Lesson 5.3: Apache HBase

UMD DATA605 - Big Data Systems

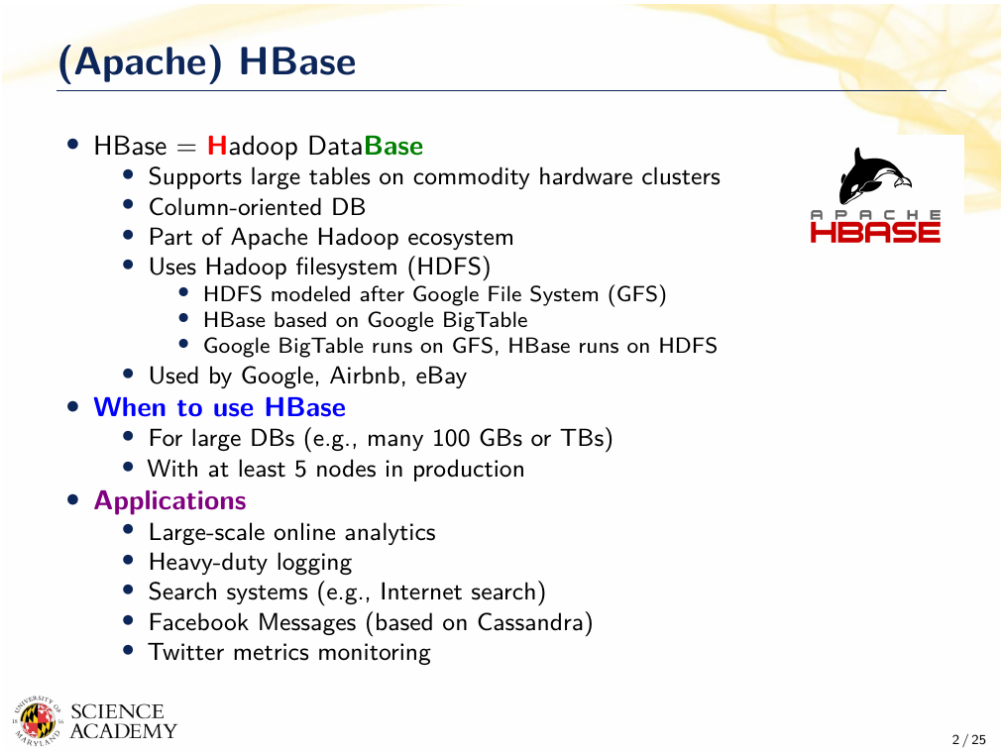
Lesson 5.3: Apache HBase

- **Instructor**: Dr. GP Saggese, 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

