

DATA 514 Summary

LEC01: Introduction to Data Management

1. Overview of Databases:

- Definition: A database is a collection of files storing related data.
- Importance: Databases are fundamental to many applications, from payroll systems to student records.
- Examples: Accounts database, payroll database, student database, product database (Amazon), reservation systems.

2. Database Management Systems (DBMS):

- Definition: A large program that efficiently manages a database and allows it to persist over time.
- Examples:
 - Commercial: Oracle, IBM DB2, Microsoft SQL Server, Vertica, Teradata.
 - Open-source: MySQL, PostgreSQL, CouchDB, SQLite.
- Focus: The course primarily focuses on relational DBMSs.

3. Course Objectives:

- Understanding query processing end-to-end.
- Integrating databases into applications.
- Designing and managing data for long-term use.
- Identifying technical decisions' impacts on users.

4. Data Models:

- Types: Relational, semi-structured, key-value, graph, and object-oriented models.
- Uses: Different models are suited for different types of data and applications.

5. Course Format:

- Grading:
 - Homework: 50%
 - Lectures and sections: 40%
 - Worksheets: 25%
 - Takeaways: 15%
 - Final project: 10%
- Policies: Collaboration is encouraged, but individual submissions are required. Late days are limited and should be used wisely.

LEC02: SQL Basics

1. Introduction to SQL:

- SQL (Structured Query Language) is used for querying relational data.

- It is a declarative programming language, specifying what data to retrieve rather than how to retrieve it.

2. Query Structure:

- Basic clauses: **SELECT** (attributes to retrieve), **FROM** (tables to query), **WHERE** (conditions for filtering rows).
- Example:

```
SELECT URL, Title
FROM URLs
WHERE NumVisits > 5;
```

3. Keys in SQL:

- Primary Key: Uniquely identifies a row in a table.
- Foreign Key: Establishes a relationship between rows in different tables.
- Example:

```
CREATE TABLE Payroll (
  UserID INT PRIMARY KEY,
  Name VARCHAR(100),
  Job VARCHAR(100),
  Salary INT
);
```

4. For-each Semantics:

- Concept: SQL processes rows in a table using for-each semantics.
- Example:

```
SELECT Name, UserID
FROM Payroll
WHERE Job = 'TA';
```

LEC03: Joins

1. Recap of SQL Basics:

- Reviews SELECT-FROM-WHERE queries, table creation, and key definitions.

2. Introduction to Joins:

- Joins combine data from multiple tables based on related columns.
- Types: Inner joins, left joins, right joins, full outer joins.
- Example:

```
SELECT P.Name, R.Car
FROM Payroll AS P
INNER JOIN Regist AS R
ON P.UserID = R.UserID;
```

3. Creating Tables and Keys:

- Syntax for creating tables and defining primary and foreign keys.
- Example:

```
CREATE TABLE Regist (
  UserID INT,
  Car VARCHAR(100),
  FOREIGN KEY (UserID) REFERENCES Payroll(UserID)
);
```

4. Join Syntax:

- Explicit Join:

```
SELECT P.Name, R.Car
FROM Payroll AS P
INNER JOIN Regist AS R
ON P.UserID = R.UserID;
```

- Implicit Join:

```
SELECT P.Name, R.Car
FROM Payroll AS P, Regist AS R
WHERE P.UserID = R.UserID;
```

LEC04: Aggregates

1. Introduction to Aggregates:

- Aggregation functions summarize data: COUNT, SUM, AVG, MAX, MIN.
- Example:

```
SELECT COUNT(*)
FROM Payroll;
```

2. Using Groups in Aggregates:

- **GROUP BY** clause groups data by specified columns before applying aggregation functions.
- Example:

```
SELECT Job, AVG(Salary)
FROM Payroll
GROUP BY Job;
```

3. HAVING Clause:

- Filters groups based on aggregate values, similar to how **WHERE** filters rows.
- Example:

```
SELECT Job
FROM Payroll
GROUP BY Job
HAVING MAX(Salary) > 60000;
```

4. Query Execution:

- Comparison of **WHERE** vs. **HAVING**:
 - **WHERE** filters individual rows before grouping.
 - **HAVING** filters groups after aggregation.
- Example:

```
SELECT Job, AVG(Salary)
FROM Payroll
WHERE Salary > 50k
GROUP BY Job;
```

LEC05: Aggregates with Joins

1. Combining Aggregates and Joins:

- Using joins and aggregation functions together to summarize related data from multiple tables.
- Example:

```
SELECT P.Job, AVG(R.Salary)
FROM Payroll AS P
INNER JOIN Regist AS R
ON P.UserID = R.UserID
GROUP BY P.Job;
```

2. Examples and Syntax:

- Detailed examples of complex queries that use both joins and aggregation functions.
- Emphasis on understanding the flow and output of such queries.

3. Advanced Use Cases:

- Handling more complex scenarios with multiple joins and nested aggregations.
- Optimization tips for efficient query formulation.

