CS 143 – Database Systems

Winter 2025 – Professor Ryan Rosario

Lecture 3/4: Relational Algebra

Notes

- Relational algebra works on unique sets of tuples
    - Using relational algebra operation removes duplicate tuples

Relational Operators

- Selection σ
    - Retrieves subset of tuples from a single relation that satisfies a particular constraint
        - $\sigma(R)$ → Gets all tuples in the relation and all attributes
            - SQL WHERE
        - σ_{ψ} → Filters tuples (not attributes)
            - $\sigma_{\psi}(R)$ → SELECT * FROM R WHERE ψ
                - $\sigma_{\text{likes} > \text{dislikes} \wedge \text{views} > 1000000}$
                - ψ is a filtering predicate
- Projection π
    - Extracts attributes from a set of tuples and removes duplicates
        - Normally the last thing in a relational algebra expression
            - $\pi_{\text{title}} (\sigma_{\text{likes} > \text{dislikes} \wedge \text{views} > 1000000})$
    - Can create new attributes
        - $\pi_{\text{likes}/(\text{likes} + \text{dislikes})}$
        - $\pi_{\text{likes}/(\text{likes} + \text{dislikes}) \rightarrow \text{favor}}$
        - Does not create a new attribute within the tuple, but only the output
    - Can rename attributes using →
        - $\pi_{\text{likes} \rightarrow \text{thumbs up}}$
    - Closest to SELECT DISTINCT

- Cartesian Product ×
  - Properties
    - Very slow
    - Requires a lot of RAM
    - Requires a lot of Disk Space
  - Notes
    - Missing or incomplete join conditions lead to Cartesian products
- Natural Join ⋈
  - RDBMS automatically picks the attributes k to join on
    - If no match, it will not output (Inner)
    - We get one copy of the join key in output
    - Edge: If no common attributes
      - R × S outputs
    - Edge: Common attributes but no matches on attributes
      - Empty set outputs
  - $R \bowtie S = \pi_{R \cup S} (\sigma_{R.k = S.k} (R \times S))$
    - K is join key
    - Cartesian Product → Apply a filter → Project attributes
- Theta Join $\bowtie_\theta$
  - Constraint θ determines how tuples from both relations will be combined
  - $R_1 \bowtie_\theta R_2 = \sigma_\theta (R_1 \times R_2)$
    - Start with Cartesian Product
    - Only keep tuples with matches on θ
      - If duplicate names on attributes, RDBMS will create aliases
        - This also creates two copies of join key
- Aggregation γ
  - Applies mathematical/statistical function to groups of tuples
    - Used for summarization: AVG(), SUM(), MIN(), MAX(), DISTINCT-COUNT()
    - $_{\text{major}}\gamma_{\text{AVG(gpa)}}$ (ugrad_students)
      - LHS: Attributes to group by

        - RHS: Functions to apply
    - Global Aggregate: $\gamma_{AVG(gpa)\rightarrow average}(R)$
    - $_{major}\gamma$ → SELECT DISTINCT major
  - Closest to GROUP BY
- Rename **ρ**
  - $\rho_{new}(R)$: renames relation R to S
  - $\rho_{new/old}(R)$: renames attribute old in relation R to new
- Set Union ∪
- Set Difference −
- Set Intersection ∩
  - Division is not implemented in SQL

Join Notations

- Equijoin: Joins of equality
  - A == B
- Nonequijoin: Joins of not equality
  - A > 3
- Inner: If match, output match. If no match, do not output
- Outer: Potential to Output NULLS
  - Left: Always keep the tuples from the LHS. If match on RHS, output RHS. If no match, output special tuple/NULL.
  - Right: Always keep the tuples from the RHS. If match on LHS, output LHS. If no match, output special tuple/NULL.

Lecture 9: Functional Dependencies, Normal Forms

Functional Dependencies

- A, B → C : Allowed
- A → B, C : Not Allowed

2NF Check

- Create Functional Dependencies
- Find Canonical Cover
    - Simplify to Singletons
    - Remove Composites comprised of Singletons
    - Remove Inferred Composites

Normal Form

- 1NF
    - All attributes are flat (No arrays, maps)
    - No columns logically reliant on each other (ex: Code1, Code2, Code3)
    - Unique Key exists (No duplicate tuples)
    - No null values
    - Row ordering does not matter
- 2NF
    - R is in 1NF
    - Attribute appears in a CK
    - Attributes depends on the **entire** key
        - If AB is CK, AB→C, and C→D, this is ok
    - Not partially dependent on **any** composite candidate key
        - If AB is CK and B→E, this is not 2NF (partial dependency)
            - Decompose into 2NF: (B, E), (B, everything else)
- Codd 3NF
    - R is in 2NF
    - Non-key attributes depend only on an entire CK

