A Conceptual Guide to Understanding Database Internals

Overview

This document provides a layered breakdown of how a Database Management System (DBMS) works internally. It explains key components such as query optimization, transaction management, concurrency control, and data storage with diagrams, examples, and cost models.

Topics covered

Request Processing Layer

- Query parsing, validation, logical & physical optimization, relational algebra

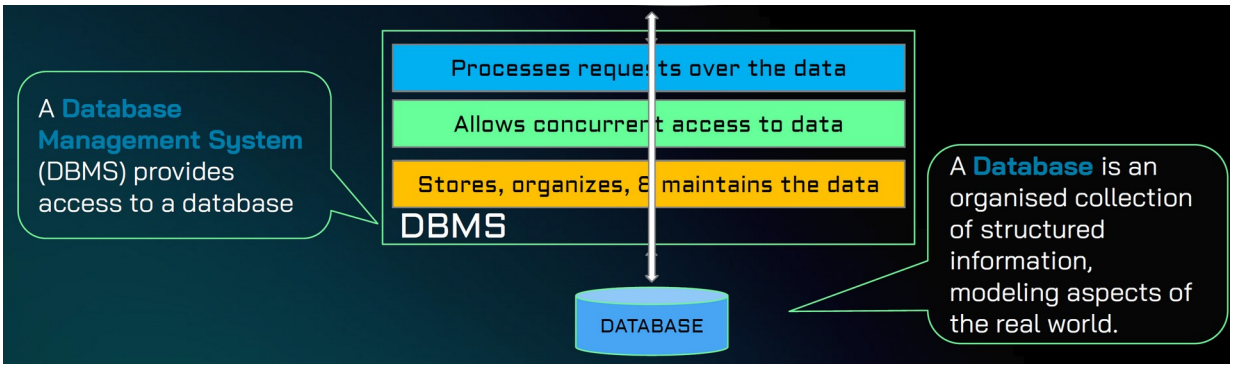
Concurrent Access Layer

- Transactions, ACID properties, isolation, scheduling, locking, logging, recovery

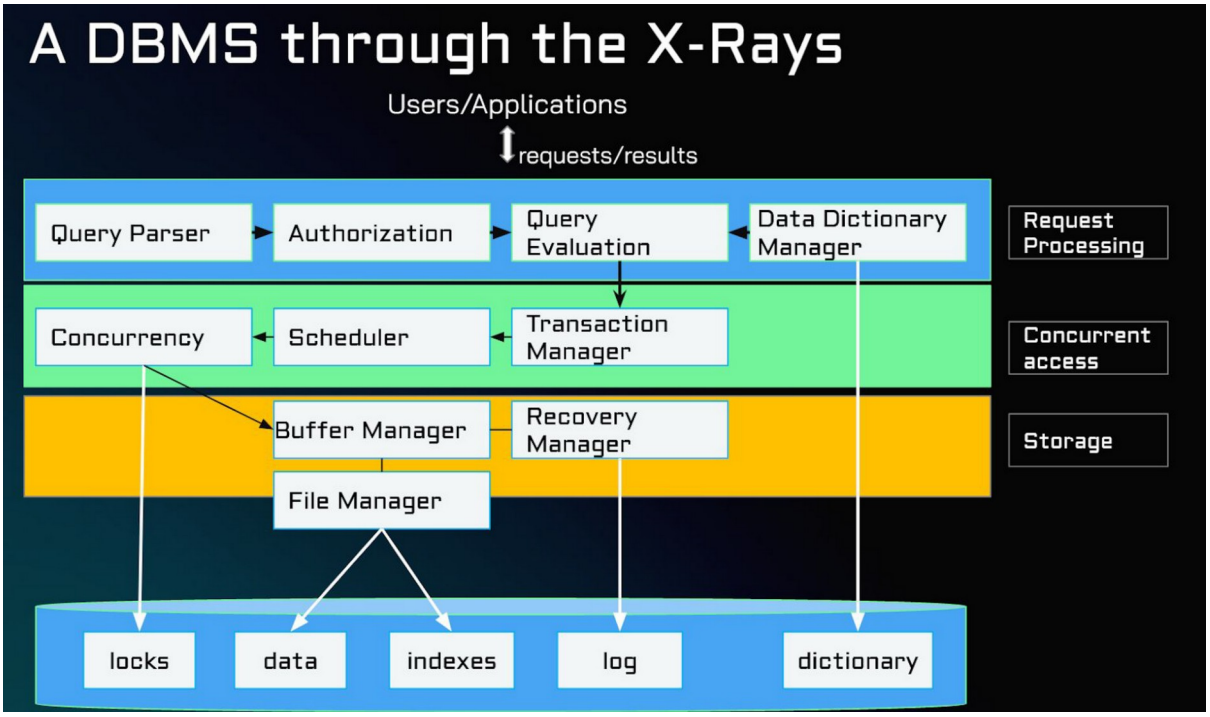
Storage Layer

- Files, pages, buffer manager, indexes, B+ trees, I/O cost, join algorithms

What is a Database Management System (DBMS)



A DBMS through the X-Rays: Understanding the Layers of a Database Management System



"A DBMS through the X-Rays" *diagram illustrates how a Database Management System (DBMS) is structured in three main layers: Request Processing, Concurrent Access and Storage.*

Each layer handles a different aspect of query processing and this architecture ensures correctness, security, concurrency and recovery.

A Database Management System (DBMS) is structured in **layers**, each handling different aspects of processing a query. At a high level, users or applications send **requests** (such as SQL queries) to the DBMS, which then **processes** these requests through its layers and returns the **results** back to the user. The top layer handles query interpretation and planning (**Request Processing**), the middle layer handles safe multi-user access (**Concurrent Access**), and the bottom layer manages data storage and retrieval (**Storage**). This layered design helps ensure that database operations are executed **correctly**, that only authorized access is allowed (providing **security**), that many users can work **concurrently** without conflict, and that the system can **recover** from errors or crashes. We'll describe the flow of a query through the DBMS and explain each layer and its components in simple terms.

Flow of Requests and Results in a DBMS

When a user or application sends a request to interact with a database (such as retrieving or updating data), the request passes through a series of layers within the **Database Management System (DBMS)**. Each layer plays a specific role to ensure that the request is correctly interpreted, safely executed, and efficiently fulfilled. The process is illustrated in the "A DBMS through the X-Rays" diagram, and typically follows this multi-step flow:

1. **User/Application Issues a Query**

   The journey begins when a user or client application issues a request — usually a SQL query such as: "SELECT * FROM Employees WHERE Department = 'Sales';". This request reaches the DBMS via a network or application connection. From there, the DBMS takes over and processes the query internally.

2. **Request Processing Layer – Parse, Check and Plan**

   The top layer handles **query interpretation and planning**:

   - **Query Parser**: Interprets the SQL request by analyzing its **syntax** (is it grammatically correct?) and **semantics** (do referenced tables/columns exist?). It transforms the query into an internal format, such as a parse tree, that represents its logical structure.
   - **Authorization**: Checks whether the requesting user or application has the required **privileges** to perform the requested operation (e.g., does the user have permission to SELECT from Employees?).

- **Query Evaluation**: Determines the most **optimal strategy** for executing the query — for example, deciding whether to scan the whole table or use an index on the Department column. This step generates the **execution plan**.

- **Data Dictionary Manager**: Manages metadata about the database (tables, columns, data types, etc.). It validates that all referenced schema elements are correct and supplies stats or definitions used by the optimizer.

By the end of this layer, the DBMS has a fully parsed, authorized, and optimized **execution plan** ready to run.

3. **Concurrent Access Layer – Manage Transactions and Multi-User Execution**

   Once the DBMS has a validated execution plan, the second layer focuses on **safe concurrent access and transaction control**:

   - **Transaction Manager**: Wraps the query inside a **transaction**, enforcing the **ACID properties**: Atomicity (all or nothing), Consistency (no violation of integrity rules), Isolation (no interference from other transactions), Durability (results persist even after crashes). It ensures the database remains in a consistent state before and after execution and guarantees that either all the operations of the query are applied successfully, or none are applied at all (e.g., in case of failure).
   - **Scheduler** (or **Lock Manager** or **Concurrency Controller**): **Schedules the execution** of operations, especially when multiple queries from different users might conflict (e.g., writing to the same row). Its goal is to allow safe concurrent access.

   - **Concurrency**: Controls **how data is accessed and modified** across transactions. It enforces isolation by managing simultaneous reads and writes, often using locks or multiversioning techniques to prevent conflicts.

   During this stage, the DBMS sets **locks**, assigns **timestamps**, and tracks the transaction's lifecycle to ensure correctness even in highly concurrent environments.

