CMPT 354 Final Exam Cheat Sheet

CMPT 354 Student

December 12, 2025

Course: CMPT 354 - Database Systems I
Date: December 12, 2025

Contents

1 Database Systems I: Comprehensive Final Exam Study Guide

1.1 Table of Contents

1.2 Introduction to DBMS [Lec 1-2]

1.2.1 What is a Database?

- **Database:** An organized collection of data.
- **DBMS (Database Management System):** A software system that stores, manages, and facilitates access to data (e.g., PostgreSQL, MySQL, Oracle).

1.2.2 Why Use a DBMS? (vs. File Systems)

In file systems, data is stored in many files and accessed by specific application programs. This causes:

1. **Redundancy/Inconsistency:** Data is duplicated across files .
2. **Access Difficulties:** Hard to retrieve data across multiple files or access single records efficiently.
3. **Concurrency Issues:** Multiple programs modifying data simultaneously cause conflicts.
4. **Atomicity Issues:** If a system crashes during a complex update, data may be left in an inconsistent state.

DBMS Solutions/Functions:

- **Persistent Storage:** Keeps data safe.
- **Concurrency Control:** Manages multi-user access safely.
- **Crash Recovery:** Ensures data integrity after failures (e.g., power outage).
- **Data Independence:** Separates the *logical* view of data from *physical* storage.

1.2.3 Evolution of Data Models

1. **Navigational (1960s):** Data organized as records with pointers (graphs/trees). Access requires following pointers (navigating). Hard to maintain .
2. **Relational (1970s - Present):** Proposed by **Edgar F. Codd**. Data stored in tables. Uses a **declarative** query language (SQL). The DBMS decides *how* to execute queries, providing **Physical Data Independence** .
3. **NoSQL (2000s):** Emerged for scalability and flexibility (e.g., Key-value, Document stores).
4. **NewSQL (2010s):** Attempts to combine SQL consistency with NoSQL scalability.

1.3 The Relational Model [Lec 3]

1.3.1 Structure

- **Relation:** A table with rows and columns.
- **Attribute:** A column. Has a name and domain (type). Set-valued attributes are not allowed.
- **Tuple:** A row. Represents a single record. Duplicate tuples are not allowed in the strict relational model.
- **Schema vs. Instance:**
  - *Schema:* Metadata specifying logical structure (defined at setup, rarely changes).
  - *Instance:* The actual content/data (changes rapidly).
  - **Analogy:** Schema is the blueprint of a house (doesn't change much). Instance is the people and furniture inside (changes often).

1.3.2 Relational Algebra (RA)

A procedural language describing operations on relations.

Core Operators:

1. **Selection (σ_p):** Filters **rows** based on a predicate p. Monotone.
   - *Kind Explanation:* Picking specific items from a list. "I want all students older than 20." (Horizontal slice)
2. **Projection (π_L):** Filters **columns** based on list L. Removes duplicates. Monotone .
   - *Kind Explanation:* Picking specific details. "I want just the names of the students, ignore their ages." (Vertical slice)
3. **Cross Product ($\times$):** Pairs every row of R with every row of S. Monotone.
   - *Kind Explanation:* Making every possible combination. If you have 3 shirts and 2 pants, cross product gives you 6 outfits.
4. **Union ($R \cup S$):** Combines rows from R and S. Requires identical schema. Monotone .
5. **Difference ($R - S$):** Rows in R but not in S. **Non-monotone** (adding rows to S can reduce output).
6. **Renaming (ρ):** Renames tables/columns to avoid ambiguity.

Derived Operators:

- **Join ($\bowtie_p$):** Shorthand for $\sigma_p(R \times S)$.
- **Natural Join ($\bowtie$):** Equates all identically named columns and removes duplicates of those columns .
- **Intersection ($R \cap S$):** $R - (R - S)$. Monotone.

1.4 Database Design: E/R Model [Lec 4]

Entity-Relationship (E/R) Model is a high-level conceptual design model.

1.4.1 Components

1. **Entities:** "Things" or objects. Represented by **Rectangles**.
2. **Attributes:** Properties of entities. Represented by **Ovals**.
   - **Keys:** Underlined attributes that uniquely identify an entity.
3. **Relationships:** Associations among entities. Represented by **Diamonds**.
   - Can have their own attributes (e.g., `fromDate` on `IsMemberOf`).

1.4.2 Constraints

- **Multiplicity:**
  - **Many-Many:** Lines with no arrows. (User $\leftrightarrow$ Group) .
  - **Many-One:** Arrow pointing to the "One" side. (Group $\rightarrow$ Owner) .
  - **One-One:** Arrows on both sides.
- **Weak Entities:**
  - Entities that cannot be identified by their own attributes alone. They depend on a "supporting" entity via a relationship.
  - Represented by **Double Rectangles** and **Double Diamonds**.
- **ISA Hierarchies:**
  - Represent subclasses/inheritance.
  - Represented by a **Triangle** (labeled ISA).

1.5 E/R to Relational Translation [Lec 5]

1.5.1 Basic Translation

1. **Entity Set $\rightarrow$ Table:** Attributes become columns. Key becomes Primary Key .
2. **Relationship $\rightarrow$ Table:** Columns are Keys of connected entities + Relationship attributes .
   - *Optimization:* If Many-One, the relationship table can often be merged into the "Many" side entity table.

1.5.2 Translating Advanced Concepts

- **Weak Entity Sets:** The table must include the key of the supporting entity set (as a Foreign Key) + its own partial key.
- **ISA (Subclasses) - 3 Approaches:**
  1. **E/R Style:** Tables for superclass and subclasses. Subclass tables strictly contain only subclass-specific attributes + Superclass Key.
  2. **OO Style:** Tables for subclasses only. Each contains *all* inherited attributes. Redundant if entities exist in multiple subclasses .
  3. **NULL Style:** One giant table. Attributes not applicable to a specific entity are set to NULL .

1.6 Relational Design Theory: Normalization [Lec 6]

Goal: Systematically remove redundancy to prevent update, insertion, and deletion anomalies. * **Analogy:** Organizing your closet so you don't have 5 pairs of the exact same socks in different drawers. If you lose one, you don't have to check 5 places.

1.6.1 Functional Dependencies (FDs)

- $X \rightarrow Y$: If two tuples agree on attributes X, they must agree on attributes Y.
- **Key:** K is a key if $K \rightarrow$ All Attributes and K is minimal.
- **Attribute Closure (X^+):** The set of all attributes functionally determined by X.

1.6.2 Normal Forms

- **BCNF (Boyce-Codd Normal Form):** A relation is in BCNF if for every non-trivial FD $X \rightarrow Y$, X is a **superkey**.
  - *Decomposition:* If $X \rightarrow Y$ violates BCNF, decompose R into $R_1(X, Y)$ and $R_2(X, Z)$ (where Z is remaining attributes). This is guaranteed to be a **lossless join** decomposition.

- **MVD (Multivalued Dependency):** $X \rightarrow\rightarrow Y$. Given X, Y is independent of the rest of the attributes.
- **4NF:** A relation is in 4NF if for every non-trivial MVD $X \rightarrow\rightarrow Y$, X is a **superkey**. 4NF implies BCNF.

1.7 SQL Basics [Lec 7-8]

1.7.1 Basic SFW Query

```
SELECT [DISTINCT] A1, A2...
FROM R1, R2...
WHERE condition;
```

- **Bag Semantics:** SQL allows duplicates by default (Bag). Relational Algebra uses Set semantics (No duplicates).
- **ORDER BY:** Sorts output. Can use `ASC` or `DESC`. Can use `LIMIT`.

1.7.2 Set Operations

- **Set Semantics:** `UNION`, `EXCEPT`, `INTERSECT` (Duplicates eliminated).
- **Bag Semantics:** `UNION ALL`, `EXCEPT ALL`, `INTERSECT ALL` (Duplicates preserved based on math of counts).

1.7.3 Joins

- **Cross Join:** `FROM A, B` (Cartesian product).
- **Inner Join:** `A JOIN B ON A.id = B.id` (Only matching rows).
- **Outer Joins:**
  - `LEFT OUTER JOIN`: Keeps all rows from left table, pads missing right columns with NULL.
  - `RIGHT OUTER JOIN`: Keeps all rows from right table.
  - `FULL OUTER JOIN`: Keeps all rows from both, padding where necessary.

1.7.4 Subqueries

- **Scalar Subquery:** Returns a single value. Can be used in `WHERE` clauses (e.g., `WHERE age = (SELECT ...)`).
- **IN / NOT IN:** Checks membership in a set of results.
- **EXISTS / NOT EXISTS:** Checks if subquery returns *any* rows. Often used for correlated subqueries.
- **ANY / ALL:** Comparison against a set (e.g., `> ALL`).

1.7.5 Aggregation

- **Functions:** `COUNT`, `SUM`, `AVG`, `MIN`, `MAX`.
- **GROUP BY:** Groups rows by values. Aggregates are computed per group.
- **HAVING:** Filters **groups** (applied after aggregation). `WHERE` filters **rows** (applied before aggregation).

1.8 SQL Intermediate: Modification & Constraints [Lec 9]

1.8.1 Data Modification

- `INSERT INTO Table VALUES (...)` or `INSERT INTO Table (SELECT ...)`.
- `DELETE FROM Table WHERE ...`.
- `UPDATE Table SET col = val WHERE ...`.

