Lesson 5.1: NoSQL Databases

UNIVERSITY OF MARYLAND

UMD DATA605 - Big Data Systems

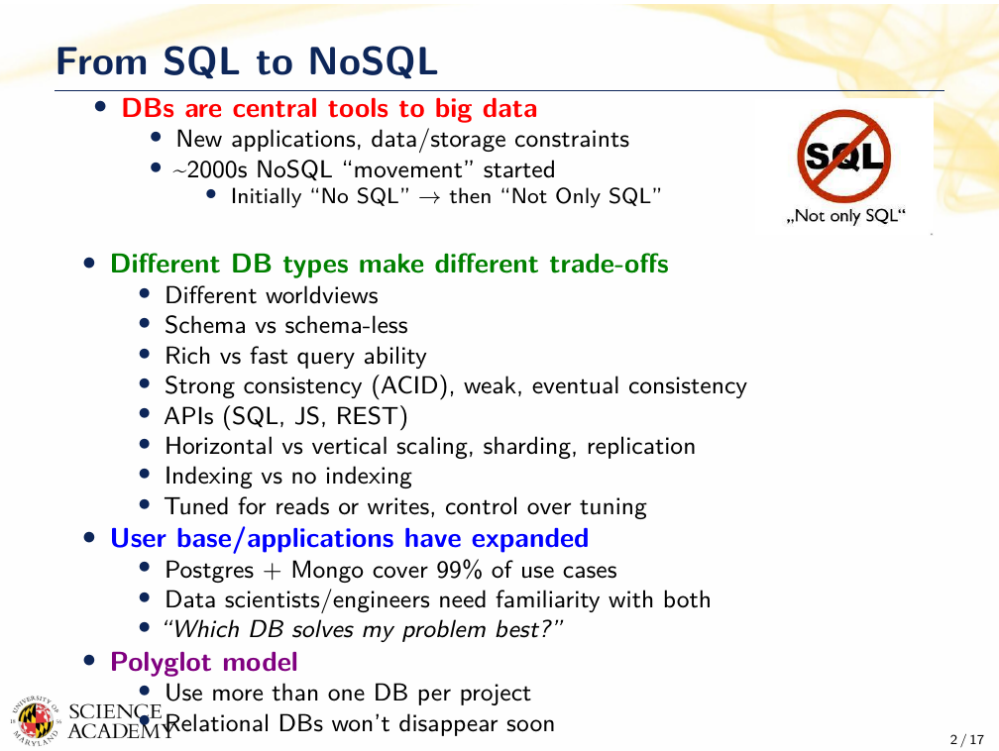
Lesson 5.1: NoSQL Databases

- **Instructor**: Dr. GP Saggese, gsaggese@umd.edu
- **References**:
  - Online tutorials
  - Silbershatz: Chap 10.2
  - Seven Databases in Seven Weeks, 2e

