SELECT: Lists columns or expressions (e.g., arithmetic operations, functions, aliases).

FROM: Specifies one or more tables; use table aliases for clarity.

WHERE: Filters rows using conditions (comparisons, AND/OR/NOT, subqueries).

Evaluation Strategy: Compute the Cartesian product of tables. Apply the WHERE clause to filter tuples. Project the desired columns. Eliminate duplicates (if DISTINCT is used).

Aliases: Use `AS` (or a space) to rename columns/tables for better readability.

Expressions: Direct arithmetic or string operations (e.g., `age/7.0`).

LIKE Operator: `%` matches zero or more characters; `_` matches exactly one. *Example:* `WHERE name LIKE 'A%n'` finds names starting with A and ending with n.

Set Operators: UNION: Combines results and removes duplicates. **UNION ALL:** Combines results with duplicates. **INTERSECT:** Returns common rows between queries. **EXCEPT/MINUS:** Returns rows in the first query not present in the second.

Subqueries: Simple (Non-correlated): Runs once; its result is used by the outer query. **Correlated:** References columns from the outer query; evaluated per row. Used with `IN`, `NOT IN`, `EXISTS`, `ANY`, and `ALL`.

GROUP BY: Divides rows into groups based on one or more columns.

Aggregate Functions: COUNT, SUM, AVG, MAX, MIN. *Note:* COUNT(column) ignores NULLs; COUNT(*) counts all rows.

HAVING Clause: Filters groups after aggregation (e.g., `HAVING COUNT(*) >= 2`).

ORDER BY: Sorts results by one or more columns (ASC/DESC).

LIMIT/OFFSET: LIMIT restricts the number of rows; OFFSET skips rows. Useful for pagination and top-n queries.

INNER JOIN: Returns rows with matching values in both tables.

LEFT/RIGHT OUTER JOIN: LEFT OUTER JOIN: All rows from the left table; unmatched right-side rows are NULL. **RIGHT OUTER JOIN:** All rows from the right table; unmatched left-side rows are NULL. **FULL OUTER JOIN:** Returns rows with a match in either table (if supported).

Correlated Subqueries: Use outer query columns; evaluated for each row.

ANY / ALL Operators: Compare a value against a set from a subquery. *Example:* `WHERE rating > ANY (SELECT rating FROM Sailors WHERE sname = 'Horatio')`.

Division Queries: Used for "for all" queries (e.g., find sailors who reserved all boats) via NOT EXISTS/EXCEPT logic.

Handling NULLs: Aggregates like AVG, SUM ignore NULLs; COUNT(column) ignores NULLs, but COUNT(*) does not.

CRUD Essential: INSERT: Adds new rows; missing values become NULL or use default values. **UPDATE:** Modifies existing rows; always include a WHERE clause. **DELETE:** Removes rows; use cautiously to avoid unintentional data loss.

Triggers: Definition: Procedures that automatically execute on INSERT, UPDATE, or DELETE. **Structure: Event:** (e.g., AFTER INSERT) **Condition:** Optional check to determine if trigger should run. **Action:** SQL code executed when triggered. **Usage:** Enforce business rules, maintain audit logs, or cascade changes.

Stored Procedures: Purpose: Encapsulate business logic and complex operations. **Call Syntax:** `CALL procedure_name(parameters)` **Parameters:** Modes: IN, OUT, INOUT.

Stored Functions: Definition: Similar to procedures but return a value. **Usage:** Can be embedded within SQL expressions.

External Procedures: Note: Some systems allow procedures in languages like Java.

SQL/PSM Constructs Components: Local variables (`DECLARE`, `SET`), control structures (`IF/THEN/ELSE`, `LOOP`), and cursors.

Views: Definition: Virtual tables defined by SQL queries; they do not store data. **Uses:** Simplify complex queries, enhance security, and provide logical data independence **Updatability:** Simple views (single table, no aggregates) can be updatable; complex views are typically read-only.

Access Control: GRANT / REVOKE: GRANT: Assign privileges (SELECT, INSERT, UPDATE, DELETE, REFERENCES). **REVOKE:** Remove privileges. **Security Strategy:** Use views and stored procedures to restrict direct access to base tables.

Schema Levels: Physical Schema: How data is stored on disk (files, indexes, partitions). **Conceptual Schema:** The logical design: tables, keys, and constraints. **External Schema:** User-specific views of the data.