LEC06a: Data in Context

1. Data in Context:

- **Simple Queries, Simple Questions:**
 - Examines how SQL can be used to answer business questions, such as identifying people using an unfair number of parking spaces.
 - Highlights the importance of data quality and the potential biases in algorithms.
- **Warning Signs of Damage:**
 - Discusses the impact of algorithms at scale, their potential negative effects, and issues of opacity and user consent.
- **Can I Do Anything?:**
 - Mentions Tracy Chou's influence on diversity in the tech industry as an example of individual impact.

2. Query Patterns:

- **Argmax/Witnessing:**
 - Introduction to the witnessing problem (argmax) and how to find the maximum value within groups.
 - Explores SQL techniques to solve the witnessing problem, including using self-joins and the HAVING clause.
 - Provides step-by-step examples to compute the highest salary per job, demonstrating common pitfalls and solutions.

LEC06b: Query Patterns

1. Query Patterns Overview:

- Introduces various query patterns such as witnessing (argmax), pivot tables, relational division, distinct pairs, and grouped n-samples.
- Emphasizes the power of joins and aggregations in solving complex problems.

2. The Witnessing Problem:

- Detailed exploration of the witnessing problem, including the challenge of finding the witness (argmax) in SQL.
- Demonstrates different approaches and SQL queries to address the witnessing problem, using self-joins and HAVING clauses.
- Provides examples of Java pseudocode to illustrate the logical process behind finding the highest salary per job.

LEC07/08: Data Management Week 3

1. Data Management Concepts:

- Review of SQL basics, including SELECT-FROM-WHERE queries, joins, and aggregates.
- Introduction to subqueries and advanced SQL topics, focusing on their practical applications in data management.

2. Query Execution and Optimization:

- Detailed discussion on query execution plans, indexing strategies, and optimization techniques.
- Emphasis on understanding the SQL engine's behavior to improve query performance and efficiency.

3. Real-World Applications:

- Examples of real-world data management scenarios and how SQL and data management principles apply to them.
- Focus on the societal impacts of data management decisions, aligning with the impact/scope/opacity rubric from the course.

LEC10/11a: Design and ER Diagrams

1. Introduction to Database Design:

- Importance of good database design, incorporating feedback from stakeholders, and determining long-term data storage needs.

2. ER Diagrams:

- Explanation of Entity-Relationship (ER) diagrams and their components: entity sets, attributes, relationships, subclassing, weak entity sets, and union types.
- Examples of how to represent real-world data and relationships using ER diagrams.

3. Design Process:

- Overview of the database design process, from conceptual models to relational schema and normalization.
- Discussion of common pitfalls in database design and how to avoid them.

LEC11b/12: Schema Design and Normalization

1. Data Anomalies and Functional Dependencies:

- Identification of data anomalies (redundancy, update, and deletion anomalies) and how to prevent them through good design.
- Introduction to functional dependencies and their role in database design.

2. Normalization:

- Explanation of normalization and its importance in eliminating data anomalies.

- Detailed discussion on Boyce-Codd Normal Form (BCNF) and the BCNF decomposition algorithm.

3. Design Theory and Armstrong's Axioms:

- Formalism of design theory to identify and remove anomalies using functional dependencies.
- Introduction to Armstrong's axioms and their use in reasoning about functional dependencies and closures.

LEC13: Physical Design and Indexes

1. Introduction to Physical Design

Database Design Process:

- **Conceptual Model:** High-level representation of organizational data, usually created using Entity-Relationship (ER) diagrams.
- **Relational Model:** Logical structure of the database, including tables (relations), columns (attributes), and relationships (keys).
- **Schema:** Detailed blueprint of the database, defining tables, columns, data types, constraints, and relationships.
- **Normalization:** Process of organizing data to reduce redundancy and improve data integrity by dividing tables and establishing relationships.

From Relations to Files:

- **Hard Reality of Hard Disks:**
 - **Mechanical Characteristics:** Includes rotation speed (RPM), number of platters, tracks, sectors, and read/write heads.
 - **Disk Latency:**
 - **Seek Time:** Time taken for the read/write head to move to the correct track.
 - **Rotational Latency:** Time taken for the disk to rotate and position the desired sector under the read/write head.
 - **Representing Relations:**
 - **Heap Files:** Unordered files where records are stored as they arrive.
 - **Sequential Files:** Ordered files where records are stored in a sorted order based on a key.

2. Using Indexes

Types of Indexes:

- **B+ Tree Index:** Balanced tree structure that maintains sorted data and allows efficient insertion, deletion, and range queries.
- **Hash Index:** Uses a hash function to map keys to positions, enabling fast equality searches.
- **R Tree:** Used for spatial data, supporting queries on multi-dimensional data.
- **Radix Tree:** Compressed version of a trie used for string data.
- **Bloom Filter:** Probabilistic data structure used to test whether an element is a member of a set.

Implementing Indexes:

- **In-memory:**
 - **B+-Trees:** Efficient for in-memory databases due to their balance and order-maintenance properties.
- **On-disk:**
 - **Clustered Indexes:** Data is stored in the same order as the index key, improving range queries and I/O efficiency.
 - **Unclustered Indexes:** Data is stored in a different order than the index key, allowing multiple indexes per table but potentially requiring more I/O for queries.

