COMP9120

Week 12: Query Processing and Evaluation

Semester 1, 2025

Professor Athman Bouguettaya
School of Computer Science

THE UNIVERSITY OF SYDNEY

Contents co-developed with Dr Lijun Chang

Acknowledgement of Country

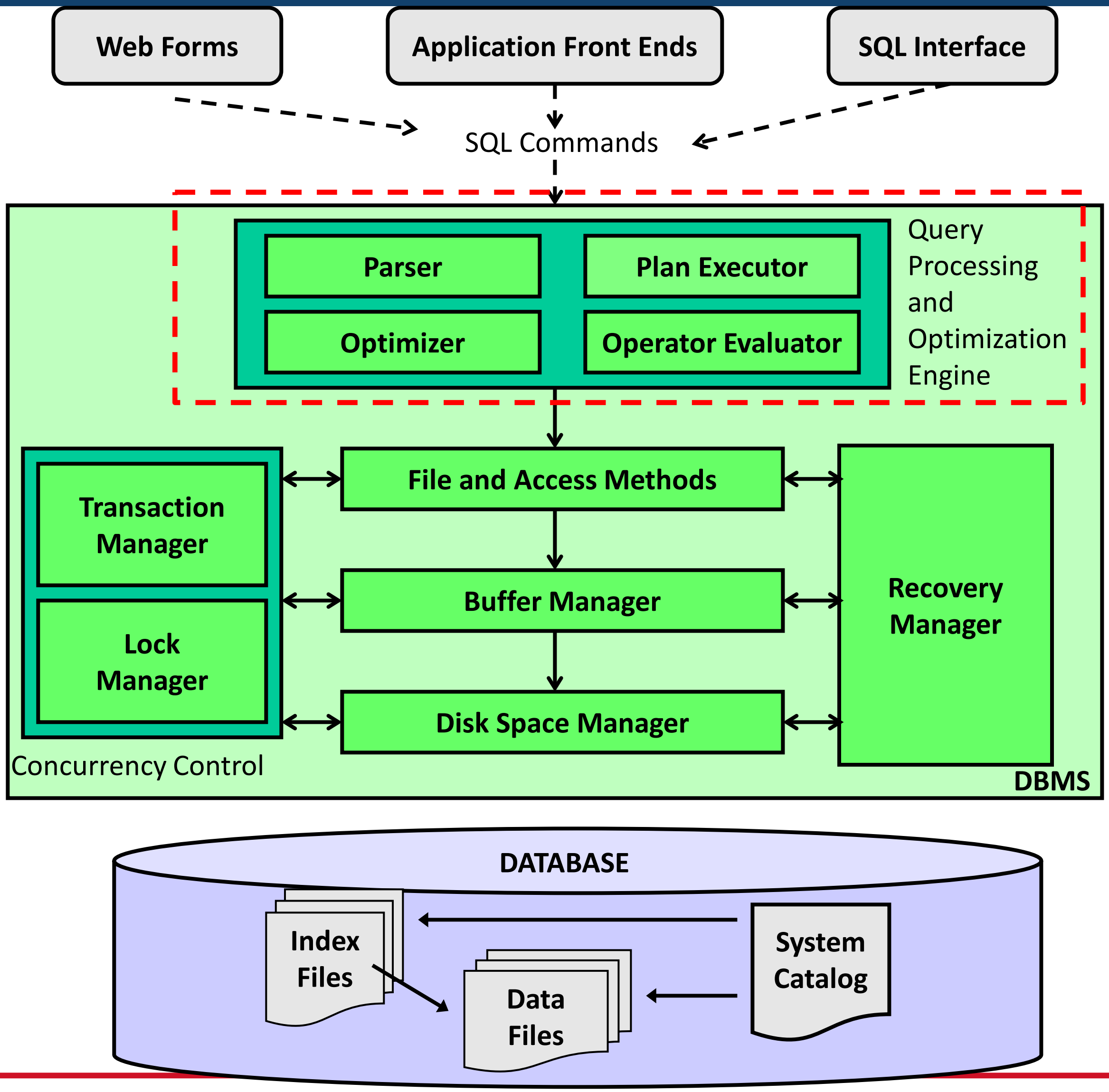
I would like to acknowledge the Traditional Owners of Australia and recognise their continuing connection to land, water and culture. I am currently on the land of the Gadigal people of the Eora nation and pay my respects to their Elders, past, present and emerging.

Copyright warning

Outline

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

Internal Structure of a DBMS



Outline

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

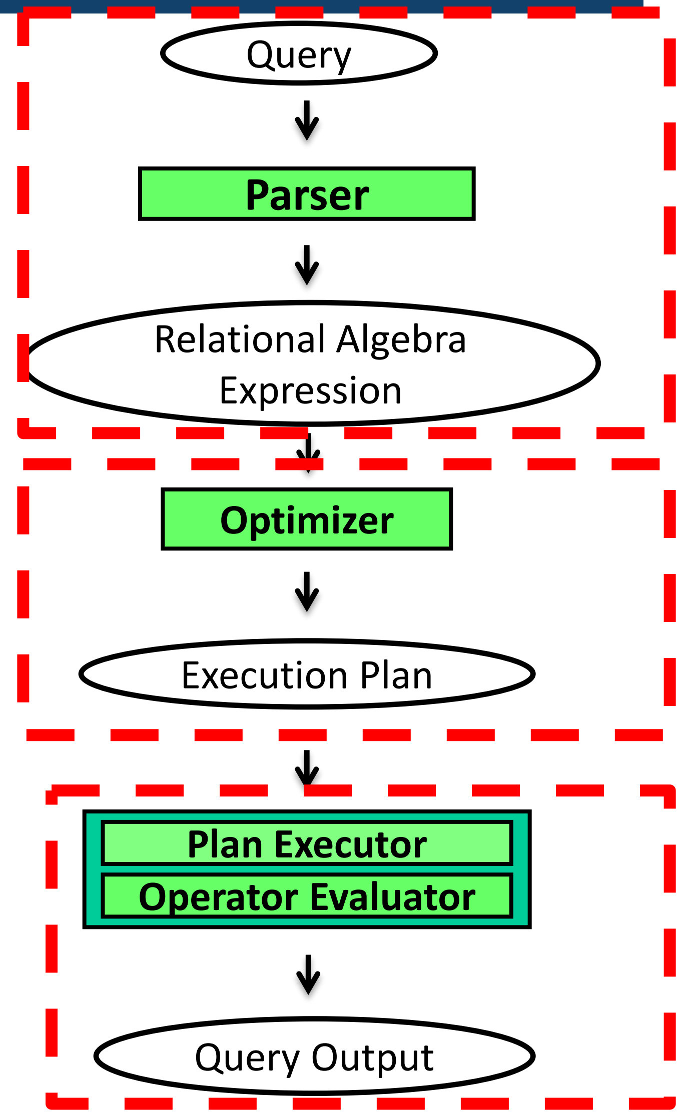
Query processing

› Main **issues**:

- 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***)
- What are the **criteria** that are used to **execute a query**? (***execution phase***)

Basic Steps in Query Processing

- Step 1: Parsing and Translation
  - **Check** for **syntactic** and **semantic errors**
  - **Translate** the **SQL query** into **relational algebra**.
  - **Rewrite Queries: Views** are **replaced** with **actual sub-query** on relations
- Step 2: Query Optimization
  - Amongst all **equivalent** query **evaluation plans, choose** the one with the **lowest expected cost**.
    - Use **heuristics** to optimize at the relational **algebra** level
    - **Select** a **query execution strategy** based on **cost estimate**
- Step 3: Query Execution
  - **Strategies** to **execute** the **operations** in a **query execution tree**



Relational model

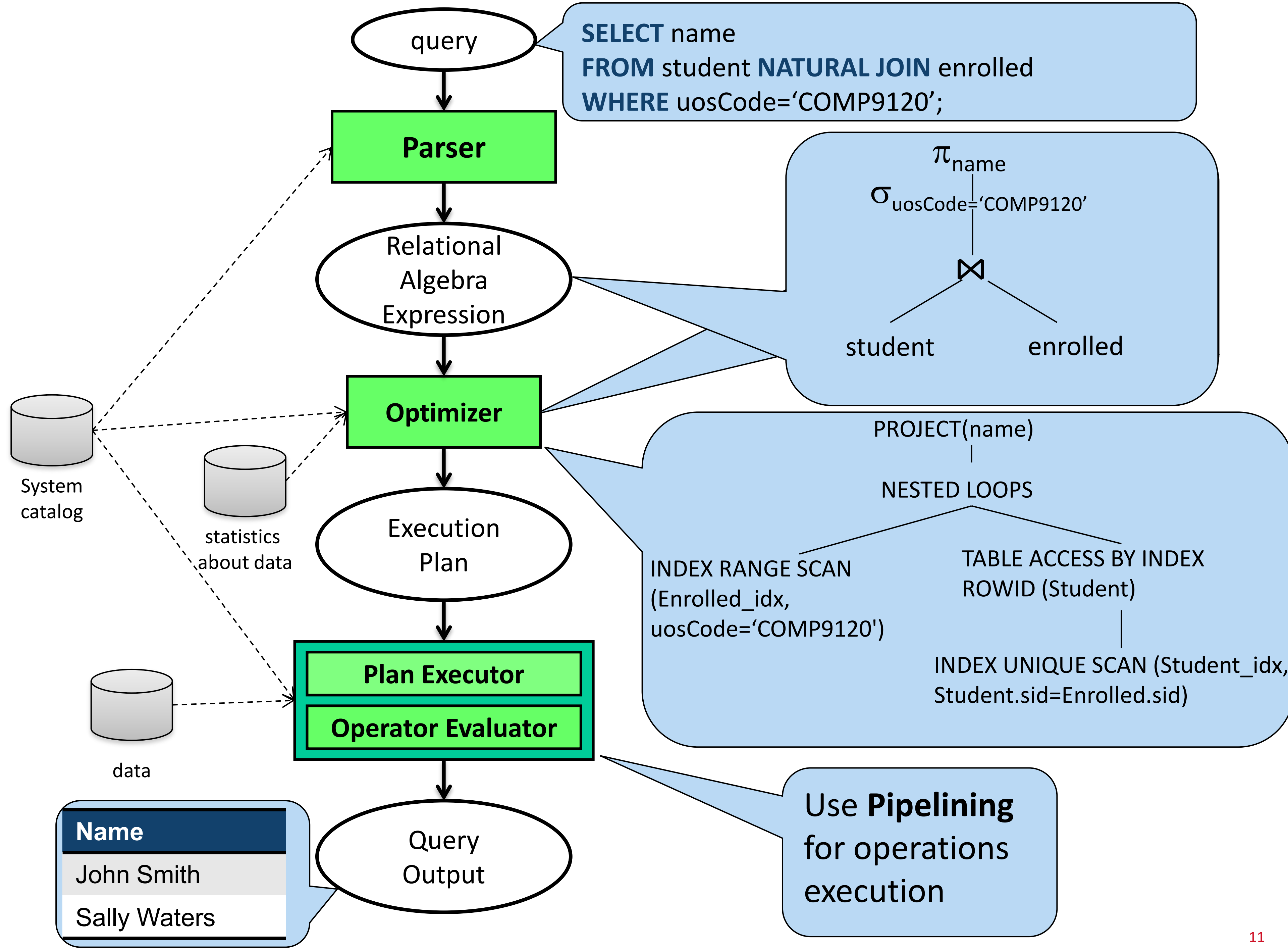
Relational model:

- Provides more **abstraction** (through the ***declarative model***) to make databases more **usable** by **general users**.
  - Drawback: **High level abstraction** also makes it **hard** for ***non-expert*** users to **optimize** queries.
    - General users are **focused** on ***correctness*** of the query, ***not*** its ***efficiency***
- 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) models
  - **Focus** of queries is *mostly* on the ***what*** and **not** the ***how***.
    - **Good news**: These declarative queries **lend** themselves to ***computer-based optimization***.

Query processing and optimization: main steps

Output of each **step**:

- *Step 1*: **Parser**
  - A **parse tree** consists of a **SFW** (**S**elect-**F**rom-**W**here) **expression**
  - **Converts** the **parse tree** into an ***initial*** **logical query plan**
- *Step 2*: **Optimization**
  - ***Heuristics***: efficient **logical query plan**
  - ***Cost estimate***: efficient **physical query plan**
- *Step 3*: **Query execution**
  - **Select** an **execution order** of **operations** in a query execution tree


query
SELECT name
FROM student NATURAL JOIN enrolled
WHERE uosCode='COMP9120';
Parser
Relational Algebra Expression
π_{name}
$\sigma_{uosCode='COMP9120'}$
$\bowtie$
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
Use Pipelining for operations execution
Query Output
Name
John Smith
Sally Waters

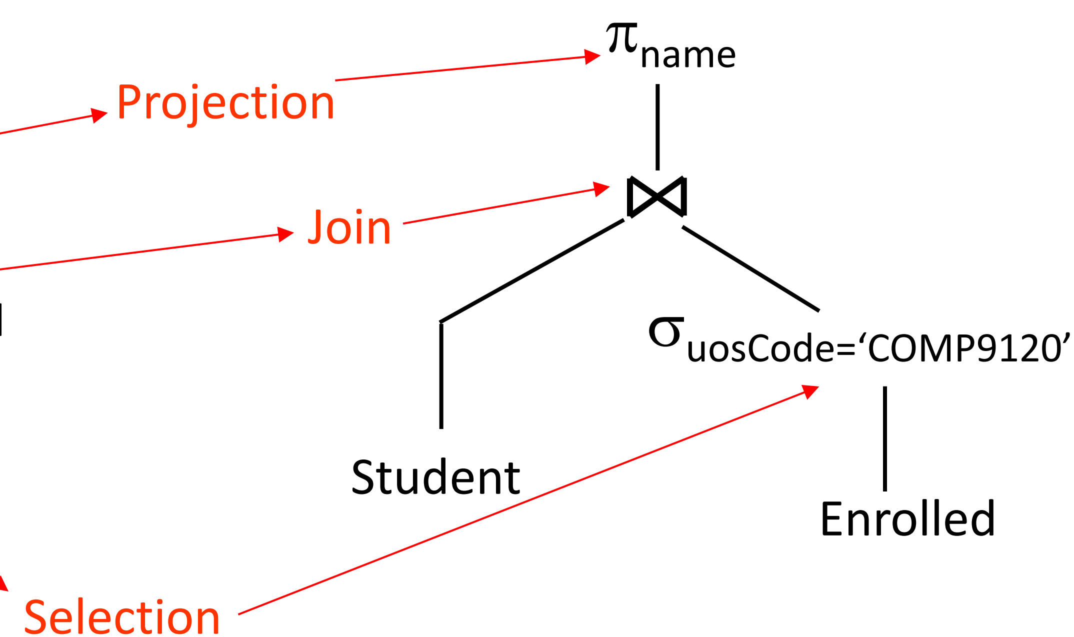
Step 1: Parsing and Translation

› SQL **query** gets **translated** into **Relational Algebra (RA) expression**, 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***
  - *Heuristics: minimize the size of intermediate results (relations) using equivalent algebraic expressions.*

- ***Cost estimate*** of different execution *strategies/plans* to **select** the one with ***minimal cost***: output is an efficient ***physical query plan***
  - ***Cost estimate objective: minimizing the number of disk I/Os.***

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

Note:

- We can **transform** any **tuple calculus expression** (i.e., SQL query) to an **equivalent algebraic expression**
  - 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 ***most likely minimize size of intermediate results.*** **Why?**
    - This would most likely **minimize I/Os**!