Storage Hierarchy: Components: Main Memory (RAM): Fast, volatile storage. **Secondary Storage (HDD/SSD):** Persistent storage; SSDs are faster but costlier. **Tertiary Storage:** For backups and archival.

Disk I/O Fundamentals: Components: Platters, Tracks, Sectors, Disk Heads. I/O Costs: Seek Time: Time to position the disk head. **Rotational Delay:** Waiting time for the correct sector. **Transfer Time:** Time to read or write data. **Optimization:** Reduce random I/O by using indexes and favoring sequential access.

General Indexing: Purpose: Accelerate data retrieval by reducing disk I/O. **Trade-Off:** Faster read operations versus slower writes due to index maintenance.

Tree-Based Indexes: ISAM: Static index; uses sequential leaf pages; may require overflow pages.

B+ Trees: Characteristics: Dynamic, balanced, high fanout (low tree height). All data stored in leaf nodes. **Operations: Insertion:** May require node splits and upward propagation. **Deletion:** May cause underflow, requiring redistribution or merging. **Composite Indexes:** Multi-column indexes where attribute order is critical.

Hash-Based Indexes: Usage: Best for equality searches using a hash function. **Techniques: Static Hashing:** Fixed number of buckets (using h(k) mod N). **Dynamic Hashing:** Extendible or linear hashing adapts to data growth. **Limitation:** Not effective for range queries.

Vertical Partitioning: Definition: Splitting a table into smaller tables by columns. **Usage:** Group frequently accessed columns to reduce the amount of scanned data.

Horizontal Partitioning: Definition: Dividing a table into subsets of rows based on a predicate or range. **Usage:** Manage large tables and improve query performance by scanning only relevant partitions.

Schema Tuning: Normalization: Organize data to reduce redundancy. (1NF, 2NF, 3NF, BCNF) **Denormalization:** Intentionally introduce redundancy for faster reads. **Materialized Views:** Precompute and store query results for rapid access.

Query Optimization: Execution Plans: Use `EXPLAIN` to see join orders, index usage, and cost estimates. **Query Rewriting:** Sometimes converting subqueries to joins (or vice versa) improves performance. **Covering Indexes:** Indexes that contain all required columns for a query, avoiding a table lookup.

Workload Analysis: Identify Critical Queries: Focus on those with high frequency or high resource usage. **Balance Read/Write Operations:** Adjust indexing and partitioning based on workload. **I/O Cost Analysis:** Evaluate seek time, rotational delay, and transfer time to pinpoint bottlenecks.

Clustered Index: Definition: Dictates the physical order of the data in the table. **Advantages:** Fast for range queries and sorting because data is stored in order. Only one clustered index per table, typically on the primary key. **Disadvantages:** Changes to the data (insertions, updates, deletions) may require reordering the table, which can be costly. **Unclustered (Non-clustered) Index: Definition:** A separate structure from the data itself; it contains pointers (RIDs) to the data rows. **Advantages:** Multiple unclustered indexes can exist on a table. Less impact on the physical ordering of data, so modifications can be less disruptive. **Disadvantages:** May require additional I/O operations: after using the index, the system must retrieve the actual data row using the pointer. Generally less efficient for range queries compared to a clustered index.

WHY b+ is better? ISAM is essentially **static** once created. The primary leaf pages are allocated in a fixed, contiguous fashion. When you insert into a "full" page, ISAM creates overflow pages instead of splitting and rebalancing. When you delete many keys, those pages remain half-empty (or you have leftover overflow pages). There is **no** mechanism to "merge" or "redistribute" pages. **In ISAM don't delete index or first 2 rows before leafs**

B+ Tree: Each node contains $d \leq m \leq 2d$ entries. The (mythical) d is called the order of the B+ tree.

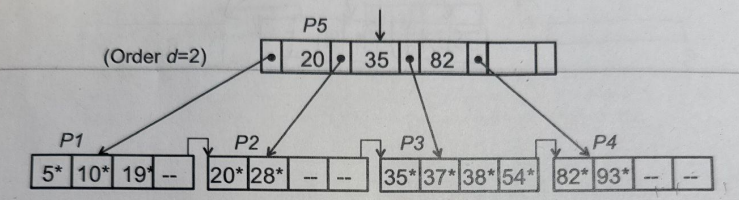
Primary vs. secondary clusters:
If the search key contains the primary key, it is called a primary index; otherwise, it is called a secondary index.