4. **Storage Layer – Retrieve, Manage and Persist Data**

   Once the query has passed through request processing and concurrent access, the third layer is responsible for **physically accessing and modifying data**:

   - **Buffer Manager**: Handles cache memory and manages data transfer between secondary storage and main memory. It checks whether requested data pages are already in memory and loads them from disk if needed, reducing disk I/O.
   - **File Manager**: Translates logical data requests into **low-level file operations**. It manages the file space and structures like tables, indexes, and heap files used to

represent information in the database. It interacts with storage to locate and retrieve data blocks.

- **Recovery Manager**: Ensures that the **data survives** in case of any system failure. It logs every operation, so that the system can **undo or redo** changes during recovery if needed.

At this stage, the actual rows from the Employees table (matching Department = 'Sales') are retrieved and brought into memory. The system ensures this data is correct, isolated, and recoverable.

5. **Results Returned to the User/Application**

   With the data fetched and/or modified, the DBMS sends the **result set** (e.g., the list of Sales employees) back up through its layers. Any necessary **post-processing or formatting** is applied before the result is returned to the client application or user. If the request modified data, the **Transaction Manager** commits the transaction, making the changes permanent and releasing any locks. If an error occurred, the transaction is **rolled back** using the **Recovery Manager's log**.

This flow, from parsing and planning, to transaction control, to storage access and recovery, ensures that every request is handled **correctly** (via parsing and optimization), **securely** (via authorization), **concurrently** (via transaction and lock management) and **reliably** (with recovery and logging). By abstracting each function into a separate **layer**, the DBMS maintains modularity and robustness. Understanding this architecture helps users and developers appreciate how a seemingly simple query request is carefully handled under the hood.

Storage Layer (Data Storage and Retrieval)

The storage layer is the bottom-most layer of the DBMS architecture. It is responsible for **storing data on disk and retrieving it efficiently**. This layer manages both the physical layout of the data on storage devices (like SSDs or HDDs) and in-memory copies of that data during execution. It allows the higher DBMS layers to operate without worrying about low-level file structures, I/O performance, or device-specific optimizations.

How is data stored in files?

- **A file is a collection of pages stored on disk**

In a DBMS, each table is stored as a **file**, which acts as a container. Internally, this file consists of multiple **pages**, the smallest units of I/O. The DBMS selects between two main file types depending on the expected I/O patterns:

- **Heap files**: No particular order; optimized for fast inserts.
- **Sorted files**: Pages and records within pages are ordered, usually by a key.

Example: Think of a folder containing files like authors.csv or books.csv. In a DBMS, each of these CSVs corresponds to a **file made up of disk pages**.

- **A page is the smallest unit of I/O and contains metadata, data, and a footer**

Pages are fixed-size blocks (typically 4KB–64KB) transferred between disk and memory. Each page includes:

- A **header** with metadata: page size, DBMS version, transaction info, etc.
- A **data section** that stores tuples (records), either fixed or variable length.
- A **footer** (slot directory) that stores [record pointer, record length] pairs, enabling efficient access to records within a page.

Example: If Authors table has 1000 rows and each page holds 50 rows, then it will span about 20 pages. When you request a row, the DBMS fetches the **entire page** the row belongs to.

Why both header and footer?
The **header** supports metadata management, recovery, and validation. The **footer** helps track record locations — especially important for **variable-length records** — without having to move or rewrite entire pages during inserts or deletes.

How is data retrieved from disk?

- **Buffer Manager**

The **buffer manager** oversees a region of memory (the buffer pool) that temporarily holds data pages fetched from disk. It caches frequently accessed data to reduce disk I/O, which is much slower than memory access. Accessing data in memory is much faster than reading from disk, so the buffer manager's job is to **minimize disk I/O by caching data in memory**. When an upper layer needs to read some data, it asks the buffer manager. The buffer manager will check, "Is the page that contains this data already in memory (in the buffer pool)?" If yes, it can directly return the in-memory

copy, avoiding a disk read. If not, the buffer manager will **fetch the page from disk** (using the file manager to do the low-level read) and bring it into the buffer pool, then give it to the requester. Similarly, when data is written (updated), the buffer manager keeps the changed page in memory (marked as "dirty") and will write it to disk later (this is often done in coordination with the recovery manager to ensure write-ordering with the log). The buffer manager uses algorithms (like **LRU – Least Recently Used**, or others) to decide which pages to evict from memory when the buffer pool is full and a new page needs to be brought in. In summary, the buffer manager **acts as the intermediary between the logical requests for data and the physical disk operations**, caching data to improve performance. It significantly speeds up database operations by reducing the number of direct disk accesses, as most queries will hit the cache for frequently used data. For example, if many queries involve the Employees table, the buffer manager will likely keep those pages in memory so subsequent queries can get the data quickly without going to disk each time.

Core operations:

- READ (page): Load from disk if not already in buffer.
- FLUSH (page): Write a modified page ("dirty") back to disk and evict it.
- RELEASE (page): Evict without writing (if no changes).

Replacement policies when space is full:

- **LRU (Least Recently Used)**: Evict the page least recently accessed.
- **MRU (Most Recently Used)**: Evict the most recently accessed page (useful in certain analytical workloads).
- **Toss Immediate**: Discard the page as soon as it's no longer needed — effective for OLTP systems with low page reuse.

OLTP vs. OLAP Workloads

- **OLTP (Online Transaction Processing):** Short, write-heavy transactions (e.g., banking systems). Often uses **Toss Immediate** or **LRU** to avoid caching pages unlikely to be reused.
- **OLAP (Online Analytical Processing):** Long-running, read-heavy queries (e.g., reporting, dashboards). Favors **LRU** or **MRU** depending on access patterns.

Why does the DBMS manage buffer space instead of the OS?
Because the DBMS has **deeper knowledge of access patterns and query logic**, it can optimize which pages stay in memory better than the OS can.

- **File Manager**

  The **file manager** (also called as **storage manager**) is responsible for managing the **physical layout of data on disk**. It knows where files and pages are stored, how records are arranged within pages, and how to read and write blocks from and to disk. It translates logical operations (like read all authors where country = 'Greece') into low-level file system commands.

  Example: If the optimizer chooses an index scan, the file manager retrieves the appropriate **index blocks** and **data blocks** using byte offsets and block pointers.

  Key difference between buffer and file manager:

  - **Buffer Manager:** Works in **main memory**, handles caching.
  - **File Manager:** Works on **disk**, handles layout, access, and storage.

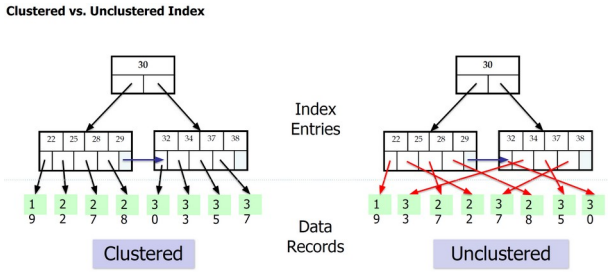
- **Indexes**

  Index is a separate data structure that contains "pointers" where the actual data is stored. They are utilized to facilitate quick and efficient data retrieval based on search criteria. They can build over one or more attributes. Indexes are used by the Query Optimizer to produce better execution plans.

  Types of indexes:

  - **Based on Attribute Type (Key vs. Non-Key):**
    - **Primary Index** is built on table's primary key. There is typically one index entry per page/block (not per row) and the data is physically sorted on this key. Often implemented as a clustered index.
    - **Secondary Index** is built on a non-primary or non-unique attribute (e.g. Department). There is one entry per record/row (Dense) and is not physically sorted on this attribute.
  - **Based on Physical Data Order (Clustered vs. Unclustered):**
    - **Clustered Index.** The data records are **physically ordered** by the index key. Because a table can be only sorted one way, there is **only one clustered index** per table. The index entries and data pages follow the same order.
    - **UnClustered Index.** The index is maintained **separately from the actual data layout**. It allows **multiple unclustered indexes** on a table. Lookups

involve following pointers to data blocks, which may be **scattered** across the file, possibly requiring more I/Os.



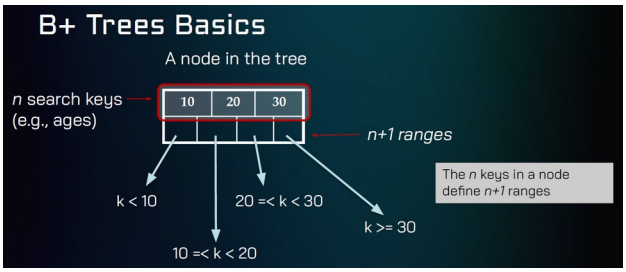
Why use Indexes?