Example

Consider the following query:

"find the <u>assets</u> and <u>names of all banks</u> which have depositors <u>living in Sydney</u>".
Assume we have three relations: ***Deposit, Customer****,* and **Branch**.

Schema:

Deposit			
branchname	account#	customername	balance

Customer		
customername	street	customercity

Branch		
branchname	assets	branchcity

The above query is *equivalent* to the following *algebraic expression*:

$$\Pi_{\text{Branchname, Assets}} (\sigma_{\text{Customercity=Sydney}} (\text{Customer} \bowtie \text{Deposit} \bowtie \text{Branch}))$$

Example

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

Note that:

We **most likely** 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?

Use some ***rearrangements*** of the **operations** (*algebraic manipulation*).

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

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

For instance: the previous expression

$\Pi_{\text{Branchname, Assets}}(\sigma_{\text{Customercity=Sydney}}(\text{Customer} \infty \text{Deposit} \infty \text{Branch}))$ is **equivalent** to:

$$\Pi_{\text{Branchname, Assets}}((\sigma_{\text{Customercity=Sydney}}(\text{Customer})) \infty \text{Deposit} \infty \text{Branch})$$

Selection optimization

Selection optimization:

Whenever possible, do a ***selection as soon as possible***.

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 therefore:

$\Pi_{\text{Branchname, Assets}}(\sigma_{\text{Customercity=Sydney} \wedge \text{Balance} > 500}(\text{Customer} \infty \text{Deposit}) \infty \text{Branch})$

The *intermediate result* has now been *reduced*.

Can we do 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:

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

Heuristics: We should ***eliminate those attributes*** that will ***not play any role*** in the **remaining operations *as soon as possible***.

In the example above, the attribute *Branchname* is the **only attribute** we need in the **first join**. The attribute *Assets* is in the table *Branch*. One **efficient** way is to do a ***projection*** of *Branchname* on the ***result of the first join.***

The resulting transformation is as follows:

$$\Pi_{\text{Branchname, Assets}} (\boldsymbol{\Pi}_{\textbf{Branchname}} (\boldsymbol{\sigma}_{\textbf{Customercity=Sydney}} \textbf{(Customer)} \infty \textbf{Deposit)} \infty \text{Branch})$$

Projection optimization

Can we do better?

Yes!

For instance, *Branchcity* in the table *Branch* is not needed. Therefore, we can then do a projection on only *Branchanme* and *Assets* on the table *Branch*.

The transformation is as follows:

$$\cancel{\Pi_{\text{Branchname, Assets}}}(\boldsymbol{\Pi}_{\textbf{Branchname}}(\boldsymbol{\sigma}_{\textbf{Customercity=Sydney}}\ \textbf{(Customer)} \infty \textbf{Deposit}) \infty \boldsymbol{\Pi}_{\textbf{Branchname, Assets}}(\text{Branch}))$$

Now the **first projection** is **redundant** because the result is exactly the same **without it: *remove it***!

The *final transformation* is as follows:

$$\boldsymbol{\Pi}_{\textbf{Branchname}}(\boldsymbol{\sigma}_{\textbf{Customercity=Sydney}}\ \textbf{(Customer)} \infty \textbf{Deposit}) \infty \boldsymbol{\Pi}_{\textbf{Branchname, Assets}}(\text{Branch})$$

Rules for equivalent algebraic transformations

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

In the previous optimization steps, we ***intuitively*** came up with ***transformation*** **rules**.

Rules: What are the ***formal algebraic transformation*** rules that enabled us to **perform** the previous **transformations**?

Here is a **sample set of rules** for algebraic transformations:

1. **Commutative rule** for joins:

$R1 \infty R2 = R2 \infty R1$

2. **Associative rule** for joins

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

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

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

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. **Distributive property of selections over joins.** σ_Θ distributes over the join operation when all the attributes in the selection condition θ involve only the attributes of one of the relations (e.g., R1) being joined.

$\sigma_\theta(R1 \infty R2) = (\sigma_\theta(R1)) \infty R2$

Example of an equivalent algebraic transformation

Assume we have the following relations:

Deposit			
branchname	account#	customername	balance

Customer		
customername	street	customercity

Branch		
branchname	assets	branchcity

Example of using algebraic rules

Using our previous 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:

$$\Pi_{\text{Branchname, Assets}} (\sigma_{\text{Customercity=Sydney} \wedge \text{Balance} > 500} (\text{Customer} \infty \text{Deposit} \infty \text{Branch}))$$

Use **Rule #5**: $\sigma_\theta(R1 \infty R2) = (\sigma_\theta(R1)) \infty R2$, to do the selection *after* doing a join on *Customer* and *Deposit*. The resulting expression is therefore:

$$\Pi_{\text{Branchname, Assets}} (\sigma_{\text{Customercity=Sydney} \wedge \text{Balance} > 500} (\text{Customer} \infty \text{Deposit}) \infty \text{Branch}).$$

Using **Rule #4:** $\sigma_{\Theta1} (\sigma_{\Theta2}(R)) = \sigma_{\Theta2} (\sigma_{\Theta1}(R)) = \sigma_{\Theta1 \wedge \Theta2} (R)$, to break up the selection condition into two selections and we get:

$$\Pi_{\text{Branchname, Assets}} (\sigma_{\text{Customercity=Sydney}} (\sigma_{\text{Balance} > 500} (\text{Customer} \infty \text{Deposit})) \infty \text{Branch})$$

Use **Rule #5 twice:** $\sigma_\theta(R1 \infty R2) = (\sigma_\theta(R1)) \infty R2$, to move the first and second selections to their respective relations:

$$\Pi_{\text{Branchname, Assets}} (\sigma_{\text{Customercity=Sydney}} (\text{Customer}) \infty \sigma_{\text{Balance} > 500} (\text{Deposit})) \infty \text{Branch})$$

Outline

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

Physical Query Plan

- Read as input the **query expression tree** of logical (RA) operations and **generate** a **Query Execution Plan:**
  - Tree of **relational algebra operators** + **choice** of **algorithm** for each **operator**.

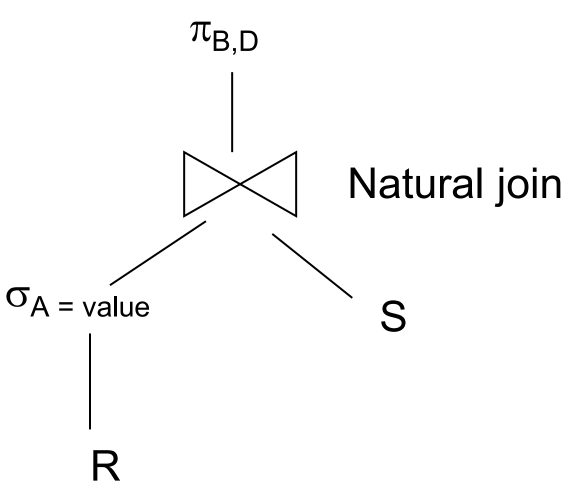
- The aim of this step is to:
  - Find an **optimal plan** among a set of all **equivalent plans**
    - Main criterion: ***Lowest estimated I/O***

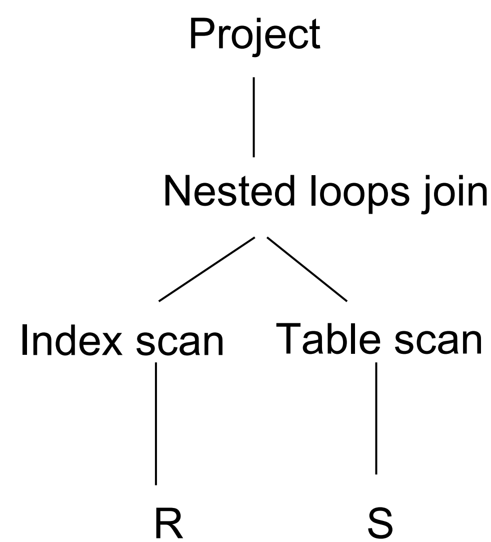
Query Execution Plan

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

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

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**

Query Execution Plan

- **Multiple** possible **query evaluation plans** for the expression tree are considered by the query optimizer. Here is another example:

Expression tree

$\pi_{balance}$

$\sigma_{balance<2500}$

account

Possible query evaluation plans

Project $_{balance}$

Filter$_{balance<2500}$

Linear Scan (*account*)

Project $_{balance}$

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

Cost of query processing

Cost of query processing:

- Compute the **cost** of **every algebraic operation** in terms of **I/Os**

To compute this cost, use

- Access methods available
- Data physical organization: collected facts (e.g., blocking factor, sorted table?, etc)
- Using statistics (e.g., selection cardinality)

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

Let us menti….

Please complete the Unit of Study Survey (USS)!

Important note: When you complete your Survey, this will give you an entry into the prize draw for a range of **JB HiFi Giftcards totalling $2500.**

THE UNIVERSITY OF SYDNEY

Unit of Study Survey (USS) now open!

How to make your USS feedback count

Your Unit of Study Survey (USS) feedback is **confidential**.

It's a way to share what you enjoyed and found most useful in your learning, and to provide constructive feedback. It's also a way to 'pay it forward' for the students coming behind you, so that their **learning experience** in this class is as good, or even better, than your own.

When you complete your USS survey (https://student-surveys.sydney.edu.au), please:

Be specific.
Which class tasks, assessments or other activities helped you to learn? *Why* were they helpful? Which one(s) *didn't* help you to learn? *Why* didn't they work for you?

Be constructive.
What practical changes can you suggest to class tasks, assessments or other activities, to help the next class learn better?

Be relevant.
Imagine you are the teacher. What sort of feedback would you find most useful to help make your teaching more effective?

Joins

› **Join** is the **most used operation** in SQL queries!

- It is also usually the ***most expensive operation*** to execute in terms of **I/Os**!
- Therefore, there is a need to **optimize** the **join operation**

Example:

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

› Note that:

- $R \times S$ (Cartesian product) followed by a selection is *semantically the same as* $R \bowtie S$
  - However, the result of $R \times S$ is usually ***significantly*** **larger** than $R \bowtie S$; therefore, *executing $R \times S$ followed by a selection is inefficient.*

› *Instead, use the **equivalent optimized join operation**!*

Join Operator

- Several algorithms that implement joins:
  - ***Nested loop*** join
  - ***Block-nested*** loop join
  - ***Indexed-nested*** loop join

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

Example Table Sizes for Cost Estimates

- We will use the following statistics:
  - ***|R|*** *:* number of tuples in R, stored in $\boldsymbol{b_R}$ pages
  - ***|S|*** *:* number of tuples in S, stored in $\boldsymbol{b_S}$ pages
    - In our example, **R** refers to the relation ***Students*** and **S** refers to the relation ***Enrolled*.**

Supposed we would like to perform a **join** of **Student** and **Enrolled**

Assume

Number of tuples of

Students (|*R*|): 1,000

Enrolled (|*S*|): 10,000

Number of pages of

Students ($\boldsymbol{b_R}$): 100

Enrolled ($\boldsymbol{b_S}$): 400

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 Loop 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 possible 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 Loop Join

- The estimated cost of nested loop 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 + 10,000 ∗ 100 = **1,000,400 disk I/Os** $(b_S + |S| * b_R)$

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

Block-Nested Loop Join

› We assume that the buffer can *only hold two pages for the two relations.* In this case, Block-Nested Loop Join behaves the same as *Page-Oriented Nested Loop Join*. This will be the case whenever we use Block-Nested Loop Join, unless otherwise stated.

- **Block-Nested Loop Join: Variant** of **nested loop 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 Loop 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 Loop 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 loop 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** ($b_S + b_S * b_R$)

Index-Nested Loop 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 loop join, the following conditions must be satisfied:
  - 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 on S** to look **up tuples in S** that satisfy the join condition with tuple *r*.

Cost Analysis: Index-Nested Loop 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**.
- If indexes are available on join attributes of both **R** and **S**, use the table with fewer tuples as the outer table.
- Example: if $c_1 = 4$ for the relation **S** and $c_2 = 3$ for the relation **R**
  - If index on **S** is available: cost is $b_R + (|R| * c_1)$
  - $100 + 1{,}000 * 4 =$ **4,400 disk I/Os**
  - If index on **R** is available: cost is $b_S + (|S| * c_2)$
  - $400 + 10{,}000 * 3 =$ **30,400 disk I/Os**

THE UNIVERSITY OF SYDNEY

Importance of Sorting for DBMS

- SQL queries can specify that the **output** be **sorted** (**ORDER BY**). Additionally, SQL operators (e.g., **JOIN**, **GROUP BY, DISTINCT, UNION, EXCEPT, etc**) can be **implemented efficiently** if the input is **sorted.**
- Example: consider the set operations **UNION, INTERSECT,** and **EXCEPT**
  - Each of the above operations *first eliminates duplicates* from the input tables, and then does the set operation
    - The *expensive* part is *removing* duplicates.
      - If the file is *not sorted*, removing duplicates may require sequentially comparing each value in a record against all other values in the file!
- Another example: **Join operation**
  - If both relations are sorted on the join key, the join operation can be efficiently implemented, i.e., **the total number of I/Os will be equal to the sum of the size (in pages) of each relation** in the case of a **natural join**. Contrast this with the nested-loop join.

