the `producerC#` from the `movies` table should match the `cert#` from the `movieexec` table. This join allows you to see which producer is associated with each movie, combining data from both tables into a single, cohesive result.

## 25 / 32: Multi-Table Queries

### Multi-Table Queries

- Consider the query:

```
SELECT title, year, producerC#, count(starName)
      FROM movies, starsIn
     WHERE title = starsIn.movieTitle AND
           year = starsIn.movieYear
    GROUP BY title, year, producerC#

```

- What about movies with no stars?
- Need to use **outer joins**

```
SELECT title, year, producerC#, count(starName)
      FROM movies LEFT OUTER JOIN starsIn
     ON title = starsIn.movieTitle AND year = starsIn.movie
    GROUP BY title, year, producerC#

```

- All tuples from 'movies' that have no matches in `starsIn` are included with `NULLs`
- So if a tuple (`m1, 1990`) has no match in `starsIn`, we get (`m1, 1990, NULL`) in the result
- The `count(starName)` works correctly then



SCIENCE  
ACADEMY

Note: `count(*)` would not work correctly (NULLs can have unintuitive behavior)

25 / 32

- **Multi-Table Queries**

- The first query example demonstrates a basic SQL query that retrieves information from two tables: `movies` and `starsIn`. It selects the movie title, year, producer code, and counts the number of stars associated with each movie.
- **Problem with the initial query:** It only considers movies that have entries in both tables. This means if a movie has no stars listed in the `starsIn` table, it won't appear in the results at all.
- **Solution: Use Outer Joins:** To include movies without stars, we use a **LEFT OUTER JOIN**. This type of join ensures that all records from the `movies` table are included in the results, even if there are no corresponding entries in the `starsIn` table.
- **Handling NULLs:** When there is no match in `starsIn`, the result will include `NULLs` for the star-related columns. For example, a movie with no stars will appear as (`m1, 1990, NULL`).
- **Counting Stars:** Using `count(starName)` correctly counts the number of stars, as it ignores `NULLs`. However, using `count(*)` would count all rows, including those with

---

NULLs, which might lead to misleading results.

## 26 / 32: Other SQL Constructs

### Other SQL Constructs

- Views

```
CREATE VIEW DisneyMovies
    SELECT *
        FROM movie m
    WHERE m.studioname = 'disney';
```

- Can use it in any place where a table name is used
- Views are used quite extensively to:
  - Simplify queries
  - Hide data (by giving users access only to specific views)
- Views may be *materialized* or not



26 / 32

- Other SQL Constructs

- Views

```
CREATE VIEW DisneyMovies
    SELECT *
        FROM movie m
    WHERE m.studioname = 'disney';
```

- **Can use it in any place where a table name is used:** Once a view is created, you can use it just like a regular table in your SQL queries. This means you can perform operations like SELECT, JOIN, and more on the view.
- **Views are used quite extensively to:**
  - \* **Simplify queries:** Views can encapsulate complex queries, making it easier to work with data. Instead of writing a long query every time, you can just reference the view.
  - \* **Hide data:** By creating views, you can control what data users can see. For example, if you only want users to see Disney movies, you can give them access to the DisneyMovies view instead of the entire movie table.
- **Views may be materialized or not:** A *materialized view* is a view where the data is stored physically, which can improve performance for complex queries. Non-materialized views, on the other hand, are virtual and do not store data physically; they are computed on the fly when queried.

### Other SQL Constructs

- NULLs
    - Value of any attribute can be NULL
    - Because: value is unknown, or it is not applicable, or hidden, etc.
    - Can lead to counterintuitive behavior
    - For example, the following query does not return movies where length = NULL

```
SELECT * FROM movies WHERE length >= 120 OR length <= 120
```
    - Aggregate operations can be especially tricky
  - Transactions
    - A transaction is a sequence of queries and update statements executed as a single unit
    - For example, transferring money from one account to another
      - Both the *deduction* from one account and *credit* to the other account should happen, or neither should
  - Triggers
    - A trigger is a statement that is executed automatically by the system as a side effect of a modification to the database
-  SCIENCE ACADEMY
- 27 / 32
- NULLs
    - In databases, any attribute can have a value of **NULL**, which essentially means that the value is unknown, not applicable, or intentionally hidden. This is important because it allows for flexibility in data representation, but it can also lead to unexpected results. For instance, when you run a query to find movies with a specific length, if the length is **NULL**, it won't be included in the results. This is because **NULL** is not considered equal to any value, including itself, which can be counterintuitive. Additionally, when performing aggregate operations like counting or averaging, **NULL** values can complicate the results, as they are typically ignored in these calculations.
  - Transactions
    - A transaction in SQL is a sequence of operations that are executed as a single unit. This is crucial for maintaining data integrity, especially in scenarios like transferring money between bank accounts. In such cases, both the deduction from one account and the credit to another must occur together. If one part of the transaction fails, the entire transaction should be rolled back to ensure consistency. This all-or-nothing approach helps prevent errors and ensures that the database remains in a valid state.
  - Triggers
    - Triggers are special types of SQL statements that are automatically executed in response to certain events on a table or view. They are used to enforce business rules, maintain audit trails, or automatically update related data. For example, a trigger might automatically update a stock inventory count when a new sale is recorded. Triggers help automate processes and ensure that certain actions are taken without requiring manual intervention, thus maintaining the integrity and consistency of the database.

### Other SQL Constructs

- Integrity Constraints
  - Predicates on the database that must always hold
  - Key Constraints: Specifying something is a primary key or unique

```
CREATE TABLE customer (
    ssn CHAR(9) PRIMARY KEY,
    cname CHAR(15),
    address CHAR(30),
    city CHAR(10),
    UNIQUE (cname, address, city));
• Attribute constraints: Constraints on the values of attributes
bname char(15) not null
balance int not null, check (balance>= 0)
```



28 / 32

- Integrity Constraints

- *Integrity constraints* are rules that ensure the accuracy and consistency of data within a database. They are conditions that the data must satisfy at all times. This is crucial for maintaining the reliability of the database.
- **Key Constraints:** These are specific types of integrity constraints that ensure uniqueness and identify records within a table. For example, a *primary key* is a unique identifier for each record in a table. In the provided SQL example, `ssn` is defined as the primary key for the `customer` table, meaning no two customers can have the same social security number. Additionally, the `UNIQUE` constraint ensures that the combination of `cname`, `address`, and `city` is unique across all records, preventing duplicate entries with the same customer name, address, and city.

```
CREATE TABLE customer (
    ssn CHAR(9) PRIMARY KEY,
    cname CHAR(15),
    address CHAR(30),
    city CHAR(10),
    UNIQUE (cname, address, city));
```

- **Attribute Constraints:** These constraints apply to individual columns within a table. They specify rules for the values that can be stored in a column. For instance, the `NOT NULL` constraint ensures that a column cannot have a null value, meaning it must always contain data. In the example, `bname` is defined as `NOT NULL`, ensuring that every record must have a value for `bname`. Similarly, the `CHECK` constraint is used to enforce a condition on the values in a column. For example, `balance` must be a non-negative integer, as specified by `check (balance>= 0)`.

```
bname char(15) not null  
balance int not null, check (balance>= 0)
```

## 29 / 32: Integrity Constraints

### Integrity Constraints

- Referential integrity: prevent dangling tuples

```
CREATE TABLE branch(bname CHAR(15) PRIMARY KEY, ...);  
CREATE TABLE loan(..., FOREIGN KEY bname REFERENCES branch);
```

- Can tell the system what to do if a referenced tuple is being deleted



29 / 32

- **Integrity Constraints**

- **Referential integrity: prevent dangling tuples**

- \* *Referential integrity* is a concept in databases that ensures relationships between tables remain consistent. When we have two tables that are related, like a `branch` table and a `loan` table, referential integrity makes sure that any reference to a row in one table corresponds to a valid row in the other table.
    - \* In the example provided, the `branch` table has a primary key `bname`, which uniquely identifies each branch. The `loan` table has a foreign key `bname` that references this primary key. This setup ensures that every loan is associated with a valid branch.
    - \* By enforcing referential integrity, we prevent “dangling tuples,” which are records in the `loan` table that reference a non-existent branch. This is crucial for maintaining data accuracy and consistency.

- **Can tell the system what to do if a referenced tuple is being deleted**

- \* When a referenced tuple (like a branch) is deleted, we need to decide how to handle related records in other tables (like loans). Options include:
      - *Cascade*: Automatically delete or update the related records.
      - *Set Null*: Set the foreign key in related records to `NULL`.
      - *Restrict*: Prevent the deletion if related records exist.
    - \* These options help maintain data integrity and allow for flexible database management.

### Integrity Constraints

- Global Constraints

- Single-table

```
CREATE TABLE branch (... , bcity CHAR(15) , assets INT ,  
CHECK (NOT(bcity = 'Bkln') OR assets>5M))
```

- Multi-table

```
CREATE ASSERTION loan-constraint  
CHECK (NOT EXISTS  
(SELECT* FROM loan AS L WHERE NOT EXISTS  
(SELECT* FROM borrower B, depositor D, account A  
WHERE B.cname = D.cname AND D.acct_no = A.acct_no  
AND L.lno= B.lno)))
```



30 / 32

- **Integrity Constraints**

- Integrity constraints are rules that ensure data accuracy and consistency within a database. They are crucial for maintaining the quality and reliability of the data stored.

- **Global Constraints**

- Global constraints are rules that apply to the entire database rather than just a single table. They help enforce data integrity across multiple tables or the entire database system.

- **Single-table**

- \* The example provided shows a constraint applied to a single table. In this case, the **branch** table has a constraint that checks if the city (**bcity**) is 'Bkln' (Brooklyn), then the **assets** must be greater than 5 million. This ensures that branches in Brooklyn have a minimum asset value, which might be a business rule for financial stability.

- **Multi-table**

- \* Multi-table constraints involve more than one table to enforce data integrity. The example uses an assertion named **loan-constraint** to ensure that every loan in the **loan** table has a corresponding borrower who is also a depositor with an account. This constraint prevents loans from existing without a valid borrower and depositor relationship, ensuring that all loans are properly linked to customer accounts. This is crucial for maintaining the integrity of financial transactions and relationships within the database.

### Additional SQL Constructs

- SELECT subquery factoring
  - To allow assigning a name to a subquery, then use its result by referencing that name

```
WITH temp AS (
    SELECT title, avg(length)
    FROM movies
    GROUP BY year)
SELECT COUNT(*) FROM temp;
```
  - Can have multiple subqueries (multiple WITH clauses)
  - Real advantage is when subquery needs to be referenced multiple times in main select
  - Helps with complex queries, both for readability and maybe performance (can cache subquery results)



31 / 32

- **SELECT subquery factoring**
  - This concept is about using the WITH clause in SQL to create a temporary result set that you can refer to multiple times within your main query. It's like giving a name to a subquery so you can easily use it later.
  - In the example provided, a subquery is named `temp`, which calculates the average length of movies grouped by year. This result can then be used in the main query to count the number of entries in `temp`.
  - **Multiple subqueries:** You can define more than one subquery using multiple WITH clauses. This is useful when you have several intermediate results you want to use in your main query.
  - **Advantages:** This approach is particularly beneficial for complex queries. It improves readability by breaking down the query into understandable parts. Additionally, it might enhance performance because the database can cache the results of the subquery, avoiding redundant calculations. This is especially useful when the subquery is referenced multiple times.

### Another SQL Construct

- SELECT HAVING clause
  - Used in combination with GROUP BY to restrict the groups of returned rows to only those where condition evaluates to true

```
SELECT year, count(*)
    FROM movies
   WHERE year > 1980
  GROUP BY year
 HAVING COUNT(*) > 10;
```

- Difference from WHERE clause is that it applies to summarized group records, and where applies to individual records



32 / 32

- SELECT HAVING clause

- The HAVING clause is a powerful tool in SQL that is used alongside the GROUP BY clause. Its main purpose is to filter groups of data that have been aggregated, allowing you to specify conditions that these groups must meet to be included in the final result set.
- In the provided SQL example, the query is selecting movies released after the year 1980. It groups these movies by their release year and then counts how many movies were released each year. The HAVING clause is then used to filter these groups, ensuring that only years with more than 10 movies are included in the results.
- It's important to note that the HAVING clause is different from the WHERE clause. While WHERE filters individual rows before any grouping takes place, HAVING filters the groups after the aggregation has been performed. This distinction is crucial when working with grouped data in SQL.

---

## Lesson 4.3: Data Storage



### Lesson 4.3: Data Storage

- **Instructor:** Dr. GP Saggese, [gsaggese@umd.edu](mailto:gsaggese@umd.edu)
- **References**
  - Silberschatz et al. 2020, Chap 12, Physical Storage Systems
  - Silberschatz et al. 2020, Chap 13: Data Storage Structures



1 / 17

## 2 / 17: Storage Characteristics

### Storage Characteristics

- **Storage media trade-offs:**
  - Speed of access (e.g., 500-3,500 MB/sec)
  - Cost per data unit (e.g., 50 USD/TB)
  - Medium reliability

- **Volatile vs non-volatile storage**

- *Volatile*: loses contents when power is switched off
- *Non-volatile*: retains contents even after power is switched off

- **Sequential vs random access**

- *Sequential*: read the data contiguously  
`SELECT * FROM employee`
- *Random*: read the data from anywhere at any time  
`SELECT * FROM employee  
WHERE name LIKE '__a__b'`



Commodore 64  
cassette player

- Need to know how data is stored in order to optimize access

- **Storage media trade-offs:**

- When choosing storage media, there are several trade-offs to consider. **Speed of access** refers to how quickly data can be read from or written to the storage. For example, speeds can range from 500 to 3,500 MB per second. Faster access speeds are generally more desirable but can be more expensive.
- **Cost per data unit** is another factor, often measured in terms of dollars per terabyte (e.g., 50 USD/TB). Lower costs are preferable, but they might come with compromises in speed or reliability.
- **Medium reliability** refers to how dependable the storage is over time. More reliable storage is less likely to fail, but it might be more costly or slower.

- **Volatile vs non-volatile storage:**

- *Volatile storage* loses its data when the power is turned off. This includes types like RAM, which are fast but temporary.
- *Non-volatile storage* retains data even when the power is off, such as hard drives or SSDs. This makes it suitable for long-term data storage.

- **Sequential vs random access:**

- *Sequential access* involves reading data in a continuous sequence. This is efficient for operations like reading an entire table, as shown in the example `SELECT * FROM employee`.
- *Random access* allows data to be read from any location at any time, which is useful for queries that need specific data points, like `SELECT * FROM employee WHERE name LIKE '__a__b'`.

- **Need to know how data is stored in order to optimize access:**

- Understanding the characteristics of storage media and access methods is crucial for optimizing data retrieval and storage efficiency. This knowledge helps in making informed

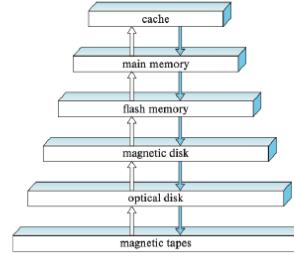
---

decisions about which storage solutions to use based on specific needs and constraints.

### 3 / 17: Storage Hierarchy (by Speed and Cost)

#### Storage Hierarchy (by Speed and Cost)

- **Cache**
  - Fastest, most costly
  - ~MBs on chip
  - DB developers consider cache effects
- **Main memory**
  - Up to 100s of GBs
  - Typically can't store entire DB
  - Volatile
- **Flash memory / SSDs**
  - Less expensive than RAM, more than magnetic disk
  - Non-volatile, random access
- **Magnetic disk**
  - Long-term online storage
  - Non-volatile
- **Optical disk (CD, Blu-ray)**
  - Mainly read-only
- **Magnetic tapes**
  - Backup, archival data
  - Stored long-term, e.g., for legal reasons



SCIENCE ACADEMY

3 / 17

- **Cache**
  - The cache is the fastest type of storage available in a computer system, but it is also the most expensive. It is typically measured in megabytes (MBs) and is located directly on the processor chip. This proximity allows for extremely quick data access, which is crucial for performance. Database developers often need to consider how their applications interact with the cache to optimize speed and efficiency.
- **Main memory**
  - Main memory, or RAM, can store up to hundreds of gigabytes (GBs) of data. However, it is usually not large enough to hold an entire database, especially for large-scale applications. It is also volatile, meaning that it loses its data when the power is turned off, which is a critical consideration for data persistence.
- **Flash memory / SSDs**
  - Flash memory, commonly found in Solid State Drives (SSDs), is less expensive than RAM but more costly than traditional magnetic disks. It is non-volatile, meaning it retains data without power, and offers random access, which allows for faster data retrieval compared to sequential access storage.
- **Magnetic disk**
  - Magnetic disks are used for long-term online storage and are non-volatile, ensuring data is retained without power. They are slower than SSDs but are more cost-effective for storing large amounts of data.
- **Optical disk (CD, Blu-ray)**
  - Optical disks are primarily used for read-only purposes. They are not as commonly used

---

for active data storage due to their slower access speeds and limited rewrite capabilities.

- **Magnetic tapes**

- Magnetic tapes are used for backup and archival purposes. They are ideal for storing data long-term, such as for legal compliance, due to their durability and cost-effectiveness. However, they are sequential-access, meaning data retrieval can be slower compared to other storage types.

## 4 / 17: How Important Is Memory Hierarchy?

### How Important Is Memory Hierarchy?

- **Trade-offs have shifted** over the last 10-15 years

- **Innovations**

- Fast networks, SSDs, large memories
- Data volume is growing rapidly

- **Observations**

- It is faster to access another computer's memory through a network than your own disk
- Cache plays a crucial role
- In-memory databases
  - Data often fits in the memory of a machine cluster
- Disk considerations are less important
  - Disks still store most data today

- **Algorithms depend on available technology**



4 / 17

- **Trade-offs have shifted** over the last 10-15 years

- In the past, the focus was on optimizing for slower, more limited memory resources. However, with technological advancements, the balance between speed, cost, and capacity has changed significantly.

- **Innovations**

- **Fast networks, SSDs, large memories:** These advancements have transformed how we handle data. Fast networks allow quick data transfer between machines, SSDs provide faster data access compared to traditional hard drives, and larger memory capacities enable more data to be stored and processed in-memory.
- **Data volume is growing rapidly:** As data generation increases, the ability to efficiently manage and process large datasets becomes crucial.

- **Observations**

- **It is faster to access another computer's memory through a network than your own disk:** This highlights the speed advantage of networked memory access over traditional disk access, emphasizing the importance of network speed and memory capacity.
- **Cache plays a crucial role:** Caches help speed up data access by storing frequently

---

accessed data closer to the processor, reducing the need to access slower memory layers.

- **In-memory databases:** These databases store data in the main memory rather than on disk, allowing for faster data retrieval and processing. With large memory capacities, data can often fit within a machine cluster's memory.
- **Disk considerations are less important:** Although disks still store most data, their role in immediate data processing has diminished due to faster alternatives like SSDs and in-memory processing.
- **Algorithms depend on available technology**
  - The design and efficiency of algorithms are influenced by the hardware they run on. As technology evolves, algorithms are adapted to leverage new capabilities, such as faster memory access and larger storage capacities.

## Magnetic Disks / SSDs

- This section likely discusses the differences between traditional magnetic disks and modern SSDs, focusing on their impact on data storage and retrieval speeds.

