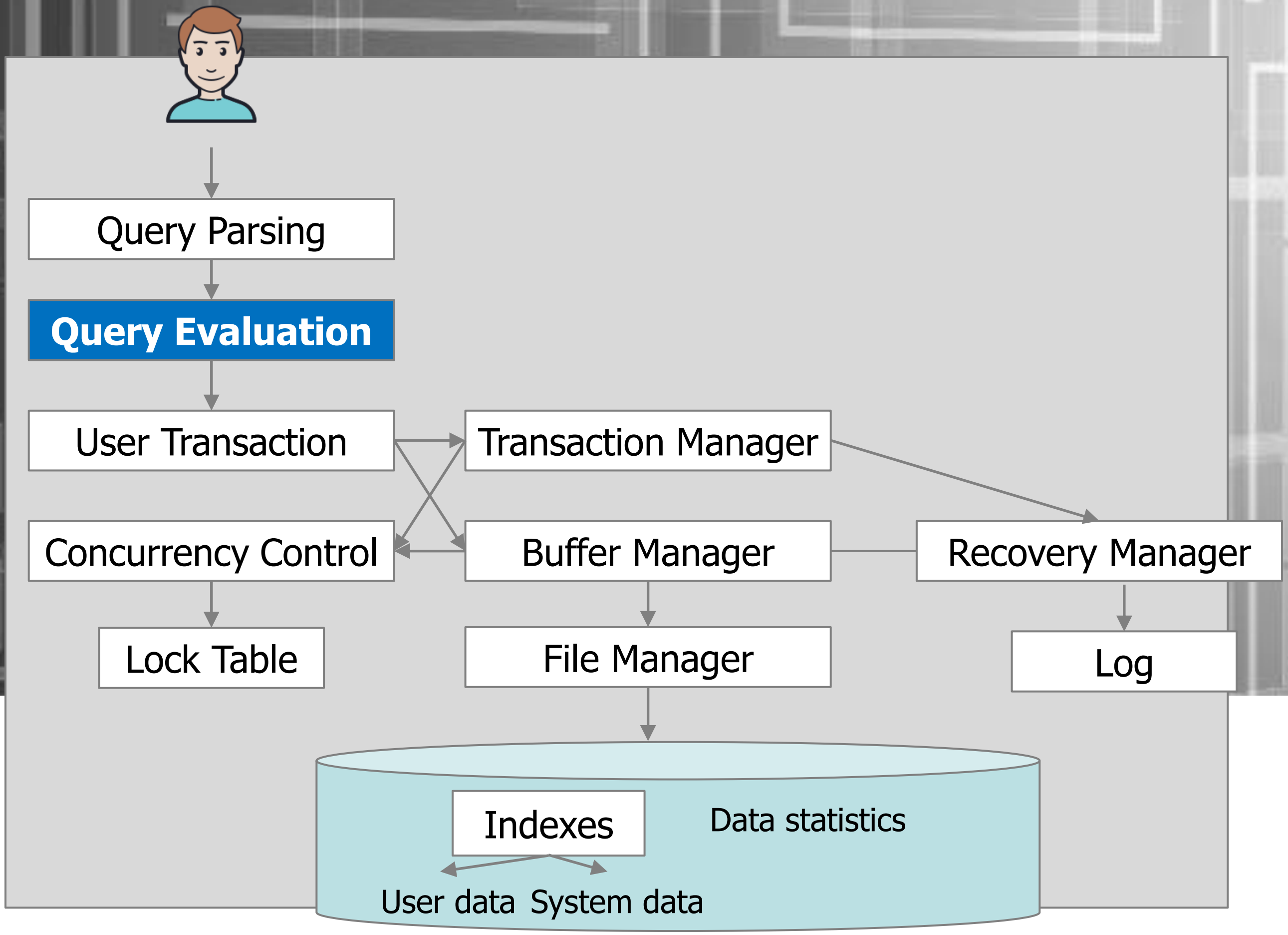
M146 Database Systems
Spring 2021

Georgia Koutrika

QUERY EVALUATION

1. How to execute joins?



SQL Processing

```
SELECT e.last_name, j.job_title, d.department_name
FROM   hr.employees e, hr.departments d, hr.jobs j
WHERE  e.department_id = d.department_id
AND    e.job_id = j.job_id
AND    e.last_name LIKE 'A%';

Execution Plan
----------------------------------------------------------
Plan hash value: 975837011

------------------------------------------------------
| Id| Operation                      | Name        |Rows|By
------------------------------------------------------
| 0 | SELECT STATEMENT               |             |  3 | 189 | 7(15)| 00:00:01 |
|*1 |  HASH JOIN                     |             |  3 | 189 | 7(15)| 00:00:01 |
|*2 |   HASH JOIN                    |             |  3 | 141 | 5(20)| 00:00:01 |
| 3 |    TABLE ACCESS BY INDEX ROWID| EMPLOYEES    |  3 |  60 | 2 (0)| 00:00:01 |
|*4 |     INDEX RANGE SCAN           | EMP_NAME_IX |  3 |     | 1 (0)| 00:00:01 |
| 5 |    TABLE ACCESS FULL           | JOBS        | 19 | 513 | 2 (0)| 00:00:01 |
| 6 |   TABLE ACCESS FULL            | DEPARTMENTS | 27 | 432 | 2 (0)| 00:00:01 |
--------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   1 - access("E"."DEPARTMENT_ID"="D"."DEPARTMENT_ID")
   2 - access("E"."JOB_ID"="J"."JOB_ID")
   4 - access("E"."LAST_NAME" LIKE 'A%')
       filter("E"."LAST_NAME" LIKE 'A%')
```

Join methods: techniques to execute a join of two tables

Access paths: techniques for retrieving data from the database.

What you will learn about in this section

1. Joins
2. Nested Loop Join (NLJ)
3. Block Nested Loop Join (BNLJ)
4. Index Nested Loop Join (INLJ)

Joins: Example

$\mathbf{R} \bowtie \boldsymbol{S}$

```
SELECT R.A, B,C,D
FROM   R, S
WHERE  R.A = S.A
```

Example: Returns all pairs of tuples $\mathrm{r} \in R, s \in S$ *such that* $r.A = s.A$

R

A	B	C
1	0	1
2	3	4
2	5	2
3	1	1

S

A	D
3	7
2	2
2	3

A	B	C	D
2	3	4	2

Joins: Example

$R \bowtie S$

```
SELECT R.A,B,C,D
FROM   R, S
WHERE  R.A = S.A
```

<u>Example:</u> Returns all pairs of tuples $r \in R, s \in S$ *such that* $r.A = s.A$

R

A	B	C
1	0	1
2	3	4
2	5	2
3	1	1

S

A	D
3	7
2	2
2	3

A	B	C	D
2	3	4	2
2	3	4	3

Joins: Example

$\mathbf{R} \bowtie \boldsymbol{S}$

```
SELECT R.A,B,C,D
FROM   R, S
WHERE  R.A = S.A
```

<u>Example:</u> Returns all pairs of tuples $\mathrm{r} \in R, s \in S$ *such that* $r.A = s.A$

R

A	B	C
1	0	1
2	3	4
2	5	2
3	1	1

S

A	D
3	7
2	2
2	3

A	B	C	D
2	3	4	2
2	3	4	3
2	5	2	2

Joins: Example

$R \bowtie S$

```
SELECT R.A,B,C,D
FROM   R, S
WHERE  R.A = S.A
```

<u>Example:</u> Returns all pairs of tuples $r \in R, s \in S$ *such that* $r.A = s.A$

R

A	B	C
1	0	1
2	3	4
2	5	2
3	1	1

S

A	D
3	7
2	2
2	3

A	B	C	D
2	3	4	2
2	3	4	3
2	5	2	2
2	5	2	3

Joins: Example

$\mathbf{R} \bowtie \boldsymbol{S}$

```
SELECT R.A,B,C,D
FROM   R, S
WHERE  R.A = S.A
```

<u>Example:</u> Returns all pairs of tuples $\mathrm{r} \in R, s \in S$ *such that* $r.A = s.A$

R

A	B	C
1	0	1
2	3	4
2	5	2
3	1	1

S

A	D
3	7
2	2
2	3

A	B	C	D
2	3	4	2
2	3	4	3
2	5	2	2
2	5	2	3
3	1	1	7

Semantically: A Subset of the Cross Product

$\mathbf{R} \bowtie \boldsymbol{S}$

```
SELECT R.A,B,C,D
FROM   R, S
WHERE  R.A = S.A
```

Example: Returns all pairs of tuples $r \in R, s \in S$ *such that* $r.A = s.A$

R

A	B	C
1	0	1
2	3	4
2	5	2
3	1	1

$\times$

S

A	D
3	7
2	2
2	3

Cross Product → … → Filter by conditions (r.A = s.A)

2	3	4	2
2	3	4	3
2	5	2	2
2	5	2	3
3	1	1	7

Can we actually implement a join in this way?

Notes

- We write $\mathbf{R} \bowtie \boldsymbol{S}$ to mean *join R and S by returning all tuple pairs where* ***all shared attributes*** *are equal*

- We write $\mathbf{R} \bowtie \boldsymbol{S}$ **on** $\boldsymbol{A}$ to mean *join R and S by returning all tuple pairs where* ***attribute(s)*** *A are equal*

- For simplicity, we'll consider joins on **two tables** and with **equality constraints** ("equijoins")

However, joins *can* merge > 2 tables, and some algorithms do support non-equality constraints!

Nested Loop Joins

Notes

- We are considering "IO aware" algorithms: ***care about disk IO***

- Given a relation R, let:
  - T(R) = # of tuples in R
  - P(R) = # of pages in R

Recall that we read / write entire pages with disk IO

Nested Loop Join (NLJ)

T(R) = # of tuples in R
P(R) = # of pages in R

```
Compute R ⋈ S on A:
  for r in R:
    for s in S:
      if r[A] == s[A]:
        yield (r,s)
```

Nested Loop Join (NLJ)

T(R) = # of tples in R
P(R) = # of pages in R

```
Compute R ⋈ S on A:
  for r in R:
    for s in S:
      if r[A] == s[A]:
        yield (r,s)
```

Cost:

P(R)

1. **Loop over the tuples in R**

Note that our IO cost is based on the number of ***pages*** loaded, not the number of tuples!

Nested Loop Join (NLJ)

T(R) = # of tuples in R
P(R) = # of pages in R

```
Compute R ⋈ S on A:
  for r in R:
    for s in S:
      if r[A] == s[A]:
        yield (r,s)
```

Cost:

P(R) + T(R)*P(S)

1. Loop over the tuples in R
2. **For every tuple in R, loop over all the tuples in S**

Have to read ***all of S*** from disk for ***every tuple in R!***

Nested Loop Join (NLJ)

T(R) = # of tuples in R
P(R) = # of pages in R

Compute R $\bowtie S \; on \; A$:

```
for r in R:
  for s in S:
    if r[A] == s[A]:
      yield (r,s)
```

Cost:

P(R) + T(R)*P(S)

1. Loop over the tuples in R
2. For every tuple in R, loop over all the tuples in S
3. **Check against join conditions**

Note that NLJ can handle things other than equality constraints… just check in the *if* statement!

Nested Loop Join (NLJ)

T(R) = # of tuples in R
P(R) = # of pages in R

Compute $\mathrm{R} \bowtie S \; on \; A$:

```
for r in R:
  for s in S:
    if r[A] == s[A]:
      yield (r,s)
```

Cost:

P(R) + T(R)*P(S) + OUT

1. Loop over the tuples in R
2. For every tuple in R, loop over all the tuples in S
3. Check against join conditions
4. **Write out (to page, then when page full, to disk)**

Nested Loop Join (NLJ)

T(R) = # of tuples in R
P(R) = # of pages in R

Compute $\mathrm{R} \bowtie S\ on\ A$:

```
for r in R:
  for s in S:
    if r[A] == s[A]:
      yield (r,s)
```

Cost:

P(R) + T(R)*P(S) + OUT

What if R ("outer") and S ("inner") switched?

P(***S***) + T(***S***)*P(***R***) + OUT

Outer vs. inner selection makes a huge difference- DBMS needs to know which relation is smaller!

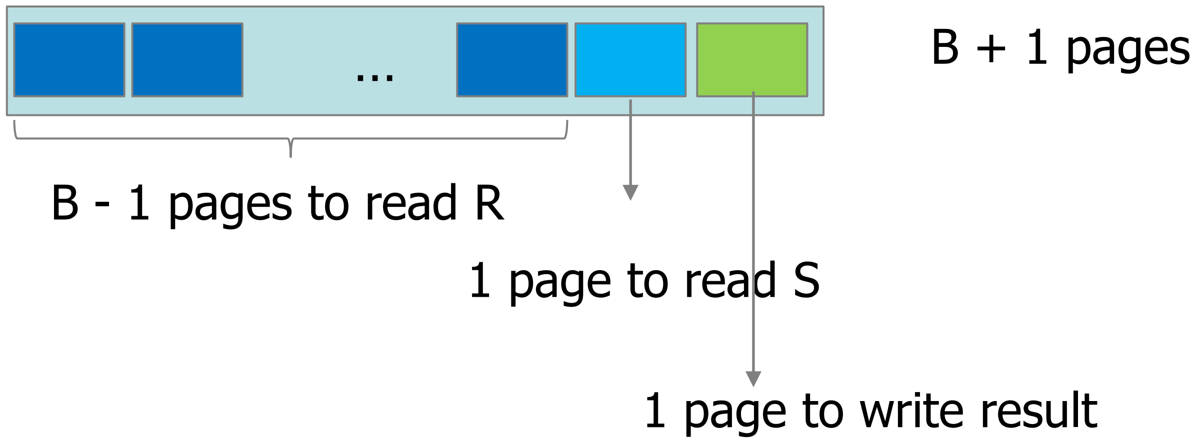
If any one of the relations fits entirely into the memory, it is a must to use that relation as the inner relation. It is because we will read the inner relation only once.

IO-Aware Approach

Block Nested Loop Join (BNLJ)

Block Nested Loop Join (BNLJ)

Buffer



Block Nested Loop Join (BNLJ)

Compute $\mathrm{R} \bowtie S \text{ on } A$:

```
for each B-1 pages pr of R:
  for page ps of S:
    for each tuple r in pr:
      for each tuple s in ps:
        if r[A] == s[A]:
          yield (r,s)
```

Given ***B+1*** pages of memory

Cost:

$\mathrm{P}(R)$

1. **Load in B-1 pages of R at a time (leaving 1 page each free for S & output)**

Note: There could be some speedup here due to the fact that we're reading in multiple pages sequentially however we'll ignore this here!