External Sorting

- The **Sort-Merge Join** (also known as **Merge-Join**) is a **join algorithm** used in one of the **implementations** of a **Join**.
- The **most expensive** part of performing a **sort-merge join** is **arranging** for **both inputs (tables)** to the algorithm to be presented in **sorted order**.
  - The key idea of the **sort-merge join algorithm** is to first **sort** the relations by the **join attribute**, so that **linear scans** can match the **corresponding values** at the **same time**.
- For small tables that fit in memory, techniques like *QuickSort* may be used.
  - However, when the database **is large**, we **cannot** use these techniques: e.g., sort 10GB of data with 4GB of RAM…

    Solution: **External Sorting** called **External Merge-Sort**

External Merge Sort Algorithm

Let ***B*** denote the **buffer size (**in **pages)**. ***N*** is the **size** (in **pages**) of the **file (table)**.

Three main **steps**:

1. **Create** sorted ***runs***. (A **run** is the name of a **sorted subset** of the **file 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 buffer
   (b) Sort the in-buffer pages
   (c) Write the 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**.

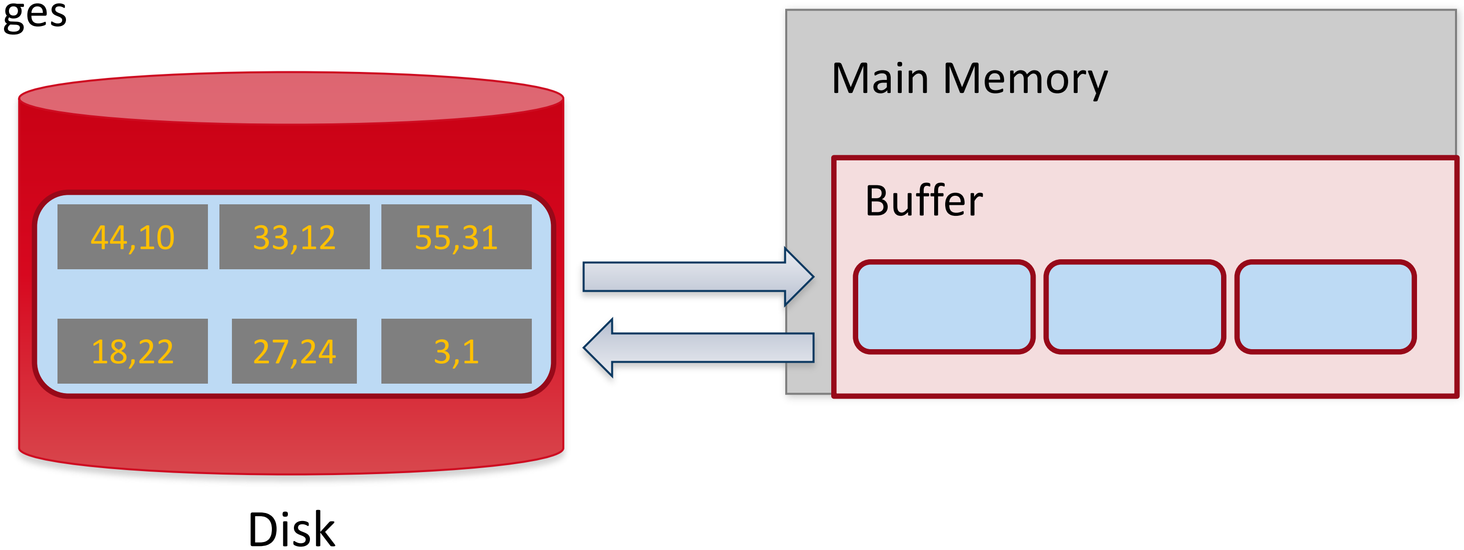
2. **Merge** each contiguous group of *B-1* runs into *1* run: (*B-1*)-way merge.

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

Step 1: Create Sorted Runs

Example:

- A file consists of **N=6** pages
- **B=3** buffer pages

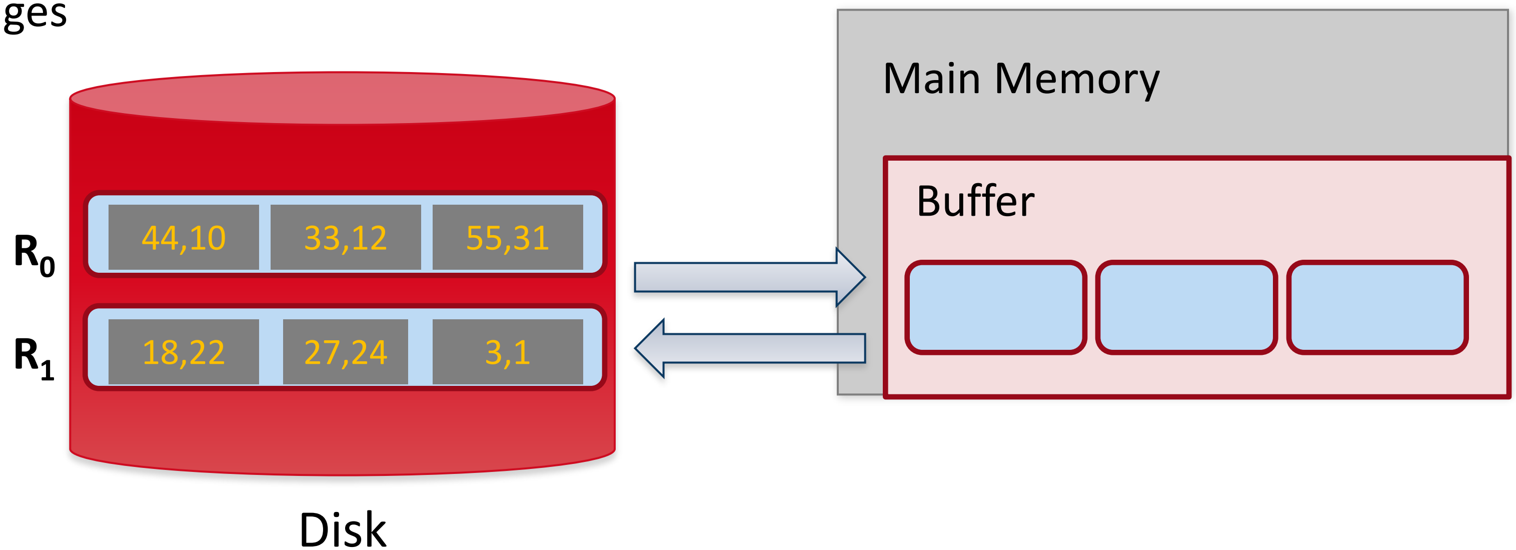


1. Split into *runs* small enough to **sort in memory**

Step 1: Create Sorted Runs

Example:

- A file consists of 6 pages
- 3 buffer pages

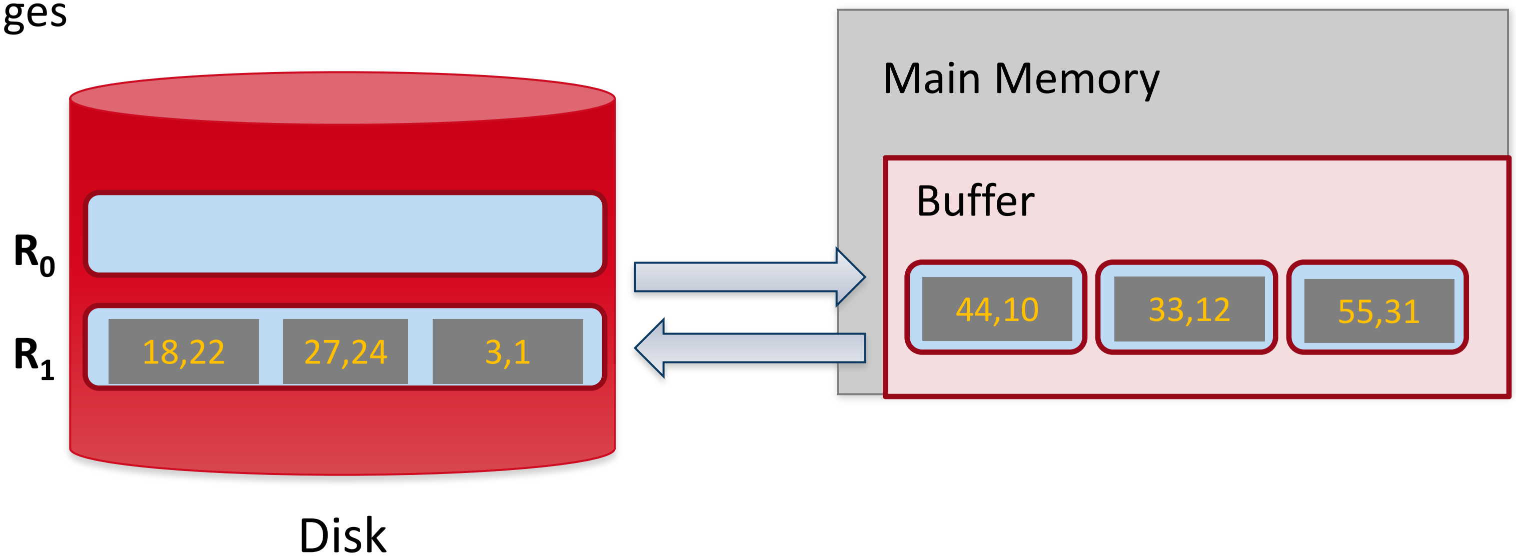


1. Split into *runs* small enough to **sort in memory**

Step 1: Create Sorted Runs

Example:

- A file consists of 6 pages
- 3 buffer pages

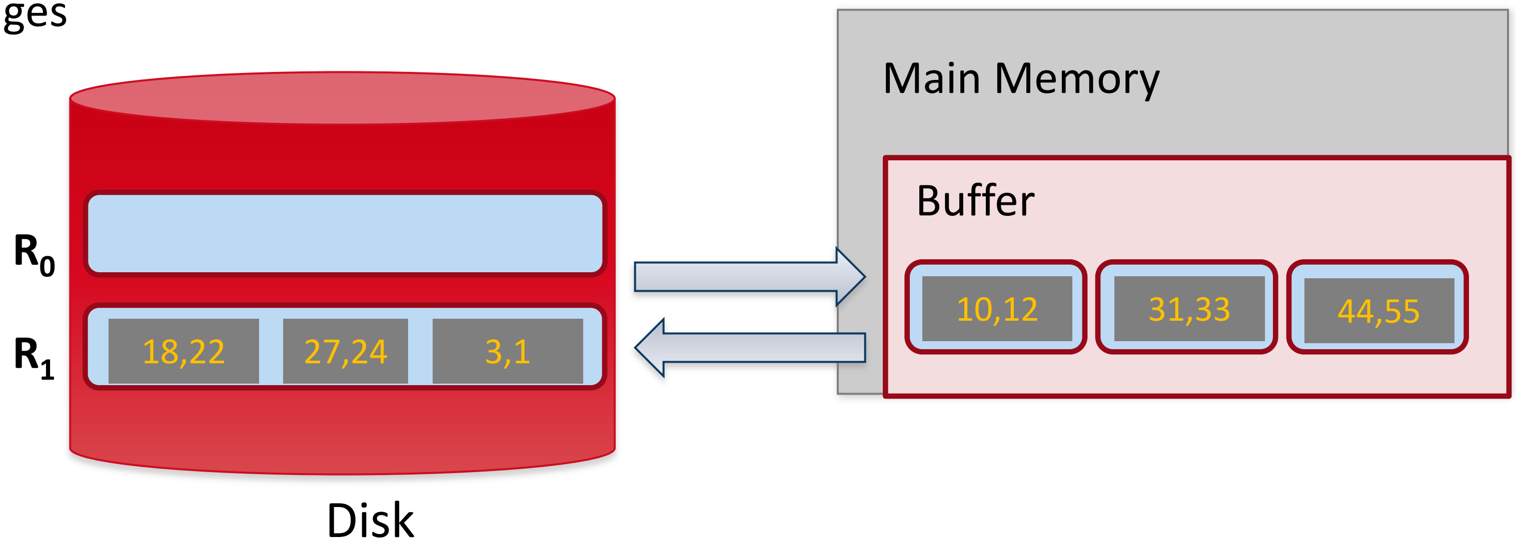


2. Load run R_0 into main memory

Step 1: Create Sorted Runs

Example:

- A file consists of 6 pages
- 3 buffer pages

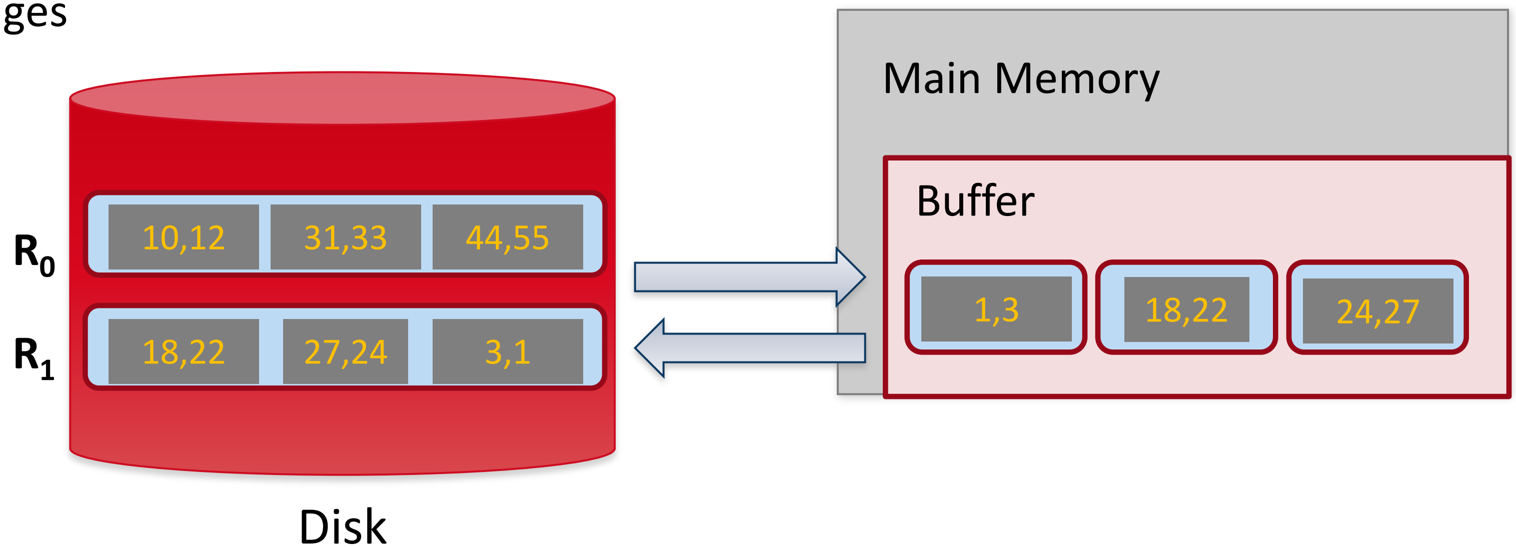


3. Sort run R_0 in main memory, and write back R_0 to disk

Example Step 1: Create Sorted Runs

Example:

- A file consists of 6 pages
- 3 buffer pages

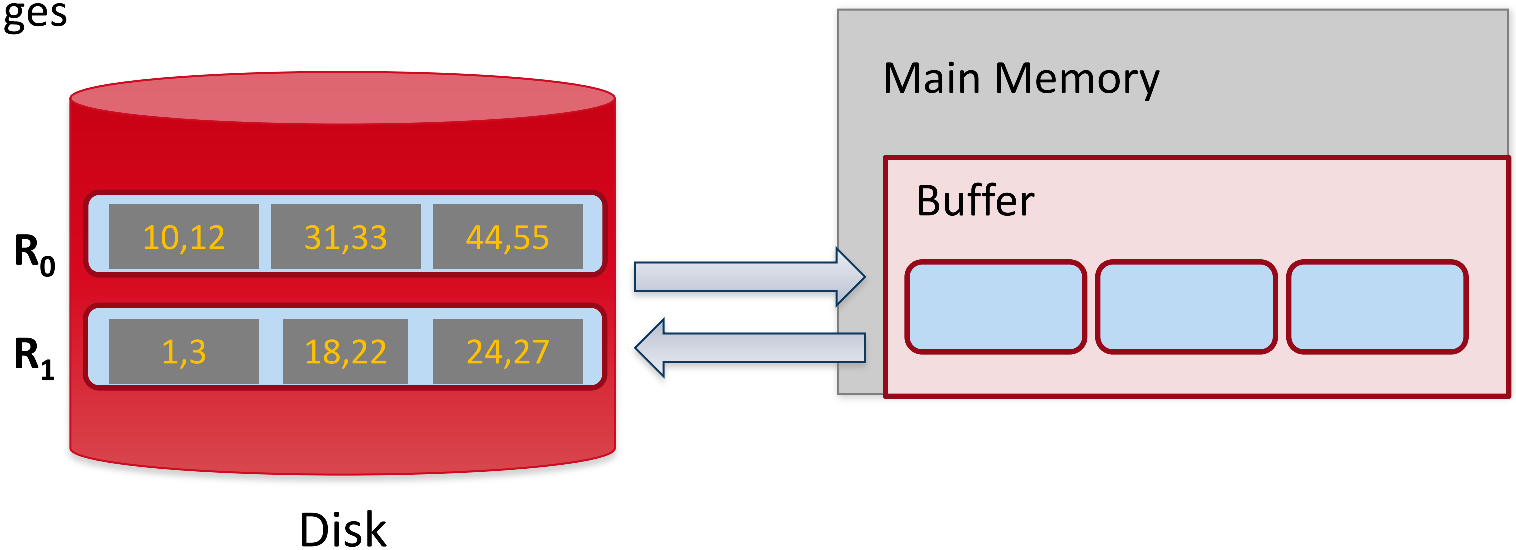


4. Similarly, load R_1 into main memory, sort it, and write it back to disk

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). N is the size (# of records) of the file.

Three main steps:

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 buffer

   (b) Sort the in-buffer 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.

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

   Read the *first page of each run* R_i into its *allocated buffer page*

   i. **repeat**

   1. Select the first record (in sorted 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 allocated to run R_i **then** read the next page of the run R_i into the buffer. If none is left in run R_i, then go to next run.

   until all input buffer pages are empty:

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

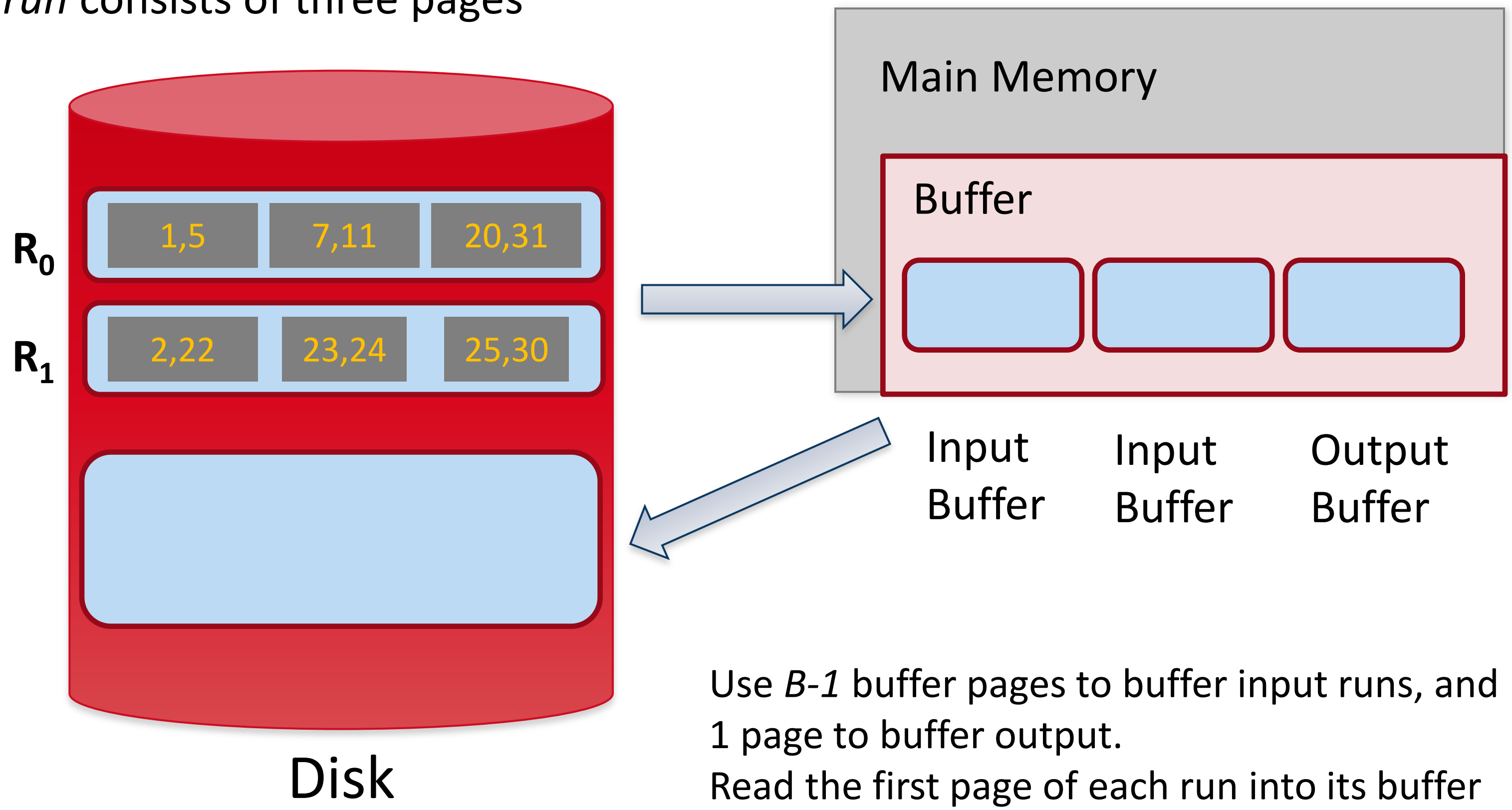
Step-2: Merge Sorted Runs

Example:

- Each *input run* consists of three pages

Input:
Two sorted runs

Output:
One *merged* sorted run



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

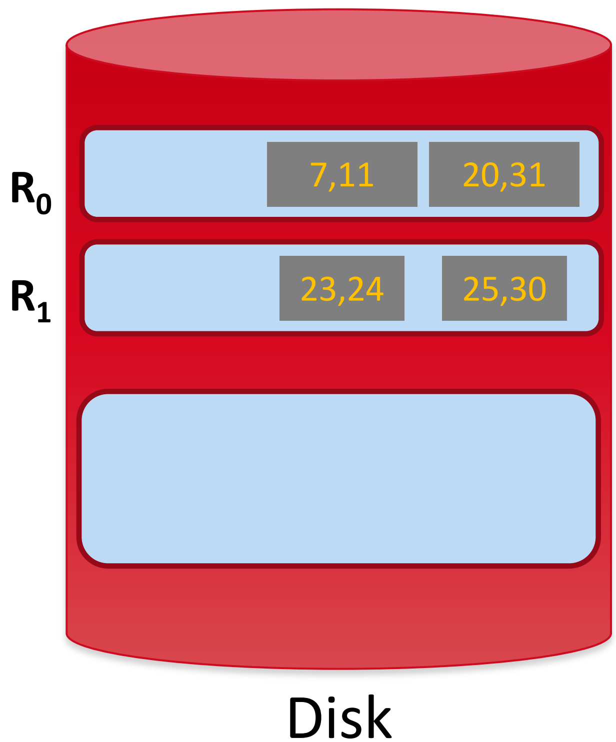
Step-2: Merge Sorted Runs

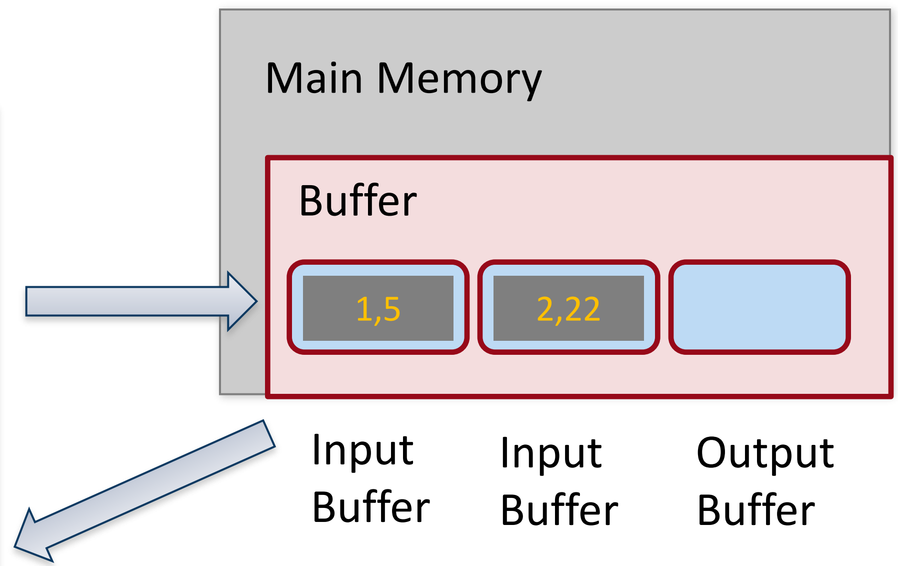
Example:

- Each input run consists of three pages

Input:
Two sorted runs

Output:
One *merged* sorted run



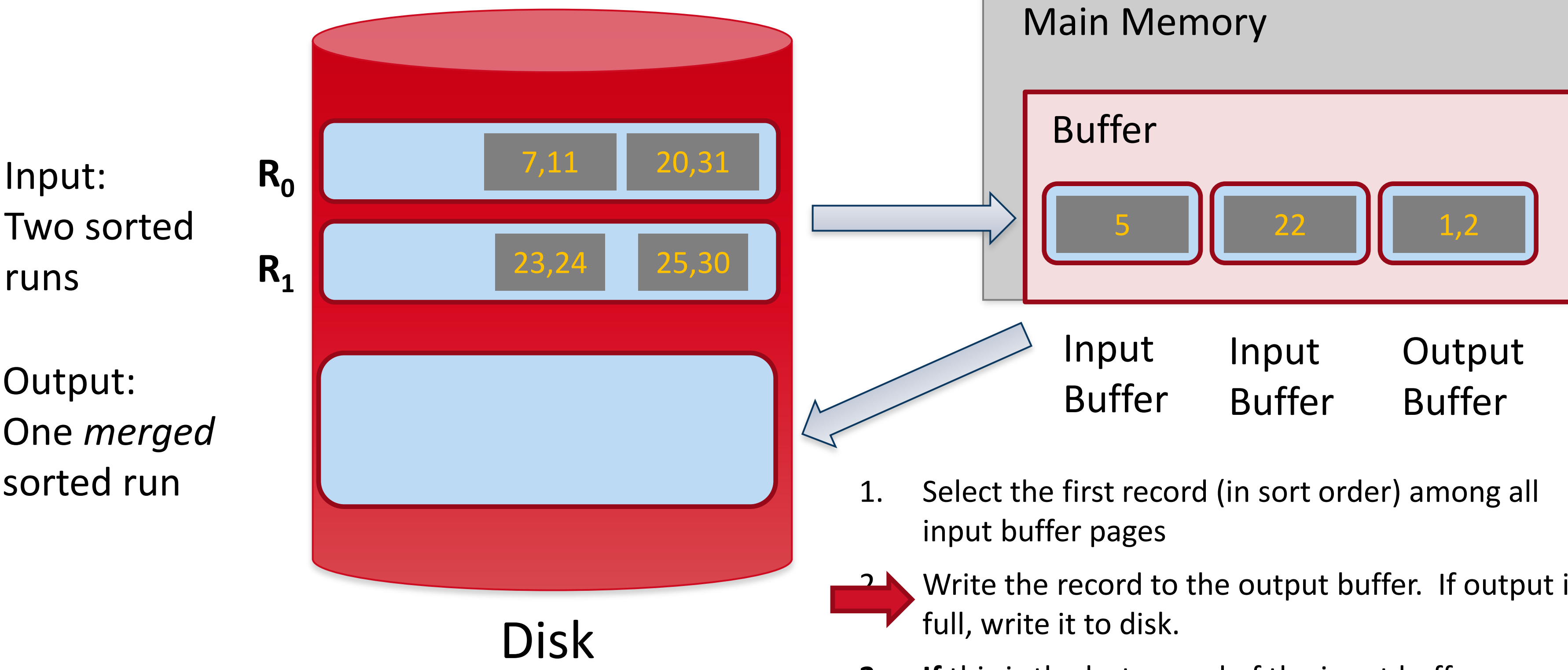


1. Select the first record (in sort order) among all input buffer pages
2. Write the record to the output buffer. If output is full, write it to disk.
3. **If** this is the last record of the input buffer page allocated to run R_i **then** read the next page of the run R_i into the buffer. If none is left in run R_i, then go to next run.

