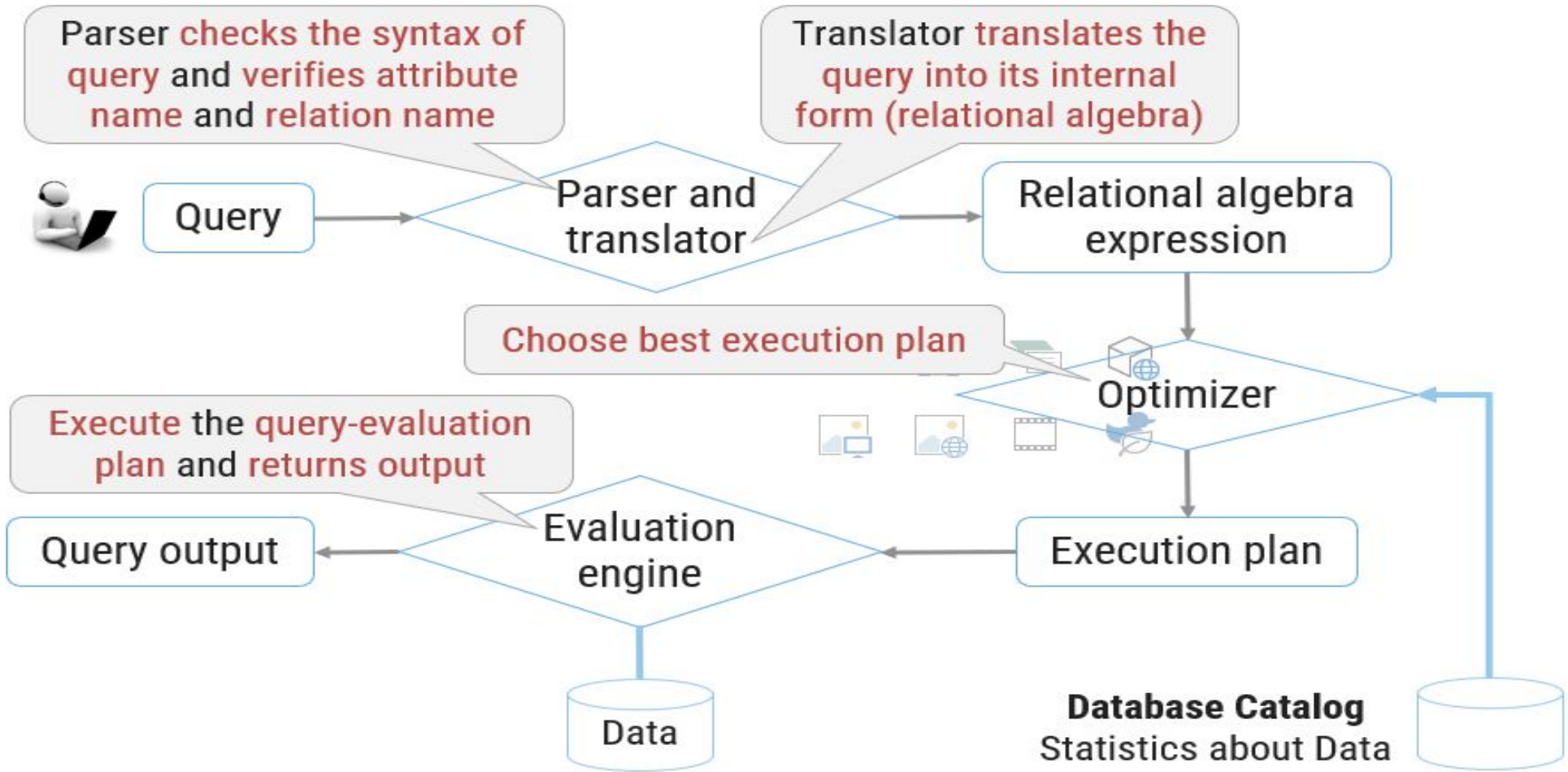
Unit 5

Query Processing and Optimization

Query Processing

- It’s a range of activities involved in extracting data from a database.
- Activities include
  - Translating high level language to the low level language
- select ENAME from employee where Empid>20
  - Execution From ,where, select