        - If all attributes are a part of a CK, R is in 3NF by definition
    - Non-key attributes must depend directly to an entire CK
        - If AB is CK, AB→C, and C→D, this is not 3NF
- Zaniolo 3NF (1 of following must be true)
    - A→B is trivial (LHS, RHS share an attribute)
    - LHS is superkey for R
    - RHS is <u>contained</u> in a CK
        - A→AB : B must be a part of a CK
        - If CK is AB, C→B is OK as B is paritally in CK
- BCNF
    - <u>For every functional dependency</u>:
        - A→B is trivial (LHS, RHS share an attribute)
        - LHS is superkey for R

Losslessness: Natural Join of subrelations returns original relation

- R1⋈R2 = R
- Decomposition Tests
    - R1 ∪ R2 = R
    - R1 ∩ R2 ≠ {} (There exists a join key)
    - R1 ∩ R2 is SK for R1, R2, or both
        - Closures: Determine if candidate is SK for R1 or R2
            - Attempt to derive all other attributes from SK, recheck for any changes in closure
            - Denoted with a + (ex: A+)
                - If A+ == A, A is a superkey for R

Dependency Preservation

- For each FD, check if you can use:
    - (Closure ∩ R)+ ∩ R to derive new attributes
        - If no, then not Dependency Preserving

Lecture 10: OLAP, OLTP, Warehousing

OLTP (Online Transaction Processing)

- Optimized for live traffic: random reads and writes, low latency, no patterned data
    - Databases optimize OLTP
- Random Access
- Normalization

OLAP (Online Analytical Processing)

- Read-only, optimized to be fast with low latency
- Used for internal systems. Other systems write to system
    - Incremental writes are very slow
- Columnar/cubic/hypercube data storage
    - Length, height, depth

OLAP Cube Vocab

- Slice: 1D query, Removes one dimension (gender=M), WHERE
- Dice: 2D query, (gender=M, location=US), WHERE AND
    - Outputs a specific piece of data
- Rollup: Hierarchical attribute, aka ranges (space and time), Up the hierarchy
- Drill Down: Take a column from a slice, aka take a more precise data
    - Outputs a range of data, used in hierarchical
- Pivot: Long data to wide data
    - Wide: (One row per student)
        - Less redundancy on student data
        - Easy to understand
        - Can have NULLs
        - Not as flexible (you need to ALTER TABLE to add a column)
    - Long: (Multiple rows per student)
        - More flexible
        - No NULLs

        - Redundancy
        - Harder to understand

Data Warehouses

- Designed for analytics, not production
- Optimized for fast reads
- Optimized for OLAP, batch processing
    - Batch Processing: Low latency reads (SELECT) of aggregated or precomputed data
        - ETL Job: Data Extract, Transform, Load (into warehouse)
            - Atomic
            - Some data is lost if insert/update while processing
                - Run ETLs on production database copy
                - Dual Writes: Updates changes replica
            - App→ProductionDB→Replica→DW
            - App → ProductionDB, Replica →DW
- When data loads, groups are preaggregated
    - By guideline (priori) decided by data engineer
    - Automatically (2^n dimensions)
- Stores normalized, denormalized/exploded tables for efficient reads
- Data is a little out of date

Data Warehouse Schemas

- Fact Table: Denormalized, contains the data to be analyzed
- Dimension Table: Data about attributes of each of facts
- Star Schema: Single fact table surrounded by several dimension tables
- Snowflake Schema: Central fact table where dimension tables may be joined to each other
- Constellation/Galaxy Schema: Multiple fact tables share dimension tables

Lecture 11: DuckDB, Distributed Systems, MapReduce

DuckDB

- Features
    - Pattern matching with columns
    - Pivot and Unpivot

Data Lake: Data storage where data is stored in different storage systems but can be queried from a single system

- Less preprocessing work, but more query minutia
    - No schema, generally raw data
- Hadoop: Distributed, scalable file system, runs on Map-Reduce
    - Better for disk operations

	RDBMS/Database	Data Warehouse	Data Lake
Data	Structured Tabular Clean	Structured Columnar/cube Fairly Clean	Unstructured
Schema	Highly normalized Fixed Schema	Not normalized Fixed Schema	No schema
Performance (Price may vary for all)	Fast for Transactions Fast for Reads, Writes	Fast for SELECTS	Mixed
Data Quality	Very high	Very high, but comes from mixed sources	Messy
Users	SWEs, SRE, DS	Analysts, DS, BA, SWE	Anyone
Analytics	Rowwise for Transactions Joins are painful	Denormalized Prejoined data Preaggregates	Anything