Multi-Attribute Indexes:

- Indexes created on multiple columns to support complex query predicates.
- Considerations for query optimization include the order of columns in the index and the types of queries expected.

3. Clustered vs. Unclustered Indexes

Clustered Index:

- **Structure:** Data rows are stored on disk in the same order as the index key.
- **Advantages:** Faster range queries, reduced I/O for sequential access.
- **Disadvantages:** Slower insert/update/delete operations due to the need to maintain order.

Unclustered Index:

- **Structure:** Index contains pointers to data rows stored in a different order.
- **Advantages:** Allows multiple indexes per table, faster index creation and maintenance.
- **Disadvantages:** Slower range queries due to random I/O access.

Range Scans:

- **Clustered Index:** Efficient, as data is contiguous on disk, minimizing disk I/O.
- **Unclustered Index:** Less efficient, as data may be scattered, requiring multiple disk I/O operations.

4. Helping the RDBMS

Index Selection:

- Choosing the right indexes based on:
 - **Application Workloads:** Understanding the types of queries (e.g., read-heavy vs. write-heavy).
 - **Query Patterns:** Analyzing common query patterns to determine the most beneficial indexes.
 - **Data Distribution:** Considering the distribution and uniqueness of data values.

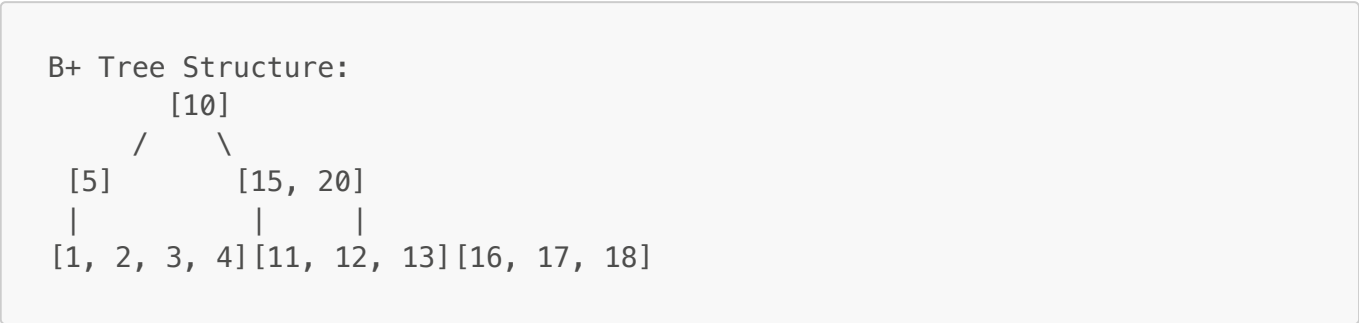
Examples:

- **Scenario 1:** A read-heavy application with frequent range queries benefits from a clustered index on the relevant columns.
- **Scenario 2:** An application with frequent equality searches on multiple attributes might benefit from a multi-attribute hash index.

- **Scenario 3:** A write-heavy application may avoid clustered indexes to improve insert/update/delete performance, opting for unclustered indexes instead.

Simple Diagrams

B+ Tree Index



Clustered vs. Unclustered Indexes

Clustered Index:

Table: Employees
Index on EmployeeID

Clustered Index

EmployeeID	Name	Department
1	John	HR
2	Jane	IT
3	Bob	Sales

Data stored in the same order as EmployeeID

Unclustered Index:

Table: Employees
Index on Department

Unclustered Index

Department	Pointer
HR	-> Record 1
IT	-> Record 2
Sales	-> Record 3

Data stored in a different order

These diagrams and detailed explanations should help clarify the key concepts of physical design and indexing in databases. Let me know if you need further details or additional sections covered!

LEC14: Data Governance

1. Data Governance Overview:

- **Definition:** Ensuring high-quality data that supports application needs throughout its lifecycle.
- **Key Aspects:** Availability, usability, consistency, data integrity, and security.

2. Personally Identifiable Information (PII):

- **Types of PII:** Direct identifiers (names, social security numbers) and indirect identifiers (birthday, gender).
- **Domain-Specific Protected Data:** Examples of protected data under laws like FERPA, HIPAA, and GLBA.

3. Managing Valuable Data:

- **Data Anomalies:** Challenges with redundant data, inconsistent updates, and undefined deletions.
- **Preventing Data Anomalies:** Separating unrelated attributes and implementing good data management practices.

4. Anonymization Techniques:

- **Methods:** Masking/obfuscation, pseudonymization, perturbation/generalization.
- **Implicit Disclosure:** Risks of revealing sensitive data through indirect means, emphasizing the importance of comprehensive anonymization strategies.

5. Access Control Mechanisms:

- **Principle of Least Privilege:** Granting minimal access necessary for job completion.
- **Role-Based Access Control (RBAC):** Using roles and permissions to manage data access effectively.

6. Data Retention Policies:

- **Retention Strategies:** Deleting or aggregating data based on time or events to minimize risk and maintain data relevance.

LEC15a: ETL (Extract/Transform/Load)

1. Introduction to ETL

Definition:

- **ETL** stands for Extract, Transform, Load, and it describes the process of moving and transforming data from one system to another.