Name: Joe Age: 11	Name: Jake Age: 15	Name: John Age: 21	Name: Bess Age: 22	Name: Bob Age: 27	Name: Sally Age: 28	Name: Sal Age: 30	Name: Sue Age: 33	Name: Jess Age: 35	Name: Alf Age: 37

Without indexes: searching age = 28 requires scanning O(n) rows, e.g. O(10).

With a **B+ Tree index**, it becomes O(log n), e.g. O(3).

B+ Tree: Structure and Search



A **B+ Tree** is a self-balancing tree used to implement indexes. It consists of nodes. Every node stores a list of *n* search keys and a list of *n + 1* pointers to child nodes (or to data). **Internal nodes** only contain keys and pointers used for search/navigation. **Leaf nodes** store actual data entries or pointers to the full records**.** Each node has **d to 2d children** and the number of keys is one less than the number of children.

Example: Page size: 4096 bytes. Key: 4 bytes, Pointer: 8 bytes. *d* = ?

In an internal node with *d* children, there are *d* pointers and *d – 1* keys. So, the total space needed is Size (4096) <= *d* * pointer_size (8) + (*d – 1)* * key_size (4) => *d* <= 341.6. But, we allow slack, so practical *d* ≈ 170.

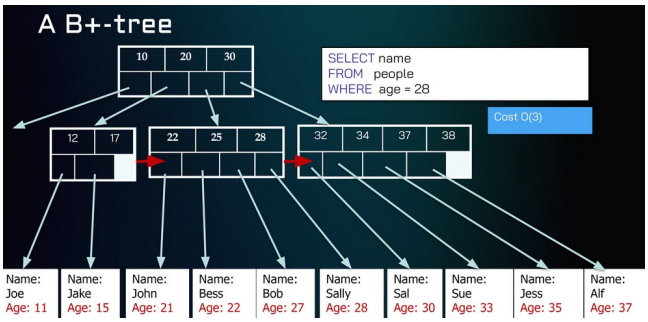
Fanout, Fill-factor and Tree Height

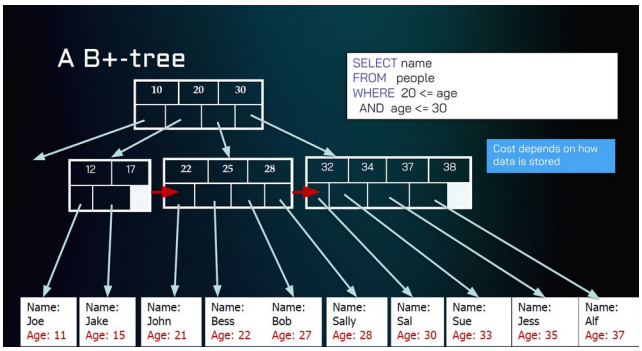
- **Fanout (f)**: Number of child pointers per internal node. A higher fanout leads to a wider tree, reducing the height and improving search speed (fewer I/Os).
- **Fill-factor (F):** Indicates how full the nodes are, typically a fraction (e.g. 2/3 = 66%). It accounts for space left for future insertions.

- **Height (h):** The number of levels from root to leaf nodes in a B+ Tree (e.g. h=3 in the figure below).

Example: We want to index N leaf pages using a B+ Tree. How tall the tree should be?

-> f^(h-1) = N / F. For F = 2/3, f = 170, N = 10000, h = 3.18 => h = 4. So the tree needs height h = 4 to index 10,000 pages with the given fanout and fill factor.





How many nodes (pages) must be accessed?

To answer the range query 20 <= age <= 30, the DBMS must:

1. Start at **root node** → access **1 page (1 I/O)**
2. Go to **middle internal node** (via 20 ≤ age < 30) → access [22, 25, 28] → **+1 page (1 I/O)**

3. Visit **leaf node** with [21, 22, 27, 28] → **+1 page (1 I/O)**

4. Follow the pointer to the **next leaf node** to [30, 33, ...] → **+1 page (1 I/O)**

Total Cost = 4 page I/Os (1 root + 1 internal + 2 leaf pages)

Summary: The Storage layer ensures persistent data storage, fast and efficient retrieval using caching and indexes, logical abstraction over complex physical storage and supports massive datasets that don't fit in memory. By combining **buffering, file layout**, and **indexing (e.g., B+ trees)**, it delivers data to higher DBMS layers with optimal speed and correctness.

Concurrent Access Layer (Transactions, Concurrency and Recovery)

The Concurrent Access layer is the middle layer of the DBMS. It is responsible for making sure that when **many users** are accessing or modifying the database at the same time, everything remains **correct, secure, and efficient**. This layer coordinates the execution of **transactions**, manages **concurrent access**, uses **locks** to prevent conflicts, and maintains **logs** so the system can recover from crashes.

Why we need Concurrent Access Management

Modern databases are accessed by thousands of users and applications. With so much activity happening at once, several challenges arise:

- **Concurrency:** Multiple users want to access/update the same data simultaneously.
- **Consistency:** If not handled properly, concurrent actions can lead to data corruption.
- **Performance:** We want users to feel like they are the only ones using the DB.
- **Crashes:** Systems can fail, but we want to ensure no data is lost or corrupted.

➢ **Transaction Manager**

The **transaction manager** oversees the execution of transactions — logical units of work composed of one or more database operations (such as multiple reads and writes). Its main role is to ensure that each transaction adheres to the **ACID properties** (Atomicity, Consistency, Isolation, and Durability), even in the presence of failures or concurrent users. When a query is issued, the transaction manager begins a new transaction or joins it to an existing one. It coordinates commit and rollback actions, and ensures the

system reaches a correct state whether the transaction succeeds or fails. The transaction manager works closely with the scheduler (for concurrency control) and the recovery manager (for durability and crash recovery). In short, the transaction manager acts as the **guarantor of correctness**: it makes sure that either all operations of a transaction are applied successfully, or none are — leaving the database in a consistent and recoverable state.

Transactions and the ACID properties

A **transaction** is a logical unit of work, usually made of multiple read/write operations (a group of tasks). A single task is the minimum processing unit which cannot be divided further. It should either be fully completed or have no effect at all.

Acct	Balance
a10	20.000
a20	15.000

This is an example transaction

Transfer $3k from a10 to a20:
1. Debit $3k from a10
2. Credit $3k to a20

Acct	Balance
a10	17.000
a20	18.000

A transaction in SQL:

```
START TRANSACTION
    UPDATE Bank SET amount = amount – 3000
    WHERE name = 'Bob'
    UPDATE Bank SET amount = amount + 3000
    WHERE name = 'Joe'
COMMIT
```

Grouping user actions (reads and writes) into transactions helps in Recovery and Durability as well as in Concurrency.

Transaction Properties: ACID. DBMSs guarantee the following properties:

•Atomicity: all the actions in a transaction are executed as a single atomic operation either they are all carried out or none are

•Consistency: if a transaction begins with the DB in a consistent state, it must finish with the DB in a consistent state

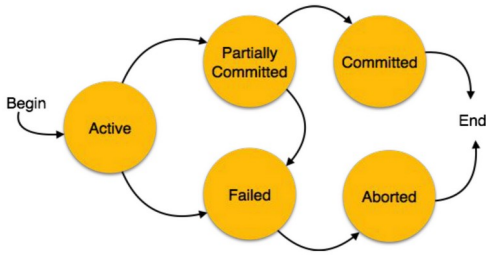
•Isolation: a transaction should execute as if it is the only one executing; it is protected (isolated) from the effects of concurrently running transactions

•Durability: if a transaction has been successfully completed, its effects should be permanent

Example: Transferring $3000 from Account A to B. Both the debit and the credit must happen together, and the change must be preserved even if there's a crash right after.

Transaction States

- **Active** – The transaction is being executed. This is the initial state of every transaction.
- **Partially Committed** – When a transaction executes its final operation.
- **Failed** – If any of the checks made by the database recovery system fails. A failed transaction can no longer proceed further.
- **Aborted** – If any of the checks fails and the transaction has reached a failed state, then the recovery manager rolls back all its write operations on the database to bring the database back to its original state prior to the execution of the transaction. The database recovery module can select to–
  - Re-start the transaction
  - Kill the transaction
- **Committed** – If a transaction executes all its operations successfully, it is said to be committed. All its effects are now permanently established on the database system.