SCIENCE ACADEMY

1 / 25

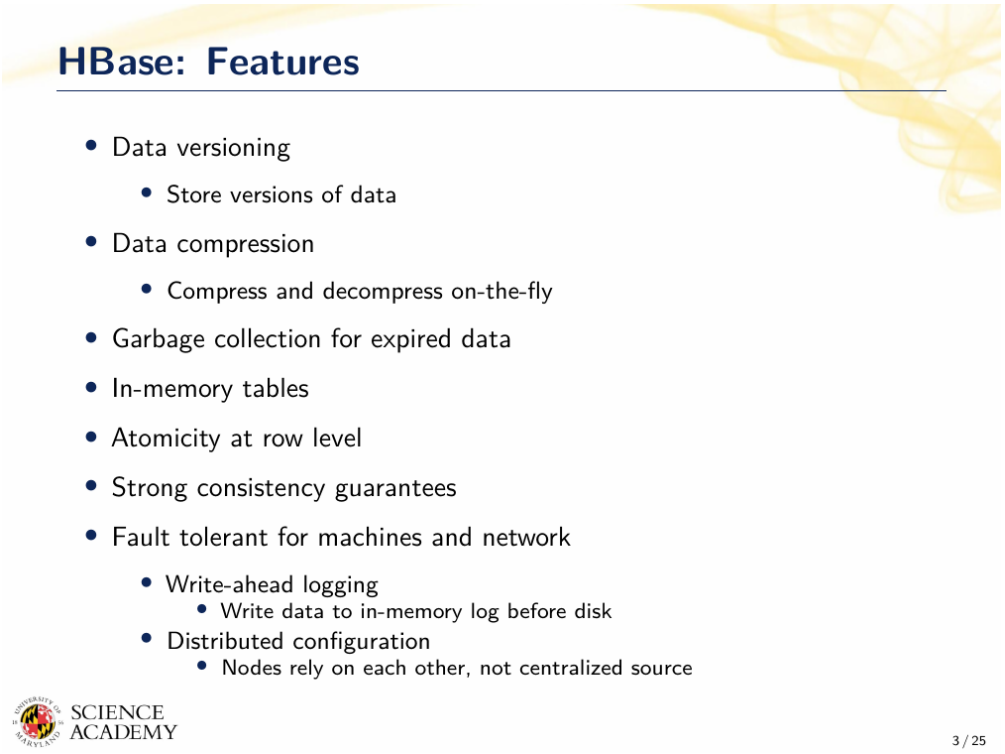
2 / 25: (Apache) HBase



- **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

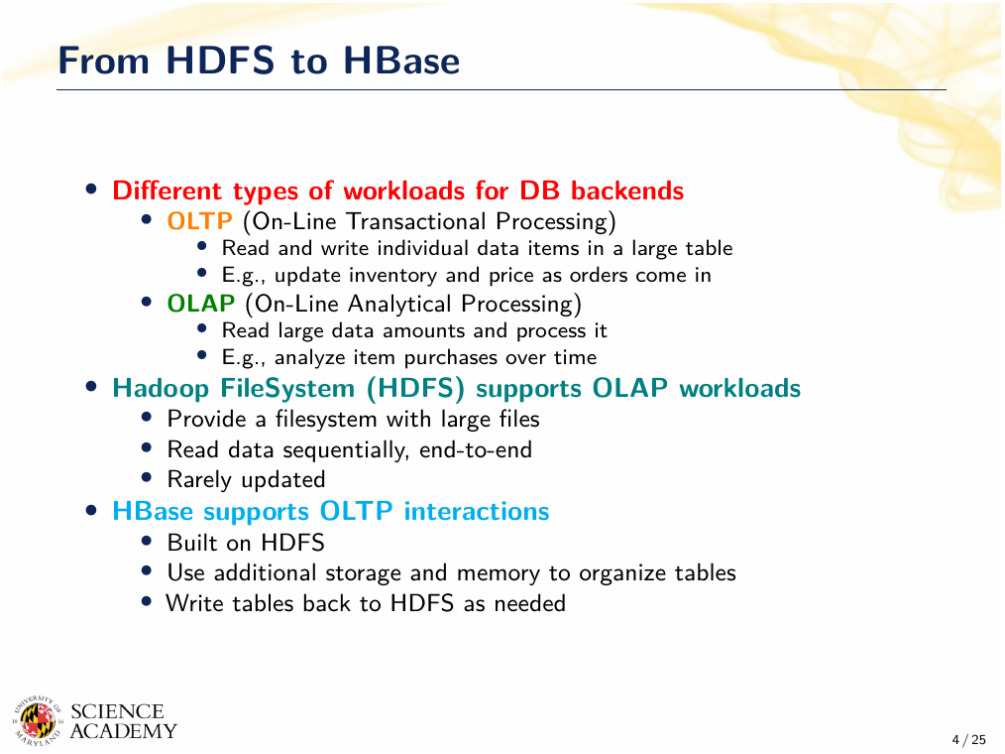


- **HBase: Features**
- **Data versioning**
  - HBase allows you to store multiple versions of the same data. This means that every time you update a piece of data, the previous version is not immediately discarded. Instead, it is kept for a certain period or number of versions, which can be useful for 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.

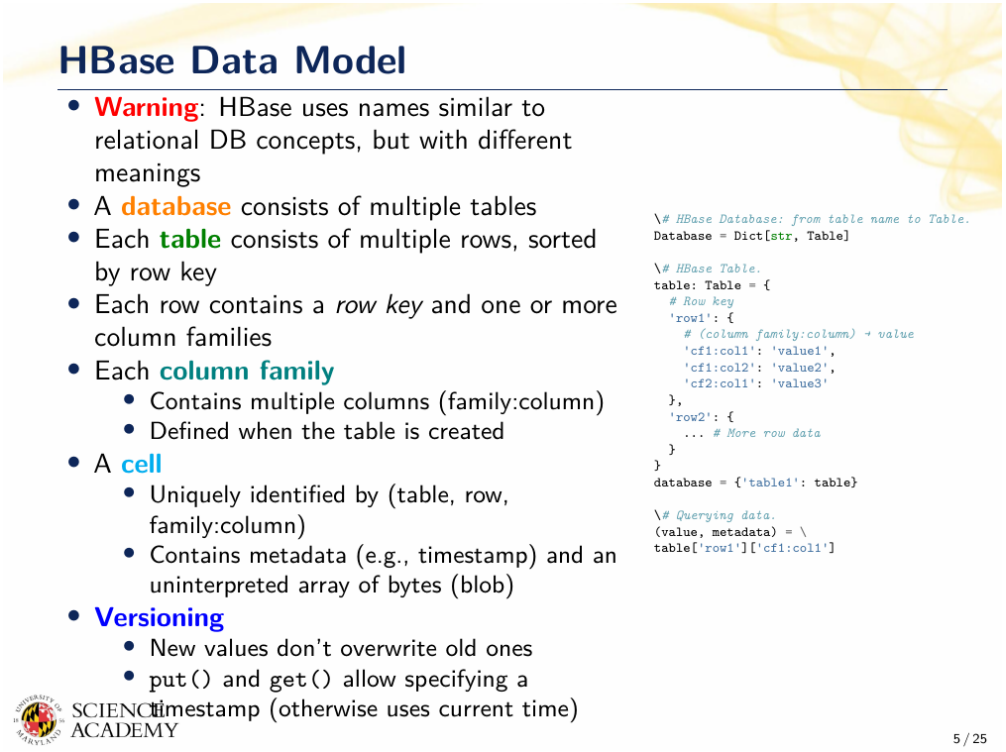
- **In-memory tables**
  - HBase can store tables in memory, which allows for faster data access and processing. This is particularly useful for applications that require quick read and write operations.
- **Atomicity at row level**
  - HBase ensures that operations on a single row are atomic, meaning 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



- **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

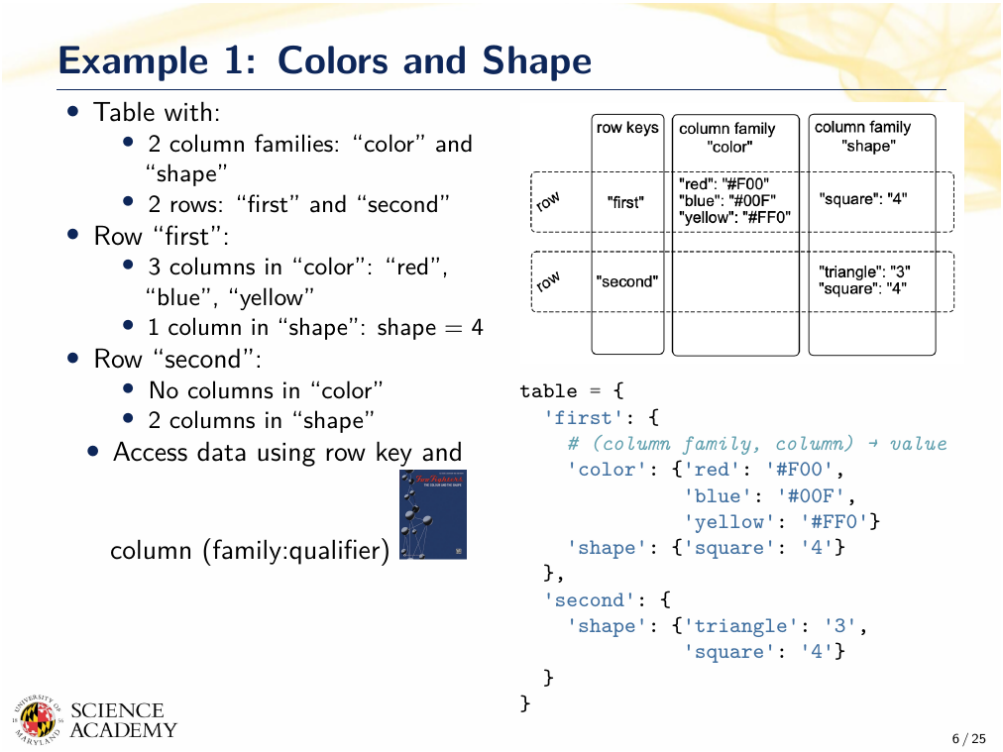


- **Warning**: It's important to note that while HBase uses terms like "database" and "table," these terms don't mean the same thing as they do in traditional relational databases. This can be confusing, so it's crucial to understand the differences.