Query Processing Steps



Measure of query cost

- Cost is generally measured as **the total time required to execute a statement/query**.
- Factors contribute to time cost
  - **Disk accesses** (time to process a data request and retrieve the required data from the storage device)
  - **CPU time to execute a query**
  - **Network communication cost**
- **Disk access is the predominant (major) cost, since disk access is slow as compared to in-memory operation**.
- **Cost to write a block is greater than cost to read a block** because data is read back after being written to ensure that the write was successful.

Selection operation

- Symbol: σ (Sigma)
- Notation: $\sigma_{condition}$ (Relation)
- Operation: **Selects tuples** from a relation that **satisfy a given condition**.
- Operators: =, <>, <, >, <=, >=, Λ (AND), V (OR)

Example Display the detail of students belongs to "CE" Branch.

Student

RollNo	Name	Branch	SPI
101	Raju	CE	8
102	Mitesh	ME	9
103	Nilesh	CI	9
104	Meet	CE	9

Answer $\sigma_{Branch='CE'}$ (Student)

Output

RollNo	Name	Branch	SPI
101	Raju	CE	8
104	Meet	CE	9

Basic Algorithms for search

- A1 (Linear Search)
- A2 (Binary Search)

Linear Search

- It **scans each blocks** and **tests all records** to see whether they **satisfy the selection condition**.
  - Cost of linear search (worst case) = **br**

    br denotes number of blocks containing records from relation r
- If the **selection condition is there on a (primary) key attribute**, then **system can stop searching if the required record is found**.
  - cost of linear search (best case) = (br /2)
- If the **selection is on non (primary) key** attribute then **multiple block may contains required records**, then **the cost of scanning such blocks need to be added to the cost estimate**.
- Linear search can be applied regardless of
  - **selection condition** or
  - **ordering of records** in the file (relation)
- This algorithm is **slower than binary search** algorithm.

Binary search(A2)

- Generally, this algorithm is used if **selection is an equality comparison on the (primary) key attribute and file (relation) is ordered (sorted) on (primary) key attribute**.
- cost of binary search = **[log2(br)]**
  - br denotes number of blocks containing records from relation r
- This algorithm is **faster than linear search** algorithm.

Selection using indices

- Index structures are referred to as access paths
- Search algorithm uses the index for the scanning process
- Store the tuples in the sorted order
  - A3(primary index, equality on key)
  - A4(primary index, equality on nonkey)
  - A5(secondary index, equality)

Selection Involving Comparisons

- A6(primary index, comparison)
- A7(secondary index, comparison)

Implementation of complex selection

- A8(conjunctive selection using one index)
- A9(conjunctive selection using composite index)
- A10(conjunctive selection by intersection of identifiers)
- A11(disjunctive selection by union of identifiers)

Join operation

- Nested-Loop join
- Block Nested join
- Indexed Nested Loop join
- Merge Join
- Hash Join
- Complex Join

1. Nested Loop Join:

In nested loop join algorithm, for each tuple in outer relation we have to compare it with all the tuples in the inner relation then only the next tuple of outer relation is considered. All pairs of tuples which satisfies the condition are added in the result of the join.

```
for each tuple t_R in T_R do
 for each tuple t_s in T_s do
  compare (t_R, t_s) if they satisfies the condition
  add them in the result of the join
 end
end
```

2. Block Nested Loop Join:

In block nested loop join, for a block of outer relation, all the tuples in that block are compared with all the tuples of the inner relation, then only next block of outer relation is considered. All pairs of tuples which satisfies the condition are added in the result of the join.

```
for each block b_R in B_R do
 for each block b_s in B_S do
  for each tuple t_R in T_R do
   for each tuple t_s in T_s do
    compare (t_R, t_s) if they satisfies the condition
    add them in the result of the join
   end
  end
 end
end
```

Index Nested loop join

- The last sentence of BNLJ above makes us to think what will happen if we have index on the columns used in join condition. When indexes are used, there is no sequential scan of records. That applies here too. When indexes are used on the columns that are used in join condition, it will not scan each records of inner table. It will directly fetch matching record. But we will have the cost for fetching the index in the index table. Indexes are useful when natural joins are joins or equijoins are used. Hence if indexes are not defined on the columns, we can create temporary indexes to run the query.