## 5 / 17: Connecting Disks to a Server

### Connecting Disks to a Server

- **Disks** (magnetic and SSDs) connect to computers via:
  - High-speed bus interconnections
  - High-speed networks
- **High-speed interconnections**
  - Serial ATA (SATA)
  - Serial Attached SCSI (SAS)
  - NVMe (Non-Volatile Memory Express)
- **High-speed networks**
  - Storage Area Network (SAN): iSCSI, Fibre Channel, InfiniBand
  - **Network Attached Storage (NAS)**
    - Provides a file-system interface (e.g., NFS)
    - Cloud storage: Data stored in the cloud, accessed via API, object store, high latency

- **Disks** (magnetic and SSDs) connect to computers via:
  - Disks, whether they are traditional magnetic hard drives or modern Solid State Drives (SSDs), need to be connected to a computer to store and retrieve data. This connection can be made through two main methods: high-speed bus interconnections and high-speed networks. These methods ensure that data can be transferred quickly and efficiently between the disk and the computer.
- **High-speed interconnections**
  - **Serial ATA (SATA):** This is a common interface used to connect hard drives and SSDs

to the motherboard of a computer. It is known for its reliability and is widely used in personal computers.

- **Serial Attached SCSI (SAS)**: SAS is similar to SATA but is typically used in enterprise environments where higher performance and reliability are required.
- **NVMe (Non-Volatile Memory Express)**: NVMe is a newer protocol designed specifically for SSDs. It provides faster data transfer speeds by connecting directly to the computer's PCIe bus, making it ideal for high-performance applications.

- **High-speed networks**

- **Storage Area Network (SAN)**: SANs are specialized networks that provide access to consolidated, block-level data storage. They use protocols like iSCSI, Fibre Channel, and InfiniBand to connect storage devices to servers, allowing for high-speed data transfer and centralized storage management.
- **Network Attached Storage (NAS)**: NAS devices provide a file-system interface, such as NFS (Network File System), allowing multiple users and devices to access shared storage over a network. NAS is often used for file sharing and backup solutions.
  - \* *Cloud storage*: This refers to storing data in the cloud, which can be accessed via APIs. Cloud storage is typically an object store, meaning it stores data as objects rather than files or blocks. While it offers scalability and accessibility, it often comes with higher latency compared to local storage solutions.

## 6 / 17: Magnetic Disks

### Magnetic Disks

- **1956**

- IBM RAMAC
- 24" platters
- 5 million characters



- **1956**

- **IBM RAMAC**: This was the first computer to use a hard disk drive (HDD) for storage. The IBM RAMAC (Random Access Method of Accounting and Control) was a groundbreaking development in data storage technology. Before this, data was stored on punch

cards or magnetic tapes, which were much slower and less efficient.

- **24” platters:** The storage medium in the IBM RAMAC consisted of large, 24-inch diameter platters. These platters were coated with a magnetic material that allowed data to be written and read by a magnetic head. The size of these platters highlights how early technology required large physical space to store relatively small amounts of data.
- **5 million characters:** The storage capacity of the IBM RAMAC was about 5 million characters, which is roughly equivalent to 5 megabytes. While this seems minuscule by today’s standards, it was a significant amount of storage at the time and represented a major advancement in the ability to store and retrieve data quickly.

The images on the slide likely depict the IBM RAMAC and its components, providing a visual context for understanding the scale and design of early magnetic disk storage systems.

## 7 / 17: Magnetic Disks

### Magnetic Disks

#### • 1979

- Seagate
- 5MB



#### • 1998

- Seagate
- 47GB



#### • 2006

- Western Digital
- 500GB



7 / 17

#### • 1979

- **Seagate:** This was the year when Seagate, a major player in the storage industry, introduced one of the first magnetic disks for personal computers.
- **5MB:** The capacity of this disk was 5 megabytes, which was considered substantial at the time. To put it in perspective, 5MB is roughly equivalent to a single high-quality photo today. This highlights how storage technology has evolved over the years.

#### • 1998

- **Seagate:** Nearly two decades later, Seagate continued to innovate in the field of magnetic storage.
- **47GB:** By 1998, the capacity of magnetic disks had increased dramatically to 47 gigabytes. This leap in storage capacity reflects the rapid advancements in technology and

the growing demand for more data storage as computers became more integral to daily life.

- **2006**

- **Western Digital:** Another key player in the storage industry, Western Digital, made significant contributions to the development of magnetic disks.
- **500GB:** By 2006, the capacity had reached 500 gigabytes. This increase was driven by the need to store more complex data, such as videos and large software applications, as digital technology became more sophisticated and widespread.

These points illustrate the exponential growth in storage capacity over the years, driven by technological advancements and increasing data demands.

## 8 / 17: Magnetic Disks: Components

### Magnetic Disks: Components

- **Platters**

- Rigid metal with magnetic material on both surfaces
- Spins at 5400 or 7200 RPM
- Tracks subdivided into *sectors* (smallest unit read/written)

- **Read-write heads**

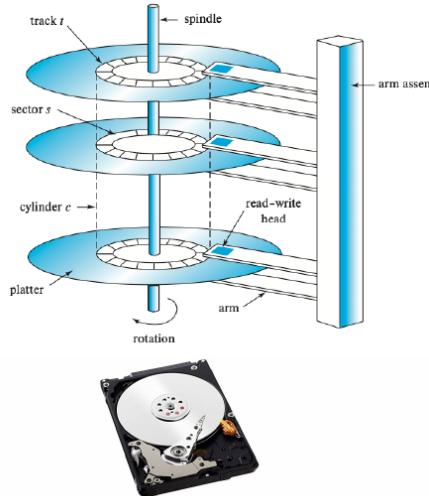
- Read/write data magnetically
- Spinning creates a cushion maintaining heads a few microns from the surface
- A *cylinder* is the i-th track of all platters (read/written together)

- **Arm**

- Moves all heads along the disks

- **Disk controller**

- Accepts commands to read/write a sector
- Operates arm/heads
- Remaps bad sectors to a different location



SCIENCE  
ACADEMY

8 / 17

- **Platters**

- Platters are the core components of a magnetic disk. They are made of a rigid metal material and are coated with a magnetic layer on both sides. This magnetic coating is crucial because it allows data to be stored magnetically.
- These platters spin at high speeds, typically 5400 or 7200 revolutions per minute (RPM). The speed of the spin affects how quickly data can be read from or written to the disk.
- The surface of each platter is organized into concentric circles called *tracks*. Each track is further divided into smaller sections known as *sectors*. A sector is the smallest unit of data that can be read or written on the disk.

- **Read-write heads**

- These are the components responsible for reading data from and writing data to the platters. They do this by detecting and altering the magnetic fields on the platter surfaces.

- As the platters spin, a cushion of air is created, which keeps the read-write heads just a few microns above the surface of the platters. This is crucial to prevent damage and ensure accurate data reading and writing.
- A *cylinder* refers to the collection of tracks located at the same position on each platter. All tracks in a cylinder can be accessed simultaneously, which can improve data access speed.
- **Arm**
  - The arm is a mechanical component that moves the read-write heads across the platters. It ensures that the heads can access different tracks on the platters as needed.
- **Disk controller**
  - This is the electronic component that manages the operation of the disk. It receives commands from the computer to read or write data and then controls the movement of the arm and the operation of the read-write heads.
  - The disk controller also handles error management, such as remapping bad sectors. If a sector becomes unreadable, the controller can redirect data to a different, healthy sector, ensuring data integrity.

## 9 / 17: Magnetic Disks: Current Specs

### Magnetic Disks: Current Specs

- **Capacity**
  - 10 terabytes and more
- **Access time**
  - Time to start reading data
  - Seek time
    - Move arm across cylinders (2-20ms)
  - Rotational latency time
    - Wait for sector access (4-12ms)
- **Data-transfer rate**
  - Transfer begins once data is reached
  - Transfer rate: 50-200MB/sec
  - Sector (disk block): logical unit of storage (4-16KB)
  - Sequential access: blocks on same or adjacent tracks
  - Random access: each request requires a seek
    - IOPS: number of random single block accesses per second (50-200 IOPS)
- **Reliability**
  - Mean time to failure (MTTF): average time system runs without failure
  - HDD lifespan: ~5 years



- **Capacity**
  - Modern magnetic disks can store a *huge* amount of data, often 10 terabytes or more. This makes them suitable for applications that require storing large datasets, such as big data analytics and machine learning.
- **Access time**
  - Access time is the duration it takes for a disk to start reading data. It consists of two main components:

- \* **Seek time:** This is the time it takes for the disk's read/write arm to move across the disk's cylinders to the correct position. It typically ranges from 2 to 20 milliseconds.
- \* **Rotational latency time:** Once the arm is in position, the disk must wait for the correct sector to rotate under the read/write head. This waiting time usually falls between 4 to 12 milliseconds.
- **Data-transfer rate**
  - Once the data is located, the transfer begins. The rate at which data is transferred can vary from 50 to 200 megabytes per second.
  - A sector, or disk block, is the smallest logical unit of storage on a disk, typically ranging from 4 to 16 kilobytes.
  - **Sequential access** involves reading blocks that are on the same or adjacent tracks, which is faster.
  - **Random access** requires moving the read/write head to different locations for each request, which is slower. The performance of random access is often measured in IOPS (Input/Output Operations Per Second), with typical values ranging from 50 to 200 IOPS.
- **Reliability**
  - The reliability of a hard disk drive (HDD) is often measured by the Mean Time to Failure (MTTF), which is the average time the system operates without failure.
  - The typical lifespan of an HDD is around 5 years, after which the risk of failure increases. This is an important consideration for data storage solutions, especially in critical applications.

## 10 / 17: Accessing Data Speed

### Accessing Data Speed

- **Random data transfer rates**
  - Time to read a random sector
  - It has 3 components
    - *Seek time:* Time to seek to the track (~4-10ms)
    - *Rotational latency:* Waiting for the sector to get under the head (~4-11ms)
    - *Transfer time:* Time to transfer the data (Very low)
  - About 10ms per access
    - Randomly accessed blocks
    - 100 block transfers ( $100/\text{sec} \times 4 \text{ KB/block} = 400 \text{ KB/s}$ )
- **Serial data transfer rates**
  - Data transfer rate without seek
  - 30-50MB/s to 200MB/s
- **Seeks are bad!**

- **Random data transfer rates**
  - When we talk about reading data randomly from a storage device, we're referring to

accessing data that isn't stored in a continuous sequence. This means the device has to jump around to different locations to get the data.

- **Time to read a random sector:** This is the time it takes to access a specific piece of data on a storage device. It involves three main components:
  - \* **Seek time:** This is the time it takes for the read/write head of a hard drive to move to the correct track where the data is stored. It usually takes between 4 to 10 milliseconds.
  - \* **Rotational latency:** Once the head is on the right track, it has to wait for the disk to spin around so that the correct sector is under the head. This can take another 4 to 11 milliseconds.
  - \* **Transfer time:** This is the actual time it takes to move the data from the disk to the computer. This time is very short compared to the other two components.
- In total, accessing data randomly can take about 10 milliseconds per access. If you are accessing 100 blocks of data per second, each 4 KB in size, you can transfer about 400 KB per second.
- **Serial data transfer rates**
  - When data is stored in a continuous sequence, it can be read much faster because the device doesn't have to jump around. This is called serial data transfer.
  - Without the need for seeking, data can be transferred at rates ranging from 30-50 MB/s to as high as 200 MB/s.
- **Seeks are bad!**
  - The process of seeking, or moving the read/write head to the correct track, is time-consuming and slows down data access. This is why random access is much slower than serial access. Reducing the need for seeks can significantly improve data transfer speeds.

## 11 / 17: Solid State Disk (SSD)

### Solid State Disk (SSD)

- Mainstream around 2000s
  - Better than HDD for all metrics, more expensive per GB
  - Like non-volatile RAM (NAND and NOR)
- **Capacity**
  - 250-500 GB (vs 1-10 TB for HDD)
- **Access time**
  - Latency for random access is 1,000x smaller than HDD
    - E.g., 20-100 us (vs 10 ms for HDDs)
  - Multiple random requests (e.g., 32) in parallel
  - 10,000 IOPS (vs 50/200 for HDDs)
  - Requires reading an entire “page” of data (typically 4KB)
    - Equivalent to a block in magnetic disks
- **Data-transfer rate**
  - 1 GB/s (vs 200 MB/s for HDD)
  - Typically limited by interface speed
  - Reads and writes ~500 MB/s for SATA and 2-3 GB/s for NVMe
  - Lower power consumption than HDDs
  - Writing to SSD is slower than reading (~2-3x)
    - Requires erasing all pages in the block
- **Reliability**



SCIENCE  
ACADEMY

Limit to how many times a flash page can be erased (~1M times)

- 
- **Solid State Disk (SSD)**
    - *Mainstream around 2000s*: SSDs became popular in the 2000s as they offered significant improvements over traditional Hard Disk Drives (HDDs). They are faster, more reliable, and consume less power, but they are more expensive per gigabyte.
    - *Like non-volatile RAM*: SSDs use NAND and NOR flash memory, which retains data even when the power is off, similar to non-volatile RAM.
  - **Capacity**
    - SSDs typically offer capacities ranging from 250 to 500 GB, which is smaller compared to HDDs that can range from 1 to 10 TB. This is a trade-off for the speed and reliability SSDs provide.
  - **Access time**
    - SSDs have significantly lower latency for random access, about 1,000 times smaller than HDDs. This means they can access data much faster, with latencies around 20-100 microseconds compared to 10 milliseconds for HDDs.
    - They can handle multiple random requests simultaneously, enhancing performance with up to 10,000 Input/Output Operations Per Second (IOPS), whereas HDDs manage only 50 to 200 IOPS.
    - SSDs read data in “pages” (typically 4KB), similar to blocks in magnetic disks, which is efficient for accessing data quickly.
  - **Data-transfer rate**
    - SSDs have a data-transfer rate of about 1 GB/s, significantly higher than the 200 MB/s typical for HDDs. However, this is often limited by the interface speed.
    - For SATA interfaces, read and write speeds are around 500 MB/s, while NVMe interfaces can reach 2-3 GB/s.
    - SSDs consume less power than HDDs, making them more energy-efficient.
    - Writing to SSDs is slower than reading because it requires erasing all pages in a block before writing new data, which can be 2-3 times slower.
  - **Reliability**
    - SSDs have a finite number of write/erase cycles, with each flash page capable of being erased approximately 1 million times before it may fail. This is an important consideration for the longevity of SSDs.
  - **RAID**
    - The slide hints at RAID, which stands for Redundant Array of Independent Disks, a technology used to improve performance and reliability by combining multiple disk drives into a single unit.

### RAID

- **RAID** = Redundant Array of Independent Disks
- **Problem**
  - Storage capacity is growing exponentially
  - Data-storage needs are growing even faster
  - There is a need for more disks
  - Mean Time To Failure (MTTF) between disk failures is shrinking (e.g., days)
    - A single data copy leads to an unacceptable frequency of data loss
- **Observations**
  - Disks are cheap
  - Failures are costly
  - Use extra disks for reliability
    - Store data redundantly
    - Data survives disk failure
- **Goal**
  - Present a logical view of a large, reliable disk from many unreliable disks
  - Different RAID levels balance reliability and performance



- **RAID** = Redundant Array of Independent Disks
  - RAID is a technology that combines multiple disk drives into a single unit to improve data reliability and performance. It stands for Redundant Array of Independent Disks, emphasizing the use of multiple disks to store data redundantly.
- **Problem**
  - **Storage capacity is growing exponentially:** As technology advances, the amount of data we generate and need to store is increasing rapidly.
  - **Data-storage needs are growing even faster:** The demand for storing more data is outpacing the growth in storage capacity, creating a need for more efficient storage solutions.
  - **There is a need for more disks:** To meet the growing data demands, more disks are required, which can lead to increased complexity and potential for failure.
  - **Mean Time To Failure (MTTF) between disk failures is shrinking (e.g., days):** As we use more disks, the likelihood of a disk failing increases, reducing the average time between failures.
    - \* *A single data copy leads to an unacceptable frequency of data loss:* Relying on a single copy of data is risky because if the disk fails, the data could be lost.
- **Observations**
  - **Disks are cheap:** The cost of individual disks is relatively low, making it feasible to use multiple disks for redundancy.
  - **Failures are costly:** The cost of data loss or downtime due to disk failure can be significant, justifying the investment in redundancy.
  - **Use extra disks for reliability:** By using additional disks, data can be stored redundantly, ensuring that it remains accessible even if one disk fails.

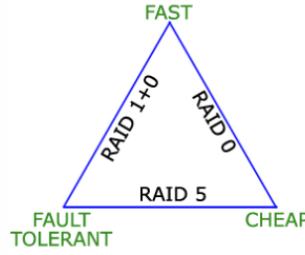
- \* *Store data redundantly:* Data is duplicated across multiple disks to prevent loss.
- \* *Data survives disk failure:* If one disk fails, the data can still be accessed from another disk.
- **Goal**
  - **Present a logical view of a large, reliable disk from many unreliable disks:** RAID aims to create the illusion of a single, large, and reliable storage system by combining multiple less reliable disks.
  - **Different RAID levels balance reliability and performance:** Various RAID configurations offer different trade-offs between data reliability and system performance, allowing users to choose the best option for their needs.

## 13 / 17: Improve Reliability / Performance with RAID

### Improve Reliability / Performance with RAID

- **Reliability**

- Use redundancy
  - Store data multiple times: e.g., mirroring
  - Reconstruct data if a disk fails
  - Increase Mean Time To Failure (MTTF)
- Assume independence of disk failure
  - Consider power failures and natural disasters
  - Aging disks increase failure probability



- **Performance**

- Parallel access to multiple disks: e.g., mirroring, increases read requests
- Stripe data across multiple disks: Increases transfer rate

- **Reliability**

- *Use redundancy:* This means having extra copies of your data. By storing data in multiple places, like with mirroring, you can protect against data loss if one disk fails. This is like having a backup plan.
  - \* *Store data multiple times:* Mirroring is a common method where data is copied exactly onto another disk. If one disk fails, the data is still safe on the other disk.
  - \* *Reconstruct data if a disk fails:* If a disk does fail, the system can rebuild the lost data using the redundant copies.
  - \* *Increase Mean Time To Failure (MTTF):* This is a measure of how long you can expect the system to run before a failure happens. Redundancy helps increase this time.
- *Assume independence of disk failure:* This means that the failure of one disk doesn't necessarily mean others will fail too. However, it's important to consider factors like

power failures or natural disasters that could affect multiple disks at once.

- \* *Aging disks increase failure probability:* As disks get older, they are more likely to fail, so it's important to monitor their health.

- **Performance**

- *Parallel access to multiple disks:* By accessing multiple disks at the same time, systems can handle more read requests. This is especially useful in setups like mirroring, where data is available on multiple disks.
- *Stripe data across multiple disks:* This technique involves spreading data across several disks. It helps increase the speed at which data can be read or written, improving the overall transfer rate. This is like having multiple lanes on a highway, allowing more cars to travel at once.

## 14 / 17: Error Correction Codes

### Error Correction Codes

- = a technique used for controlling errors in data transmission over unreliable communication channels
- **Idea:**
  - the sender encodes the message in a redundant way
  - the receiver can detect errors and correct (a limited number of) errors
- 1940-1960s: Hamming, Reed-Solomon, Shannon, Viterbi
- E.g., triple redundancy
  - Send the same bit 3 times, receiver does majority voting
  - Detect and correct one bit errors
- E.g. parity bit
  - Add an extra bit representing the number of 1s
  - Detect (but not correct) one bit errors



Triplet received	Interpreted as
000	0 (error-free)
001	0
010	0
100	0
111	1 (error-free)
110	1
101	1
011	1

7 bits of data	(count of 1-bits)	8 bits including parity	
		even	odd
0000000	0	00000000	00000001
1010001	3	10100011	10100010
1101001	4	11010010	11010011
1111111	7	11111111	11111110



14 / 17

- **Error Correction Codes** are essential for ensuring data integrity during transmission over unreliable channels. These techniques help in identifying and fixing errors that may occur when data is sent from one place to another.