- **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**
  - 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,

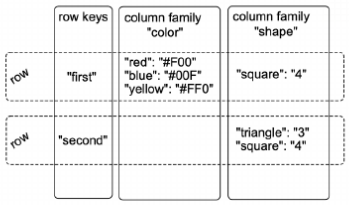
which is common in databases like HBase.

- **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



SCIENCE ACADEMY

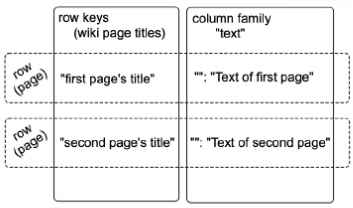
7 / 25

- **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




```
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']
```

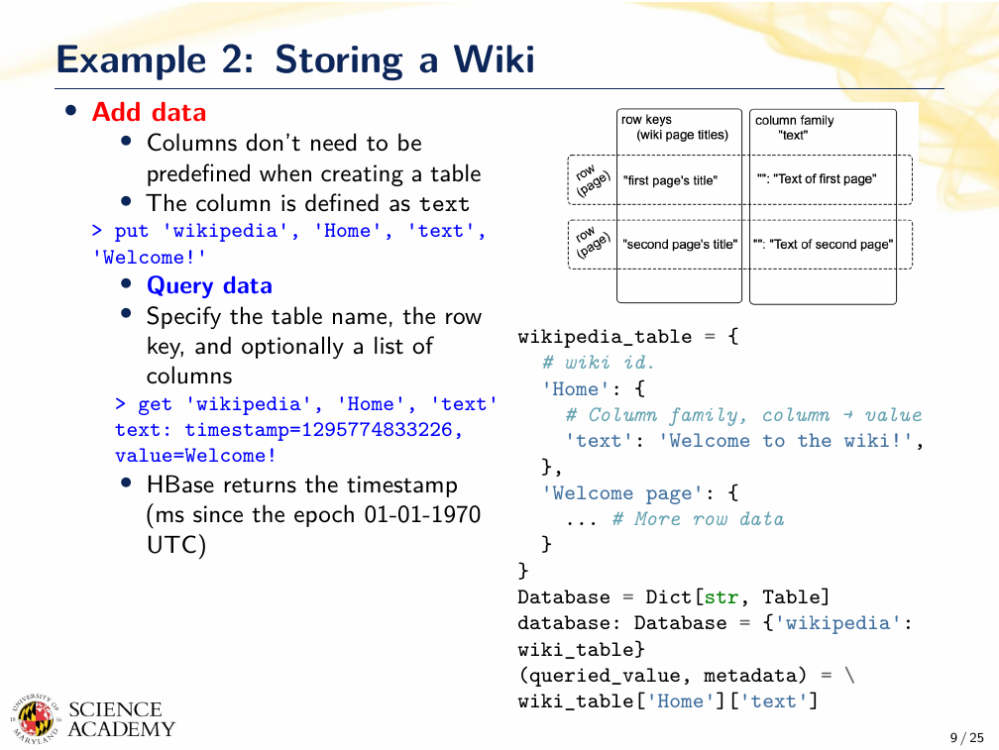
SCIENCE ACADEMY

8 / 25

- **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



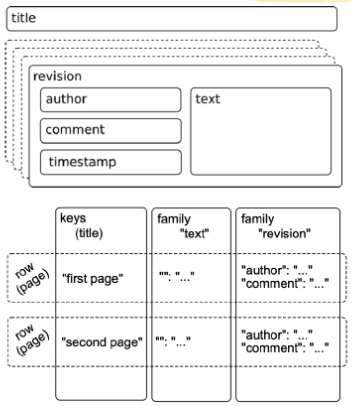
- **Add data**
  - In this example, we're using a database system where you don't need to define all the

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

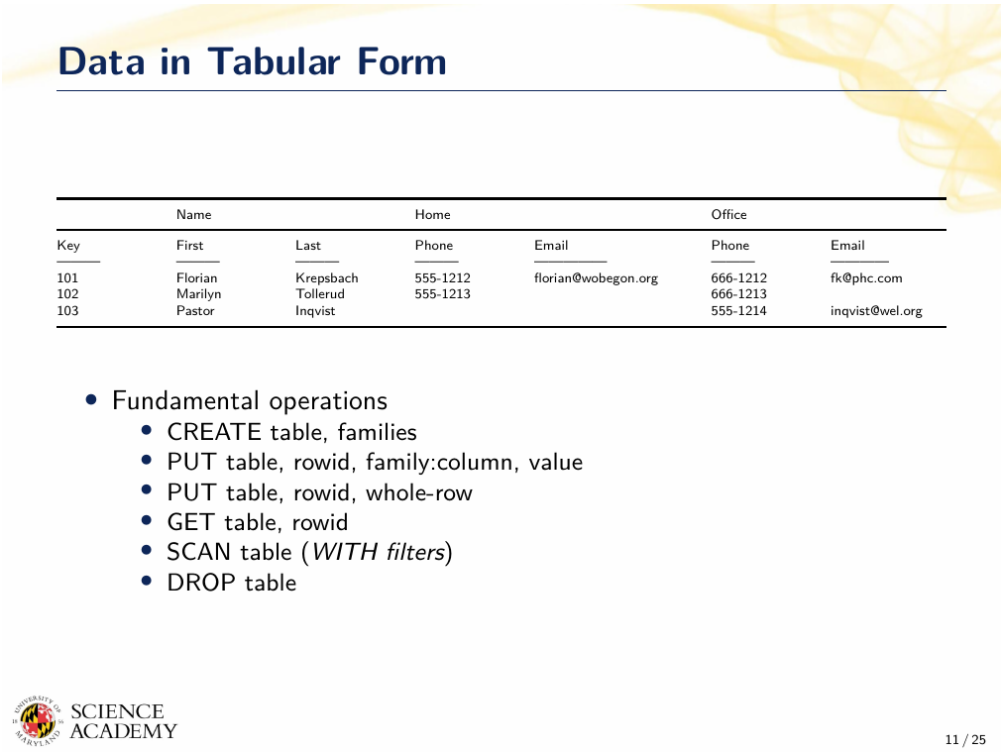


SCIENCE ACADEMY

10 / 25

- **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.