1.8.2 NULLs

- **Logic:** 3-valued logic (TRUE, FALSE, UNKNOWN).
- Comparisons with NULL return UNKNOWN.
- `WHERE` clause only accepts TRUE (drops FALSE and UNKNOWN).
- **Aggregates:** Generally ignore NULLs (except `COUNT(*)`).
- **Functions:** `COALESCE(x, y)` returns first non-null value.

1.8.3 Constraints

- **NOT NULL:** Forbids NULL values.
- **PRIMARY KEY:** Unique and Not Null.
- **FOREIGN KEY:** Enforces referential integrity (no dangling pointers). Options on delete: `REJECT`, `CASCADE`, `SET NULL`.
- **CHECK:** Enforces specific conditions on a row/attribute (e.g., `age > 0`).

1.8.4 Triggers

Event-Condition-Action (ECA) rules.

- **Event:** INSERT, UPDATE, DELETE.
- **Type:** `FOR EACH ROW` vs `FOR EACH STATEMENT` .
- **Timing:** `BEFORE` vs `AFTER` vs `INSTEAD OF` .
- **Variables:** Access data via `OLD ROW/TABLE` and `NEW ROW/TABLE`.

1.8.5 Views

- **Virtual Tables:** Defined by a query. Not stored physically (unless materialized).
- **Usage:** Simplifies complex queries, hides data (security), provides logical data independence .
- **Updating Views:** Difficult/Impossible for many views (e.g., those with aggregates). `INSTEAD OF` triggers can handle view updates logic.

1.9 SQL Programming & Application Architecture [Lec 10]

1.9.1 Integration Approaches

- **API (Call-Level Interface):** Application sends SQL commands to DBMS at runtime (e.g., Python `psycopg2`, `SQLAlchemy`, JDBC, ODBC).
- **Embedded SQL:** SQL embedded directly in host language (e.g., C) with a preprocessor. Hard to maintain.
- **SQL/PSM (Persistent Stored Modules):** Stored procedures/functions stored inside the DB. Reduces data shipping, pushes logic to data.
- **ORM (Object-Relational Mapping):** Automatically maps database tables to classes/objects (e.g., SQLAlchemy). Convenient but complex for advanced queries.
- **Language Integration:** SQL-like constructs built into the language (e.g., LINQ).

1.9.2 The Impedance Mismatch

- **Problem:** SQL operates on **sets** (bags) of records, while low-level programming languages operate on **one record** at a time.
- **Solution (Cursors):** A mechanism to iterate over a result set one row at a time.
  - *Operations:* Open, Get Next (Fetch), Close.

1.9.3 Security: SQL Injection

- **Attack:** Malicious user input alters the intended SQL query structure (e.g., "Bobby Tables": `name = "Robert'); DROP TABLE Students;--"`).
- **Prevention:**
  1. **Sanitization:** Escaping characters (e.g., single quotes).
  2. **Prepared Statements:** Use placeholders (e.g., `:name` or `$1`) so the DBMS treats input as data, not code.

1.9.4 Application Architectures

- **Two-Tier (Client-Server):** Client handles presentation; Server handles business logic + data.
- **Three-Tier:**
  1. **Presentation Layer:** Client (Browser/Mobile).
  2. **Middle Layer (App Server):** Business logic, control flow.
  3. **Data Management Layer:** DBMS.

1.10 SQL Transactions & ACID [Lec 11]

1.10.1 What is a Transaction?

- **Definition:** A transaction is a container for multiple operations (statements) that are treated as a single logical unit of work.
- **Analogy:** Think of a bank transfer. You deduct money from Account A and add it to Account B. These are two separate operations, but they must happen together or not at all. The "Transaction" wraps these two steps.

1.10.2 ACID Properties

- **Atomicity (All-or-Nothing)**
  - **Concept:** A transaction is indivisible. Either all its operations happen, or none of them do.
  - **Why?** If a crash happens halfway through a transfer (money left A but didn't reach B), we want to cancel everything so money isn't lost.
  - **Mechanism:** Handled by **Logging (Undo)**.
- **Consistency (Correctness)**
  - **Concept:** The database must move from one valid state to another valid state. It must satisfy all defined rules (constraints) like "Account balance cannot be negative".
  - **Why?** To ensure data makes sense.
- **Isolation (Independence)**
  - **Concept:** Multiple transactions running at the same time should not interfere with each other. Each transaction should feel like it's the only one running.
  - **Why?** If two people try to buy the last ticket at the exact same time, we need to handle it cleanly so they don't both think they got it.
  - **Mechanism:** Handled by **Locking** or **MVCC**.
- **Durability (Permanence)**
  - **Concept:** Once a transaction is "Committed" (saved), the changes are permanent, even if the system crashes or power fails immediately after.
  - **Why?** You don't want to lose your deposit just because the server restarted.
  - **Mechanism:** Handled by **Logging (Redo)**.

1.10.3 Isolation Levels & Anomalies

SQL standard levels (weakest to strongest):

Isolation Level	Dirty Read	Non-repeatable Read	Phantoms	Implementation Note
READ UNCOMMITTED	Possible	Possible	Possible	Short duration locks.
READ COMMITTED	Impossible	Possible	Possible	Long write locks, short read locks. Default in Postgres.
REPEATABLE READ	Impossible	Impossible	Possible	Long duration locks on all items accessed.
SERIALIZABLE	Impossible	Impossible	Impossible	Range locks.

- **Dirty Read:** Reading uncommitted data.
- **Non-repeatable Read:** Reading the same item twice yields different results (due to another TX update).
- **Phantom:** A range query returns different set of rows (due to another TX insert/delete).
- **Snapshot Isolation:** Used in Oracle/Postgres. Avoids ANSI anomalies but suffers from **Write Skew**.

1.11 Semi-structured Data (XML) [Lec 12]

1.11.1 Characteristics

- **Semi-structured:** Data is "self-describing" (tags + content). Structure (schema) is flexible or implied.
- **XML vs. HTML:** XML captures content/data; HTML captures presentation.
- **Well-Formed:** Follows basic syntax (single root, properly nested tags).
- **Valid:** Conforms to a DTD (Document Type Definition) or XML Schema.

1.11.2 XPath (Query Language)

Navigates the XML tree. * `/`: Separator (child). * `//`: Descendant (anywhere below). * `*`: Any element. `id(@ID)`: De-reference ID. * `[]`: Predicate/Condition. * **Examples:** * `/bibliography/book[author='Abiteboul']/@price`: Price of books by Abiteboul. * `//section/title`: Titles of all sections anywhere.

1.11.3 XQuery

XPath + SQL-like logic (FLWR expressions). * **F**or: Iterate over nodes. * **L**et: Bind variables. * **W**here: Filter conditions. * **R**eturn: Construct result. * *Note:* Can construct new XML structures in the `Return` clause.

1.11.4 Mapping XML to Relational

- **Node/Edge-based:** Store as a graph in relational tables.
  - Tables: `Element(eid, tag)`, `Attribute(eid, attrName, value)`, `ElementChild(eid, pos, child)`.
  - *Drawback:* Path expressions require many joins.

1.12 NoSQL (MongoDB) [Lec 13]

1.12.1 Data Model

- **JSON:** Lightweight, key-value pairs (Objects), ordered lists (Arrays).

- **BSON:** Binary JSON (storage format).
- **Hierarchy:** Database → Collections → Documents (JSON objects).
- **Schemaless:** No rigid schema required upfront.

1.12.2 Querying (find)

- Basic: `db.collection.find({ criteria }, { projection })`.
- **Dot Notation:** Access nested fields (e.g., `"roles.party": "Republican"`). Requires quotes.
- **Array Semantics:** `{ authors: "Widom" }` matches if "Widom" is *any* element in the authors array.
- **Logical Operators:** `\$or`, `\$and`, `\$not`, `\$nor`. Example: `{ $or: [ { a: 1 }, { b: 2 } ] }`.

1.12.3 Aggregation Pipeline

Sequence of stages transforming data. * `$match`: Filtering (like SQL WHERE). * `$project`: Renaming/Adding fields (like SQL SELECT). * `$unwind`: "Flattens" an array. Creates a new document for *each* element in the array. Crucial for joining/grouping on array elements. * `$group`: Aggregation (like SQL GROUP BY). Uses `_id` for the grouping key. Accumulators: `$sum`, `$push`. * `$lookup`: Performs a Left Outer Join with another collection.

1.13 Storage Basics [Lec 14]

1.13.1 Storage Hierarchy

- **Volatile:** Registers (Fastest) → Cache → Memory.
- **Non-Volatile:** SSD → Disk (HDD) → Tape (Slowest).
- **Gap:** I/O is the dominant cost factor in DBs. Accessing disk is ~10^6 times slower than memory.

1.13.2 Hard Drives (HDD) vs. SSD

- **HDD Access Time** = Seek Time (move arm) + Rotational Delay (spin disk) + Transfer Time.
  - *Sequential Access* is orders of magnitude faster than *Random Access*.
- **SSD:** No moving parts. Faster random access than HDD, but random writes are tricky (erase-before-write).

1.13.3 Buffer Management

- **Buffer Pool:** Memory reserved to cache disk blocks.
- **Strategy:** Minimize I/O by reading into buffer, modifying in memory (dirty pages), and flushing later.

1.13.4 Record & Page Layout

- **Records:** Fixed-length (fast offset calc) vs. Variable-length (requires offset table).
- **Page/Block Layouts:**
  - **NSM (Row Store):** Stores complete records sequentially. Good for writing and fetching full rows. Bad for scanning specific columns.
  - **PAX:** Mini-column store within a page. Keeps all fields of a record on the same page but groups them by column. Improves cache locality.
  - **Column Store:** Stores columns in separate files/pages. Great for analytics/aggregates. Compression friendly.