Steps:

- **Extract:** Reading relevant data from multiple sources, which could include databases, flat files, APIs, etc.
 - **Sources:** Databases, CSV files, APIs, logs, etc.
 - **Methods:** SQL queries, web scraping, file reading.
 - **Tools:** Apache Nifi, Talend, Informatica.
- **Transform:** Applying mapping functions like aggregations, normalization, and other data transformations to clean and structure the data.
 - **Operations:** Filtering, sorting, joining, aggregating, enriching.
 - **Tools:** Apache Spark, Python (Pandas), SQL scripts.
- **Load:** Writing the transformed data to the destination system, which could be a data warehouse, data lake, or another database.
 - **Destinations:** Data warehouses (e.g., Amazon Redshift, Google BigQuery), databases (e.g., MySQL, PostgreSQL).
 - **Tools:** Airflow, AWS Glue, DataStage.

2. Data Wrangling

Definition:

- Data wrangling is similar to ETL but involves more exploration and interactivity to understand, clean, and prepare data for analysis.

Importance of Visualization:

- Visualization tools (e.g., Tableau, Power BI) are used to iterate on data transformations and improve understanding by providing immediate visual feedback.

Tools:

- Trifacta, DataWrangler, Excel.

3. Examples

Pivoting Data:

- Transforming "skinny and tall" datasets into "short and wide" formats for easier analysis and reporting.
- **Example:**
 - Input:

Date	Product	Sales
2023-01-01	A	100
2023-01-01	B	150

- Output:

Date	A	B
------	---	---

Date	A	B
2023-01-01	100	150

Unpivoting Data:

- Storing data in granular, unpivoted form to reduce NULLs and enable detailed cleaning.
- **Example:**
 - Input:

Date	A	B
2023-01-01	100	150

- Output:

Date	Product	Sales
2023-01-01	A	100
2023-01-01	B	150

4. Course Roadmap

ETL Workloads:

- Importance of ETL in data science, particularly for large datasets requiring parallel processing.
- **Challenges:** Scalability, data consistency, error handling, monitoring.

SQL Queries as Pipelines:

- Using SQL queries to form pipelines, emphasizing phase ordering and using good estimates to optimize query performance.
- **Example:**
 - Phase 1: Extraction using SQL SELECT statements.
 - Phase 2: Transformation using SQL JOINS, GROUP BY, and other transformations.
 - Phase 3: Loading using INSERT INTO or COPY commands.

LEC15: B-Trees and Indexes (Supplement)

1. Index Structures

Types:

- **B+ Tree Index:** Balanced tree structure for maintaining sorted data and efficient range queries.
- **Hash Index:** Uses a hash function to map keys to positions, enabling fast equality searches.
- **R Tree:** Used for spatial data, supporting multi-dimensional queries.
- **Radix Tree:** Compressed trie structure used for string data.
- **Bloom Filter:** Probabilistic data structure for testing membership in a set.

Characteristics:

- **B-Trees:** Balanced tree data structure maintaining sorted order and supporting logarithmic time complexity for insertion, deletion, and search.

- **B+-Trees:** Enhanced version of B-Trees with all values at the leaf level and linked leaves for efficient range queries.

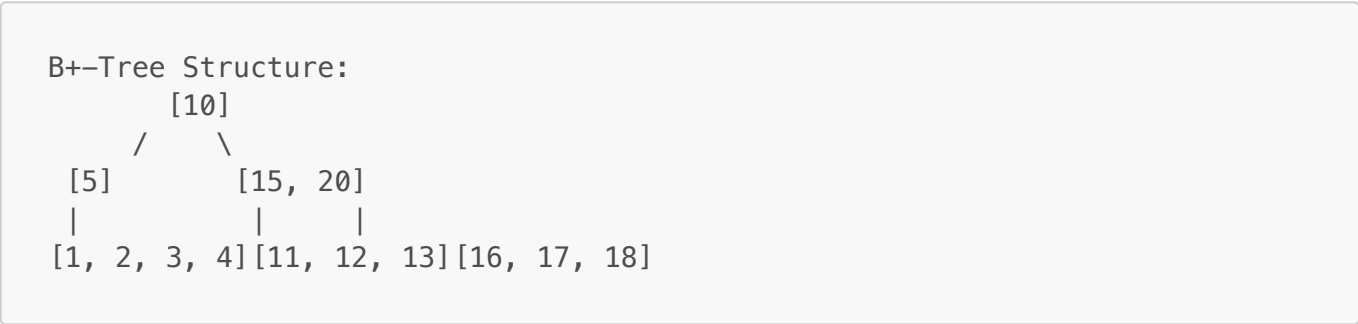
2. B+-Tree Implementation

Insertion and Deletion:

- **Insertion:** Insert key, split nodes if necessary, adjust pointers.
- **Deletion:** Remove key, merge nodes if necessary, adjust pointers.