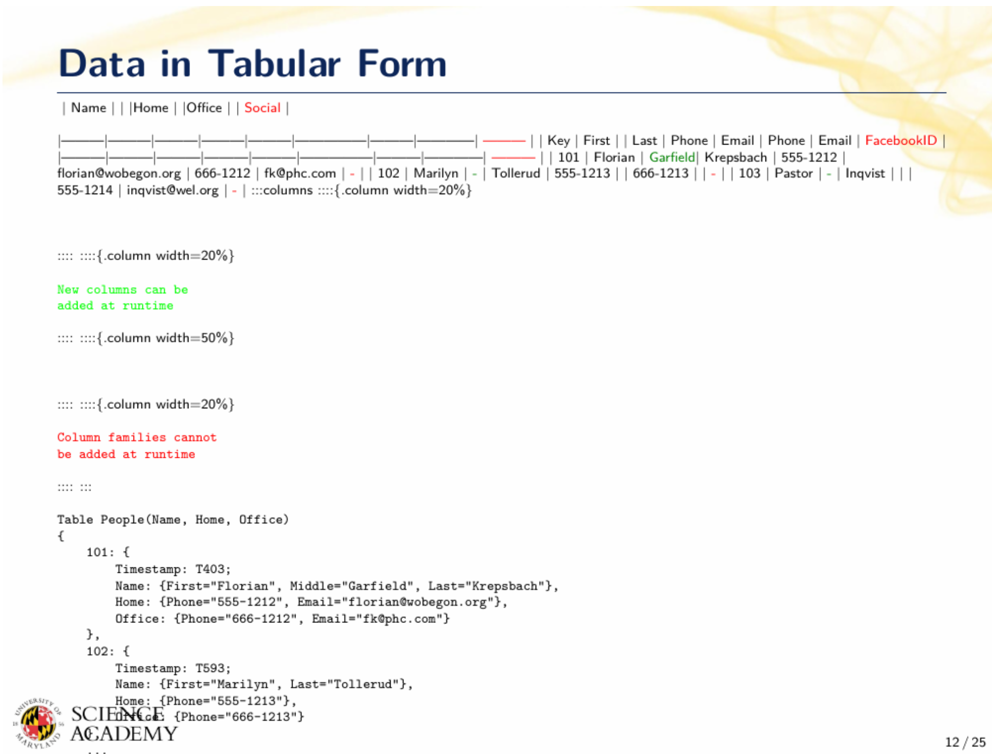
11 / 25: Data in Tabular Form



- **Data in Tabular Form**
  - 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**: 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