Distributed Systems

- Architectures
    - Replication: Decentralized, all nodes expected to have same data
        - All writes populate to all other nodes
    - Sharding: Centralized, different data on each machine
        - Typically, a hybrid model is used
- File Systems
    - Distributed
        - Distributed File Systems are logical
            - Systems are clusters, not individual
            - Files are divided into blocks
        - Loss of relational processing
    - Traditional
        - Storage devices are divided into partitions
        - Partitions are formatted with file systems
            - File systems are physical

Big Data: Datasets too big for RAM where parallel computing is required to handle data

- Maximizes usage of compute resources, increases amount of work per unit time, increase throughput (not speed)

MapReduce: Data Processing Platform

- Ideas
    - MapReduce functions on each split independently
    - Operates under Shared-Nothing Model → Extreme parallelism
- Map (Process and Transform)
    - Map: Outputs key-value pairs
        - Represented by relational algebra select σ
    - Combine (Preaggregate, optional)
        - Not used when there is not enough RAM
        - Allows for fewer disk writes, fewer network transfers
        - Combiner is normally the same as the reducer function

    - Partition: Merge results from split
        - Merge based on record
- Shuffle
    - Partitions send data to reducers using hash(key) % n
        - n is number of reducers
    - Reducers receive all key-value pairs with same hash
    - Represented by relational algebra aggregation Υ, otherwise it is represented by Map and/or Reduce phase
- Reduce (Aggregate)
    - Mergesort
        - Merge based on split
    - Reduce (Aggregation): Call reducer function on output to aggregate
        - Each reduce task outputs one output file containing all values associated with processed keys
            - If no reducers: Number of files = Number of map tasks = Number of input splits
        - Aggregation function represented by relational algebra reduce
        - Key Skew: Reducing of outlier data takes significant more time than a regular case
            - Solution: Custom partitioner
                - Hash uniformly by appending a salt (such that the same word maps to different buckets)
                    - Outputs a partial count
                - Remove uniform number
                - Aggregate (and do a 2nd Map Reduce Job)

Lecture 12: MapReduce, Spark, Streaming Systems

Pipeline: Processes input via series of algebraic steps

Spark (Resilient Distributed Datasets)

- Operations completed in RAM
    - Changes only persist after commits
- Lazy Evaluation
    - More efficient for iterative problems
    - head(), first(), take(), count(), collect(), collectAsList(), reduce(), show()
- Joins have poor performance
- File System is treated as truth

RDDs (Resilient Distributed Datasets)

- Processes datasets rather than files
- Immutable: You cannot modify relations → Increases performance
- Resilient: If one node fails, another replicate exists somewhere else in system
- Distributed: parts of data are in different nodes of system

Streaming Systems:

- Incomplete, Unbounded, continuous data
    - Streaming System is ground truth
    - Process data as it arrives without storing it
        - Lambda architecture: Streaming layer processes unstoreable data, Complete layer stores batched data
    - Events: Individual records
    - Messages: Data about event
- Streaming Models
    - Producer/Consumer
        - Producer: Creates/sends data to send into system
            - INSERT
        - Consumer: Reads data and processes it

        - SELECT → This is extremely inefficient
    - Push Model
        - Bipartite Graph
            - Direct connection between producer, consumer
            - Issues
                - Not Flexible: producers, consumers are hardcoded
                - If consumer goes down, entire system needs to be reconfigured
                - Producers are independent → Decentralized
- Data Loss
    - Use Cases:
        - 1. Replicate a write to the Database
        - 2. Computing Aggregates
    - Cases:
        - Is data loss a big deal in the streaming system?
            - 1. Yes
            - 2. No, assuming the messages are missing at random
        - Is receiving duplicate messages a big deal?
            - 1. Not really as long as a Primary Key exists
            - 2. Usually no, duplicates may wash out
        - Is receiving data in the wrong order a big deal?
            - 1. No. Relation is not ordered!
            - 2. No. Relation is aggregated!

