Contents

1 Database Systems I: Comprehensive Final Exam Study Guide

1.1 Table of Contents

1.2 1. 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 2. 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).

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.
2. **Projection (π_L):** Filters **columns** based on list L. Removes duplicates. Monotone .
3. **Cross Product ($\times$):** Pairs every row of R with every row of S. Monotone.
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 3. 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 4. E/R to Relational Translation [Lec 5]

1.5.1 Basic Translation

1. **Entity Set → Table:** Attributes become columns. Key becomes Primary Key .
2. **Relationship → 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 5. Relational Design Theory: Normalization [Lec 6]

Goal: Systematically remove redundancy to prevent update, insertion, and deletion anomalies .

1.6.1 Functional Dependencies (FDs)

- $X \to Y$: If two tuples agree on attributes X, they must agree on attributes Y.
- **Key:** K is a key if $K \to$ 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 $X \to Y$, X is a **superkey**.
  - *Decomposition:* If $X \to Y$ 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):** $X \to\to Y$. Given X, Y is independent of the rest of the attributes.
- **4NF:** A relation is in 4NF if for every non-trivial MVD $X \to\to Y$, X is a **superkey**. 4NF implies BCNF.

1.7 6. 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 7. 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 8. 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 9. SQL Transactions & ACID [Lec 11]

1.10.1 ACID Properties

- **Atomicity:** All-or-nothing. Never "half-done". Handled by **Logging (Undo)**.
- **Consistency:** DB moves from one valid state to another (constraints satisfied).
- **Isolation:** Transactions behave as if executed alone. Handled by **Locking/MVCC**.
- **Durability:** Committed changes survive crashes. Handled by **Logging (Redo)**.

1.10.2 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 10. 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 11. 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.

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 12. 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 13. 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.

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 14. Query Processing Basics [Lec 16]

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

1.15.1 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 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 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 15. Query Optimization Basics [Lec 17]

1.16.1 Architecture

SQL $\rightarrow$ Parser $\rightarrow$ Validator $\rightarrow$ **Logical Plan** $\rightarrow$ **Optimizer** $\rightarrow$ **Physical Plan** $\rightarrow$ 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.

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 16. 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 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 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:**

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

2.1.6 Subquery (IN)

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

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

2.1.7 INSERT

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

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

2.1.8 UPDATE

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

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

2.1.9 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;
```

2.1.10 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';
```

2.1.11 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);
```

2.2 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 3. XML & XPath

2.3.1 XPath: Child (/)

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

2.3.2 XPath: Descendant (//)

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

2.3.3 XPath: Attribute (@)

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

2.3.4 XPath: Predicate ([])

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

2.3.5 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>
```

2.4 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"
    }}
])
```