- **Idea:**

- The sender encodes the message with extra information, known as redundancy, which helps the receiver identify and correct errors. This redundancy allows the receiver to detect errors and, in some cases, correct them without needing the sender to resend the data.

- **Historical Context:**

- Between the 1940s and 1960s, significant advancements were made by researchers like

Hamming, Reed-Solomon, Shannon, and Viterbi. These pioneers developed foundational techniques that are still in use today.

- **Examples:**

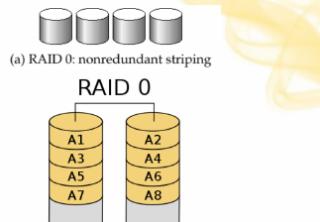
- *Triple Redundancy*: This method involves sending each bit of data three times. The receiver uses majority voting to determine the correct bit, allowing it to detect and correct single-bit errors.
- *Parity Bit*: This simpler method adds an extra bit to the data, which indicates whether the number of 1s in the data is even or odd. While it can detect single-bit errors, it cannot correct them.
- The images on the right likely illustrate these concepts visually, showing how redundancy and error detection/correction work in practice.

## 15 / 17: RAID Levels

### RAID Levels

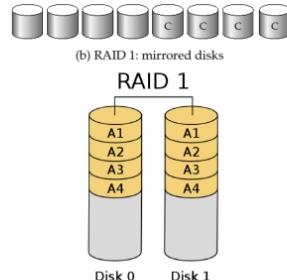
- **RAID 0: Striping / no redundancy**

- Array of independent disks
- Same access time
- Increase transfer rate



- **RAID 1: Mirroring**

- Copy of disks
- If one disk fails, you have data copy
  - Double redundancy like ECC
- Parallel access to multiple disks
- Reads
  - Can go to either disk
  - Same access time
  - Increase read latency with same transfer rate
  - Same read latency with increased transfer rate
- Writes
  - Write to both disks



- **RAID 0: Striping / no redundancy**

- *Array of independent disks*: RAID 0 involves spreading data across multiple disks without any redundancy. This means that data is divided into blocks and each block is written to a separate disk.
- *Same access time*: Since data is distributed evenly, each disk can be accessed simultaneously, leading to uniform access times.
- *Increase transfer rate*: By using multiple disks, RAID 0 can significantly increase the data transfer rate because multiple disks can be read or written to at the same time.

- **RAID 1: Mirroring**

- *Copy of disks*: RAID 1 duplicates the same data on two or more disks. This means that if one disk fails, the data is still safe on the other disk.

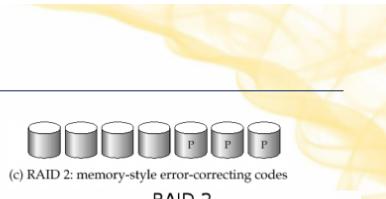
- If one disk fails, you have data copy: This redundancy is similar to error-correcting code (ECC) in that it provides a backup in case of failure.
- Parallel access to multiple disks: Both disks can be accessed at the same time, which can improve performance.
- Reads:
  - \* Can go to either disk: Data can be read from either disk, which can balance the load and improve read performance.
  - \* Same access time: Access time remains consistent because data is available on both disks.
  - \* Increase read latency with same transfer rate: While read latency can improve, the transfer rate remains the same because data is mirrored, not striped.
  - \* Same read latency with increased transfer rate: The redundancy allows for consistent read times while potentially increasing the transfer rate due to parallel reads.
- Writes:
  - \* Write to both disks: Every write operation is duplicated on both disks, ensuring data integrity but potentially slowing down write operations compared to RAID 0.

## 16 / 17: RAID Levels

### RAID Levels

- **RAID 2: Memory-style error correction**

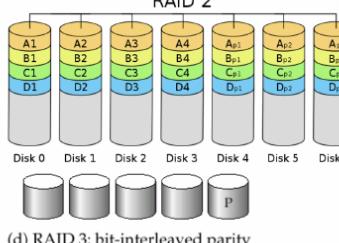
- Use extra bits to reconstruct data (like ECC in RAM)
- Trade-off error detection and recovery levels



RAID 2

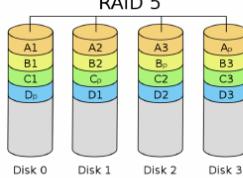
- **RAID 3: Interleaved parity**

- One disk contains parity for main data disks
- Handle single disk failure
- Little overhead (only 25%)



- **RAID 5: Block-interleaved distributed parity**

- Distributed parity blocks instead of bits



- **RAID 2: Memory-style error correction**

- RAID 2 uses a method similar to error-correcting code (ECC) in RAM. This means it adds extra bits to the data to help detect and correct errors. This is useful for ensuring data integrity, but it can be complex and costly because it requires synchronized spinning of all disks.
- The trade-off here is between the level of error detection and recovery you want and the cost and complexity of implementing it. RAID 2 is not commonly used today because

other RAID levels offer better performance and simpler implementations.

- **RAID 3: Interleaved parity**

- In RAID 3, one disk is dedicated to storing parity information, which is used to recover data if one of the main data disks fails. This setup allows for the recovery of data from a single disk failure.
- The overhead is relatively low, around 25%, because only one additional disk is needed for parity. However, RAID 3 can be limited by the single parity disk, which can become a bottleneck during data recovery.

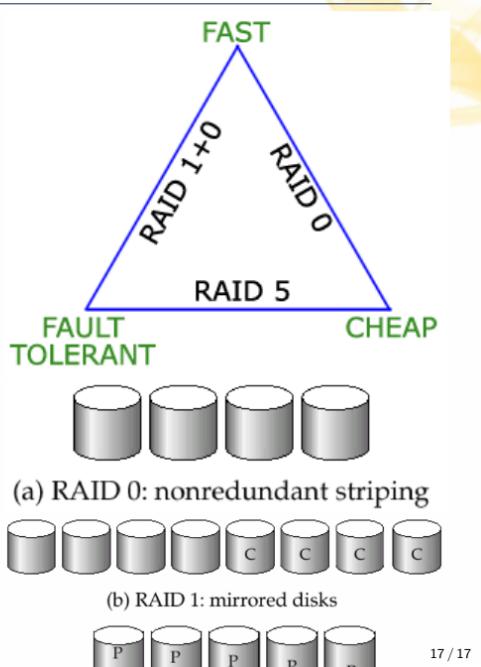
- **RAID 5: Block-interleaved distributed parity**

- Unlike RAID 3, RAID 5 distributes parity blocks across all disks rather than storing them on a single disk. This distribution helps balance the load and improves performance, especially during read operations.
- RAID 5 is popular because it offers a good balance between performance, storage efficiency, and fault tolerance. It can handle a single disk failure, but rebuilding data after a failure can be time-consuming, especially with large disks.

## 17 / 17: Choosing a RAID Level

### Choosing a RAID Level

- Main choice between RAID 0, RAID 1, and RAID 5
- **RAID 0 (striping)**
  - Better performance, no reliability
- **RAID 1 (mirroring)**
  - Better performance and reliability
  - High cost
  - E.g., to write a single block
    - RAID 1: 2 block writes
    - RAID 5: 2 block reads, 2 block writes
  - Preferred for high update rate, small data (e.g., log disks)
- **RAID 5 (interleaved parity)**
  - Lower storage cost
  - Preferred for low update rate, large data (e.g., analytics)



- Choosing a RAID Level ::::columns ::::{.column width=50%}
- When deciding on a RAID level, the main options are **RAID 0**, **RAID 1**, and **RAID 5**. Each has its own strengths and weaknesses, and the choice depends on your specific needs for performance, reliability, and cost.
- **RAID 0 (striping)**
  - This setup improves performance by spreading data across multiple disks, allowing for faster read and write speeds. However, it offers no data redundancy, meaning if one disk

---

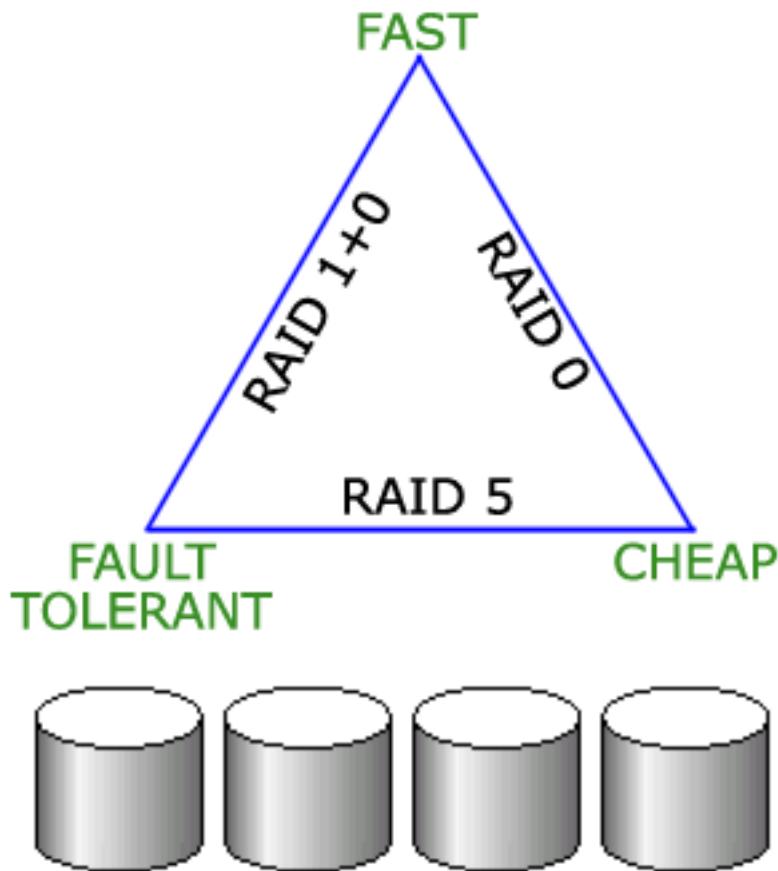
fails, all data is lost.

- **RAID 1 (mirroring)**

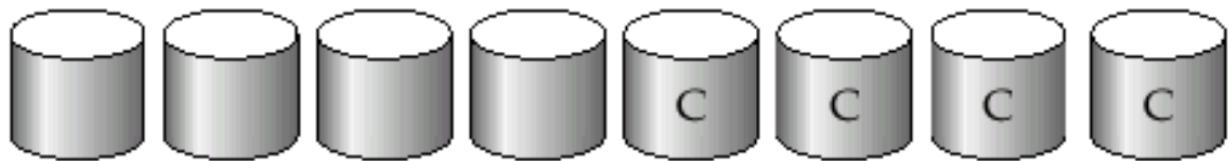
- Provides both improved performance and data reliability by duplicating data on two disks. This means if one disk fails, the data is still safe on the other. The downside is the higher cost since you need double the storage capacity. For example, writing a single block requires writing it to both disks. RAID 1 is ideal for systems with high update rates and smaller data sizes, like log disks.

- **RAID 5 (interleaved parity)**

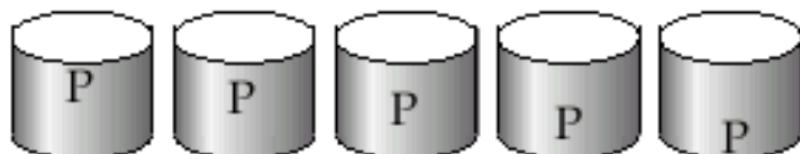
- Offers a balance between cost and reliability by using parity information to recover data in case of a disk failure. It requires fewer disks than RAID 1 for the same amount of data, making it more cost-effective. RAID 5 is best suited for environments with low update rates and large data sizes, such as data analytics. :::: :::{.column width=50%}



(a) RAID 0: nonredundant striping



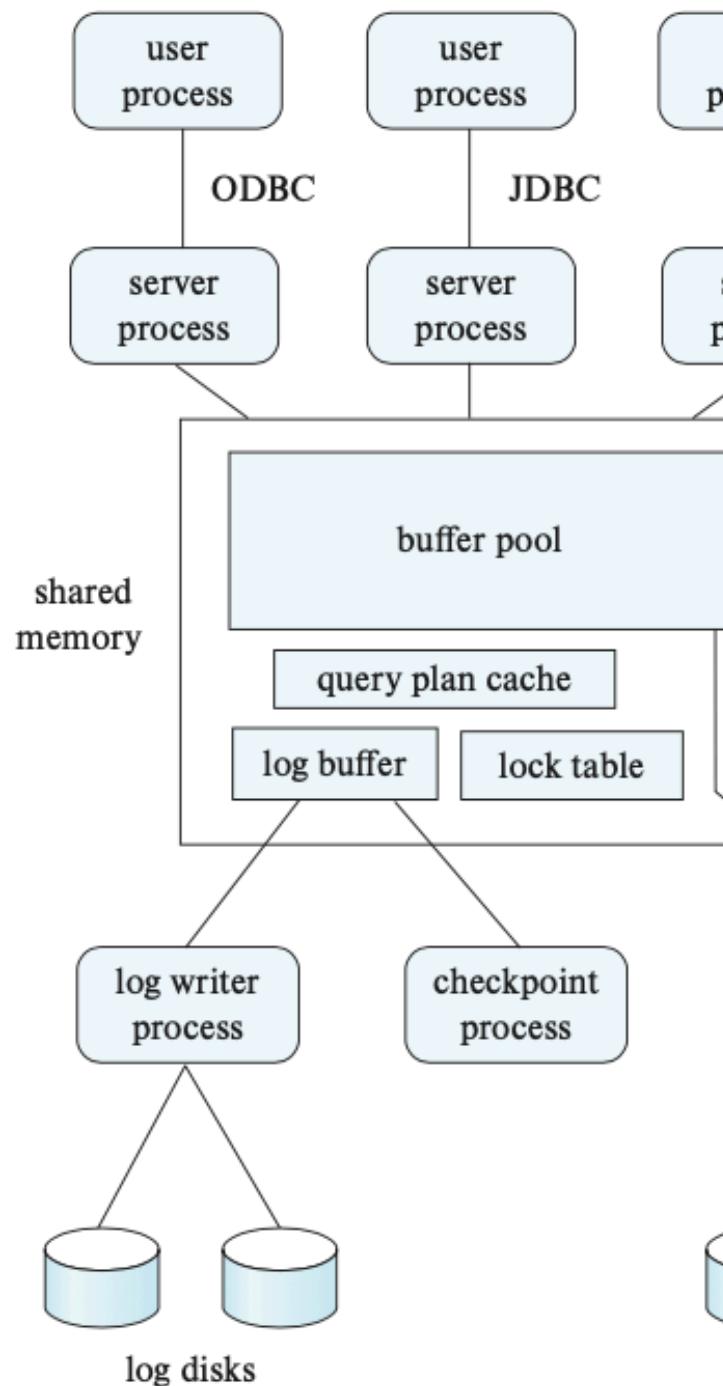
(b) RAID 1: mirrored disks



## (f) RAID 5: block-interleaved distributed parity

::: :: - DB Internals :::columns :::{.column width=60%} - **User processes** - These are the actions initiated by users to interact with the database, such as querying or updating data.

- **Server processes**
  - These processes handle the commands from user processes, executing the necessary database operations.
- **Process monitor process**
  - This process oversees the database operations, ensuring everything runs smoothly and recovering from any failures that occur.
- **Lock manager process**
  - Manages access to data by granting and releasing locks, and it also detects deadlocks to prevent system hang-ups.
- **Database writer process**
  - Continuously writes modified data from memory to disk to ensure data persistence.
- **Log writer process**
  - Records changes to the database in a log, which is crucial for data recovery in case of a failure.
- **Checkpoint process**
  - Periodically saves the current state of the database to minimize data loss during a crash.
- **Shared memory**
  - This is a common area where data is stored temporarily for quick access, including the buffer pool, lock table, log buffer, and caches. It uses mutual exclusion locks to protect



data integrity. :::: :::: {.column width=40%}

:::: :::

- DB Internals ::::columns :::: {.column width=50%}

- **Query Processing Engine**

- This component is responsible for executing user queries. It determines the sequence of pages to be accessed in memory and processes the data to produce the desired results.

- **Buffer Manager**

- Manages the transfer of data pages between disk and memory, optimizing the use of

---

limited memory resources to ensure efficient data access.

- **Storage hierarchy**

- Organizes data by mapping tables to files and tuples to disk blocks, facilitating efficient data retrieval and storage management. :::: :::: { .column width=50% }

```
digraph SystemArchitecture {
    graph [rankdir=TB, splines=ortho, nodesep=0.5, ranksep=0.8];
    node [fontname="Helvetica", fontsize=14, shape=box];
    edge [penwidth=2, color=blue, arrowsize=1.2];
    node [style=filled, fillcolor=yellow, width=1.5, height=0.5];
    user_query [label="user\nquery"];
    node [style="bold, rounded", color=blue, fillcolor=white, penwidth=2, width=3.5, height=0.5];
    query_engine [label="Query Processing Engine"];
    node [style=filled, fillcolor="cadetblue", width=1.5, height=0.5];
    results [label="results"];
    node [style=filled, fillcolor=yellow, width=1.5, height=0.5];
    page_requests [label="page\nrequests"];
    node [style="bold, rounded", color=blue, fillcolor=white, penwidth=2, width=3.5, height=0.5];
    buffer_manager [label="Buffer Manager"];
    node [style=filled, fillcolor="cadetblue", width=1.5, height=0.5];
    pointers [label="pointers\npto pages"];
    node [style=filled, fillcolor=yellow, width=1.5, height=0.5];
    block_requests [label="block\nrequests"];
    node [style="bold, rounded", color=blue, fillcolor=white, penwidth=2, width=3.5, height=0.5];
    space_management [label="Space Management on\nPersistent Storage"];
    node [style=filled, fillcolor="cadetblue", width=1.5, height=0.5];
    data [label="data"];
    { rank=same; user_query; results; }
    { rank=same; query_engine; }
    { rank=same; page_requests; pointers; }
    { rank=same; buffer_manager; }
    { rank=same; block_requests; data; }
    { rank=same; space_management; }
    user_query -> query_engine [style=invis];
    results -> query_engine [style=invis];
    page_requests -> buffer_manager [style=invis];
    pointers -> buffer_manager [style=invis];
    block_requests -> space_management [style=invis];
    data -> space_management [style=invis];
    user_query -> query_engine [arrowhead=normal, constraint=false];
    query_engine -> results [arrowhead=normal, constraint=false];
    page_requests -> buffer_manager [arrowhead=normal, constraint=false];
    buffer_manager -> pointers [arrowhead=normal, constraint=false];
    block_requests -> space_management [arrowhead=normal, constraint=false];
    space_management -> data [arrowhead=normal, constraint=false];
    query_engine -> page_requests [style=invis, weight=10];
    buffer_manager -> block_requests [style=invis, weight=10];
}
```

---

... :)

---

## Lesson 5.1: NoSQL Databases



UMD DATA605 - Big Data Systems

### Lesson 5.1: NoSQL Databases

- **Instructor:** Dr. GP Saggese, [gsaggese@umd.edu](mailto:gsaggese@umd.edu)
- **References:**
  - Online tutorials
  - Silberschatz: Chap 10.2
  - Seven Databases in Seven Weeks, 2e

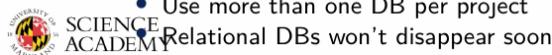


1 / 17

## 2 / 17: From SQL to NoSQL

### From SQL to NoSQL

- **DBs are central tools to big data**
  - New applications, data/storage constraints
  - ~2000s NoSQL “movement” started
    - Initially “No SQL” → then “Not Only SQL”
- **Different DB types make different trade-offs**
  - Different worldviews
  - Schema vs schema-less
  - Rich vs fast query ability
  - Strong consistency (ACID), weak, eventual consistency
  - APIs (SQL, JS, REST)
  - Horizontal vs vertical scaling, sharding, replication
  - Indexing vs no indexing
  - Tuned for reads or writes, control over tuning
- **User base/applications have expanded**
  - Postgres + Mongo cover 99% of use cases
  - Data scientists/engineers need familiarity with both
  - “Which DB solves my problem best?”
- **Polyglot model**
  - Use more than one DB per project



2 / 17

- **DBs are central tools to big data**
  - Databases (DBs) are crucial for managing and analyzing large datasets, which is essential in big data applications. As new applications emerged and data storage needs evolved, traditional databases faced limitations. Around the 2000s, the NoSQL movement began, initially standing for “No SQL” as a rejection of traditional SQL databases. However, it evolved to mean “Not Only SQL,” indicating a broader approach that includes both SQL and NoSQL databases.
- **Different DB types make different trade-offs**
  - Different databases are designed with different priorities and trade-offs. Some focus on having a fixed structure (schema) while others are more flexible (schema-less). There are trade-offs between having rich query capabilities and fast query performance. Consistency models vary, with some databases ensuring strong consistency (ACID properties) and others offering weaker or eventual consistency. Databases also differ in their scaling approaches (horizontal vs. vertical), data distribution methods (sharding, replication), and indexing capabilities. Some are optimized for read-heavy workloads, while others are better for write-heavy tasks, and users can often tune these settings.
- **User base/applications have expanded**
  - The range of applications and users for databases has grown significantly. Databases like Postgres and MongoDB are versatile enough to cover the vast majority of use cases. It’s important for data scientists and engineers to be familiar with both types to choose the best solution for their specific problem. The key question is often, “Which database solves my problem best?”
- **Polyglot model**
  - The polyglot model involves using multiple types of databases within a single project

---

to leverage the strengths of each. While NoSQL databases have gained popularity, relational databases are still widely used and are not expected to disappear anytime soon. This approach allows for more flexibility and efficiency in handling diverse data needs.