Lecture 13: Stream Processing, Probabilistic Data Structures

Streaming Use Cases

- 1. Replicating writes to a backup database
    - Database handles everything
        - Performant but less flexible
    - Separate multiplexer sends requests to both production and replica databases
    - Streaming Systems
- 2. Processing aggregates from a stream
- 3. Making real-time decisions

Streaming Test Cases

- Can we tolerate data loss?
    - 1. No! We need to maintain data integrity
    - 2. Yes, as long as messages are missing at random
        - If not at random, bias can occur over long periods of time in a global aggregate
    - 3. Yes, but only when an event consists of many messages at a high velocity
        - No if we receive the message once every N seconds
- Can we tolerate message delay?
    - 1. To an extent yes
        - Data is complete, but not up-to-date
            - Client must adapt, but is ok for data warehouses
    - 2. Yes. Data will still be aggregated unless it arrives extremely late
    - 3. Yes. As long as it's not extremely late
- Can we tolerate out of order messages?
    - 1. Yes! There's no order in relations anyway, and we can add a time stamp
    - 2. Yes except for rolling/cumulative statistics or rolling windows
    - 3. Yes
        - No if we are dealing with a sequential/complex event

Message Broker: Lives in message bus, communicates with producers and consumers

- Maintains state of streaming system
    - Messages stored in broker buffer until consumer ACKs it
    - However, client and broker buffers may become full, losing packets
        - Delete oldest message in queue
        - Delete random message in queue
- Determining which client gets a message
    - Load balancing: deliver message randomly or based on shard key
    - Fan out: deliver message to all clients of a group
    - Hybrid is commonly used

Computing aggregates in streams

- Compute over window w
    - Issue: Stragglers (events that arrive in window but are not included in computation)
        - Ignore stragglers
        - Open window w for a little longer
            - We may store more data in buffer than needed
        - Update aggregation afterwards
            - Aggregation function needs to be updatable
                - Easy: count(), min(), max(), avg(), sd()
                - Hard: mode(), median()
- Compute over all time
- Update Rule (Can use with or without windows)
    - Compute aggregates across all messages of a given window, even if all messages are evicted from the window

Bloom Filter (Filters Duplicate messages)

- Returns false positives with small probability
    - Bit string of length m associated with k hash functions

HyperLogLog (outputs amount of distinct values in a set)

- Treat bitstring as a binary number, provide an estimate of number of distinct objects
    - Log2log2(2^32) can be stored in 5 bits
        - Average error of 2%

Lecture 14: CAP, NoSQL (MongoDB, Neo4j)

NoSQL

- "Not only SQL" → DML or DDL is something other than SQL
- "No SQL" → No use of relational model

CAP Theorem

- It is impossible for a distributed system to satisfy all 3 of the following:
  - **C**onsistency: Every read receives the *most recent* write, or an error is thrown
  - **A**vailability: Every request receives some kind of response (not an error), without guaranteeing the most recent write. *It always returns a response*.
  - **P**artition Tolerance: The system continues to function even if messages are lost or are delayed by the network between or among nodes
- MongoDB **supports** CP/AP

NoSQL Data Models

- Key-value store (Redis, Memcached)
  - Redis is single node, so CAP does not apply → AP
  - Redis commands take a key and a value, or just a key
  - Everything is stored in RAM (for indexing)
- Columnar (Cassandra, HBase, DynamoDB)
  - Fixed number of columns but each row has a subset of columns
    - Key-value pair where the value is a collection of key-value pairs
  - Cassandra
    - Strictly ordered, not relational
- Document store (MongoDB, CouchDB)
  - Documents are schemaless → Stored in JSON/BSON
    - No joins
    - Documents ⊆ collections ⊆ databases
  - CP
    - Consistency from allowing writes on only primary replica

        - Availability from using secondary replica sets, one or more with the exact same data
        - Horizontal Sharding → Data is distributed across nodes of network based on shard key
    - ODMs (like Mongoose) support CRUD
        - Create, Read, Update, Delete
    - Indexed by B-Tree → Simple and compound fields
- Graph (Neo4j, OrientDB, Giraph)
    - Vocabulary
        - Nodes represent entities
        - Edges represent foreign keys
            - Nodes and edges have attributes called properties and also labels
    - Neo4j does not have CAP → Graphs cannot be distributed
    - Indexed with B-Tree → indexes properties of nodes and relationships

Lecture 15: Data System Architecture, Disk Storage, B+ Trees

Data System Architecture

Data System:	Use:
DBMS	Production Data
Key-value store/cache	Fast access to cache, frequently accessed data
Data Warehouse	Dashboarding, Reporting, Fast access to aggregated data
Search Engine	Quickly locate records
Big Data System, Relational Layer	ETL Jobs

Moving Data

- ETL Jobs
    - Runs nightly, transfers data from one system to another
- Replication
    - Simply copies data to multiple locations of same type
- Streaming / Message Bus
    - Publisher/Subscriber model: Subscribers read data from bus and push to databases

HDDs

- Vocabulary
    - Tracks are divided into sectors
        - Sectors can achieve locality if data is on same cylinder
    - Data is addressed by block number and transferred via blocks
        - Pages are the same as blocks
        - We want to minimize seeks and block transfers
- Disk Performance
    - Seek Time: Time it takes to move head from parked state to a track
    - Rotational Latency: Once head is on a track, the time to find a sector
    - Access Time: Time from request to start of transfer.

