COMP9120

Week 11: Query Processing and Evaluation

Semester 2, 2022

Professor Athman Bouguettaya
School of Computer Science

THE UNIVERSITY OF SYDNEY

Contents developed by Dr Lijun Chang

Acknowledgement of Country

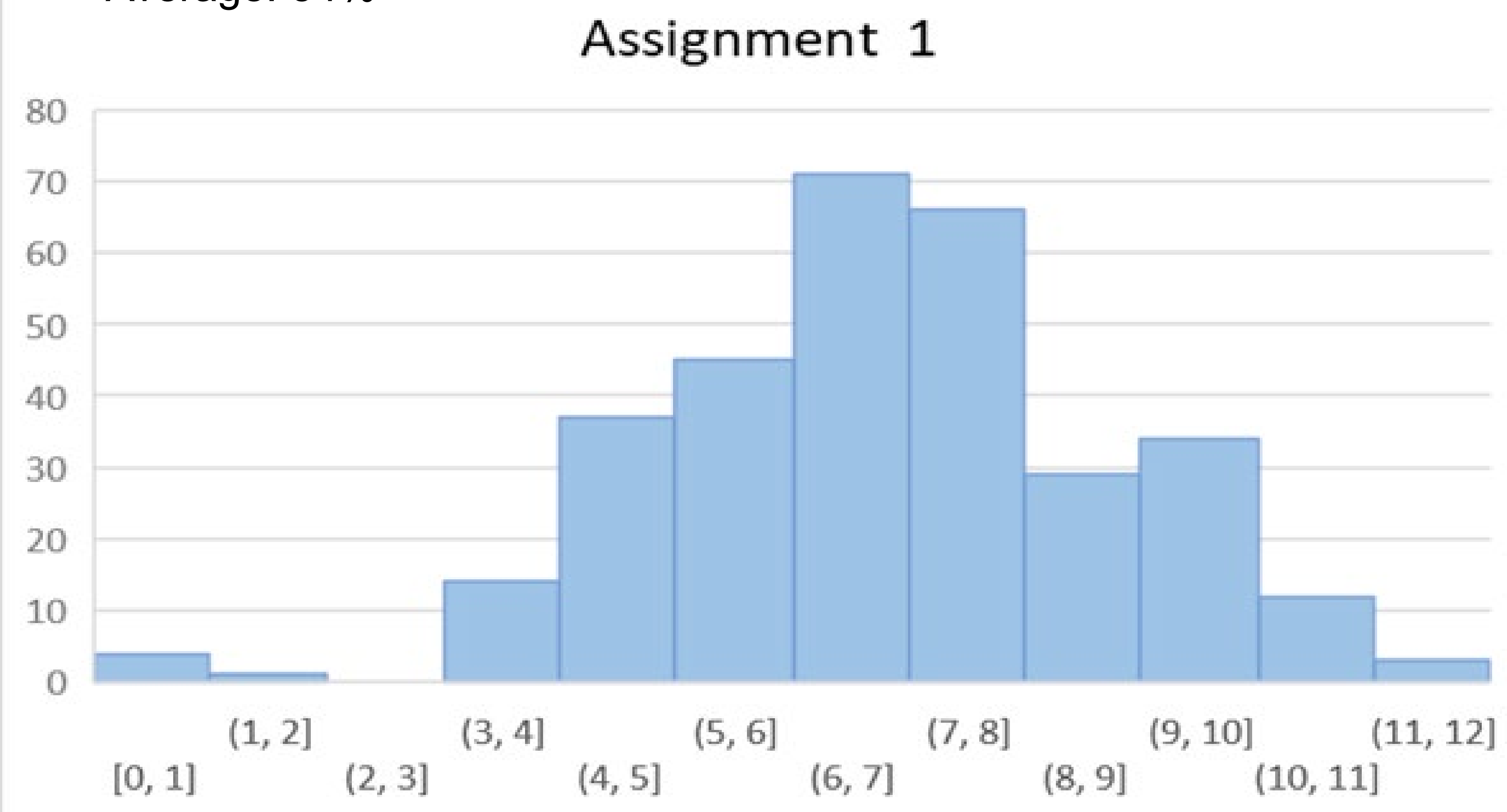
I would like to acknowledge the Traditional Owners of Australia and recognise their continuing connection to land, water and culture. I am currently on the land of the Darug people and pay my respects to their Elders, past, present and emerging.

I further acknowledge the Traditional Owners of the country on which you are on and pay respects to their Elders, past, present and future.

Copyright warning

Assignment 1 Summary

Average: 54%



Outline

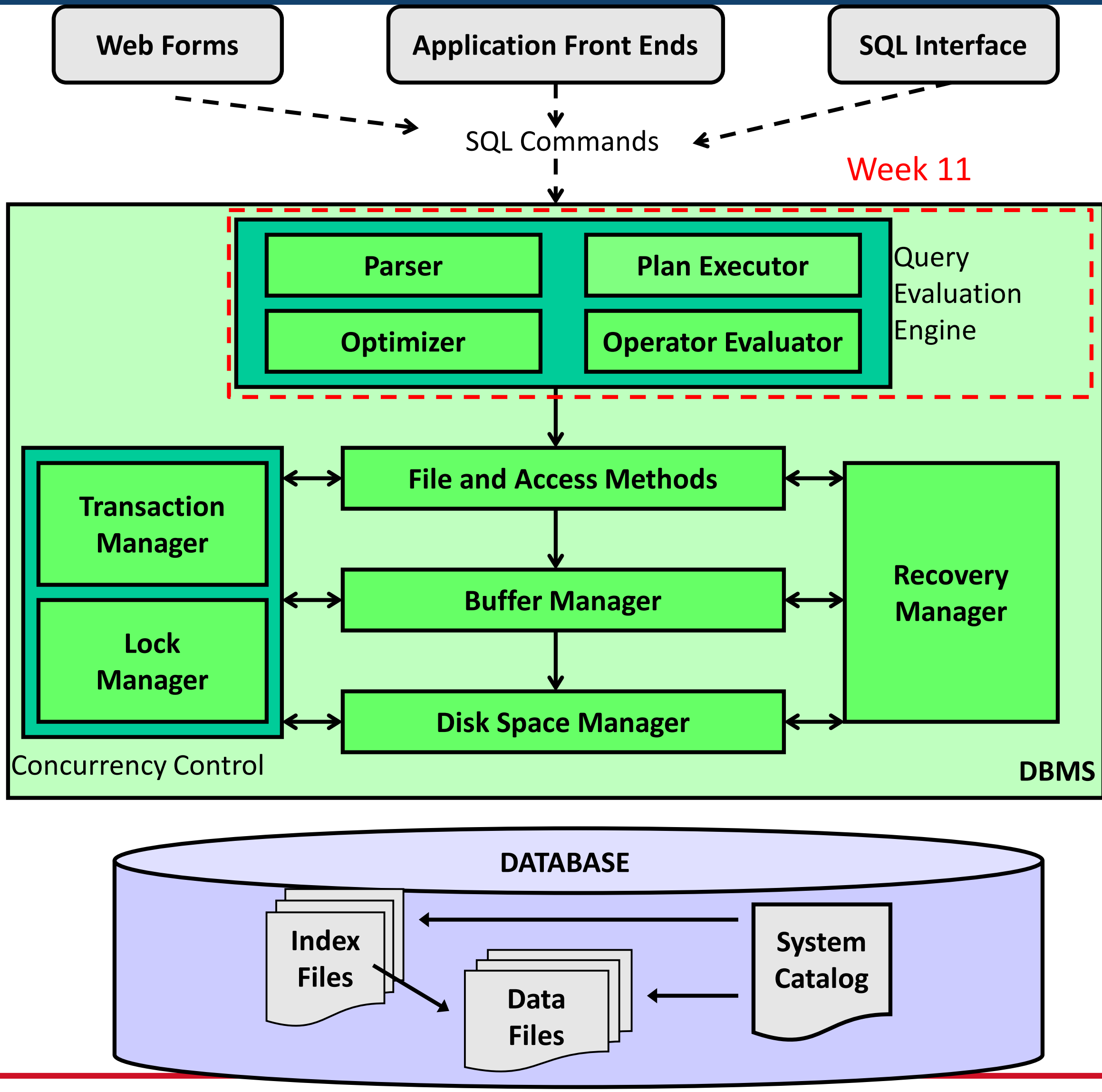
- **Basic Steps in Query Processing**
- **Query Optimization**
  - **Logical Query Plan: Heuristic-based Optimization**
  - **Physical Query Plan: Cost Estimate Optimization**
- **Query Execution**

Query processing

› Problems of processing queries:

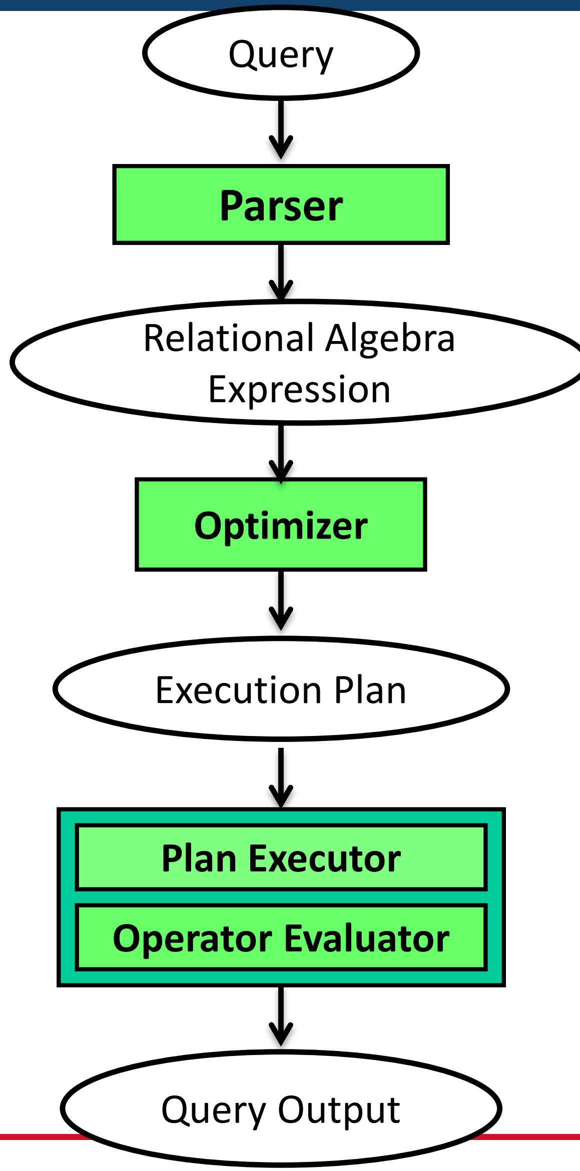
- How is a query transformed in a form understood by the DBMS? (processing phase)
- What is the best strategy to execute a query? (optimization phase)
- Criteria used to select a strategy: I/Os and to a lesser extent, cpu processing (evaluation phase)

Internal Structure of a DBMS



Basic Steps in Query Processing

- Step 1: Parsing and Translation
  - Check for syntactic and semantic errors
  - Translate the SQL query into relational algebra.
  - Query Rewriter (e.g. views) is replaced with actual sub-query on relations
- Step 2: Query Optimization
  - Amongst all equivalent query-evaluation plans choose the one with the lowest cost.
    - Use heuristics to optimize at the relational algebra level
  - Select a query execution strategy based on cost estimate
- Step 3: Query Execution
  - Depends on available storage structure and access methods



Relational model

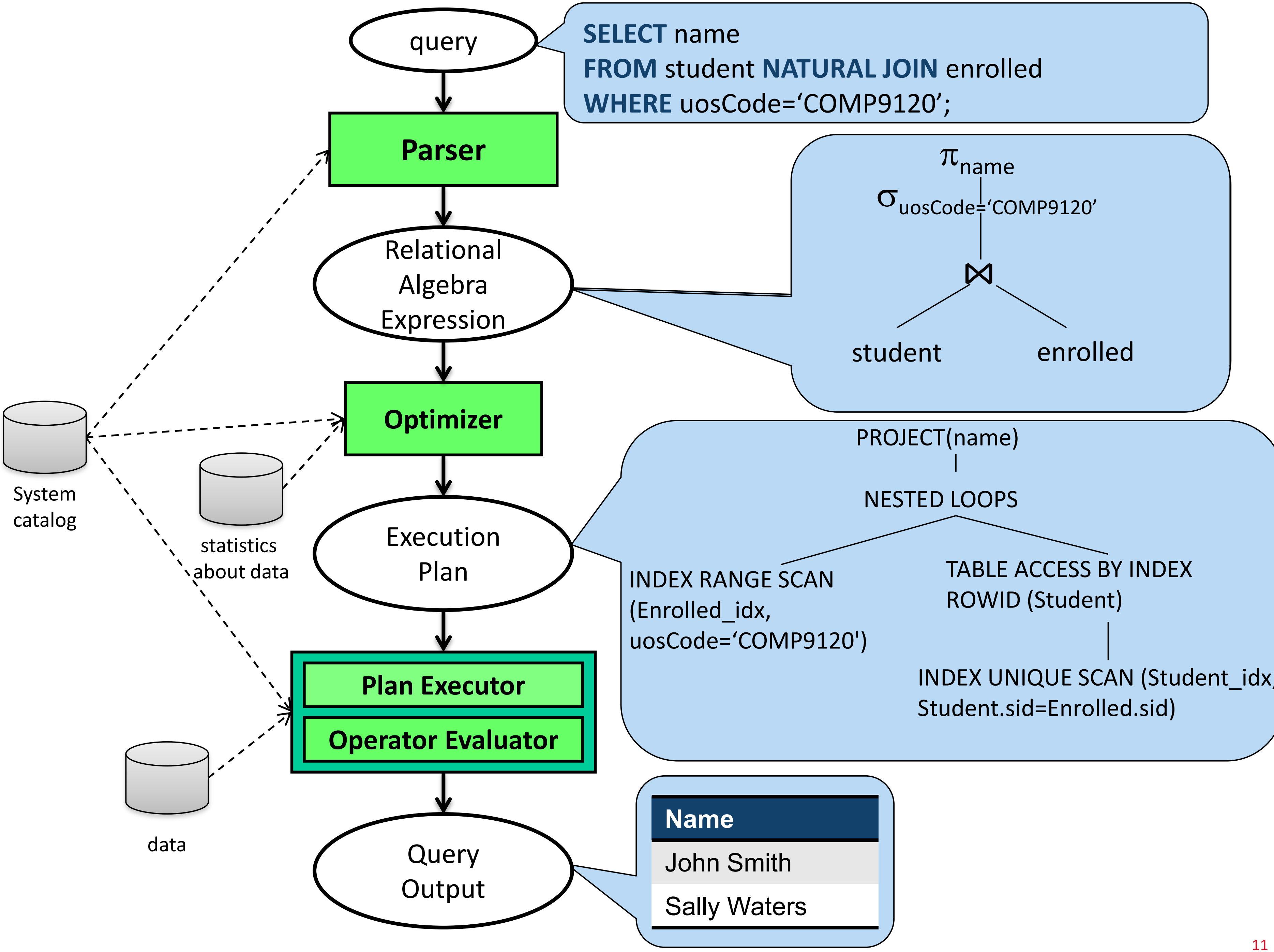
Relational model:

- Unlike early database models (e.g., hierarchical/network) where optimization was mostly left to the application programmer, relational models are based on the declarative (SQL queries - user interface) and equivalent algebraic (execution) models. These lend themselves to computer based optimization.
- Provides more abstraction (through the declarative model) to make databases more usable by the general users.
- High level abstraction also makes it hard for non-expert users to optimize queries.