Block Nested Loop Join (BNLJ)

Compute $\mathrm{R} \bowtie S\ on\ A$:

for each B-1 pages pr of R:

  for page ps of S:

    for each tuple r in pr:

      for each tuple s in ps:

        if r[A] == s[A]:

          yield (r,s)

Given ***B+1*** pages of memory

<u>Cost:</u>

$$\mathrm{P}(R) + \frac{P(R)}{B-1} P(S)$$

1. Load in B-1 pages of R at a time (leaving 1 page each free for S & output)

2. **For each (B-1)-page segment of R, load each page of S**

Note: Faster to iterate over the *smaller* relation first!

Block Nested Loop Join (BNLJ)

Given ***B+1*** pages of memory

```
Compute R ⋈ S on A:
  for each B-1 pages pr of R:
    for page ps of S:
      for each tuple r in pr:
        for each tuple s in ps:
          if r[A] == s[A]:
            yield (r,s)
```

<u>Cost:</u>

$$\mathrm{P}(R) + \frac{P(R)}{B-1} P(S)$$

1. Load in B-1 pages of R at a time (leaving 1 page each free for S & output)

2. For each (B-1)-page segment of R, load each page of S

3. **Check against the join conditions**

BNLJ can also handle non-equality constraints

Block Nested Loop Join (BNLJ)

Compute $\mathrm{R} \bowtie S \; on \; A$:

```
for each B-1 pages pr of R:
  for page ps of S:
    for each tuple r in pr:
      for each tuple s in ps:
        if r[A] == s[A]:
          yield (r,s)
```

OUT could be bigger than P(R)*P(S)… but usually not that bad

Given ***B+1*** pages of memory

Cost:

$$\mathrm{P}(R) + \frac{P(R)}{B-1} P(S) + \mathrm{OUT}$$

1. Load in B-1 pages of R at a time (leaving 1 page each free for S & output)
2. For each (B-1)-page segment of R, load each page of S
3. Check the join conditions
4. **Write out**

BNLJ vs. NLJ: Benefits of IO Aware

- In BNLJ, by loading larger chunks of R, we minimize the number of full *disk reads* of S
  - We only read all of S from disk for ***every (B-1)-page segment of R***!
  - Still the full cross-product, but more done only *in memory*

NLJ

P(R) + T(R)*P(S) + OUT

BNLJ

$$\mathrm{P}(R) + \frac{P(R)}{B-1} P(S) + \mathrm{OUT}$$

BNLJ is faster by roughly $\frac{(B-1)T(R)}{P(R)}$!

BNLJ vs. NLJ: Benefits of IO Aware

NLJ

P(R) + T(R)*P(S) + OUT

BNLJ

$P(R) + \frac{P(R)}{B-1} P(S)$ + OUT

Ignore OUT
Assume IO cost 0.01 sec

- Example:
  - R: 500 pages
  - S: 1000 pages
  - 100 tuples / page
  - We have 12 pages of memory (B = 11)

- NLJ: Cost = 500 + **50,000*1000** = **50 Million IOs ~= <u>140 hours</u>**

- BNLJ: Cost = 500 + $\frac{500*1000}{10}$ = **50 *Thousand* IOs ~= <u>0.14 hours</u>**

A very real difference from a small change in the algorithm!

Smarter than Cross-Products

From Quadratic to Nearly Linear

- All joins that compute the ***full cross-product*** have some **quadratic** term
  - For example we saw:

NLJ $P(R) + T(R)P(S) + OUT$

BNLJ $P(R) + \frac{P(R)}{B-1} P(S) + OUT$

- Now we'll see (nearly) linear joins:
  - ~ O(P(R) + P(S) + ***OUT***), where again ***OUT*** could be quadratic but is usually better

We get this gain by ***taking advantage of structure*** - moving to equality constraints ("equijoin") only!

Index Nested Loop Join (INLJ)

Compute $\mathrm{R} \bowtie S\ on\ A$:

  Given index idx on S.A:

    for r in R:

      s in idx(r[A]):

        yield r,s

<u>Cost:</u>

P(R) + T(R)****L**** + OUT

where ***L*** is the cost of finding matching S tuples. For each R tuple, cost of probing S index is about 2-4 for B+ tree. Cost of then finding S tuples depends on index clustering

→ We can use an **index** (e.g. B+ Tree) to ***avoid doing the full cross-product!***

Summary

- We covered joins--an ***IO aware*** algorithm makes a big difference.

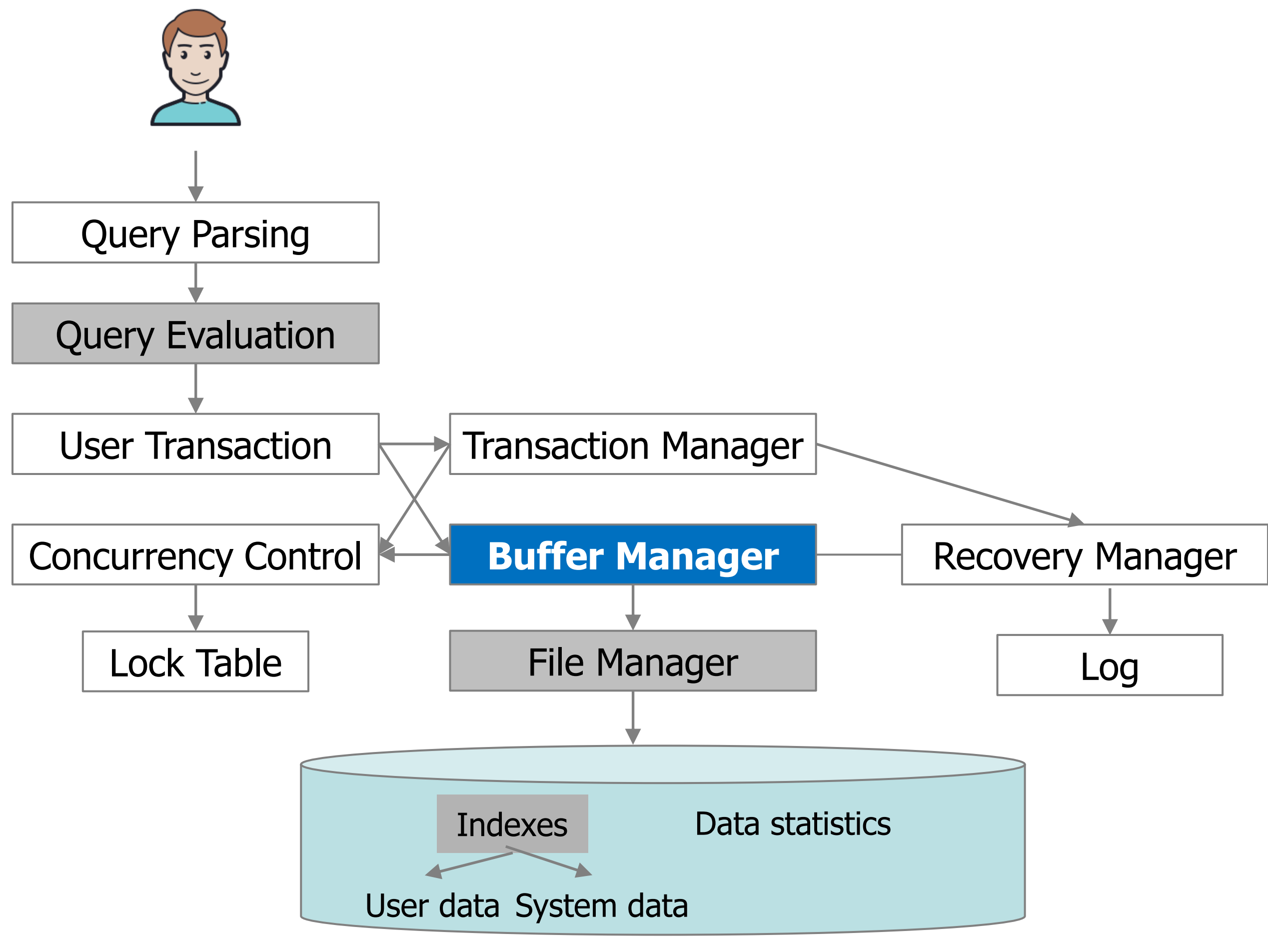
- Fundamental strategies: blocking and reorder loops (asymmetric costs in IO)

Can we do better/differently?
What happens if we have very large tables (that do not fit in memory?)

Returning to

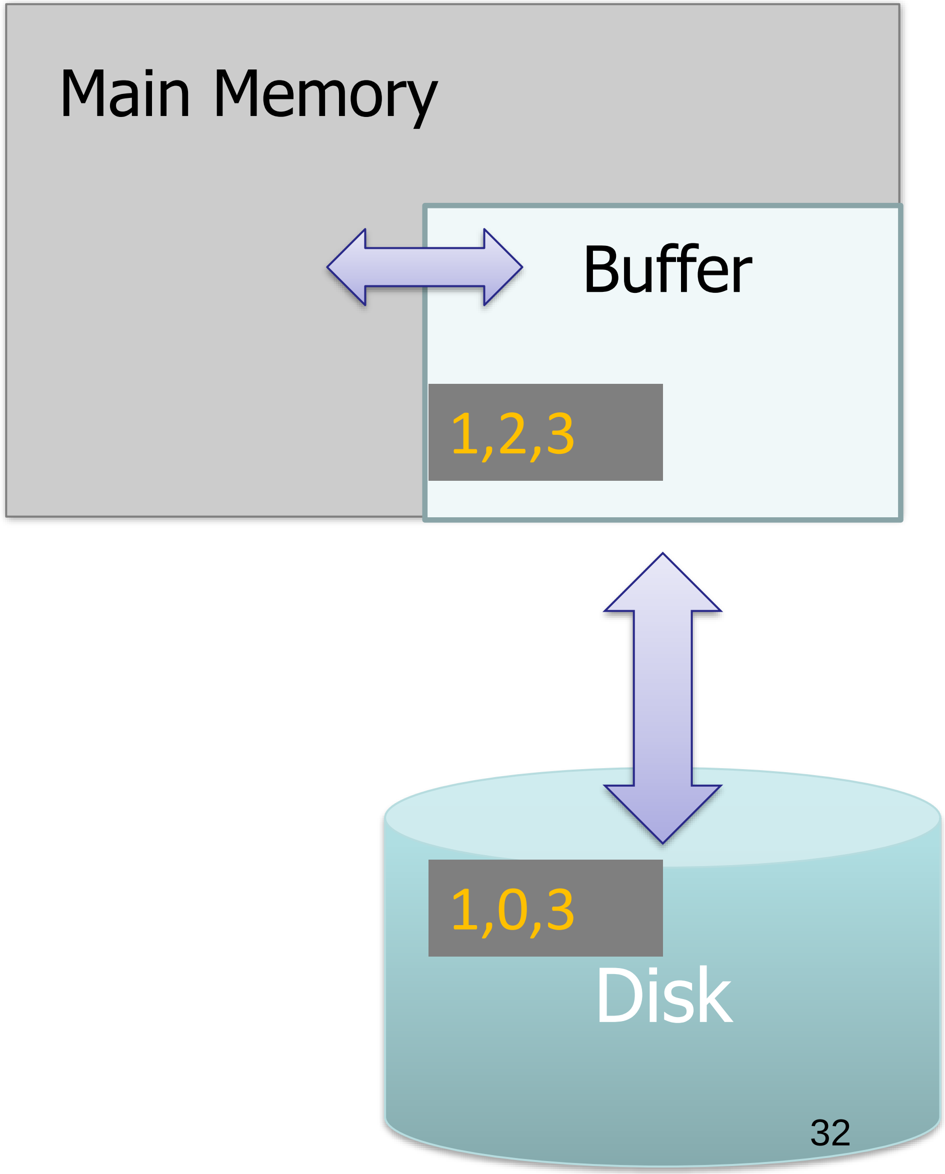
Buffer Basics:

Efficiently merge two sorted files when both are much larger than our main memory buffer



The (Simplified) Buffer

- **<u>Read(page):</u>** Read page from disk -> buffer *if not already in buffer*
- **<u>Flush(page):</u>** Evict page from buffer & write to disk
- **<u>Release(page):</u>** Evict page from buffer *without* writing to disk



Challenge: Merging Big Files with Small Memory

How do we *efficiently* merge two sorted files when both are much larger than our main memory buffer?