        - Access Time = Seek Time + Rotational Latency
    - Data Transfer Rate: Time to transfer one sector from disk to RAM
    - IOPS: Cloud computing, burstiness, Number of Read/Writes per second
- Data Access
    - Sequential (HDD)
        - Access blocks in a pre-determined order
            - Very fast if data is contiguous
    - Random Access (SSD)
        - Blocks fetched from arbitrary locations
            - Bad on HDD
- Speeding up Disk Reads/Seeks
    - Buffering: Read blocks into memory, when out of RAM, evict based on policy
        - Evict using Most Recently Used (MRU)
    - Read-Ahead: Capture the next few blocks when retrieving a block from disk
    - Scheduling: Group requests by cylinder
    - File Organization: Organize blocks and records into files
        - Records $\subseteq$ Blocks $\subseteq$ Files $\subseteq$ Relations
    - Non-Volatile Write Buffers: Queue Read/Write requests in NVRAM on HDD

Organizing Records into Files

- Fixed types
    - Pointer to first record
    - Pointer to first empty record (Free List)
        - On deletion, update previous record's pointer to next record
        - Update the free list to store new space
    - Overflow Block
        - Create a new block, store the overflow, update the pointers
- Variable-length types
    - Table of Contents
        - Variable Sized Attributes, Data for fixed sized attributes, null bitmap, data for variable sized attributes

        - We store a pointer to the first record in the file header

Organizing Records into Blocks

- Blocks contain headers
    - Number of records in the block
    - Location of end of free space in the block
    - Location and size of each record
- Modifying the file means you need to rebuild the entire file

The Index Data Structure (for indexing files)

- Index: Key → Block Number, Byte Offset
    - Ordered Indices: Based on sorted ordering of sort-keyvalues
    - Hash Indices: Based on uniform distribution of values across a range of buckets
- Types of Ordered Indices
    - Primary (Clustering)
        - Both values of keys in index and records in corresponding data file are sorted by search key
            - Dictates ordering of underlying data file
            - Index and Records are separate
        - Does not need to be a Primary Key or Unique, but generally is
            - There is at most one primary index in a table
        - If there are duplicates, the first value is returned
        - Good for sequential access
    - Secondary (Non-Clustering)
        - Index and keys in records are sorted differently
            - Generally Random Access performance
    - Dense: Every value of the search key from the records is also in the index
    - Sparse: Only some of the possible key values are in the index
        - To search for a key, we find the nearest key that is less than the one we are trying to find
        - We walk the pointers until we find what we need

        - Keys in index and data are still ordered in the same way
    - Index may fit in RAM
    - More block transfers
        - To minimize block transfers, pick keys associated with records at the beginning of blocks

B+ Trees

- Concept
    - Good for range queries (non-equi), ok for equality (equi)
    - Root and interior nodes form sparse index
    - Leaf nodes for a dense index on top of the records
- Guarantees
    - Requires all paths have the same length from the root
    - Balanced and self balancing
    - Has O(n) fan out to reduce number of lock reads
    - Represents 1 disk seek and Block Transfer at each level of the tree
    - Left is <, Right is ≥
    - Holds up to n pointers but must contain roof (n/2) pointers
    - Some values may be duplicated within the tree
- Complexities
    - Insertions and deletions: O(log(roof n/2) k)
    - Additional space overhead
    - Wasted space since nodes only need to be half full
    - Reading each level of tree + sibling leaves = 1 seek + 1 block transfer

Lecture 16: Hash Indices, Nested Loop Join Algorithms

Hash Indices

- Concepts
    - Good for equi joins (equality)
    - Each key index maps into a bucket via a hash function h
        - Bucket contains a pointer to a linked list of records or blocks of records in RAM
        - H is hash mod function → h: K → B
    - Insertion
        - Compute Bucket via hash
        - Add the record
            - If does not fit, add an overflow bucket
    - Deletion
        - Compute h(K)
        - Find all records that match key K, delete them
            - If block is empty, delete the block
    - Hash Collisions/Bucket Overflow
        - Too little buckets causes too many collisions
            - Static Hashing: We can preallocate extra buckets beforehand
        - Too many buckets wastes resources
        - Bucket skew: Some buckets have a lot of records and overflow buckets while others don't
    - Ideal Buckets = [(number of records in relation R) / ratio of records per bucket] * 1.2
    - Avoiding Bucket Overflow
        - Choose hash function based on anticipated size of file
            - Performance degradation lessened, space is wasted
        - Reorganize the hash structure due to file growth
            - Slow
        - Dynamically Resize Buckets