Range Queries:

- Efficient range scans using linked leaf nodes to traverse sorted data.
- **Diagram:**



3. Physical Design

Clustered vs. Unclustered Indexes:

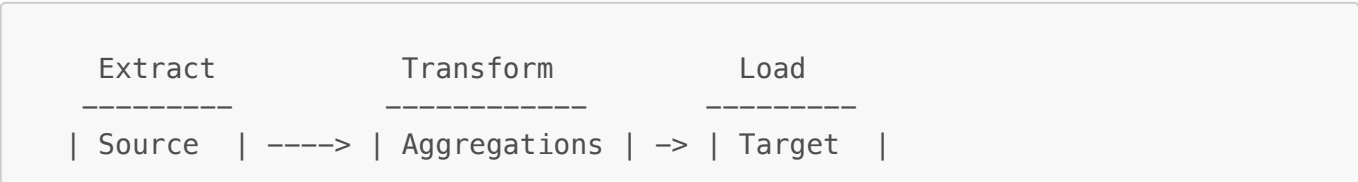
- **Clustered Index:**
 - **Structure:** Data rows are stored on disk in the same order as the index key.
 - **Advantages:** Faster range queries, reduced I/O for sequential access.
 - **Disadvantages:** Slower insert/update/delete operations due to order maintenance.
- **Unclustered Index:**
 - **Structure:** Index contains pointers to data rows stored in a different order.
 - **Advantages:** Allows multiple indexes per table, faster index creation and maintenance.
 - **Disadvantages:** Slower range queries due to random I/O access.

Multiple Indexes:

- Creating and using multiple indexes on a table for optimizing different types of queries.
- **Considerations:** Balancing read and write performance, index maintenance overhead, query optimization strategies.

Simple Diagrams

ETL Process



System	Normalization	System
-----	-----	-----

Clustered vs. Unclustered Indexes

Clustered Index:

Table: Employees
Index on EmployeeID

Clustered Index

EmployeeID	Name	Department
1	John	HR
2	Jane	IT
3	Bob	Sales

Data stored in the same order as EmployeeID

Unclustered Index:

Table: Employees
Index on Department

Unclustered Index

Department	Pointer
HR	-> Record 1
IT	-> Record 2
Sales	-> Record 3

Data stored in a different order

These detailed explanations and diagrams should help clarify the key concepts of ETL processes, B-Trees, and index structures. Let me know if you need further details or additional sections covered!

LEC16/17: Relational Algebra

1. Introduction to Relational Algebra (RA)

Purpose:

- Relational Algebra (RA) provides a procedural foundation for query operations in databases, transforming declarative SQL queries into executable plans.

RA Operators:

- **Basic Operators:**
 - **Selection (σ)**: Selects rows that satisfy a given predicate.
 - **Projection (π)**: Selects specific columns from a table.
 - **Union (\cup)**: Combines rows from two tables, removing duplicates.
 - **Intersection (\cap)**: Returns rows common to both tables.
 - **Difference ($-$)**: Returns rows present in the first table but not in the second.
- **Extended Operators:**
 - **Grouping (γ)**: Groups rows based on specified columns and computes aggregate functions.
 - **Sorting (τ)**: Orders rows based on specified columns.
 - **Duplicate Elimination (δ)**: Removes duplicate rows.
 - **Joins (\bowtie)**: Combines rows from two tables based on a related column.

Diagram:

```
Relational Algebra Operators:
σ (Selection) - Selects rows with a predicate
π (Projection) - Selects specific columns
∪ (Union) - Combines rows, removes duplicates
∩ (Intersection) - Common rows in both tables
- (Difference) - Rows in first table not in second
γ (Grouping) - Groups rows, computes aggregates
τ (Sorting) - Orders rows
δ (Duplicate Elimination) - Removes duplicates
⋈ (Join) - Combines rows from two tables
```

2. SQL to RA Translation

Flat Queries:

- Translating simple SQL queries into RA expressions.
- **Example:**
 - **SQL:** `SELECT Name FROM Employees WHERE Department = 'IT';`
 - **RA:** `$\pi_{Name}(\sigma_{Department='IT'}(Employees))$`

Subqueries:

- Handling subqueries in **FROM** and **WHERE/HAVING** clauses in RA.
- **Example:**
 - **SQL:** `SELECT Name FROM Employees WHERE Salary > (SELECT AVG(Salary) FROM Employees);`
 - **RA:** `$\pi_{Name}(\sigma_{Salary > (\gamma_{avg}(Salary)(Employees))}(Employees))$`

3. RA Trees

Tree Transformations:

- Optimizing query execution by transforming RA trees.
- **Example:**
 - **Initial RA Tree:**

```
π_Name
|
σ_Department='IT'
|
Employees
```

- **Optimized RA Tree:** Combine selections and projections to minimize intermediate results.

Execution Plans:

- Converting RA trees into physical plans for efficient query processing.
- **Diagram:**

```
Execution Plan:
π_Name
|
σ_Department='IT'
|
Scan(Employees)
```

4. Sets vs. Bags

Set Semantics:

- Standard relational algebra operates with distinct tuples.
- **Example:** Union of two sets results in a set with unique elements.