Merge Join

- Tables used in the join query may be sorted or not sorted. Sorted tables give efficient costs while joining. In this method, the column used to join both the tables is used to sort the two tables. It uses merge-sort technique to sort the records in two tables. Once the sorting is done then join condition is applied to get the result. Joining is also similar to any other joins, but if two records have same column value, then care should be taken to sort records based on all the columns of the record. This method is usually used in natural joins or equijoins. Assume that all the records can be accommodated into the memory block. The each block of records is read only once to join.

Hash join

- This method is also useful in case of natural and equijoins. We use hash function h on the joining column, to divide the records of each tables into different blocks. Assume each of these hash divided block of records fit the memory block

Evaluation of Expression

- It means evaluate one operation at a time, in an appropriate order.
- Expression can be evaluate by two ways
  - Materialized
  - pipeline

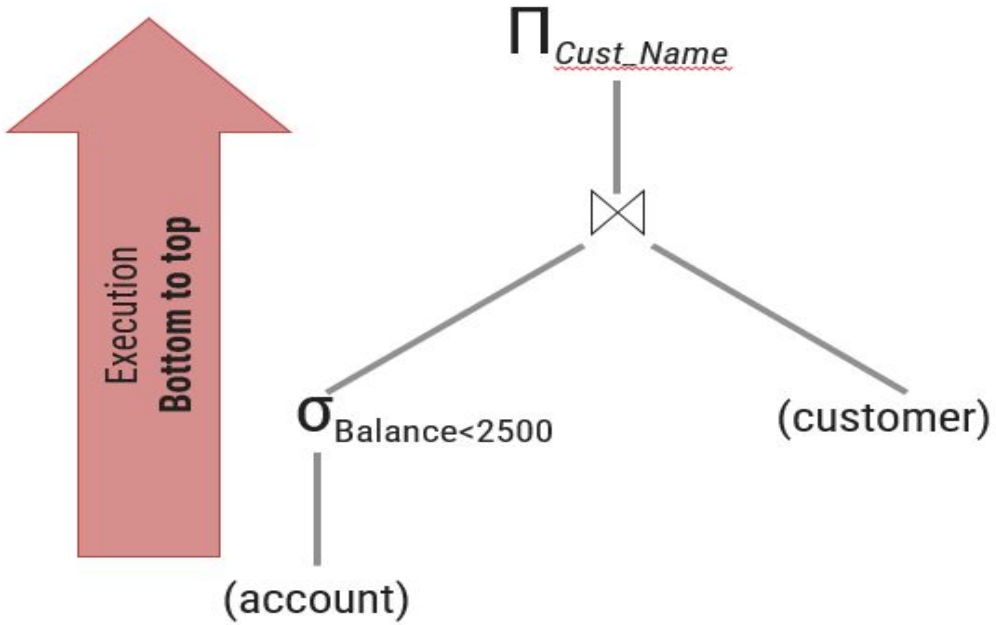
- **Materialization**: generate results of an expression whose inputs are relations or are already computed, **materialize** (store) it on disk. Repeat.