Step-2: Merge Sorted Runs

Example:

- Each input run consists of three pages

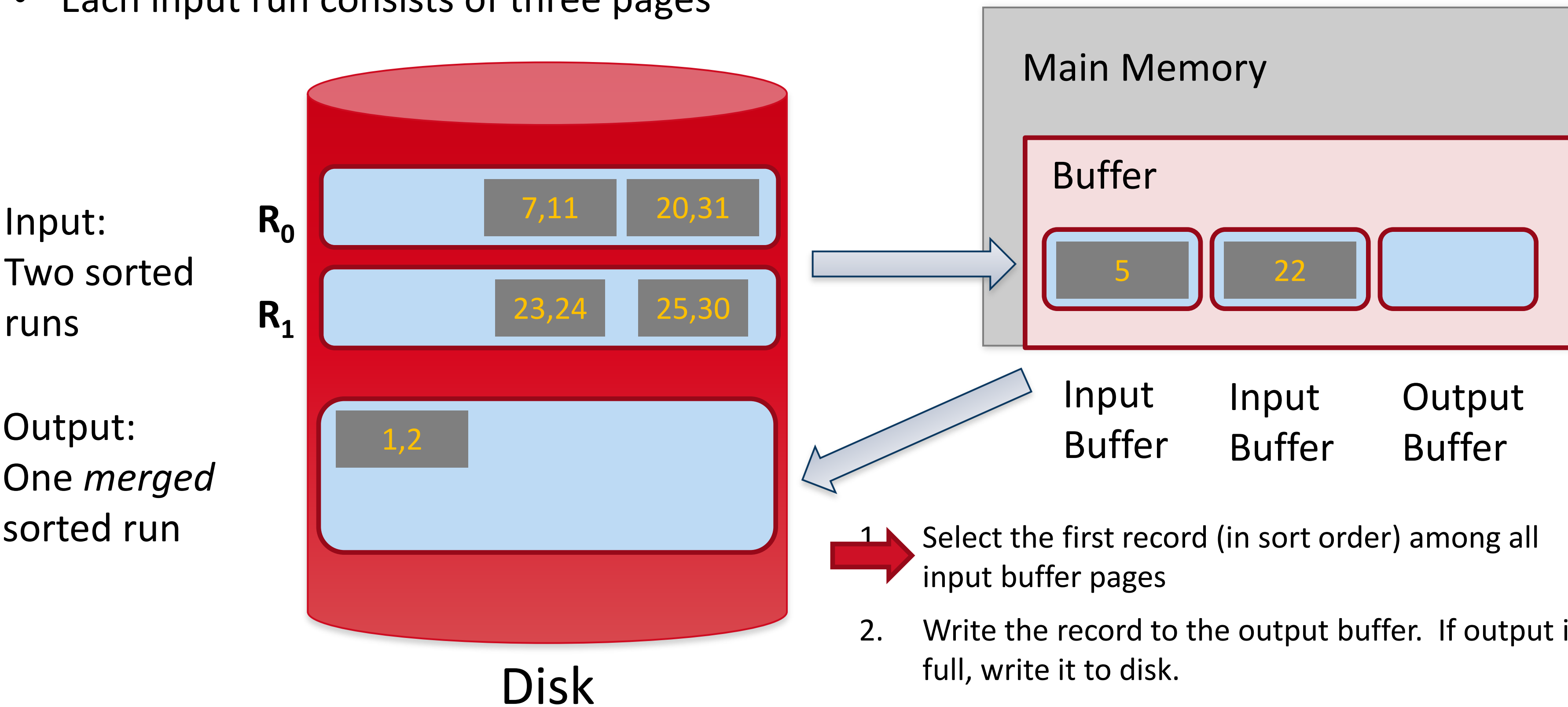


1. Select the first record (in sort order) among all input buffer pages
2. Write the record to the output buffer. If output is full, write it to disk.
3. **If** this is the last record of the input buffer page allocated to run R_i **then** read the next page of the run R_i into the buffer. If none is left in run R_i, then go to next run.

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 allocated to run R_i **then** read the next page of the run R_i into the buffer. If none is left in run R_i, then go to next run.

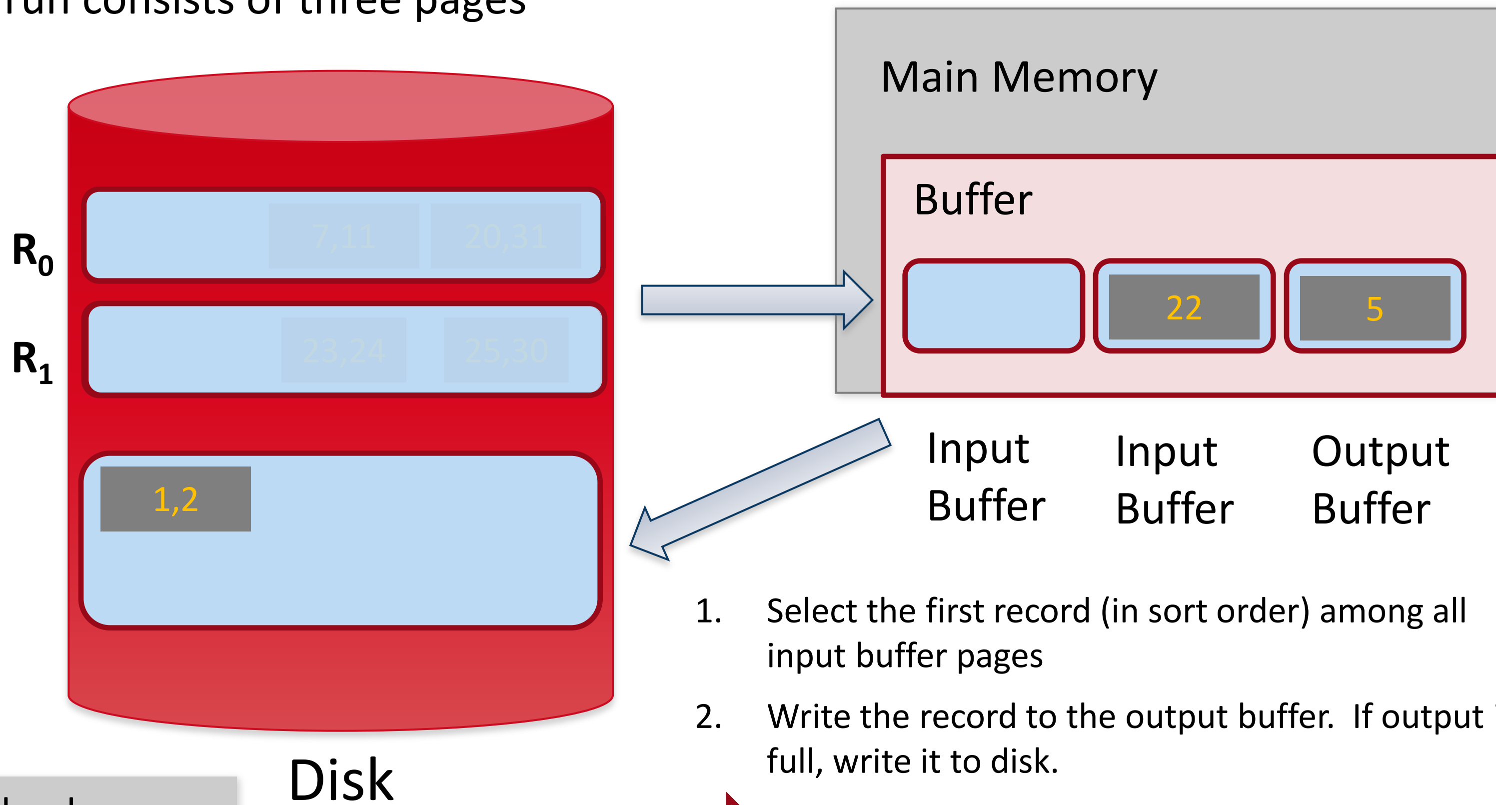
Step-2: Merge Sorted Runs

Example:

- Each input run consists of three pages

Input:
Two sorted
runs

Output:
One *merged*
sorted run



Which run to load a page from next?

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 allocated to run R_i **then** read the next page of the run R_i into the buffer. If none is left in run R_i, then go to next run.

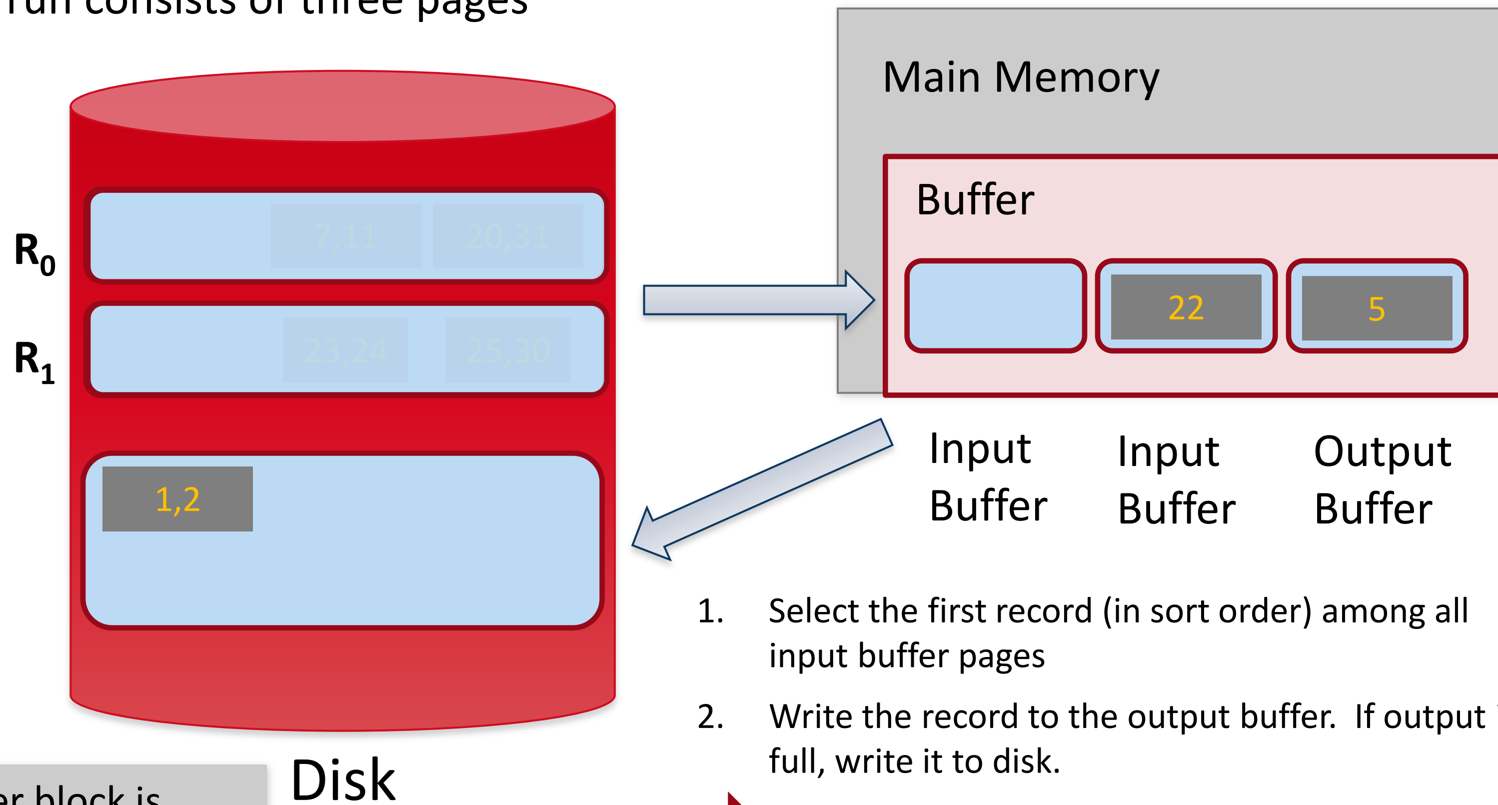
Step-2: Merge Sorted Runs

Example:

- Each input run consists of three pages

Input:
Two sorted runs

Output:
One *merged* sorted run



The empty buffer block is reserved for R_0 … so we should load from R_0 if it is not empty!