Bag Semantics:

- Extended relational algebra allows duplicate tuples, commonly used in DBMS.
- **Example:** Union of two bags results in a bag that may contain duplicates.
- **Diagram:**

```
Set Semantics:
{1, 2} ∪ {2, 3} = {1, 2, 3}

Bag Semantics:
[1, 2] ∪ [2, 3] = [1, 2, 2, 3]
```


LEC18: Cardinality Estimation

1. Introduction to Cardinality Estimation

Definition:

- Cardinality estimation refers to predicting the number of rows returned by a query operation.

Importance:

- Accurate cardinality estimation is crucial for query optimization and efficient execution plans.
- **Example:** Underestimating or overestimating the number of rows can lead to inefficient query plans and poor performance.

2. Histograms and Sampling

Histograms:

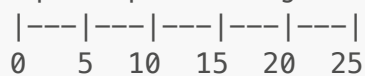
- Used to approximate the distribution of data values in a column, aiding in cardinality estimation.
- **Types:**
 - **Equi-width Histograms:** Divides the range of data into equal-sized intervals.
 - **Equi-depth Histograms:** Divides the data so each interval contains approximately the same number of tuples.
 - **Hybrid Histograms:** Combines features of equi-width and equi-depth histograms for better accuracy.

Diagram:

Equi-width Histogram:



Equi-depth Histogram:



Sampling:

- Collecting random samples of data to estimate cardinality. It's less accurate but more efficient than full scans.
- **Example:** Using a 10% sample to estimate the distribution of values in a column.

3. Challenges and Techniques

Skew and Correlation:

- Addressing issues caused by non-uniform data distribution and correlations between columns.
- **Example:** Data skew can lead to inaccurate cardinality estimates and suboptimal query plans.

Dynamic Programming:

- Techniques to improve accuracy, like multi-dimensional histograms and dynamic programming-based algorithms.
- **Example:** Using multi-dimensional histograms to capture correlations between multiple columns for more accurate cardinality estimates.

Simple Diagrams

SQL to RA Translation Example

SQL:

```
SELECT Name FROM Employees WHERE Department = 'IT';
```

RA:

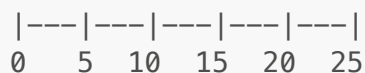
```
 $\pi_{\text{Name}}(\sigma_{\text{Department}='IT'}(\text{Employees}))$ 
```

Cardinality Estimation with Histograms

Equi-width Histogram:



Equi-depth Histogram:



These detailed explanations and diagrams should help clarify the key concepts of relational algebra and cardinality estimation in databases. Let me know if you need further details or additional sections covered!

LEC19/20a: Parallel Processing

1. Parallelizing Data Management

Introduction:

- **Motivation:** The need to handle large datasets and improve performance by distributing the workload across multiple processors or machines.
- **Benefits:** Improved query performance, scalability, and faster data processing.

Partitioning:

- **Definition:** Dividing data across multiple nodes to enable parallel processing.
- **Techniques:**
 - **Horizontal Partitioning (Sharding):** Distributing rows across nodes.
 - **Vertical Partitioning:** Distributing columns across nodes.
 - **Hash Partitioning:** Using a hash function to distribute data evenly.
 - **Range Partitioning:** Distributing data based on a range of values.

Skew:

- **Definition:** Uneven data distribution across nodes, leading to performance bottlenecks.
- **Challenges:** Some nodes may become overloaded while others remain underutilized.
- **Solutions:** Techniques to handle skew include dynamic partitioning, data redistribution, and load balancing.

2. Shared-Nothing Model

Definition:

- A parallel processing architecture where each node is independent and self-sufficient, with its own memory and disk storage.

Partitioned Operations:

- **Selection:**
 - **Partitioned Selection:** Each node independently processes a subset of data, and the results are combined.
 - **Example:**

```
Node 1: SELECT * FROM Employees WHERE Department = 'HR'  
Node 2: SELECT * FROM Employees WHERE Department = 'IT'
```

- **Aggregation:**
 - **Local Aggregation:** Each node performs aggregation on its subset of data.
 - **Reshuffling:** Data is redistributed to ensure all relevant data for aggregation is on the same node.
 - **Example:**

```
Node 1: SUM(Salary) WHERE Department = 'HR'  
Node 2: SUM(Salary) WHERE Department = 'IT'  
Combined Result: SUM(Node 1 Result + Node 2 Result)
```

- **Joins:**
 - **Hash Join:** Data is partitioned based on a hash of the join key, then joined locally on each node.

- **Broadcast Join:** One table is small enough to be broadcasted to all nodes, and the join is performed locally on each node.
- **Example:**

```
Partitioned Hash Join:
Node 1: Employees ⋈ Hash(DepartmentID)
Node 2: Departments ⋈ Hash(DepartmentID)
```

Example:

- **Partitioned Hash Join:**
 - **Step 1:** Reshuffle tuples based on join keys.
 - **Step 2:** Perform local join on each node.
 - **Step 3:** Collect and combine results from all nodes.
 - **Diagram:**

Node 1	Node 2
-----	-----
Employees	Departments
hash	hash
join	join
-----	-----

LEC20b/21: MapReduce

1. Introduction to MapReduce

Definition:

- A programming model designed for processing large datasets with a distributed algorithm on a cluster.