- **Pipelining**: pass on tuples to parent operations even as an operation is being executed

Materialization

- Materialization **evaluates the expression tree** of the relational algebra operation **from the bottom** and **performs the innermost or leaf-level operations first**.
- The **intermediate result of each operation is materialized (store in temporary relation)** and **becomes input for subsequent (next) operations**.
- The **cost of materialization** is the **sum of the individual operations plus the cost of writing the intermediate results to disk**.
- The problem with materialization is that
  - it **creates lots of temporary relations**
  - it **performs lots of I/O operations**

$$\Pi_{Cust\_Name} (\sigma_{Balance<2500} (account) \bowtie (customer))$$



Pipelining

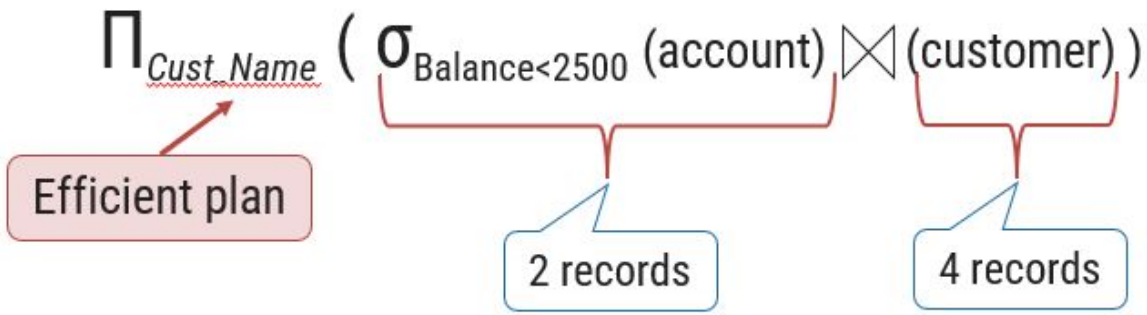
- In pipelining, **operations form a queue, and results are passed from one operation to another** as they are calculated.
- **To reduce number of intermediate temporary relations, we pass results of one operation to the next operation in the pipelines**.
- Combining operations into a pipeline **eliminates the cost of reading and writing temporary relations**.
- Pipelines can be executed in two ways:
  - **Demand driven** (System makes repeated requests for tuples from the operation at the top of pipeline)
  - **Producer driven** (Operations do not wait for request to produce tuples, but generate the tuples eagerly.)

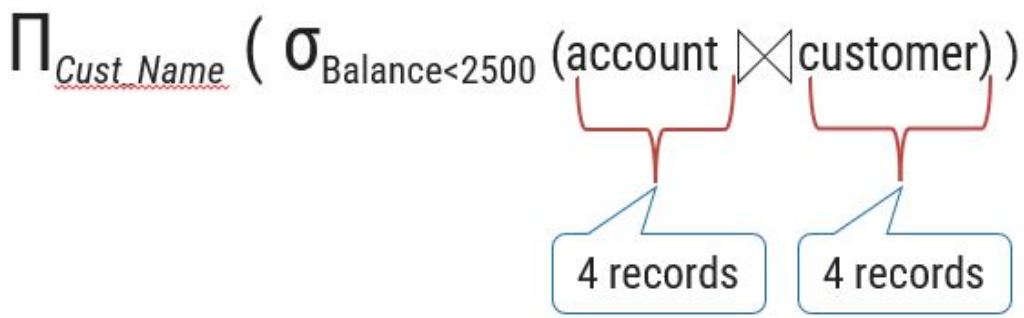
- In **demand driven** or **lazy** evaluation
  - system repeatedly requests next tuple from top level operation
  - Each operation requests next tuple from children operations as required, in order to output its next tuple
  - In between calls, operation has to maintain "**state**" so it knows what to return next
- In **producer-driven** or **eager** pipelining
  - Operators produce tuples eagerly and pass them up to their parents
    - Buffer maintained between operators, child puts tuples in buffer, parent removes tuples from buffer
    - if buffer is full, child waits till there is space in the buffer, and then generates more tuples
  - System schedules operations that have space in output buffer and can process more input tuples
- Alternative name: **pull** and **push** models of pipelining

- Implementation of demand-driven pipelining
  - Each operation is implemented as an **iterator** implementing the following operations
    - **open()**
      - E.g. file scan: initialize file scan
        - » state: pointer to beginning of file
      - E.g.merge join: sort relations;
        - » state: pointers to beginning of sorted relations
    - **next()**
      - E.g. for file scan: Output next tuple, and advance and store file pointer
      - E.g. for merge join: continue with merge from earlier state till
        next output tuple is found. Save pointers as iterator state.
    - **close()**

Query Optimization

- It is a **process of selecting the most efficient query evaluation plan from the available possible plans**.