SCIENCE ACADEMY

1 / 17

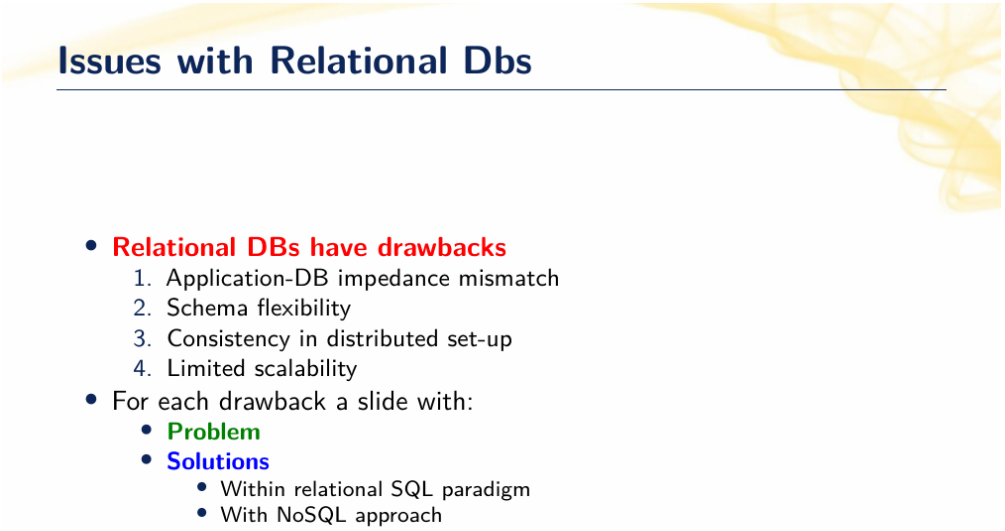
2 / 17: From SQL to NoSQL



- **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

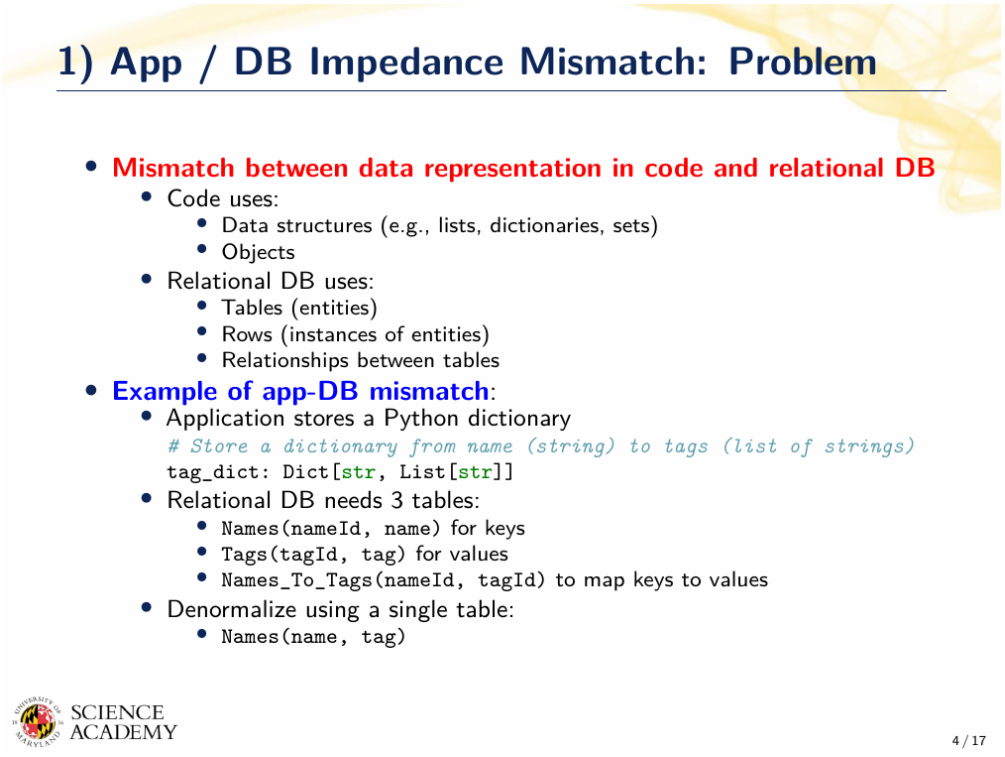


- **Issues with Relational Dbs**
- **Relational DBs have drawbacks**
  - **Application-DB impedance mismatch**
    - *Problem*: This refers to the difficulty in aligning the way data is structured in a relational database with how it is used in applications. Applications often use object-oriented programming, 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.

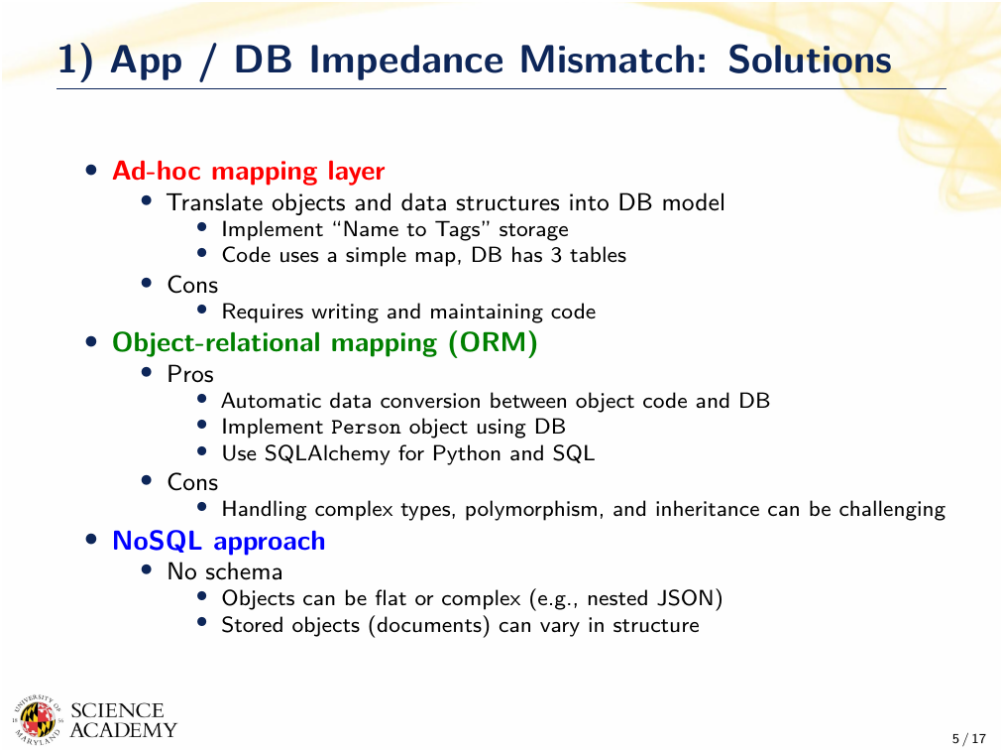
4 / 17: 1) App / DB Impedance Mismatch: Problem



- **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.