## 3 / 17: Issues with Relational Dbs

### Issues with Relational Dbs

- **Relational DBs have drawbacks**
  1. Application-DB impedance mismatch
  2. Schema flexibility
  3. Consistency in distributed set-up
  4. Limited scalability
- For each drawback a slide with:
  - **Problem**
  - **Solutions**
    - Within relational SQL paradigm
    - With NoSQL approach



3 / 17

- **Issues with Relational Dbs**
- **Relational DBs have drawbacks**
  - **Application-DB impedance mismatch**
    - \* *Problem:* This refers to the difficulty in aligning the way data is structured in a relational database with how it is used in applications. Applications often use object-oriented programming, which doesn't naturally fit with the tabular format of relational databases.
    - \* *Solutions:*
      - *Within relational SQL paradigm:* Use Object-Relational Mapping (ORM) tools to bridge the gap between object-oriented applications and relational databases.
      - *With NoSQL approach:* NoSQL databases often use data models that align more closely with application structures, reducing the mismatch.
  - **Schema flexibility**
    - \* *Problem:* Relational databases require a predefined schema, which can be inflexible when dealing with evolving data requirements.
    - \* *Solutions:*
      - *Within relational SQL paradigm:* Use techniques like schema evolution or database migrations to adapt the schema over time.
      - *With NoSQL approach:* NoSQL databases often allow for dynamic schemas,

---

providing greater flexibility to accommodate changes.

- **Consistency in distributed set-up**

- \* *Problem:* Ensuring data consistency across distributed systems can be challenging with relational databases, especially when scaling out.

- \* **Solutions:**

- *Within relational SQL paradigm:* Implement distributed transactions or use techniques like sharding and replication to manage consistency.
    - *With NoSQL approach:* Some NoSQL databases offer eventual consistency models, which can be more suitable for distributed environments.

- **Limited scalability**

- \* *Problem:* Relational databases can struggle to scale horizontally, which means adding more servers to handle increased load.

- \* **Solutions:**

- *Within relational SQL paradigm:* Use techniques like partitioning and replication to improve scalability.
    - *With NoSQL approach:* NoSQL databases are often designed to scale out easily, making them a good choice for applications with large-scale data needs.

## 4 / 17: 1) App / DB Impedance Mismatch: Problem

### 1) App / DB Impedance Mismatch: Problem

- **Mismatch between data representation in code and relational DB**

- Code uses:
    - Data structures (e.g., lists, dictionaries, sets)
    - Objects
  - Relational DB uses:
    - Tables (entities)
    - Rows (instances of entities)
    - Relationships between tables

- **Example of app-DB mismatch:**

- Application stores a Python dictionary
    - # Store a dictionary from name (string) to tags (list of strings)
    - tag\_dict: Dict[str, List[str]]
  - Relational DB needs 3 tables:
    - Names(nameId, name) for keys
    - Tags(tagId, tag) for values
    - Names\_To\_Tags(nameId, tagId) to map keys to values
  - Denormalize using a single table:
    - Names(name, tag)

- **Mismatch between data representation in code and relational DB**

- In programming, we often use *data structures* like lists, dictionaries, and sets to organize and manipulate data. These structures are intuitive and flexible for developers.
  - In contrast, relational databases use a more rigid structure with tables, rows, and defined relationships between tables. This structure is optimized for storing and querying large amounts of data efficiently.

---

- **Example of app-DB mismatch:**

- Imagine you have an application that uses a Python dictionary to store data. This dictionary maps names (as strings) to tags (as lists of strings). It's a straightforward way to handle data in code.
- However, when you need to store this data in a relational database, you can't directly store a dictionary. Instead, you need to break it down into multiple tables:
  - \* One table (**Names**) to store the names.
  - \* Another table (**Tags**) to store the tags.
  - \* A third table (**Names\_To\_Tags**) to map each name to its corresponding tags.
- Alternatively, you might choose to *denormalize* the data by using a single table that combines names and tags, but this can lead to data redundancy and other issues.

## 5 / 17: 1) App / DB Impedance Mismatch: Solutions

### 1) App / DB Impedance Mismatch: Solutions

- **Ad-hoc mapping layer**

- Translate objects and data structures into DB model
  - Implement "Name to Tags" storage
  - Code uses a simple map, DB has 3 tables
- Cons
  - Requires writing and maintaining code

- **Object-relational mapping (ORM)**

- Pros
  - Automatic data conversion between object code and DB
  - Implement Person object using DB
  - Use SQLAlchemy for Python and SQL
- Cons
  - Handling complex types, polymorphism, and inheritance can be challenging

- **NoSQL approach**

- No schema
  - Objects can be flat or complex (e.g., nested JSON)
  - Stored objects (documents) can vary in structure

- **App / DB Impedance Mismatch: Solutions**

- **Ad-hoc mapping layer**

- This solution involves creating a custom layer that translates between the application's objects and the database's data structures. For example, if your application uses a simple map to manage data, but your database requires three separate tables, this layer will handle the conversion.
- *Cons:* The downside is that you need to write and maintain this translation code yourself, which can be time-consuming and error-prone.

- **Object-relational mapping (ORM)**

- ORMs are tools that automatically handle the conversion between your application's objects and the database. For instance, you can define a **Person** object in your code, and the ORM will manage how this object is stored and retrieved from the database.

- \* *Pros:* This approach simplifies data handling by automating conversions, making it easier to work with databases in languages like Python using tools like SQLAlchemy.
  - \* *Cons:* However, ORMs can struggle with more complex data types, such as those involving polymorphism and inheritance, which can complicate their use in certain scenarios.
- **NoSQL approach**
- NoSQL databases offer a flexible schema-less design, allowing you to store data in various formats, such as flat or complex nested JSON objects. This flexibility means that stored objects, or documents, can have different structures, which can be advantageous for applications with evolving data models.

## 6 / 17: 2) Schema Flexibility

### 2) Schema Flexibility

- **Problem**
  - Data may not fit into a schema
  - E.g., nested or dis-homogeneous data (List[Obj])
- **Within relational DB**
  - Use a general schema covering all cases
  - Cons
    - Complicated schema with implicit relations
    - Sparse DB tables
    - Violates relational DB assumptions
- **NoSQL approach**
  - E.g., MongoDB does not enforce schema
  - Pros
    - No schema concerns when writing data
  - Cons
    - Handle various schemas during data processing
    - Related to ETL vs ELT data pipelines

- **Problem**
  - **Data may not fit into a schema:** In traditional databases, data is expected to fit into a predefined structure or schema. However, real-world data can be complex and varied, such as having nested structures or being inconsistent in format (like a list of objects). This makes it challenging to fit such data into a rigid schema.
- **Within relational DB**
  - **Use a general schema covering all cases:** To accommodate diverse data, one might try to create a very broad schema that can handle all possible data variations.
  - **Cons**
    - \* **Complicated schema with implicit relations:** This approach can lead to overly complex schemas where relationships between data are not clear, making it difficult to understand and manage.
    - \* **Sparse DB tables:** A broad schema might result in many empty fields for certain

records, leading to inefficient storage and potential performance issues.

- \* **Violates relational DB assumptions:** Relational databases are designed with certain assumptions about data uniformity and structure, which can be compromised by trying to fit all data into a single schema.

- **NoSQL approach**

- **E.g., MongoDB does not enforce schema:** NoSQL databases like MongoDB offer flexibility by not requiring a fixed schema, allowing data to be stored in its natural form.
  - **Pros**

- \* **No schema concerns when writing data:** This flexibility means you can store data without worrying about fitting it into a predefined structure, which simplifies the data ingestion process.

- **Cons**

- \* **Handle various schemas during data processing:** While writing data is easier, processing it can become complex as you need to handle different data formats and structures.

- \* **Related to ETL vs ELT data pipelines:** This challenge is linked to the choice between ETL (Extract, Transform, Load) and ELT (Extract, Load, Transform) data processing strategies, where the latter might be more suitable for NoSQL due to its flexibility in handling diverse data.

## 7 / 17: 3) Consistency in Relational DBs

### 3) Consistency in Relational DBs

- **All systems fail**
  - Application error (corner case, internal error)
  - Application crash (OS issue)
  - Hardware failure (RAM ECC error, disk)
  - Power failure
- **Relational DBs enforce ACID properties**
  - Guarantee for system reliability
- **Atomicity**
  - Transactions are “all or nothing”
  - Transaction succeeds completely or fails
- **Consistency**
  - Transaction moves DB from one valid state to another
  - Maintain DB invariants (primary, foreign key constraints)
- **Isolation**
  - Concurrent transactions yield the same result as sequential execution
- **Durability**
  - Committed transaction content preserved after system failure



SCIENCE Record data in non-volatile memory  
ACADEMY

DEPARTURES Southwest					
DESTINATION	FLIGHT	AIRLINE	TIME	DAY	STATUS
Phoenix	2275	Southwest	3:10 PM	15	Canceled
Reno	459	Southwest	12:45 PM	16	Canceled
Albuquerque	1207	Southwest	1:45 PM	17	Canceled
Banff	2403	Southwest	1:55 PM	14	Canceled
Gatlinburg	3133	Southwest	12:00 PM	16A	Canceled
Gatlinburg	2403	Southwest	1:55 PM	14	Canceled
Las Vegas	1358	Southwest	1:55 PM	13	Canceled
San Jose	2279	Southwest	2:00 PM	15	Canceled
Salt Lake City	1384	Southwest	10:10 AM	13	Delayed
St. Louis	2275	Southwest	3:10 PM	15	Canceled

Application error



Hardware failure

- **All systems fail**

- *Application error:* This refers to mistakes or unexpected situations in the software, such as handling unusual inputs or bugs in the code.
  - *Application crash:* Sometimes, the software stops working due to issues with the operat-

ing system or other software conflicts.

- *Hardware failure*: Physical components like RAM or disks can malfunction, causing data issues or system crashes.
- *Power failure*: Loss of electricity can abruptly stop operations, risking data loss or corruption.

- **Relational DBs enforce ACID properties**

- These properties are crucial for ensuring that databases remain reliable and trustworthy, even when things go wrong.

- **Atomicity**

- This means that a transaction in a database is treated as a single unit. It either completes fully or not at all, preventing partial updates that could lead to data inconsistencies.

- **Consistency**

- Ensures that any transaction will bring the database from one valid state to another, maintaining rules like primary and foreign key constraints to keep data accurate and meaningful.

- **Isolation**

- This property ensures that transactions do not interfere with each other. Even if multiple transactions occur at the same time, the final result will be as if they were executed one after the other.

- **Durability**

- Once a transaction is completed and committed, its results are permanent. Even if the system crashes, the data will not be lost, as it is stored in a way that survives failures, typically in non-volatile memory.

## 8 / 17: 3) Consistency in Distributed DB

### 3) Consistency in Distributed DB

- Scale data or clients → **distributed setup**

- **Goals:**

- Performance (transactions per second) \*\*-
- Availability (up-time guarantee)
- Fault-tolerance (recover from faults)

- **Achieving ACID consistency:**

- *Not easy* in single DB
  - E.g., PostgreSQL guarantees ACID
  - E.g., MongoDB doesn't
- *Impossible* in distributed DB
  - Due to CAP theorem
  - Even weak consistency is difficult

**A = Atomicity**

**C = Consistency**

**I = Isolation**

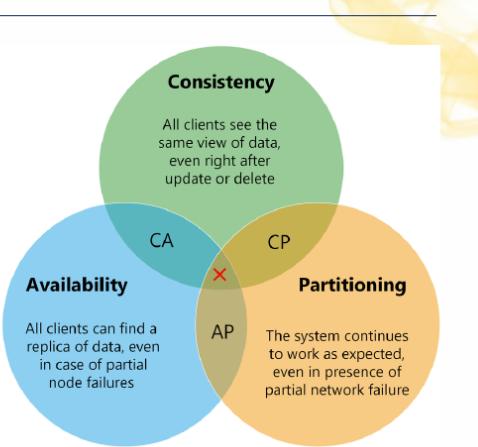
**D = Durability**

- **Scale data or clients → distributed setup**
  - When a database needs to handle more data or more users, it often requires a *distributed setup*. This means spreading the database across multiple servers or locations to manage the increased load effectively.
- **Goals:**
  - **Performance (transactions per second):** In a distributed database, one of the main goals is to maintain high performance, measured by how many transactions can be processed each second.
  - **Availability (up-time guarantee):** Ensuring that the database is always accessible, even if some parts of the system fail, is crucial. This is known as availability.
  - **Fault-tolerance (recover from faults):** The system should be able to recover from errors or failures without losing data or functionality.
- **Achieving ACID consistency:**
  - In a single database, maintaining ACID (Atomicity, Consistency, Isolation, Durability) properties is challenging. For example, PostgreSQL is known for providing these guarantees, whereas MongoDB does not fully support them.
  - In a distributed database, achieving ACID consistency is *impossible* due to the CAP theorem, which states that a distributed system can only provide two out of the three: Consistency, Availability, and Partition tolerance. Even achieving weak consistency, where some data might be temporarily out of sync, is difficult in such setups.

## 9 / 17: CAP Theorem

### CAP Theorem

- **CAP theorem:** Any distributed DB can have at most two of the following
  - **Consistency:**
    - All clients see the same data
    - Writes are atomic; subsequent reads retrieve the new value
  - **Availability:** Returns a value if at least one server is running
  - **Partition tolerance:** The system works even if communication is temporarily lost (network is partitioned)
- Originally a conjecture (Eric Brewer)
  - Proved formally (Gilbert, Lynch, 2002)



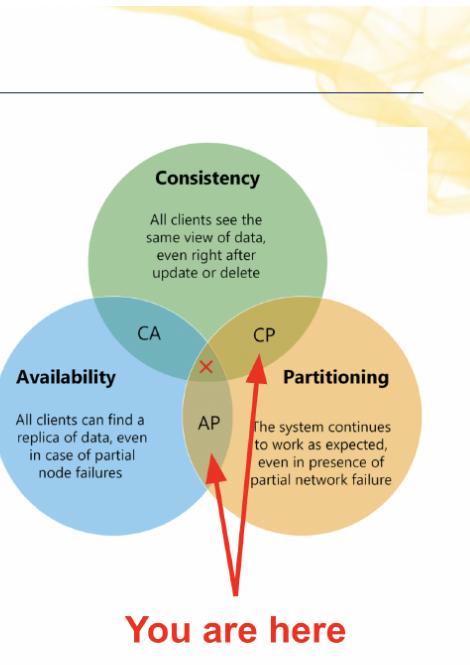
- **CAP theorem:** This is a fundamental principle in the design of distributed databases. It states that a distributed database can only guarantee two out of the three following properties at any given time:

- **Consistency:** This means that every read from the database will return the most recent write. In other words, all clients will see the same data at the same time. This is crucial for applications where it is important that everyone has the same view of the data.
- **Availability:** This ensures that the database will always respond to requests, even if some of the servers are down. It means that the system is designed to be operational and provide a response, regardless of failures.
- **Partition tolerance:** This property allows the system to continue functioning even if there are network failures that prevent some parts of the system from communicating with others. It is essential for systems that need to be resilient to network issues.
- Originally a conjecture (Eric Brewer): The CAP theorem was first proposed by Eric Brewer in 2000 as a conjecture. It was later proven formally by Seth Gilbert and Nancy Lynch in 2002. This theorem is crucial for understanding the trade-offs involved in designing distributed systems, as it highlights that achieving all three properties simultaneously is impossible. Developers must choose which two properties are most important for their specific application.

## 10 / 17: CAP Corollary

### CAP Corollary

- **CAP Theorem:** pick 2 among consistency, availability, partition tolerance
- **Network partitions**
  - Cannot be prevented in large-scale distributed systems
  - Can be reduced in probability using redundancy and fault tolerance
- You must sacrifice either:
  - **Availability**
    - Allow system downtime
    - E.g., banking system
  - **Consistency**
    - Allow different system views
    - E.g., social network



- **CAP Theorem:** This is a fundamental principle in distributed systems that states you can only achieve two out of the three following properties: *consistency*, *availability*, and *partition tolerance*. In simpler terms, when designing a distributed system, you have to make a trade-off because it's impossible to have all three properties at the same time.
- **Network partitions:** These occur when there is a failure in the network that prevents some parts of the system from communicating with others. In large-scale distributed systems, network partitions are inevitable due to the complexity and scale. However, their impact can be minimized by implementing redundancy (having multiple copies of data) and fault tolerance (designing the system to continue operating even when parts fail).

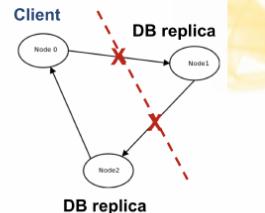
- You must sacrifice either:
  - **Availability:** This means the system might not always be operational or responsive. For example, in a banking system, it's crucial to maintain consistency (accurate data) even if it means the system is temporarily unavailable.
  - **Consistency:** This means different parts of the system might have different data at the same time. For example, in a social network, it's acceptable for users to see slightly different versions of their feed to ensure the system is always available.

The image likely illustrates these concepts, showing how different systems prioritize these properties based on their specific needs and use cases.