Customer

CID	ANO	Name
C01	A01	Raj
C02	A02	Meet
C03	A03	Jay
C04	A04	Ram

Account

ANO	Balance
A01	3000
A02	1000
A03	2000
A04	4000

Approaches to Query Optimization

- Exhaustive Search Optimization
  - **Generates all possible query plans** and then the **best plan is selected**.
  - It **provides best solution**.
- Heuristic Based Optimization
  - Heuristic **based optimization uses rule-based optimization approaches** for query optimization.
  - **Performs select and project operations before join operations**. This is done by moving the select and project operations down the query tree. This reduces the number of tuples available for join.
  - **Avoid cross-product operation** because they result in very large-sized intermediate tables.
  - This **algorithms do not necessarily produce the best query plan**.

Customer

CID	ANO	Name	Balance
C01	1	Raj	3000
C02	2	Meet	1000
C03	3	Jay	2000
C04	4	Ram	4000

Equivalence Rules

1. Conjunctive selection operations can be deconstructed into a sequence of individual selections.

$$\sigma_{\theta_1 \wedge \theta_2}(E) = \sigma_{\theta_1}(\sigma_{\theta_2}(E))$$

2. Selection operations are commutative.

$$\sigma_{\theta_1}(\sigma_{\theta_2}(E)) = \sigma_{\theta_2}(\sigma_{\theta_1}(E))$$

3. Only the last in a sequence of projection operations is needed, the others can be omitted.

$$\Pi_{L_1}(\Pi_{L_2}(\ldots(\Pi_{Ln}(E))\ldots)) = \Pi_{L_1}(E)$$

4. Selections can be combined with Cartesian products and theta joins.
   a. $\sigma_{\theta}(E_1 \times E_2) = E_1 \bowtie_{\theta} E_2$
   b. $\sigma_{\theta 1}(E_1 \bowtie_{\theta 2} E_2) = E_1 \bowtie_{\theta 1 \wedge \theta 2} E_2$

5. Theta-join operations (and natural joins) are commutative.

$$E_1 \bowtie_\theta E_2 = E_2 \bowtie_\theta E_1$$

6. (a) Natural join operations are associative:

$$(E_1 \bowtie E_2) \bowtie E_3 = E_1 \bowtie (E_2 \bowtie E_3)$$

(b) Theta joins are associative in the following manner:

$$(E_1 \bowtie_{\theta 1} E_2) \bowtie_{\theta 2 \wedge \theta 3} E_3 = E_1 \bowtie_{\theta 1 \wedge \theta 3} (E_2 \bowtie_{\theta 2} E_3)$$

where θ_2 involves attributes from only E_2 and E_3.

7. The selection operation distributes over the theta join operation under the following two conditions:
   (a) When all the attributes in θ_0 involve only the attributes of one of the expressions (E_1) being joined.

$$\sigma_{\theta 0}(E_1 \bowtie_\theta E_2) = (\sigma_{\theta 0}(E_1)) \bowtie_\theta E_2$$

   (b) When θ_1 involves only the attributes of E_1 and θ_2 involves only the attributes of E_2.

$$\sigma_{\theta 1 \wedge \theta 2}(E_1 \bowtie_\theta E_2) = (\sigma_{\theta 1}(E_1)) \bowtie_\theta (\sigma_{\theta 2}(E_2))$$

8. The projection operation distributes over the theta join operation as follows:

(a) if θ involves only attributes from $L_1 \cup L_2$:

$$\Pi_{L_1 \cup L_2}(E_1 \bowtie_\theta E_2) = (\Pi_{L_1}(E_1)) \bowtie_\theta (\Pi_{L_2}(E_2))$$

(b) Consider a join $E_1 \bowtie_\theta E_2$.