Query processing and optimization: main steps

Output of each phase:

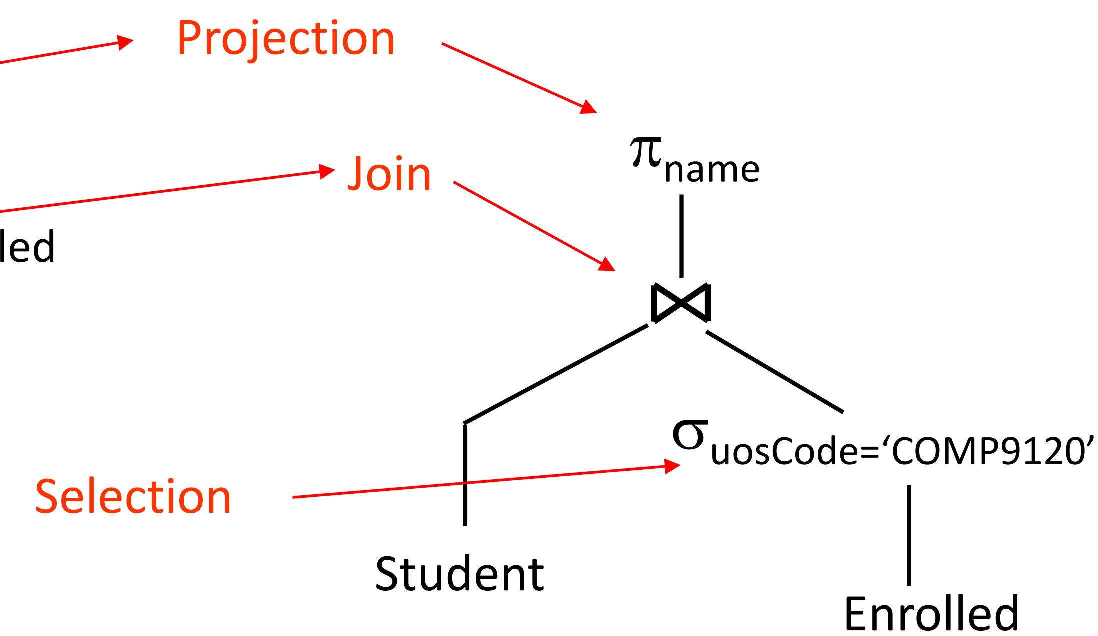
- Step 1: Parser: A parse tree consisting of a SFW (Select-From-Where) expression
  - Processing: converts the parse tree into an initial **logical query plan**
- Step 2: Optimization
  - heuristics: **efficient** logical query plans
  - cost estimate: **physical query plan**
- Step 3: Query execution


query
SELECT name
FROM student NATURAL JOIN enrolled
WHERE uosCode='COMP9120';
Parser
Relational Algebra Expression
π_{name}
$\sigma_{uosCode='COMP9120'}$
⋈
student
enrolled
Optimizer
System catalog
statistics about data
Execution Plan
PROJECT(name)
NESTED LOOPS
INDEX RANGE SCAN (Enrolled_idx, uosCode='COMP9120')
TABLE ACCESS BY INDEX ROWID (Student)
INDEX UNIQUE SCAN (Student_idx, Student.sid=Enrolled.sid)
Plan Executor
Operator Evaluator
data
Query Output
Name
John Smith
Sally Waters

Step 1: Parsing and Translation

- SQL query gets translated into Relational Algebra (RA), which is represented by **a logical query plan** (also called expression tree).
  - Operators have one or more input sets and return one output set.
  - Leafs correspond to (base) tables.
- Example:

```
SELECT name
  FROM student NATURAL JOIN enrolled
 WHERE uosCode = 'COMP9120'
```



Step 2: Query Optimization

Query optimization:

Two (2) main thrusts:

- *Heuristic* rules to rearrange operations in a query tree: output is an efficient ***logical query plan***
- *Cost estimate* of different execution strategies to pick the one with minimal cost: output is a ***physical query plan***

Heuristics: *minimize* the size of intermediate results (relations) using the manipulation of *equivalent* algebraic expressions.

Cost estimate: *minimize* the number of disk I/Os.

Step 2: Query Optimization

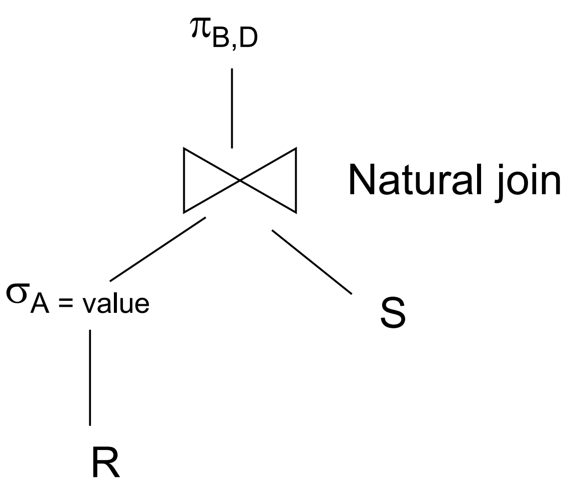
- Read in query expression tree of logical (RA) operations and generate a **Query Execution Plan**
  - Tree of relational algebra operators + choice of algorithm for each operator.
- Aim:
  - Find an optimal plan among a subset of all equivalent plans
    - Main criterion: lowest estimated I/O
  - Don’t take too long to choose an efficient plan

Step 2: Output Query Execution Plan

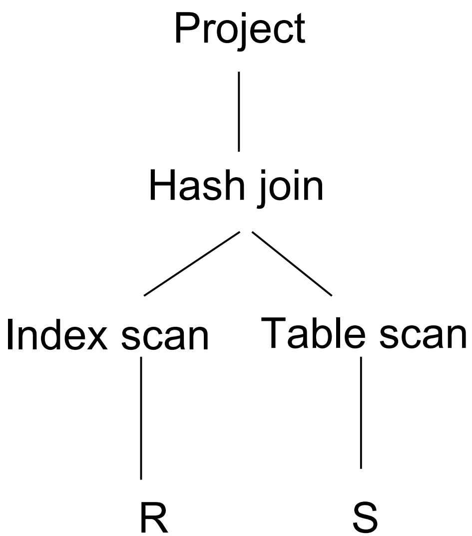
› An annotated expression tree which specifies a *detailed evaluation* strategy with *physical operators* is called an **evaluation plan** or **query plan**

- **RA operators** are logical operators
- **Physical operators** show how query is evaluated

Given a natural join between two relations R(A,B,C) and S(C,D) and the following algebraic expression: $\pi_{B,D}\ (\sigma_{A\ =\ value}(R) \infty\ S)$



Logical Plan vs.



Physical Plan

Step 2: Output a Query Execution Plan

- Multiple possible query evaluation plans for the expression tree are considered by the query optimizer

Expression tree

π_{balance}

$\sigma_{\mathit{balance}<2500}$

account

Possible query evaluation plans

Project $_{\mathit{balance}}$

Filter$_{\mathit{balance}<2500}$

Linear Scan (*account*)

Project $_{\mathit{balance}}$

Index Range Scan (*account.balance*, *balance*<2500)

Outline

- Basic Steps in Query Processing
- **Query Optimization**
  - **Logical Query Plan: Heuristic-based Optimization**
  - Physical Query Plan: Cost Estimate Optimization
- Query Execution

Equivalence of expressions

Equivalence of expressions

Note:

- We can transform any tuple calculus expression (i.e., SQL query) to an equivalent algebraic expression
- We then make sure the equivalent algebraic expression is executed efficiently
- Heuristic optimization is mostly concerned with unary operations (e.g., selection, projection)
- Strategy is to always pick a sequence of operations that would *minimize size of intermediate results*

Example

Consider the following query:

"find the assets and names of all banks which have depositors living in Sydney".
assume we have three relations: *customer, deposit*, and *branch*.

Schema:

Table 1: DEPOSIT

Branchname	*Account#*	*Customername*	*Balance*

Table 2: CUSTOMER

Customername	*Street*	*Customercity*

Table 3: BRANCH

Branchname	*Assets*	*Branchcity*

This query is equivalent to the following algebraic expression:

$$\Pi_{\text{branchname, assets}} (\sigma_{\text{customercity=Sydney}} (\text{Customer} \infty \text{Deposit} \infty \text{Branch}))$$

THE UNIVERSITY OF SYDNEY

Example

The join of the three relations may yield a large relation that may not fit in memory.

Note that:

We only need a handful of tuples to begin with (those with customercity = Sydney). Furthermore, we are only interested in two attributes (branchname and assets).

Question: Could we make the evaluation a bit more "intelligent"?

Algebraic manipulation

Answer: YES!

but how?

Using some rearrangements of the operations (algebraic manipulation)

How do we insure the **rearrangement** is **equivalent** to the **original arrangement**?

Using **knowledge** about the **rules governing algebraic operations**.

For instance: the previous expression is equivalent to:

$\Pi_{\text{branchname, assets}}((\sigma_{\text{customercity=Sydney}}(\text{Customer})) \infty \text{ Deposit } \infty \text{ Branch})$

Algebraic manipulation

Another example:

Query: "find the *assets* and *names* of all banks which have depositors living in Sydney and have a *balance* of more than $500".

This query is equivalent to the following algebraic expression:

$\Pi_{\text{branchname, assets}} (\sigma_{\text{customercity=Sydney} \ \wedge \ \text{balance} > 500} (\text{Customer} \ \infty \ \text{Deposit} \infty \text{Branch}))$

Problem: cannot do the selection on *Customer* only, because balance is an attribute of *Deposit*.

What is the solution then?

Do the selection *after* doing a join on *Customer* and *Deposit*. the resulting expression is then:

$\Pi_{\text{branchname, assets}} (\sigma_{\text{customercity=Sydney} \ \wedge \ \text{balance} > 500} (\text{Customer} \ \infty \ \text{Deposit}) \infty \text{Branch})$

The intermediate result has now been reduced.

Algebraic manipulation

Can we do even better?

The answer is YES!

but how?

Break up the selection condition into two selections and we get:

$$\Pi_{\text{branchname, assets}} (\sigma_{\text{customercity=Sydney}} (\sigma_{\text{balance > 500}} (\text{Customer} \infty \text{Deposit})) \infty \text{Branch})$$

And then move the second selection past the first join:

$$\Pi_{\text{branchname, assets}} (\sigma_{\text{customercity=Sydney}} (\text{Customer}) \infty \sigma_{\text{balance > 500}} (\text{Deposit})) \infty \text{Branch})$$

Projection optimization

Projection optimization:

Whenever possible, do a projection as soon as possible. Consider the query in the first example:

$$\Pi_{\text{branchname, assets}} ((\sigma_{\text{customercity=Sydney}} (\text{Customer}) \infty \text{Deposit}) \infty \text{Branch})$$

When we compute the subexpression:

$$(\sigma_{\text{customercity=Sydney}} (\text{Customer}) \infty \text{Deposit})$$

We would like to eliminate those attributes that will not play any role in the remaining operations.

In the example above, the attribute *branchname* is the only attribute we need in the first join. Do a projection on the subresult.

The transformation is as follows:

$$\Pi_{\text{branchname, assets}} (\Pi_{\text{branchname}} (\sigma_{\text{customercity=Sydney}} (\text{Customer}) \infty \text{Deposit}) \infty \text{Branch})$$

Rules for equivalent algebraic transformations

Heuristics based optimization consists of applying rules that yield equivalent transformations to obtain more efficient *logical query plans*:

The internal form of a query is usually implemented using a *logical query plan*.

In the previous optimization steps, we intuitively came up with *transformation* rules. What are the *formal algebraic transformation* rules that enabled us to perform the previous transformations?

Example of rules for algebraic transformations:

1. **Commutative laws** for joins:

$R1 \infty R2 = R2 \infty R1$

2. **Associative laws** for joins

$(R1 \infty R2) \infty R3 = R1 \infty (R2 \infty R3)$

3. **Cascade of projections**: if attributes A1,.....,An are a subset of B1,.....,Bn then

$\Pi_{A1,....,An}(\Pi_{B1,.....,Bn}(R)) = \Pi_{A1,....,An}(R)$

4. **Cascade of selections**

$\sigma_{\Theta 1}(\sigma_{\Theta 2}(R)) = \sigma_{\Theta 2}(\sigma_{\Theta 1}(R)) = \sigma_{\Theta 1 \wedge \Theta 2}(R)$

5. **Commuting selections and projections**: If formula Θ involves only attributes B1,.....,Bn that are a subset of A1,.....,An then

$\Pi_{A1,....,An}(\sigma_{\Theta}(R)) = \sigma_{\Theta}(\Pi_{A1,....,An}(R))$

If formula Θ involves attributes B1,.....,Bn that are not in the set A1,.....,An then

$\Pi_{A1,....,An}(\sigma_{\Theta}(R)) = \Pi_{A1,....,An}(\sigma_{\Theta}(\Pi_{A1,....,An,B1,.....,Bn}(R)))$

Outline

- Basic Steps in Query Processing
- **Query Optimization**
  - Logical Query Plan: Heuristic-based Optimization
  - **Physical Query Plan: Cost Estimate Optimization**
- Query Execution

Cost of query processing

Cost of query processing:

- Cost of every algebraic operation in terms of I/Os

Use

- Access methods available
- Data physical organization: collected facts (e.g., blocking factors)
- Using statistics (e.g., selection cardinality)

Output of the cost estimate optimization: Efficient *physical query plan*

Reminder: What are the join types?

All joins combine tuples from two tables into a new table

- **Theta join**: Tuples from R and S combine if some condition θ is true: **R ∞_θ S**
  - e.g. θ = R.age > S.minimum age
- **Equi-join**: Tuples from R and S combined if specified attributes match: **R ∞_θ S**
  - e.g: θ = R.x = S.y AND R.m=S.n
  - (special case of θ join)
- **Natural join**: Tuples from R and S combine if all attributes of same name and compatible types match in value: **R ∞ S**
  - e.g R.x=S.x AND R.m=S.m
  - (special case of equi-join)

Example: Joins

Assessment

sid	uosCode	sem	year	mark
316424328	INFO2120	S1	2012	72
305678453	INFO2120	S1	2012	86
316424328	INFO3005	S1	2010	63
305678453	COMP5138	S1	2012	94

UoSLecturer

uosCode	sem	year	lecturer
INFO2120	S1	2012	Uwe Roehm
INFO3005	S1	2010	Irena Koprinska
COMP5138	S2	2012	Bryn Jeffries

sid	mark	uosCode	sem	year	lecturer
316424328	72	INFO2120	S1	2012	Uwe Roehm
305678453	86	INFO2120	S1	2012	Uwe Roehm
316424328	63	INFO3005	S1	2010	Irena Koprinska

Example SQL:

SELECT * **FROM** Assessment **NATURAL JOIN** UoSLecturer;

RA:

Assessment $\bowtie$ UosLecturer

Joins

› Join is the most common operation in SQL queries!