External Merge

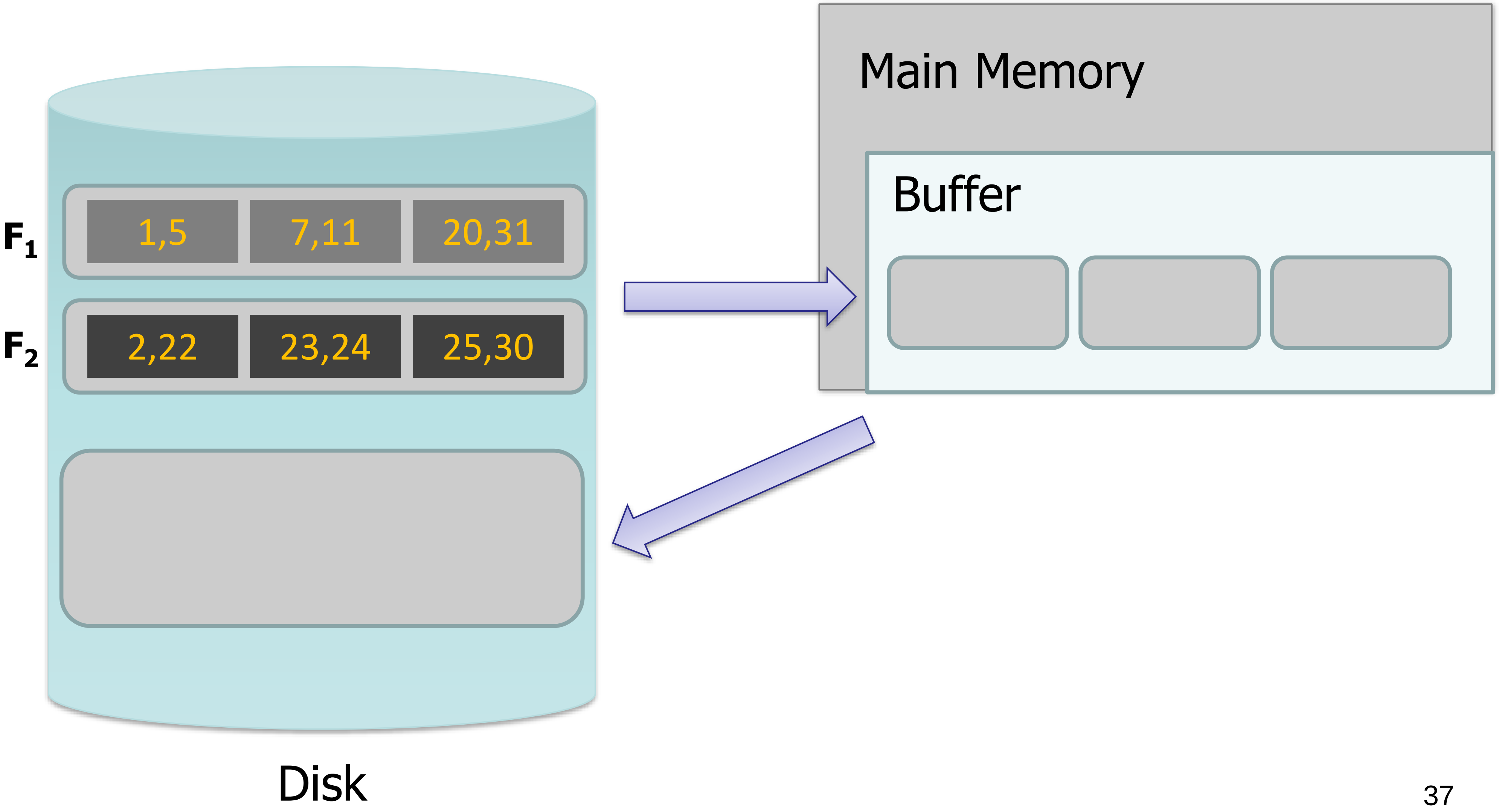
External Merge Algorithm

- **Input**: 2 sorted lists of length M and N
- **Output:** 1 sorted list of length M + N
- **Required:** At least 3 Buffer Pages
- **IOs**: 2(M+N)