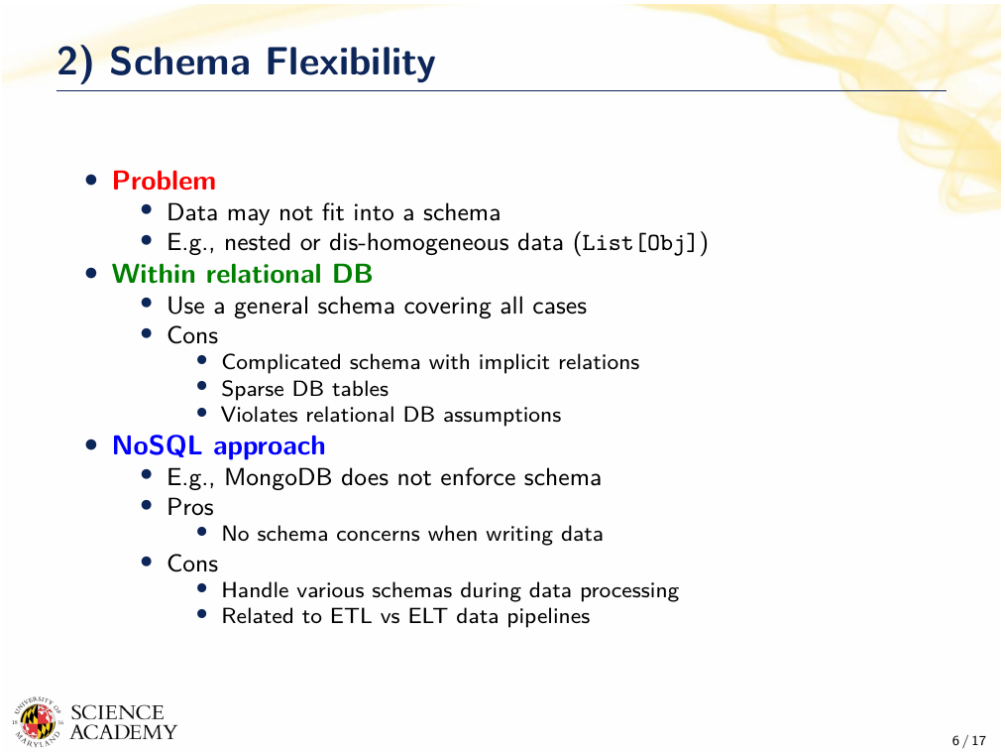
5 / 17: 1) App / DB Impedance Mismatch: Solutions



- **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.