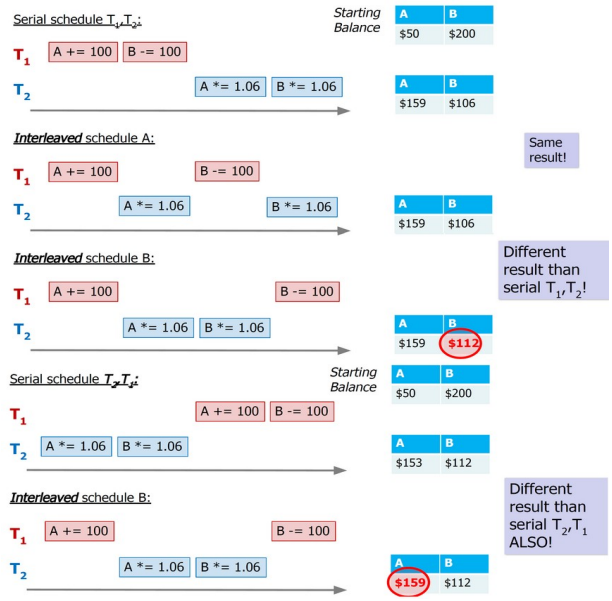
➢ Scheduler

The **scheduler** controls the order in which operations from different transactions are executed. It is responsible for ensuring that multiple transactions can safely execute at the same time without corrupting the database. In environments with many users or applications, transactions often overlap in time, trying to read or write the same data. The scheduler prevents harmful interactions between these operations by coordinating access using **concurrency control mechanisms**, typically locks or timestamps. The scheduler determines the execution order of operations so that the resulting outcome is equivalent to some **serial execution** of the same transactions — even though they may run concurrently. This property is called **serializability**, and it's essential for correctness. To do this, the scheduler may delay, reorder, or block operations that would cause conflicts. For instance, if one transaction is updating a record, the scheduler may prevent another from reading or writing it until the first transaction completes. The scheduler also manages **deadlock detection and resolution**, ensuring the system doesn't freeze due to cyclic waits. In essence, the scheduler acts as the **traffic controller of the DBMS**, managing interleaved execution of transactions to ensure correctness, consistency, and fairness — while maximizing performance.

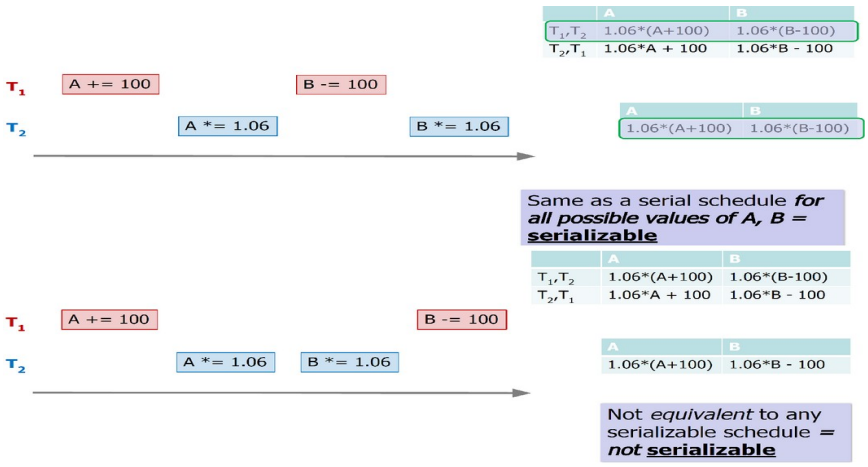
Example: T1 transfers $100 from B to A. T2 applies 6% interest to both.

- If scheduled badly, the result is wrong (non-serializable).
- If scheduled well, the outcome is the same as if one transaction fully finished before the next.

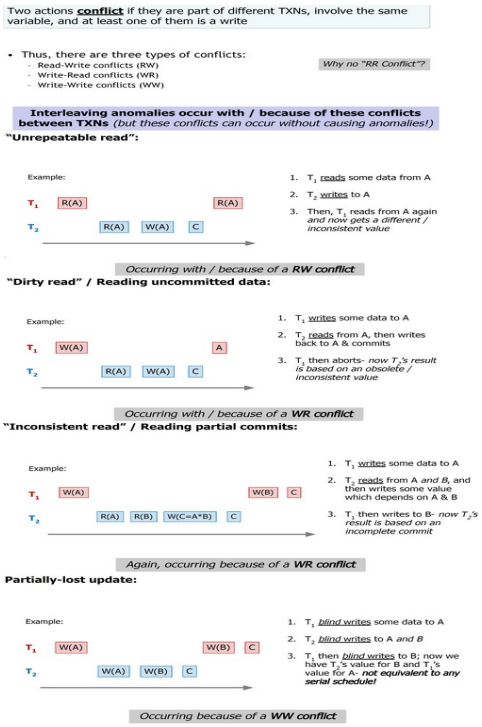
Scheduling Examples



- A **serial schedule** is one that does not interleave the actions of different transactions
- A and B are **equivalent schedules** if, ***for any database state***, the effect on DB of executing A **is identical to** the effect of executing B
- *A* **serializable schedule** is a schedule that is equivalent to ***some*** serial execution of the transactions.



Conflict Types & Anomalies



To safely allow concurrent execution, the DBMS uses **concurrency control**. Without it, 4 basic types of problems can occur:

- **Unrepeatable Read**: Reading the same data twice and getting different results (interleaving reads and writes).
- **Dirty Read**: Reading data from a transaction that later aborts (uncommitted data).
- **Lost Update**: One transaction overwrites another's result (concurrent updates).
- **Phantom Row**: A query returns different rows the second time (new data not appearing in the result of a query).
    - **Lost Update**: two people, in different shops, buy the very last ticket for the U2 concert in Rome (!?)
    - **Dirty Read**: the U2 tour schedule shows a date in Bologna on 15/07/17, but when you try to buy the ticket for that concert the system tells that no such date exists (!?)
    - **Unrepeatable Read**: for the U2 concert (finally, the date has been decided!) you see a price of 90 €, you think about it a little, but when you're decided, the price is risen to 110 € (!?)
    - **Phantom Row**: you want to go see both U2 concerts in Italy, but when you try to buy tickets, you discover that there are now three dates (!?)

Locks: The Main Tool for Isolation

A technique commonly used by DBMSs to avoid previous problems consists in **locks.** Locks are a mechanism normally used by operating systems to regulate access to shared resources. Before executing any operation, it is required to "acquire" a lock on the requested resource (e.g. a record). The lock request is implicit, thus invisible at SQL level. The basics are:

- **Shared Lock (S):** For reading a value.
- **Exclusive Lock (X):** For writing/updating a value.

The **Lock Manager** is a DBMS module in charge of keeping track which resources are currently used and which transactions are using them (and how)

- When a transaction T wants to operate on a value Y, a lock request on Y is sent to the Lock Manager
- Lock is granted to T according to the following compatibility table

T requests a lock of type	Another transaction has on Y a lock of type: S	Another transaction has on Y a lock of type: X
S	OK	NO
X	NO	NO

- When T finishes using Y, can release the lock (unlock(Y))

Locks ≠ Logs: **Locks** manage access and prevent concurrency errors, while **Logs** help **recover** the DB if something goes wrong.

➢ Recovery Manager

The **recovery manager** ensures that the database can recover to a consistent state after a crash, error, or aborted transaction. It works hand in hand with the transaction manager and scheduler to preserve the **durability** and **atomicity** properties of transactions. Before any changes are written to the actual database, they are first recorded in the log. This allows the system to **redo** committed changes (that may not have reached disk) or **undo** uncommitted ones (to roll back partial work). During normal execution, the recovery manager ensures that every update operation is logged properly and that commit operations are fully durable. If a crash occurs, the DBMS replays the log during recovery. It first **redoes** all operations from committed transactions to ensure their effects are preserved, and then **undoes** operations from transactions that were active but never completed. This guarantees that the database will always be brought back to a **valid, consistent state** — even after an unexpected failure. The recovery manager is the DBMS's **safety net**, protecting data integrity under all circumstances.

The following actions are recorded in the **log**:

- Every write operation
- Transaction begin, commit, abort events

The log record must be on disk **before** the data record reached the disk. Log is duplexed and archived on stable storage. All log records are handled by the DBMS and saved before confirming COMMIT. The user does know anything.

Types of failures

- Transaction failure: is the case of an aborted transaction
  - The effects of such transaction on the DB have to be un-done
- System failure: the system has an hardware or software failure, stopping all current transactions, but the secondary memory (disk) is not damaged
- Media/device failure: in this case the (persistent) content of the database is damaged

Why do we need logging for atomicity?