	Name		Home		Office	
Key	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"
```

SCIENCE ACADEMY

13 / 25

- **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.

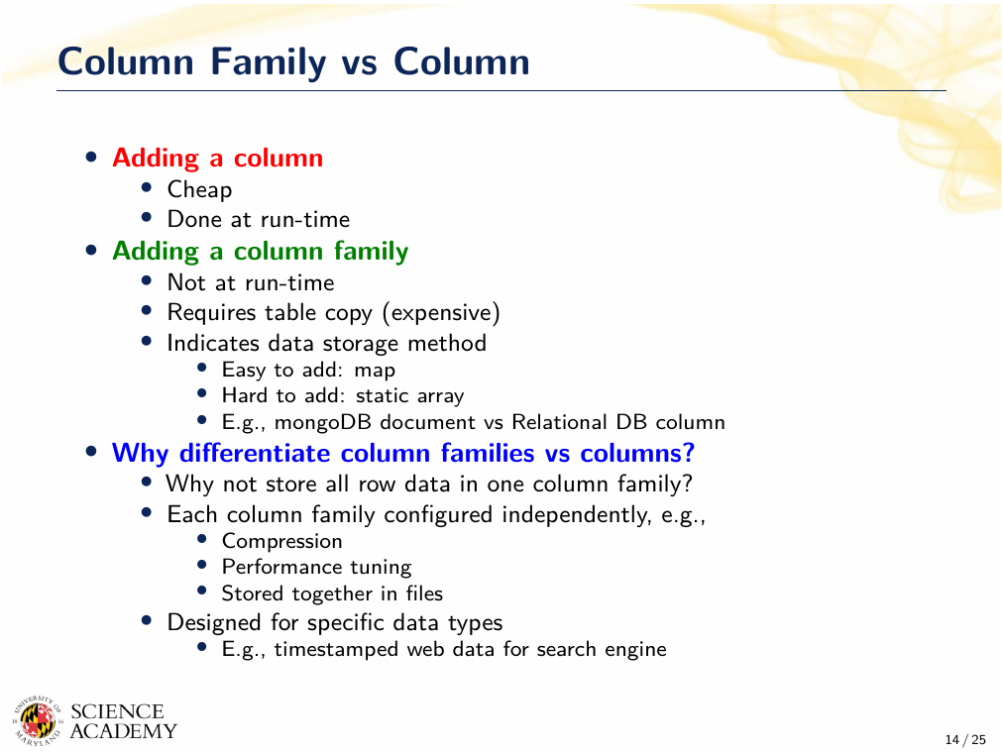
- **Table Explanation**: The table shows a list of people with their contact information. Each person has a unique **Key** (101, 102, 103) and associated details such as **Name** (split into **First** and **Last**), and contact information for **Home** and **Office** (including **Phone** and **Email**).

- **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.

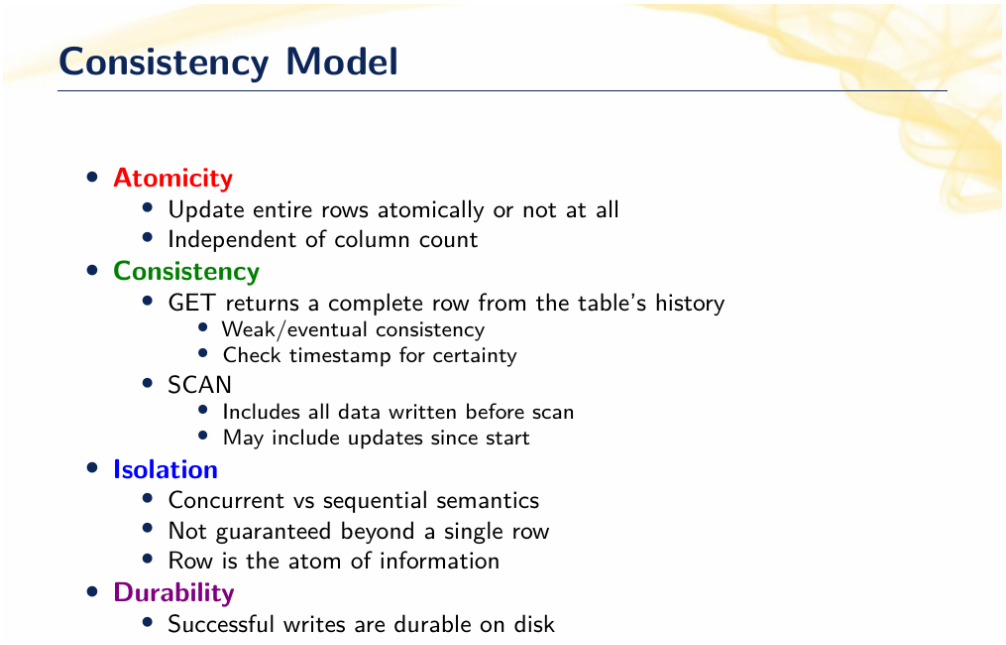
14 / 25: Column Family vs Column



- **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.

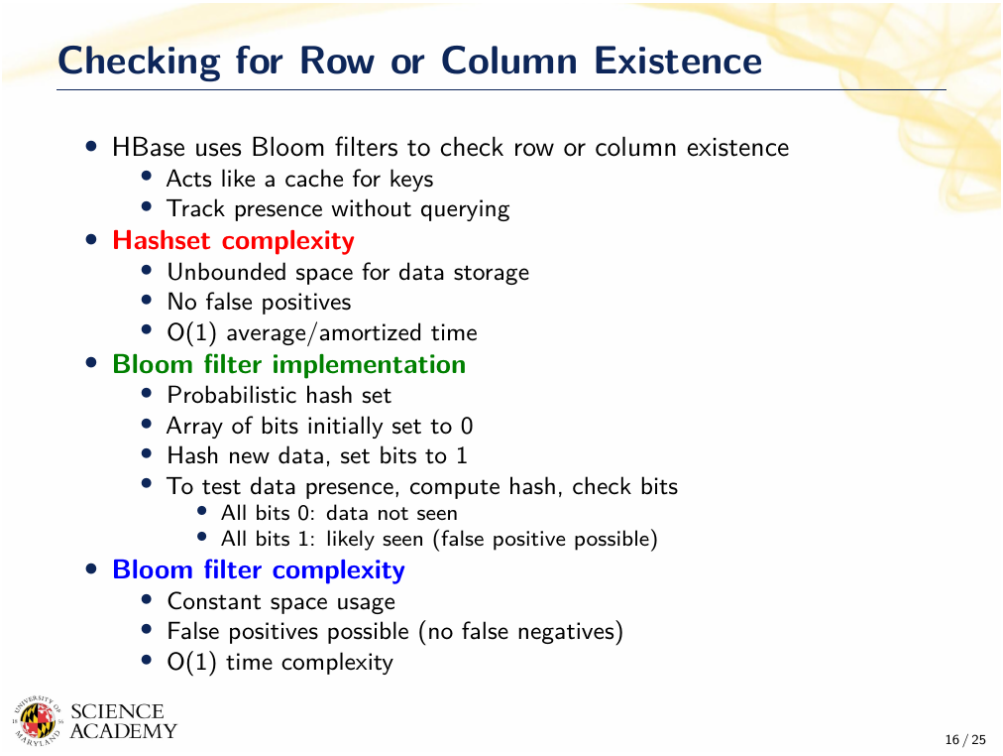
  - *Indicates data storage method*: The ease of adding a column family depends on how data is stored.
    * *Easy to add: map*: In databases that use a map-like structure, adding a column family is simpler.
    * *Hard to add: static array*: In databases using static arrays, adding a column family is 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**
    - 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.

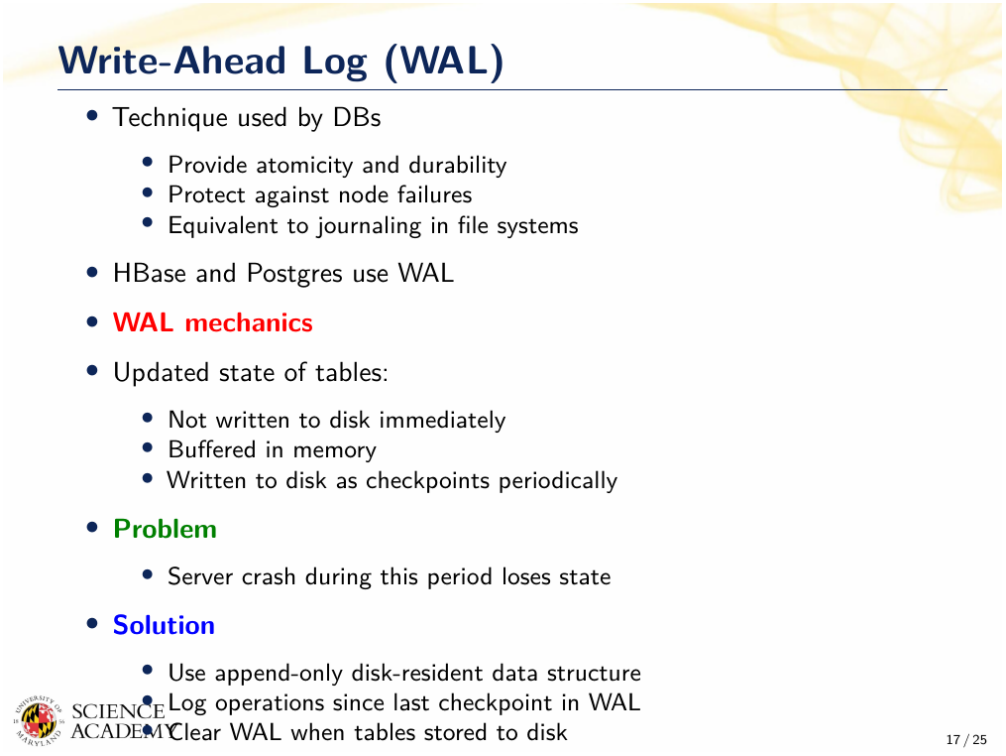
16 / 25: Checking for Row or Column Existence



- **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*: 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"}
  },
  ...
}
```