## 11 / 17: CAP Theorem: Intuition

### CAP Theorem: Intuition

- Consider:
  - Client (*Node0*)
  - Two DB replicas (*Node1*, *Node2*)
- **Network partition occurs**
  - DB servers (*Node1*, *Node2*) can't communicate
  - Users (*Node0*) access only one (*Node2*)
  - *Reads*: Access data on the same partition
  - *Writes*: Can't update due to potential inconsistency
- **CAP theorem:** Sacrifice consistency or availability
- **Available, not consistent**
  - Inconsistency acceptable (e.g., social networking)
  - Allow updates on accessible replica
- **Consistent, not available**
  - Inconsistency unacceptable (e.g., banking)
  - Stop service to maintain consistency



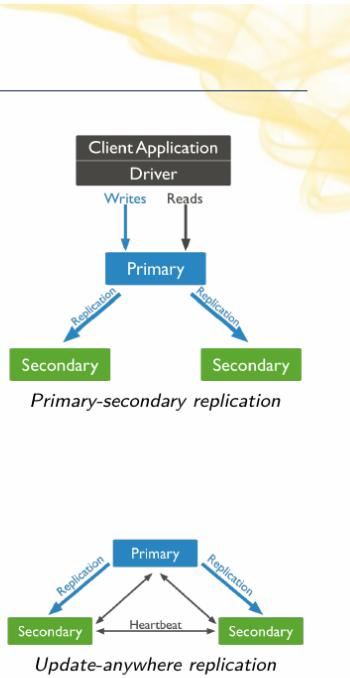
- **Consider:**
  - Imagine a scenario with a *client* (referred to as *Node0*) and two *database replicas* (referred to as *Node1* and *Node2*). These nodes are part of a distributed system where data is stored across multiple locations to ensure reliability and performance.
- **Network partition occurs:**
  - A network partition is a situation where the communication between nodes is disrupted. In this case, the database servers (*Node1* and *Node2*) cannot communicate with each other.
  - The client (*Node0*) can only access one of the database replicas, specifically *Node2*.
  - For *reads*, the client can still access data from the partition it is connected to, but for *writes*, it cannot update the data because doing so might lead to inconsistencies across the partitions.
- **CAP theorem: Sacrifice consistency or availability:**

- The CAP theorem states that in the presence of a network partition, a distributed system can only guarantee either *consistency* or *availability*, but not both.
- **Available, not consistent:**
  - In some systems, like social networking platforms, slight inconsistencies are acceptable. These systems prioritize availability, allowing updates on the accessible replica even if it means the data might not be consistent across all nodes.
- **Consistent, not available:**
  - In other systems, such as banking, consistency is crucial. These systems prioritize consistency over availability, meaning they might stop the service temporarily to ensure that all data remains consistent across the network.

## 12 / 17: Replication Schemes

### Replication Schemes

- **Replication schemes:** Organize multiple servers for a distributed DB
- **Primary-secondary replication**
  - Application communicates with the primary
  - Replicas require the primary for updates
  - Single point of failure
- **Update-anywhere replication**
  - Aka “multi-master replication”
  - Every replica can update data, which is propagated to others
- **Quorum-based replication**
  - $N$ : Total replicas
  - Write to  $W$  replicas
  - Read from  $R$  replicas, pick the latest update (timestamps)



- **Replication schemes:** These are strategies used to manage multiple servers in a distributed database system. The goal is to ensure data is consistently available and reliable across different locations.
- **Primary-secondary replication:**
  - In this setup, the application interacts directly with a primary server. This server is responsible for handling all updates.
  - Secondary servers, or replicas, rely on the primary server to receive updates. This means they are essentially copies of the primary.
  - A major downside is that if the primary server fails, the entire system can be disrupted, as it is a *single point of failure*.
- **Update-anywhere replication:**

- Also known as “multi-master replication,” this scheme allows any server in the system to update data.
- These updates are then shared with other servers, ensuring all replicas have the latest information.
- This approach provides more flexibility and resilience compared to primary-secondary replication.

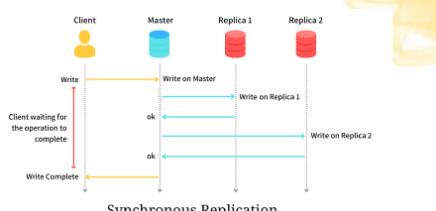
- **Quorum-based replication:**

- Involves a set number of replicas, denoted as  $N$ .
- To write data, it must be sent to  $W$  replicas.
- For reading, data is retrieved from  $R$  replicas, with the system selecting the most recent update based on timestamps.
- This method balances consistency and availability by requiring a majority agreement for operations.

## 13 / 17: Synchronous Replication

### Synchronous Replication

- **Synchronous replication:**  
updates propagate to replicas in a single transaction
- **Implementations**
  - **2-Phase Commit (2PC):**  
original method
    - Single point of failure
    - Can't handle primary server failure
  - **Paxos:** widely used
    - No primary required
    - More fault tolerant
  - Both are complex/expensive
- **CAP theorem:** only one of Consistency or Availability can be guaranteed during a network partition
  - Many systems use relaxed consistency models



- **Synchronous replication:** This is a method where updates to data are immediately copied to all replicas as part of a single transaction. This ensures that all copies of the data are consistent at any given time. It's like making sure every copy of a book is updated with the same changes at the same time.
- **Implementations:**
  - **2-Phase Commit (2PC):** This is one of the earliest methods used for synchronous replication. It involves two steps to ensure all parties agree on a transaction. However, it has a major drawback: if the primary server fails, the whole system can be stuck, as

it relies heavily on a single point of control.

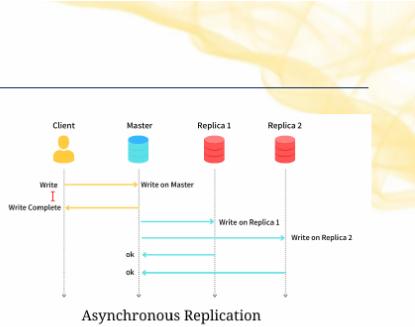
- **Paxos**: This is a more modern approach that doesn't rely on a primary server, making it more robust against failures. It allows for more flexibility and fault tolerance, meaning the system can continue to operate even if some parts fail. However, both 2PC and Paxos are known to be complex and can be costly to implement.
- **CAP theorem**: This is a fundamental principle in distributed systems that states you can only guarantee two out of three properties: Consistency, Availability, and Partition tolerance. During a network partition, you have to choose between keeping the system consistent or available. Many systems opt for relaxed consistency models, which means they allow for temporary inconsistencies to maintain availability. This is a trade-off that system designers often have to consider.

## 14 / 17: Asynchronous Replication

### Asynchronous Replication

- **Asynchronous replication**

- Primary node updates replicas
- Transaction completes before replicas update
- Quick commits, less consistency



- **Eventual consistency**

- Popularized by AWS DynamoDB
- Consistency only on eventual outcome
- "Eventual" may mean after server/network fix

- **"Freshness" property**

- Read from replica may not be latest
- Request version with specific "freshness"
  - E.g., "data from not more than 10 minutes ago"
  - E.g., show airplane ticket price a few minutes old
- Replicas use timestamps for data versioning
- Use local replica if fresh, else request primary node



14 / 17

- **Asynchronous replication**

- *Primary node updates replicas*: In this setup, the main server (primary node) sends updates to other servers (replicas) but doesn't wait for them to confirm they've received the updates. This means the primary node can continue processing new transactions without delay.
- *Transaction completes before replicas update*: The system allows a transaction to be marked as complete even if the replicas haven't been updated yet. This can speed up operations but might lead to temporary inconsistencies.
- *Quick commits, less consistency*: The advantage here is speed—transactions are committed quickly. However, the downside is that the data might not be consistent across all replicas immediately.

- **Eventual consistency**

- *Popularized by AWS DynamoDB*: This concept is widely used in distributed databases like AWS DynamoDB, where the system guarantees that, eventually, all replicas will be consistent.
- *Consistency only on eventual outcome*: The system doesn't promise immediate consistency but ensures that, over time, all copies of the data will become consistent.
- *"Eventual" may mean after server/network fix*: The term "eventual" can vary; it might mean after a network issue is resolved or once the system has had time to synchronize.
- “Freshness” property
  - *Read from replica may not be latest*: When you read data from a replica, it might not be the most current version because of the delay in updates.
  - *Request version with specific “freshness”*: Users can specify how recent the data should be. For example, they might request data that's no older than 10 minutes.
    - \* *E.g., “data from not more than 10 minutes ago”*: This ensures that the data is relatively up-to-date without needing to be the absolute latest.
    - \* *E.g., show airplane ticket price a few minutes old*: In some cases, like checking ticket prices, slightly older data might be acceptable.
  - *Replicas use timestamps for data versioning*: Each piece of data is tagged with a timestamp to track its version and freshness.
  - *Use local replica if fresh, else request primary node*: If the local replica has sufficiently fresh data, it's used. Otherwise, the system queries the primary node for the latest information.

## 15 / 17: 4) Scalability Issues with RDMS: Problem

### 4) Scalability Issues with RDMS: Problem

- Sources of SQL DB scalability issues:
  1. **Locking data**
    - DB engine locks rows/tables for ACID
    - When locked higher latency → Fewer updates/second → Slower application
  2. **Worse in distributed set-up**
    - Requires data replication over multiple servers (scaling out)
    - Slower application due to:
      - Network delays
      - Locks across networks for DB consistency
      - Overhead of replica consistency (2PC, Paxos)

#### • Scalability Issues with RDMS: Problem

This slide discusses the challenges faced by traditional SQL databases, also known as Re-

lational Database Management Systems (RDMS), when it comes to scaling. These issues become more pronounced as the demand for data processing increases.

- **Sources of SQL DB scalability issues:**

1. **Locking data**

- In order to maintain *ACID* (Atomicity, Consistency, Isolation, Durability) properties, the database engine locks rows or entire tables. This is crucial for ensuring that transactions are processed reliably.
- However, when data is locked, it can lead to higher latency. This means that the database takes longer to process requests, resulting in fewer updates per second. Consequently, the application relying on the database becomes slower.

2. **Worse in distributed set-up**

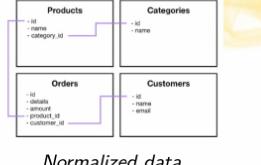
- When databases are distributed across multiple servers to handle more data (a process known as scaling out), the complexity increases.
- The application can slow down due to several factors:
  - \* *Network delays* occur as data needs to be communicated across different servers.
  - \* Locks need to be maintained across networks to ensure database consistency, which can be challenging.
  - \* Ensuring that all replicas of the database are consistent adds overhead. Techniques like Two-Phase Commit (2PC) and Paxos are used, but they can be resource-intensive and slow down the system.

## 16 / 17: 4) Scalability Issues with RDMS: Solutions

### 4) Scalability Issues with RDMS: Solutions

- **Table denormalization**

- Increase performance by adding redundant data
- Pros
  - Faster reads: Lock one table, no joins
- Cons
  - Slower writes: More data to update
  - Loss of table relations

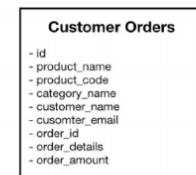


Normalized data

- **Relax consistency**

- Compromise on ACID
- Weaken consistency (e.g., eventual consistency)

- **NoSQL**



Denormalized data

- **Table denormalization**

- *Denormalization* is a technique used to improve the performance of a database by adding

redundant data. This means that instead of having data spread across multiple tables, some data is duplicated in a single table to make data retrieval faster.

- **Pros**

- \* *Faster reads:* Since all the necessary data is in one table, the system doesn't need to perform complex joins between tables, which speeds up data retrieval.

- **Cons**

- \* *Slower writes:* When data is updated, the system has to update multiple places where the data is stored, which can slow down the process.
- \* *Loss of table relations:* The logical connections between different pieces of data can become less clear, making the database harder to maintain.

- **Relax consistency**

- This involves compromising on the strict *ACID* properties (Atomicity, Consistency, Isolation, Durability) that traditional databases follow. By relaxing these rules, systems can achieve better performance and scalability.
- *Weaken consistency:* An example is *eventual consistency*, where the system allows temporary inconsistencies but ensures that all changes will eventually be reflected across the system.

- **NoSQL**

- NoSQL databases are designed to handle large volumes of data and high user loads. They are more flexible than traditional relational databases and can scale horizontally, making them suitable for big data applications.

## 17 / 17: NoSQL Stores

### NoSQL Stores

- **Use cases of large-scale web applications**
  - Real-time access with ms latencies
    - E.g., facebook: 4ms for reads
  - No need for ACID properties
  - MongoDB started at DoubleClick (AdTech), acquired by Google
- **Solve problems with relational databases**
  - Application-DB impedance mismatch
  - Schema flexibility
  - Consistency in distributed setup
  - Scalability
- **To scale out, give up something**
  - Consistency
  - Joins
    - Most NoSQL stores don't allow server-side joins
    - Require data denormalization and duplication
  - Restricted transactions
    - Most NoSQL stores allow single-object transactions
    - E.g., one document/key

- **Use cases of large-scale web applications**

- Large-scale web applications, like social media platforms, require *real-time access* to

---

data, often with very low latency. For instance, Facebook aims for read operations to be completed in about 4 milliseconds. This speed is crucial for providing a seamless user experience.

- In these scenarios, the traditional ACID (Atomicity, Consistency, Isolation, Durability) properties of databases are often not necessary. Instead, the focus is on speed and availability.
- MongoDB, a popular NoSQL database, originated from DoubleClick, a company in the advertising technology sector, which was later acquired by Google. This highlights how NoSQL databases are often born out of the need to handle large volumes of data efficiently.
- **Solve problems with relational databases**
  - NoSQL databases address the *application-database impedance mismatch*, which is the difficulty in translating data between the database and application layers.
  - They offer *schema flexibility*, allowing developers to store data without a fixed structure, which is beneficial for applications that evolve over time.
  - NoSQL databases are designed to maintain *consistency in distributed setups*, which is challenging for traditional relational databases.
  - They are built for *scalability*, making it easier to handle growing amounts of data and user requests.
- **To scale out, give up something**
  - In order to achieve scalability, NoSQL databases often sacrifice *consistency*. This is part of the CAP theorem, which states that a distributed database can only provide two out of the three: Consistency, Availability, and Partition tolerance.
  - NoSQL databases typically do not support *server-side joins*, which means that data often needs to be denormalized and duplicated across the database to avoid complex queries.
  - They offer *restricted transactions*, usually allowing transactions on a single object, such as one document or key, rather than across multiple objects. This simplifies the database architecture but requires careful data management.

---

## Lesson 5.2: Database Taxonomy



UMD DATA605 - Big Data Systems

### Lesson 5.2: Database Taxonomy

- **Instructor:** Dr. GP Saggese, [gsaggese@umd.edu](mailto:gsaggese@umd.edu)
- **References:**
  - Online tutorials
  - Silberschatz: Chap 10.2
  - Seven Databases in Seven Weeks, 2e



1 / 8

### DB Taxonomy

- **At least five DB genres**
  - *Relational* (e.g., PostgreSQL)
  - *Key-value* (e.g., Redis)
  - *Document* (e.g., MongoDB)
  - *Columnar* (e.g., Apache Parquet)
  - *Graph* (e.g., Neo4j)
- **Criteria to differentiate DBs**
  - Data model
  - Trade-off with CAP theorem
  - Querying capability
  - Replication scheme



2 / 8

- **DB Taxonomy**
  - **At least five DB genres**
    - \* ***Relational* (e.g., PostgreSQL)**: These databases organize data into tables with rows and columns, using a structured query language (SQL) for defining and manipulating data. They are great for complex queries and transactions.
    - \* ***Key-value* (e.g., Redis)**: This type of database stores data as a collection of key-value pairs. It's simple and fast, making it ideal for caching and real-time applications.
    - \* ***Document* (e.g., MongoDB)**: Document databases store data in JSON-like formats, allowing for flexible and hierarchical data structures. They are well-suited for applications with varying data types.
    - \* ***Columnar* (e.g., Apache Parquet)**: These databases store data in columns rather than rows, optimizing for read-heavy operations and analytical queries. They are often used in big data and data warehousing.
    - \* ***Graph* (e.g., Neo4j)**: Graph databases use nodes, edges, and properties to represent and store data, making them perfect for applications that involve complex relationships, like social networks.
  - **Criteria to differentiate DBs**
    - \* **Data model**: This refers to how data is structured and stored, which can significantly impact the database's performance and suitability for different tasks.
    - \* **Trade-off with CAP theorem**: The CAP theorem states that a distributed database can only guarantee two out of three properties: Consistency, Availability, and Partition tolerance. Different databases prioritize these properties differently.
    - \* **Querying capability**: This involves the complexity and flexibility of the queries

that can be performed, which varies across different database types.

- \* **Replication scheme:** This refers to how data is copied and maintained across multiple locations, affecting the database's reliability and performance.

## 3 / 8: Relational DB

### Relational DB

- E.g., *Postgres*, MySQL, Oracle, SQLite
- **Data model**
  - Set-theory, relational algebra
  - Data as tables with rows and columns
  - Many attribute types (e.g., numeric, strings, dates, arrays, blobs)
  - Strictly enforced attribute types
  - SQL query language
  - ACID compliance
- **Good for**
  - Known data layout, unknown access pattern
  - Schema complexity for query flexibility
  - Regular data
- **Not so good for**
  - Hierarchical data (not easily represented as rows in tables)
  - Variable/heterogeneous data (record-to-record variation)



3 / 8

#### • Relational DB

- Examples of relational databases include *Postgres*, MySQL, Oracle, and SQLite. These are popular systems used to store and manage data in a structured way.

#### • Data model

- Relational databases are based on *set-theory* and *relational algebra*. This means they use mathematical concepts to organize and manipulate data.
- Data is organized into tables, which are like spreadsheets with rows and columns. Each row represents a record, and each column represents an attribute of the data.
- There are many types of attributes you can use, such as numbers, text, dates, and even more complex types like arrays and blobs (binary large objects).
- Attribute types are strictly enforced, meaning that each column in a table must contain data of a specific type.
- SQL (Structured Query Language) is used to query and manipulate the data. It's a powerful language that allows you to perform complex operations on the data.
- Relational databases are ACID compliant, which stands for Atomicity, Consistency, Isolation, and Durability. This ensures that transactions are processed reliably.

#### • Good for

- Relational databases are ideal when you know the structure of your data but not necessarily how it will be accessed. This flexibility allows for complex queries.
- They are great for handling complex schemas, which means you can have intricate rela-

tionships between different tables and still query them efficiently.

- They work well with regular data, where the structure doesn't change much over time.

- **Not so good for**

- Relational databases struggle with hierarchical data, which is data that naturally forms a tree-like structure. This type of data doesn't fit well into tables.
- They are not ideal for variable or heterogeneous data, where each record might have a different structure. This can make it difficult to fit such data into a fixed schema.

## 4 / 8: Key-Value Store

### Key-Value Store

- E.g., Redis, DynamoDB, *Git*, AWS S3, filesystem

- **Data model**

- Map keys (e.g., strings) to complex values (e.g., binary blob)
- Support get, put, delete operations on a primary key

- **Application**

- Cache data
- Store users' session data in web applications
- Store shopping carts in e-commerce applications

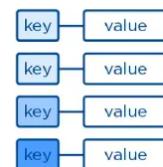
- **Good for**

- Unrelated data (e.g., no joins)
- Fast lookups
- Easy horizontal scaling using partitioning

- **Not so good for**

- Data queries
- Lacking secondary indexes and scanning

#### Key-Value



- **Key-Value Store Examples**

- Examples include *Redis*, *DynamoDB*, *Git*, *AWS S3*, and traditional filesystems. These are systems designed to store, retrieve, and manage data using a simple key-value pair model.

- **Data Model**

- The data model involves mapping *keys* (like strings) to *complex values* (such as binary blobs). This means each piece of data is stored with a unique identifier (the key), and the data itself can be any form of complex data.
- Operations supported include *get* (retrieve data), *put* (store data), and *delete* (remove data) using the primary key.