Single-column indexes (e.g., on `PHLogger(name)` and `Observable(rate)`) are great for equality and range queries on that one field, or trailing-wildcard string lookups.

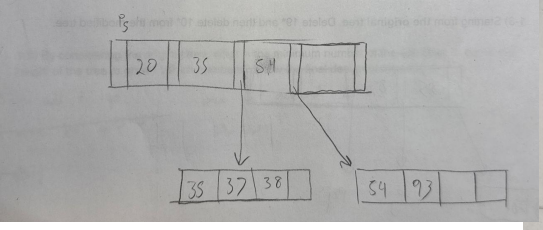
Composite indexes (e.g., `(manufacturer, model)`) shine when queries filter on multiple columns in the correct order.

Question 1) B+ Tree(30 Points)

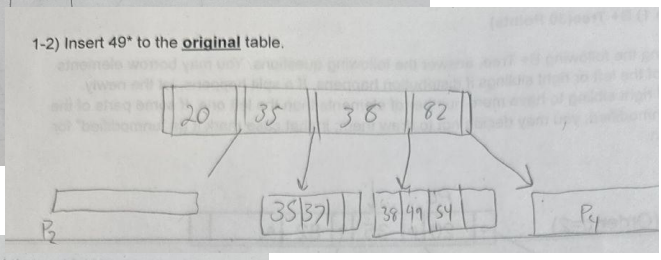
Considering the following B+ Tree, answer the following questions. You may borrow elements from any of the left or right siblings if distribution happens. If a split happens, let the newly generated right sibling to have more number of elements than the left one. If some parts of the tree are unmodified, you may decide not to draw them; in that case mark it as "unmodified" for clarification.



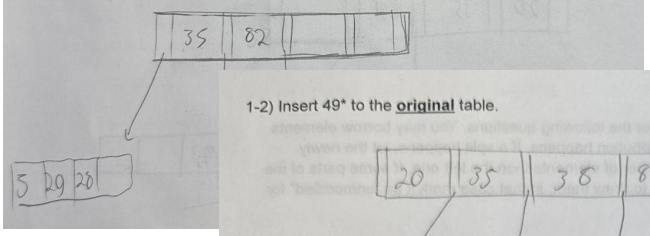
1-1) Delete 82* from the **original** table.



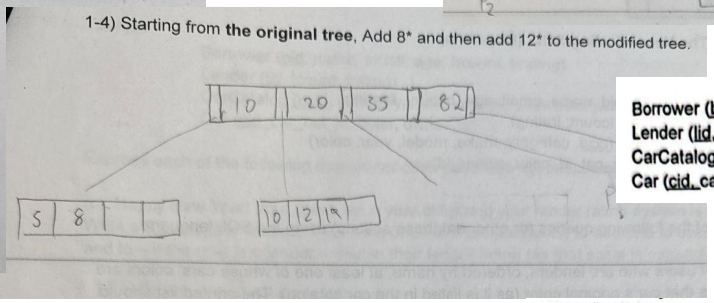
1-2) Insert 49* to the **original** table.



1-3) Starting from **the original tree**, Delete 19* and then delete 10* from the modified tree.



1-4) Starting from **the original tree**, Add 8* and then add 12* to the modified tree.



Borrower (bid, name, email, age, bcount, brating)
Lender (lid, lcount, lrating)
CarCatalog (ccid, category, make, model, year, color)
Car (cid, car_cat_id, color, owner_id)

3)Define a row-level trigger to ensure that no update can reduce an employee's age.

```
delimiter //
CREATE TRIGGER NoLowerAge BEFORE UPDATE ON Emp
    FOR EACH ROW
    BEGIN
      IF NEW.age < OLD.age THEN
        SET NEW.age = OLD.age;
      END IF;
    END;//
delimiter ;
```

```
CREATE INDEX PHLogger_name_index
    ON PHLogger(name);
```

Update Borrower
SET age = age+1;

Update Lender
Set lrating = 2+lrating;

4) Create a view named "emp_info" that shows the name and salary of employees who are working in a department with a budget more than $100,000.

```
create view emp_info as
select  distinct e1.ename,e1.salary from Emp e1, Works w1, Dept d1
where e1.eid = w1.eid and w1.did=d1.did and d1.budget >100000;
```

Emp(eid:integer, ename:string, age:integer, salary:real)
Works(eid:integer, did:integer, pcttime:integer)
Dept(did:integer, budget:real, managerid:integer)