1.14 Indexing [Lec 15]

1.14.1 Concepts

- **Clustered Index:** Data records are sorted/ordered based on the index key. Only one per table.
- **Unclustered (Secondary) Index:** Index order $\neq$ Data order. Can have many.
- **Dense vs. Sparse:**
  - *Dense:* Index entry for every search key value.
  - *Sparse:* Index entry per block. Requires clustered data.

1.14.2 B+-Tree

- **Structure:** Balanced tree. Data pointers only in leaf nodes. Internal nodes guide search. Leaves linked for sequential scanning.
- **Properties:**
  - **Height Balanced:** All leaves at same depth.
  - **Fan-out:** Large (hundreds). Height is usually small ($log_{fanout}N$).
  - **Occupancy:** Nodes must be at least half full (except root).
- **Operations:**
  - *Lookup:* Traverse root to leaf. Cost $\approx$ height.
  - *Insert:* Find leaf. If full, **split** node and push middle key up to parent. Propagates up.
  - *Delete:* Find leaf. If $<$ half full, **steal** from sibling or **merge** (coalesce) with sibling. Propagates up.
- **Traversal Logic Example:**
  - Rule: "less than 7, go left, geq 7 go right."

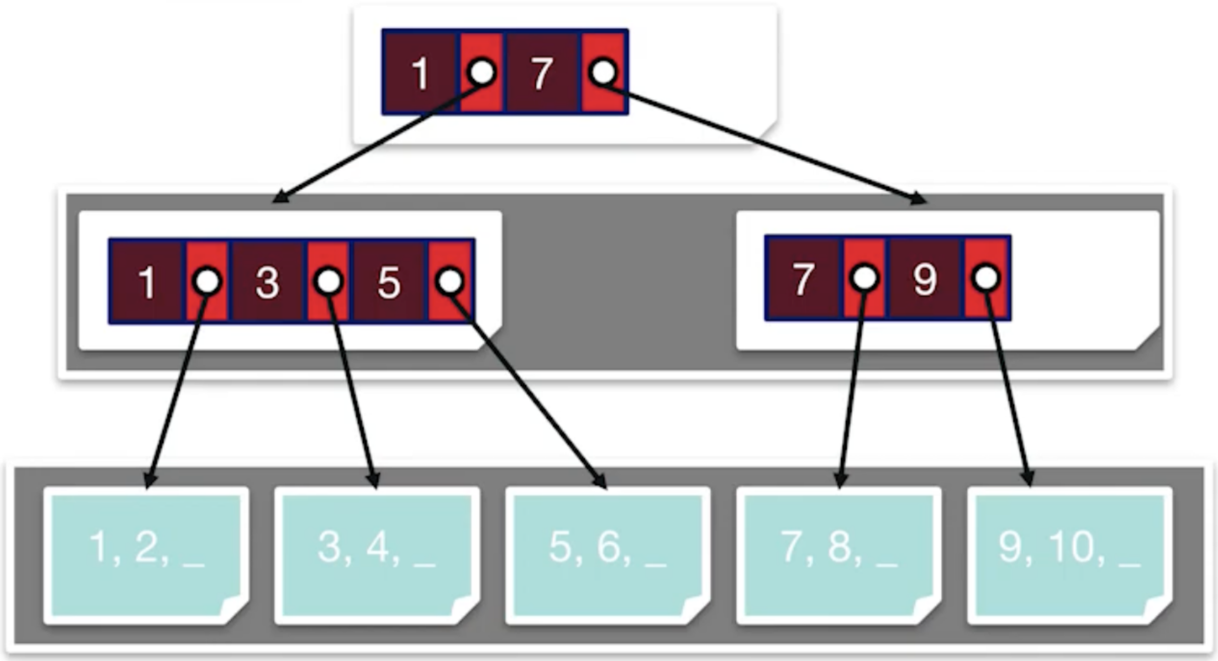

Figure 1: Tree Split Logic

  –

1.14.3 Other Indexes

- **ISAM:** Static structure. Does not rebalance (uses overflow chains). Good for static data.

- **Hash Index:** Good for equality (=), useless for range queries ($>, <$).

1.15 Query Processing Basics [Lec 16]

Cost Metrics: $B(R)$ = # blocks in R, M = Memory blocks available.

1.15.1 Selection (Scan)

- **Table Scan:** Read all blocks. Cost: $B(R)$.
- **Index Scan:**
  - *Clustered:* Cheap. Find start, scan sequentially.
  - *Unclustered:* Expensive if matching many records (1 random I/O per match). Scan wins if $> 5-10\%$ of tuples match.

1.15.2 Joins ($R \bowtie S$)

- **Nested Loop Join (NLJ):**
  - *Tuple-based:* For every row in R, scan S. Very slow.
  - *Block-based (BNLJ):* Read block of R, scan S. Cost: $B(R) + B(R) \cdot B(S)$.
  - *Chunk-based:* Load $M - 2$ blocks of R, scan S. Cost: $\approx B(R) \cdot B(S)/M$.
- **Sort-Merge Join (SMJ):**
  - Sort R and S, then merge.
  - Cost: Sort Cost + $B(R) + B(S)$.
  - Great if data is already sorted or need sorted output.
- **Hash Join (HJ):**
  - *Phase 1 (Partition):* Hash R and S into $M - 1$ buckets using hash function $h1$.
  - *Phase 2 (Probe):* Load bucket R_i into memory (build hash table $h2$), stream S_i and probe.
  - Cost: $3(B(R) + B(S))$ (Read/Write partition + Read probe).
  - Memory Requirement: $\sqrt{min(B(R), B(S))}$.
- **Index Nested Loop Join (INLJ):**
  - For every tuple in R, probe Index on S.
  - Cost: $B(R) + |R| \cdot$ (IndexLookupCost). Excellent if R is small.

1.15.3 Sorting (External Merge Sort)

- **Pass 0:** Create $\lceil B(R)/M \rceil$ sorted runs of size M.
- **Pass 1+:** Merge $M - 1$ runs at a time.
- Cost: $2 \cdot B(R) \cdot$ (number of passes). Complexity $O(B(R) \log_M B(R))$.

1.16 Query Optimization Basics [Lec 17]

1.16.1 Architecture

SQL → Parser → Validator → **Logical Plan** → **Optimizer** → **Physical Plan** → Executor.

1.16.2 Logical Optimization (Rewriting)

Using Relational Algebra equivalences to improve performance *heuristically*. * **Push down Selection (σ):** Filter data as early as possible. * **Push down Projection (π):** Remove unused columns early. * **Join Reordering:** $A \bowtie B \bowtie C$ is associative/commutative. Order matters for intermediate sizes.

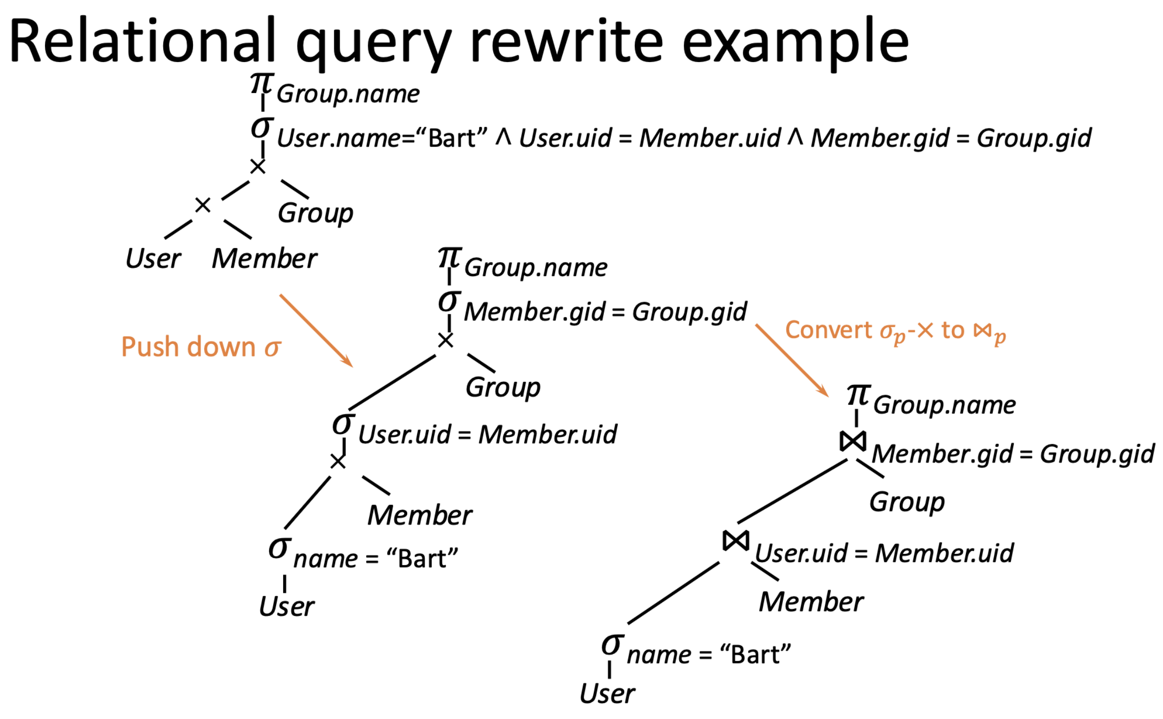

Figure 2: Relational Query Rewrite Example

1.16.2.1 Example: Relational Query Rewrite

1. **Initial Plan:** Cartesian products with a complex selection on top.
2. **Push Down Selection:** Move selection conditions down the tree to filter tuples early.
3. **Introduce Joins:** Convert Selection + Cartesian Product into Joins ($\sigma_p \times \rightarrow \bowtie_p$).