- It is also usually the *most expensive operation* to execute in terms of I/Os!

In SQL:

SELECT * **FROM** Students R, Enrolled S **WHERE** R.sid=S.sid

- In algebra: $R \bowtie S$

› Note that:

- Semantically, $R \bowtie S$ is the same as $R \times S$ (cartesian product) followed by a selection
- However, the result of $R \times S$ is usually *significantly* larger than $R \bowtie S$; therefore, $R \times S$ followed by a selection is inefficient.
- Therefore, there is a need to **optimize** the join execution

Join Operator

- Several different algorithms to implement joins
  - *Nested loops* join
  - *Block-nested* loops join
  - *Indexed-nested* loops join

- Choice based on cost estimate (i.e., choose the strategy with the smallest cost)
  - *Cost metric*: # of I/Os

Short 5 mn break:

please stand up, stretch, and move around

Example Table Sizes for Cost Estimates

- We will use the following statistics:
  - $|R|$ tuples in R, stored in b_R pages
  - $|S|$ tuples in S, stored in b_S pages
    - In our examples, R refers to the relation *Students* and S refers to the relation *Enrolled*.

- Specific values: Assume
  - Number of tuples of
    - **students ($|R|$)**: 1,000
    - **enrolled ($|S|$)**: 10,000
  - Number of pages of
    - **students (b_R)**: 100
    - **enrolled (b_S)**: 400

Student $\bowtie$ Enrolled

Student			
sid	**name**	**gender**	**country**
1001	Ian	M	AUS
1002	Ha Tschi	F	ROK
1003	Grant	M	AUS

Enrolled		
sid	**uos_code**	**semester**
1001	COMP5138	2020-S2
1002	COMP5702	2020-S2
1003	COMP5138	2020-S2
1006	COMP5318	2020-S2

Nested Loops Join

- To compute the theta join $R \bowtie_{\theta} S$

for each tuple r in B_R do

for each tuple s in B_S do

if θ(r,s)=true then add <r,s> to the result

Requires I/O

R is called the *outer table*, S the *inner table* of the join

- For each tuple in the *outer* table R, we scan the entire *inner* table S.
- **Pro:** Requires no indexes and can be used with any kind of join condition.
- **Con:** Expensive since it examines every pair of tuples in the two tables.
- The number of I/Os of table R is b_R
  - each page of R is read only once
- The number of I/Os of table S is $|R|*b_S$
  - each page of S is read once for every tuple of R

Cost-Analysis: Nested Loops Join

- The estimated cost of nested loops join is

$$b_R + |R| * b_S$$

- Example:
  - students (R) as outer table: 100 + 1000 ∗ 400 = **400,100 disk I/Os**
  - enrolled (S) as outer table: 400 + 10000 ∗ 100 = **1,000,400 disk I/Os** $(b_S + |S| * b_R)$

Number of tuples of **students (*/R/*)**: 1,000
enrolled (*/S/*): 10,000
Number of pages of **students (b_R)**: 100
enrolled (b_S): 400

Block-Nested Loops Join

- Variant of nested loops join in which every page of inner table is paired with every page of outer table.
  - For each *page* of R, get each *page* of S, and write out matching pairs of tuples <r, s>, where r is in R-page and S is in S-page.

for each page B_R of R do
    for each page B_S of S do
        for each tuple r in B_R do
            for each tuple s in B_S do
                if θ(r,s)=true then output <r,s>

Requires I/O

Nested Loops Join

for each page B_R of R do
    for each tuple r in B_R do
        for each page B_S of S do
            for each tuple s in B_S do
                if θ(r,s)=true then add <r,s> to the result

Cost Analysis: Block-Nested Loops Join

- The number of I/Os of table R is b_R (each page of R is read only once)
- The number of I/Os of table S is $b_R * b_S$ (each page of S is read once for every page of R)

- Cost of block-nested loops join is

$$b_R + b_R * b_S$$

- Example:
  - students (R) as outer table: 100 + 100 ∗ 400 = **40,100 disk I/Os**
    enrolled (S) as outer table: 400 + 400 ∗ 100 = **40,400 disk I/Os**

Index-Nested Loops Join

Given an index *idx* built on the join attribute of S

for each page B_R of R do
for each tuple r in B_R do
for each tuple s in *idx(r)* do
add <r,s> to result

Requires I/O

- To use index-nested loops join, the following conditions must satisfy:
  - join is an **equi-join or natural join**, and
  - an **index is available** on the inner table's join attribute
- For each tuple *r* in the outer table *R*, use the index to look up tuples in *S* that satisfy the join condition with tuple *r*.

Cost Analysis: Index-Nested Loops Join

- For each tuple in R, we perform an index lookup on *S*.
  - Cost: $b_R + (|R| * c)$
  - Where *c* is the cost of traversing index and fetching all matching *S* tuples for one tuple of *R*
  - *c* can be estimated as the cost of a single selection on *S* using the join condition.

- If indexes are available on join attributes of both *R* and *S*, use the table with fewer tuples as the outer table.

Physical Projection Operator

- Importance of sorting for DBMS:
  - SQL queries can specify that the output be sorted (**ORDER BY**)
  - SQL operators (e.g., **selection**, **selection, join**) can be implemented efficiently if the input is **sorted**
- A simple projection operation:
  - Returning a subset of columns

```
SELECT DISTINCT sid, bid
FROM   Reserves
```

- The **expensive part** is removing **duplicates**.
  - SQL doesn't remove duplicates unless the keyword **DISTINCT** is specified in a query.
- ***Sorting Approach***: Sort on <sid, bid> and remove duplicates.

Unsorted data:

10 13

10 20 5 30 80 73 45

Sorted data:

10 10 20 30 40 50 60 70

10

Why (External) Sorting?

- For small tables that fit in memory, techniques like *QuickSort* may be used.
- Challenge: e.g., sort 10GB of data with 4GB of RAM…

  => **external merge-sort**

External Merge Sort Algorithm

Let *B* denote memory size (in pages).

1. Create sorted ***runs***. (A run is a sorted subset of records)

   Let *i* be 0 initially. **Repeatedly** do the following till the end of the file:

   (a) Read *B* pages of records from disk into memory

   (b) Sort the in-memory pages

   (c) Write sorted data to run R_i; increment *i by 1*.

   Let the final value of *i* be *m* ($= \lceil N / B \rceil$); there are *m* sorted runs. N is the size of the file.

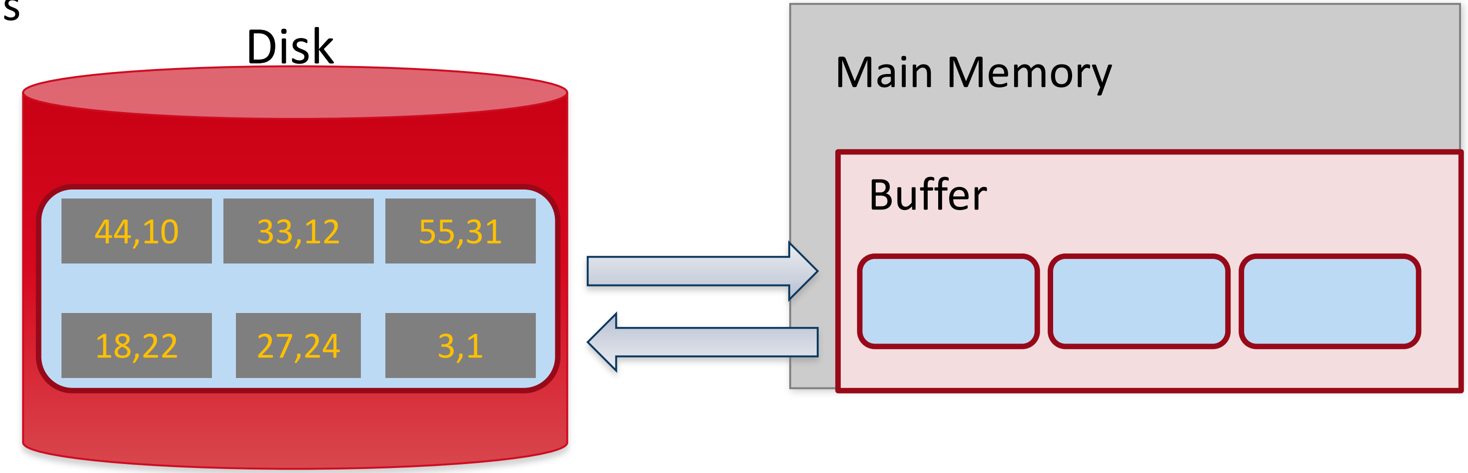
2. Merge each contiguous group of *B-1* runs into *1* run: (*B-1*)-way merge.

3. After each merge pass, the number of runs is reduced by a factor of B-1. If $m \geq B$, several merge passes are required. The number of passes (including the initial sorting pass) for the multiway merging is $\lceil \log_{(b-1)} (N/B) \rceil + 1$

Example Step 1: Create Sorted Runs

Example:

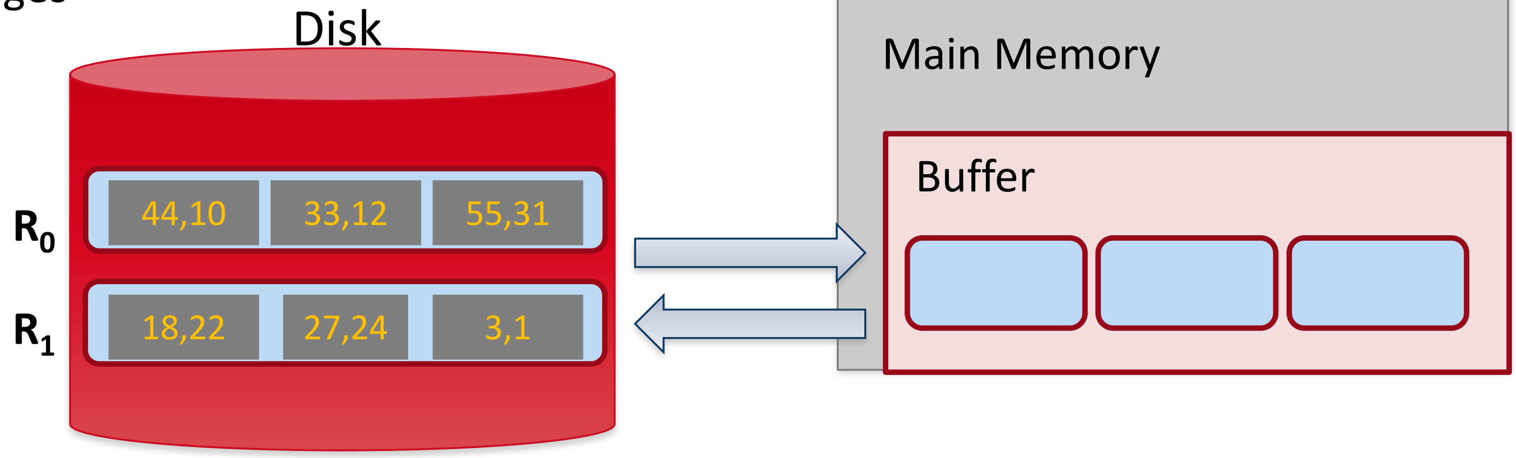
- A file consists of 6 pages
- 3 buffer pages



1. Split into runs small enough to **sort in memory**

Example Step 1: Create Sorted Runs

Example:
- A file consists of 6 pages
- 3 buffer pages

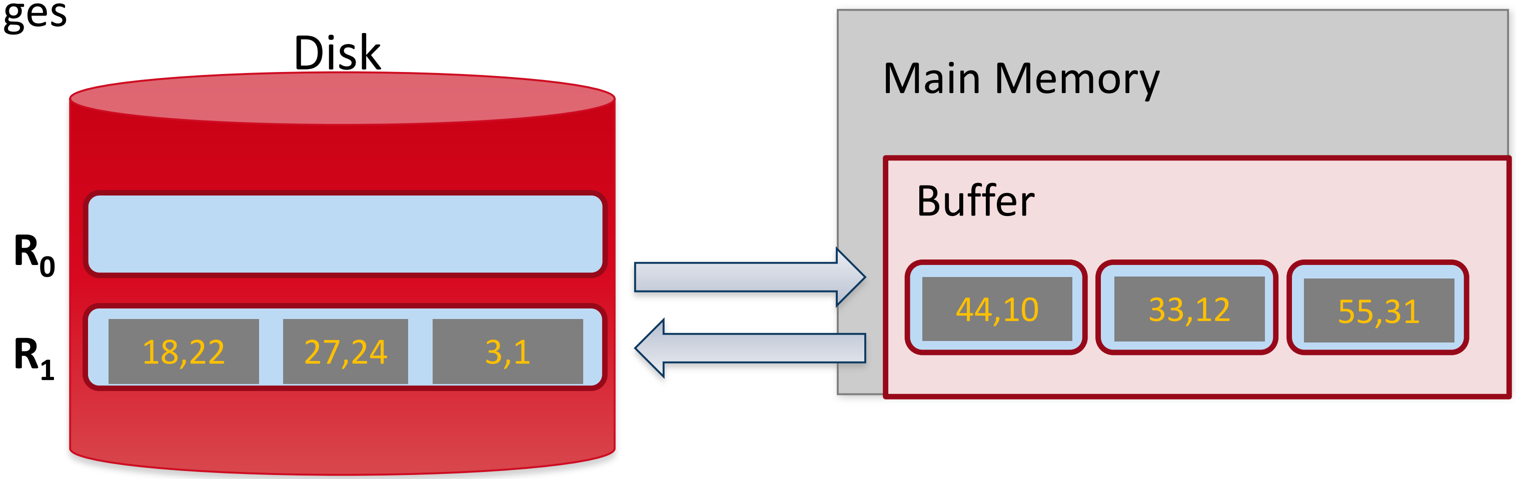


1. Split into runs small enough to **sort in memory**

Example Step 1: Create Sorted Runs

Example:

- A file consists of 6 pages
- 3 buffer pages

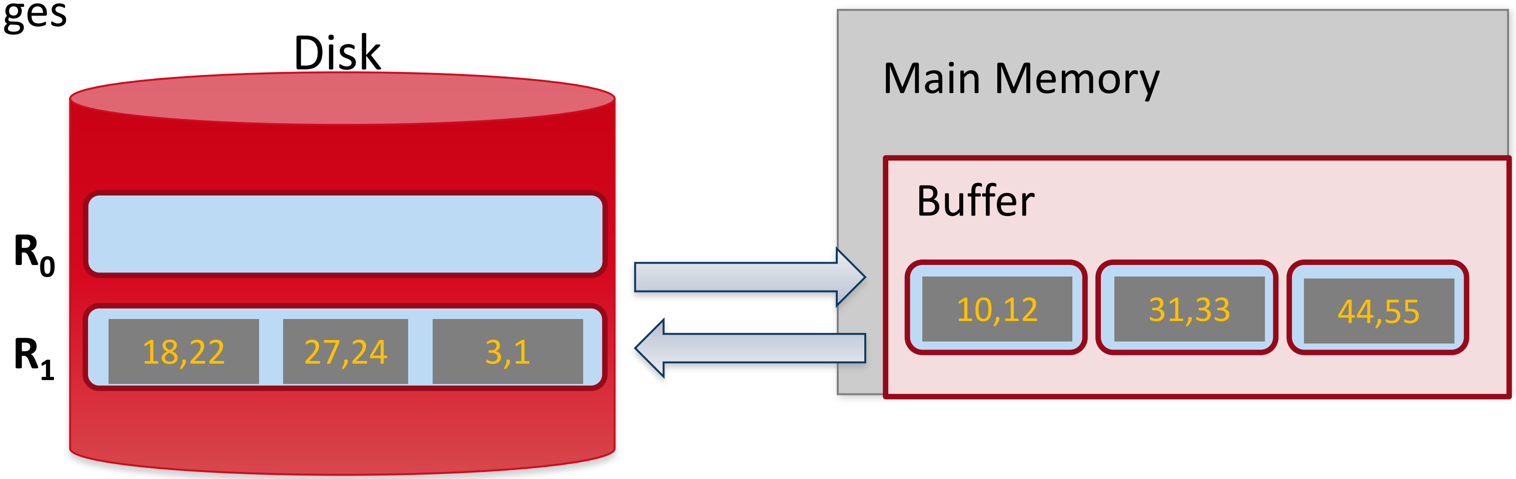


2. Load run R_0 into main memory

Example Step 1: Create Sorted Runs

Example:

- A file consists of 6 pages
- 3 buffer pages

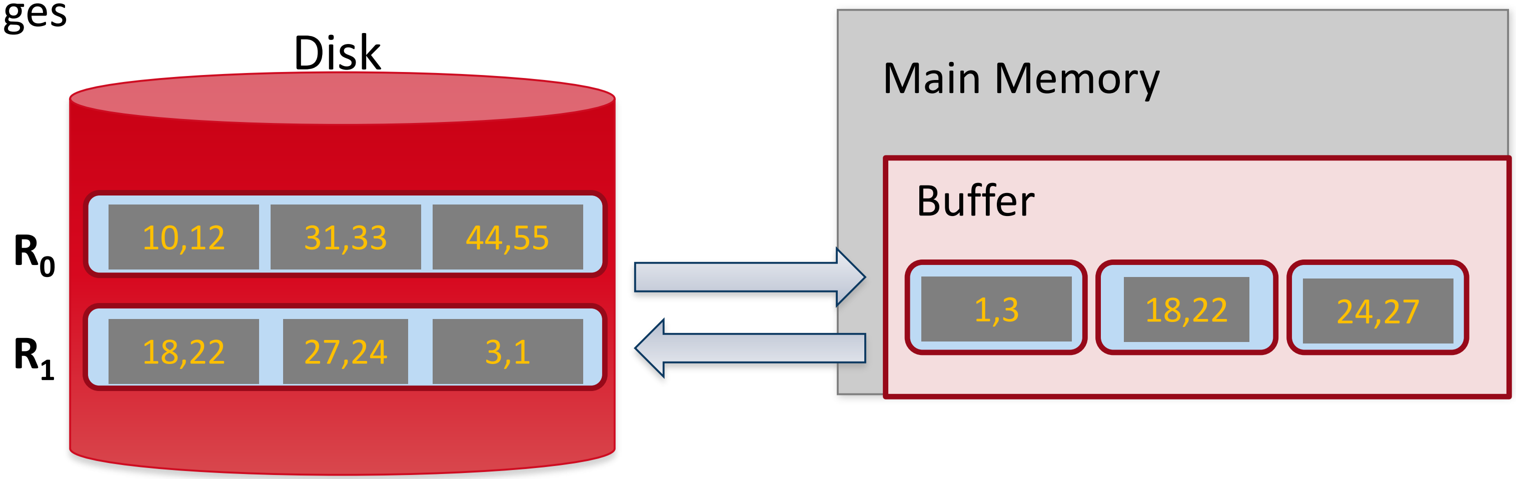


3. Sort run R_0 in main memory, and write back R_0 to disk

Example Step 1: Create Sorted Runs

Example:

- A file consists of 6 pages
- 3 buffer pages

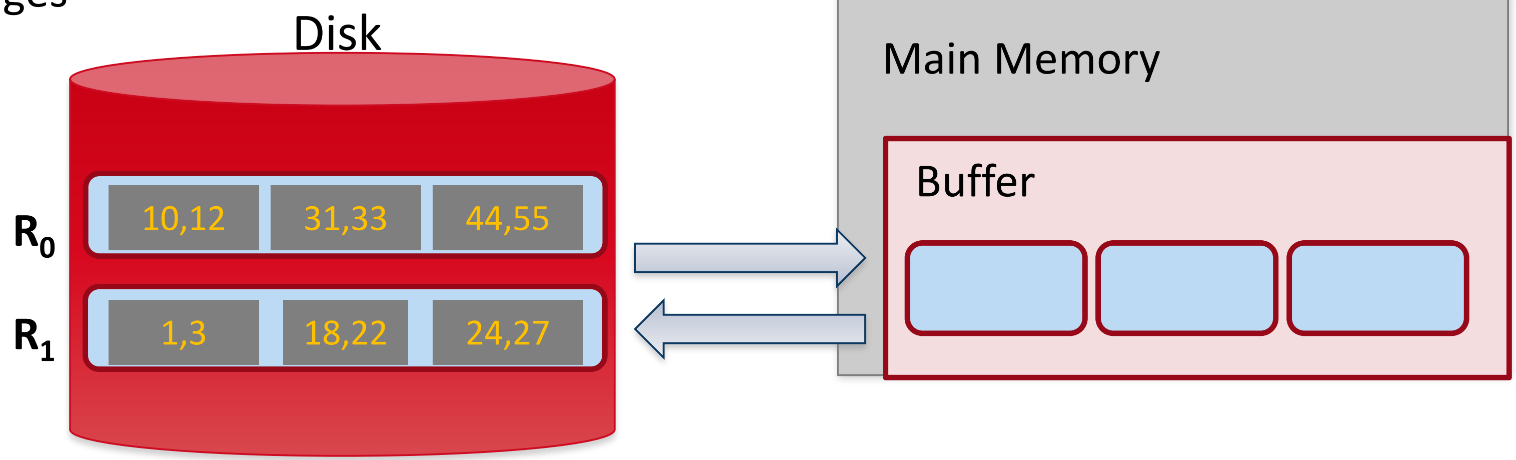


4. Similarly, load R_1 into main memory, sort it, and write it back to disk

Example Step 1: Create Sorted Runs

Example:

- A file consists of 6 pages
- 3 buffer pages



5. Now, we have sorted runs, and we next run the second step of the external merge sort algorithm

External Merge Sort Algorithm

Let B denote memory size (in pages).

1. Create sorted ***runs***. (A run is a sorted subset of records)

   Let i be 0 initially. **Repeatedly** do the following till the end of the file:

   (a) Read B pages of records from disk into memory

   (b) Sort the in-memory blocks

   (c) Write sorted data to run i; increment i *by 1*.

   Let the final value of i be m (= $\lceil N / B \rceil$); there are m sorted runs. N is the size of the file.

2. Merge each contiguous group of *B-1* runs into *1* run: (*B-1*)-way merge. Use *B-1* pages of memory to buffer input runs, and 1 page to buffer output.
   Read the first page of each run into its buffer page

   i. repeat

   1. Select the first record (in sort order) among all input buffer pages
   2. Write the record to the output buffer. If output is full, write it to disk
   3. **If** this is the last record of the input buffer page
      then read the next page (if any) of the run into the buffer

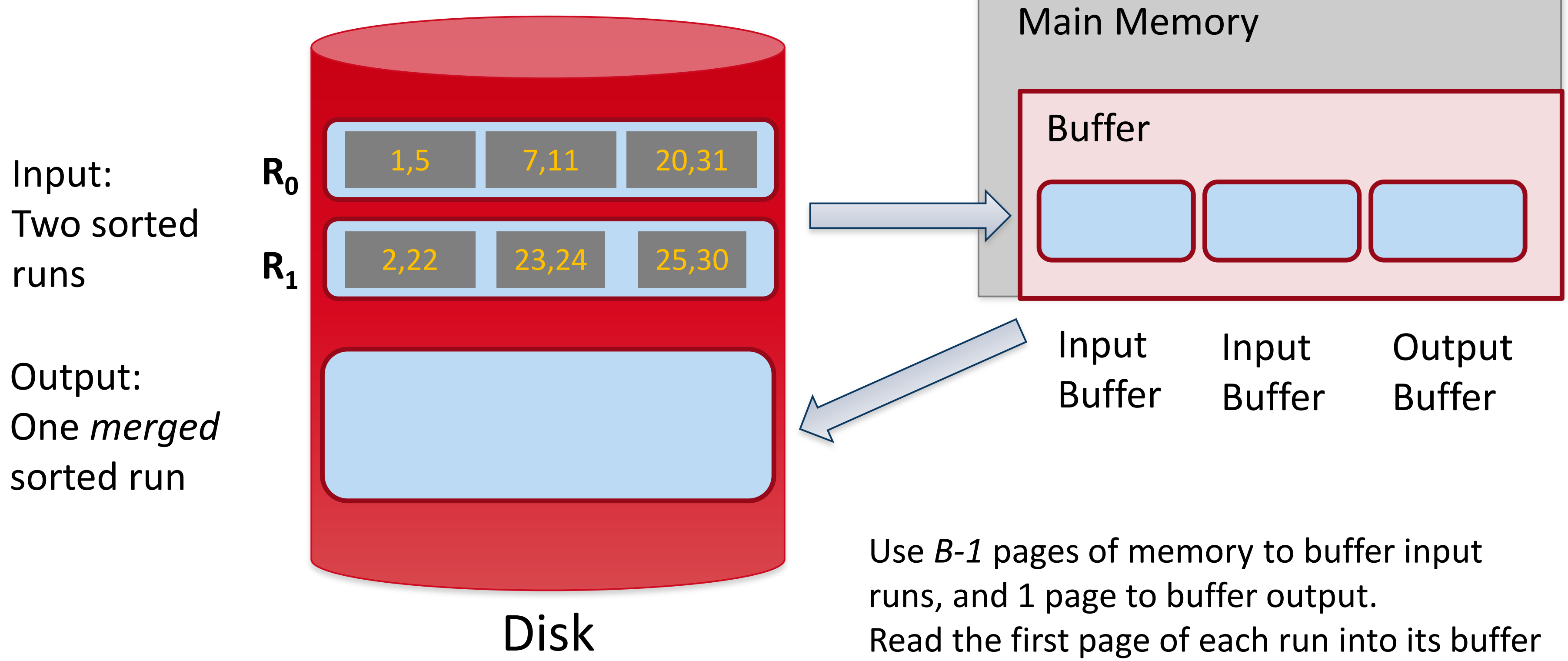
   until all input buffer pages are empty:

3. After each merge pass, the number of runs is reduced by a factor of B-1. If $m \geq B$, several merge passes are required. The number of passes (including the initial sorting pass) for the multiway merging is $\lceil \log_{(b-1)} (N/B) \rceil + 1$

Example Step-2: Merge Sorted Runs

Example:

- Each input run consists of three pages



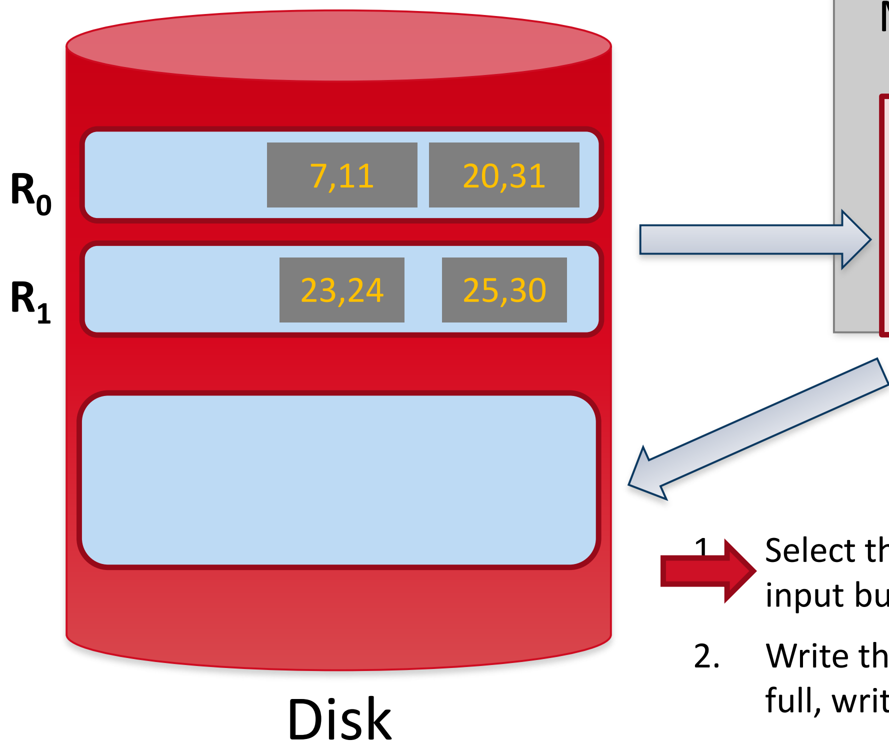
Use *B-1* pages of memory to buffer input runs, and 1 page to buffer output.
Read the first page of each run into its buffer page

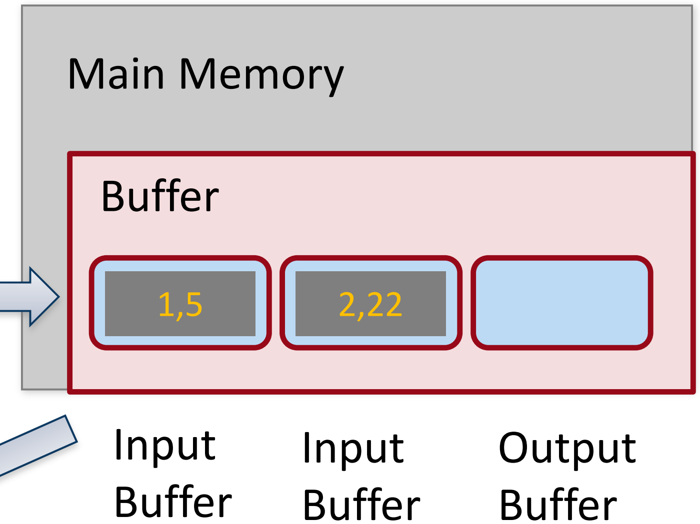
Example Step-2: Merge Sorted Runs

Example:
- Each input run consists of three pages

Input:
Two sorted runs

Output:
One *merged* sorted run





1. Select the first record (in sort order) among all input buffer pages
2. Write the record to the output buffer. If output is full, write it to disk.
3. **If** this is the last record of the input buffer page **then** read the next page (if any) of the run into the buffer.