- Couldn't we just write TXN to disk **only** once whole TXN complete?
  - Then, if abort / crash and TXN not complete, it has no effect- atomicity!
  - *With unlimited memory and time, this could work…*
- However, we **need to log partial results of TXNs** because of:
  - Memory constraints (space for full TXN??)
  - Time constraints (what if one TXN takes very long?)

We need to write partial results to disk!
…And so we need a **log** to be able to ***undo*** these partial results!

If a crash happens:

- **Undo**: Reverse actions of uncommitted transactions
- **Redo**: Reapply actions of committed ones

- DB uses **Write-Ahead Logging (WAL)** Protocol:

1. Must *force log record* for an update *before* the corresponding data page goes to storage

2. Must *write all log records* for a TX *before commit*

Crash recovery (ARIES Protocol: used by most DBMS):

1. **Analysis**: Identify active and committed transactions at crash time.
2. **Redo**: Reapply all updates from committed transactions.

3. **Undo**: Roll back uncommitted changes.

The Concurrent Access Layer gives the DBMS its power to handle **many users safely**, **preserve correctness**, and **recover from errors**. It combines transactions, scheduling, locking, logging, and recovery into a single mechanism that ensures ACID compliance without sacrificing performance.

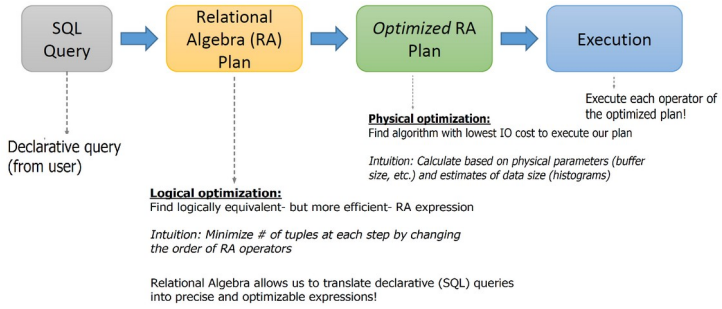
Request Processing Layer

The **Request Processing Layer** is the **topmost layer** of the DBMS architecture. Its primary responsibility is to interpret the user's SQL query, validate it, plan how to retrieve the result, and pass a well-defined execution plan to the lower layers for processing.

Think of it as the “front-end intelligence” of the DBMS — it doesn't fetch data directly, but it decides **how** data should be fetched. Main responsibilities:

- **Query Parser:** The query parser checks the syntax and structure of the SQL query. It breaks the query into components (SELECT, FROM, WHERE, etc) builds an internal representation (typically a parse tree) and validates SQL grammar and object names. Example: For “SELECT * FROM Employees WHERE Dept = 'Sales';”, the parser checks SQL validity and builds a parse tree showing a selection operation over the Employees table.
- **Authorization Manager:** After syntax validation, the **Authorization Manager** ensures that the user has the necessary **access rights** to execute the query by checking the permissions for operations like SELECT, INSERT and UPDATE and ensuring security policies on restricted attributes or tables. Example: A user without SELECT access on the Salary column will be denied execution if the query includes it.

- **Query Optimizer:** Once the query is valid and authorized, the **Query Optimizer** determines the **best strategy to execute it**. It has two key phases:



  - PHASE 1: **Logical Optimization**
    - Converts the parse tree into a **Relational Algebra (RA) expression.**
    - Applies **heuristic rules** (e.g., pushing down selections, reordering joins).
    - Produces an **optimized RA plan** that is logically equivalent but more efficient.

    Logical optimization minimizes intermediate results (e.g., number of tuples) and simplifies the plan before considering costs. Relational Algebra allows us to translate SQL queries into precise and optimizable expressions.

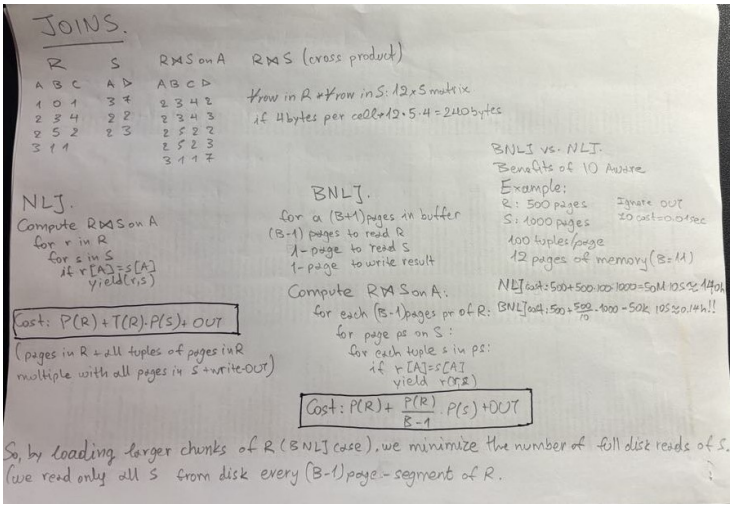
  - PHASE 2: **Physical Optimization**
    - Takes the optimized RA plan and searches for the **lowest-cost execution strategy**
    - Chooses among physical algorithms (e.g., hash join vs. nested loop join)
    - Estimates **I/O cost** based on table statistics (e.g. cardinality, selectivity), physical parameters (e.g. page size, available memory) and indexes or clustering.

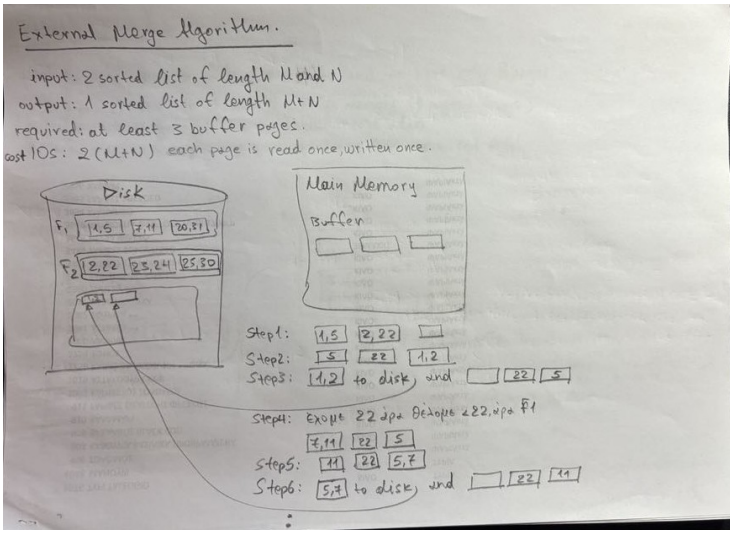
    Example: For a join, the optimizer may prefer a **Block Nested Loop Join** over a **Naive Nested Loop** because it requires fewer disk reads.

Illustrative Examples: Join Strategies, External Sorting, and Relational Algebra Optimization

Now that we've described how the optimizer selects a physical plan, let's walk through practical examples of **join strategies** (such as NLJ and BNLJ), **external merge algorithms**, and **relational algebra optimizations**. These illustrations help visualize how the DBMS evaluates cost, reduce intermediate results, and improve performance at each step of query execution.

We use **external merge** and **external merge sort** algorithms when the data is too large to fit in memory, requiring efficient disk-based strategies that minimize the number of expensive I/O operations.





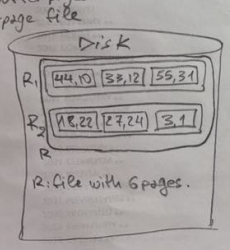
External Merge Sort Algorithm.

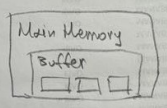
a classic problem in CS!

what happens when a file has 6 pages and we have only 3 pages in memory?

1. Sort phase: Split into chunks to sort in memory (3 pages) - (runs)
2. Merge phase: Merge pairs using external merge algo
3. Keep merging the resulting runs until one left one sorted file!

example with
3 buffer pages
6-page file





1. phase 1: split R into 3-pages chunks and sort them into buffer
step1: in buffer 44,10 | 33,12 | 55,31 → 10,12 | 31,33 | 44,55 and back to disk

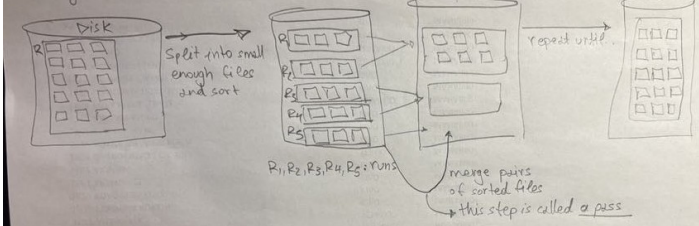
step2: the same for R2, so we have in disk:

10,12	31,33	44,55
1,3	18,22	24,27

2. phase 2: run external merge algo for this & we're done!