1.16.3 Cost Estimation (Cardinality)

Estimating result sizes ($|Q|$) to choose the best physical plan. * **Selectivity ($A = v$):** $1/|\pi_A R|$ (assuming uniform distribution). * **Selectivity ($A > v$):** $(High - v)/(High - Low)$. * **Join Size ($R \bowtie S$):** $(|R| \cdot |S|)/\max(|\pi_A R|, |\pi_A S|)$ (assuming containment/FK).

1.16.4 Plan Enumeration

- Search Space is huge (Factorial).
- Optimizer selects physical operators (e.g., Hash Join vs. Merge Join) based on estimated cost.

1.17 Concurrency Control (Transaction Processing) [Lec 18]

1.17.1 Schedules

- **Serial Schedule:** $T1$ then $T2$ (or vice versa). No interleaving. Always consistent.
- **Conflict:** Operations conflict if they access the same item and at least one is a **Write**. ($R - W, W - R, W - W$).
- **Conflict Serializable:** A schedule is conflict serializable if its **Precedence Graph** is acyclic.
  - *Precedence Graph:* Edge $T_i \rightarrow T_j$ if T_i conflicts with and precedes T_j.

1.17.2 Locking

- **2PL (Two-Phase Locking):**
  - *Phase 1 (Growing):* Acquire locks.
  - *Phase 2 (Shrinking):* Release locks.
  - **Guarantee:** Ensures Conflict Serializability.
- **Strict 2PL:**
  - Holds all Exclusive (X) locks until the transaction **Commits** or **Aborts**.
  - **Guarantee:** Ensures Serializability AND Recoverability (avoids Cascading Aborts).

1.17.3 Deadlocks

- Occurs when transactions wait for each other (Cycle in wait-for graph).
- Must abort one transaction to break the cycle.

2 Final Exam Coding Guide

This guide provides simple code snippets and structures for the languages covered in Lecture Notes 1 & 2.

2.1 SQL (Structured Query Language)

2.1.1 Basic Query (SFW)

- **Structure:** `SELECT <columns> FROM <table> WHERE <condition>`
- **Explanation:** Retrieves specific columns from a table where rows match a condition.
- **Example:**

```sql
SELECT name, age
FROM Students
WHERE age > 18;
```

2.1.2 JOIN (Inner)

- **Structure:** `SELECT ... FROM T1 JOIN T2 ON T1.col = T2.col`
- **Explanation:** Combines rows from two tables where the join condition is true.
- **Example:**

```sql
SELECT S.name, E.course
FROM Students S
JOIN Enrolled E ON S.sid = E.sid;
```

2.1.3 LEFT OUTER JOIN

- **Structure:** `SELECT ... FROM T1 LEFT JOIN T2 ON T1.col = T2.col`
- **Explanation:** Keeps all rows from the left table (T1). If no match in T2, T2 columns are NULL.
- **Example:**

```sql
-- List all students and their courses, even if they haven't enrolled in any.
SELECT S.name, E.course
FROM Students S
LEFT JOIN Enrolled E ON S.sid = E.sid;
```

2.1.4 GROUP BY

- **Structure:** `SELECT col, AGG(col) FROM table GROUP BY col`
- **Explanation:** Groups rows that have the same values in specified columns into summary rows.
- **Example:**

```sql
-- Count students per department
SELECT dept_name, COUNT(*)
FROM Students
GROUP BY dept_name;
```

2.1.5 Transaction Control

- **COMMIT**
  - **Structure:** `COMMIT;`
  - **Explanation:** Saves all changes made in the current transaction permanently to the database.
  - **Example:**
    ```
    BEGIN TRANSACTION;
    UPDATE Accounts SET balance = balance - 100 WHERE id = 1;
    UPDATE Accounts SET balance = balance + 100 WHERE id = 2;
    COMMIT; -- Changes are now permanent
    ```
- **ROLLBACK**
  - **Structure:** `ROLLBACK;`
  - **Explanation:** Undoes all changes made in the current transaction, reverting the database to its state before the transaction started.
  - **Example:**
    ```
    BEGIN TRANSACTION;
    INSERT INTO Log VALUES ('Error happened');
    -- Something went wrong
    ROLLBACK; -- The INSERT is undone
    ```

2.1.6 HAVING

- **Structure:** `SELECT ... GROUP BY col HAVING <condition>`
- **Explanation:** Filters **groups** created by `GROUP BY`. Unlike `WHERE` (which filters rows before grouping), `HAVING` filters after aggregation.
- **Example:**

```
-- Only show departments with more than 100 students
SELECT dept_name, COUNT(*)
FROM Students
GROUP BY dept_name
HAVING COUNT(*) > 100;
```

2.1.7 Subquery (IN)

- **Structure:** `WHERE col IN (SELECT col FROM ...)`
- **Explanation:** Checks if a value exists in a list returned by a subquery.
- **Example:**

```
-- Find students who are enrolled in 'CS101'
SELECT name
FROM Students
WHERE sid IN (SELECT sid FROM Enrolled WHERE course = 'CS101');
```

2.1.8 INSERT

- **Structure:** `INSERT INTO table (col1, col2) VALUES (val1, val2)`
- **Explanation:** Adds a new row to a table.
- **Example:**

```
INSERT INTO Students (sid, name) VALUES (123, 'Alice');
```

2.1.9 UPDATE

- **Structure:** `UPDATE table SET col = val WHERE condition`
- **Explanation:** Modifies existing rows. **Always use WHERE** unless you want to change every row.
- **Example:**

  ```
  UPDATE Students SET gpa = 4.0 WHERE sid = 123;
  ```

2.1.10 DELETE

- **Structure:** `DELETE FROM table WHERE condition`
- **Explanation:** Removes rows. **Always use WHERE** unless you want to wipe the table.
- **Example:**

  ```
  DELETE FROM Students WHERE sid = 123;
  ```

2.1.11 CREATE VIEW

- **Structure:** `CREATE VIEW ViewName AS SELECT ...`
- **Explanation:** Saves a query as a virtual table.
- **Example:**

  ```
  CREATE VIEW ActiveStudents AS
  SELECT * FROM Students WHERE status = 'Active';
  ```

2.1.12 TRIGGER

- **Structure:**

  ```
  CREATE TRIGGER name
  AFTER INSERT ON table
  FOR EACH ROW
  BEGIN ... END
  ```

- **Explanation:** Automatically executes code in response to database events (INSERT, UPDATE, DELETE).
- **Example:**

  ```
  -- Log new student insertions
  CREATE TRIGGER LogStudent
  AFTER INSERT ON Students
  FOR EACH ROW
  INSERT INTO Logs(message) VALUES ('New student: ' || NEW.name);
  ```

2.1.13 DISTINCT

- **Structure:** `SELECT DISTINCT col FROM table`
- **Explanation:** Removes duplicate rows from the result.
- **Example:** `SELECT DISTINCT dept_name FROM Students;`

2.1.14 ORDER BY

- **Structure:** `SELECT ... ORDER BY col [ASC|DESC]`
- **Explanation:** Sorts the result set.
- **Example:** `SELECT name FROM Students ORDER BY age DESC;`

2.1.15 LIMIT

- **Structure:** `SELECT ... LIMIT n`
- **Explanation:** Restricts the number of rows returned.
- **Example:** `SELECT * FROM Students LIMIT 5;`

2.1.16 AS (Alias)

- **Structure:** `SELECT col AS new_name FROM table`
- **Explanation:** Renames a column or table for the duration of the query.
- **Example:** `SELECT name AS student_name FROM Students;`

2.1.17 WITH (CTE)

- **Structure:** `WITH Name AS (SELECT ...) SELECT ... FROM Name`
- **Explanation:** Defines a temporary result set (Common Table Expression).
- **Example:**

```
WITH HighGPA AS (SELECT * FROM Students WHERE gpa > 3.5)
SELECT * FROM HighGPA;
```

2.1.18 RIGHT / FULL OUTER JOIN

- **Structure:** `... RIGHT JOIN ... / ... FULL JOIN ...`
- **Explanation:** `RIGHT` keeps all rows from right table. `FULL` keeps all rows from both.
- **Example:** `SELECT * FROM Students S FULL JOIN Enrolled E ON S.sid = E.sid;`

2.1.19 NATURAL JOIN

- **Structure:** `T1 NATURAL JOIN T2`
- **Explanation:** Joins tables on columns with the same name automatically.
- **Example:** `SELECT * FROM Students NATURAL JOIN Enrolled;`

2.1.20 Set Operations (UNION / INTERSECT / EXCEPT)

- **Structure:** `Query1 UNION Query2`
- **Explanation:** Combines results. `UNION` (OR), `INTERSECT` (AND), `EXCEPT` (Difference).
- **Example:**

```
SELECT name FROM Students
UNION
SELECT name FROM Professors;
```

2.1.21 LIKE

- **Structure:** `WHERE col LIKE pattern`
- **Explanation:** Pattern matching. `%` matches any sequence, `_` matches single char.
- **Example:** `SELECT * FROM Students WHERE name LIKE 'A%';` (Starts with A)

2.1.22 IS NULL

- **Structure:** `WHERE col IS NULL`
- **Explanation:** Checks for NULL values.
- **Example:** `SELECT * FROM Students WHERE gpa IS NULL;`