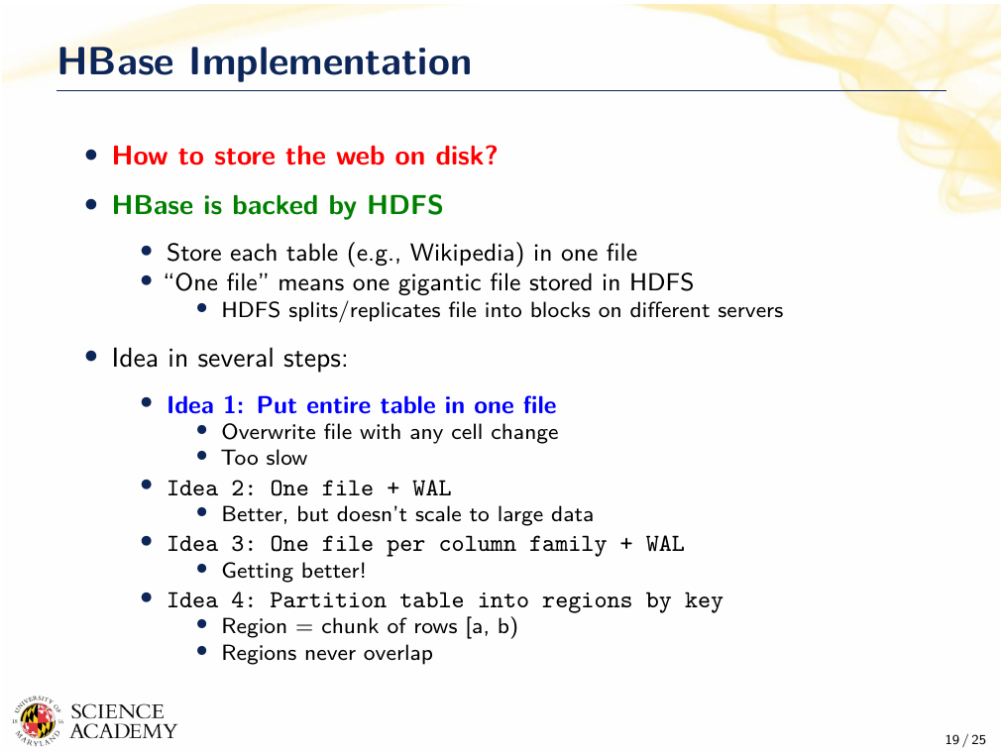
Need to use pointers

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.

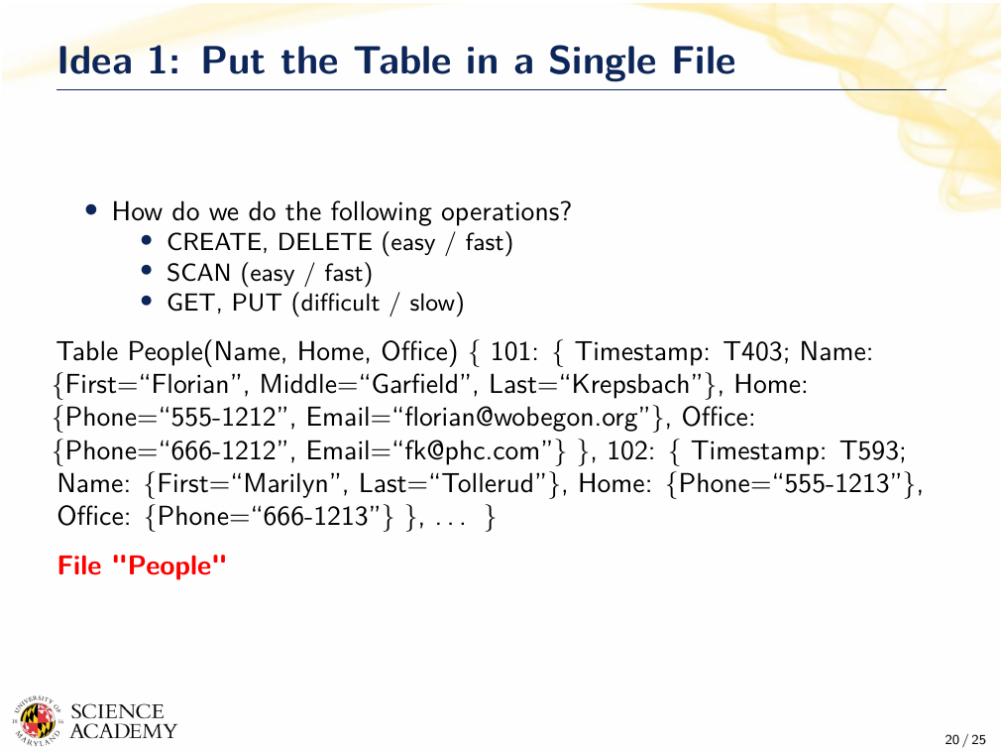
19 / 25: HBase Implementation



- **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.