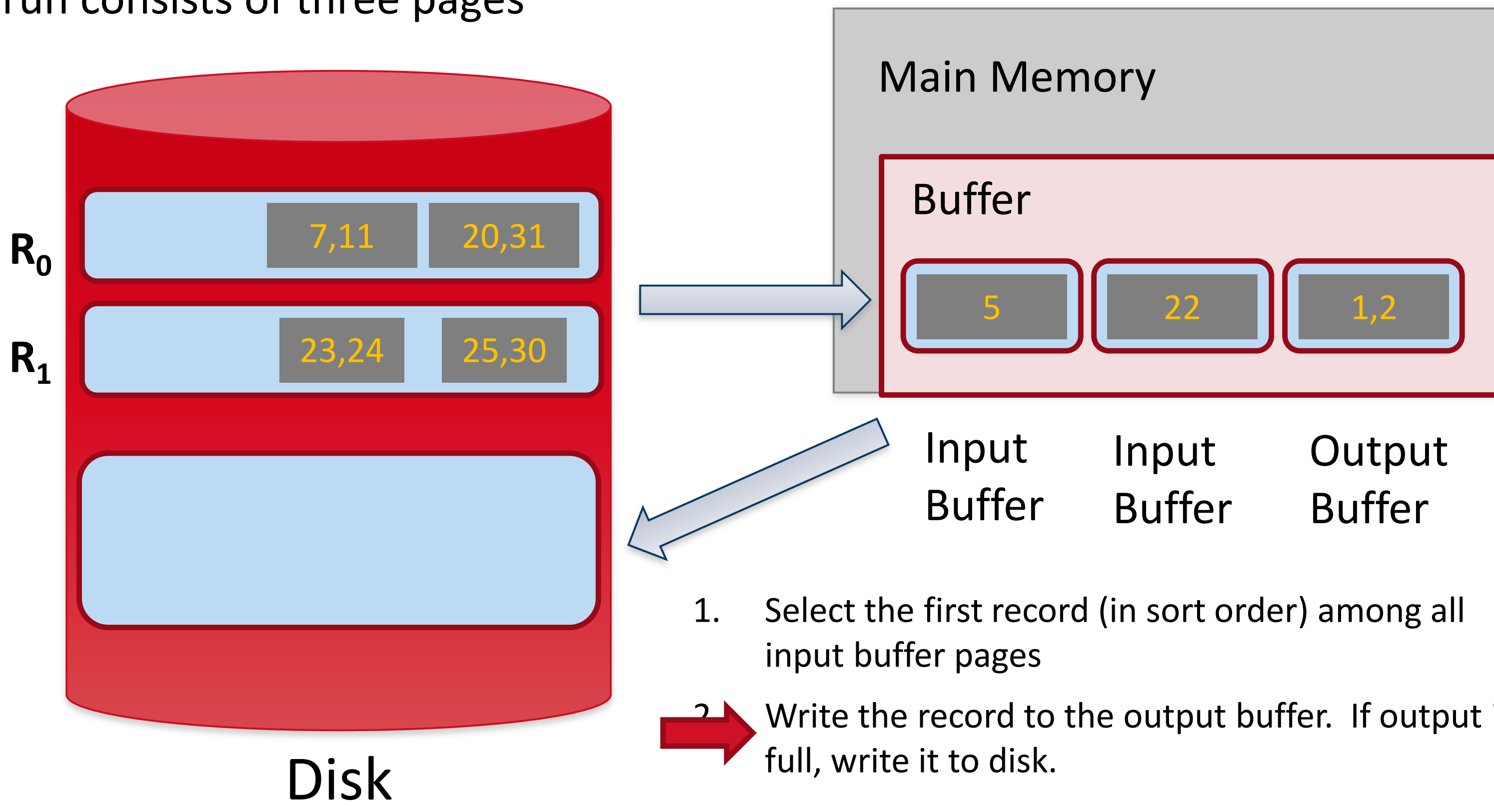
Example Step-2: Merge Sorted Runs

Example:

- Each input run consists of three pages

Input:
Two sorted runs

Output:
One *merged* sorted run

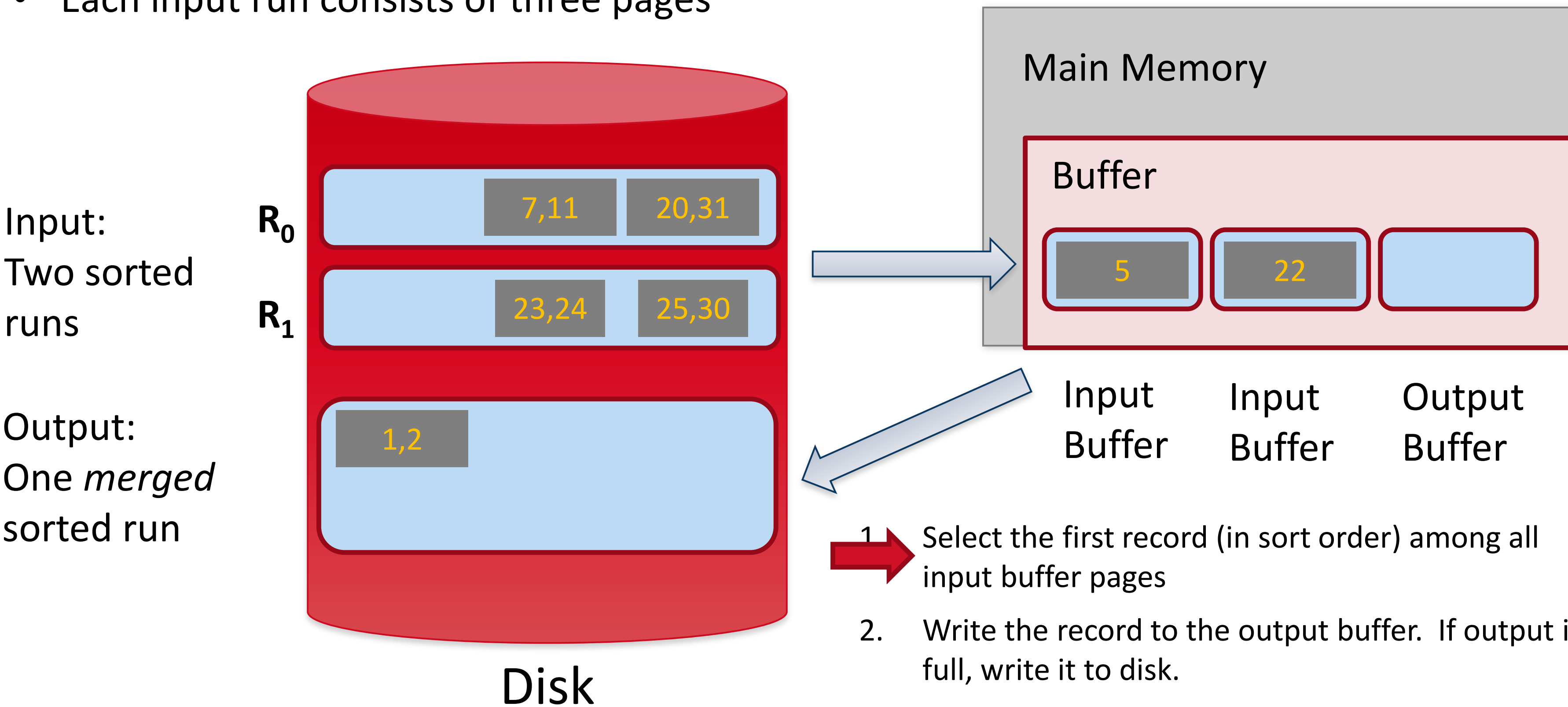


1. Select the first record (in sort order) among all input buffer pages
2. Write the record to the output buffer. If output is full, write it to disk.
3. **If** this is the last record of the input buffer page **then** read the next page (if any) of the run into the buffer.

Example Step-2: Merge Sorted Runs

Example:

- Each input run consists of three pages



1. Select the first record (in sort order) among all input buffer pages
2. Write the record to the output buffer. If output is full, write it to disk.
3. **If** this is the last record of the input buffer page **then** read the next page (if any) of the run into the buffer.

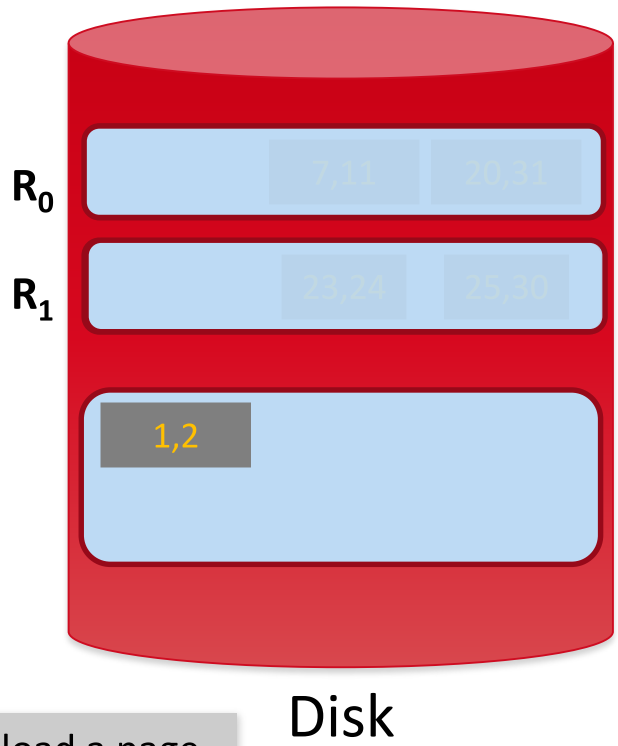
Example Step-2: Merge Sorted Runs

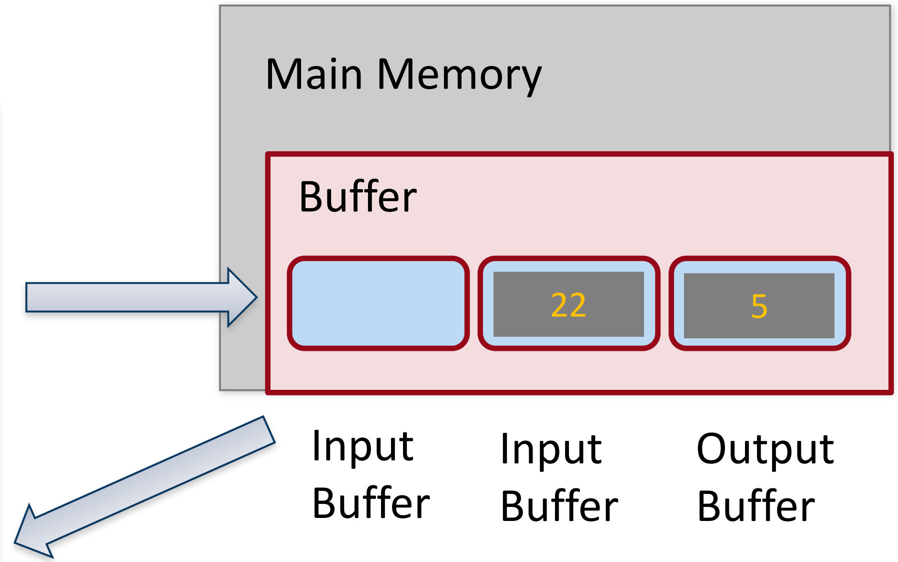
Example:

- Each input run consists of three pages

Input:
Two sorted runs

Output:
One *merged* sorted run





1. Select the first record (in sort order) among all input buffer pages
2. Write the record to the output buffer. If output is full, write it to disk.

If this is the last record of the input buffer page **then** read the next page (if any) of the run into the buffer.

Which run to load a page from next?

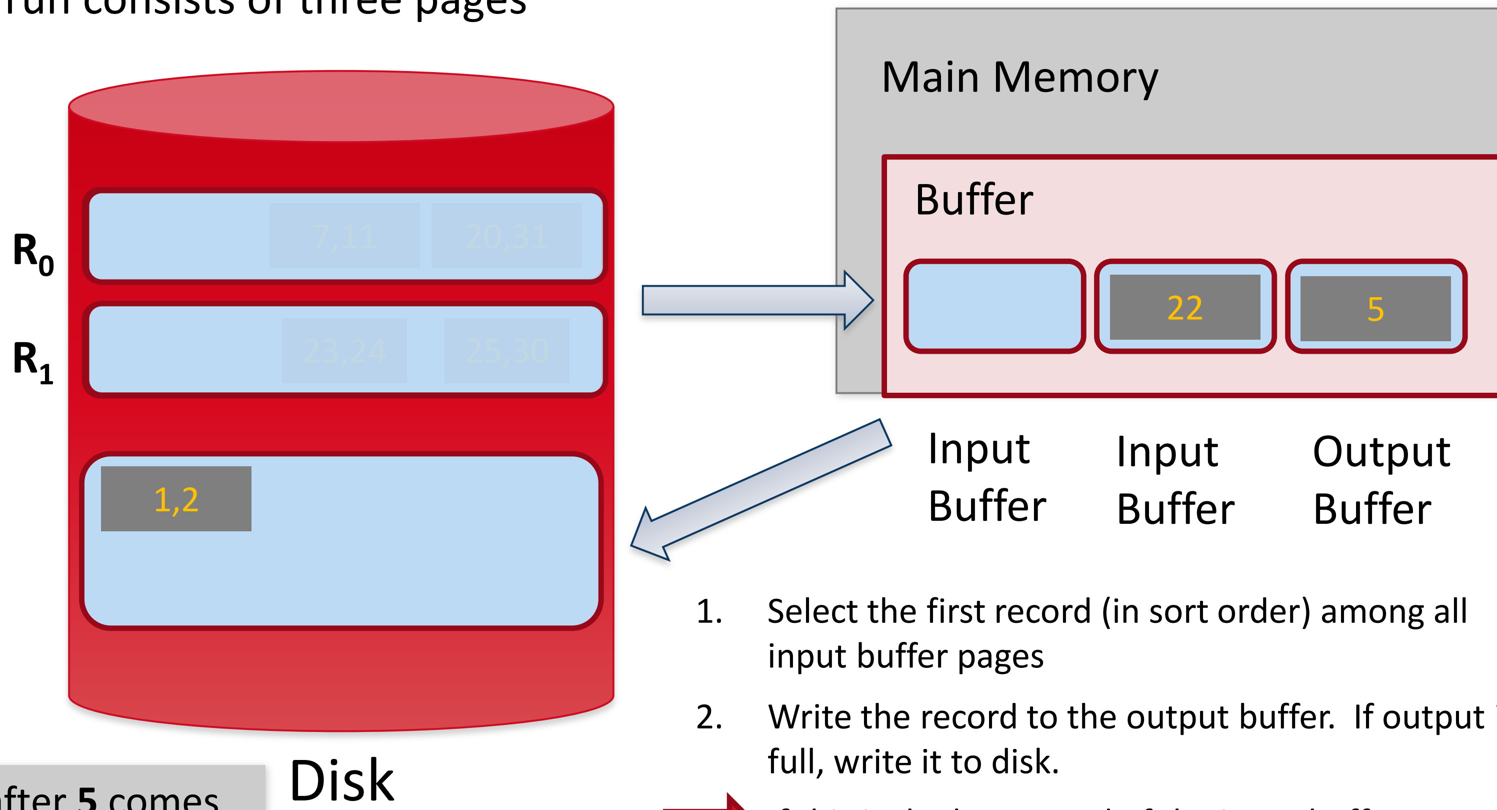
Example Step-2: Merge Sorted Runs

Example:

- Each input run consists of three pages

Input:
Two sorted runs

Output:
One *merged* sorted run



1. Select the first record (in sort order) among all input buffer pages
2. Write the record to the output buffer. If output is full, write it to disk.