2.1.23 EXISTS

- **Structure:** `WHERE EXISTS (subquery)`
- **Explanation:** True if subquery returns any rows.
- **Example:**

```sql
SELECT * FROM Courses C
WHERE EXISTS (SELECT * FROM Enrolled E WHERE E.cid = C.cid);
```

2.1.24 ANY / ALL

- **Structure:** `WHERE col > ALL (subquery)`
- **Explanation:** Compares value to a set of values.
- **Example:** `SELECT * FROM Students WHERE gpa > ALL (SELECT gpa FROM Students WHERE dept = 'Art');`

2.1.25 Aggregates (SUM, AVG, MIN, MAX)

- **Structure:** `SELECT AVG(col) ...`
- **Explanation:** Calculates summary statistics.
- **Example:** `SELECT AVG(gpa), MAX(age) FROM Students;`

2.1.26 COALESCE

- **Structure:** `COALESCE(val1, val2, ...)`
- **Explanation:** Returns the first non-null value.
- **Example:** `SELECT COALESCE(phone, 'No Phone') FROM Students;`

2.1.27 NULLIF

- **Structure:** `NULLIF(val1, val2)`
- **Explanation:** Returns NULL if val1 equals val2, otherwise val1.
- **Example:** `SELECT NULLIF(gpa, 0) FROM Students;`

2.1.28 DDL (CREATE / DROP / ALTER TABLE)

- **Structure:** `CREATE TABLE ...`, `DROP TABLE ...`, `ALTER TABLE ...`
- **Explanation:** Defines or modifies table structure.
- **Example:**

```sql
CREATE TABLE Students (sid INT, name VARCHAR(50));
ALTER TABLE Students ADD email VARCHAR(100);
DROP TABLE Students;
```

2.1.29 Constraints (PK, FK, CHECK, UNIQUE, NOT NULL)

- **Structure:** Defined in `CREATE TABLE`.
- **Example:**

```sql
CREATE TABLE Enrolled (
    sid INT REFERENCES Students(sid), -- FK
    cid INT,
    grade CHAR(1) NOT NULL,
    CONSTRAINT pk_enrolled PRIMARY KEY (sid, cid),
        CONSTRAINT valid_grade CHECK (grade IN ('A', 'B', 'C'))
);
```

2.1.30 Isolation Levels

- **Structure:** `SET TRANSACTION ISOLATION LEVEL <LEVEL>`
- **Explanation:** Controls visibility of changes made by other transactions.
- **Example:** `SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;`

2.2 Relational Algebra (RA)

2.2.1 Selection (σ)

- **Structure:** $\sigma_{condition}(Relation)$
- **Explanation:** Filters **rows**. Equivalent to SQL `WHERE`.
- **Example:** $\sigma_{age>18}(Students)$

2.2.2 Projection (π)

- **Structure:** $\pi_{col1,col2}(Relation)$
- **Explanation:** Filters **columns**. Equivalent to SQL `SELECT`.
- **Example:** $\pi_{name}(Students)$

2.2.3 Natural Join ($\bowtie$)

- **Structure:** $R \bowtie S$
- **Explanation:** Joins on common attributes.
- **Example:** $Students \bowtie Enrolled$

2.2.4 Cartesian Product ($\times$)

- **Structure:** $R \times S$
- **Explanation:** Pairs every row of R with every row of S.
- **Example:** $Students \times Courses$

2.3 XML & XPath

2.3.1 XML Structure & DTD

- `<?xml ...?>`: Processing instruction (e.g., `<?xml version="1.0"?>`).
- `<!DOCTYPE ...>`: Defines the document type.
- `<!ELEMENT ...>`: Defines an element type.
- `<!ATTLIST ...>`: Defines an attribute list.
- `#PCDATA`: Parsed Character Data (text).
- `CDATA`: Character Data (unparsed text).
- `ID` / `IDREF`: Unique identifier and reference.
- `#REQUIRED` / `#IMPLIED`: Mandatory / Optional attribute.
- **Example:**

```
<!DOCTYPE note [
  <!ELEMENT note (to,from,heading,body)>
  <!ELEMENT to (#PCDATA)>
  <!ATTLIST note id ID #REQUIRED>
]>
```

2.3.2 XPath: Child (/)

- **Structure:** `/path/to/node`
- **Explanation:** Selects direct children.
- **Example:** `/bibliography/book` (Selects `book` elements directly under `bibliography`)

2.3.3 XPath: Descendant (//)

- **Structure:** `//node`
- **Explanation:** Selects descendants anywhere in the document.
- **Example:** `//author` (Selects `author` elements anywhere)

2.3.4 XPath: Attribute (@)

- **Structure:** `@attributeName`
- **Explanation:** Selects an attribute of an element.
- **Example:** `//book/@price` (Selects the `price` attribute of all books)

2.3.5 XPath: Wildcard (*)

- **Structure:** `*`
- **Explanation:** Matches any element node.
- **Example:** `/bibliography/*` (Selects all children of bibliography)

2.3.6 XPath: Current/Parent (. / ..)

- **Structure:** `.` (Current), `..` (Parent)
- **Explanation:** Relative path navigation.
- **Example:** `//book/..` (Selects the parent of book elements)

2.3.7 XPath: Operators (and, or, not, div, mod)

- **Structure:** `price > 10 and price < 50`
- **Explanation:** Boolean and arithmetic operations.
- **Example:** `//book[price > 10 and price < 50]`

2.3.8 XPath: Predicate ([])

- **Structure:** `node[condition]`
- **Explanation:** Filters nodes based on a condition.
- **Example:** `//book[price<50]` (Selects books with price less than 50)

2.3.9 XPath: Functions

- **contains(x, y)**
  - **Example:** `//book[contains(title, "Database")]` (Selects books with “Database” in title)
- **count(node-set)**
  - **Example:** `count(//book)` (Returns the number of book elements)
- **position()**
  - **Example:** `//book[position() = 1]` (Selects the first book)

- **last()**
  - **Example:** `//book[last()]` (Selects the last book)
- **name()**
  - **Example:** `//*[name()='book']` (Selects elements with tag name ‘book’)
- **id()**
  - **Example:** `id("b1")` (Selects element with unique ID “b1”)

2.3.10 XQuery (FLWR)

- **Structure:**

  ```
  for $x in path
  where condition
  return result
  ```

- **Explanation:** Iterates over nodes, filters them, and constructs a result.

- **Example:**

  ```
  for $b in doc("bib.xml")//book
  where $b/price > 50
  return <expensive>{ $b/title }</expensive>
  ```

2.3.11 XQuery: Let / Assignment (:=)

- **Structure:** `let $var := value`
- **Explanation:** Binds a variable to a value.
- **Example:** `let $x := 5 return $x * 2`

2.3.12 XQuery: Conditional (if-then-else)

- **Structure:** `if (condition) then ... else ...`
- **Explanation:** Conditional logic.
- **Example:** `if ($x > 10) then "High" else "Low"`

2.3.13 XQuery: Quantifiers (some/every)

- **Structure:** `some $x in sequence satisfies condition`
- **Explanation:** Checks if at least one (some) or all (every) items meet a condition.
- **Example:** `some $b in //book satisfies $b/price > 100`

2.3.14 XQuery: Functions (doc, distinct-values, avg, sort by)

- **Structure:** `doc(...)`, `distinct-values(...)`, `avg(...)`, `order by ...`

- **Explanation:** File access, aggregation, and sorting.

- **Example:**

  ```
  for $p in distinct-values(//book/price)
  order by $p
  return $p
  ```

2.4 MongoDB (NoSQL)

2.4.1 Find (Basic)

- **Structure:** `db.coll.find({ criteria })`

- **Explanation:** Retrieves documents matching the criteria.
- **Example:**

  ```
  // Find users with age 25
  db.users.find({ age: 25 })
  ```

2.4.2 Find (Nested Field)

- **Structure:** `db.coll.find({ "parent.child": value })`
- **Explanation:** Matches a field inside a nested object. **Quotes are required**.
- **Example:**

  ```
  // Find users whose address city is NY
  db.users.find({ "address.city": "NY" })
  ```

2.4.3 Find (Array)

- **Structure:** `db.coll.find({ arrayField: value })`
- **Explanation:** Matches if the value exists *anywhere* in the array.
- **Example:**

  ```
  // Find users who have "admin" in their roles array
  db.users.find({ roles: "admin" })
  ```

2.4.4 Aggregate: $match

- **Structure:** `{ $match: { criteria } }`
- **Explanation:** Filters documents (like SQL `WHERE`).
- **Example:** `{ $match: { status: "A" } }`

2.4.5 Aggregate: $group

- **Structure:** `{ $group: { _id: "$field", total: { $sum: 1 } } }`
- **Explanation:** Groups documents by `_id` and calculates aggregates.
- **Example:**

  ```
  // Count users per city
  db.users.aggregate([
      { $group: { _id: "$city", count: { $sum: 1 } } }
  ])
  ```

2.4.6 Aggregate: $unwind

- **Structure:** `{ $unwind: "$arrayField" }`
- **Explanation:** Deconstructs an array field, creating a new document for each element. Essential for grouping by array elements.
- **Example:**

  ```
  // If a user has 3 roles, this creates 3 documents, one for each role.
  db.users.aggregate([
      { $unwind: "$roles" }
  ])
  ```