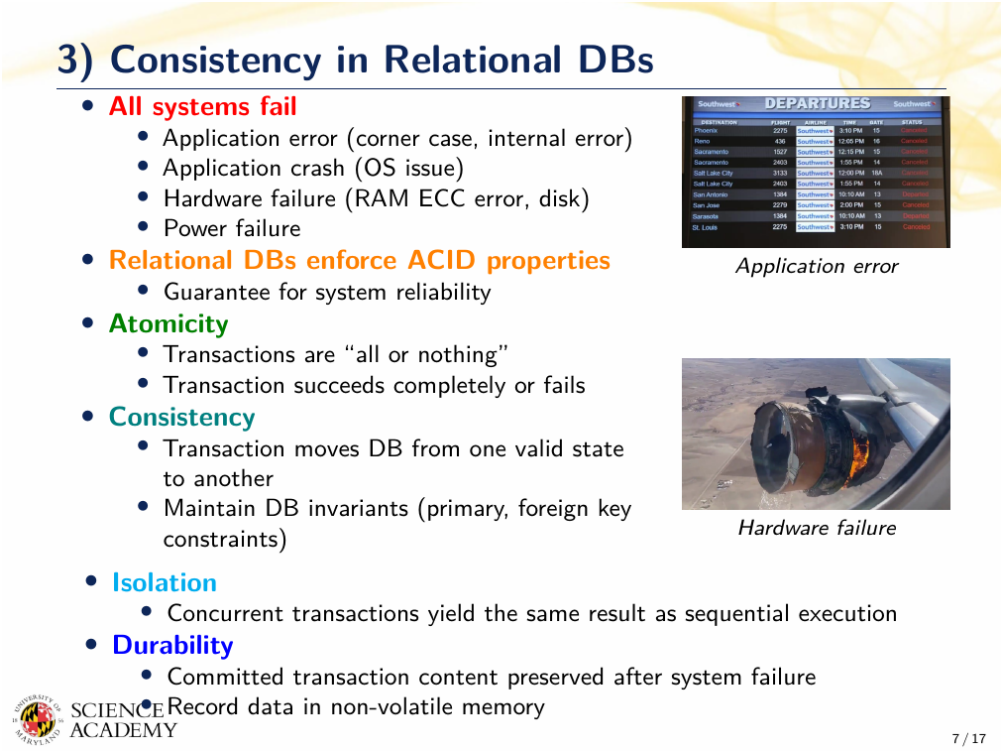
6 / 17: 2) Schema Flexibility



- **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.

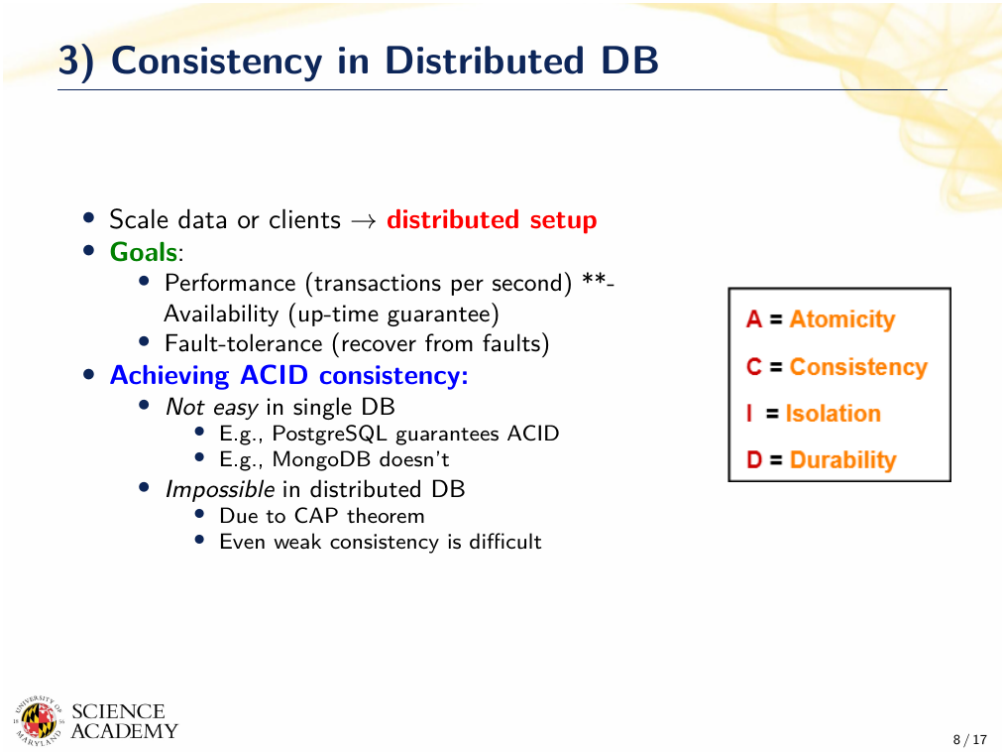
7 / 17: 3) Consistency in Relational DBs