        - Wastes space, minimizes collisions
    - Dynamic Hashing: Grow or shrink amount of buckets
        - Linear Hashing, Extendible Hashing

Query Processing

- Optimizer: Optimizes query into query/execution plan
- Process
    - Parsing and Translation
        - Checks for table existence, converts to relational algebra
        - Optimizer selects most optimal plan
    - Optimization
        - Criterium:
            - Block Transfers and Seeks
            - Number of tuples
            - Current state of CPU/RAM
            - Data transfer speed
            - Disk Space
            - Time
    - Evaluation
        - Execution Time
        - Block reads and disk seeks

Formulas

- t_T : Time to transfer 1 block from disk to memory
- t_s : Average block access time (including rotational latency)
- Average time to transfer B blocks with S random accesses: $Bt_T + St_s$
- SELECT σ
    - b_r is number of blocks in the file for R
    - Execution time:
        - Sequential File
            - $t_s + b_r * t_T$

- Non-Contiguous File then we must seek for each block
  - $b_r * t_s + b_r * t_T = b_r (t_T + t_s)$
- Search on key using ISAM, there is only one match
  - Best Case: Key/record is in first block
    - $t_s + b_r * t_T$
  - Worst Case: Key/record is in last block
    - $t_s + t_T$
  - Average Complexity for Uniform Distribution
    - $t_s + (b_r / 2) * t_T$
- Indexed using clustering B+ tree, search on equality and unique key
  - h: height of the B+ tree, can be depth
  - Traverse from root to leaf node
    - $h(t_s + t_T)$
  - Search for record pointer in block
    - 0 → it's already in RAM
  - Retrieve record associated with the key
    - $t_s + t_T$
  - Total Time doing disk I/O
    - $h(t_s + t_T) + t_s + t_T = (h + 1)(t_s + t_T)$
- Indexed using a primary/clustering B+ tree, search on a key v such that K $\geq$ v or K > v
  - Find v and go to leaf node
    - $h(t_s + t_T)$
  - Fetch the first record
    - $t_s + t_T$
  - Because it's primary, we can walk to pointers
  - Total:
    - $h(t_s + t_T) + t_s + b * t_T$
    - b: number of record blocks we need to read to find a match

Nested Loop Join

- Three for loops: Iterate through each pairing of outer, inner relations
    - Does not require an index
    - No restrictions on join condition θ
    - Checks per tuple
    - Super Complex: $O(n_r * n_s)$
        - n_r : number of tuples in R
        - n_s : number of tuples in S
    - Assuming maximum of 1 block of R, S can fit in RAM and R, S are sequential
        - Note: head can only be in one place at a time
        - b_r : number of blocks in R
        - b_s : number of blocks in S
    - Seeks and Block Transfers
        - Outer Relation Inner Relation
        - $n_r + b_r$ seeks
        - $n_r * b_s + b_r$ block transfers

Block Nested Loop Join

- Idea: Process outer relation by block and not tuple
    - Each block in inner relation is read only once for each block in outer relation
- Complexity: Outer Relation Inner Relation
    - $b_r + b_r$ seeks
    - $b_r + b_s * b_r$ block transfers
    - Total Disk I/O:
        - $(2\ b_r)t_s + (br * b_s + b_r)t_T = 2\ b_r * t_s + (t_T * b_r)(b_s + 1)$

Indexed Nested Join Loop

- Idea: Place an index on the inner relation to replace file scans on lookups
    - For each tuple in R, we lookup a key from the tuple is S and we can retrieve all matching tuples
- Complexity (Assuming index already exists)
    - $b_r\ (t_T + t_s)$ time to scan

- $c * n_r$ lookups
  - c is disk cost of a select from the index
  - $c = h(t_T + t_s) + (t_T + t_s)$
    - Find a key, fetch a record
- Total Disk I/O:
  - $b_r (t_T + t_s) + c * n_r$

Lecture 17: Merge Join, Hash Join, Spark Joins, Indexing Strategies

Merge Join/Interleaved Linear Scan

- Idea: Sort both relations before we join them, so we only need to scan each table once
    - Sort based on common attributes in R ∩ S
- Complexity:
    - $b_r + b_s$ block transfers
    - 2 seeks (1 for S, 1 for R)