Cost: [1R+1W for each run (phase 1)] + [2·(M+N) (phase 2)] = 2·(3+3) + 2(3+3) = 24 IOs

Running External Merge Sort on Larger Files.



for N-page file in disk and a single-page runs:
first pass: $\frac{N}{2}$ pairs of runs, each of length 1 page
second pass: $\frac{N}{4}$ pairs of runs, each of length 2 pages

In general, we need $\log_2 N$ passes + 1 (initial split & sort)
Each pass involves reading in and writing out all pages → 2N IO

Cost: $2N*(\lceil \log_2 N \rceil + 1)$ IOs! ⟹ using B+1 buffer pages → $2N*(\lceil \log_{B-1} \frac{N}{B} \rceil + 1)$ (for B if we perform B-way passes.)

Query Logical Optimization.

Example 1.

Assume these tables:

PROJECT(Pnumber, Plocation, Dnum)
DEPARTMENT(Dnumber, Dname, Mgr_ssn)
EMPLOYEE(SSN, Fname, Lname, Address, Bdate)

Query: for every project located in 'Stafford' that has 'Smith' as manager, retrieve the project name, the controlling department number, and the department manager's last name, address and birth date

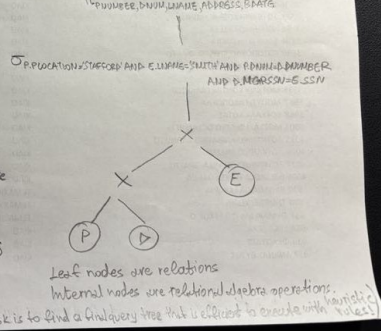
SQL:

```
SELECT P.Pnumber, P.Dnum, E.Lname, E.Address, E.Bdate
FROM PROJECT P, DEPARTMENT D, EMPLOYEE E
WHERE P.Dnum=D.Dnumber, D.Mgr_ssn=E.SSN
  AND P.Plocation='Stafford' AND E.Lname='Smith';
```

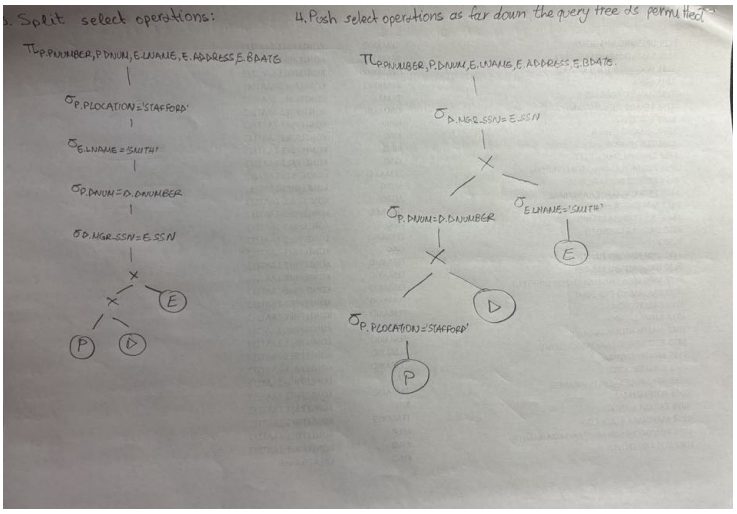
1. Rewrite the query into relational algebra:

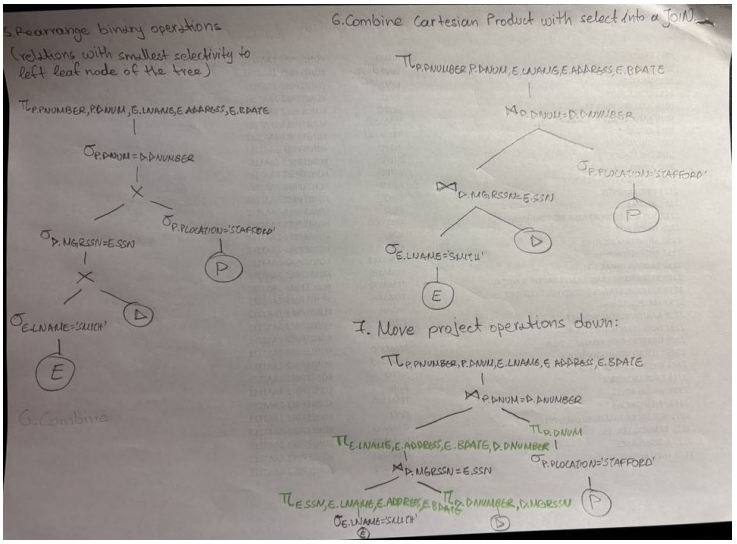
$\pi_{PNUMBER, DNUM, LNAME, ADDRESS, BDATE}(\sigma_{PLOCATION='STAFFORD' \text{ AND } LNAME='SMITH' \text{ AND } DNUM=DNUMBER \text{ AND } MGRSSN=SSN}(PROJECT \times DEPT \times EMP))$

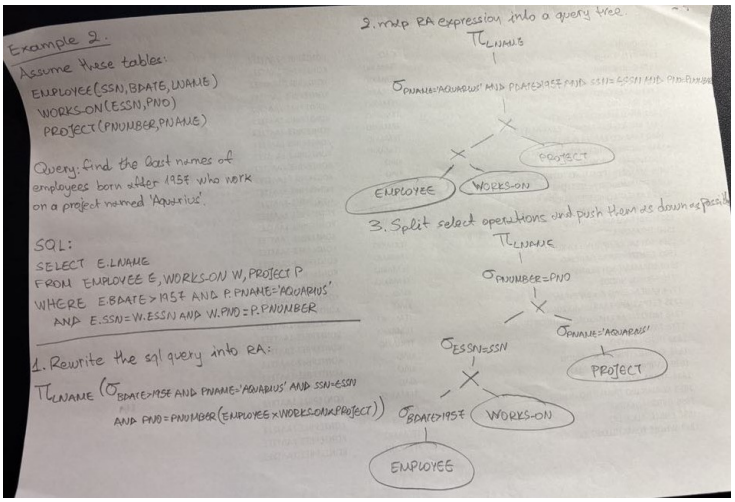
2. Map the relational algebra expression into a query tree

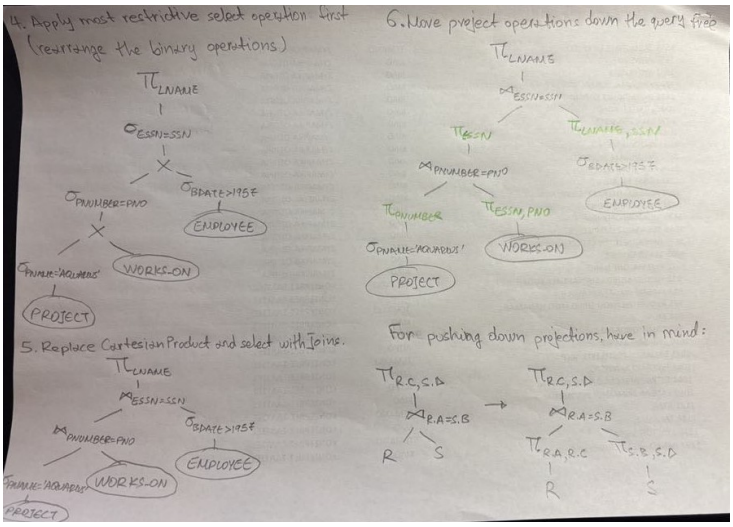


Leaf nodes are relations
Internal nodes are relational algebra operations.
(the task is to find a final query tree that is efficient to execute (usually))


4. Split select operations:
4. Push select operations as far down the query tree as permitted:


5. Rearrange binary operations (relations with smallest selectivity to left leaf node of the tree)
6. Combine Cartesian Product with select into a JOIN
7. Move project operations down:


Example 2:
Assume these tables:
EMPLOYEE(SSN,BDATE,LNAME)
WORKS-ON(ESSN,PNO)
PROJECT(PNUMBER,PNAME)
Query: find the last names of employees born after 1957 who work on a project named 'Aquarius'.
SQL:
SELECT E.LNAME
FROM EMPLOYEE E, WORKS-ON W, PROJECT P
WHERE E.BDATE>1957 AND P.PNAME='AQUARIUS'
AND E.SSN=W.ESSN AND W.PNO=P.PNUMBER
1. Rewrite the sql query into RA:
2. map RA expression into a query tree
3. Split select operations and push them as down as possible



After optimizing a query's structure with relational algebra and join strategies, the DBMS still needs to decide **which physical plan is the most efficient to execute**. This choice is made using **cost estimation techniques**, which evaluate how many I/O operations different plans would require.