- **Applications**

- Key-value stores are often used to *cache data*, which helps in speeding up data retrieval.
- They are used to store *users' session data* in web applications, ensuring quick access and updates.
- In e-commerce applications, they can store *shopping carts*, allowing for fast retrieval and updates as users add or remove items.

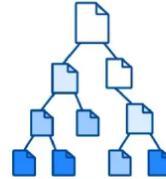
- **Good for**
  - They excel in handling *unrelated data*, where there is no need for complex relationships or joins between data.
  - They provide *fast lookups* due to the simplicity of the key-value model.
  - They allow for *easy horizontal scaling* through partitioning, which means they can handle large amounts of data by distributing it across multiple servers.
- **Not so good for**
  - They are not ideal for *data queries* that require complex searching or filtering.
  - They lack *secondary indexes* and the ability to perform *scanning* operations, which limits their use in scenarios where such features are necessary.

## 5 / 8: Document Store

### Document Store

- E.g., *MongoDB*, *CouchBase*
- **Data model**
  - Key-value with document as value (nested dict)
  - Unique ID for each document (e.g., hash)
  - Any number of fields per document, including nested
    - E.g., JSON, XML, dict data
- **Application**
  - Semi-structured data
- **Good for**
  - Unknown data structure
  - Maps to OOP models (less impedance mismatch)
  - Easy to shard and replicate over distributed servers
- **Not so good for**
  - Complex join queries
  - Denormalized form is standard

Document



- **Document Store:** This slide introduces the concept of a document store, which is a type of database designed to store, retrieve, and manage document-oriented information. Examples include *MongoDB* and *CouchBase*.

- **Data model:**

- The data model in document stores is based on a key-value structure where the *document* acts as the value. This document is typically a nested dictionary, which means it can contain other dictionaries within it.
- Each document is assigned a unique identifier, often a hash, which helps in efficiently retrieving and managing the documents.
- Documents can have any number of fields, and these fields can be nested, allowing for complex data structures. Common formats for these documents include JSON, XML, and dictionary data structures.

- **Application:**

- Document stores are particularly useful for handling *semi-structured data*, which doesn't fit neatly into traditional table structures.

- **Good for:**

- They are ideal when the data structure is unknown or likely to change, as they offer flexibility in how data is stored.
- Document stores align well with object-oriented programming (OOP) models, reducing the complexity of translating between the database and application code (known as impedance mismatch).
- They are designed to be easily distributed across multiple servers, making them scalable and reliable for large-scale applications.

- **Not so good for:**

- Document stores are not optimized for complex join queries, which are common in relational databases.
- Data is typically stored in a denormalized form, meaning that data redundancy is common, which can lead to inefficiencies in certain scenarios.

## 6 / 8: Columnar Store

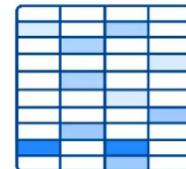
### Columnar Store

- E.g., *HBase*, *Cassandra*, *Parquet*

- **Data model**

- Store data by columns, not rows
- Similar to key-value and relational DBs
  - Use keys to query values
  - Values are groups of columns

#### Wide-column



- **Application**

- Store web pages
- Store time series data
- OLAP workloads

- **Good for**

- Horizontal scalability
- Enable compression and versioning
- Sparse tables without extra storage cost
- Inexpensive to add columns

- **Not so good for**

- Designing schema based on query plans
- No native joins; applications handle joins

- **Columnar Store Examples**

- Examples include *HBase*, *Cassandra*, and *Parquet*. These are popular technologies used to store and manage large datasets efficiently by organizing data in columns rather than rows.

- **Data Model**

- 
- Columnar stores organize data by columns instead of rows. This is different from traditional databases that store data in rows.
  - They are similar to key-value and relational databases in that they use keys to access data. However, the values retrieved are groups of columns, which can be more efficient for certain types of queries.

- **Application**

- Columnar stores are well-suited for storing web pages and time series data, which often involve large datasets with many columns.
- They are also ideal for OLAP (Online Analytical Processing) workloads, which require fast query performance on large datasets.

- **Good for**

- These systems are designed for horizontal scalability, meaning they can easily expand by adding more servers.
- They support data compression and versioning, which can save storage space and track changes over time.
- Columnar stores handle sparse tables efficiently, avoiding extra storage costs for empty or null values.
- Adding new columns is inexpensive, making it easy to adapt to changing data requirements.

- **Not so good for**

- Designing schemas based on query plans can be challenging because columnar stores are optimized for reading rather than writing.
- They do not support native joins, so applications must handle joins, which can complicate application development and reduce performance.

The image on the right likely illustrates the concept of columnar storage, showing how data is organized by columns rather than rows, which can help visualize the differences from traditional row-based storage systems.

### Graph DB

- E.g., *Neo4j*, GraphX
- **Data model**
  - Interconnected data: nodes, relationships
  - Nodes and edges have properties (key-value pairs)
  - Queries traverse nodes and relationships
- **Applications**
  - Social data
  - Recommendation engines
  - Geographical data
- **Good for**
  - Networked data, hard to model with a relational model
  - Matches object-oriented (OO) systems
- **Not so good for**
  - Poor scalability, hard to partition graph across different nodes
    - Store graph in graph DB, relations in key-value store



7 / 8

- **Graph DB Examples**
  - *Neo4j* and GraphX are popular examples of graph databases. These tools are designed to handle data that is naturally interconnected, like social networks or recommendation systems.
- **Data Model**
  - Graph databases use a model that consists of *nodes* and *relationships*. Nodes represent entities, while relationships connect these entities. Both nodes and edges (relationships) can have properties, which are essentially key-value pairs that store additional information.
  - Queries in graph databases are designed to traverse these nodes and relationships, making it easy to explore complex networks of data.
- **Applications**
  - Graph databases are particularly useful for managing *social data*, where relationships between users are crucial.
  - They are also used in *recommendation engines*, which rely on understanding connections between users and products.
  - *Geographical data* can benefit from graph databases due to the natural way they handle spatial relationships.
- **Good for**
  - Graph databases excel at handling *networked data*, which can be challenging to represent using traditional relational databases.
  - They align well with *object-oriented (OO) systems*, as both use a similar approach to modeling data.
- **Not so good for**

- One downside is their *poor scalability*. It can be difficult to partition a graph across multiple nodes, which can limit performance as the dataset grows.
- A common workaround is to store the graph structure in a graph database while using a key-value store for relationships, but this can add complexity.

## 8 / 8: Taxonomy by CAP

**Taxonomy by CAP**

- **CA (Consistent, Available) systems**
  - Struggle with partitions, use replication
  - Traditional RDBMSs (PostgreSQL, MySQL)
- **CP (Consistent, Partition-Tolerant) systems**
  - Struggle with availability, maintain consistency across partitions
  - BigTable (column-oriented/tabular)
  - HBase (column-oriented/tabular)
  - Redis (key-value)
  - Berkeley DB (key-value)
- **AP (Available, Partition-Tolerant) systems**
  - Achieve “eventual consistency” via replication and verification
  - MongoDB (document-oriented)
  - Memcached (key-value)
  - Dynamo (key-value)
  - Cassandra (column-oriented/tabular)
  - CouchDB (document-oriented)

SCIENCE ACADEMY

8 / 8

- **CA (Consistent, Available) systems**
  - These systems are designed to ensure that data is always consistent and available, but they face challenges when network partitions occur. A network partition is when there is a break in communication between different parts of a system. To handle this, CA systems often use replication, which means they keep multiple copies of data to ensure availability and consistency. Traditional relational database management systems (RDBMSs) like PostgreSQL and MySQL are examples of CA systems. They are well-suited for applications where data consistency and availability are critical, but they may not perform well in distributed environments where partitions are common.
- **CP (Consistent, Partition-Tolerant) systems**
  - These systems prioritize consistency and can handle network partitions, but they may struggle with availability. This means that during a partition, the system ensures that all nodes have the same data, but some parts of the system might not be accessible. Examples include BigTable and HBase, which are column-oriented databases, and Redis and Berkeley DB, which are key-value stores. These systems are ideal for applications where data consistency is more important than availability, such as financial transactions.
- **AP (Available, Partition-Tolerant) systems**
  - AP systems focus on being available and partition-tolerant, but they achieve only *eventual consistency*. This means that while data might not be immediately consistent

---

across all nodes, it will become consistent over time through replication and verification processes. Examples include MongoDB and CouchDB, which are document-oriented databases, and Memcached, Dynamo, and Cassandra, which are key-value or column-oriented stores. These systems are suitable for applications where availability is crucial, such as social media platforms, where temporary inconsistencies are acceptable.

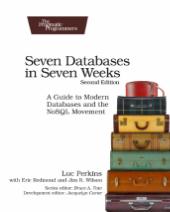
---

## Lesson 5.3: Apache HBase



### Lesson 5.3: Apache HBase

- **Instructor:** Dr. GP Saggese, [gsaggese@umd.edu](mailto:gsaggese@umd.edu)
- **References**
  - Web
    - 2006, BigTable paper
    - <https://hbase.apache.org/>
    - <https://github.com/apache/hbase>
  - Good overview:
    - Seven Databases in Seven Weeks, 2e



1 / 25

### (Apache) HBase

- HBase = **Hadoop DataBase**
  - Supports large tables on commodity hardware clusters
  - Column-oriented DB
  - Part of Apache Hadoop ecosystem
  - Uses Hadoop filesystem (HDFS)
    - HDFS modeled after Google File System (GFS)
    - HBase based on Google BigTable
    - Google BigTable runs on GFS, HBase runs on HDFS
  - Used by Google, Airbnb, eBay
- **When to use HBase**
  - For large DBs (e.g., many 100 GBs or TBs)
  - With at least 5 nodes in production
- **Applications**
  - Large-scale online analytics
  - Heavy-duty logging
  - Search systems (e.g., Internet search)
  - Facebook Messages (based on Cassandra)
  - Twitter metrics monitoring



2 / 25

- **HBase = Hadoop DataBase**
  - HBase is a database that is designed to handle very large tables across clusters of standard, affordable hardware. This makes it a good fit for organizations that need to manage large amounts of data without investing in expensive infrastructure.
  - It is a *column-oriented* database, which means it stores data in columns rather than rows. This can be more efficient for certain types of queries, especially those that involve aggregating data.
  - HBase is part of the Apache Hadoop ecosystem, which is a collection of open-source software utilities that facilitate using a network of many computers to solve problems involving massive amounts of data and computation.
  - It uses the Hadoop Distributed File System (HDFS) to store its data. HDFS is inspired by the Google File System (GFS), and HBase itself is modeled after Google's BigTable. Essentially, HBase is to HDFS what BigTable is to GFS.
  - Companies like Google, Airbnb, and eBay use HBase, which highlights its reliability and scalability for large-scale applications.
- **When to use HBase**
  - HBase is ideal for databases that are extremely large, such as those that are hundreds of gigabytes or even terabytes in size. This makes it suitable for enterprises dealing with big data.
  - It is recommended to have at least five nodes in a production environment to ensure performance and reliability. This means HBase is best suited for organizations that can support a multi-node setup.
- **Applications**
  - HBase is used for large-scale online analytics, where it can efficiently process and analyze

vast amounts of data.

- It is also used for heavy-duty logging, which involves recording and storing large volumes of log data.
- Search systems, such as those used for Internet search, benefit from HBase's ability to handle large datasets and provide quick access to data.
- While Facebook Messages uses Cassandra, another type of database, HBase is similar in that it can handle large-scale, distributed data storage needs.
- Twitter uses HBase for metrics monitoring, which involves tracking and analyzing data to understand user interactions and system performance.

### 3 / 25: HBase: Features

## HBase: Features

- Data versioning
  - Store versions of data
- Data compression
  - Compress and decompress on-the-fly
- Garbage collection for expired data
- In-memory tables
- Atomicity at row level
- Strong consistency guarantees
- Fault tolerant for machines and network
  - Write-ahead logging
    - Write data to in-memory log before disk
  - Distributed configuration
    - Nodes rely on each other, not centralized source



3 / 25

- **HBase: Features**

- **Data versioning**

- HBase allows you to store multiple versions of the same data. This means that every time you update a piece of data, the previous version is not immediately discarded. Instead, it is kept for a certain period or number of versions, which can be useful for auditing changes or recovering from accidental updates.

- **Data compression**

- HBase can compress data to save storage space and reduce the amount of data that needs to be transferred over the network. This compression and decompression happen automatically as data is read and written, making it efficient and seamless for users.

- **Garbage collection for expired data**

- HBase automatically cleans up data that is no longer needed, such as old versions of data that have expired. This helps in managing storage efficiently and ensures that the system does not get bogged down with unnecessary data.

- **In-memory tables**

- HBase can store tables in memory, which allows for faster data access and processing. This is particularly useful for applications that require quick read and write operations.

- **Atomicity at row level**

- HBase ensures that operations on a single row are atomic, meaning they are completed entirely or not at all. This is crucial for maintaining data integrity, especially in systems where multiple operations might be happening simultaneously.

- **Strong consistency guarantees**

- HBase provides strong consistency, meaning that once a write operation is completed, any subsequent read will reflect that change. This is important for applications that require reliable and predictable data access.

- **Fault tolerant for machines and network**

- HBase is designed to handle failures gracefully. It uses *write-ahead logging*, where data is first written to an in-memory log before being saved to disk, ensuring that no data is lost in case of a failure. Additionally, its *distributed configuration* means that nodes work together without relying on a single point of failure, enhancing the system's resilience.

## 4 / 25: From HDFS to HBase

### From HDFS to HBase

- **Different types of workloads for DB backends**
  - **OLTP** (On-Line Transactional Processing)
    - Read and write individual data items in a large table
    - E.g., update inventory and price as orders come in
  - **OLAP** (On-Line Analytical Processing)
    - Read large data amounts and process it
    - E.g., analyze item purchases over time
- **Hadoop FileSystem (HDFS) supports OLAP workloads**
  - Provide a filesystem with large files
  - Read data sequentially, end-to-end
  - Rarely updated
- **HBase supports OLTP interactions**
  - Built on HDFS
  - Use additional storage and memory to organize tables
  - Write tables back to HDFS as needed

- Different types of workloads for DB backends
  - **OLTP** (On-Line Transactional Processing)
    - \* This is about handling lots of small transactions, like reading or writing individual pieces of data in a big table. Think of it like updating the stock levels and prices in a store's database every time someone makes a purchase. It's all about quick, small updates.
  - **OLAP** (On-Line Analytical Processing)
    - \* This involves reading and analyzing large amounts of data. Imagine looking at all the sales data over a year to find trends or patterns. It's more about big-picture analysis rather than small, frequent updates.
- Hadoop FileSystem (HDFS) supports OLAP workloads
  - HDFS is designed to handle large files and is great for reading data from start to finish. It's like a big library where you can read entire books (datasets) but don't often change them. This makes it ideal for OLAP tasks where you need to process lots of data at once.
- HBase supports OLTP interactions
  - HBase is built on top of HDFS but is designed to handle OLTP tasks. It uses extra storage and memory to keep tables organized, allowing for quick updates and reads. When necessary, it writes these tables back to HDFS. This makes HBase suitable for applications that require frequent updates and fast access to individual data items.

## 5 / 25: HBase Data Model

### HBase Data Model

- **Warning:** HBase uses names similar to relational DB concepts, but with different meanings
- A **database** consists of multiple tables
- Each **table** consists of multiple rows, sorted by row key
- Each row contains a *row key* and one or more column families
- Each **column family**
  - Contains multiple columns (family:column)
  - Defined when the table is created
- A **cell**
  - Uniquely identified by (table, row, family:column)
  - Contains metadata (e.g., timestamp) and an uninterpreted array of bytes (blob)
- **Versioning**
  - New values don't overwrite old ones
  - `put()` and `get()` allow specifying a timestamp (otherwise uses current time)

```
\# HBase Database: from table name to Table.
Database = Dict[str, Table]

\# HBase Table.
table: Table = {
    # Row key
    'row1': {
        # (column family:column) + value
        'cf1:col1': 'value1',
        'cf1:col2': 'value2',
        'cf2:col1': 'value3'
    },
    'row2': {
        ... # More row data
    }
}
database = {'table1': table}

\# Querying data.
(value, metadata) = \
    table['row1']['cf1:col1']
```



SCIENCE  
ACADEMY

- **Warning:** It's important to note that while HBase uses terms like "database" and "table," these terms don't mean the same thing as they do in traditional relational databases. This can be confusing, so it's crucial to understand the differences.

- **Database:** In HBase, a database is a collection of tables. This is similar to relational databases, but the way data is stored and accessed is different.
- **Table:** Each table in HBase is made up of rows, which are sorted by a unique identifier called a row key. This sorting helps in quickly accessing data.
- **Row:** A row is identified by a row key and contains one or more column families. The row key is crucial for accessing data efficiently.
- **Column Family:** This is a group of columns that are defined when the table is created. Each column family can have multiple columns, and they are accessed using the format family:column.
- **Cell:** A cell is the intersection of a row and a column. It's uniquely identified by the combination of table, row, and family:column. Each cell can store a blob of data and metadata like timestamps.
- **Versioning:** HBase supports versioning, meaning new data doesn't overwrite old data. You can specify a timestamp when using `put()` and `get()` operations, or it defaults to the current time. This feature is useful for keeping track of changes over time.

The code snippet on the right illustrates how data is structured and accessed in HBase. It shows a dictionary-like structure where you can query data using row keys and column identifiers.

## 6 / 25: Example 1: Colors and Shape

### Example 1: Colors and Shape

- Table with:
  - 2 column families: “color” and “shape”
  - 2 rows: “first” and “second”
- Row “first”:
  - 3 columns in “color”: “red”, “blue”, “yellow”
  - 1 column in “shape”: shape = 4
- Row “second”:
  - No columns in “color”
  - 2 columns in “shape”
- Access data using row key and column (family:qualifier)

row keys	column family “color”	column family “shape”
“first”	“red”: “#F00” “blue”: “#00F” “yellow”: “#FF0”	“square”: “4”
“second”		“triangle”: “3” “square”: “4”

```
table = {
  'first': {
    # (column family, column) -> value
    'color': {'red': '#F00',
              'blue': '#00F',
              'yellow': '#FF0'},
    'shape': {'square': '4'}
  },
  'second': {
    'shape': {'triangle': '3',
              'square': '4'}
  }
}
```



6 / 25

- **Example 1: Colors and Shape**

- This slide presents a data structure example using a table format with two main column families: “color” and “shape”. Column families are a way to group related data together,

which is common in databases like HBase.

– **Table Structure:**

- \* The table has two rows labeled as “first” and “second”. Each row can have different columns under each column family.

– **Row “first”:**

- \* Under the “color” column family, there are three columns: “red”, “blue”, and “yellow”. These columns likely represent different color values.
- \* Under the “shape” column family, there is one column with a value of 4, which might represent a shape’s attribute, such as the number of sides for a square.

– **Row “second”:**

- \* This row has no columns under the “color” column family, indicating that it does not store any color data.
- \* Under the “shape” column family, there are two columns, which could represent different shapes or attributes, such as a triangle with 3 sides and a square with 4 sides.

– **Data Access:**

- \* Data is accessed using a combination of the row key and the column family with a qualifier, which is a common practice in column-oriented databases. This allows for efficient retrieval of specific data points.

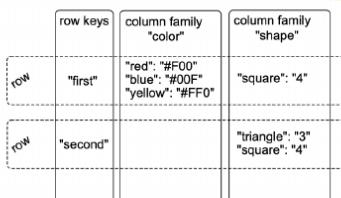
– **Python Representation:**

- \* The slide includes a Python dictionary representation of the table, showing how the data can be structured programmatically. This helps in understanding how such data might be stored and accessed in a real-world application.