- **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-

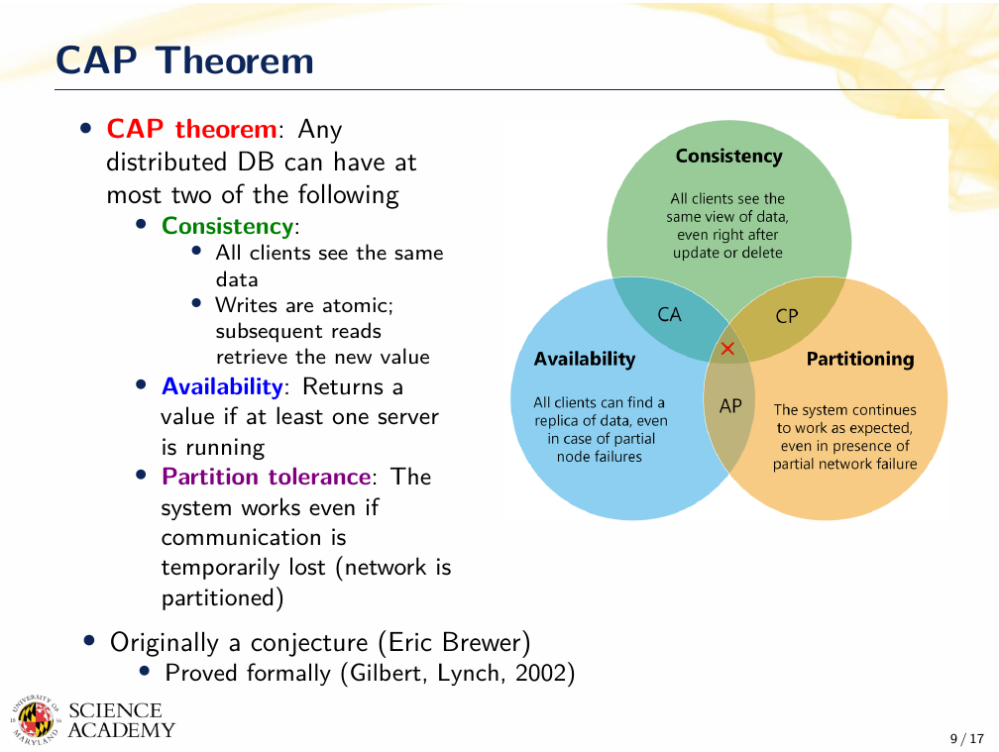
ing system or other software conflicts.
  - *Hardware failure*: Physical components like RAM or disks can malfunction, causing data issues or system crashes.
  - *Power failure*: Loss of electricity can abruptly stop operations, risking data loss or corruption.
- **Relational DBs enforce ACID properties**
  - 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



- **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.