Hash Join

- Idea: Build and Probe Sides
    - Build Side S: Builds a hash table from build side relation to variety of things in memory
        - Must fit in memory
        - Value can be: Linked list of records, Linked list of pointers to records on disk, or linked list on disk
    - Probe Side R: Checks keys if it is in hash table on build side
        - If match, there might be a match on join key
            - Hash collisions → Lookup hash on join side to find records with proper key
- Procedure
    - Construct a hash table using the values of the join key from S (and hash them)
    - Hash table value can be one of three things:
        - Linked list, in memory, of pointers to records on disk (common)
        - Linked list, on disk, of blocks of records (disk-based)
        - Linked list, on memory, of records (memory-based)
    - Do a full-table scan of R and search for records that match the join key
        - Due to hash collisions, we must walk the linked list to find all the records that match the key (and not just the hash)
- Complexity
    - Common Approach

        - Build Phase (S):
            - b_s Block Transfers + 1 Seek
        - Probe Phase (R):
            - b_r seeks + m seeks
        - Total:
            - m is number of matches on the join key
            - Block Transfers: $b_r + b_s + m$
            - Seeks: $b_r + m + 1$
    - Disk-Based
        - Build Phase (S):
            - 1 seek + b_s block transfers
        - Probe Phase (R):
            - 1 seek + b_r block transfers
        - Total
            - Seeks: 2
            - Block Transfers: $b_r + b_s$

Misc questions

- Difference between hash table and hash index
    - In hash join, RDBMS creates an ephemeral hash table
        - Lives as long as the join does, some part is in memory
        - Keys are in memory → Less disk I/O
    - A hash index is persisted on disk
        - Lives in the database schema
        - Lots of disk I/O

Spark's Selection of Best Join condition

- Size of tables involved in the join
- Join condition
- Configuration values (defaults or spec of cluster)
- Hints given by the user

Spark Join Algorithms

- Broadcast based joins: Table is broadcasted to all executors in the cluster
    - Automatically chosen if the table is less than 10MB
    - Broadcast Hash Join
        - Small dataframe R that fits in memory on all executors and driver
        - R is divided across worker nodes
        - Only supports equijoins
    - Broadcast Nested-Loop Join
        - Process cartesian join in each executor as a nested loop join after broadcasting R across the cluster
        - Least performant, used as a last resort
        - Can handle all join types
        - Complexity
            - $n_r * b_s + b_r$ Block Transfers
            - $n_r + b_r$ Seeks
- Shuffle based joins
    - Called when neither relation fits in RAM on every executor and the driver
        - Heavily impacted by key shew
        - Lots of Disk I/O and Network I/O
    - Shuffle Hash Join
        - Each row of R and S is shuffled across the network to a partition such that all rows with the same value on the join key ends up in the same partition. Local joins afterwards
            - Doesn't require keys to be sortable
            - Only works for equijoins
            - A lot more expensive than a broadcast
            - Uses additional RAM and time
    - Shuffle Sort-Merge Join
        - After R, S shuffled to executed partitioned by key-value, each executor may contain rows of R and S with multiple different key-values

        - All rows of the same key-value must be co-located in the same executor
    - Notes
        - Keys must be sortable
        - Works on any equijoin

Indexing Strategies

- Spatial Data and Temporal Data
    - Nearest Neighbor Query: Circle within certain distance around user
    - If 1D, divide and conquer via subintervals
    - If 2D, Use a k-d tree
        - O(logN) time on average, O(N) in sequential files
    - Data visualization: Random Forest, kNN, Voronoi tesselation

Lecture 18: Transactions, Isolation Levels

Concurrency: Two transactions start, execute, and complete in overlapping time periods to share a resource

- Increasing concurrency/performance decreases Isolation and Consistency

Parallelism: Two operations in a transaction literally execute at the same time

Serial: Transaction runs entirely in full before another one begins

Transactions Commits

- Changes to database does not occur until commit is called
    - Begin transaction, begin work, begin
    - Atomic → Either all execute correctly, or nothing happens
        - Failures
            - Constraint (FK, PK, UK)
            - Hardware/Software Failure
- Lifecycle
    - Committed: Transaction completes successfully and entirely
    - Partially Committed: After all steps run successfully, but before the result has been written to the database
    - Failed: All changes are rolled back and transaction becomes aborted
    - Active: Transaction is still processing data
- Errors
    - External Writes: RDBMS commits a transaction but the application does not complete its work
        - App must ask the database system to issue a compensating transaction to undo the actions of the first transaction