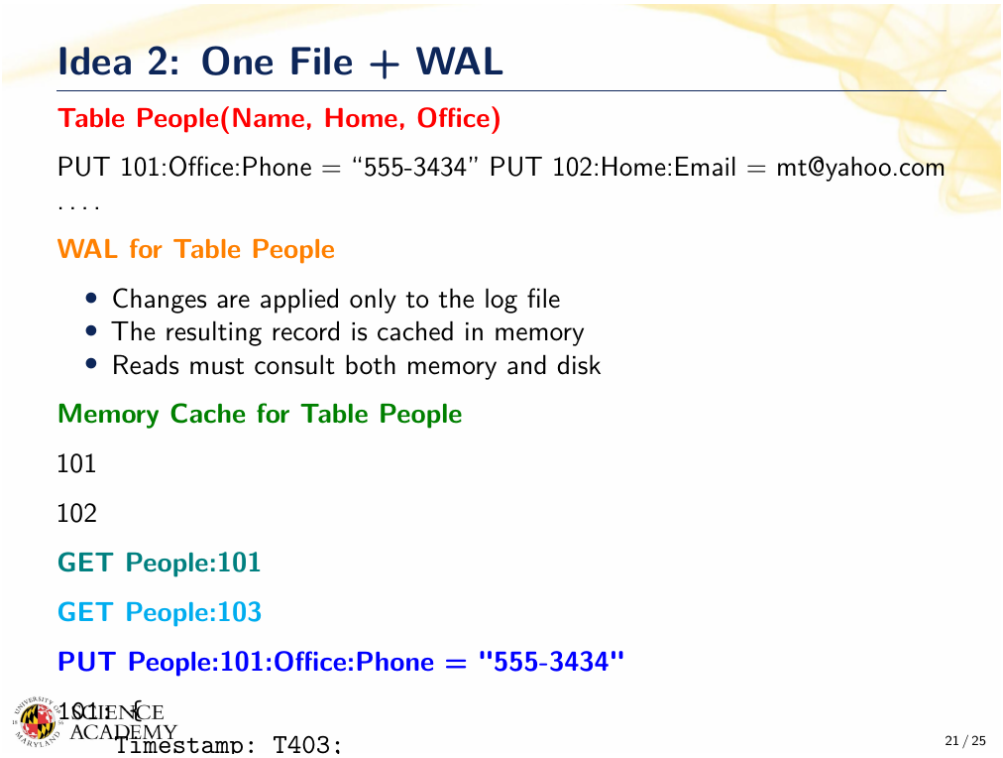
20 / 25: Idea 1: Put the Table in a Single File



- **Idea 1: Put the Table in a Single File**
  - 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

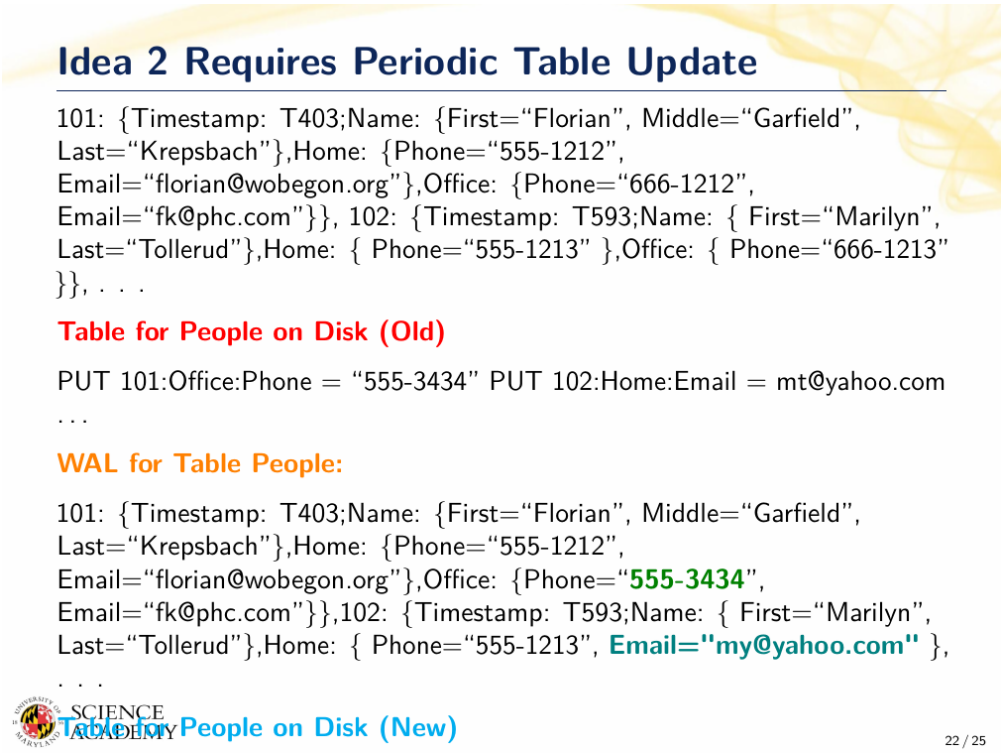


- **Idea 2: One File + WAL**
  - This slide introduces the concept of using a single file along with a Write-Ahead Log (WAL) for managing data changes. The WAL is a technique used to ensure data integrity by recording changes before they are applied to the main database.
- **Table People(Name, Home, 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

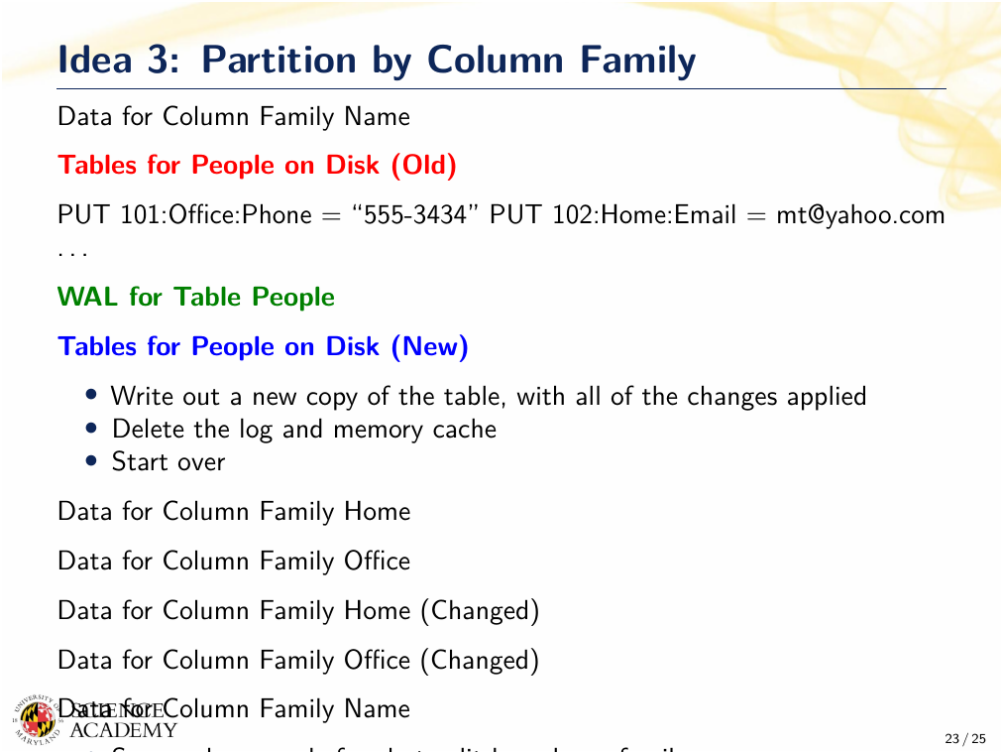


- **Idea 2 Requires Periodic Table Update**: This slide discusses the need to periodically update a data table, which is a common requirement in database management systems. The example provided involves a table of people with their contact information.
- **Table for People on Disk (Old)**: Initially, 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