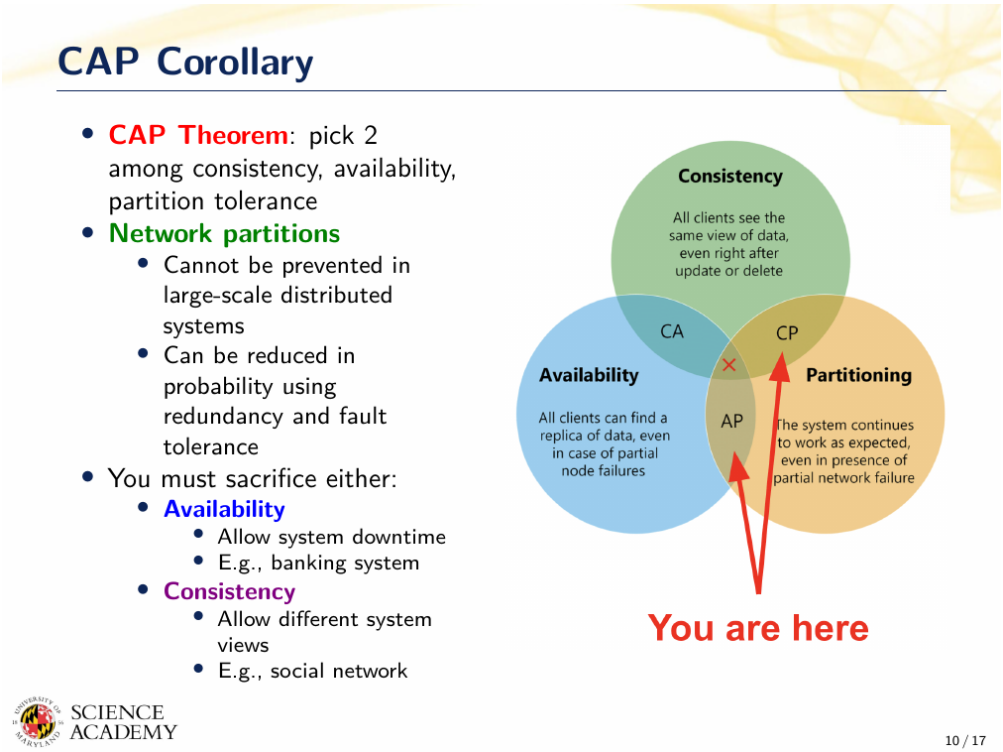
9 / 17: CAP Theorem



- **CAP theorem**: 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

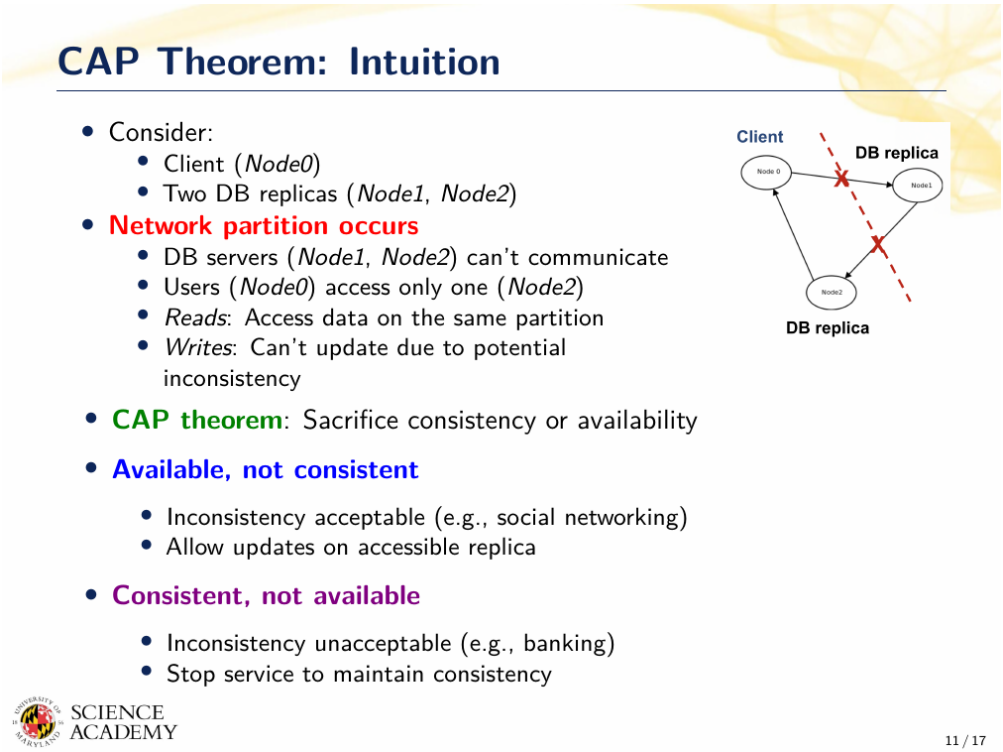


- **CAP Theorem**: This is a fundamental principle in distributed systems that states you can only achieve two out of the three following properties: *consistency*, *availability*, and *partition tolerance*. In simpler terms, when designing a distributed system, you have to make a trade-off because it's 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



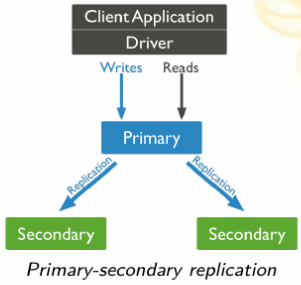
- **Consider:**
  - Imagine a scenario with a *client* (referred to as *Node0*) and two *database replicas* (referred to as *Node1* and *Node2*). These nodes are part of a distributed system where data is stored across multiple locations to ensure reliability and performance.
- **Network partition occurs:**
  - 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.

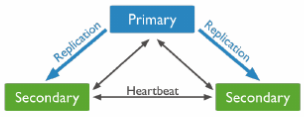
12 / 17: Replication Schemes

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)



Primary-secondary replication



Update-anywhere replication

SCIENCE ACADEMY

12 / 17

- **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**:

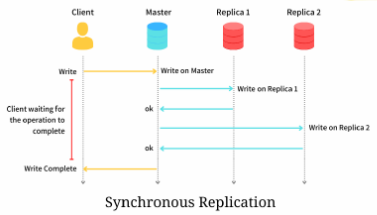
  - Also known as "multi-master replication," this scheme allows any server in the system to update data.
  - These updates are then shared with other servers, ensuring all replicas have the latest information.
  - This approach provides more flexibility and resilience compared to primary-secondary replication.