- Let L_1 and L_2 be sets of attributes from E_1 and E_2, respectively.
- Let L_3 be attributes of E_1 that are involved in join condition θ, but are not in $L_1 \cup L_2$, and
- let L_4 be attributes of E_2 that are involved in join condition θ, but are not in $L_1 \cup L_2$.

$$\Pi_{L_1 \cup L_2}(E_1 \bowtie_\theta E_2) = \Pi_{L_1 \cup L_2}((\Pi_{L_1 \cup L_3}(E_1)) \bowtie_\theta (\Pi_{L_2 \cup L_4}(E_2)))$$

9. The set operations union and intersection are commutative

$$E_1 \cup E_2 = E_2 \cup E_1$$
$$E_1 \cap E_2 = E_2 \cap E_1$$

- (set difference is not commutative).

10. Set union and intersection are associative.

$$(E_1 \cup E_2) \cup E_3 = E_1 \cup (E_2 \cup E_3)$$
$$(E_1 \cap E_2) \cap E_3 = E_1 \cap (E_2 \cap E_3)$$

11. The selection operation distributes over $\cup$, $\cap$ and $-$.

$$\sigma_\theta (E_1 - E_2) = \sigma_\theta (E_1) - \sigma_\theta(E_2)$$

and similarly for $\cup$ and $\cap$ in place of $-$

Also: $$\sigma_\theta (E_1 - E_2) = \sigma_\theta(E_1) - E_2$$

and similarly for $\cap$ in place of $-$, but not for $\cup$

12. The projection operation distributes over union

$$\Pi_L(E_1 \cup E_2) = (\Pi_L(E_1)) \cup (\Pi_L(E_2))$$

Branch-schema = (*branch-name*, *branch-city*, *assets*)

Account-schema = (*account-number*, *branch-name*, *balance*)

Depositor-schema = (*customer-name*, *account-number*)

Sorting

- Several of the relational operations, such as **joins, can be implemented efficiently if the input relations are first sorted**.
- We can **sort a relation by building an index on the relation** and then using that index to read the relation in sorted order.
- Such a process orders the relation only logically rather than physically.
- Hence reading of tuples in the sorted order may lead to disk access for each record, which can be very expensive.
- So it is desirable to order the records physically.
- **Sorting of relation that fit into main memory**, standard sorting techniques such as **quick-sort can be used**.
- **Sorting of relations that do not fit in main memory is called external sorting**.
- Most commonly used **algorithm** for this type of sorting is **external sort merge algorithm**.

Merge Sort(Algorithm)

- Let M denote memory size (in pages).