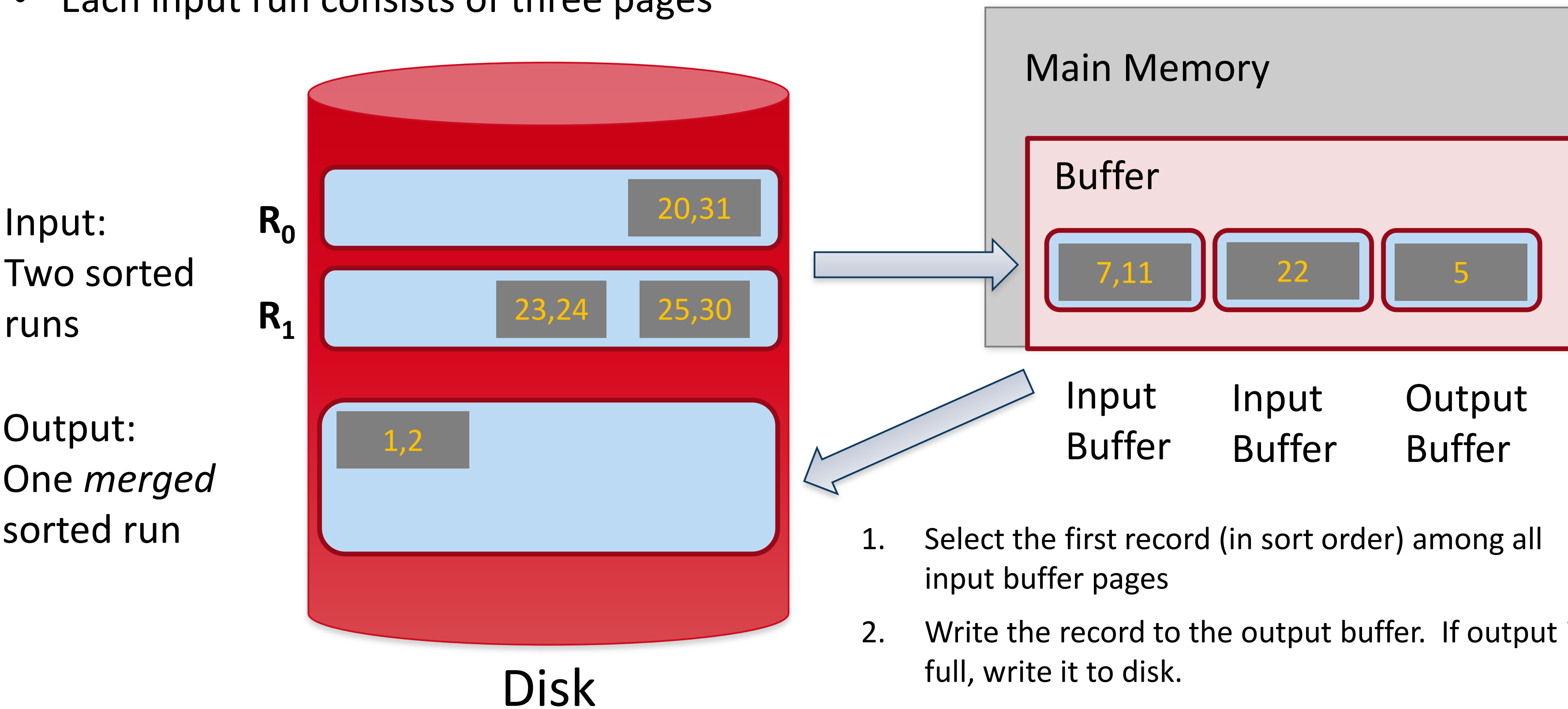
If this is the last record of the input buffer page **then** read the next page (if any) of the run into the buffer.

The next value after **5** comes from R_0 … so we should load from R_0!

Example Step-2: Merge Sorted Runs

Example:

- Each input run consists of three pages

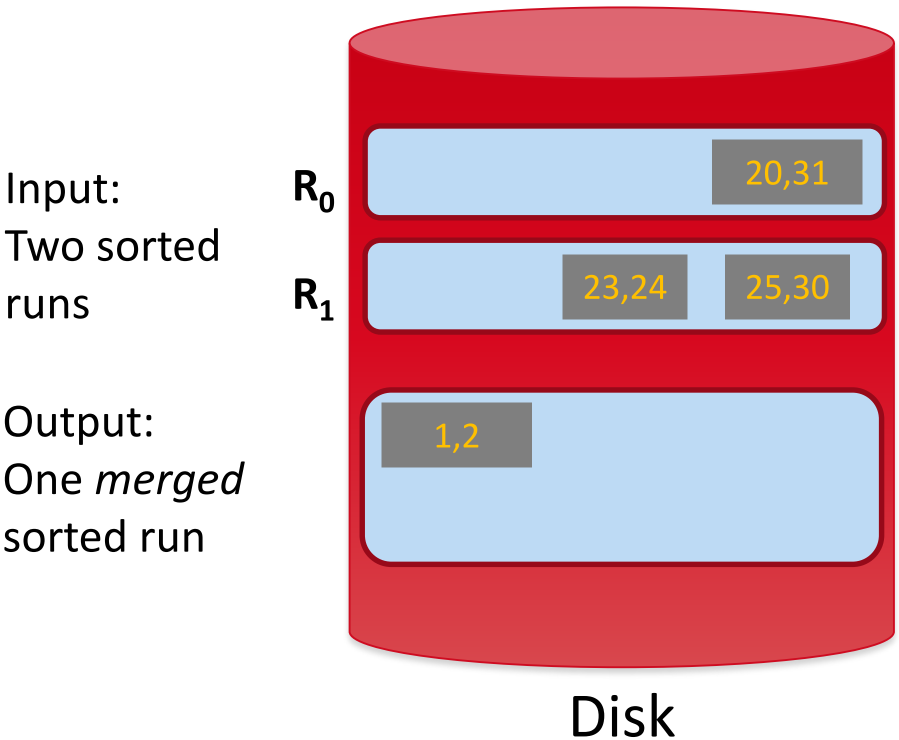


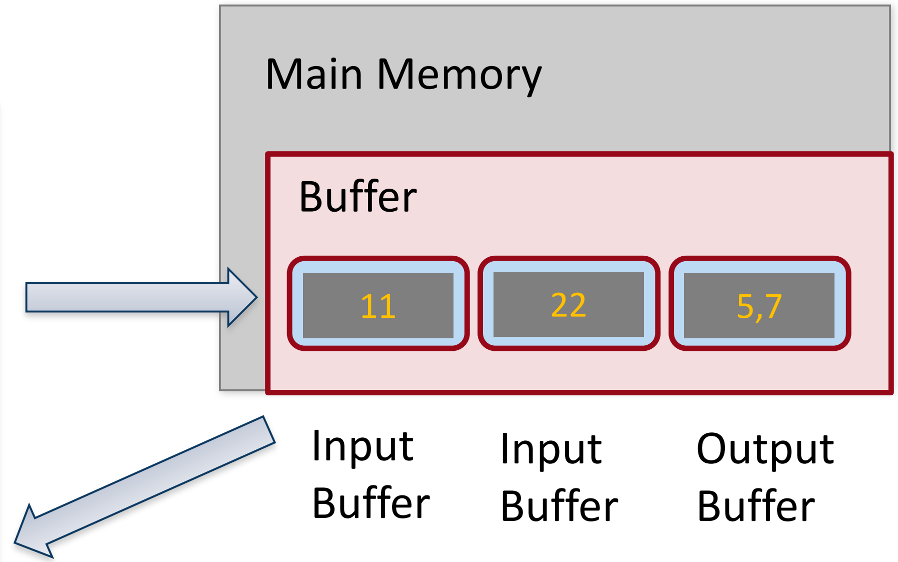
1. Select the first record (in sort order) among all input buffer pages
2. Write the record to the output buffer. If output is full, write it to disk.
3. **If** this is the last records of the input buffer page **then** read the next page (if any) of the run into the buffer.

Example Step-2: Merge Sorted Runs

Example:

- Each input run consists of three pages



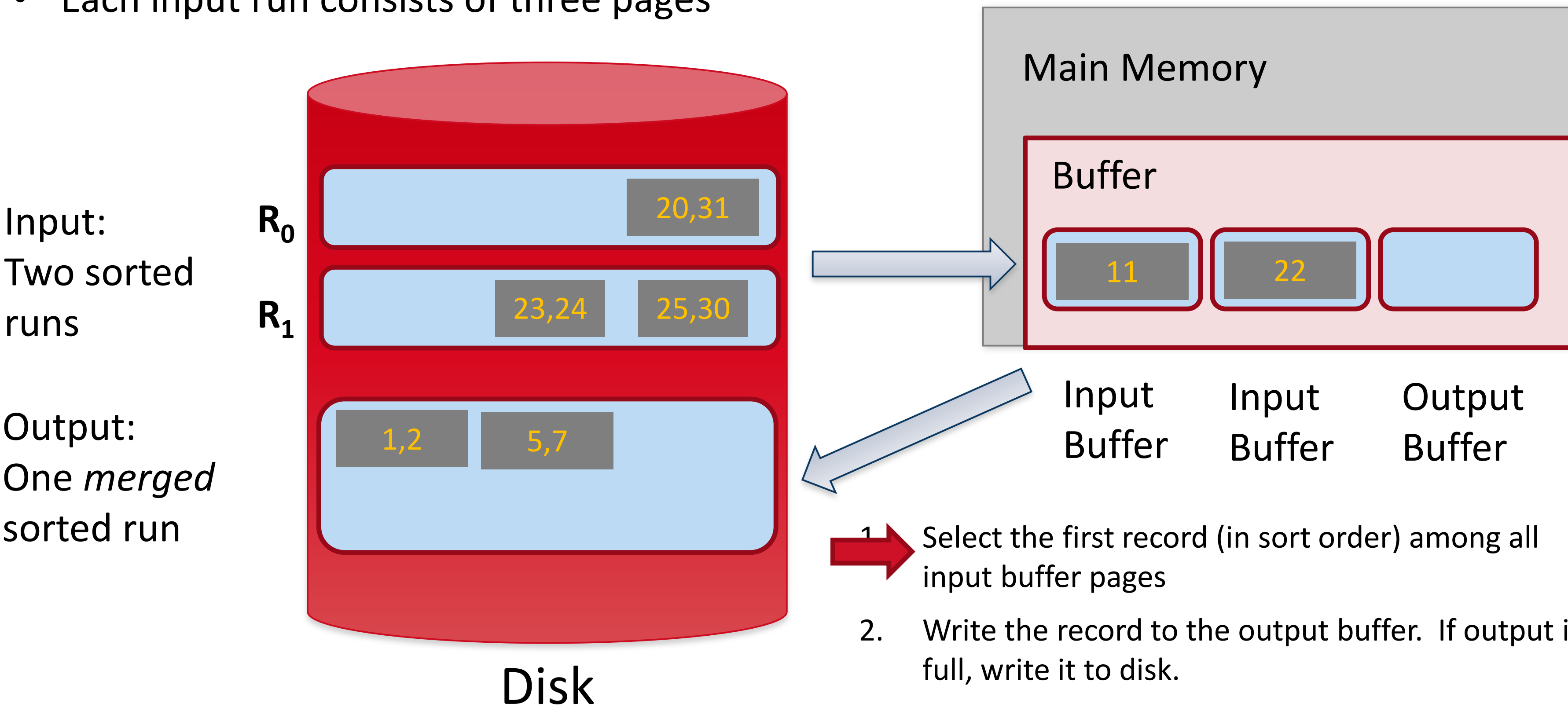


1. Select the first record (in sort order) among all input buffer pages
2. Write the record to the output buffer. If output is full, write it to disk.
3. **If** this is the last record of the input buffer page **then** read the next page (if any) of the run into the buffer.

Example Step-2: Merge Sorted Runs

Example:

- Each input run consists of three pages



1. Select the first record (in sort order) among all input buffer pages
2. Write the record to the output buffer. If output is full, write it to disk.
3. **If** this is the last records of the input buffer page **then** read the next page (if any) of the run into the buffer.