- **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

SCIENCE ACADEMY

13 / 17

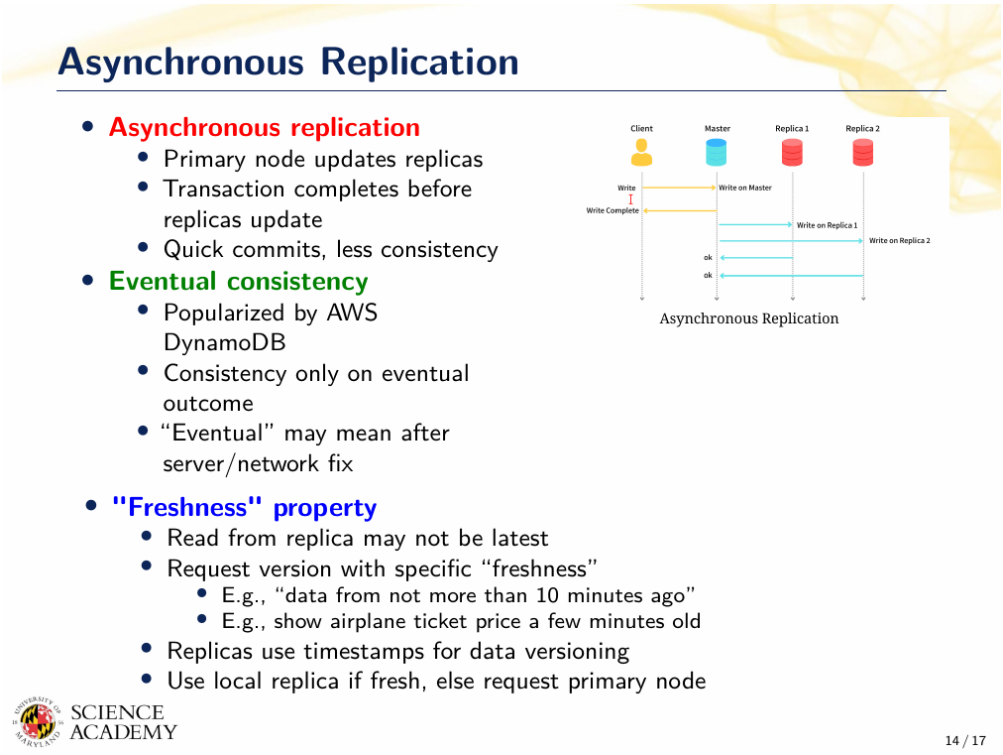
- **Synchronous replication**: This is a method where updates to data are immediately copied to all replicas as part of a single transaction. This ensures that all copies of the data are consistent at any given time. It's like making sure every copy of a book is updated with the same changes at the same 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.

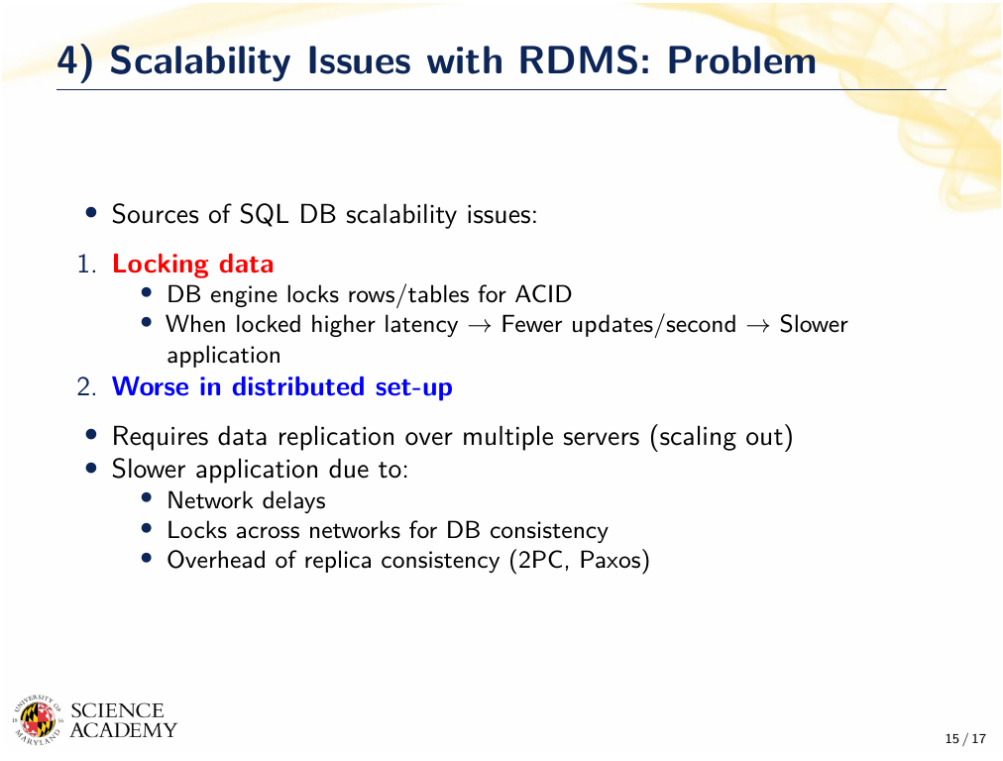
14 / 17: Asynchronous Replication



- **Asynchronous replication**
  - *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