External Merge Algorithm

Input:
Two sorted files

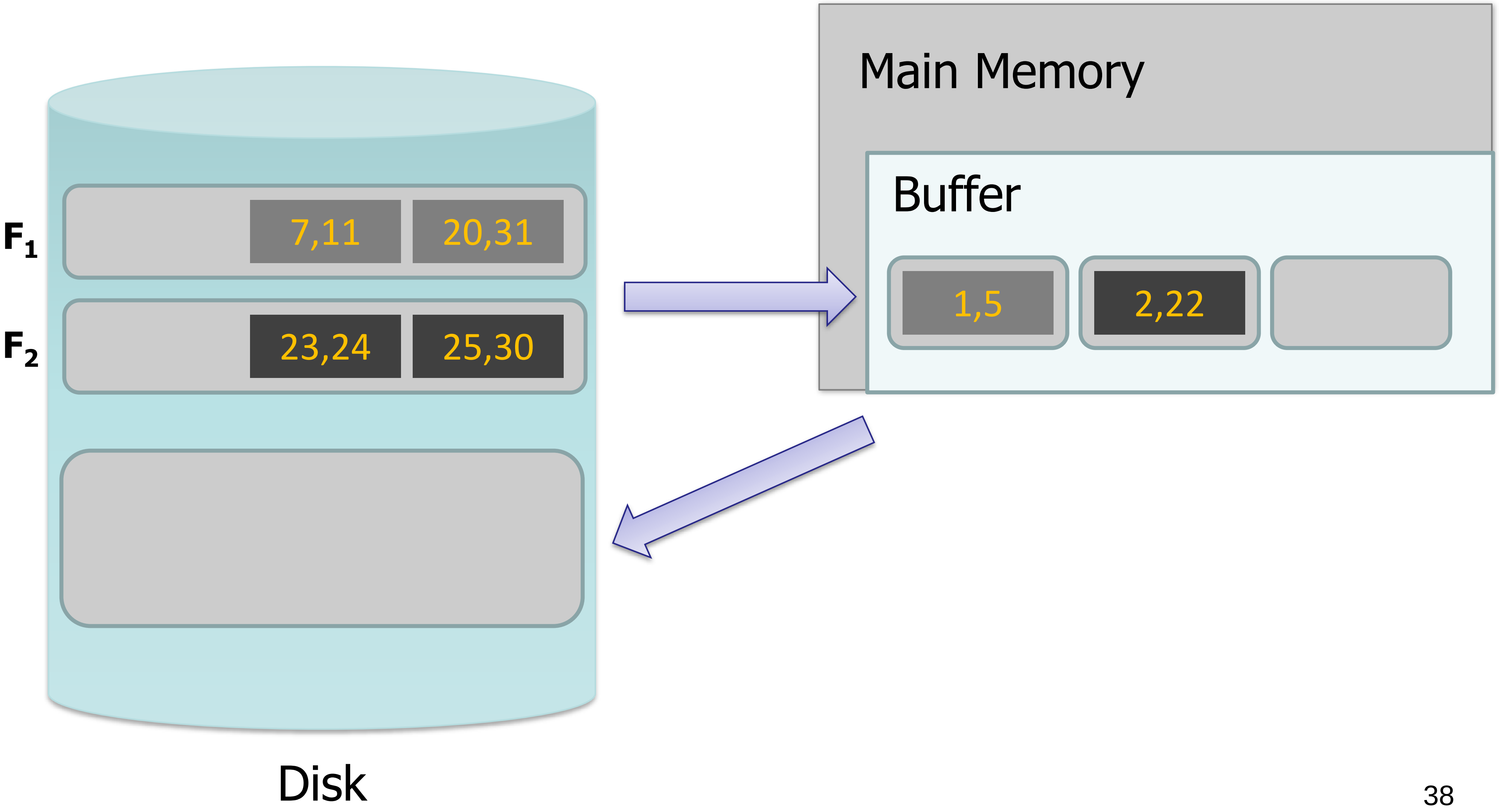
Output:
One *merged* sorted file



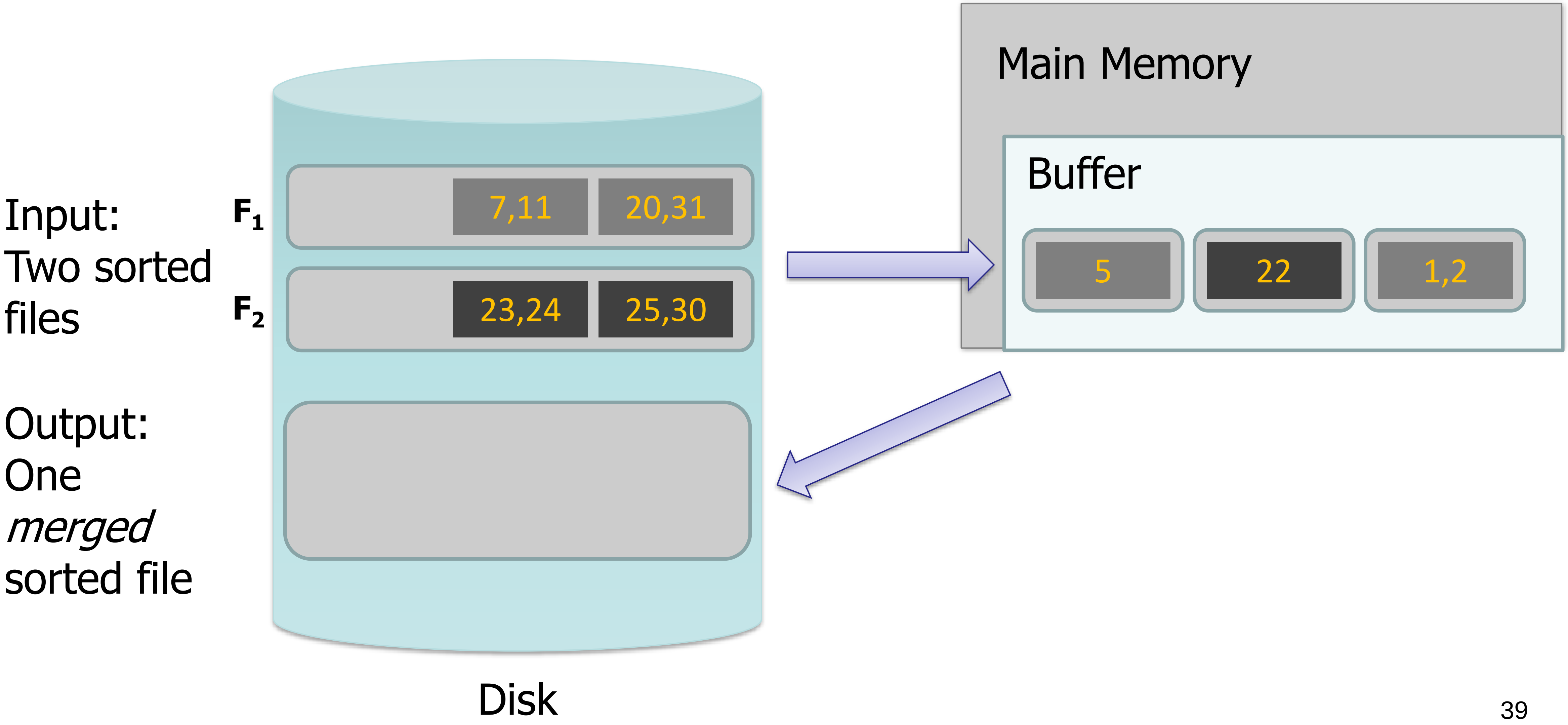
External Merge Algorithm

Input:
Two sorted
files

Output:
One
merged
sorted file



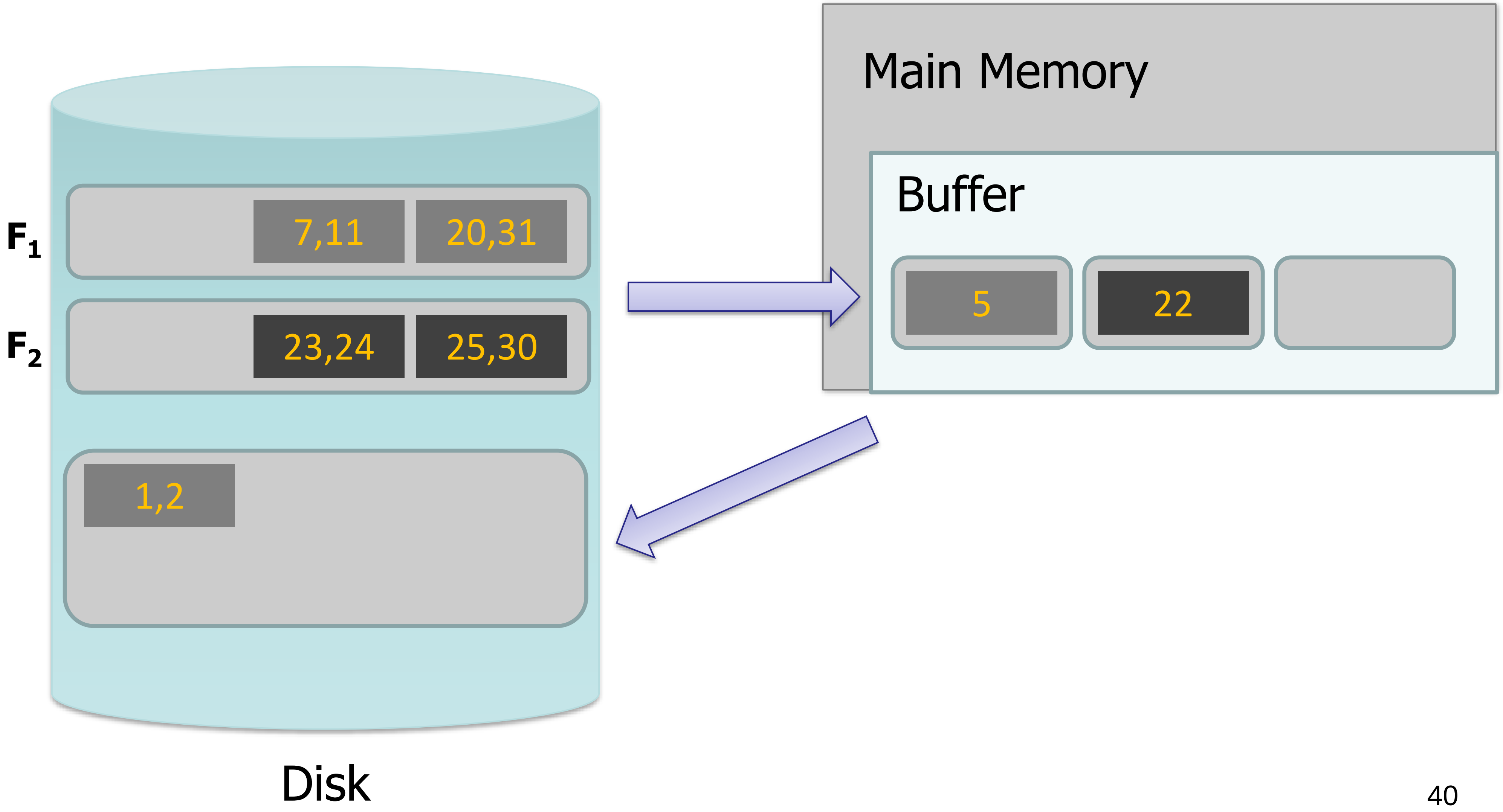
External Merge Algorithm



External Merge Algorithm

Input:
Two sorted
files

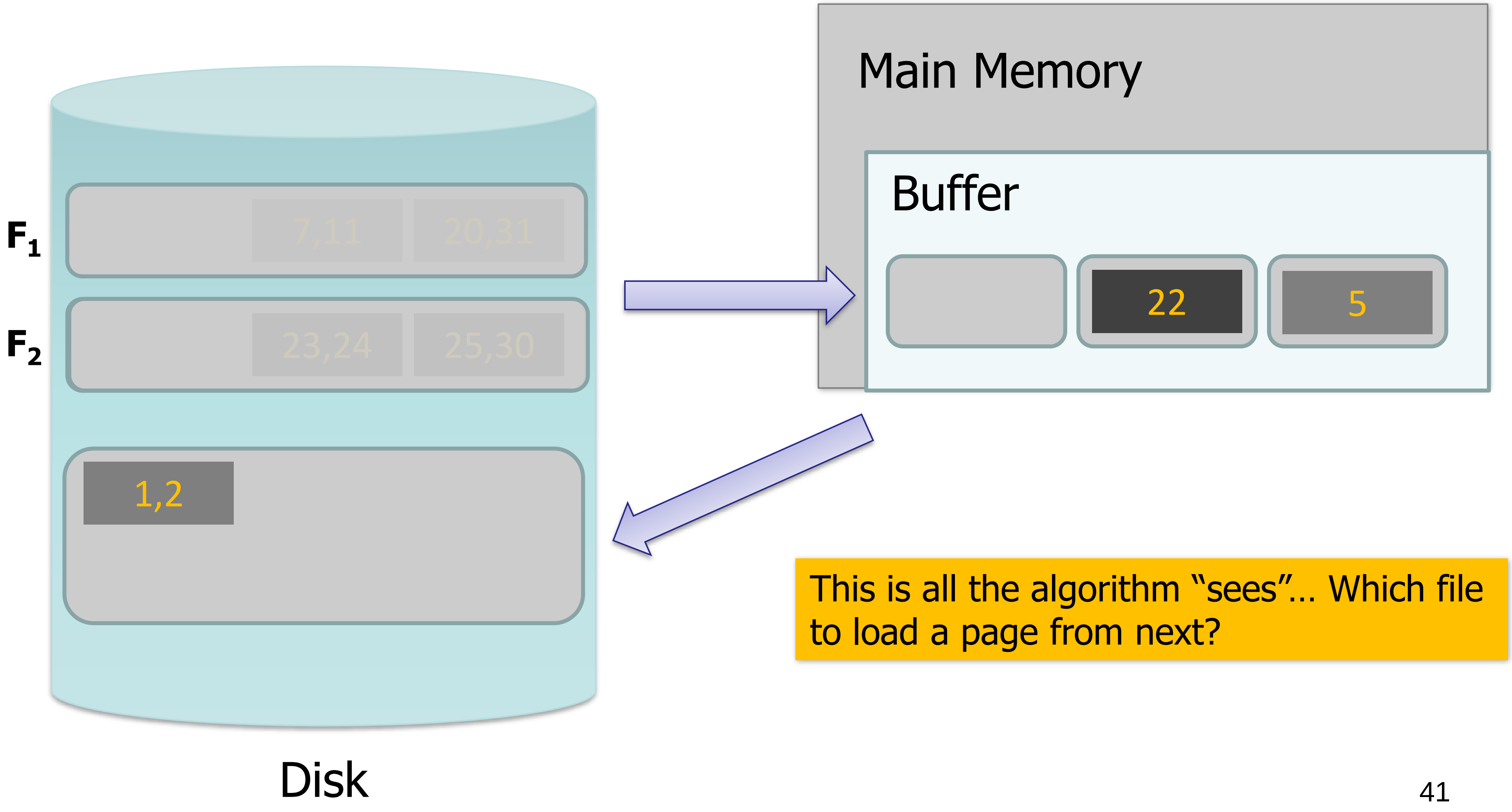
Output:
One
merged
sorted file



External Merge Algorithm

Input:
Two sorted
files

Output:
One
merged
sorted file

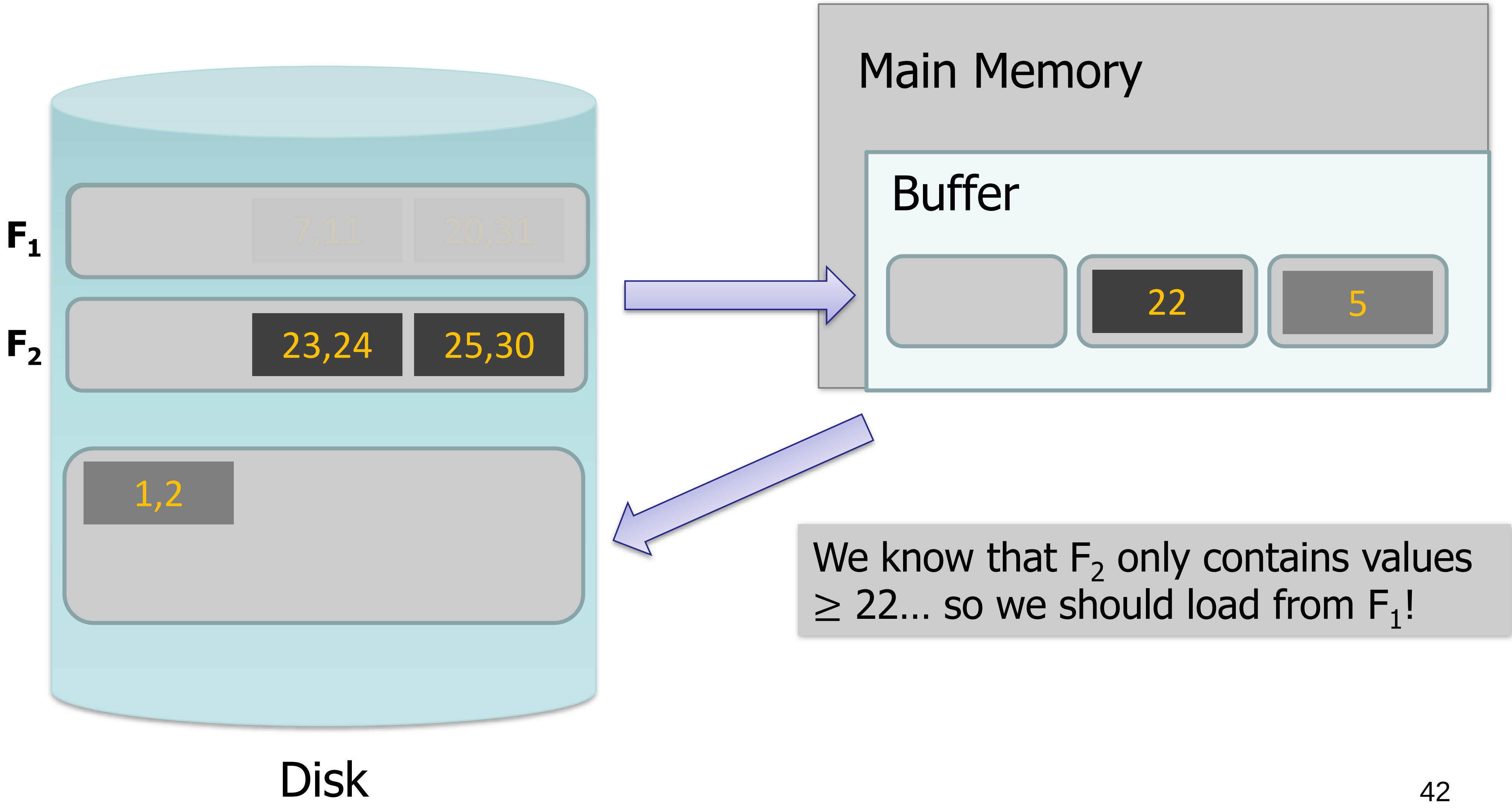


This is all the algorithm "sees"… Which file to load a page from next?

External Merge Algorithm

Input:
Two sorted files

Output:
One *merged* sorted file

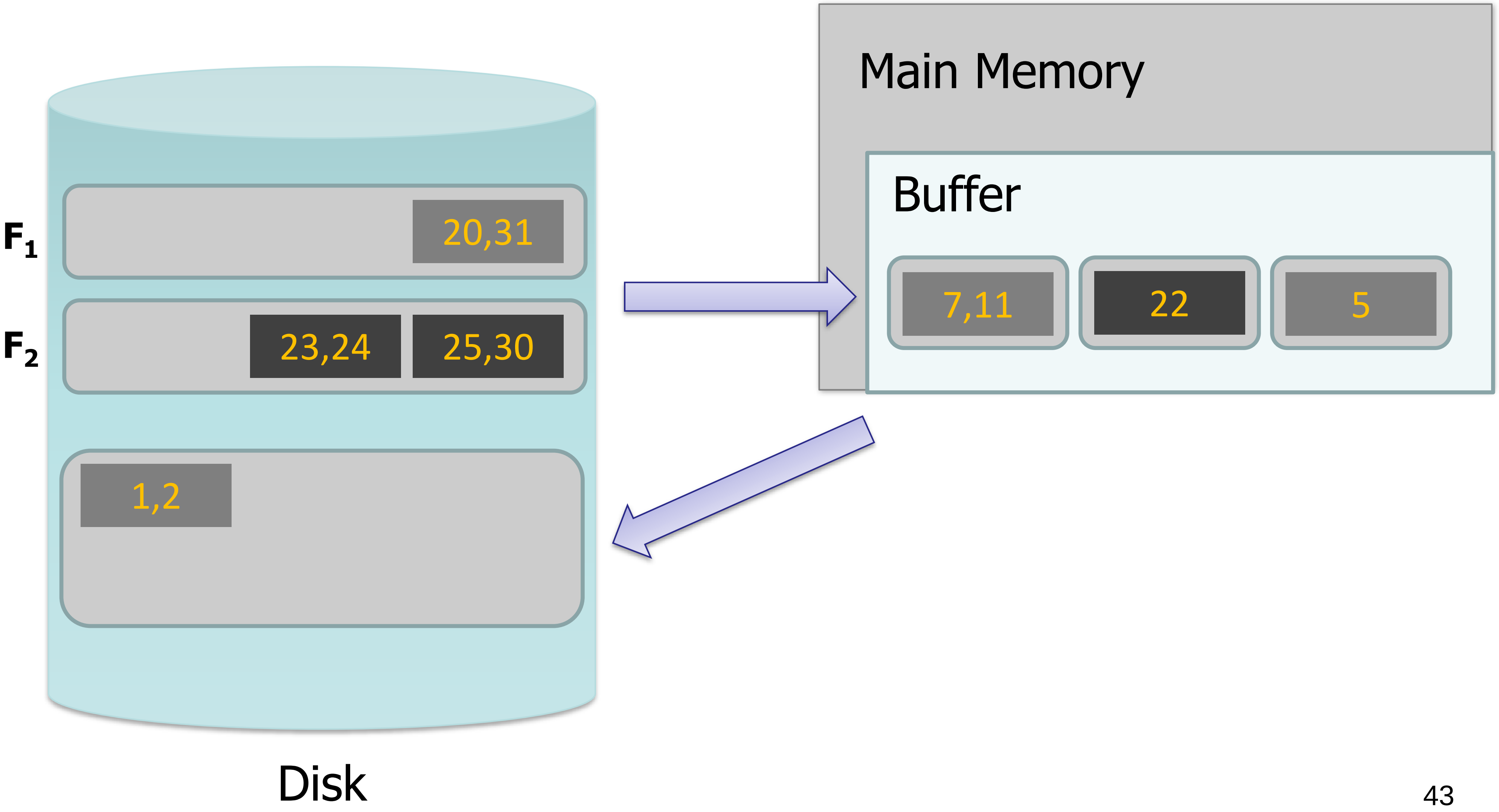


We know that F_2 only contains values $\geq$ 22… so we should load from F_1!