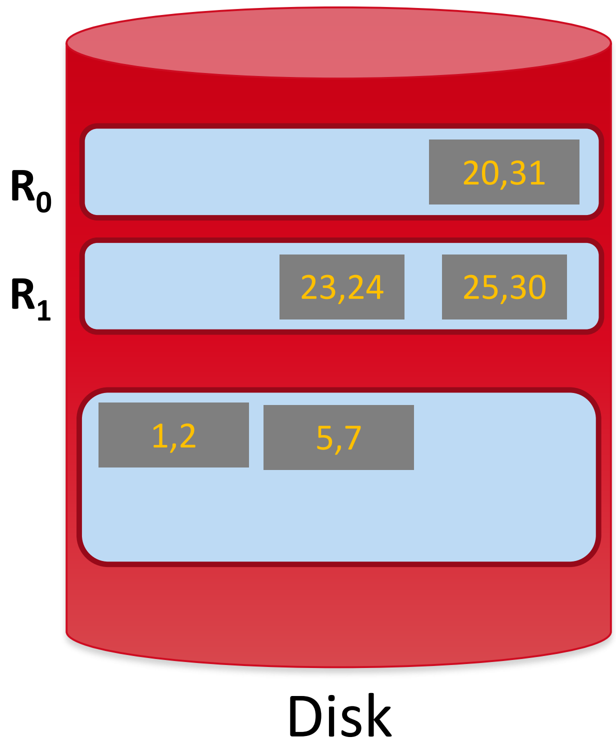
Example Step-2: Merge Sorted Runs

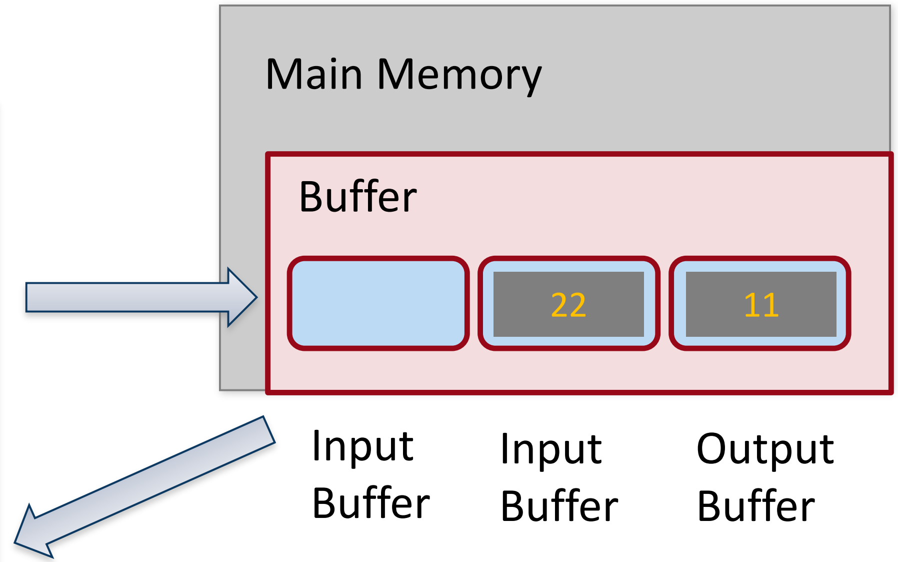
Example:

- Each input run consists of three pages

Input:
Two sorted runs

Output:
One *merged* sorted run



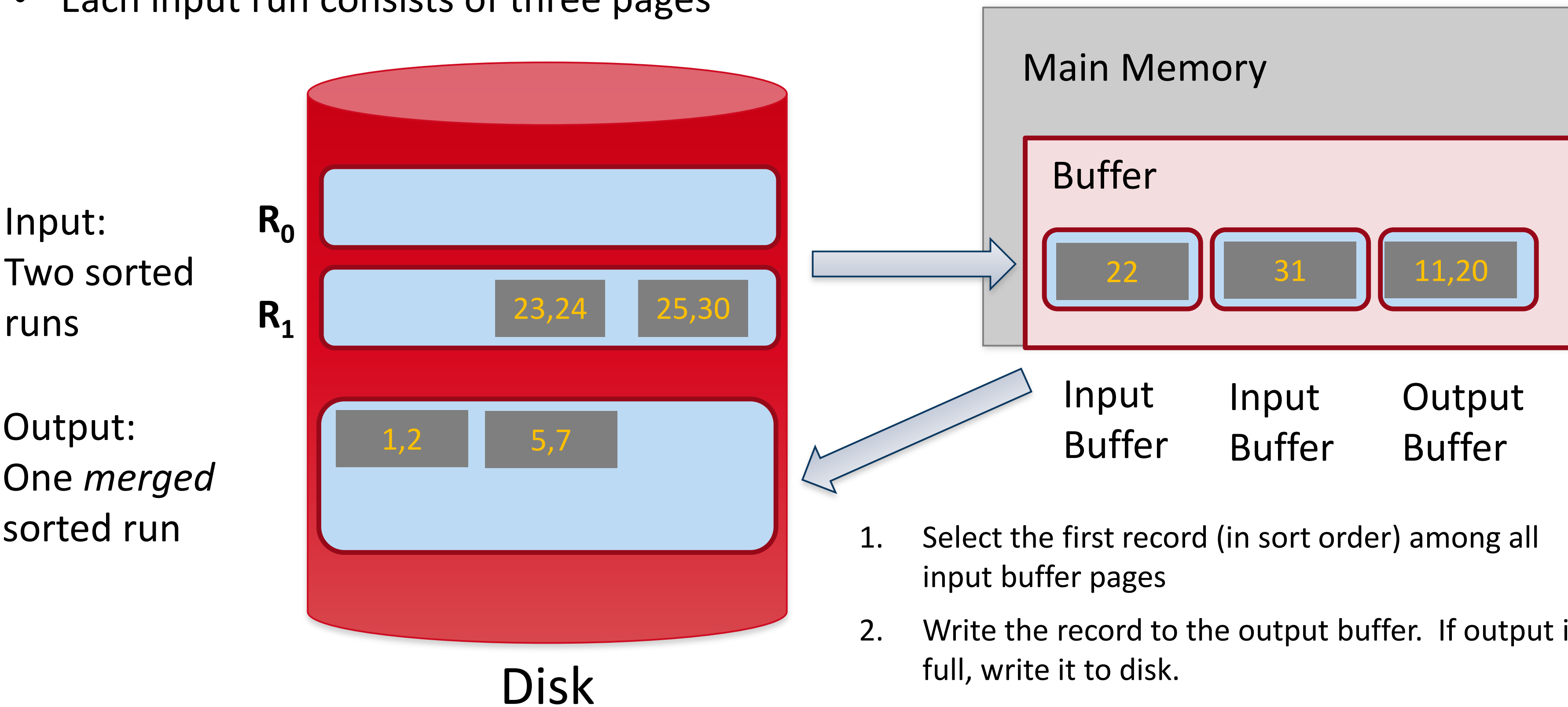


1. Select the first record (in sort order) among all input buffer pages
2. Write the record to the output buffer. If output is full, write it to disk.
3. **If** this is the last record of the input buffer page **then** read the next page (if any) of the run into the buffer.

Example Step-2: Merge Sorted Runs

Example:

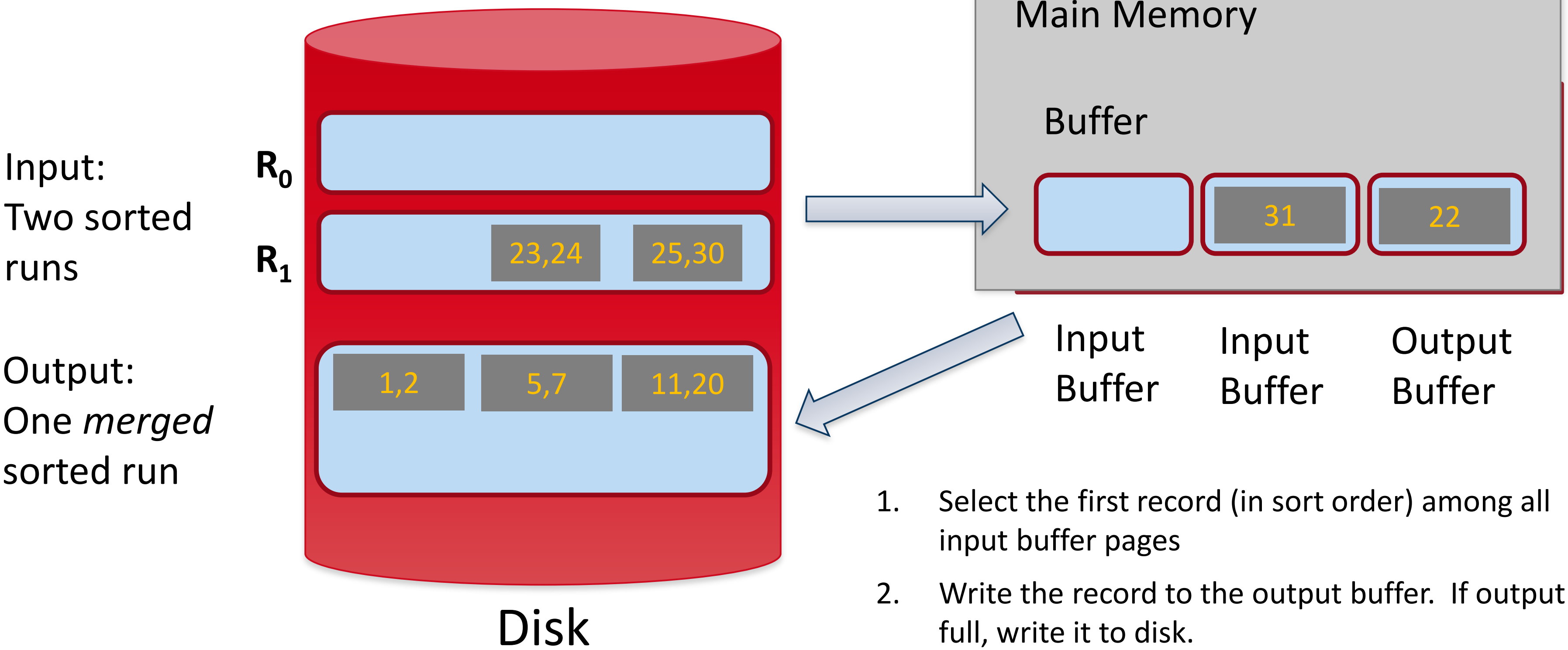
- Each input run consists of three pages



1. Select the first record (in sort order) among all input buffer pages
2. Write the record to the output buffer. If output is full, write it to disk.
3. **If** this is the last records of the input buffer page **then** read the next page (if any) of the run into the buffer.

Example Step-2: Merge Sorted Runs

Example:
- Each input run consists of three pages



1. Select the first record (in sort order) among all input buffer pages
2. Write the record to the output buffer. If output is full, write it to disk.
3. **If** this is the last records of the input buffer page **then** read the next page (if any) of the run into the buffer.

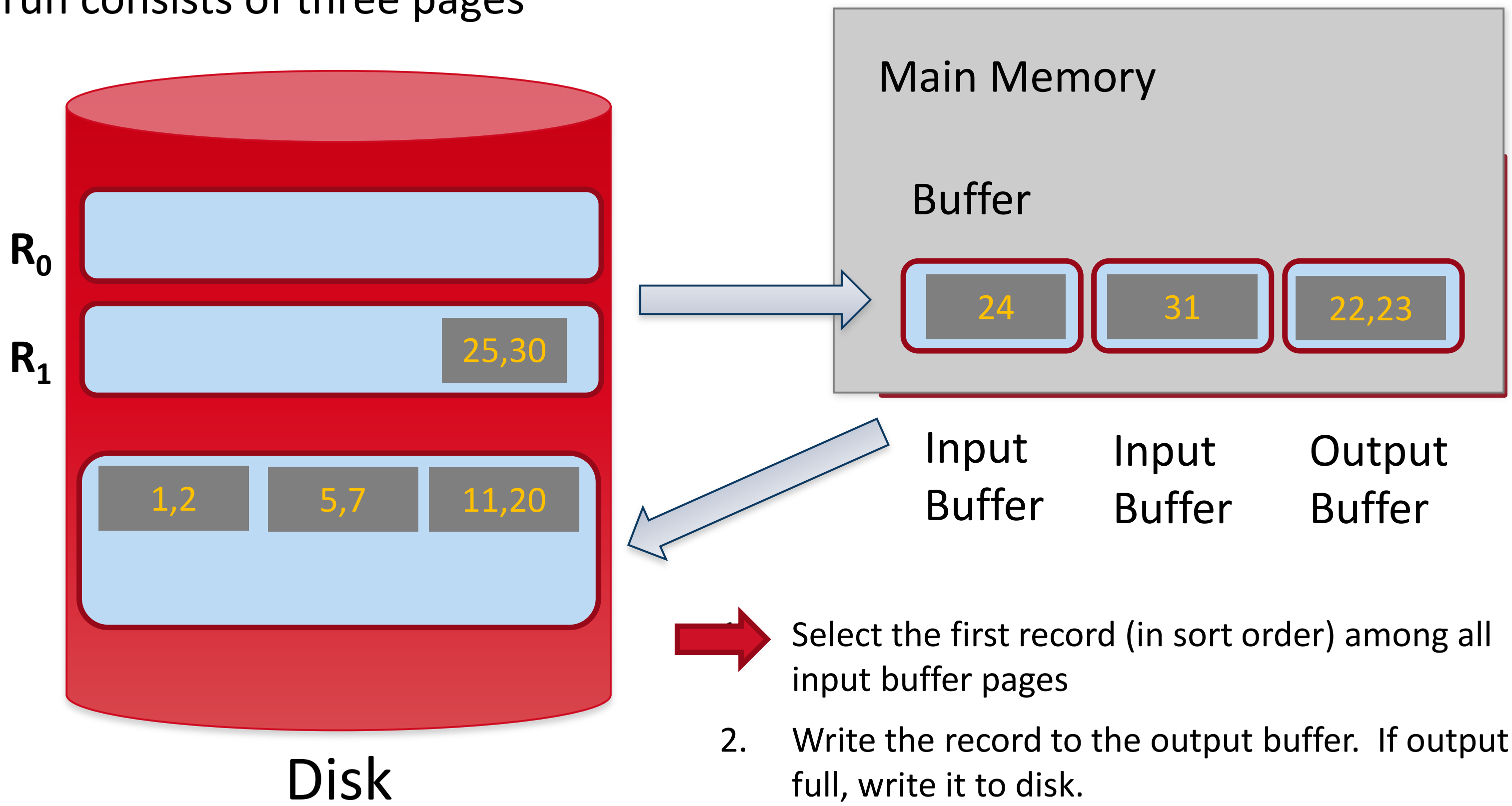
Example Step-2: Merge Sorted Runs

Example:

- Each input run consists of three pages

Input:
Two sorted runs

Output:
One *merged* sorted run



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

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

3. **If** this is the last records of the input buffer page **then** read the next page (if any) of the run into the buffer.

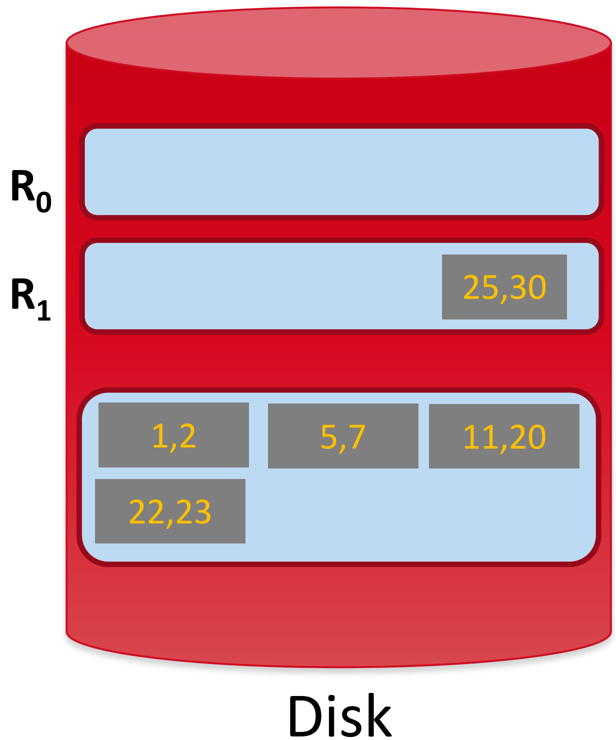
Example Step-2: Merge Sorted Runs

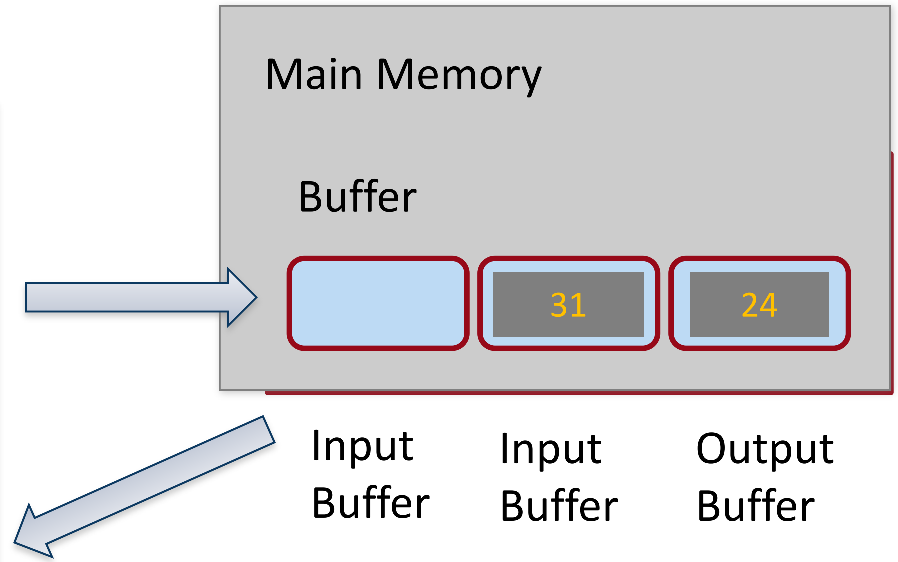
Example:

- Each input run consists of three pages

Input:
Two sorted
runs

Output:
One *merged*
sorted run





1. Select the first record (in sort order) among all input buffer pages
2. Write the record to the output buffer. If output is full, write it to disk.
3. **If** this is the last records of the input buffer page **then** read the next page (if any) of the run into the buffer.