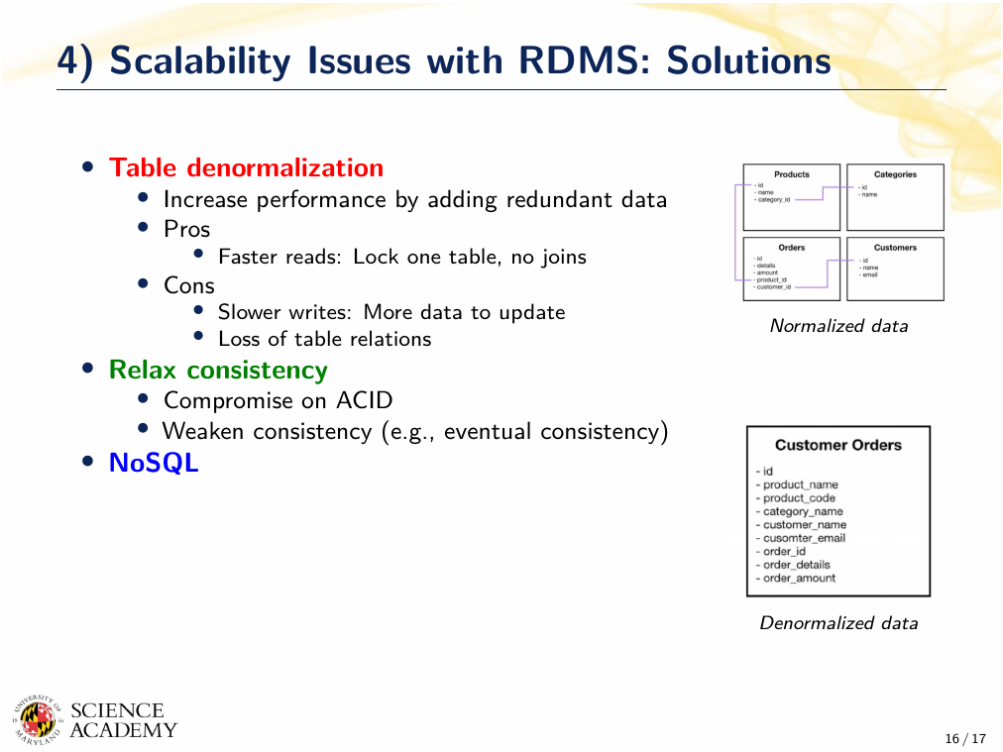
- **Scalability Issues with RDMS: Problem**

  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.

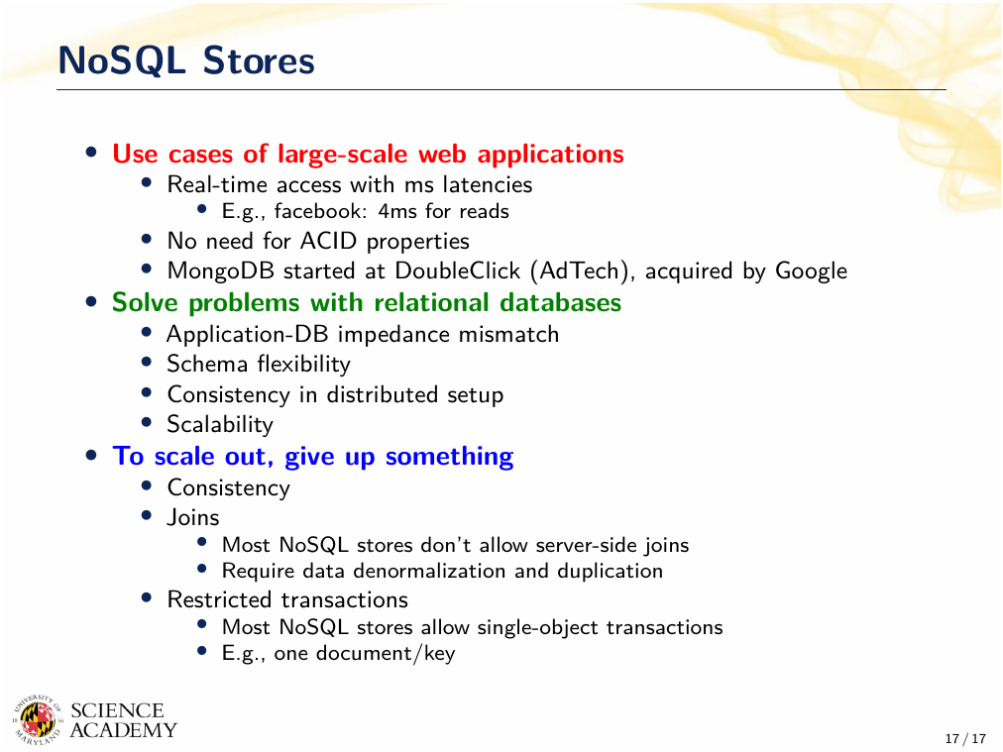
16 / 17: 4) Scalability Issues with RDMS: Solutions



- **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



- **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.