ACID

- Atomicity
    - No partial results
- Consistency

    - We do not gain or lose data
- Isolation
    - Concurrent transactions operate independently and do not intersect
        - Maximizes consistency at the cost of performance
- Durability:
    - Once a transaction is committed, you cannot roll back and the data persists on disk

Relational databases follow ACID

- High consistency

NoSQL transactions follow BASE

- Basically Available
    - Ensure availability of data by spreading and replicating across nodes of cluster
- Soft-state
    - Because of lack of consistency, data values may change over time
- Eventually Consistent
    - Over time, data may become consistent (immediate consistency is not guaranteed
- High availability with eventual consistency

Thread Chart notes

- Consistent: Sum of initial registers is same as sum of threads after serialization
    - Conflict Serializable: If a schedule can be rearranged to become serial/matches behavior of a serial schedule
    - Serializability
        - Improved throughput
        - Reduced waiting time
- Writes are global after commits
    - Rollbacks no longer affect registers after commit
- Swaps move one instruction up and another down across two threads
    - You cannot swap incompatible instructions

- Errors
    - WAR, WAW are not compatible
    - RAR is ok
- Problems
    - Dirty Writes: T_i writes/modifies a register that T_j has already written/modified while T_j is running
    - Dirty Reads: When transaction can read uncommitted changes from other transactions
        - Only read uncommitted allows dirty reads
        - Causes data inconsistencies
    - Fuzzy Read/Non-Repeatable Read: T_i reads a value and then another transaction overwrites the value and commits it
        - T_i reads the data twice and receives different values
    - Phantom Phenomenon/Phantom Read: One transaction inserts/deletes rows on a table in between two fetches (SELECTs) in another transaction
        - Occurs on a range of records
        - Occurs in Read Uncommitted, Read Committed, and Repeatable Read
        - The existing individual records do not change in T_2
        - Preventable using serialization or predicate locking

Precedence Graph

- Draw edge if:
    - WAR, RAW, WAW
    - Denote duplicate edges for correction
- Properties
    - If no cycles → S is conflict serializable and guarantees consistency
    - If T_i → T_j, exists in the graph, then any serial schedule S' == S if T_i finishes before T_j begins executing
    - Use topological sort to find proper ordering

Isolation Layers

Most Consistency, Isolation. Least Concurrency

- Serializable
    - Maximum Consistency, Poor performance
    - Little concurrency, equivalent to a serial schedule
- Repeatable Read
    - If T_i reads X, no other transaction T_j can update it until T_i commits
        - Lots of locking!
        - MySQL
- Read Committed
    - T_i can only read data from T_j that is committed
        - DB2, SQLServer, Postgres
- Read Uncommitted
    - T_i can read data written from any other transaction even if not committed

Least Consistency, Isolation. Most Concurrency

Lecture 19: Transactions, Locking

Locking

- Shared Locks
    - Allows T_i to read a value of X
    - Transactions can hold multiple shared/read locks on X
- Exclusive Locks
    - Allows read/write access to T_i
        - But we must use an exclusive lock to write/update X
    - If T_i holds an exclusive lock on X, no other transaction can update it
        - Other transaction will block

Two-Phase Locking (2PL)

- Properties
    - Ensures serializability
        - As soon as a transaction releases one lock, it can no longer acquire any new locks
    - No Anomalies
        - Each transaction pulls on the data it needs and no other transaction can modify these values
    - No guarantee of consistency
    - Not schedule specific
- Growing Phase
    - T_i can acquire all needed locks in the phase, but it cannot release any of them
- Shrinking Phase
    - Begins as soon as a transaction releases a lock
    - T_i can release locks but cannot acquire any more

Locking Issues

- Starvation: A given thread will never get the lock because other transactions keep requesting shared locks
    - Solution: Lock Manager: Grants a lock on data item A in a particular mode M if:

        - No other transaction holds a lock on A that conflicts with M
        - There is no other transaction that is waiting for a lock on A and that made its lock request before T_i
    - Queue processes lock requests via a Lock Table
- Timing
    - Unlocking Too Early
        - Lose Serializability
        - Dirty Reads, Non-Repeatable Reads
    - Unlocking too late
        - Deadlock
- Deadlock: T_1 wants T_2's lock, T_2 wants T_1's lock, and nothing happens
    - Prevention:
        - Lock Manager or roll backs
            - Wait-for graphs computed to detect deadlock
        - Acquire all required locks at once
        - Acquire locks in non-cyclical order
        - Preemption: Pause the other transaction until current transaction finishes with lock. Roll back
        - Lock timeouts
    - Detection and Recovery
        - Choose a victim transaction to abort and roll back
        - Rollback
        - Repeat until no deadlock
            - Oftentimes Round Robin works better