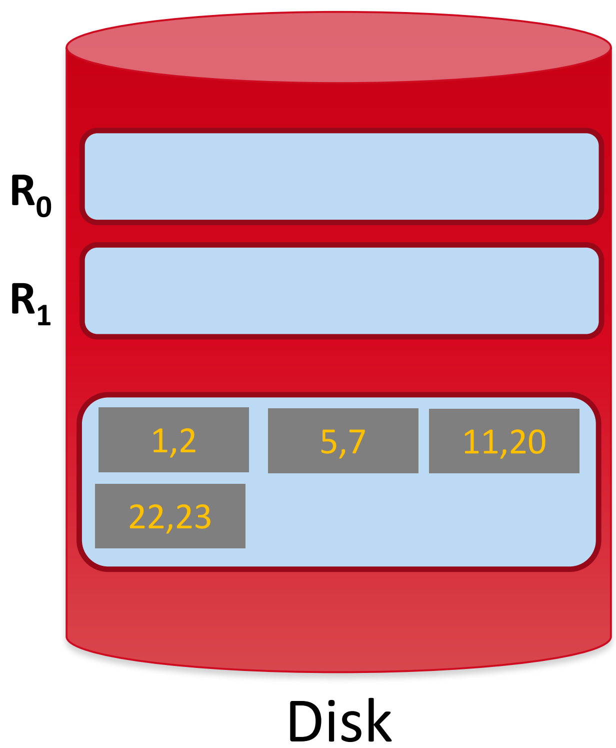
Example Step-2: Merge Sorted Runs

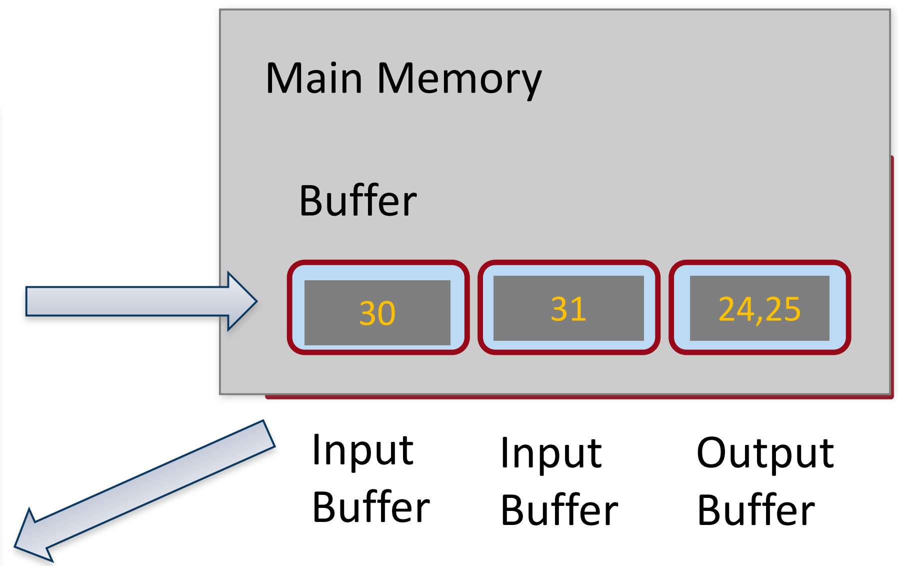
Example:

- Each input run consists of three pages

Input:
Two sorted
runs

Output:
One *merged*
sorted run





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

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

3. **If** this is the last records of the input buffer page **then** read the next page (if any) of the run into the buffer.

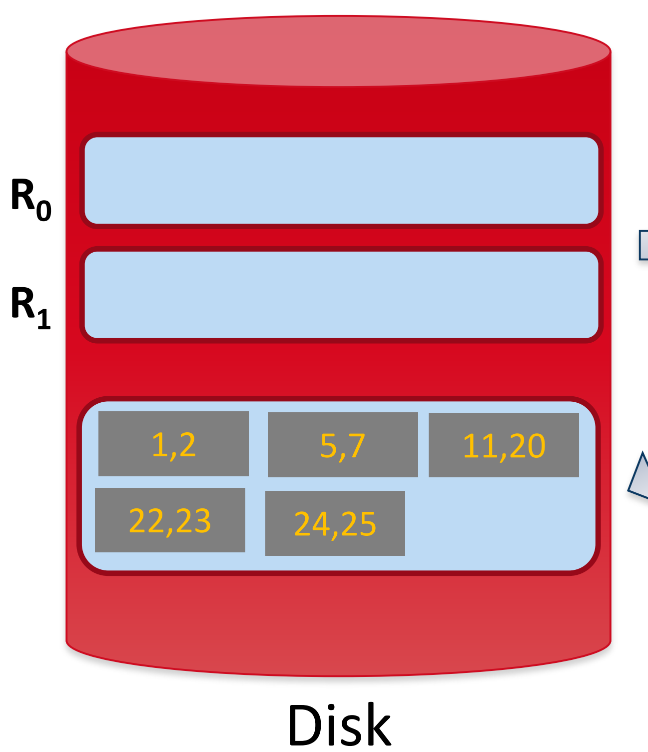
Example Step-2: Merge Sorted Runs

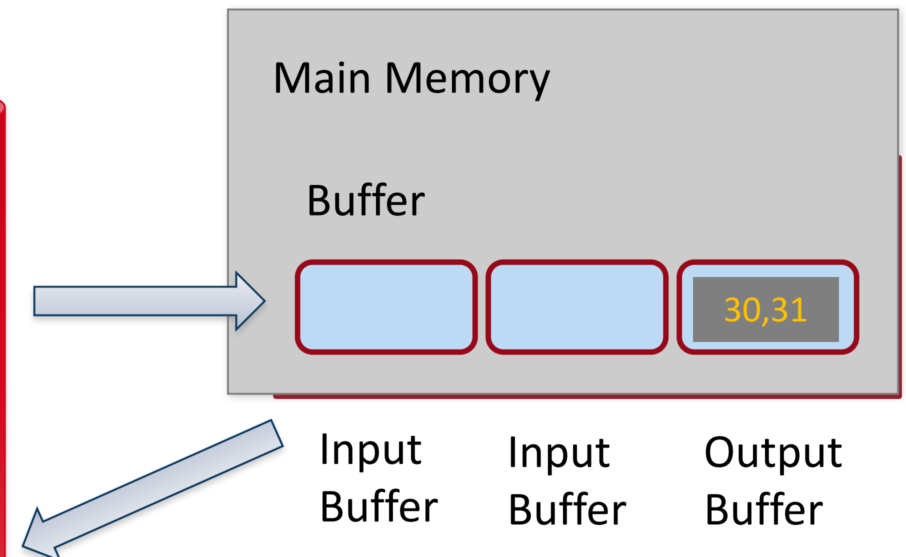
Example:

- Each input run consists of three pages

Input:
Two sorted runs

Output:
One *merged* sorted run





1. Select the first record (in sort order) among all input buffer pages
2. Write the record to the output buffer. If output is full, write it to disk.
3. **If** this is the last records of the input buffer page **then** read the next page (if any) of the run into the buffer.

External Merge Sort Algorithm

Let B denote memory size (in pages).

1. Create sorted ***runs***. (A run is a sorted subset of records)

   Let i be 0 initially. **Repeatedly** do the following till the end of the file:
   (a) Read B pages of records from disk into memory
   (b) Sort the in-memory blocks
   (c) Write sorted data to run i; increment i by 1.
   Let the final value of i be m ($= \lceil N / B \rceil$); there are m sorted runs. N is the size of the file.

2. Merge each contiguous group of $B-1$ runs into 1 run: ($B-1$)-way merge.

   i. Use $B-1$ pages of memory to buffer input runs, and 1 page to buffer output. Read the first page of each run into its buffer page

   ii. repeat

   1. Select the first record (in sort order) among all input buffer pages
   2. Write the record to the output buffer. If output is full, write it to disk.
   3. **If** this is the last record of the input buffer page **then** read the next page (if any) of the run into the buffer.

   until all input buffer pages are empty:

3. After each merge pass, the number of runs is reduced by a factor of B-1. If $m \geq B$, several merge passes are required. The number of passes (including the initial sorting pass) for the multiway merging is $\lceil \log_{(B-1)} (N/B) \rceil + 1$

Outline

- Basic Steps in Query Processing
- Query Optimization
  - Logical Query Plan: Heuristic-based Optimization
  - Physical Query Plan: Cost Estimate Optimization
- **Query Execution**

Step 3: Query Execution

- Input: Data organization and access
  - ***Organization:***
    - Heap file (unsorted)
    - Sorted file
    - Mixed
  - ***Access***
    - Heap files can be accessed
      - with a linear/file/table scan
      - Indexes
    - Sorted files
      - Specialized access algorithms
      - Indexes
- An **access path** is a method of retrieving tuples

Step 3: Evaluation of Expressions

- **Materialization** (also: **set-at-a-time**):
  simply evaluate one operation at a time. The result of each evaluation is materialized (stored) in a temporary relation for subsequent use.
- In other words: Output of one operator **written to disk** and the next operator will read it from the disk.

- **Pipelining** (also: **tuple-at-a-time** or **on-the-fly processing**):
  evaluate several operations simultaneously in a pipeline
- In other words: Output of one operator is **directly** input to next operator

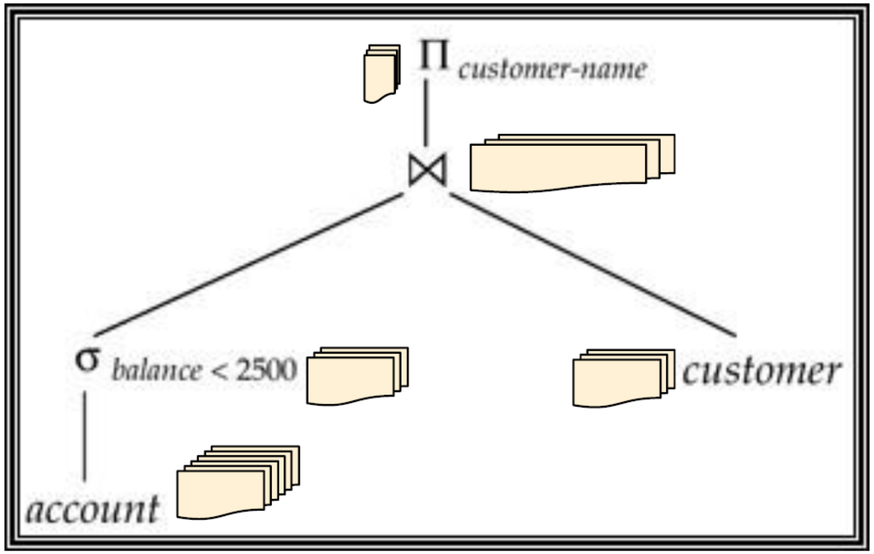
Materialization

› **Materialized evaluation:** evaluate one operation at a time, starting at the lowest-level. Use intermediate materialized (stored) results into temporary tables to evaluate next-level operations.

› E.g., in figure below:

1. compute and store new table for $\sigma_{balance<2500}(account)$
2. Compute and store result of materialized result joined with **customer**
3. Read back new materialized result and compute the projections on **customer-name**.

- Can always apply materialized evaluation
- Costs can be quite high

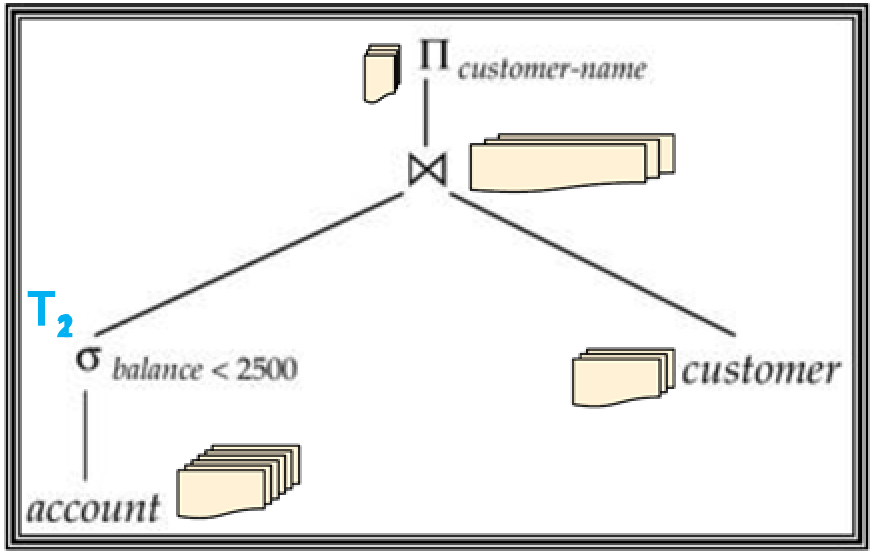


Pipelining

› **Pipelined evaluation**: evaluate several operations simultaneously, passing the results of one operation on to the next.

› E.g., in the expression tree

1. Find tuple matching $\sigma_{balance<2500}(account)$
   a) Join matching tuple with tuples of customer until a new tuple is generated
   b) Project customer name from joined tuple
   c) Repeat for all join results
2. Repeat for next selection result



› Much cheaper than materialization:

- no need to store a temporary table to disk.

Summary

- Understanding of Role and Structure of Query Processing
  - From SQL to physical data access
  - 3 Steps: Query Parsing, Optimization, Execution
  - Expression Tree vs. Evaluation Plan
  - Query Execution Algorithms
- Operator Algorithms
  - External Merge Sort
  - Joins
    - Simple Nested
    - Block Nested
    - Index Nested

Next Week: Guest Lecture and Revision

- Guest Lecture
- Discussion regarding Final Exam
  - Instructions
  - Question types
- Content Review

See you next week!