2.4.7 Aggregate: $lookup (Join)

- **Structure:**

```
{ $lookup: {
    from: "otherColl",
    localField: "localCol",
    foreignField: "otherCol",
    as: "outputArray"
}}
```

- **Explanation:** Performs a Left Outer Join with another collection.
- **Example:**

```
// Join orders with inventory
db.orders.aggregate([
    { $lookup: {
        from: "inventory",
        localField: "item",
        foreignField: "sku",
        as: "inventory_docs"
    }}
])
```

2.4.8 CRUD: Insert / Update / Delete

- **Structure:** `db.coll.insertOne(...)`, `db.coll.updateOne(...)`, `db.coll.deleteOne(...)`
- **Explanation:** Basic data manipulation.
- **Example:**

```
db.users.insertOne({ name: "Bob", age: 30 });
db.users.updateOne({ name: "Bob" }, { $set: { age: 31 } });
```

2.4.9 Query Operators ($lt, $gte, $exists, $and, $or)

- **Structure:** `{ field: { \$op: value } }` or `{ \$op: [ { criteria1 }, { criteria2 } ] }`
- **Explanation:** Comparison and logical operators.
- **Example:**

```
// Implicit AND: Age >= 18 AND has email
db.users.find({
    age: { $gte: 18 },
    email: { $exists: true }
})

// Explicit OR: Age < 18 OR Age > 65
db.users.find({
    $or: [
        { age: { $lt: 18 } },
        { age: { $gt: 65 } }
    ]
})

// Explicit AND: (Price < 10) AND (Price > 5)
// Useful when same field is used multiple times or complex logic
```

```
db.products.find({
    $and: [
        { price: { $lt: 10 } },
        { price: { $gt: 5 } }
    ]
})
```

2.4.10 Query: Array ($elemMatch)

- **Structure:** `{ arrayField: { $elemMatch: { criteria } } }`
- **Explanation:** Matches documents where at least one array element matches *all* criteria.
- **Example:** `db.users.find({ scores: { $elemMatch: { $gt: 80, $lt: 90 } } })`

2.4.11 Aggregate: $project

- **Structure:** `{ $project: { field: 1, newField: "$otherField" } }`
- **Explanation:** Reshapes documents (selects/renames fields).
- **Example:** `{ $project: { name: 1, status: 1 } }`

2.4.12 Aggregate: $addFields

- **Structure:** `{ $addFields: { newField: expression } }`
- **Explanation:** Adds new fields to documents.
- **Example:** `{ $addFields: { totalScore: { $sum: "$scores" } } }`

2.4.13 Aggregate: $sort

- **Structure:** `{ $sort: { field: 1 | -1 } }`
- **Explanation:** Sorts documents (1 = Ascending, -1 = Descending).
- **Example:** `{ $sort: { age: -1 } }`

2.4.14 Aggregate: $replaceRoot

- **Structure:** `{ $replaceRoot: { newRoot: document } }`
- **Explanation:** Replaces the input document with the specified document.
- **Example:** `{ $replaceRoot: { newRoot: "$address" } }`

2.4.15 Aggregate: Array Operators ($map, $filter)

- **Structure:** `$map`, `$filter` inside `$project` or `$addFields`.
- **Explanation:** Transforms or filters arrays.
- **Example:**

```
// $map: Adds 10 to each score in the 'scores' array
{ $project: {
    adjustedScores: {
        $map: {
            input: "$scores",       // $scores is the array field from the document
            as: "score",            // "score" is a temp variable for the current item
            in: { $add: ["$$score", 10] } // Use $$ to reference the temp variable
        }
    }
}}

// $filter: Keeps only scores greater than 80
```

```
{ $project: {
    highScores: {
        $filter: {
            input: "$scores",       // The array to filter
            as: "score",            // Temp variable name
            cond: { $gt: ["$$score", 80] } // Condition using the temp variable ($$)
        }
    }
}}
```

2.4.16 Aggregate: Accumulators ($sum, $push, $last)

- **Structure:** Used in `$group`.
- **Explanation:** Aggregates values across a group.
- **Example:**

```
{ $group: {
    _id: "$city",
    totalPop: { $sum: "$pop" },
    allNames: { $push: "$name" }
}}
```

3 Final Exam Practice Questions

3.1 Problem (20 points)

3.1.1 (T/F) In a program, multiple statements can be grouped together as a transaction.

Answer: True

- **Explanation:** A transaction is a logical unit of work that contains one or more SQL statements. Grouping them ensures they are treated as a single operation.
- **Key Concept:** Transaction Definition.

3.1.2 (T/F) The actions in a transaction are atomic and either they are all performed or none of them are performed.

Answer: True

- **Explanation:** This is the definition of **Atomicity** in the ACID properties. It ensures that if any part of the transaction fails, the entire transaction is rolled back.
- **Key Concept:** ACID Properties (Atomicity).

3.1.3 (T/F) Since setting up a database connection is expensive, libraries like SQLAlchemy/ODBC often cache connections for future use.

Answer: True

- **Explanation:** Establishing a database connection involves network overhead and authentication, which is resource-intensive. **Connection pooling** allows applications to reuse existing connections, improving performance.
- **Key Concept:** Connection Pooling.

3.1.4 (T/F) We cannot add constraints to the semi-structured data model.

Answer: False

- **Explanation:** Semi-structured data models (like XML or JSON) *can* have constraints. For example, XML has **DTD (Document Type Definition)** and **XML Schema (XSD)** to define structure and data types. JSON has **JSON Schema**.
- **Counter Example:** An XML Schema requiring that every `<book>` element must have an `isbn` attribute.

3.1.5 (T/F) Sequential IO is much slower than random IO.

Answer: False

- **Explanation:** Sequential IO is significantly **faster** than random IO. Random IO requires the disk head to seek to different locations (seek time) and wait for the disk to rotate (rotational latency), whereas sequential IO reads contiguous blocks.
- **Key Concept:** Disk I/O Characteristics.

3.1.6 (T/F) In a B+ tree of maximum fanout 100, it is possible for an internal node (i.e., one that is neither the root nor a leaf) to have exactly 32 children.

Answer: False

- **Explanation:**
  - Let n be the maximum fanout (number of pointers). Here, $n = 100$.
  - The minimum number of children for an internal node (except the root) is $\lceil n/2 \rceil$.
  - $\lceil 100/2 \rceil = 50$.

    – Since 32 is less than 50, it is not possible for a valid B+ tree internal node to have only 32 children.
- **Formula:** Min children = $\lceil \text{Max Fanout}/2 \rceil$.

3.1.7 (T/F) For a range query, it is always better to use an index-based plan than a scan-based plan.

Answer: False

- **Explanation:** If the range query selects a large portion of the table (e.g., > 10-20% of rows), a **scan-based plan** (sequential scan) is often faster. An unclustered index would require random I/O for each matching record, which is slower than reading the whole table sequentially.
- **Key Concept:** Index Selectivity / Scan vs. Seek.

3.1.8 (T/F) consider a table $R(A, B)$ with 5000 rows, where B is a unique key but not the primary key. Suppose that each B+-tree index block can hold up to 9 keys and 10 pointers. The minimum number of levels needed for a B+-tree index on $R(B)$ is 4 (the root counts as a level).

Answer: True

- **Step-by-step Solution:**
  1. **Leaf Level:** We need to index 5000 entries.
     – Max keys per block = 9.
     – Number of leaf nodes = $\lceil 5000/9 \rceil = 556$.
  2. **Level 1 (above leaves):** We need to point to 556 leaf nodes.
     – Max pointers per internal node = 10.
     – Number of nodes = $\lceil 556/10 \rceil = 56$.
  3. **Level 2:** We need to point to 56 nodes.
     – Number of nodes = $\lceil 56/10 \rceil = 6$.
  4. **Level 3 (Root):** We need to point to 6 nodes.
     – Number of nodes = $\lceil 6/10 \rceil = 1$.
  5. **Total Levels:** Leaf, Level 1, Level 2, Root. Total = 4 levels.

3.1.9 (T/F) Consider the following two XPath queries:

- `//A[B/C = "foo" and B/D = "bar"]`
- `//A[B[C = "foo" and D = "bar"]]`

Every element returned by the first query will be returned by the second. Answer: False

- **Explanation:**
  – **Query 1:** Checks if `A` has *some* `B` child with `C="foo"` AND *some* `B` child with `D="bar"`. These can be different `B` nodes.
  – **Query 2:** Checks if `A` has *a specific* `B` child that has BOTH `C="foo"` and `D="bar"`.
- **Counter Example:**

```
<A>
  <B><C>foo</C></B>
  <B><D>bar</D></B>
</A>
```

  This matches Query 1 but not Query 2.

3.1.10 (T/F) consider the XPath queries above, every element returned by the second query will also be returned by the first.

Answer: True

- **Explanation:**
  - If an element matches Query 2, there exists a B node that satisfies both conditions (C = "foo" and D = "bar").
  - Since this B node exists, the condition for Query 1 is also satisfied (there is a B with C = "foo" and there is a B with D = "bar"—it happens to be the same one).