## 7 / 25: Why All This Convoluted Stuff?

### Why All This Convoluted Stuff?

- **A row in HBase is like a mini-database**
  - A cell has many values
  - Data stored sparsely
- **Rows in HBase are “deeper” than in relational DBs**
  - Relational DBs: rows have many column values (fixed array with types)
  - HBase: rows like a two-level nested dictionary with metadata (e.g., timestamp)
- **Applications**
  - Store versioned website data
  - Store a wiki



- A row in HBase is like a mini-database
  - In HBase, each row can be thought of as a small database because it can hold multiple values within a single cell. This is different from traditional databases where each cell typically holds a single value. The ability to store multiple values in a cell allows for more complex data structures and relationships to be represented within a single row.
  - Data in HBase is stored sparsely, meaning that only the cells with data are stored, which can save space and improve efficiency. This is particularly useful when dealing with large datasets where not all fields are populated.
- Rows in HBase are “deeper” than in relational DBs
  - In relational databases, rows are structured with a fixed number of columns, each with a specific data type. This structure is like a fixed array, where each position has a defined purpose and type.
  - In contrast, HBase rows are more flexible and can be seen as a two-level nested dictionary. This means that each row can have a varying number of columns, and each column can have multiple versions, often stored with metadata such as timestamps. This flexibility allows HBase to handle more complex and dynamic data structures.
- Applications
  - HBase is well-suited for storing versioned website data, where each version of a webpage can be stored as a separate entry within a row, allowing for easy retrieval and comparison of different versions.
  - It can also be used to store a wiki, where each page can be represented as a row, and edits or revisions can be stored as different versions within that row. This makes it easy to track changes and manage content over time.

## 8 / 25: Example 2: Storing a Wiki

### Example 2: Storing a Wiki

- Wiki (e.g., Wikipedia)
  - Contains pages
  - Each page has a title, article text varies over time
- HBase data model
  - Table name → wikipedia
  - Row → entire wiki page
  - Row keys → wiki identifier (e.g., title or URL)
  - Column family → text
  - Column → " (empty)
  - Cell value → article text

	row keys (wiki page titles)	column family "text"
row (page)	"first page's title"	"": "Text of first page"
row (page)	"second page's title"	"": "Text of second page"

```
wikipedia_table = {
    # wiki id.
    'Home': {
        # Column family:column $\to$ value
        ':text': 'Welcome to the wiki!',
    },
    'Welcome page': {
        ... # More row data
    }
}
Database = Dict[str, Table]
database: Database = {'wikipedia':
    wiki_table}
(article, metadata) = \
    wiki_table['Home']['text']
```

- **Wiki (e.g., Wikipedia)**
  - A wiki is a collection of web pages that can be edited by users. Wikipedia is a well-known example.
  - Each page in a wiki has a unique title and contains article text. This text can change over time as users edit the page.
- **HBase data model**
  - HBase is a distributed database that uses a table-based structure to store data. Here, the table is named `wikipedia`.
  - Each row in the table represents an entire wiki page. This means that all the information about a single page is stored in one row.
  - **Row keys** are unique identifiers for each page, such as the page title or URL. This helps in quickly locating the page in the database.
  - **Column family** is a way to group related data. In this example, the column family is named `text`, which stores the article content.
  - **Column** is left empty here, indicating that the column family `text` contains only one type of data.
  - **Cell value** holds the actual article text, which is the main content of the wiki page.

The code snippet on the right shows how this data model is implemented in Python. It uses a dictionary to represent the `wikipedia` table, where each key is a wiki page identifier, and the value is another dictionary representing the column family and its content. The example shows how to access the text of the “Home” page.

## 9 / 25: Example 2: Storing a Wiki

### Example 2: Storing a Wiki

- **Add data**

- Columns don't need to be predefined when creating a table
- The column is defined as `text`

```
> put 'wikipedia', 'Home', 'text',
'Welcome!'
```

- **Query data**

- Specify the table name, the row key, and optionally a list of columns

```
> get 'wikipedia', 'Home', 'text'
text: timestamp=1295774833226,
value=Welcome!
```

- HBase returns the timestamp (ms since the epoch 01-01-1970 UTC)

	row keys (wiki page titles)	column family "text"
row (page)	"first page's title"	
row (page)	"second page's title"	

```
wikipedia_table = {
    # wiki id.
    'Home': {
        # Column family, column + value
        'text': 'Welcome to the wiki!',
    },
    'Welcome page': {
        ... # More row data
    }
}
Database = Dict[str, Table]
database: Database = {'wikipedia':
    wiki_table}
(queried_value, metadata) = \
    wiki_table['Home']['text']
```

9 / 25

- **Add data**

- In this example, we're using a database system where you don't need to define all the

columns in advance. This is particularly useful for storing data that might have varying structures, like a wiki.

- The column is defined as `text`, which means it can store any string of characters. This is flexible for storing different kinds of text data.
- The command `put 'wikipedia', 'Home', 'text', 'Welcome!'` is used to add data to the database. Here, ‘wikipedia’ is the table name, ‘Home’ is the row key, and ‘text’ is the column where the value ‘Welcome!’ is stored.

- **Query data**

- To retrieve data, you specify the table name, the row key, and optionally the column name. This allows you to access specific pieces of data efficiently.
- The command `get 'wikipedia', 'Home', 'text'` retrieves the value stored in the ‘text’ column for the ‘Home’ row. The output includes a timestamp, which indicates when the data was last updated. This timestamp is in milliseconds since January 1, 1970, UTC, a common format in computing known as Unix time.

- **Python Representation**

- The right side of the slide shows a Python dictionary representation of the same data. This helps in understanding how the data is structured programmatically.
- The `wikipedia_table` dictionary uses a nested structure where each key is a row identifier (like ‘Home’), and each value is another dictionary representing columns and their values. This mirrors the flexible, schema-less nature of the database.

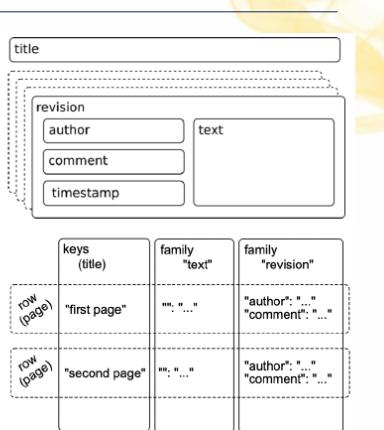
- **Context**

- This slide illustrates how a NoSQL database like HBase can be used to store and retrieve data without a fixed schema, making it ideal for applications like wikis where the data structure can vary widely.

## 10 / 25: Example 2: Improved Wiki

### Example 2: Improved Wiki

- **Improved wiki using versioning**
- A page
  - Uniquely identified by title
  - Can have multiple revisions
- A revision
  - Made by an author
  - Optionally contains a commit comment
  - Identified by timestamp
  - Contains text
- **HBase data model**
- Add family column “revision” with multiple columns
  - E.g., author, comment,
- Timestamp automatically binds article text and metadata
- Title not part of revision
  - Fixed and uniquely identifies page (like a primary key)
  - To change title, re-write entire row



- 
- **Improved wiki using versioning**
    - This concept involves enhancing a wiki system by incorporating version control. Versioning allows tracking changes over time, making it easier to manage and revert to previous states if needed.
  - **A page**
    - Each page in the wiki is *uniquely identified by its title*. This means that no two pages can have the same title, ensuring that each page is distinct.
    - Pages can have *multiple revisions*, which means that as changes are made, new versions of the page are created and stored.
  - **A revision**
    - Each revision is *made by an author*, indicating who made the changes.
    - Revisions can *optionally contain a commit comment*, which is a brief note explaining what changes were made and why.
    - Revisions are *identified by a timestamp*, providing a chronological order of changes.
    - Each revision *contains text*, which is the actual content of the page at that point in time.
  - **HBase data model**
    - In this model, a *family column named “revision”* is added, which can have multiple sub-columns like author and comment. This structure helps organize the data related to each revision.
    - The *timestamp automatically binds the article text and metadata*, ensuring that each piece of information is associated with the correct version of the page.
  - **Title not part of revision**
    - The title is *fixed and uniquely identifies the page*, similar to a primary key in databases. This means the title remains constant even as the content changes.
    - If the title needs to be changed, the *entire row must be rewritten*, which is a significant operation because it involves updating the unique identifier of the page.

## 11 / 25: Data in Tabular Form

### Data in Tabular Form

Key	Name		Home		Office	
	First	Last	Phone	Email	Phone	Email
101	Florian	Krepsbach	555-1212	florian@wobegon.org	666-1212	fk@phc.com
102	Marilyn	Tollerud	555-1213		666-1213	
103	Pastor	Inqvist			555-1214	inqvist@wel.org

- Fundamental operations
  - CREATE table, families
  - PUT table, rowid, family:column, value
  - PUT table, rowid, whole-row
  - GET table, rowid
  - SCAN table (*WITH filters*)
  - DROP table



11 / 25

#### • Data in Tabular Form

- This section presents a table that organizes contact information for individuals. The table is structured with columns for different types of data, such as names and contact details, and rows for each individual.
- **Columns:** The table includes columns for *Key*, *First Name*, *Last Name*, *Home Phone*, *Home Email*, *Office Phone*, and *Office Email*. These columns help categorize the data for easy access and understanding.
- **Rows:** Each row represents a unique individual, identified by a *Key*. For example, Florian Krepsbach has a home phone number and two email addresses, while Marilyn Tollerud has phone numbers but no email listed.
- **Missing Data:** Notice that some fields are empty, indicating missing data. For instance, Pastor Inqvist does not have a home phone or email listed, which is common in real-world datasets.

#### • Fundamental Operations

- **CREATE table, families:** This operation is used to create a new table, which is a structured format to store data. “Families” refers to groups of related columns.
- **PUT table, rowid, family:column, value:** This operation allows you to insert or update a specific value in the table. You specify the table, the row identifier (rowid), the column (within a family), and the value to be inserted.
- **PUT table, rowid, whole-row:** This is similar to the previous operation but allows you to insert or update an entire row at once, rather than individual columns.
- **GET table, rowid:** This operation retrieves data from the table for a specific row, identified by the rowid. It’s useful for accessing all information related to a particular entry.

- **SCAN table (*WITH filters*):** This operation is used to search through the table, potentially using filters to narrow down the results. Filters can be based on specific criteria, such as finding all entries with a certain email domain.
- **DROP table:** This operation deletes an entire table, removing all data and structure associated with it. It's a powerful operation that should be used with caution.

## 12 / 25: Data in Tabular Form

**Data in Tabular Form**

Name	Home	Office	Social	Key	First	Last	Phone	Email	Phone	Email	FacebookID
florian@wobegon.org	666-1212	fk@phc.com	-	101	Florian	Garfield	Krepsbach	555-1212	555-1212	Inqvist	-
555-1214	inqvist@wel.org	-		102	Marilyn	-	Tollerud	555-1213	666-1213	Pastor	-

... :::::{.column width=20%}

New columns can be added at runtime

... :::::{.column width=50%}

... :::::{.column width=20%}

Column families cannot be added at runtime

... :::::

```
Table People(Name, Home, Office)
{
    101: {
        Timestamp: T403;
        Name: {"First": "Florian", Middle: "Garfield", Last: "Krepsbach"},
        Home: {"Phone": "555-1212", Email: "florian@wobegon.org"},
        Office: {"Phone": "666-1212", Email: "fk@phc.com"}
    },
    102: {
        Timestamp: T593;
        Name: {"First": "Marilyn", Last: "Tollerud"},
        Home: {"Phone": "555-1213"},
        Office: {"Phone": "666-1213"}
    }
}
SCIENCE ACADEMY
```

12 / 25

- **Data in Tabular Form:** This section presents a table that organizes data in a structured format, making it easier to read and analyze. The table includes columns for personal information such as *Name*, *Home*, *Office*, and *Social* details. Each row represents a unique individual identified by a *Key*.

- **Columns Explained:**

- **Key:** A unique identifier for each person.
- **Name:** Divided into *First*, *Middle*, and *Last* names. Note that some entries may not have a middle name, indicated by a dash.
- **Home and Office:** These columns contain contact information, including *Phone* numbers and *Email* addresses. Some entries might be missing certain details.
- **Social:** This column is highlighted in red, indicating it might be a special category. It includes a *FacebookID*, but all entries currently show a dash, meaning no data is available.

- **Column Management:**

- **Adding New Columns:** The green text suggests that new columns can be added at runtime, allowing for flexibility in data management.

- **Column Families:** The red text indicates that column families, which are groups of related columns, cannot be added at runtime. This suggests a limitation in how data can be structured dynamically.

- **Data Representation in Code:**

- The code snippet shows how the data is structured programmatically. Each entry is represented as an object with a *Timestamp* and nested objects for *Name*, *Home*, and *Office* details.
- This representation highlights the hierarchical nature of the data, where each person's information is encapsulated within their unique key.

This slide provides a comprehensive view of how data can be organized and manipulated in a tabular format, emphasizing both flexibility and constraints in data management.

## 13 / 25: Nested Data Representation

### Nested Data Representation

Key	Name		Home		Office	
	First	Last	Phone	Email	Phone	Email
101	Florian	Krepsbach	555-1212	florian@wobegon.org	666-1212	fk@phc.com
102	Marilyn	Tollerud	555-1213		666-1213	
103	Pastor	Inqvist			555-1214	inqvist@wel.org

```
GET People:101
{
    Timestamp: T403;
    Name: {First:"Florian", Last="Krepsbach"},
    Home: {Phone="555-1212", Email="florian@wobegon.org"},
    Office: {Phone="666-1212", Email="fk@phc.com"}
}

GET People:101:Name
{First="Florian", Last="Krepsbach"}

GET People:101:Name:First
"Florian"
```

- **Nested Data Representation:** This slide introduces the concept of nested data representation, which is a way to organize data in a hierarchical structure. This is particularly useful when dealing with complex data that has multiple levels of related information.
- **Table Explanation:** The table shows a list of people with their contact information. Each person has a unique **Key** (101, 102, 103) and associated details such as **Name** (split into **First** and **Last**), and contact information for **Home** and **Office** (including **Phone** and **Email**).
- **Data Retrieval Example:** The slide provides examples of how to retrieve specific pieces of information using a nested data structure.
  - The command `GET People:101` retrieves all information for the person with Key 101,

showing a structured format with **Timestamp**, **Name**, **Home**, and **Office** details.

- The command `GET People:101:Name` retrieves just the **Name** object, which includes the **First** and **Last** names.
- The command `GET People:101:Name:First` drills down further to retrieve only the **First** name, “Florian”.
- **Importance of Nested Structures:** This approach is beneficial for efficiently accessing and managing data, especially when dealing with large datasets. It allows for precise queries and can help in reducing data redundancy by organizing related information together.
- **Practical Application:** Understanding nested data representation is crucial for working with databases and APIs, where data is often stored and accessed in a hierarchical manner. This knowledge is essential for tasks such as data retrieval, data manipulation, and ensuring data integrity.

## 14 / 25: Column Family vs Column

### Column Family vs Column

- **Adding a column**
  - Cheap
  - Done at run-time
- **Adding a column family**
  - Not at run-time
  - Requires table copy (expensive)
  - Indicates data storage method
    - Easy to add: map
    - Hard to add: static array
  - E.g., mongoDB document vs Relational DB column
- **Why differentiate column families vs columns?**
  - Why not store all row data in one column family?
  - Each column family configured independently, e.g.,
    - Compression
    - Performance tuning
    - Stored together in files
  - Designed for specific data types
    - E.g., timestamped web data for search engine

- **Adding a column**
  - *Cheap:* Adding a new column to a database is inexpensive in terms of resources and time.
  - *Done at run-time:* You can add columns while the database is running, without needing to stop or restart it.
- **Adding a column family**
  - *Not at run-time:* Unlike columns, adding a new column family cannot be done while the database is actively running.
  - *Requires table copy (expensive):* To add a column family, you need to create a copy of the table, which is resource-intensive.

- 
- *Indicates data storage method:* The ease of adding a column family depends on how data is stored.
    - \* *Easy to add: map:* In databases that use a map-like structure, adding a column family is simpler.
    - \* *Hard to add: static array:* In databases using static arrays, adding a column family is more complex.
    - \* *E.g., mongoDB document vs Relational DB column:* MongoDB, which uses a flexible document model, contrasts with traditional relational databases where adding columns can be more rigid.
  - **Why differentiate column families vs columns?**
    - *Why not store all row data in one column family?:* Storing all data in a single column family might seem simpler, but it limits flexibility.
    - *Each column family configured independently:* Different column families can have unique settings.
      - \* *Compression:* You can apply different compression techniques to each column family.
      - \* *Performance tuning:* Each column family can be optimized for performance based on its specific needs.
      - \* *Stored together in files:* Data in a column family is stored together, which can improve access speed.
    - *Designed for specific data types:* Column families can be tailored for particular types of data.
      - \* *E.g., timestamped web data for search engine:* For example, a column family might be optimized for handling large volumes of timestamped data, which is useful for search engines.

## 15 / 25: Consistency Model

### Consistency Model

- **Atomicity**
  - Update entire rows atomically or not at all
  - Independent of column count
- **Consistency**
  - GET returns a complete row from the table's history
    - Weak/ eventual consistency
    - Check timestamp for certainty
  - SCAN
    - Includes all data written before scan
    - May include updates since start
- **Isolation**
  - Concurrent vs sequential semantics
  - Not guaranteed beyond a single row
  - Row is the atom of information
- **Durability**
  - Successful writes are durable on disk

- **Consistency Model**
  - **Atomicity**
    - \* This concept ensures that when you update a row in a database, the update happens completely or not at all. This means that if something goes wrong during the update, the database will not be left in a partial state. It's like flipping a switch; it either happens or it doesn't, regardless of how many columns are in the row.
  - **Consistency**
    - \* When you perform a GET operation, it retrieves a complete row from the database's history. This can be tricky because sometimes the data might not be the most recent due to *weak* or *eventual consistency*. To be sure of the data's freshness, you can check the timestamp. For SCAN operations, they include all data written before the scan started, but might also include updates that happened during the scan.
  - **Isolation**
    - \* This refers to how database transactions are handled when multiple operations occur at the same time. It ensures that transactions appear to be executed in a sequence, even if they are happening concurrently. However, this isolation is only guaranteed for a single row, meaning that the row is the smallest unit of data that maintains this property.
  - **Durability**
    - \* Once a write operation is successfully completed, the data is stored permanently on disk. This means that even if the system crashes, the data will not be lost, ensuring that your information is safe and persistent.

## 16 / 25: Checking for Row or Column Existence

### Checking for Row or Column Existence

- HBase uses Bloom filters to check row or column existence
  - Acts like a cache for keys
  - Track presence without querying
- **Hashset complexity**
  - Unbounded space for data storage
  - No false positives
  - $O(1)$  average/amortized time
- **Bloom filter implementation**
  - Probabilistic hash set
  - Array of bits initially set to 0
  - Hash new data, set bits to 1
  - To test data presence, compute hash, check bits
    - All bits 0: data not seen
    - All bits 1: likely seen (false positive possible)
- **Bloom filter complexity**
  - Constant space usage
  - False positives possible (no false negatives)
  - $O(1)$  time complexity

- **HBase uses Bloom filters to check row or column existence**

- HBase, a distributed database, uses Bloom filters as a tool to efficiently determine if a row or column exists.
- Think of Bloom filters as a *cache* for keys, which helps in quickly checking the presence of data without having to perform a full query on the database.
- **Hashset complexity**
  - A hashset is a data structure that can store data with unbounded space, meaning it can grow as needed to accommodate more data.
  - It guarantees no false positives, meaning if it says an item is present, it definitely is.
  - The average time to check for an item in a hashset is  $O(1)$ , which is very fast.
- **Bloom filter implementation**
  - A Bloom filter is a *probabilistic* data structure, meaning it can sometimes give incorrect results (false positives).
  - It starts with an array of bits, all set to 0. When new data is added, it is hashed, and certain bits are set to 1.
  - To check if data is present, the data is hashed again, and the corresponding bits are checked. If all bits are 0, the data is definitely not present. If all bits are 1, the data is likely present, but there could be a false positive.
- **Bloom filter complexity**
  - Bloom filters use a fixed amount of space, regardless of the amount of data, which makes them efficient.
  - They can produce false positives, meaning they might say data is present when it is not, but they never produce false negatives.
  - Checking for data presence in a Bloom filter is very fast, with a time complexity of  $O(1)$ .