External Merge Algorithm

Input:
Two sorted files

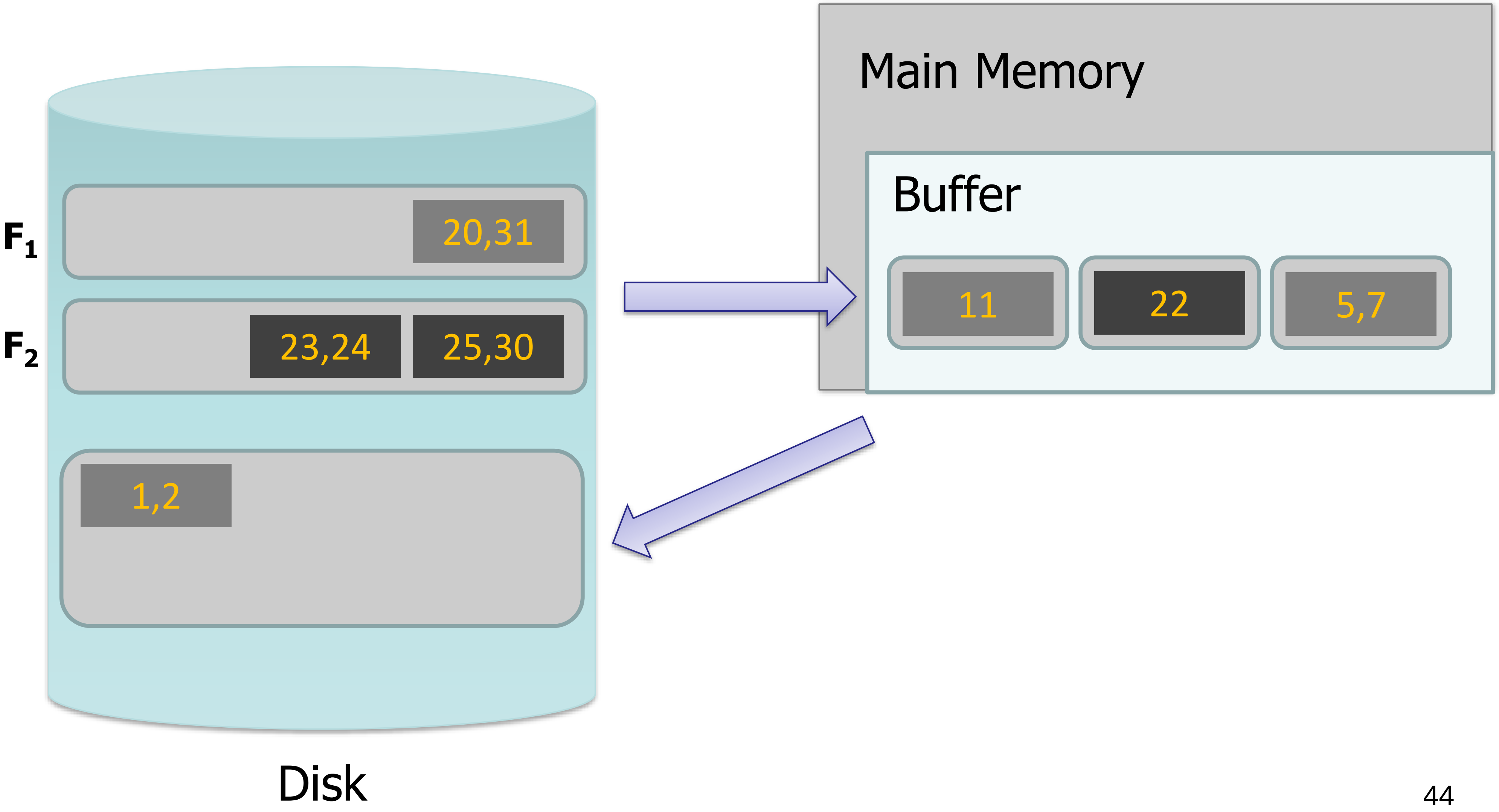
Output:
One *merged* sorted file



External Merge Algorithm

Input:
Two sorted
files

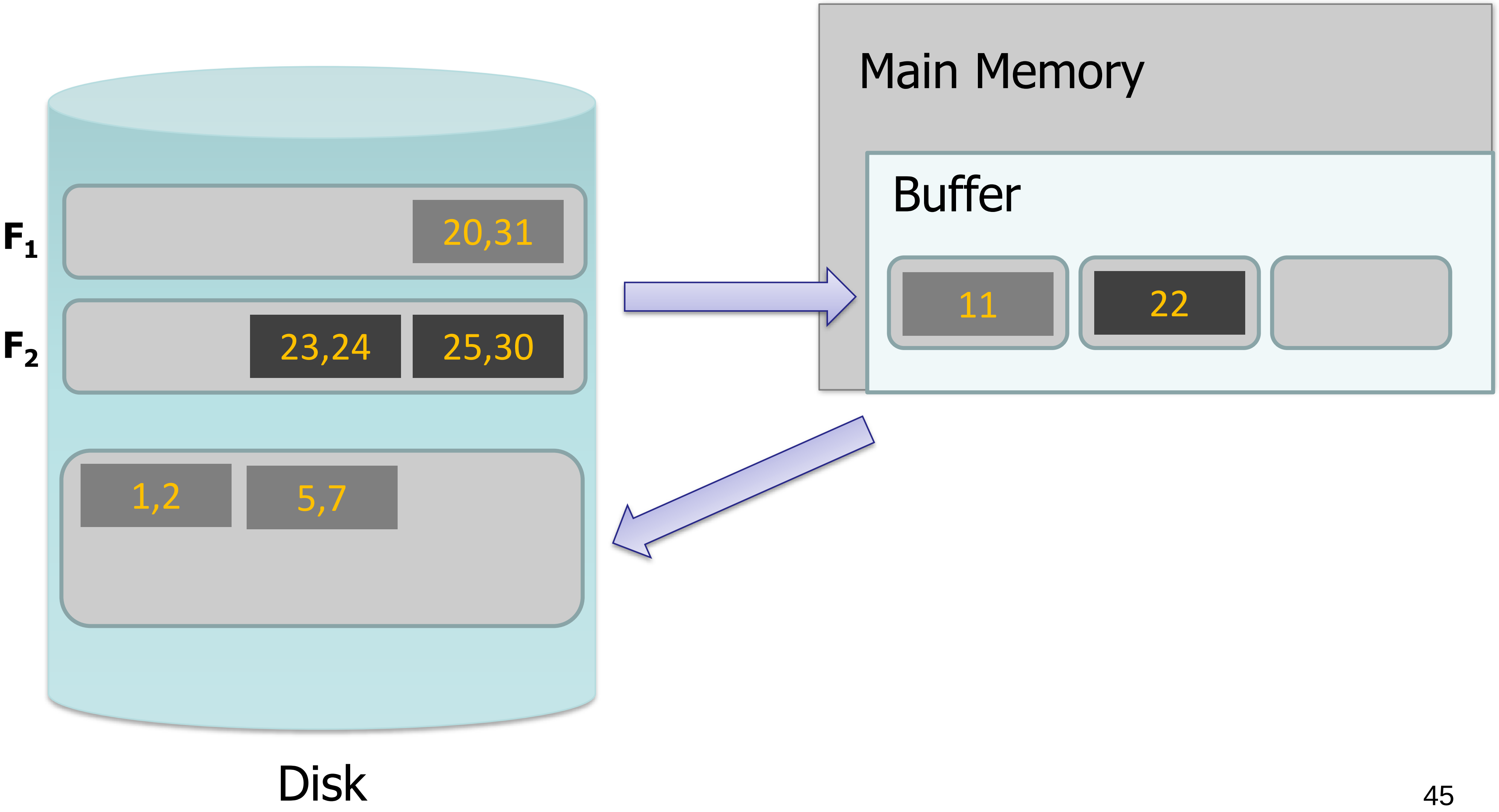
Output:
One
merged
sorted file



External Merge Algorithm

Input:
Two sorted
files

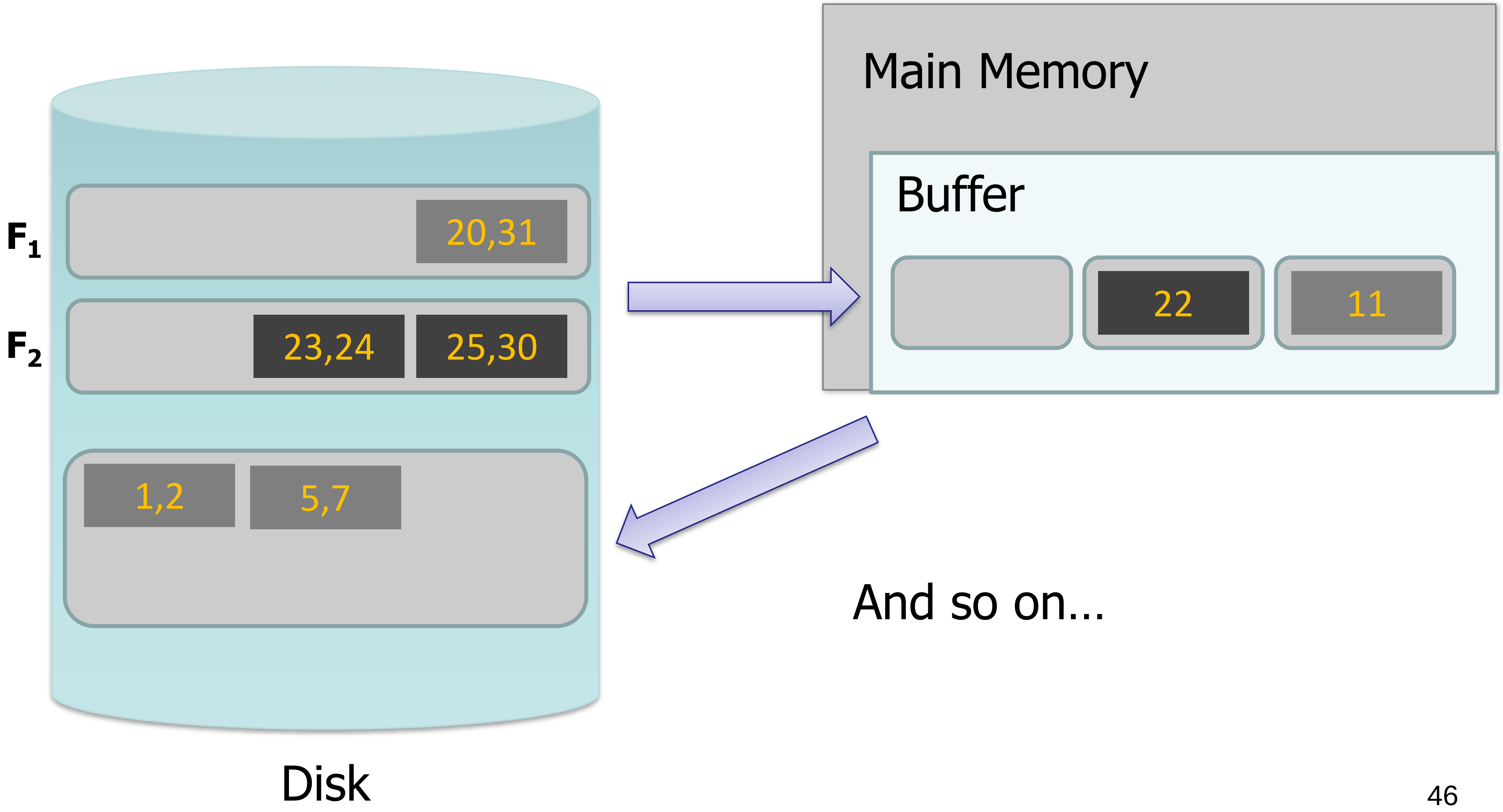
Output:
One
merged
sorted file



External Merge Algorithm

Input:
Two sorted files

Output:
One *merged* sorted file



And so on…

External Merge Algorithm

We can merge lists of **arbitrary length** with *only* 3 buffer pages.

If lists of size M and N, then

Cost: 2(M+N) IOs

Each page is read once, written once

What if our files are not sorted?

External Merge Sort

What you will learn about in this section

1. External merge sort
2. External merge sort on larger files
3. Optimizations for sorting

External Merge Sort

Why are Sort Algorithms Important?

- Data requested from DB in sorted order is **extremely common**
  - e.g., find students in increasing GPA order

Why not just use quicksort in main memory??

Average performance: O(n log n)

What about if we need to sort 1TB of data with 1GB of RAM…

A classic problem in computer science!

More reasons to sort…

- Sorting useful for eliminating *duplicate copies* in a collection of records (Why?)

- *Sort-merge* join algorithm involves sorting

Next lecture

So how do we sort big files?

1. SORT phase: Split into chunks small enough to **sort in memory** ***("runs")***

2. MERGE phase **:** Merge pairs (or groups) of runs ***using the external merge algorithm***

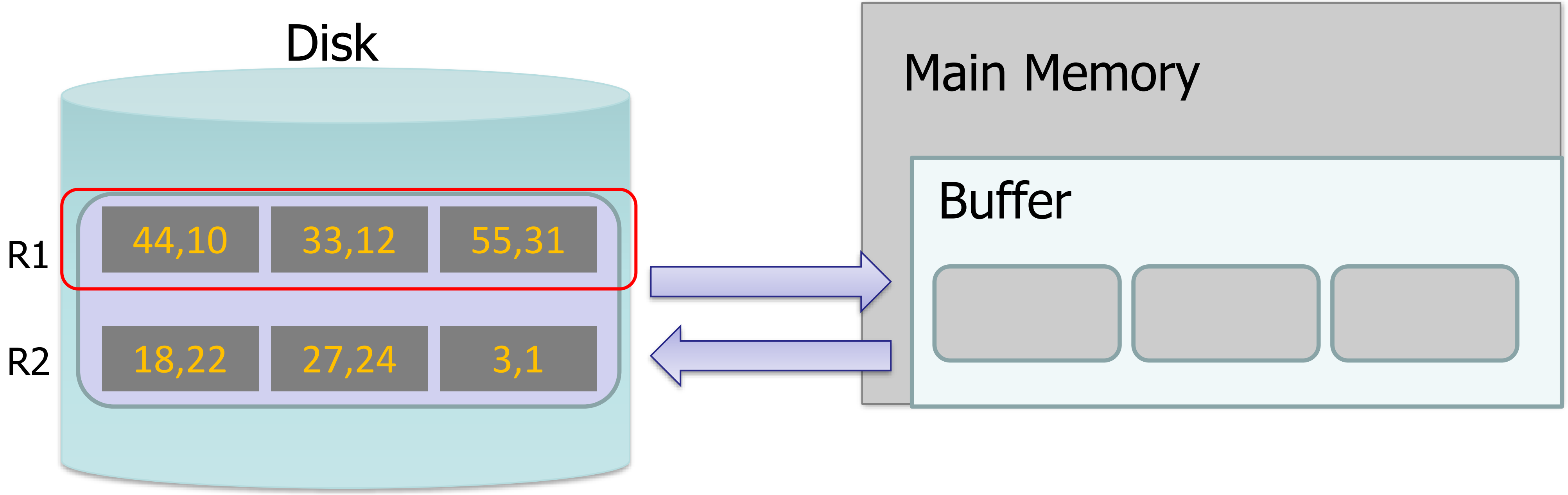
3. **Keep merging** the resulting runs ***(each time = a "pass")*** until left with one sorted file!