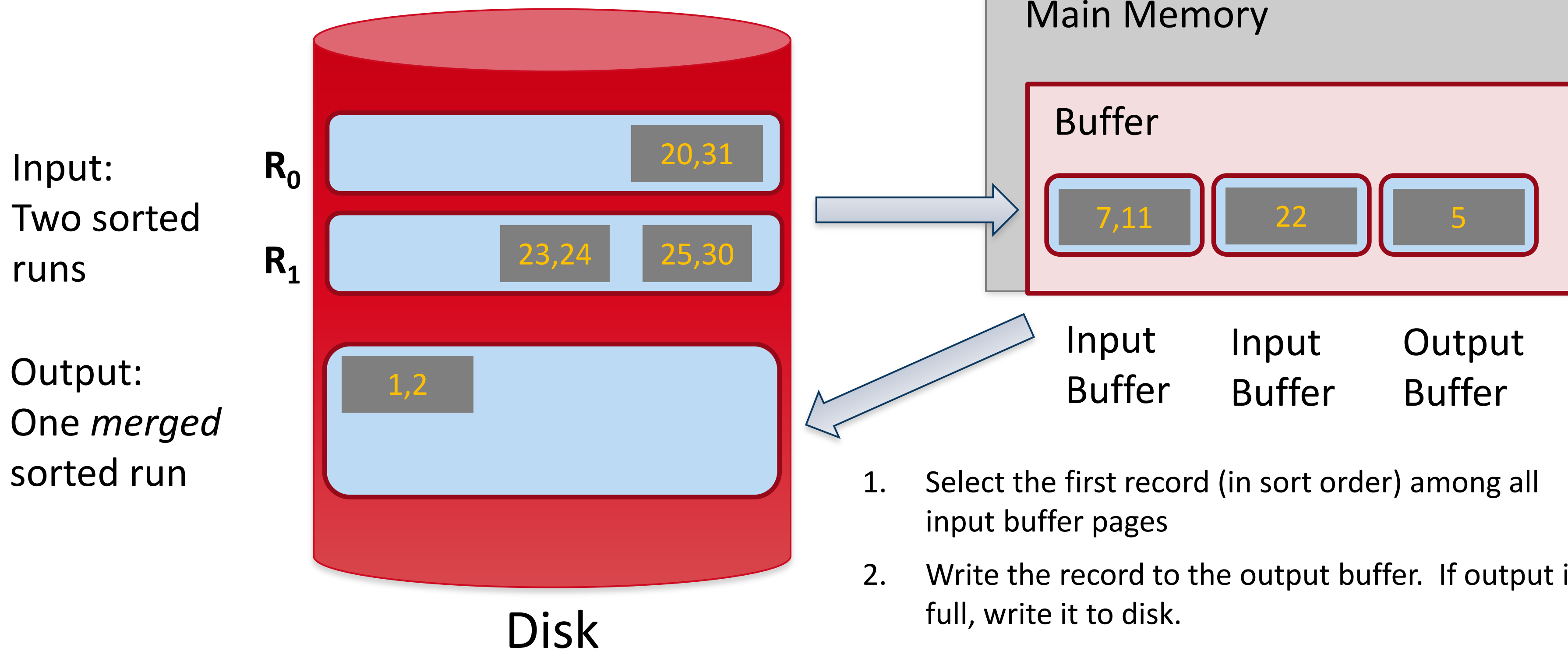
1. Select the first record (in sort order) among all input buffer pages
2. Write the record to the output buffer. If output is full, write it to disk.

If this is the last record of the input buffer page allocated to run R_i **then** read the next page of the run R_i into the buffer. If none is left in run R_i, then go to next run.

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 allocated to run R_i **then** read the next page of the run R_i into the buffer. If none is left in run R_i, then go to next run.

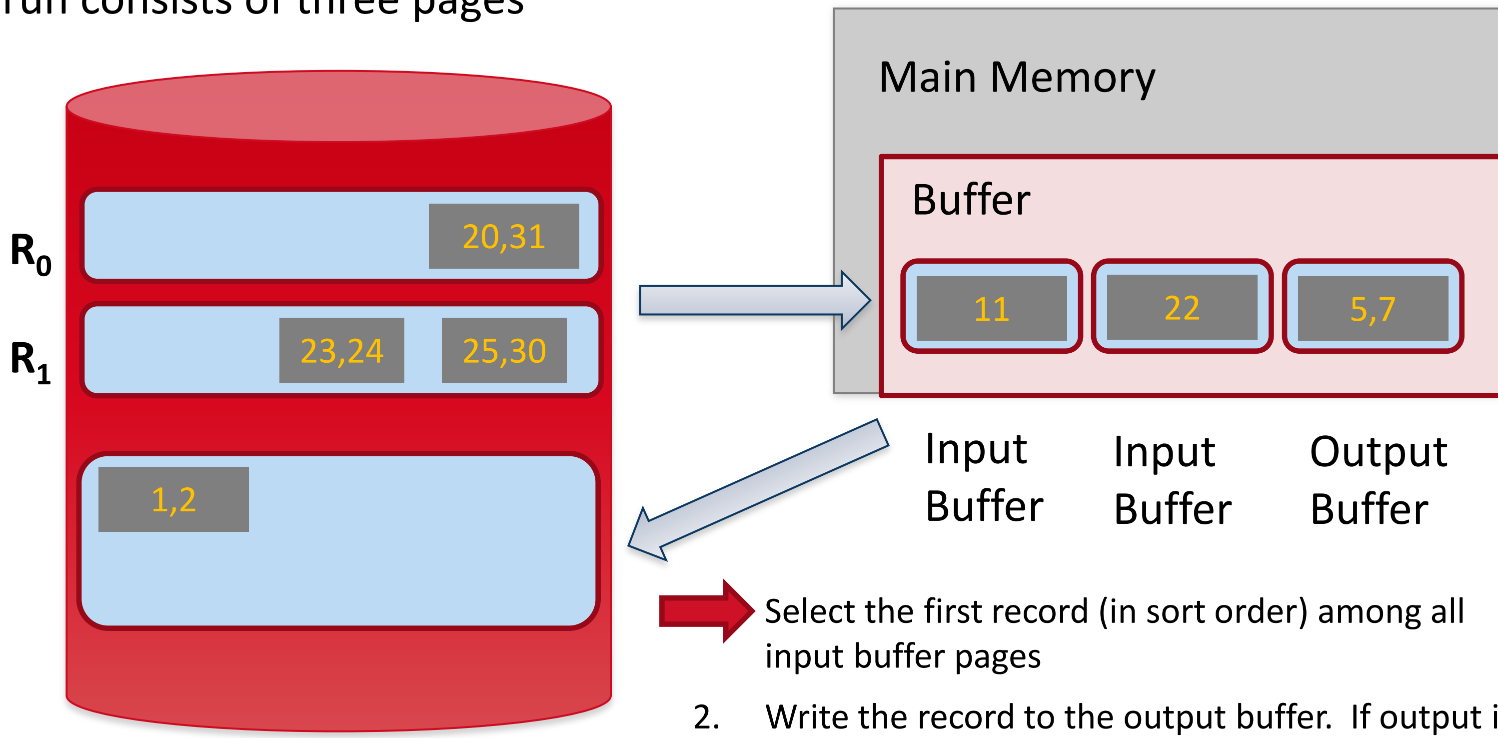
Step-2: Merge Sorted Runs

Example:

- Each input run consists of three pages

Input:
Two sorted runs

Output:
One *merged* sorted run



Disk

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

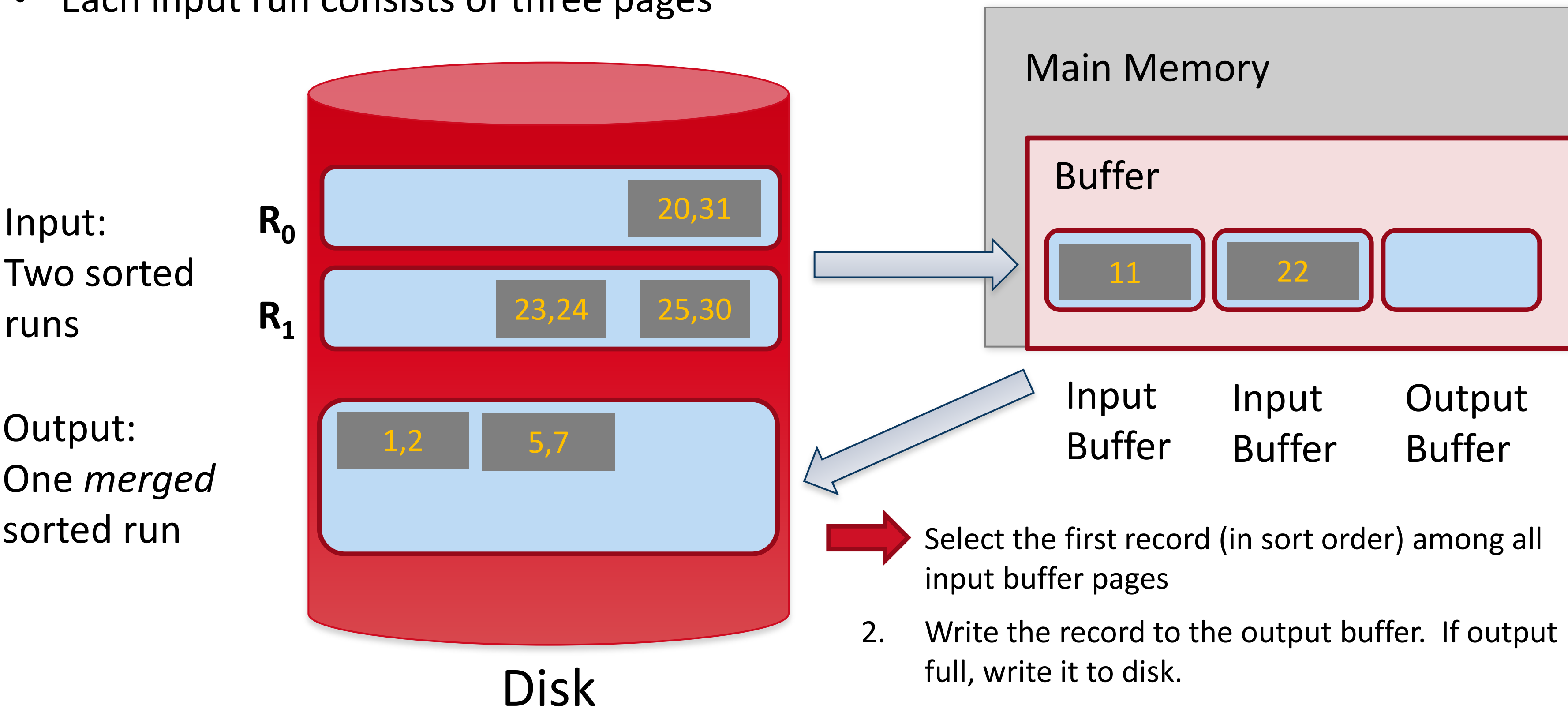
2. Write the record to the output buffer. If output is full, write it to disk.

3. **If** this is the last record of the input buffer page allocated to run R_i **then** read the next page of the run R_i into the buffer. If none is left in run R_i, then go to next run.

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 allocated to run R_i **then** read the next page of the run R_i into the buffer. If none is left in run R_i, then go to next run.

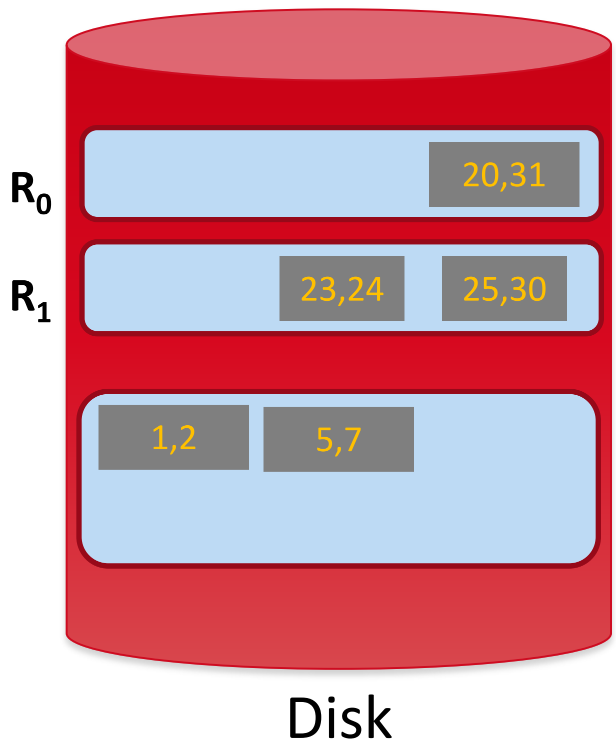
Step-2: Merge Sorted Runs

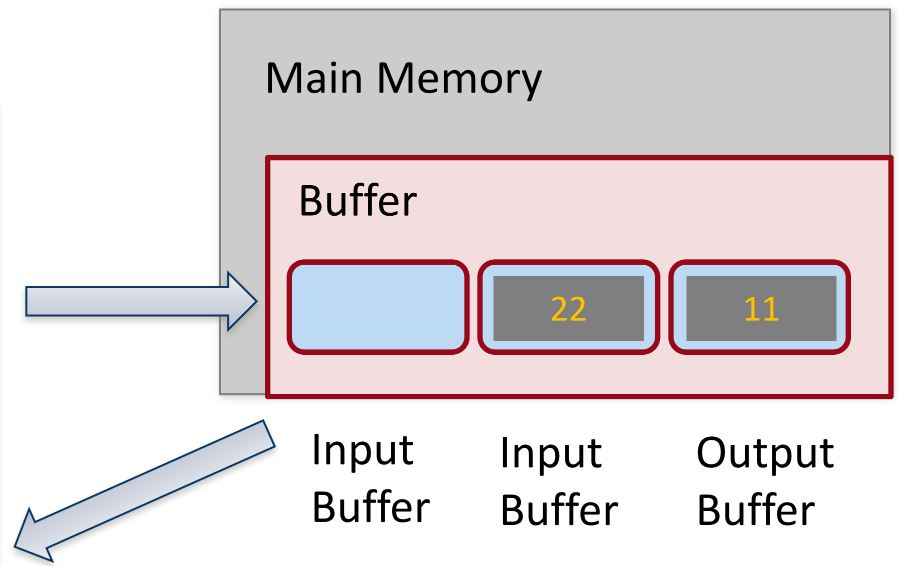
Example:

- 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 allocated to run R_i **then** read the next page of the run R_i into the buffer. If none is left in run R_i, then go to next run.