3.2 Problem (Semi-structured data) (17 points)

Consider a course registration XML document:

```
<Registration>
  <Course capacity='140'>
    <Number>CMPT 354</Number>
    <Student><Name>Abby</Name><Grade>98</Grade></Student>
    <Student><Name>Burnie</Name><Grade>75</Grade></Student>
  </Course>
  <Course capacity='50'>
    <Number>CMPT 459</Number>
    <Student><Name>Colin</Name><Grade>90</Grade></Student>
    <Student><Name>Demi</Name><Grade>100</Grade></Student>
  </Course>
</Registration>
```

3.2.1 2a (4 points) Write an XPath expression that are equivalent to the XQuery below.

```
for $c in /Registration/Course
return
  if (exists($c/Student[Grade >= 90 and Grade < 95])) then $c/Number
```

Answer:

```
/Registration/Course[Student[Grade >= 90 and Grade < 95]]/Number
```

3.2.2 2b (4 points) Describe what this XPath returns in English

```
/Registration/Course[Number[contains(., 'CMPT 4')] and count(Student[Grade < 80]) = 0]
```

Answer: It returns the `Course` elements for 400-level CMPT courses (courses where the number contains 'CMPT 4') where **all** students have a grade of 80 or higher (count of students with grade $<$ 80 is 0).

3.2.3 2c (9 points) Consider the MongoDB database storing the same course registration info as the XML document in 2a.

```
[
  {'capacity':140,
   'number': 'CMPT 354',
   'students': [
     {'name': 'Abby', 'grade': 98},
     {'name': 'Burnie', 'grade': 75}
   ]},
  {'capacity':50,
   'number': 'CMPT 459',
   'students': [
     {'name': 'Colin', 'grade': 90},
     {'name': 'Demi', 'grade': 100}
   ]},
    ...
]
```

Complete the MongoDB query below to retrieve the students whose grade is above the average grade for each course. Each output object has three fields, course number, student name, and student grade. Only need to write down the answers in each slot [Fill in].

```
db.courses.aggregate([
  { [Fill in] },
  { // Group by course and calculate the average grade
    $group: {
      [Fill in],
      averageGrade: { $avg: "$students.grade" },
      students: { [Fill in] }
    }
  },
  { [Fill in] },
  { // compare student grade with the course average, $gt is >
    [Fill in]: {
      $expr: { $gt: ["$students.grade", "$averageGrade"] }
    }
  },
  {
    [Fill in]: {
      _id: 0,
      courseNumber: "$_id",
      studentName: "$students.name",
      studentGrade: "$students.grade"
    }
  }
]);
```

Answer: 1. `$unwind: "$students"` 2. `_id: "$number"` 3. `$push: "$students"` 4. `$unwind: "$students"` 5. `$match` 6. `$project`

3.3 Problem 3 (Transactions)

Consider the following schedule involving three transactions T1, T2, T3 (r1(A) means T1 reads A etc.)
`r1(A), w1(A), r3(B), w2(A), w3(B), w1(B)`

3.3.1 3a (3 points). Draw the precedence graph.

Answer: any graph representing T3 -> T1 -> T2

- **Explanation:**
  - **Identify Conflicts:** We look for operations on the same data item by different transactions where at least one is a write.
    * `w1(A)` and `w2(A)`: **T1 -> T2** (Write-Write conflict on A).
    * `r3(B)` and `w1(B)`: **T3 -> T1** (Read-Write conflict on B).
    * `w3(B)` and `w1(B)`: **T3 -> T1** (Write-Write conflict on B).
  - **Graph Construction:**
    * Edge from T3 to T1.
    * Edge from T1 to T2.
  - **Result:** The graph is a chain: $T3 \to T1 \to T2$.

3.3.2 3b. (T/F, 2 points) The schedule is conflict serializable.

Answer: True

- **Explanation:**
  - A schedule is conflict serializable if and only if its precedence graph is **acyclic** (has no cycles).
  - Our graph $T3 \rightarrow T1 \rightarrow T2$ has no cycles.
  - Therefore, it is conflict serializable.
  - **Equivalent Serial Schedule:** $T3, T1, T2$.

3.3.3 3c: (T/F, 2 points) The schedule is possible under two-phase locking (2PL).

Answer: False

- **Explanation:**
  - **2PL Rule:** A transaction cannot acquire a new lock after it has released any lock (Growing Phase -> Shrinking Phase).
  - **Trace:**
    1. T1 needs lock on A for `w1(A)`.
    2. T2 needs lock on A for `w2(A)`. For T2 to proceed, T1 must release its lock on A.
    3. Once T1 releases lock on A, it enters the **shrinking phase**.
    4. Later in the schedule, T1 performs `w1(B)`. This requires T1 to acquire a lock on B.
    5. **Violation:** T1 cannot acquire the lock on B because it is already in the shrinking phase (it released A).
  - **Alternative Scenario:** If T1 held onto the lock on A until the end (to satisfy 2PL), T2 would be blocked at `w2(A)` until T1 finished. But in the given schedule, `w2(A)` happens *before* T1 finishes (before `w1(B)`). Thus, the specific interleaving in the schedule is impossible under 2PL.

3.3.4 3d: (T/F, 2 points) The schedule avoids cascading rollback.

Answer: True

- **Explanation:**
  - **Cascading Rollback Definition:** Occurs if a transaction reads uncommitted data (dirty read) from another transaction that later aborts.
  - **Analysis:**
    * Does any transaction read data written by an *active* (uncommitted) transaction?
    * `r1(A)`: Reads initial value.
    * `r3(B)`: Reads initial value.
    * There are no other reads. Specifically, no one reads the values written by `w1(A)`, `w2(A)`, or `w3(B)`.
  - Since there are **no dirty reads** (reading data written by another uncommitted transaction), the schedule avoids cascading rollback (ACR).

3.3.5 3e: (T/F, 2 points) The schedule has two equivalent serial schedules.

Answer: False

- **Explanation:**
  - The precedence graph $T3 \rightarrow T1 \rightarrow T2$ imposes a strict total order.
  - There is only one valid topological sort of this graph: $T3, T1, T2$.
  - Therefore, there is only **one** equivalent serial schedule.

3.4 Problem 4 (Indexing and Query Optimization)

Given three relations R1(A,B,C), R2(B,C,D) and R3(D,E), consider following SQL Query:

```sql
SELECT A, E
FROM R1 NATURAL JOIN R2 NATURAL JOIN R3
WHERE R2.B > 100;
```

3.4.1 4a (5 points): Draw the logical plan of this query and try pushing down selections and projections as much as possible.

Answer:

- **Logical Plan Tree:**

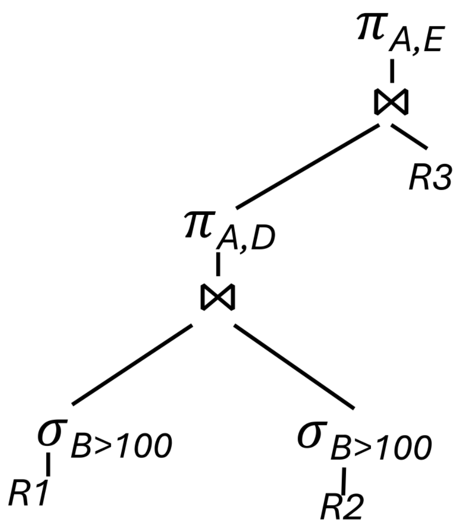

Figure 3: Logical Plan Tree

- **Explanation:**
  - **Pushing Selections:** The condition `R2.B > 100` applies to R2. Since R1 is joined on B, the condition `B > 100` also applies to R1. Pushing this filter down to both R1 and R2 reduces the input size for the join significantly.
  - **Pushing Projections:**
    - We only need A and E for the final output.
    - For the join between (R1, R2) and R3, we need D (join key) and A (for output). So we project $\pi_{A,D}$ after the first join.
    - For the join between R1 and R2, we need A (output), D (for next join), and B, C (join keys). Wait, B and C are consumed by the join. We don't need them after the join? The join is natural, so B and C are common. But we don't need B or C for the *next* join (on D) or the *output* (A, E). So we can drop B and C immediately after the first join.
    - Thus, the projection $\pi_{A,D}$ above the first join is correct.

3.4.2 4b, 4c, 4d (2 points each): Consider we have the following index:

```sql
CREATE INDEX idx on R2 (B, C);
```

For each of the following SQL queries, decide whether the index idx will (A) may speed it up, or (B) will not affect its performance, or (C) will slow it down.

3.4.2.1 4b: `SELECT * FROM R2 WHERE D = '001'` Answer: B (will not affect performance)

- **Explanation:** The index is on `(B, C)`. The query filters on `D`. Since `B` is the leading column of the index and is not involved in the predicate, the index cannot be used for a seek. A full table scan is required regardless.

3.4.2.2 4c: `INSERT INTO R2 VALUES (200, 'Bart', '002')` Answer: C (will slow it down)

- **Explanation:** Every `INSERT` statement requires updating the data file *and* all active indexes. The database must insert the new entry `(200, 'Bart')` into the B-tree index `idx`, which adds overhead compared to inserting into the heap file alone.

3.4.2.3 4d: `SELECT * FROM R2 WHERE C LIKE 'Bart %'` Answer: A (may speed it up)

- **Explanation:**
  - Typically, a B-tree index on `(B, C)` cannot be used efficiently if the leading column `B` is not specified.
  - **Exception (Skip Scan):** If column `B` has **very low cardinality** (few distinct values), the query optimizer might use a "Skip Scan". It effectively probes the index for `(DistinctValue1, 'Bart %')`, `(DistinctValue2, 'Bart %')`, etc. This can be faster than a full table scan.

3.4.3 4e (8 points): Cost Estimation

Assume no indexes, 3 memory blocks. Join algorithm: Block Nested Loops Join. Query:

```sql
SELECT *
FROM R2, R3
WHERE R2.D = R3.D AND R3.E = 777;
```

Stats: * R2: 50,000 tuples, 5,000 blocks. * R3: 10,000 tuples, 500 blocks. * R3.E range [701, 900] (200 values), uniform. * R2.D FK to R3.D.