External Merge Sort Algorithm

Example:

- **3 Buffer pages**
- **6-page file**

Orange file = unsorted

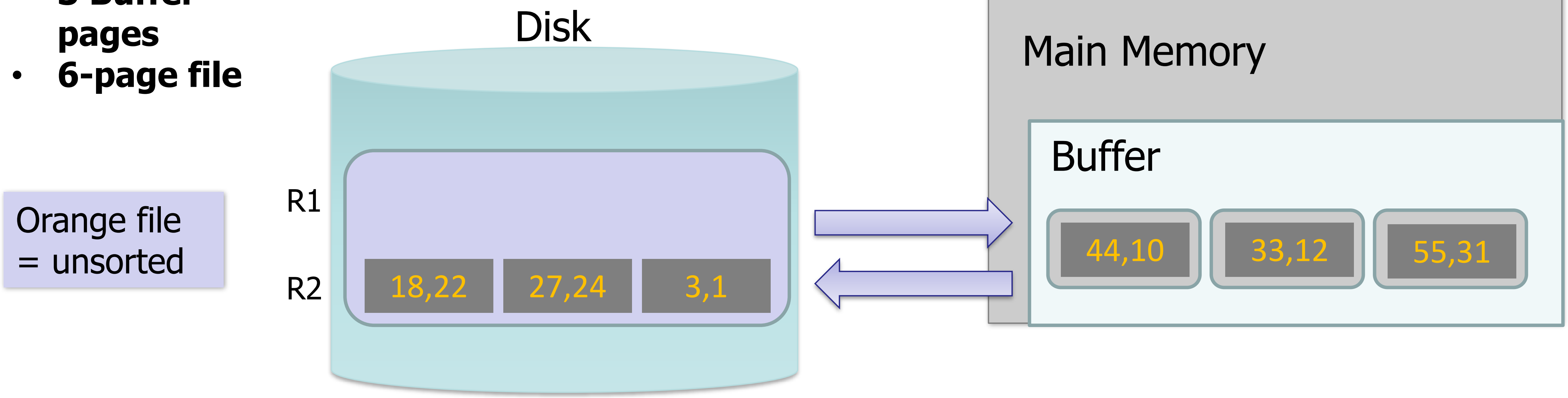


1. SORT phase:

   Split into chunks small enough to **sort in memory**

External Merge Sort Algorithm

Example:
- **3 Buffer pages**
- **6-page file**

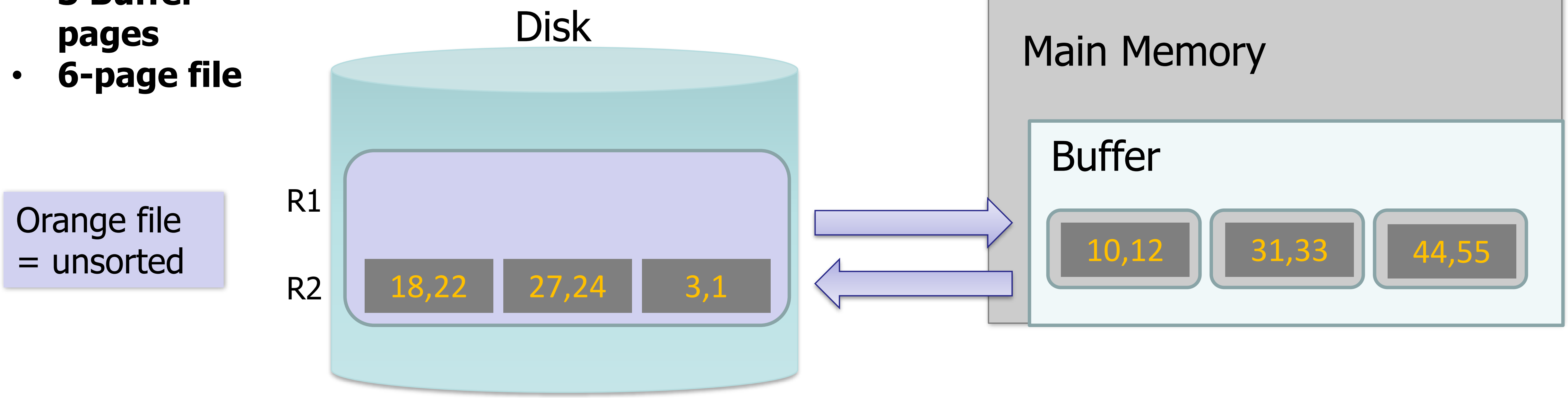


1. SORT phase:

   Split into chunks small enough to **sort in memory**

External Merge Sort Algorithm

Example:
- **3 Buffer pages**
- **6-page file**



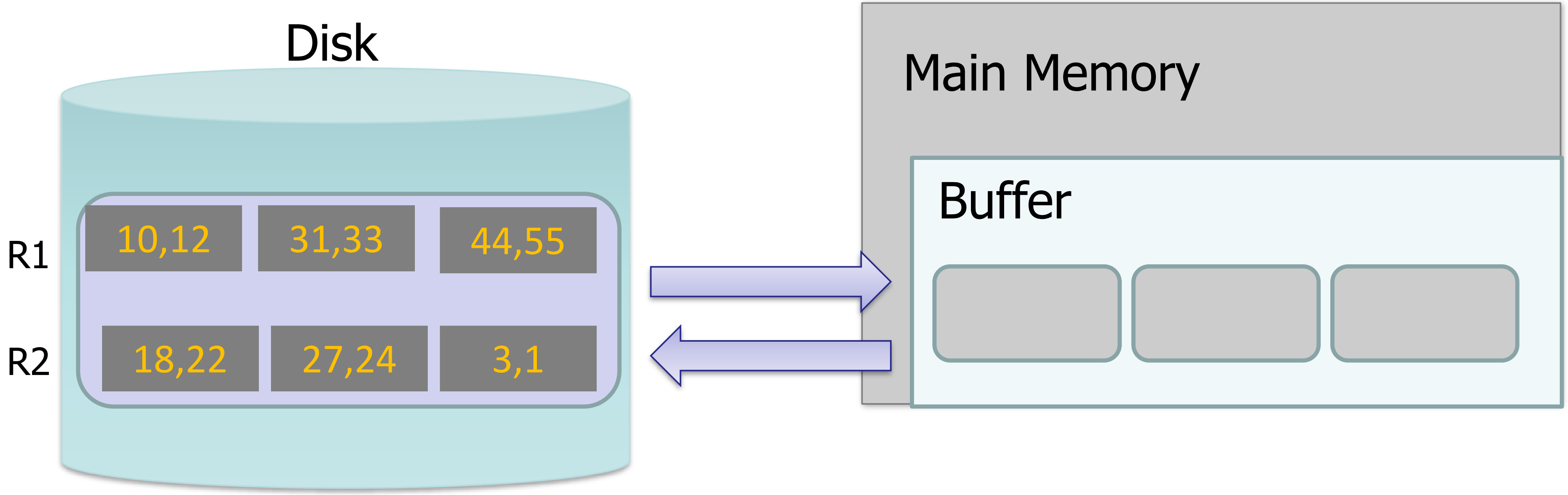
1. SORT phase:

   Sort the current run in memory

External Merge Sort Algorithm

Example:
- **3 Buffer pages**
- **6-page file**

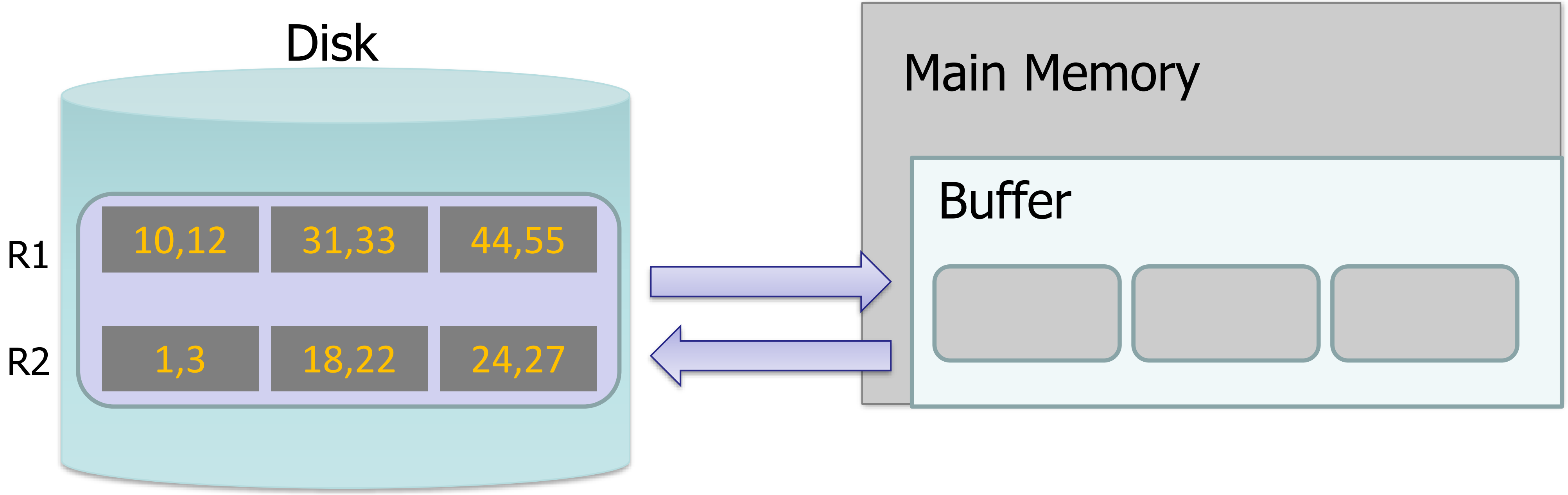
Orange file = unsorted



1. SORT phase:
   Similarly for R2

External Merge Sort Algorithm

Example:
- **3 Buffer pages**
- **6-page file**



2. MERGE phase:

   Now just run the **external merge** algorithm & we're done!

Calculating IO Cost

External Merge Sort

For 3 buffer pages, 6 page file:

1. Split into **two 3-page runs** and **sort in memory**

   Cost = 1 R + 1 W for each run = 2*(3 + 3) = 12 IO operations

2. ***Merge*** each pair of sorted chunks ***using the external merge algorithm***

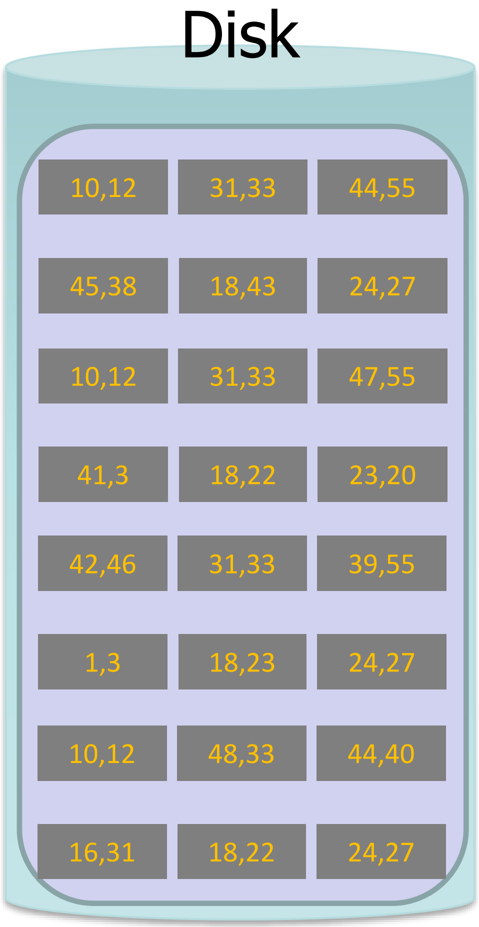
   Cost = 2*(3 + 3) = 12 IO operations

3. **Total cost = 24 IO**

External Merge : 2(M+N) IOs

Running External Merge Sort on Larger Files

Assume we still only have *3* buffer pages *(Buffer not pictured)*



Running External Merge Sort on Larger Files

Assume we still only have *3* buffer pages *(Buffer not pictured)*

1. **Split** into files small enough to sort in buffer

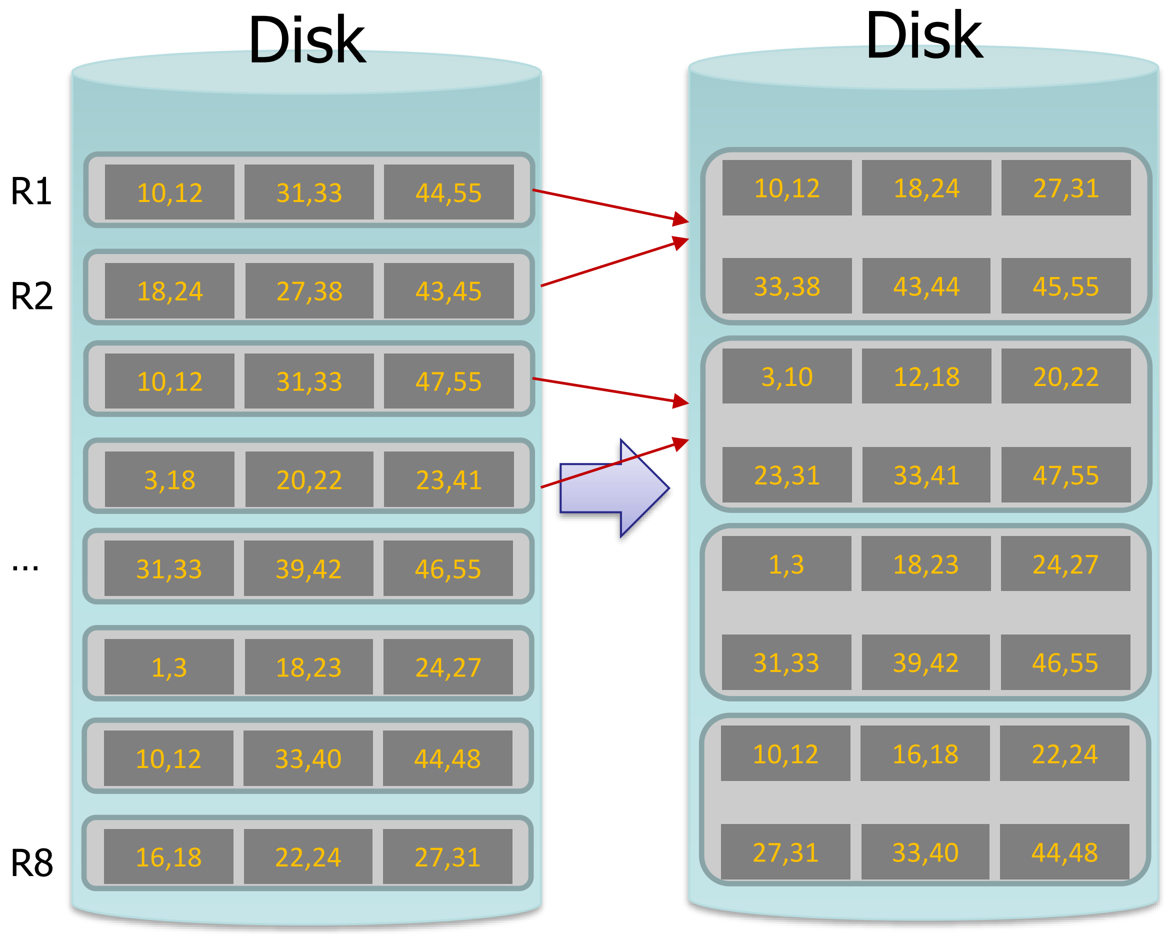
Running External Merge Sort on Larger Files