1) For each department, print the department id, budget, and the average salary of employees who are work there for at least 40 percent of their time.

```
Select d.did, d.budget, avg(e.salary)
from Emp e, Dept d, Works w
where e.eid=w.eid and w.did=d.did and w.pcttime>=40
group by d.did,d.budget;
```

2) Print employees age and average salary based on their age, only if there are at least 4 employees w have that age.

```
select e.age,avg(e.salary)
from Emp e
group by e.age having count(*)>=4;
```

(b) Print a list of information about users who are under 18 years of age with e-mail accounts in the hotmail.com domain. The information printed should be each user's user id, name, email, age, borrow count, borrower rating, lend count, and lender rating. As some of the users may be borrowers but not lenders, the last two fields of the result may be null for some users, but all qualified users should indeed be represented in the results.

```
Select B.bid, B.name, B.email, B.age, B.bcount,
    B.brating, L.lcount, L.lrating
FROM Borrower B Left outer join Lender L
    ON (B.bid = L.lid)
WHERE Bage<18
AND B.email Like '%.hotmail.com'
```

d) Create a view with the columns of "Borrower Name", "Borrower's email address", "Borrowed Car ID", "Owner Rating" which shows the mentioned information about the borrowers, lenders, and cars that have been borrowed.

```
Create Views details ( Fullname, Email, CarColor, OwnerRating)
As  select B.name As Fullname, Bemail As Email,
    C.color As CarColor, L.lrating As OwnerRating
From Borrower B, Lender L, Car C
WHERE B.bid=L.lid AND L.lid = C.owner_id;
```

(a) Print a list of users who are lenders, ordered by name, at least one of whose cars' colors are still the same as that car's original color (as it is listed in the car catalog). The printed list should include the lender's user id, name, email, age, and their borrower and lender counts and ratings.

```
Select B.bid, B.name, B.email, B.age, B.bcount,
    B.rating, L.lcount, L.rating
From Borrower B, Lender L
WHERE B.bid = L.lid
    AND Exists (select * From Car C, CarCatalog CC
        WHERE C.car_cat_id = CC.ccid
        AND C.owner_id = L.lid
        AND C.color = CC.color)
ORDER BY B.name;
```

Express each of the following queries (or other database actions) using the SQL language.

(c) Happy New Year! Your users are a year older and your lender rating system is changing! Write a SQL update statement (or a set of UPDATE statements) to add one to every user's age and to – if the user is a lender – double their lender rating (as that scale is expanding by 2x).

```
1)
Select S.name
From Student S
Where NOT EXISTS (
   Select C.cid
   From Course C
   Where NOT EXISTS (
      Select T.cid
      From Taken T
      Where T.cid = C.cid AND T.sid = S.sid
   )
);

2)
Select S.sname
From Student S
Where S.sid IN(
   Select T.sid
   From Taken T, Course C
   Where T.cid = C.cid AND C.cname LIKE '%logy'
);
```

Q1) Print the name of all students, the cid of the course that they have taken and the year that they have taken each course. The output should include all students including those who did not take any course and put null for their cid and year of the course.

Q2) Print the quarter, year, cname and description of all courses including those courses that have never been taken by any student. The quarter and year should be null for such courses.

```
3)
DELIMITER $$
CREATE TRIGGER default_gpa_to_two
BEFORE INSERT ON Student
FOR EACH ROW
BEGIN
   IF NEW.gpa IS NULL OR NEW.gpa < 0 THEN
      SET NEW.gpa = 2;
   END IF;
END $$
DELIMITER;
```

1) Write a SQL query to print the name of the students who have taken all of the courses.(Subquery)

2) Write a SQL query to print the name of the students who have taken at least one course that ends with "logy"(like biology).(Subquery)

3) Write a trigger that when a new student is getting inserted with GPA less than zero or with GPA being null, it sets their GPA to 2.

```
Q1)
Select S.sname, T.cid, T.year
From Student S
LEFT OUTER JOIN Taken T ON (s.sid = t.sid);

Q2)
Select T.quarter, T.year, C.cname, C.description
From Course C
LEFT OUTER JOIN Taken T ON (C.cid = T.cid);
```

Student(sid,sname, major, gpa) //PK: sid

Course(cid, cname, description)//PK: cid

Taken(sid, cid, quarter, year) //PK: (sid,cid)