## 17 / 25: Write-Ahead Log (WAL)

### Write-Ahead Log (WAL)

- Technique used by DBs
  - Provide atomicity and durability
  - Protect against node failures
  - Equivalent to journaling in file systems
- HBase and Postgres use WAL
- **WAL mechanics**
- Updated state of tables:
  - Not written to disk immediately
  - Buffered in memory
  - Written to disk as checkpoints periodically
- **Problem**
  - Server crash during this period loses state
- **Solution**
  - Use append-only disk-resident data structure
  - Log operations since last checkpoint in WAL
  - Clear WAL when tables stored to disk



---

- **Write-Ahead Log (WAL)**

- *Technique used by DBs:* WAL is a method used by databases to ensure that data is not lost and remains consistent even if something goes wrong, like a power failure or system crash.
  - \* **Provide atomicity and durability:** These are two important properties in databases. Atomicity means that a series of operations are completed fully or not at all, while durability ensures that once a transaction is committed, it will remain so, even in the event of a crash.
  - \* **Protect against node failures:** WAL helps in safeguarding data when a part of the system fails.
  - \* *Equivalent to journaling in file systems:* Just like journaling helps file systems recover from crashes, WAL helps databases do the same.

- **HBase and Postgres use WAL:** These are examples of systems that implement WAL to maintain data integrity and reliability.

- **WAL mechanics**

- *Updated state of tables:* When changes are made to the database, they aren't immediately saved to the disk.
  - \* **Not written to disk immediately:** Instead of writing every change right away, the changes are temporarily stored in memory.
  - \* **Buffered in memory:** This temporary storage helps in speeding up operations.
  - \* **Written to disk as checkpoints periodically:** At certain intervals, these changes are saved to the disk in batches, known as checkpoints.

- **Problem**

- *Server crash during this period loses state:* If the server crashes before the changes are written to the disk, the data in memory is lost.

- **Solution**

- *Use append-only disk-resident data structure:* WAL logs every change made to the database in a sequential manner.
- **Log operations since last checkpoint in WAL:** This log keeps track of all changes made since the last time data was saved to the disk.
- **Clear WAL when tables stored to disk:** Once the data is safely written to the disk, the log can be cleared.
- **Use WAL to recover state if server crashes:** If a crash occurs, the database can use the WAL to restore the data to its last known good state.

- **Disable WAL during big import jobs to improve performance**

- *Trade off disaster recovery protection for speed:* Sometimes, when importing large amounts of data, it might be beneficial to turn off WAL to make the process faster, but this means losing the safety net that WAL provides in case of a crash.

## 18 / 25: Storing Variable-Length Data in Dbs

### Storing Variable-Length Data in Dbs

#### SQL Table

```
People(ID: Integer, FirstName: CHAR[20], LastName: CHAR[20], Phone: CHAR[8])
UPDATE People SET Phone="555-3434" WHERE ID=403;
```

ID	FirstName	LastName	Phone
101	Florian	Krepsbach	555-3434
102	Marilyn	Tollerud	555-1213
103	Pastor	Ingvist	555-1214

- Each row: 52 bytes
- Move to next row: fseek(file,+52)
- Get to Row 401: fseek(file, 401\*52)
- Overwrite data in place

#### HBase Table

```
People(ID, Name, Home, Office)
PUT People, 403, Home:Phone, 555-3434
```

```
{
  101: {
    Timestamp: T403,
    Name: {First:"Florian", Middle:"Garfield", Last:"Krepsbach"},
    Home: {Phone="555-1212", Email="florian@wobegon.org"},
    Office: {Phone="666-1212", Email="fk@phc.com"}
  },
  ...
}
```

Need to use  
pointers



SCIENCE  
ACADEMY

18 / 25

#### • Storing Variable-Length Data in Dbs

- **SQL Table:** This example shows how data is stored in a traditional SQL database. The table `People` has columns for `ID`, `FirstName`, `LastName`, and `Phone`, with fixed character lengths. For instance, `FirstName` and `LastName` are both set to `CHAR[20]`, meaning each entry in these columns will always occupy 20 characters, even if the name is shorter. This can lead to wasted space but simplifies data retrieval because each row is a fixed size (52 bytes in this case). The `UPDATE` statement demonstrates how to change a phone number for a specific `ID`. The database uses file operations like `fseek` to navigate and update specific rows efficiently.
- **HBase Table:** In contrast, HBase, a NoSQL database, handles data differently. It allows for more flexible, variable-length data storage. The `PUT` command updates the phone number for a specific `ID` without needing fixed-length fields. Data is stored in a more complex structure, often in JSON-like formats, allowing for nested information such as `Name`, `Home`, and `Office` details. This flexibility comes at the cost of needing pointers to manage data locations, as the data isn't stored in a fixed-size format. This approach is beneficial for handling large volumes of diverse data.

## 19 / 25: HBase Implementation

### HBase Implementation

- **How to store the web on disk?**
- **HBase is backed by HDFS**
  - Store each table (e.g., Wikipedia) in one file
  - “One file” means one gigantic file stored in HDFS
    - HDFS splits/replicates file into blocks on different servers
- Idea in several steps:
  - **Idea 1: Put entire table in one file**
    - Overwrite file with any cell change
    - Too slow
  - Idea 2: One file + WAL
    - Better, but doesn't scale to large data
  - Idea 3: One file per column family + WAL
    - Getting better!
  - Idea 4: Partition table into regions by key
    - Region = chunk of rows [a, b)
    - Regions never overlap



19 / 25

- **How to store the web on disk?**
  - When we talk about storing the web on disk, we're referring to how we can efficiently save and manage vast amounts of web data, like the content of websites, in a way that allows for quick access and updates.
- **HBase is backed by HDFS**
  - **Store each table (e.g., Wikipedia) in one file**
    - \* In HBase, each table, such as a collection of Wikipedia articles, is stored as a single large file in the Hadoop Distributed File System (HDFS).
    - \* **“One file” means one gigantic file stored in HDFS**
      - This file isn't just sitting on one server. Instead, HDFS breaks it into smaller pieces, called blocks, and spreads these blocks across multiple servers. This distribution helps with data redundancy and access speed.
  - **Idea in several steps:**
    - **Idea 1: Put entire table in one file**
      - \* Initially, the thought was to store the whole table in one file and update this file whenever any data changed. However, this approach was too slow because rewriting the entire file for small changes is inefficient.
    - **Idea 2: One file + WAL**
      - \* The next improvement was to use a Write-Ahead Log (WAL) alongside the file. This helped with data recovery but still wasn't efficient for large datasets.
    - **Idea 3: One file per column family + WAL**
      - \* By breaking the table into smaller parts called column families, each with its own file, performance improved. This approach allowed for more manageable updates and better organization.

---

- **Idea 4: Partition table into regions by key**

- \* Finally, the table is divided into regions based on row keys. Each region is a chunk of rows, ensuring that no two regions overlap. This partitioning allows for even better scalability and performance, as different regions can be managed independently.

## 20 / 25: Idea 1: Put the Table in a Single File

### Idea 1: Put the Table in a Single File

- How do we do the following operations?
  - CREATE, DELETE (easy / fast)
  - SCAN (easy / fast)
  - GET, PUT (difficult / slow)

```
Table People(Name, Home, Office) { 101: { Timestamp: T403; Name:  
{First="Florian", Middle="Garfield", Last="Krepsbach"}, Home:  
{Phone="555-1212", Email="florian@wobegon.org"}, Office:  
{Phone="666-1212", Email="fk@phc.com"} }, 102: { Timestamp: T593;  
Name: {First="Marilyn", Last="Tollerud"}, Home: {Phone="555-1213"},  
Office: {Phone="666-1213"} }, ... }
```

File "People"



20 / 25

- **Idea 1: Put the Table in a Single File**

- The concept here is to store all the data for a table in one file. This approach can simplify some operations but complicate others.

- **How do we do the following operations?**

- **CREATE, DELETE (easy / fast):**

- \* Adding or removing entries in a single file is straightforward. You can append new data or remove existing data without needing to manage multiple files or locations.

- **SCAN (easy / fast):**

- \* Scanning through the data is efficient because all the information is in one place. You can read through the file sequentially to find what you need.

- **GET, PUT (difficult / slow):**

- \* Retrieving (GET) or updating (PUT) specific entries can be slow. Since all data is in one file, you might have to search through a lot of unrelated data to find what you're looking for, especially if the file is large.

- **Table People(Name, Home, Office)**

- This is an example of how data might be structured in a single file. Each entry has a unique identifier (like 101 or 102) and contains detailed information about a person, such as their name, home, and office contact details.

- **Timestamp:** Each entry has a timestamp, which can be useful for tracking when the

data was last updated.

- **Name, Home, Office:** These are the categories of information stored for each person. They include nested details like phone numbers and email addresses.
- **File “People”**
  - This is the name of the file where all the data is stored. Having a single file can make it easier to manage and back up the data, but it can also become a bottleneck if the file grows too large or if many users need to access it simultaneously.

## 21 / 25: Idea 2: One File + WAL

### Idea 2: One File + WAL

#### Table People(Name, Home, Office)

PUT 101:Office:Phone = "555-3434" PUT 102:Home>Email = mt@yahoo.com

....

#### WAL for Table People

- Changes are applied only to the log file
- The resulting record is cached in memory
- Reads must consult both memory and disk

#### Memory Cache for Table People

101

102

**GET People:101**

**GET People:103**

**PUT People:101:Office:Phone = "555-3434"**



21 / 25

- **Idea 2: One File + WAL**

- This slide introduces the concept of using a single file along with a Write-Ahead Log (WAL) for managing data changes. The WAL is a technique used to ensure data integrity by recording changes before they are applied to the main database.

- **Table People(Name, Home, Office)**

- The example table, “People,” contains fields for Name, Home, and Office. This structure is used to illustrate how data is stored and modified.

- **PUT 101:Office:Phone = "555-3434"**

- This operation represents updating the phone number for the office of the person with ID 101. The “PUT” command is used to insert or update data in the table.

- **WAL for Table People**

- *Changes are applied only to the log file:* When a change is made, it is first recorded in the WAL. This ensures that even if a system failure occurs, the change can be recovered.
  - *The resulting record is cached in memory:* After logging, the change is also stored in memory for quick access.
  - *Reads must consult both memory and disk:* To retrieve data, the system checks both the

memory cache and the disk to ensure it has the most recent information.

- **Memory Cache for Table People**
  - The memory cache stores recent changes for quick access. In this example, IDs 101 and 102 are cached, meaning their data is readily available without needing to access the disk.
- **GET People:101**
  - This command retrieves the data for the person with ID 101. The system will check the memory cache first and then the disk if necessary.
- **GET People:103**
  - Attempting to get data for ID 103, which is not in the cache, will require accessing the disk to retrieve the information.
- **PUT People:101:Office:Phone = “555-3434”**
  - This detailed example shows the structure of the data for ID 101, including timestamps and various fields. It highlights how data is organized and stored, with timestamps indicating when each piece of information was last updated.

## 22 / 25: Idea 2 Requires Periodic Table Update

### Idea 2 Requires Periodic Table Update

```
101: {Timestamp: T403;Name: {First="Florian", Middle="Garfield",
Last="Krepsbach"},Home: {Phone="555-1212",
Email="florian@wobegon.org"},Office: {Phone="666-1212",
Email="fk@phc.com"}}, 102: {Timestamp: T593;Name: { First="Marilyn",
Last="Tollerud"},Home: { Phone="555-1213" },Office: { Phone="666-1213"
}}, ...
```

#### Table for People on Disk (Old)

```
PUT 101:Office:Phone = "555-3434" PUT 102:Home:Email = mt@yahoo.com
```

```
...
```

#### WAL for Table People:

```
101: {Timestamp: T403;Name: {First="Florian", Middle="Garfield",
Last="Krepsbach"},Home: {Phone="555-1212",
Email="florian@wobegon.org"},Office: {Phone="555-3434",
Email="fk@phc.com"}},102: {Timestamp: T593;Name: { First="Marilyn",
Last="Tollerud"},Home: { Phone="555-1213", Email="my@yahoo.com" },
...}
```

#### Table for People on Disk (New)

22 / 25

- **Idea 2 Requires Periodic Table Update:** This slide discusses the need to periodically update a data table, which is a common requirement in database management systems. The example provided involves a table of people with their contact information.
- **Table for People on Disk (Old):** Initially, the table contains entries for individuals with their contact details, such as phone numbers and email addresses. Each entry is identified by a unique ID (e.g., 101, 102).
- **PUT Operations:** These operations represent updates to the existing data. For instance,



---

the phone number for the office of the person with ID 101 is changed, and an email address is added for the home contact of the person with ID 102.

- **WAL for Table People:** WAL stands for Write-Ahead Logging, a technique used to ensure data integrity. It logs changes before they are applied to the main table. Here, the updated phone number and email address are highlighted, showing the changes made to the original data.
- **Table for People on Disk (New):** After applying all updates, a new version of the table is written to disk. This process involves writing out the updated table, deleting the log and memory cache, and starting fresh. This is akin to how caching works in a memory hierarchy, where data is periodically refreshed to ensure consistency and accuracy.

## 23 / 25: Idea 3: Partition by Column Family

### Idea 3: Partition by Column Family

Data for Column Family Name

#### Tables for People on Disk (Old)

PUT 101:Office:Phone = "555-3434" PUT 102:Home:Email = mt@yahoo.com

...

#### WAL for Table People

#### Tables for People on Disk (New)

- Write out a new copy of the table, with all of the changes applied
- Delete the log and memory cache
- Start over

Data for Column Family Home

Data for Column Family Office

Data for Column Family Home (Changed)

Data for Column Family Office (Changed)

DATA FOR COLUMN FAMILY NAME  
ACADEMY

23 / 25

- **Idea 3: Partition by Column Family**

This slide introduces the concept of partitioning data by column family, which is a method used in databases to organize and manage data more efficiently. A column family is a group of related data columns that are often accessed together. By partitioning data this way, it can improve performance and make data retrieval more efficient.

- **Tables for People on Disk (Old)**

This section shows how data was previously stored. For example, a phone number and an email address are stored with identifiers like "101:Office:Phone" and "102:Home:Email". This method can become inefficient as the data grows because all changes are logged and stored together, making it harder to manage and retrieve specific data quickly.

- **WAL for Table People**

---

WAL stands for Write-Ahead Logging, a technique used to ensure data integrity. It logs changes before they are applied to the database, which helps in recovering data in case of a failure. However, this can become cumbersome if the data is not well-organized.

- **Tables for People on Disk (New)**

The new approach involves writing out a new copy of the table with all changes applied, then deleting the log and memory cache to start fresh. This method helps in maintaining a clean and efficient database by ensuring that only the most recent data is stored.

- **Data for Column Family Home/Office/Name (Changed)**

By organizing data into column families like Home, Office, and Name, each with its own set of changes, it becomes easier to manage and retrieve specific data. This separation allows for more efficient data handling and reduces the complexity of managing large datasets.

- **Same scheme as before but split by column family**

The slide emphasizes that the same data organization scheme is used, but now it is split by column family. This highlights the importance of understanding the difference between column families and columns, as it can significantly impact the performance and efficiency of data management systems.

## 24 / 25: Idea 4: Split Into Regions

### Idea 4: Split Into Regions

Region 1: Keys 100-200

Region 2: Keys 100-200

Region 3: Keys 100-200

Region 4: Keys 100-200

Region Server

Region Master

Region Server

Region Server

Region Server

Transaction Log

Memory Cache

Table



24 / 25

- **Idea 4: Split Into Regions**

- The concept of splitting data into regions is crucial for managing large datasets efficiently. By dividing data into smaller, manageable parts, systems can handle requests more effectively and improve performance.

- **Region 1: Keys 100-200**

- Each region is defined by a range of keys. Here, Region 1 handles keys from 100 to 200.

---

This means any data with keys in this range will be managed by this specific region.

- **Region 2: Keys 100-200**
  - Similarly, Region 2 also manages keys from 100 to 200. This might be a typo, as typically each region would handle a unique range of keys to distribute the load evenly.
- **Region 3: Keys 100-200**
  - Again, Region 3 is listed with the same key range. In practice, each region should have a distinct range to avoid overlap and ensure efficient data distribution.
- **Region 4: Keys 100-200**
  - Like the previous regions, Region 4 is also shown with the same key range. This repetition suggests a need for clarification or correction in the key distribution.
- **Region Server**
  - A region server is responsible for managing one or more regions. It handles read and write requests for the data within its regions, ensuring data is stored and retrieved efficiently.
- **Region Master**
  - The region master oversees the region servers. It manages the distribution of regions across servers and ensures the system is balanced and running smoothly.
- **Transaction Log**
  - The transaction log records all changes made to the data. This is crucial for maintaining data integrity and recovering data in case of failures.
- **Memory Cache**
  - A memory cache temporarily stores frequently accessed data to speed up read operations. This helps reduce the time it takes to access data from disk storage.
- **Table**
  - Tables are the primary structure for storing data in a database. Each table can be split into regions to manage large datasets more effectively.
- **HBase Client**
  - The HBase client is the interface through which users interact with the HBase system. It sends requests to the region servers to read or write data.
- **(Detail of One Region Server)**
  - This section likely provides specifics about how a single region server operates, including its components and responsibilities.
- **Column Family Name**
  - Column families group related columns together in a table. This helps organize data and optimize storage and retrieval.
- **Column Family Home**
  - This is an example of a column family, possibly storing data related to home addresses or similar information.
- **Column Family Office**
  - Another example of a column family, potentially storing office-related data.
- **Where are the servers for table People?**
  - This question highlights the importance of knowing the physical or logical location of servers managing specific tables, like “People,” to optimize data access and management.
- **Access data in tables**
  - Accessing data efficiently is a key goal of splitting data into regions. By organizing data into regions and using region servers, systems can handle large volumes of data more effectively.

## Final HBase Data Layout



25 / 25

- **Final HBase Data Layout**

- *HBase* is a distributed, scalable, big data store that is part of the Hadoop ecosystem. It is designed to handle large amounts of data across many servers.
- **Data Layout:** In HBase, data is stored in a table-like format, but it is different from traditional relational databases. Instead of rows and columns, HBase uses a key-value store model.
- **Row Key:** Each row in an HBase table is identified by a unique row key. This key is crucial because it determines how data is distributed across the cluster.
- **Column Families:** Data in HBase is grouped into column families. Each family can contain multiple columns, and all columns within a family are stored together on disk, which optimizes read and write performance.
- **Timestamps:** HBase stores multiple versions of a cell, each with a unique timestamp. This allows for versioning and historical data retrieval.
- **Regions:** Tables are divided into regions, which are distributed across the cluster. This allows HBase to scale horizontally by adding more servers.
- *Understanding the final data layout in HBase is essential* for optimizing performance and ensuring efficient data retrieval and storage.