Answer:

- **Plan:**
  1. **Selection on R3:** $\sigma_{E=777}(R3)$
  2. **Join:** Result of (1) $\bowtie_D$ R2
- **Estimations:**
  - **Selection Output Size:**
    * Selectivity of `E=777` is 1/200.
    * Tuples $= 10,000 \times (1/200) = 50$ tuples.
    * Blocks: 50 tuples fit in $\lceil 50/(10000/500) \rceil = \lceil 50/20 \rceil = 3$ blocks.
  - **Join Output Size:**
    * We have 50 tuples from R3.
    * Join on D (FK in R2 -> PK in R3).
    * Each R3 tuple matches roughly $N(R2)/N(R3) = 50,000/10,000 = 5$ tuples in R2.
    * Total Result Tuples $= 50 \times 5 = 250$ tuples.
- **Cost Calculation:**
  - **Selection Cost:** Scan R3 = **500 I/Os**.
  - **Join Cost (Block Nested Loop):**
    * **Outer Relation:** The filtered R3 (50 tuples, 3 blocks).
    * **Inner Relation:** R2 (5000 blocks).
    * **Memory:** 3 blocks.
      · 1 block for Input (Outer).
      · 1 block for Input (Inner).
      · 1 block for Output.
      · (Or use 1 block for Outer, scan Inner).
    * **Passes:** Since Outer (3 blocks) doesn't fit in 1 block (M-2=1), we need multiple passes?
    * Actually, with M=3, we can hold $M - 2 = 1$ block of outer.

        - Number of chunks for outer = $\lceil 3/1 \rceil = 3$ chunks.
        - For each chunk, we scan Inner (R2).
        - Cost = $B(Outer) + (\#\text{ chunks}) \times B(Inner)$
        - Cost = $3 + 3 \times 5000 = 15,003$ I/Os.
    - **Total Cost:** $500(\text{Select}) + 15,003(\text{Join}) = 15,503$. (The question asks for cost of *each* operator, so 500 and 15003).

3.4.4 4f (3 points): Sort-Merge Join Feasibility

- R2: 10,000 blocks.
- Memory: 1002 blocks.
- Query: `R2 JOIN R3`.
- Find max B(R3) for 2-pass Sort-Merge Join.

Answer: B (10^6)

- **Explanation:**
    - **Phase 1 (Sort):** Create sorted runs.
        - Runs for R2: $\lceil 10,000/1002 \rceil = 10$ runs.
        - Runs for R3: $\lceil B(R3)/1002 \rceil$ runs.
    - **Phase 2 (Merge):** Merge all runs in one pass.
        - We need 1 buffer for each run + 1 for output.
        - Total runs $\leq M - 1$ (or roughly M).
        - $10 + \text{Runs}(R3) \leq 1002$.
        - $\text{Runs}(R3) \leq 992$.
    - **Max Size of R3:**
        - $B(R3) \approx \text{Runs}(R3) \times M$
        - $B(R3) \approx 992 \times 1002 \approx 994,000 \approx 10^6$.

3.4.5 4g (3 points): Hash Join Feasibility

- Same setup, but 2-pass Hash Join.

Answer: D (10^{10})

- **Explanation:**
    - **Requirement:** For a 2-pass Hash Join (Grace Hash Join), we partition both relations into k buckets (where $k \approx M$). The condition is that for each bucket pair, the partition from the *smaller* relation must fit in memory.
    - **Check R2:**
        - $B(R2) = 10,000$.
        - $M = 1002$.
        - Number of partitions $k \approx 1000$.
        - Size of one R2 partition $\approx 10,000/1000 = 10$ blocks.
    - **Feasibility:**
        - Since 10 blocks $\ll$ 1002 blocks (Memory), the partitions of R2 easily fit in memory during the probe phase.
        - Since R2 fits the requirement, R3 can be arbitrarily large (we just stream R3 partitions through).
        - Therefore, R3 can be 10^{10} (or practically any size).

3.5 Problem 5 (RA/SQL database)

Consider a database storing information about coding competitions on database queries hosted by SFU in the following schema: * `Student(sid, sname, dept)` * `Contest(cid, host_dept, starttime, endtime)` * `Participate(sid, cid, num_questions_solved, rank)`

Note that: * Students have unique `sid`. * Contests have unique `cid`. `host_dept` is the name of the department that provide questions. `starttime` should be no later than `endtime`. * Participants solve some number of questions (`num_questions_solved` is a non-negative integer). `rank` is optional, and the value of `rank` (if present) is an integer >= 1. Also, `sid` and `cid` are foreign keys referring to `Student` and `Contest`, respectively.

3.5.1 5a (5 points) Write an SQL statement to create the Participate table. Include all constraints you find necessary.

Answer:

```
CREATE TABLE Participate (
    sid INTEGER NOT NULL REFERENCES Student(sid),
    cid INTEGER NOT NULL REFERENCES Contest(cid),
    num_questions_solved INTEGER NOT NULL,
    rank INTEGER,
    PRIMARY KEY (sid, cid),
    CHECK (num_questions_solved >= 0 AND (rank IS NULL OR rank > 0))
);
```

- **Explanation:**
  - **Foreign Keys:** `sid` and `cid` link to the `Student` and `Contest` tables.
  - **Primary Key:** The combination `(sid, cid)` identifies a unique participation record (a student in a specific contest).
  - **Check Constraints:**
    * `num_questions_solved >= 0`: Questions solved cannot be negative.
    * `rank IS NULL OR rank > 0`: Rank must be positive (1, 2, 3...) if it exists.

3.5.2 5b (10 points) Write an SQL query to find all contests (cid) where all top-3 participants are from the same department. You can assume that there are no more than three top-3 participants (no ties).

Answer:

```
SELECT C.cid
FROM Student S, Participate P, Contest C
WHERE S.sid = P.sid AND P.cid = C.cid AND P.rank IN (1, 2, 3)
GROUP BY C.cid
HAVING COUNT(DISTINCT S.dept) = 1;
```

- **Explanation:**
  - **Join:** We join `Student`, `Participate`, and `Contest` to link students to the contests they participated in.
  - **Filter:** `WHERE P.rank IN (1, 2, 3)` filters for only the top 3 participants.
  - **Group:** `GROUP BY C.cid` groups the results by contest.
  - **Having:** `HAVING COUNT(DISTINCT S.dept) = 1` ensures that for a given contest, the set of departments of the top 3 students contains exactly one unique department. This means they are all from the same department.

3.5.3 5c (4 points) Describe succinctly what the following relational algebra query returns in English.

$\pi_{sname,dept}(Student \bowtie \rho_{p1}(\sigma_{rank=1}Participate) \bowtie_{p1.sid=p2.sid \wedge p1.num\_problem\_solved<p2.num\_problem\_solved} \rho_{p2}(\sigma_{rank>1}Partic$

Answer: Find the name and department of students who won (ranked 1st in) one contest but solved fewer questions than in another contest they did not win (rank > 1).

- **Explanation:**
  - $\rho_{p1}(\sigma_{rank=1} Participate)$: Selects participation records where the student won (rank=1). Renames to `p1`.
  - $\rho_{p2}(\sigma_{rank>1} Participate)$: Selects participation records where the student did *not* win (rank > 1). Renames to `p2`.
  - **Join Condition:** `p1.sid = p2.sid` (same student) AND `p1.num_problem_solved < p2.num_problem_solved` (solved fewer questions in the winning contest `p1` than in the losing contest `p2`).
  - **Projection:** Returns the student's name and department.

3.5.4 5d (5 points) Read the trigger declaration below and state what the constraint this trigger enforces.

```
CREATE FUNCTION TF_check() RETURNS TRIGGER AS $$
BEGIN
  IF TG_TABLE_NAME = 'participate' THEN
    IF EXISTS (SELECT 1 FROM Participate as p
      WHERE p.sid <> NEW.sid AND p.cid = NEW.cid
      AND ( (p.num_questions_solved < NEW.num_questions_solved
             AND p.rank < NEW.rank) OR
            (p.num_questions_solved > NEW.num_questions_solved
             AND p.rank > NEW.rank) ) ) THEN
      RAISE EXCEPTION 'constraint violation';
    END IF;
  END IF;
  RETURN NEW;
END;
$$ LANGUAGE plpgsql;
```

Answer: In the same contest, there are no student pairs $< s1, s2 >$ where s1's rank is better than s2 but s1 solves fewer questions than s2 does.

- **Explanation:**
  - The trigger fires to check consistency between `num_questions_solved` and `rank`.
  - **Logic:** It raises an exception if it finds another participant `p` in the same contest (`p.cid = NEW.cid`) where:
    * `p` solved fewer questions but has a better (lower) rank.
    * OR `p` solved more questions but has a worse (higher) rank.
  - **Goal:** Ensures that rank correlates with the number of questions solved (more questions $\rightarrow$ better rank).

3.5.5 5e (3 points) Assume there is a trigger on Contest and a trigger on Participate (not the one in 5d), can an UPDATE statement on Contest cause the trigger on Participate to fire? (Y or N)

Answer: Y

- **Explanation:**
  - **Cascading Actions:** If the trigger on `Contest` performs an `UPDATE` or `INSERT` on the `Participate` table as part of its action, then the trigger on `Participate` will fire.
  - **Example:** A trigger on `Contest` might update the `Participate` table to invalidate ranks if a contest's time is changed, which would then fire the `Participate` trigger.