Key Concepts:

- **Map Phase:** Processes input data and generates key-value pairs.
- **Shuffle Phase:** Distributes key-value pairs to reducers based on the key.
- **Reduce Phase:** Processes each group of key-value pairs to produce the final output.

Examples:

- **Word Count:**
 - **Map:** Reads text and emits a key-value pair for each word (`<word, 1>`).
 - **Shuffle:** Groups key-value pairs by key (word).
 - **Reduce:** Sums the values for each key to get word count.
- **Relational Join:**

- **Map:** Emits key-value pairs for join keys.
- **Shuffle:** Groups key-value pairs by join keys.
- **Reduce:** Combines values for each key to perform the join.

2. Execution Details

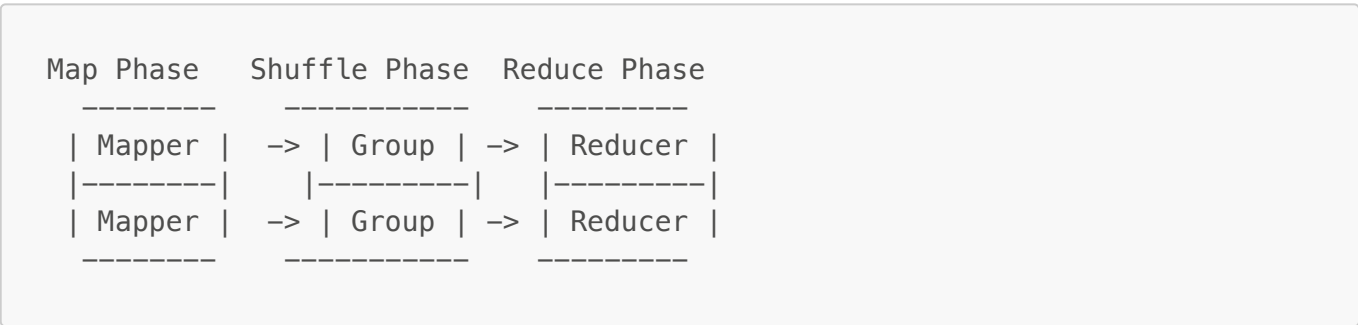
Job and Task Management:

- **Job:** A complete MapReduce computation, consisting of multiple tasks.
- **Task:** A unit of work, either a map task or a reduce task.
- **Task Execution:** Managed by a master node, which assigns tasks to worker nodes and monitors their progress.

Optimizations:

- **Combiners:** Intermediate reduction step to minimize data transfer during the shuffle phase.
- **Backup Tasks:** Speculative execution of slow tasks on other nodes to handle failures and stragglers.

Diagram:



3. Relational Queries

Implementation:

- Transforming relational queries into MapReduce jobs, including handling joins and aggregations.
- **Example:**
 - **SQL Query:** `SELECT Employee.Name, Department.Name FROM Employee JOIN Department ON Employee.DepartmentID = Department.DepartmentID;`
 - **MapReduce:**
 - **Map:** Emits key-value pairs for `DepartmentID`.
 - **Reduce:** Joins the values for each `DepartmentID`.

Challenges:

- **Data Skew:** Uneven distribution of data can lead to performance issues.
- **Efficient Execution:** Ensuring that MapReduce jobs are efficiently executed by balancing load and optimizing data transfer.

Simple Diagrams

Partitioned Hash Join Example

Node 1	Node 2
-----	-----
Employees	Departments
hash(DeptID)	hash(DeptID)
join(DeptID)	join(DeptID)
-----	-----

MapReduce Workflow

Map Phase	Shuffle Phase	Reduce Phase
-----	-----	-----
Mapper	-> Group	-> Reducer
-----	-----	-----
Mapper	-> Group	-> Reducer
-----	-----	-----

These detailed explanations and diagrams should help clarify the key concepts of parallel processing and MapReduce. Let me know if you need further details or additional sections covered!

LEC22: NoSQL and Key-Value Stores

1. ACID Properties

ACID:

- **Atomic:** All operations in a transaction are completed successfully or none are.
- **Consistent:** Transactions lead the database from one valid state to another, maintaining data integrity.
- **Isolated:** Transactions do not interfere with each other, maintaining the intermediate state invisible to other transactions.
- **Durable:** Once a transaction is committed, it remains so, even in the event of a system crash.

Importance in Traditional Databases:

- Ensures reliability and correctness in transaction processing, crucial for financial and critical applications.

2. Scaling

Challenges in Relational Databases:

- **Consistency Maintenance:** Keeping data consistent across multiple partitions and replicas can be complex.
- **Performance:** As the volume of data grows, maintaining performance while ensuring ACID properties becomes difficult.

NoSQL:

- **Relaxing ACID Constraints:** NoSQL databases often sacrifice some ACID properties to achieve better performance and scalability.
- **Benefits:** Improved scalability, higher availability, and better performance for large-scale data applications.