Assume we still only have *3* buffer pages *(Buffer not pictured)*

1. **Split** into files small enough to sort in buffer… and **sort**

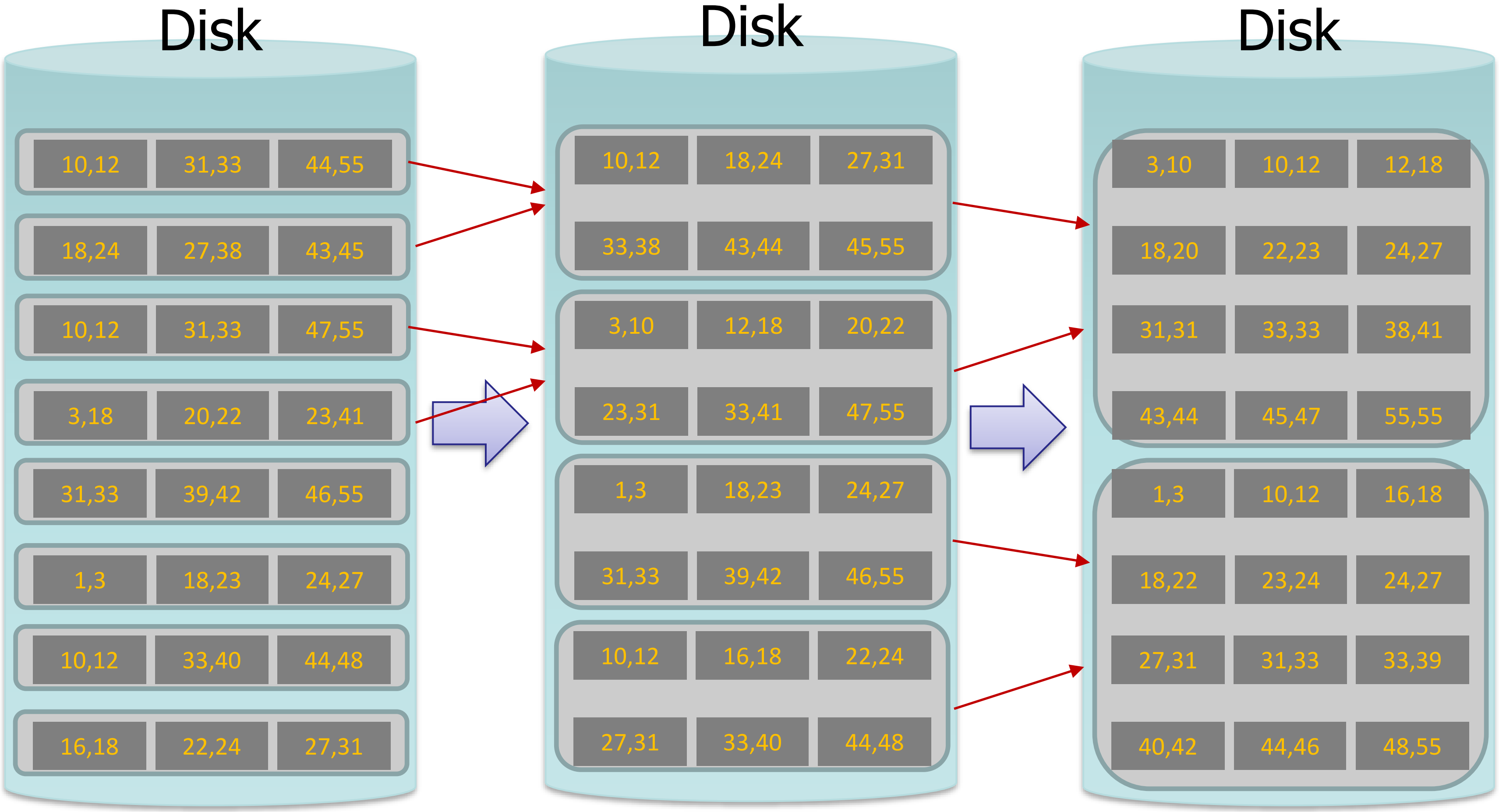
Call each of these sorted files a ***run***

Running External Merge Sort on Larger Files



2. Now **merge pairs of (sorted) files…** the resulting files will be sorted!

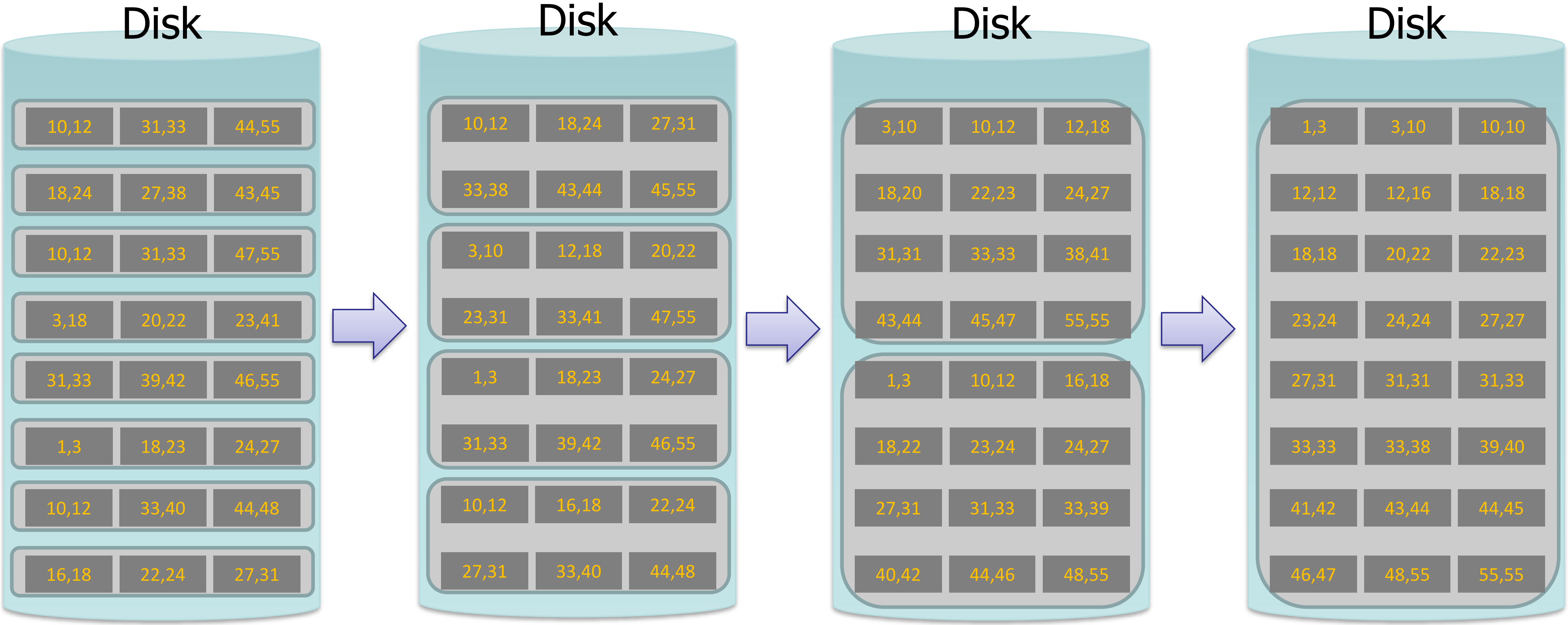
Running External Merge Sort on Larger Files



3. Repeat until all sorted!

Call each of these steps a ***pass***

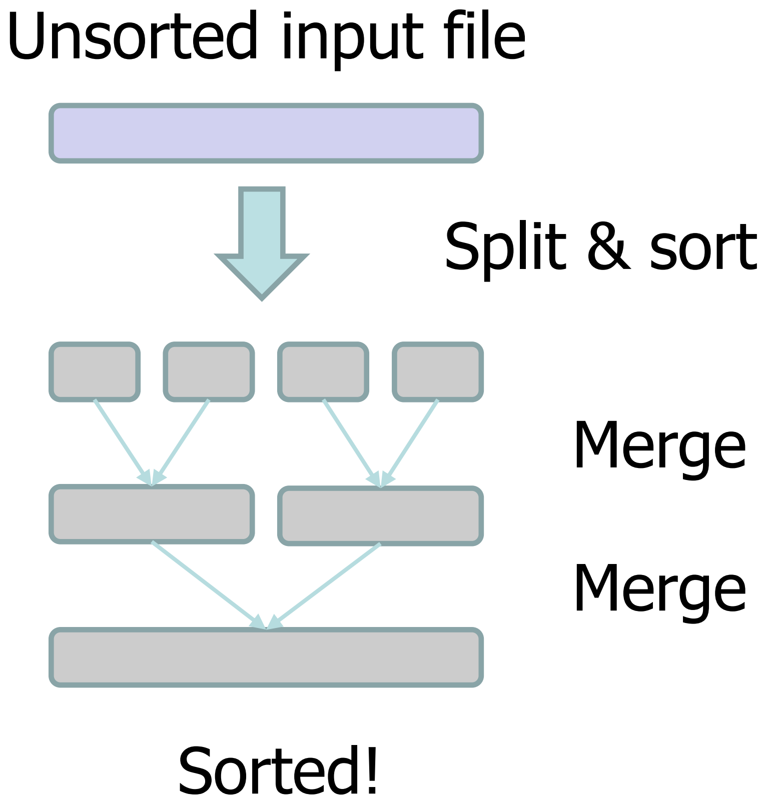
Running External Merge Sort on Larger Files



Simplified 3-page Buffer Version

Assume for simplicity that we split an N-page file into N single-page ***runs*** and sort these; then:

- First pass: Merge **N/2 *pairs* of runs** each of length **1 page**
- Second pass: Merge **N/4 *pairs* of runs** each of length **2 pages**
- In general, for **N** pages, we do $\lceil \log_2 N \rceil$ passes
  - +1 for the initial split & sort
- Each pass involves reading in and writing out all the pages = ***2N IO***



→ **2N*(**$\lceil \log_2 N \rceil$**+1)** total IO cost!

awesome

Using B+1 buffer pages to reduce # of passes

Suppose we have B+1 buffer pages now; we can:

1. **Increase length of initial runs**. Sort B+1 at a time!

At the beginning, we can split the N pages into runs of length B+1 and sort these in memory

<u>IO Cost:</u>

$$2N(\lceil \log_2 N \rceil + 1)$$

$$2N\left(\left\lceil \log_2 \frac{\boldsymbol{N}}{\boldsymbol{B+1}} \right\rceil + 1\right)$$

Starting with runs of length 1

Starting with runs of length ***B+1***

Using B+1 buffer pages to reduce # of passes

Suppose we have B+1 buffer pages now; we can:

2. Perform a B-way merge.

On each pass, we can merge groups of ***B*** runs at a time (vs. merging pairs of runs)!

<u>IO Cost:</u>

$$2N(\lceil \log_2 N \rceil + 1)$$ → $$2N\left(\left\lceil \log_2 \frac{\boldsymbol{N}}{\boldsymbol{B+1}} \right\rceil + 1\right)$$ → $$2N\left(\left\lceil \log_B \frac{\boldsymbol{N}}{\boldsymbol{B+1}} \right\rceil + 1\right)$$

Starting with runs of length 1

Starting with runs of length ***B+1***

Performing ***B***-way merges

Repacking

Repacking for even longer initial runs

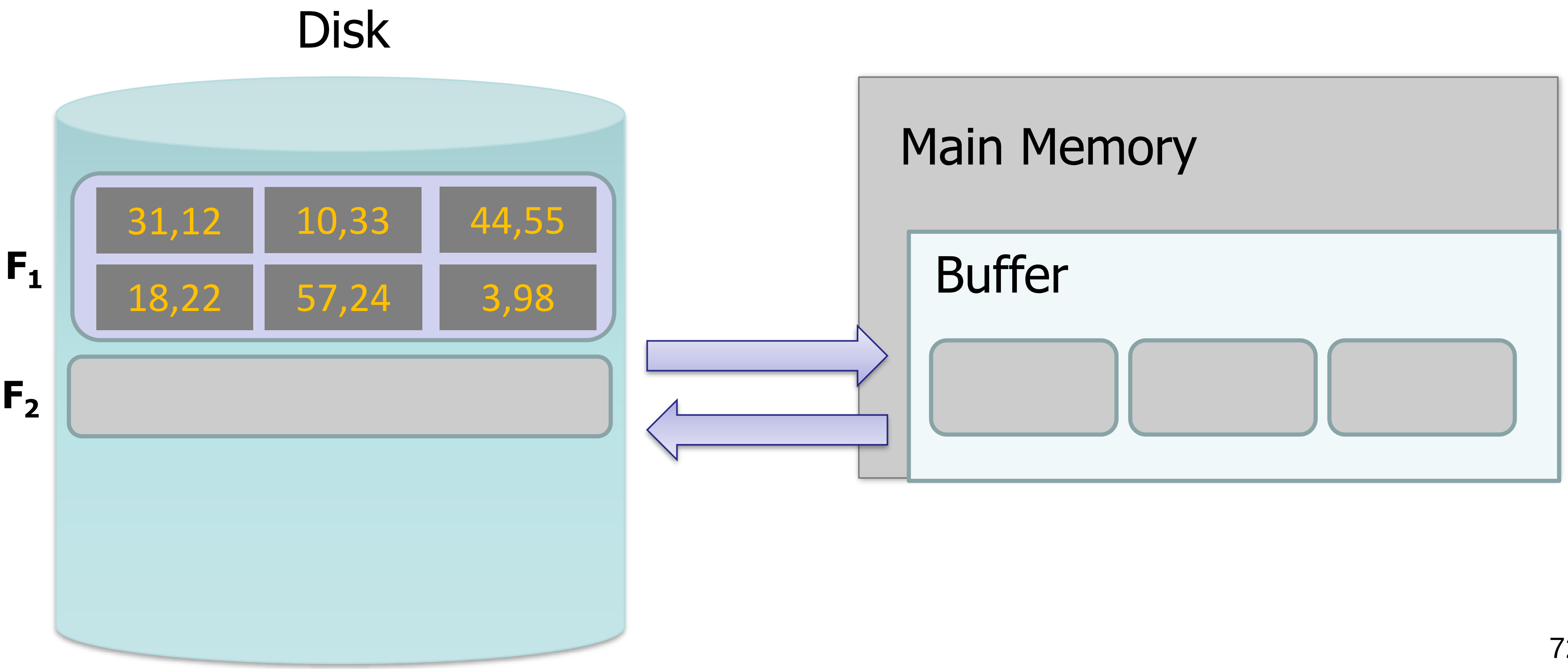
- With B+1 buffer pages, we can now start with ***B+1-length initial runs*** (and use ***B-way merges***) to get $2N(\left\lceil \log_B \frac{\boldsymbol{N}}{\boldsymbol{B+1}} \right\rceil + 1)$ IO cost…

- Can we reduce this cost more by getting even longer initial runs?