Why This Matters:

Estimating cost helps the Query Optimizer choose the fastest way to run a query, especially when there are many joins or conditions. A smart optimizer can reduce query time from **minutes to milliseconds** just by choosing a better plan.

The example below shows a sample query and its relational algebra tree, along with the metadata needed to estimate the **cost of executing the plan**:

(a)

Table_name	Column_name	Num_distinct	Low_value	High_value
PROJECT	Plocation	200	1	200
PROJECT	Pnumber	2000	1	2000
PROJECT	Dnum	50	1	50
DEPARTMENT	Dnumber	50	1	50
DEPARTMENT	Mgr_ssn	50	1	50
EMPLOYEE	Ssn	10000	1	10000
EMPLOYEE	Dno	50	1	50
EMPLOYEE	Salary	500	1	500

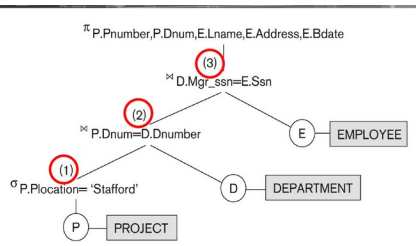
(b)

Table_name	Num_rows	Blocks
PROJECT	2000	100
DEPARTMENT	50	5
EMPLOYEE	10000	2000

(c)

Index_name	Uniqueness	Blevel*	Leaf_blocks	Distinct_keys
PROJ_PLOC	NONUNIQUE	1	4	200
EMP_SSN	UNIQUE	1	50	10000
EMP_SAL	NONUNIQUE	1	50	500

*Blevel is the number of levels without the leaf level.



Let us assume that we have these tables:
PROJECT(Pnumber, Plocation, Dnum, PStartDate)
DEPARTMENT(Dnumber, Dname, Mgr_ssn)
EMPLOYEE(SSN, Fname, Lname, Address, Bdate)

SQL:

```
SELECT P.Pnumber, P.Dnum, E.Lname, E.Address, E.Bdate
FROM   PROJECT P, DEPARTMENT D, EMPLOYEE E
WHERE P.Dnum = D.Dnumber AND D.Mgr_ssn = E.SSN
  AND P.Plocation = 'STAFFORD';
```

How Do We Compute the Cost of an Execution Plan?

The execution plan is evaluated **bottom-up**, starting from the **leftmost leaf node** (typically the first base relation accessed), and proceeds upward through relational algebra operations. At each internal node, intermediate results are computed and passed up the tree until the final result is produced at the root.

- First, we estimate the **selectivity**, which measures how "restrictive" a condition is. It tells us what **fraction of rows** in a table will match a condition. For example, for "P.Plocation='STAFFORD'", and 200 distinct values in the column, the selectivity = 1 / 200 = 0.005.
- Next, we compute the **selectivity cardinality**, which is the estimated number of **tuples (rows)** that match the condition. If the PROJECT table has 2000 rows, then, selection cardinality = 2000 * 0.005 = 10 tuples.

- Since I/O cost is based on **pages (blocks)**, we estimate how many pages are needed to store these tuples. Use the **Blocking Factor** = number of rows per page (e.g., 20). Pages needed = Ceiling(selection cardinality / Blocking Factor) = 10 / 20 = 1 page. So, number of **blocks needed = 1**.

- Finally, we estimate the I/O cost for this access (bottom leftmost leaf node).
    - For a full table scan, **cost = 100 IOs** (total number of pages in the table).
    - For an index scan, assuming Index B-level = 1, Leaf block = 1 I/O and, for un-clustered data => each of the 10 matching tuples may be on separate page, **cost = 12 IOs** (1 (B-level) + 1 (leaf block) + 10 (data blocks)). This cost is the estimated cost based on the worst-case scenario that each matching tuple lies on a different page.

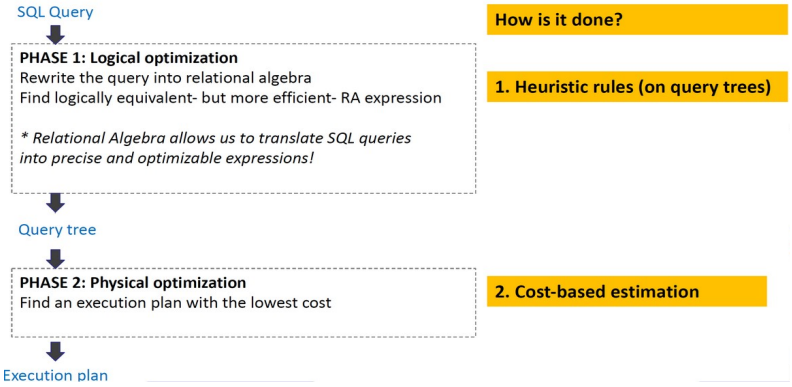
- Moving to the next level (2) and level (3), we need to estimate the Join cost using:
    - Nested Loop Join (NLJ): Cost = Pages of outer + (Number of tuples in outer × Pages in inner) + OUT.
    - Block Nested Loop Join (BNLJ): Cost = Pages of outer + (Ceiling (Pages of outer ÷ (B - 2)) × Pages of inner) + OUT.
    - Index Nested Loop Join (INLJ): Cost = Pages of outer + (Tuples in outer × (Index depth L + 1)) + OUT.

    For example for level (2): NLJ, TEMP1 (the result of previous, first access) has 10 rows and the DEPARTMENT table has 5 pages, **Cost** = 1 + (10 × 5) + OUT (cost to write join output to TEMP2) = 51 + 2 **= 53 IOs.** Here, for estimated blocking factor = 5, we will need 2 blocks, so the cost of OUT is becoming 2. For level (3): we can use INLJ, so **Cost** = 2 + 10* (1 + 1 + 1) + OUT (Cost per lookup = 1 (B-level) + 1 (leaf) + 1 (data)) = 32 + OUT (writing final output).

    So, **total cost = 12 + 53 + 32 + cost of writing the final output.**

    So, the Query Optimizer needs

- Information about how to compute relational operators in the tree based on access paths and algorithms variables.
- Information about the data stored (System Catalog information).
- Formulas to compute costs and cardinalities.
- Strategy to generate plans and select the one to be executed.



As shown in the diagrams:

- **Logical Optimization** uses **heuristic rules** on query trees
- **Physical Optimization** uses **cost-based estimation**

Together, these ensure the final plan is correct and efficient.

- **Data Dictionary Manager:** The **Data Dictionary Manager** handles access to the **system catalog**, a repository of metadata about the database like
  - Table and column definitions
  - Indexes, constraints
  - User privileges and roles
  - Statistics for optimization

Both parser and optimizer consult the data dictionary:

- The parser checks whether objects exist and how they're structured
- The optimizer retrieves table sizes, index information, and histograms to estimate cost

By the end of this layer, the DBMS has:

- Checked if the query is well-formed and secure
- Transformed it into **relational algebra**
- Optimized it using **heuristics and cost models**
- Produced a **query execution plan**

This plan is passed to the **Query Execution Engine**, which carries it out using the lower DBMS layers (including the Storage Layer).

In conclusion, the DBMS architecture depicted in "A DBMS through the X-Rays" can be understood as a pipeline of layers, each with distinct responsibilities that together handle a user's request. A query travels from the **Request Processing layer** (where it is parsed, checked, and optimized) through the **Concurrent Access layer** (where transactions and concurrency control guarantee correctness and isolation, and logging guarantees recoverability) down to the **Storage layer** (where the data resides and is fetched/updated). The results then propagate back up to the user. Each component — from the Query Parser and Authorization checks at the start, to the Transaction and Recovery Managers in the middle, to the Buffer and File Managers at the bottom — plays a crucial role in ensuring that the DBMS operates **correctly, securely, and efficiently**. This layered approach allows the system to manage complexity: for example, the query parser doesn't need to know how data is stored on disk, and the storage manager doesn't need to know the intricacies of SQL syntax. By understanding these layers and components, students can appreciate how a DBMS **takes a high-level request and handles all the low-level details** to return the correct result while preserving the integrity and security of the data.

Summary:

Request Processing Layer

- **Query Parser**
  - Parses SQL into internal form (e.g., relational algebra tree).
  - Checks syntax and semantics using the **Data Dictionary**.
- **Authorization**
  - Verifies user permissions for the requested operation (SELECT, UPDATE, etc.).
- **Query Evaluation**

- Converts query into **relational algebra**, performs logical and physical **optimization**.
- Generates an **execution plan** using metadata/statistics from the **Data Dictionary**.