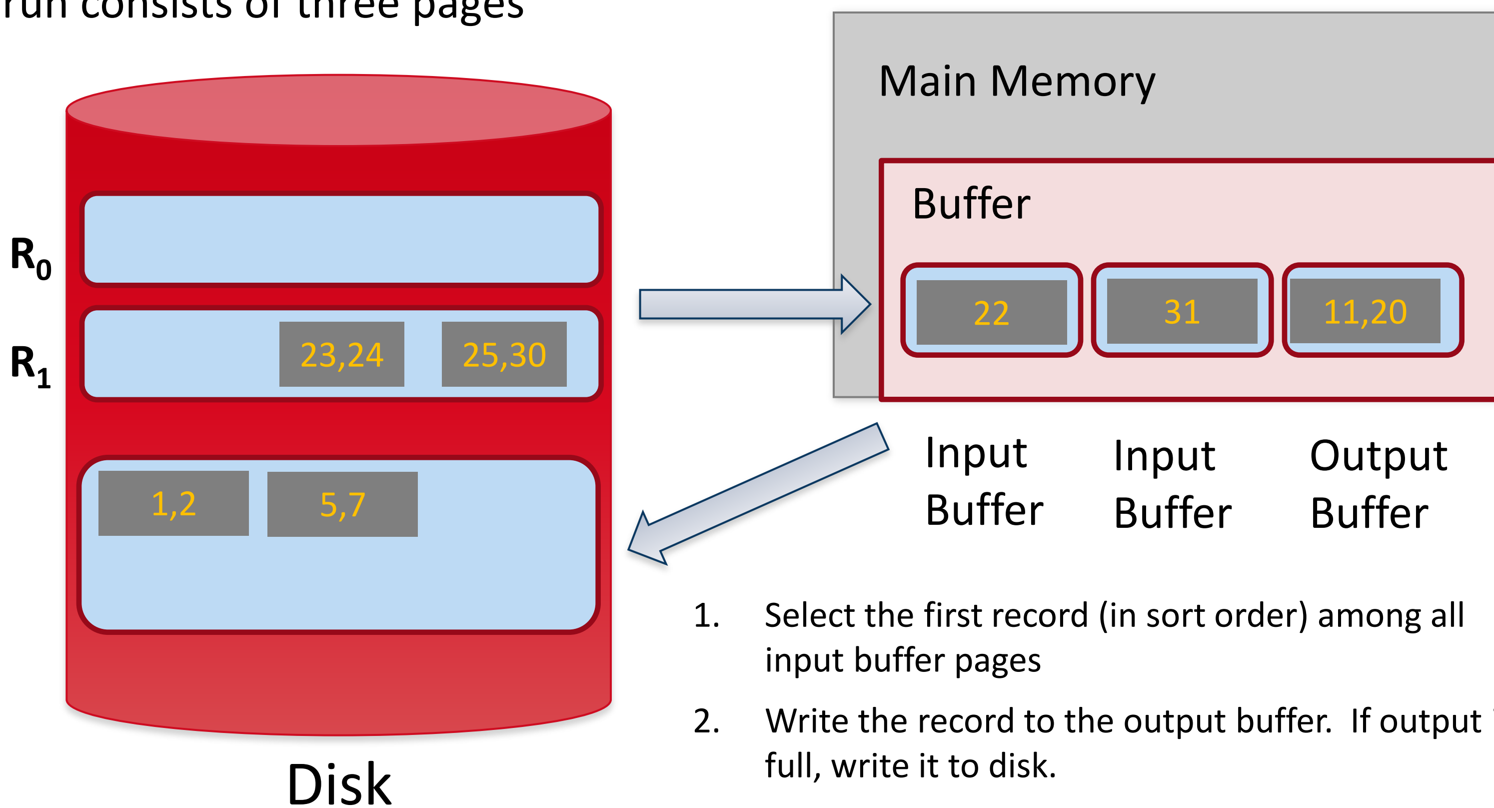
Step-2: Merge Sorted Runs

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 allocated to run R_i **then** read the next page of the run R_i into the buffer. If none is left in run R_i, then go to next run.

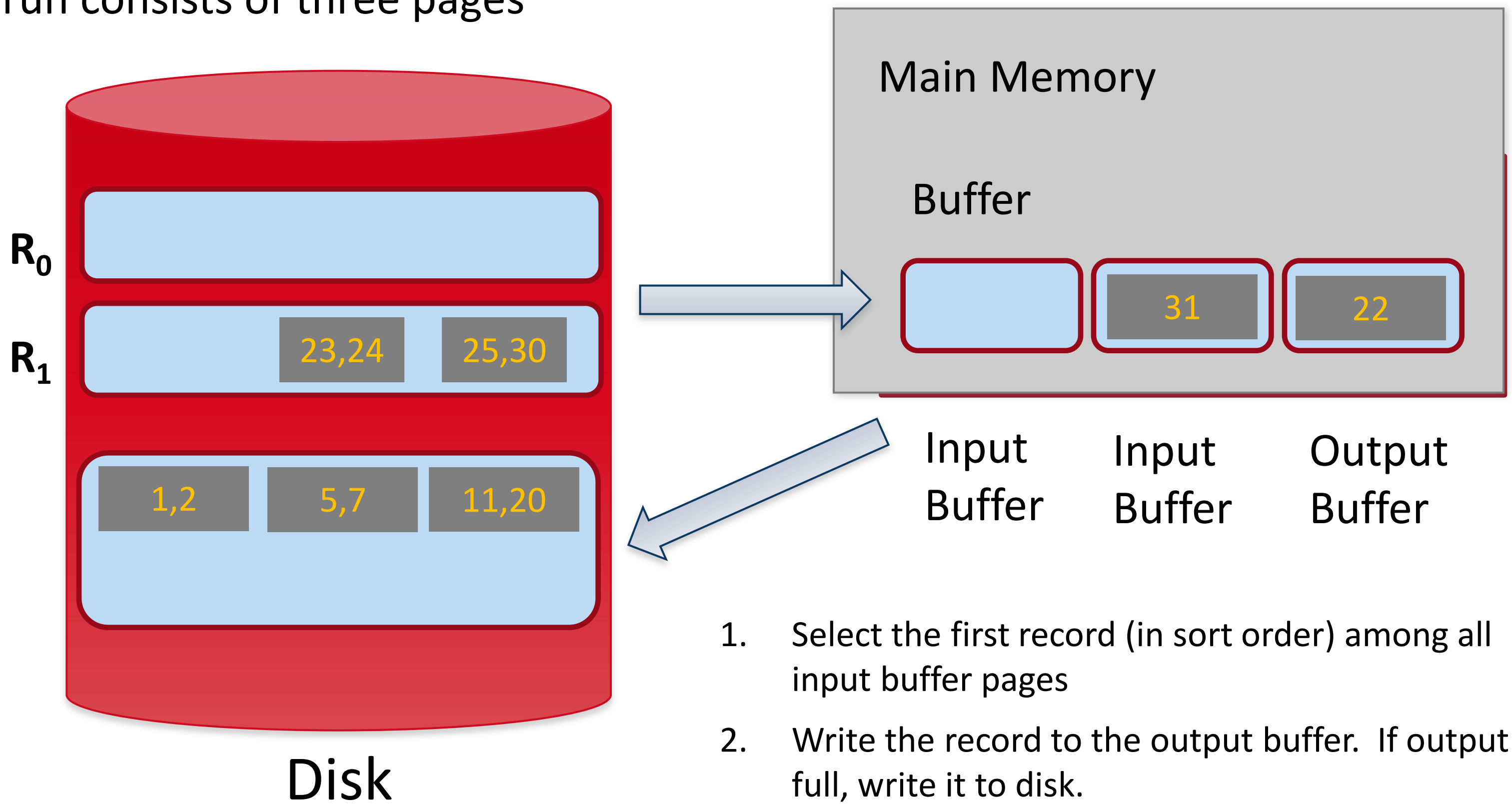
Step-2: Merge Sorted Runs

Example:

- Each input run consists of three pages

Input:
Two sorted runs

Output:
One *merged* sorted run

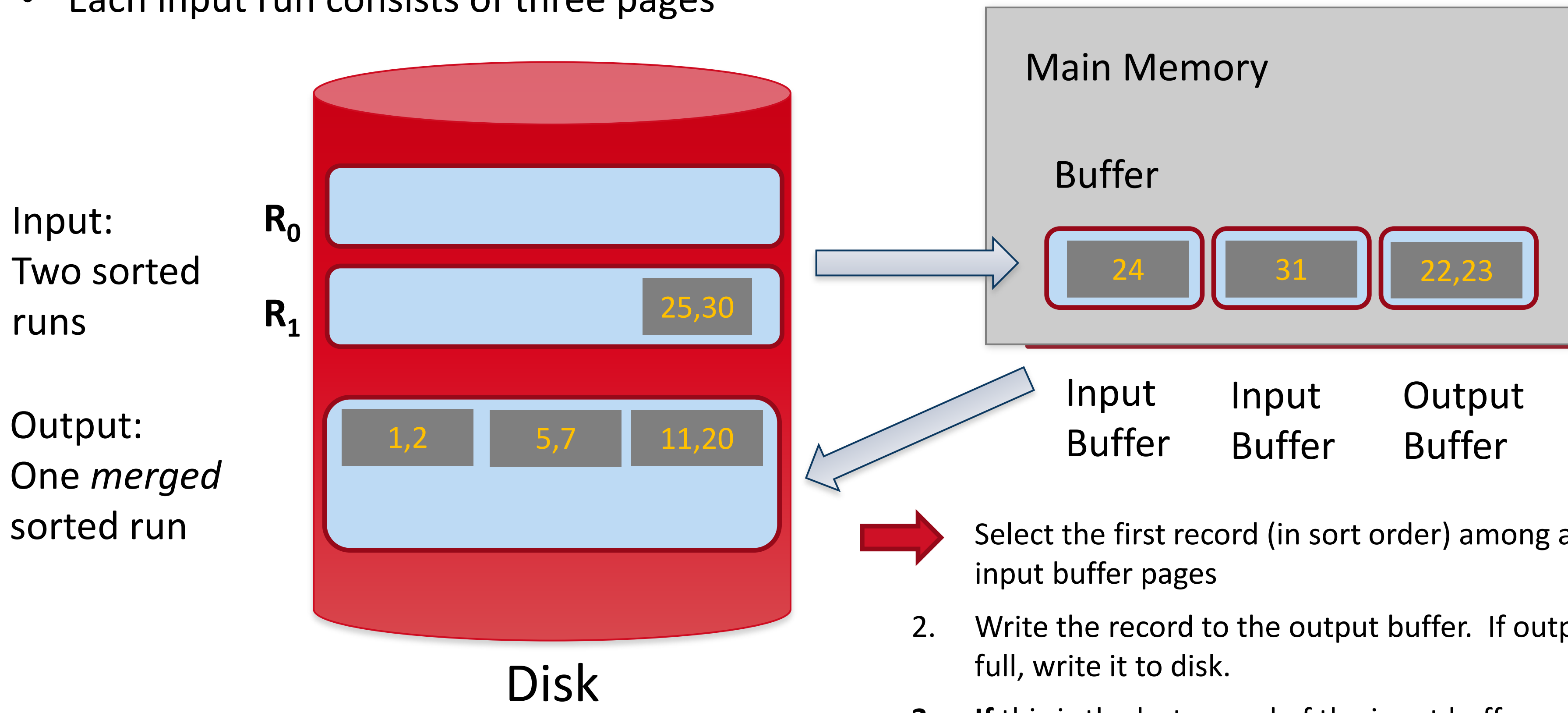


1. Select the first record (in sort order) among all input buffer pages
2. Write the record to the output buffer. If output is full, write it to disk.
3. **If** this is the last record of the input buffer page allocated to run R_i **then** read the next page of the run R_i into the buffer. If none is left in run R_i, then go to next run.

Step-2: Merge Sorted Runs

Example:

- Each input run consists of three pages

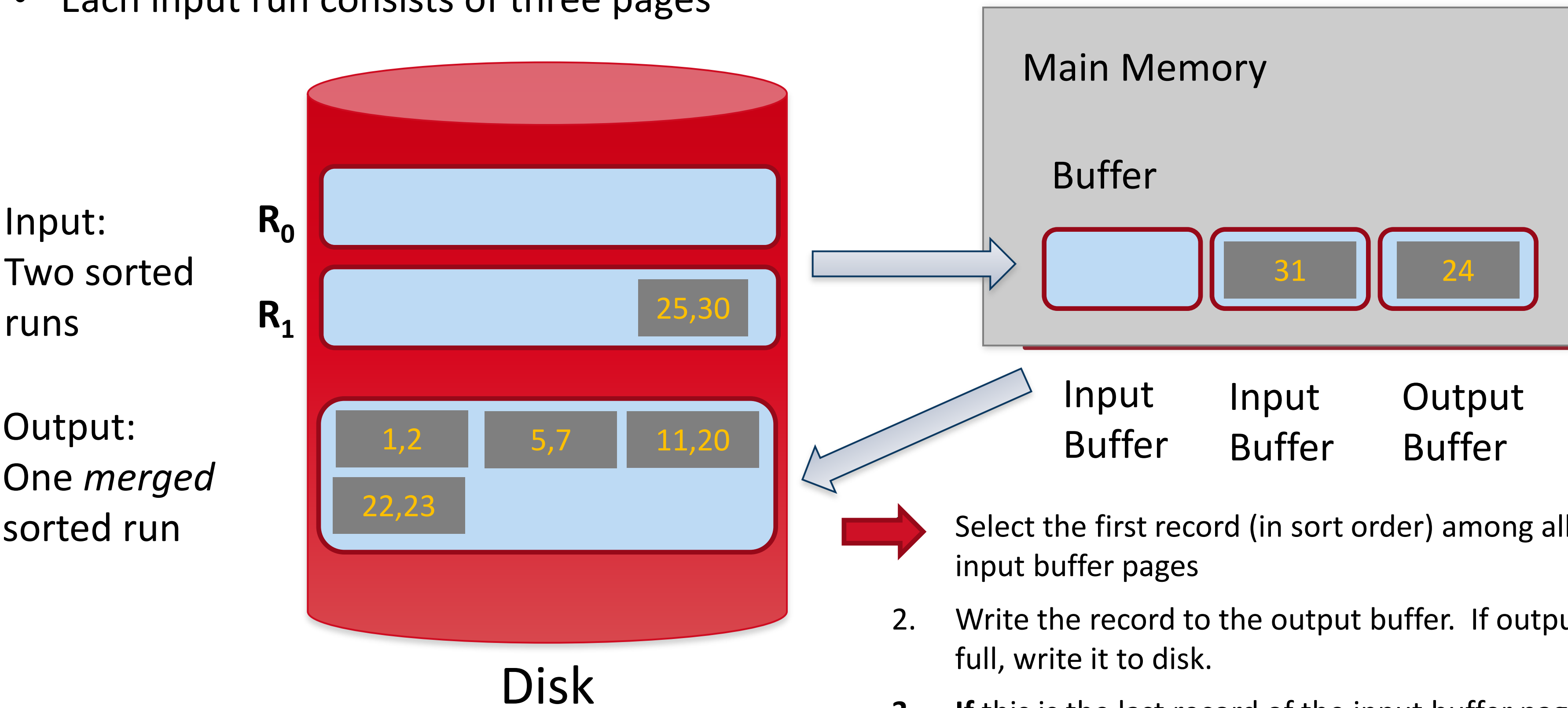


1. Select the first record (in sort order) among all input buffer pages
2. Write the record to the output buffer. If output is full, write it to disk.
3. **If** this is the last record of the input buffer page allocated to run R_i **then** read the next page of the run R_i into the buffer. If none is left in run R_i, then go to next run.