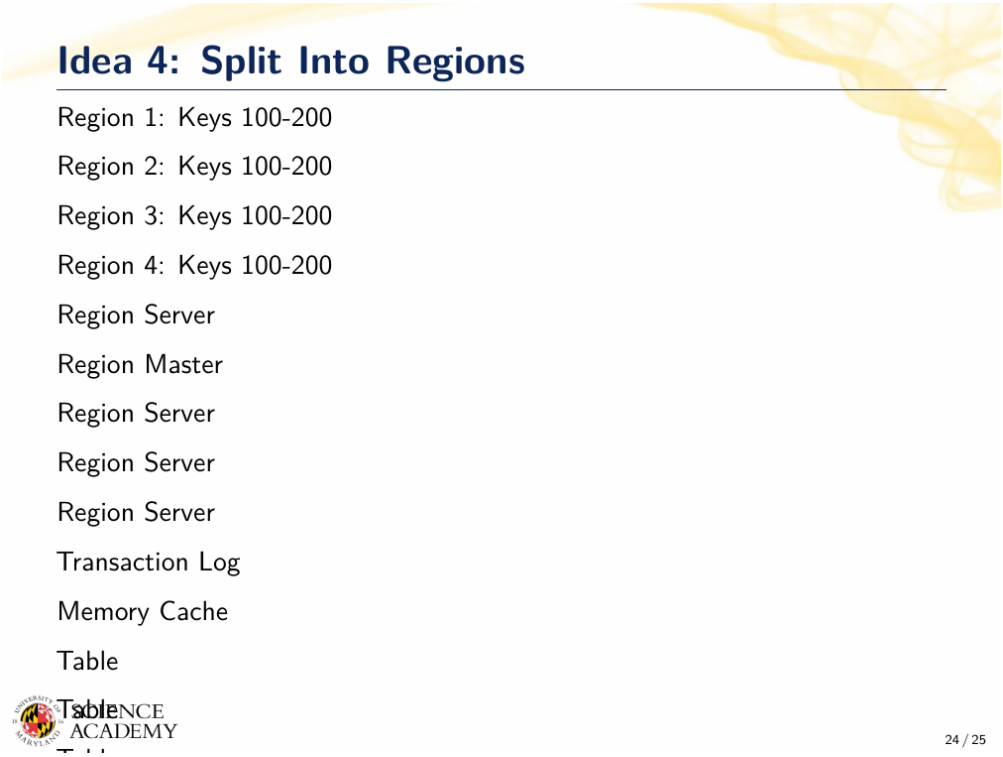


- **Idea 3: Partition by Column Family**
  This slide introduces the concept of partitioning data by column family, which is a method used in databases to organize and manage data more efficiently. A column family is a group of related data columns that are often accessed together. By partitioning data 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

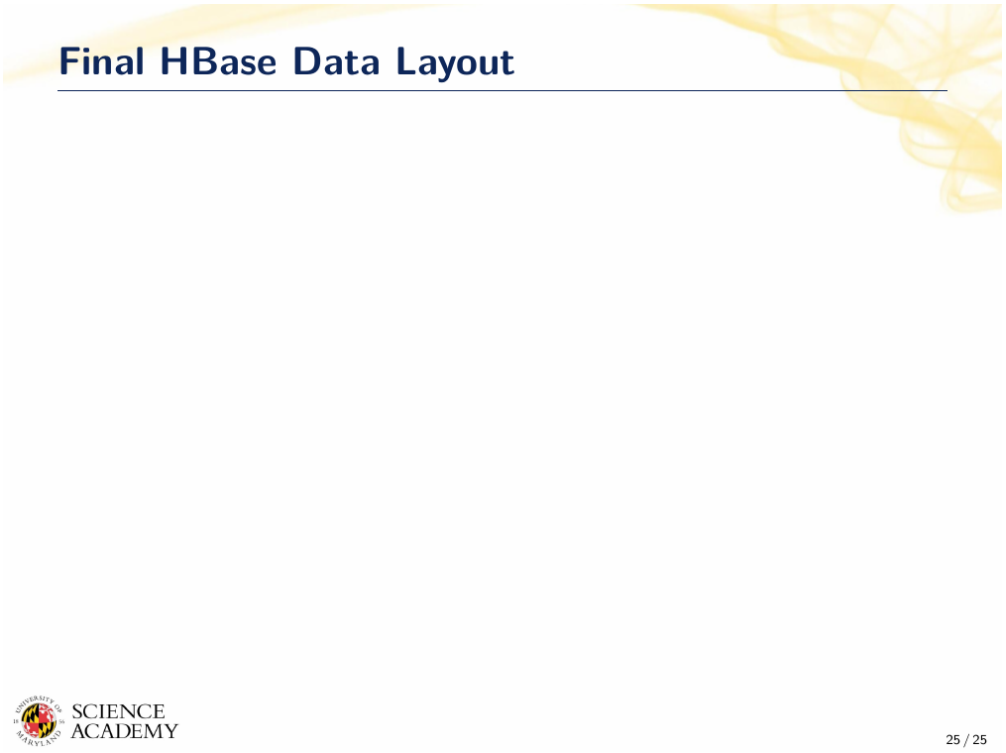


- **Idea 4: Split Into Regions**
  - The concept of splitting data into regions is crucial for managing large datasets efficiently. By dividing data into smaller, manageable parts, systems can handle requests more effectively and improve performance.
- **Region 1: Keys 100-200**
  - Each region is defined 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.

25 / 25: Final HBase Data Layout



- **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.