- **Data Dictionary Manager**
  - Manages the **Data Dictionary**: metadata about tables, attributes, indexes, users, etc.
  - Supports parsing, authorization, and optimization.

Concurrent Access Layer

- **Transaction Manager**
  - Groups operations into transactions and manages BEGIN, COMMIT, and ROLLBACK.
  - Ensures **ACID properties**: Atomicity, Consistency, Isolation, Durability.
- **Scheduler**
  - Implements **concurrency control** (e.g., locking, timestamp ordering).
  - Coordinates **safe access** to shared data by concurrent transactions.
- **Recovery Manager**
  - Ensures **durability** and **atomicity** using logs and checkpoints.
  - Restores database after failures via **UNDO** (rollback) and **REDO**.

Storage Layer

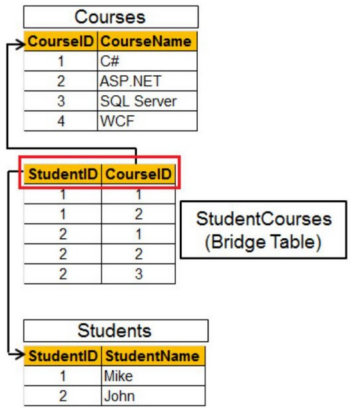
- **Buffer Manager**
  - Caches data pages in **main memory** to reduce disk I/O.
  - Interfaces between in-memory operations and disk-resident data.
- **File Manager**
  - Handles physical storage: allocates, reads, and writes data on disk.

- Manages access methods (e.g., heap files, indexes).

- **Locks**
  - Represent **lock states** on disk-resident data (e.g., rows, pages).
  - Controlled by the **Scheduler** during transaction execution.

Types of DBMS

- **Relational DBMS (RDBMS)**



  - Organizes data into **structured tables** consisting of rows and columns.
  - Each table represents an entity (e.g., Students, Courses), and each row is a record.
  - Uses unique identifiers **(primary keys)** to ensure each row is uniquely identifiable.
  - Supports **relationships between tables** using foreign keys (e.g., StudentCourses links Students and Courses).
  - Enforces data integrity through constraints like primary key, foreign key and not null.
  - Allows complex querying and data manipulation using **SQL (Structured Query Language)**.
  - **Supports normalization**, which reduces redundancy and improves consistency.
  - Used in most traditional applications like banking, education, inventory, etc.

- **NoSQL Databases**

NoSQL (Not Only SQL) databases are designed for flexibility, scalability, and handling unstructured or semi-structured data. They **don't rely on fixed schemas** like traditional relational databases. Commonly used in **big data**, **real-time web apps**, and **distributed systems**. Some types are:

- **Key-Value Databases:** Data is stored as simple key–value pairs (e.g., Redis, DynamoDB). Best for fast lookups, caching, and session management.
- **Document Databases:** Store data as flexible documents, typically in **JSON** or **XML** format (e.g., **MongoDB**, **CouchDB**). Ideal for content management systems, user profiles, or product catalogs.
- **Graph Databases**: Represent data using **nodes** (entities) and **edges** (relationships) (e.g., **Neo4j**). Excellent for modeling and querying complex relationships, such as **social networks**, **recommendation engines**, or **fraud detection**.

Examples DBMD

- Commercial: Oracle, MySQL Server, Teradata, IBM Db2, Databricks.
- Open-Source: MySQL, Postgres, Cassandra, Redis, MongoDB, Presto.

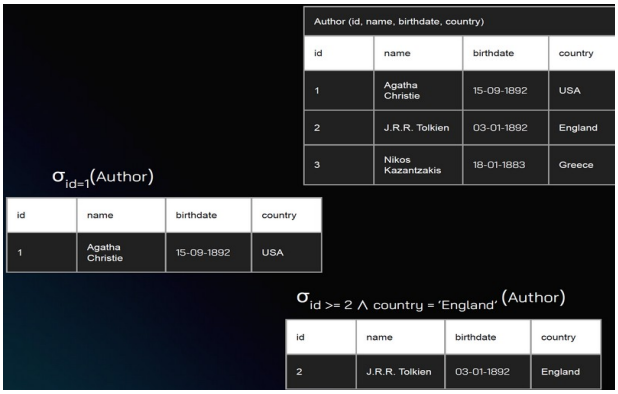
Relational Model – Key Concepts

- Based on relations (tables): Data is organized into tables (called **relations**) made up of **rows (tuples)** and **columns (attributes)**. A **tuple** is a complete set of attribute values for one entity (e.g., an author).
- Each table has a primary key: The **Primary Key** uniquely identifies each tuple in a relation.
- Tables are linked via foreign keys: A **Foreign Key** connects one table to another by referencing its primary key, enabling relationships between entities.
- **SQL** is the standard query language, but some systems also support **Natural Language Queries.**
- Enables schema definition and enforcement: A schema defines the structure of the database including tables, keys and data types.
- Declarative vs. Procedural Access
    - **Declarative** (SQL, Relational Calculus): describe what to retrieve.
    - **Procedural** (Relational Algebra): define how to retrieve it.

Relational Algebra – Key Concepts

Relational Algebra describes the way we want to manipulate our data. A relational model uses operators to perform queries. Operators accept relations as their input and yield relations as their output. These operators are also known as the relational algebra.

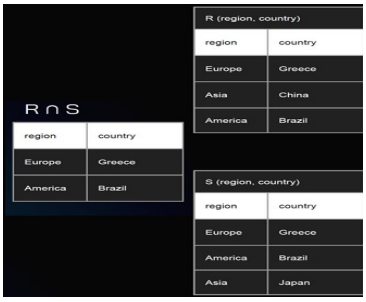
- **Selection (σ):** Filters rows (tuples) based on a specified condition and returns only those rows that satisfy the condition. Example: $\sigma_{country='USA'}$ (Authors)



- **Projection (π):** Extracts specific columns (attributes) from a relation. Example: $\pi_{name,\ country}$ (Authors)

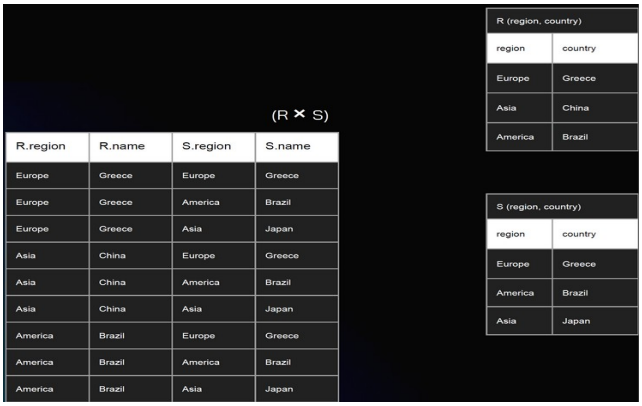
- **Union (∪):** Generates a relation containing tuples being present either on one table or both tables of the same schema (if a tuple is in both tables, then this tuple is inserted twice in the output).

- **Intersection (∩):** Generates a relation containing tuples being present in both tables.

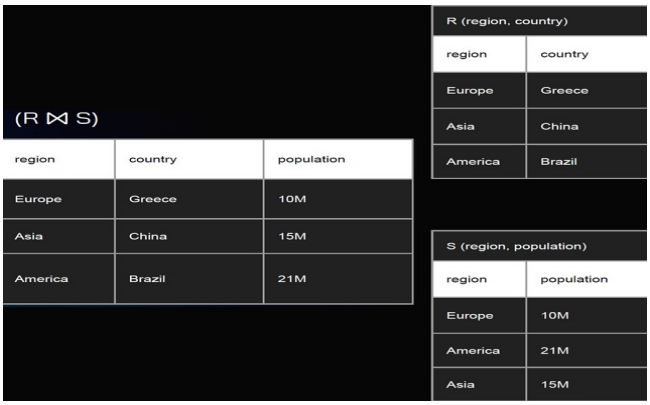


- **Difference (–):** Generates a relation containing tuples being present in the first but not in the second relation.

- **Cartesian Product (×):** Generates a relation containing the Cartesian product of the two relations.



- **Join (⋈):** Generates a relation containing the set of all combinations of tuples in both relations are equal on their common attribute names.



SQL

SQL (Structured Query Language) is the standard language used to query, manipulate, and manage data in relational databases. It described what shape the desired data should have. Explore example SQL folder here: https://github.com/GiX007/sql-nlidb-lab/tree/main/queries.
This directory contains practical SQL examples demonstrating SELECT, JOIN, GROUP BY, and more.