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 allocated to run R_i **then** read the next page of the run R_i into the buffer. If none is left in run R_i, then go to next run

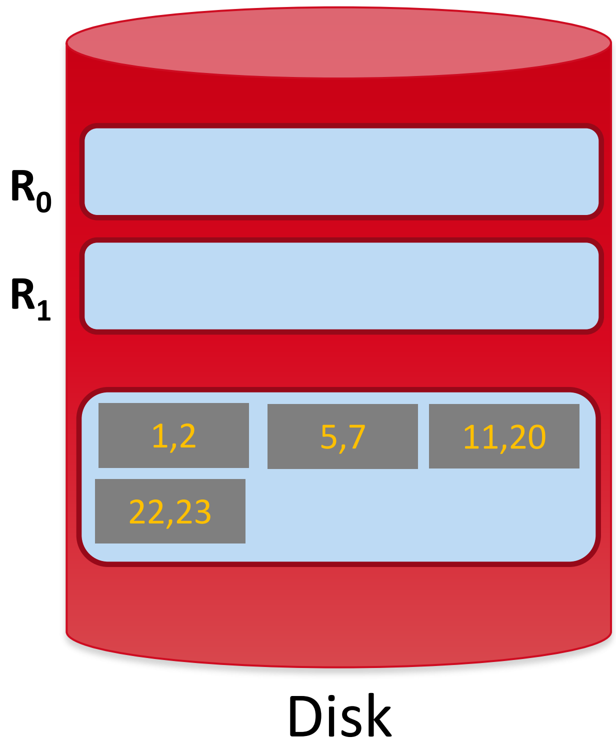
Step-2: Merge Sorted Runs

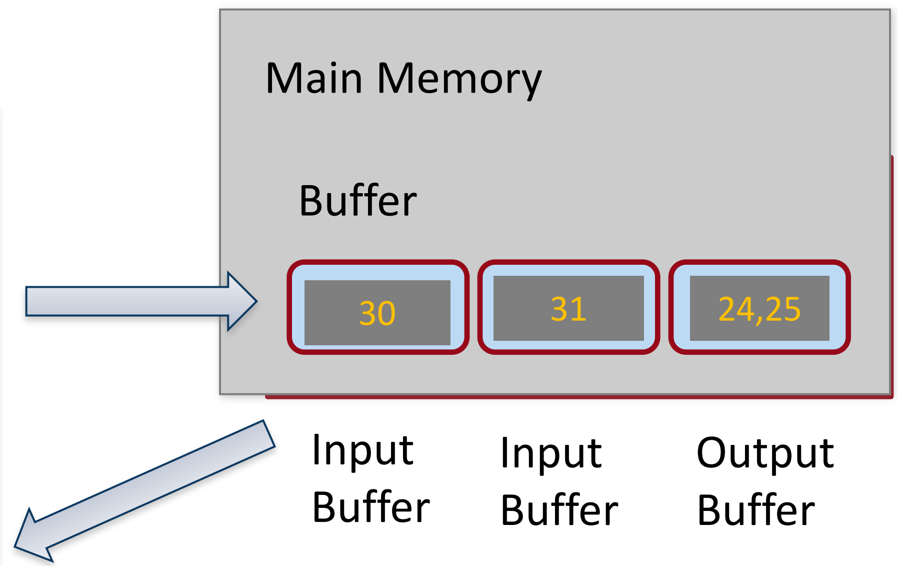
Example:

- 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 record of the input buffer page allocated to run R_i **then** read the next page of the run R_i into the buffer. If none is left in run R_i, then go to next run.

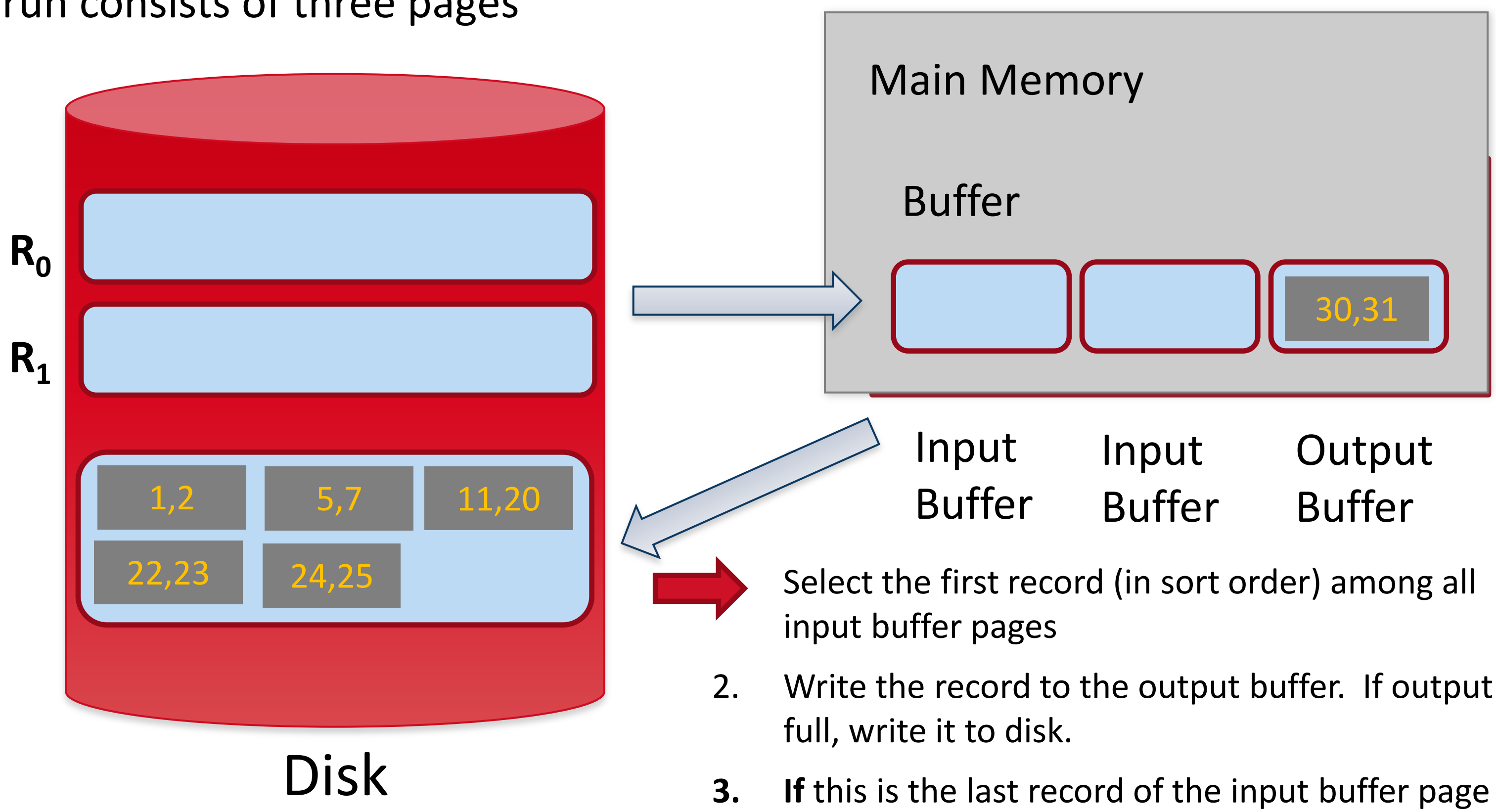
Step-2: Merge Sorted Runs

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 record of the input buffer page allocated to run R_i **then** read the next page of the run R_i into the buffer. If none is left in run R_i, then go to next run

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 allocated to run R_i **then** read the next page of the run R_i into the buffer. If none is left in run R_i, then go to next run.

   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 $\lceil \log_{(B-1)} (N/B) \rceil + 1$. In this case here, the value is $\lceil \log_2 2 \rceil + 1 = 1 + 1 =$ **2 passes**

Outline

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

Step 3: Evaluation of Operations

- **Two approaches:**
  - **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** in a **pipeline**
    - In other words: **Output** of **one operator** is **directly input** to **next operator**

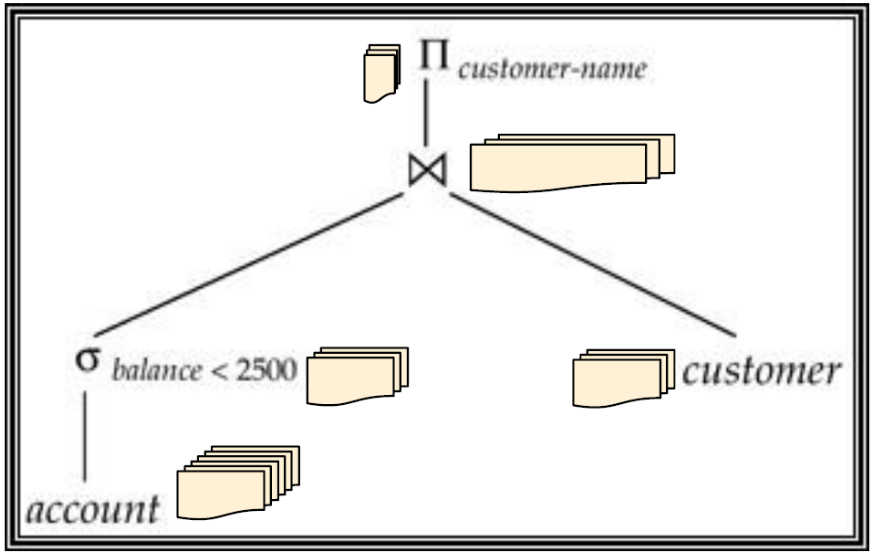
Materialization

› **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.

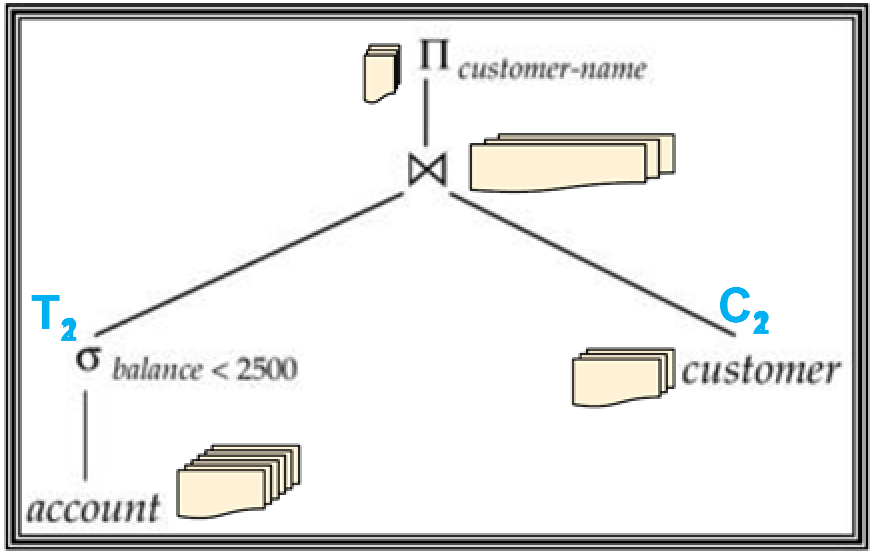
› Example

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 projection on **customer-name**.

- Advantage: Can always apply materialized evaluation
- Disadvantage: Costs can be quite high



- **Pipelined evaluation**: **evaluate** several **operations concurrently**, *passing* the *results* of one *operation* on to the *next*. *Each tuple goes through a* ***pipeline*** *of* ***operations***.
- Example
  1. Find next tuple matching $\sigma_{balance<2500}(account)$
     a) Join matching tuple with tuples of customer until new tuples are generated
     b) Project customer name from joined tuples
     c) Repeat for all join results
  2. Repeat for next selection result
- *Much cheaper* than materialization:
  - *no need* to store a *temporary table*
  - Issues: If algorithm requires **sorted output**, pipelining may not work well if data is not already sorted.



THE UNIVERSITY OF SYDNEY

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
  - Joins
    - Nested loop
    - Block-nested loop
    - Indexed-nested loop
  - External Merge Sort (for Merge-Join)

References

- Ramakrishnan/Gehrke – Chapters 13 and 14
- Kifer/Bernstein/Lewis – Chapter 10
- Garcia-Molina/Ullman/Widom – Chapter 15

Next Week: Review Session

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

See you next week!