3. CAP Theorem

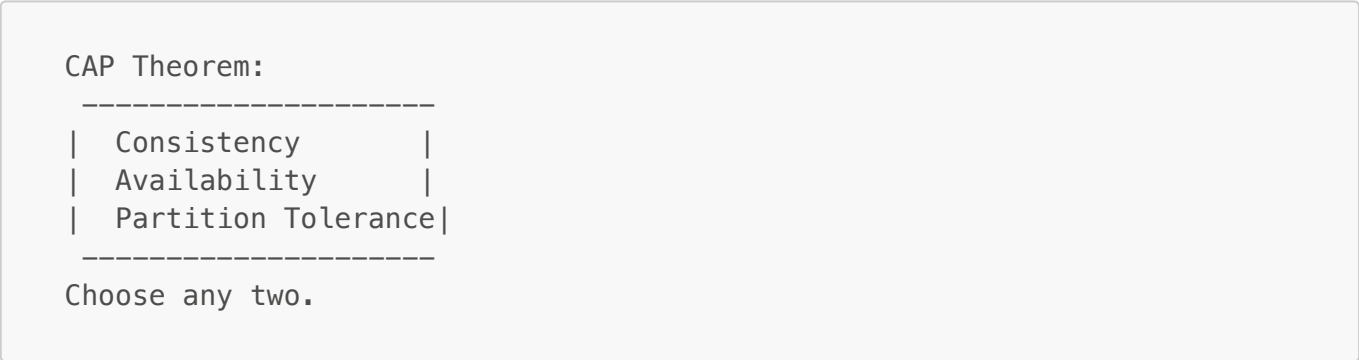
Explanation:

- **CAP Theorem:** In a distributed data store, you can only guarantee two out of three properties: Consistency, Availability, and Partition tolerance.
 - **Consistency:** Every read receives the most recent write.
 - **Availability:** Every request receives a response, without guarantee that it contains the most recent write.
 - **Partition Tolerance:** The system continues to operate despite network partitions.

Implications:

- **Distributed RDMS:** Typically ensure Consistency and Partition tolerance.
- **NoSQL Systems:** Typically ensure Availability and Partition tolerance.

Diagram:



4. Key-Value Stores

Data Model:

- **Simple Model:** Consists of key-value pairs, where the key is unique and the value can be anything.

Operations:

- **Get:** Retrieves the value associated with a given key.
- **Put:** Inserts or updates the value associated with a given key.
- **Delete:** Removes the key-value pair.

Examples:

- **Use Cases:** Caching, session management, user profiles.
- **Examples:** Redis, DynamoDB.

Diagram:



Key-Value Store:

Key	Value
123	Alice
124	Bob
125	Carol

Operations:

- Get(Key)
- Put(Key, Value)
- Delete(Key)

LEC23/24a: NoSQL Document Stores

1. NoSQL Landscape

Types of NoSQL Databases:

- Key-Value Stores:** Simple key-value pairs (e.g., Redis).
- Document Stores:** Store data as documents, typically JSON or BSON (e.g., MongoDB, CouchDB).
- Wide-Column Stores:** Store data in tables with flexible columns (e.g., Cassandra).
- Graph Databases:** Store data as nodes and edges (e.g., Neo4j).

Use Cases:

- Key-Value Stores:** Caching, real-time analytics.
- Document Stores:** Content management systems, web applications.
- Wide-Column Stores:** Time-series data, big data analytics.
- Graph Databases:** Social networks, recommendation engines.

Diagram:

NoSQL Database Types:

Key-Value Stores	Redis
Document Stores	MongoDB
Wide-Column Stores	Cassandra
Graph Databases	Neo4j

2. Document Stores

Data Model:

- Document-Based:** Stores data as documents, typically in JSON or BSON format.
- Example Document:**


```
{
  "userID": "123",
  "name": "Alice",
  "age": 30,
  "address": {
    "street": "123 Main St",
    "city": "Springfield"
  }
}
```

Advantages:

- **Flexibility:** Dynamic schema allows for storing varied data structures.
- **Performance:** Better performance for hierarchical and nested data.
- **Ease of Use:** JSON/BSON is easily readable and writable, making it convenient for web applications.

Examples:

- **MongoDB:** Widely used document store known for flexibility and scalability.
- **CouchDB:** Known for its ease of replication and offline capabilities.

Diagram:

Document Store Example:

DocumentID	Document
1	{JSON data}
2	{JSON data}
3	{JSON data}

3. Design Considerations

Schema Design:

- **Best Practices:** Designing schemas based on use cases and query patterns to optimize performance.
- **Example:** Embedding vs. referencing documents to balance read and write performance.

Performance:

- **Indexing:** Creating indexes on frequently queried fields to speed up searches.
- **Sharding:** Distributing data across multiple nodes to enhance scalability.
- **Diagram:**

```
Document Store Schema Design:
-----
```

| Embedding vs. Referencing |

Embedding:

```
{
  "userID": "123",
  "name": "Alice",
  "orders": [
    {"orderID": "A1", "product": "Book"},
    {"orderID": "A2", "product": "Pen"}
  ]
}
```

Referencing:

```
{
  "userID": "123",
  "name": "Alice",
  "orderIDs": ["A1", "A2"]
}
```

These detailed explanations and diagrams should help clarify the key concepts of NoSQL and key-value stores, as well as document stores. Let me know if you need further details or additional sections covered!