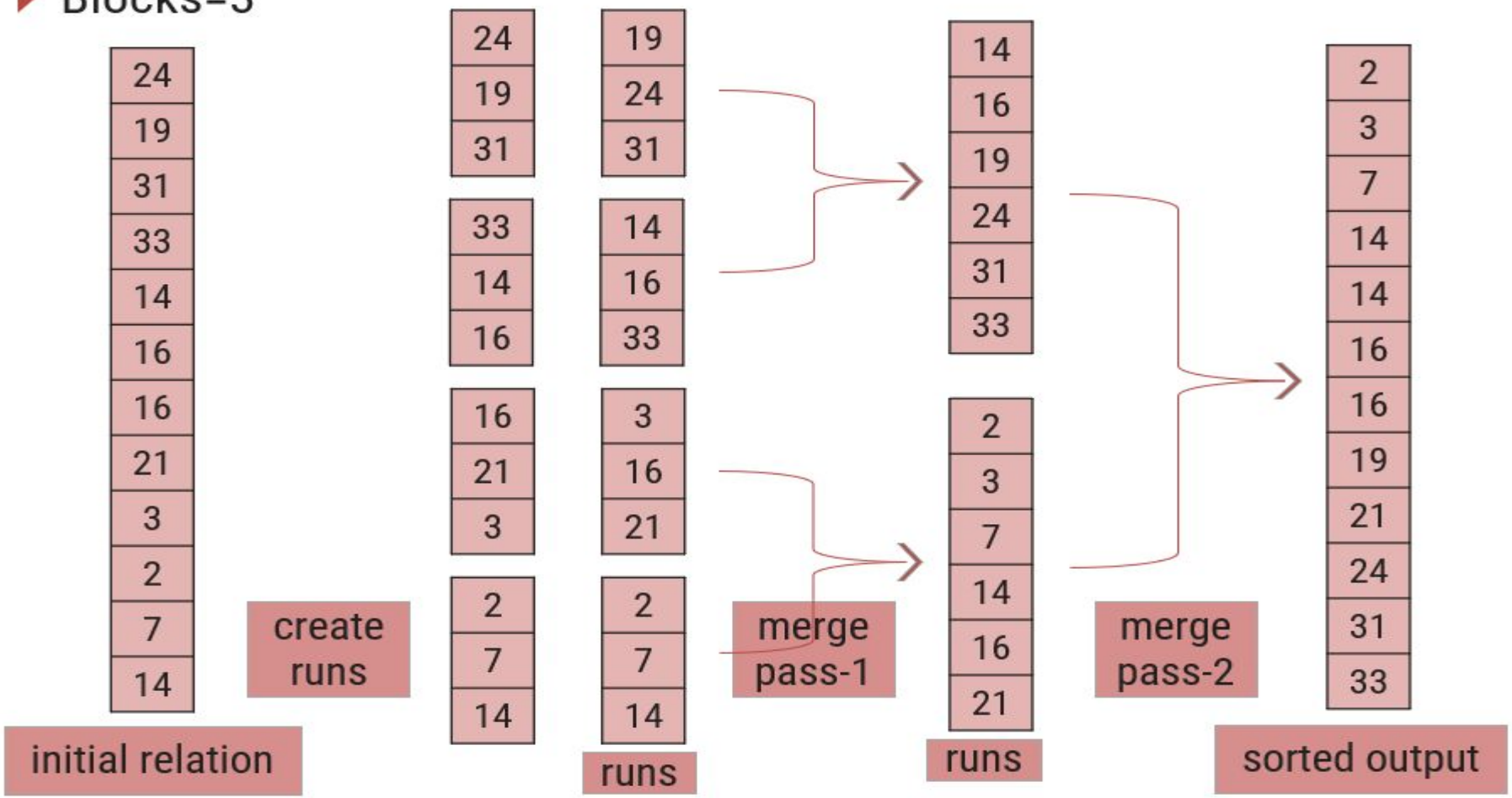
1. Create sorted runs. Let i be 0 initially.
   - Repeatedly do the following till the end of the relation:
     1) Read M blocks of relation into memory
     2) Sort the in-memory blocks
     3) Write sorted data to run Ri; then increment i.
   - Let the final value of i be N
2. Merge the runs (N-way merge). We assume (for now) that N < M.
   - Use N blocks of memory to buffer input runs, and 1 block to buffer output. Read the first block of each run into its buffer page
   - repeat

     Select the first record (in sort order) among all buffer pages

     Write the record to the output buffer. If the output buffer is full write it to disk.

     Delete the record from its input buffer page.
     - If the buffer page becomes empty then read the next block (if any) of the run into the buffer.
   - until all input buffer pages are empty:

- If N ≥ M, several merge passes are required.
  - In each pass, contiguous groups of M - 1 runs are merged.
  - A pass reduces the number of runs by a factor of M -1, and creates runs longer by the same factor.
    - E.g. If M=11, and there are 90 runs, one pass reduces the number of runs to 9, each 10 times the size of the initial runs
  - Repeated passes are performed till all runs have been merged into one.
    - Blocks=3



Questions asked in GTU

1. Explain query processing steps. **OR** Discuss various steps of query processing with proper diagram.
2. Explain Heuristics in optimization.
3. Explain the purpose of sorting with example with reference to query optimization.
4. Explain the measures of finding out the cost of a query in query processing.