- Use **<u>repacking</u>**- produce longer initial runs by "merging" in buffer as we sort at initial stage

Repacking Example: 3 page buffer

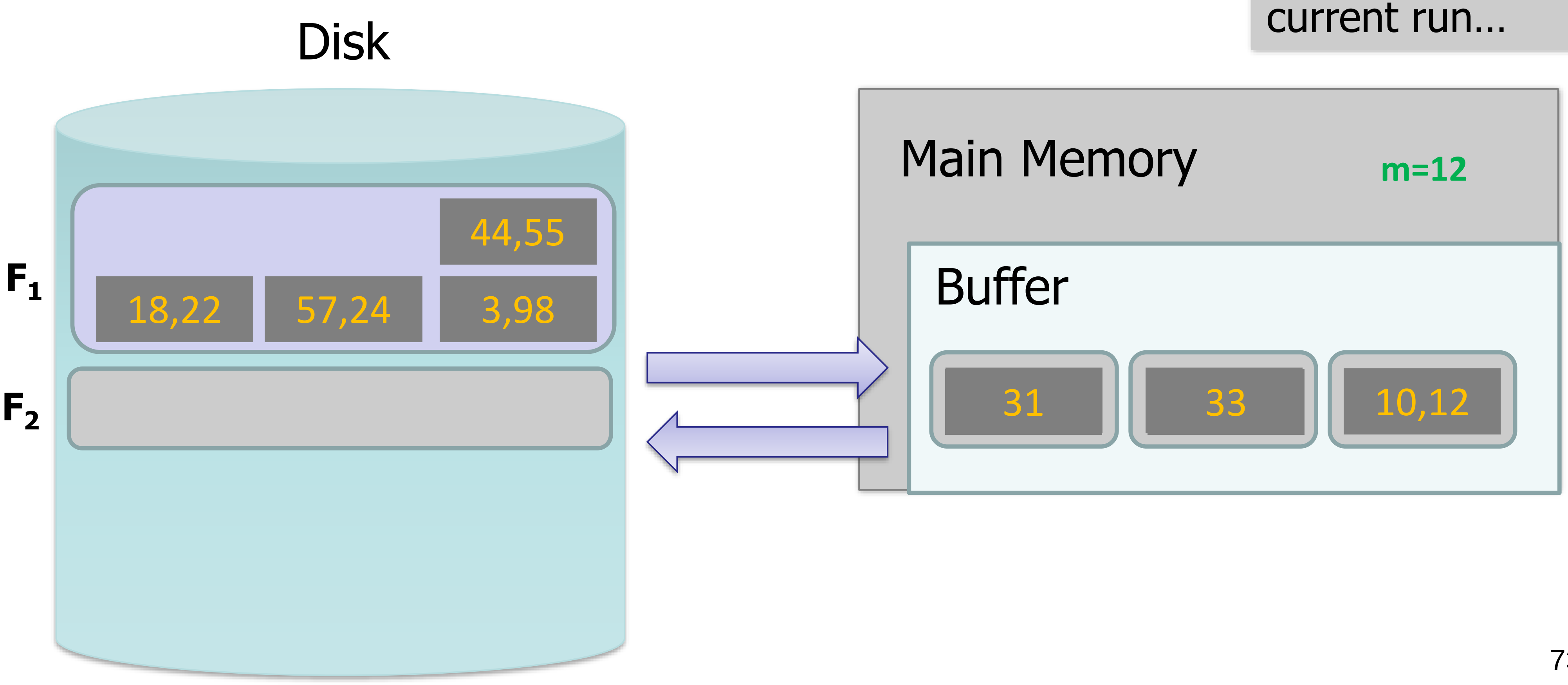
- Start with unsorted single input file, and load 2 pages



Repacking Example: 3 page buffer

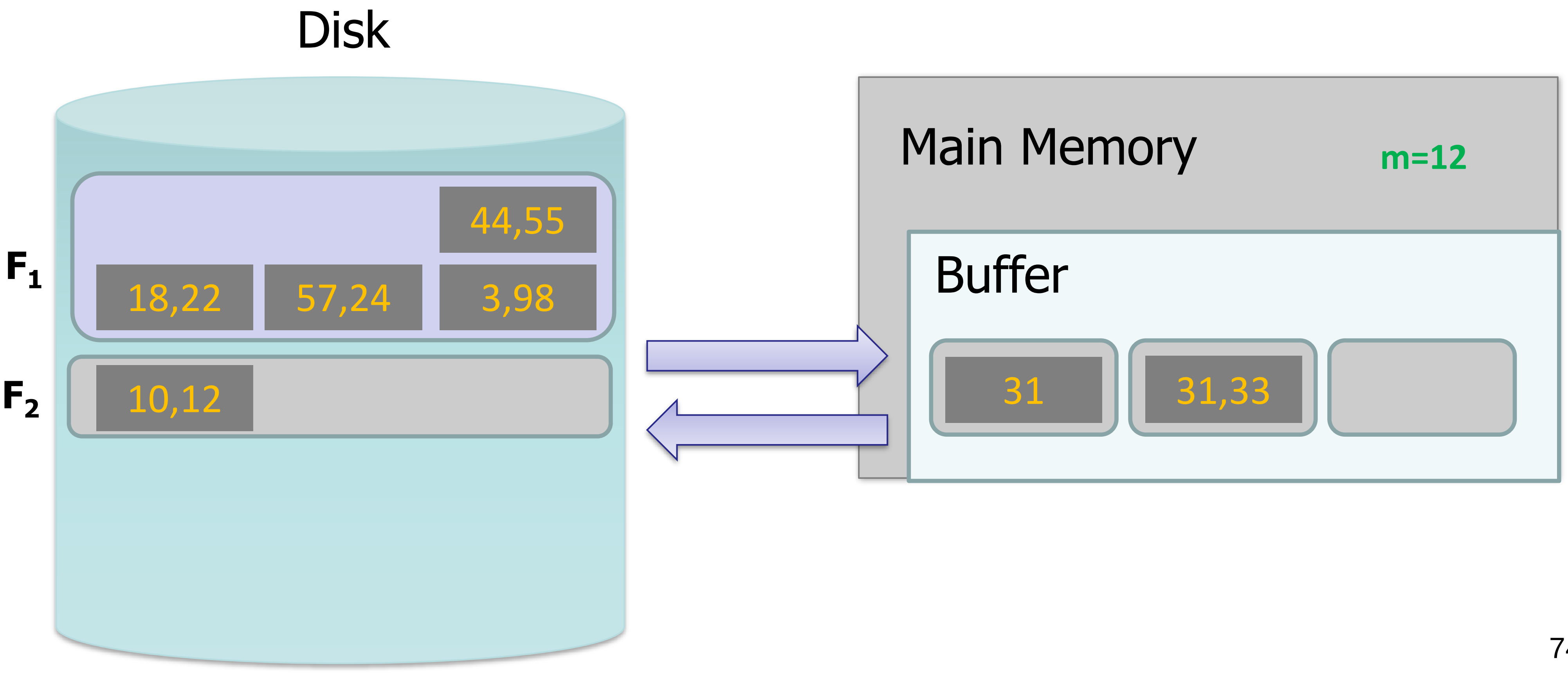
- Take the minimum two values, and put in output page

Also keep track of max (last) value in current run…



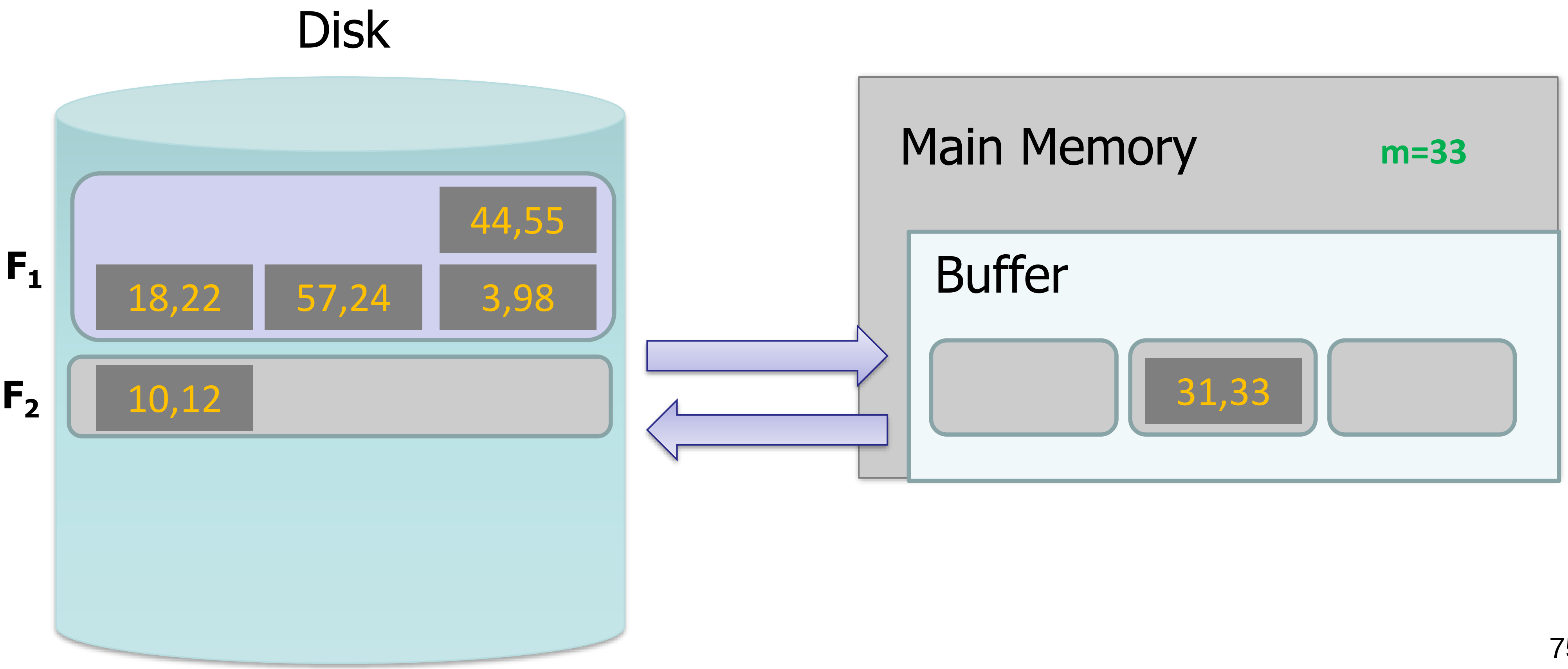
Repacking Example: 3 page buffer

- Next, ***repack***



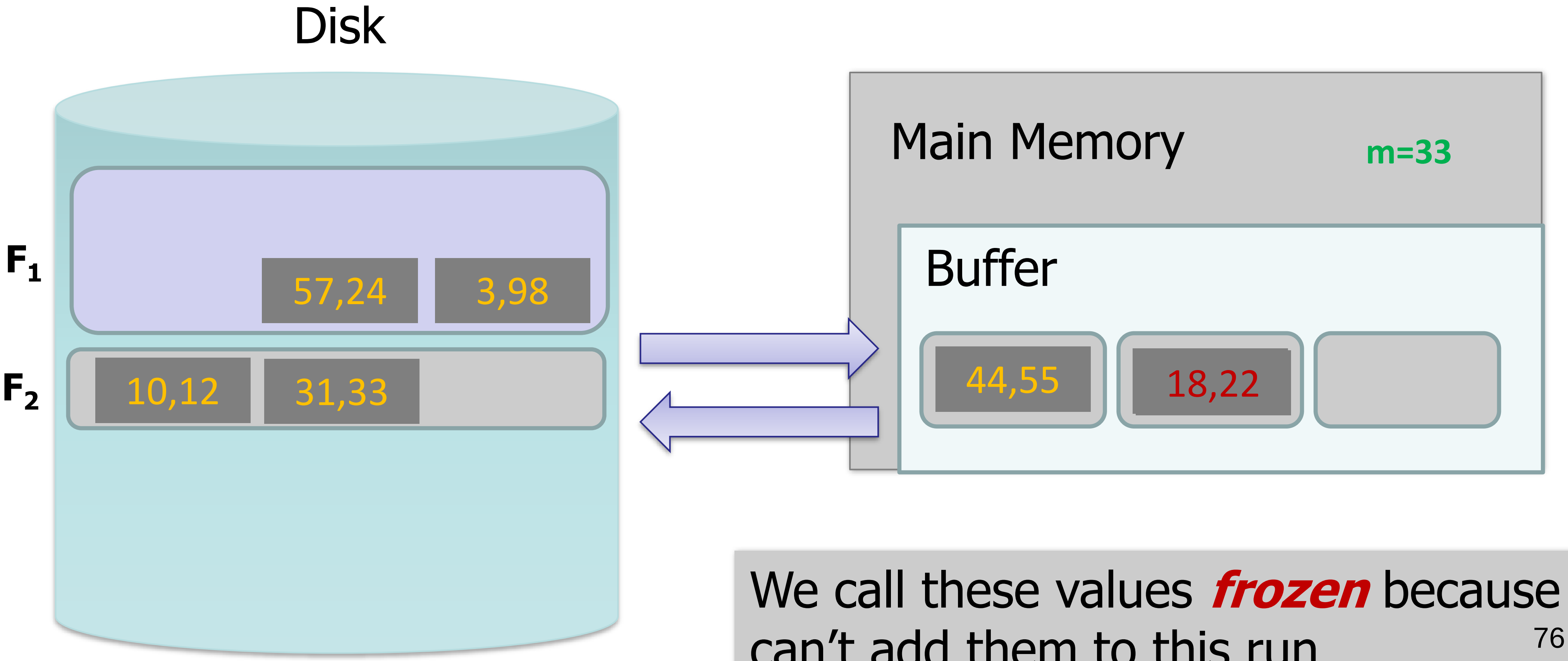
Repacking Example: 3 page buffer

- Next, ***repack***, then load another page and continue!



Repacking Example: 3 page buffer

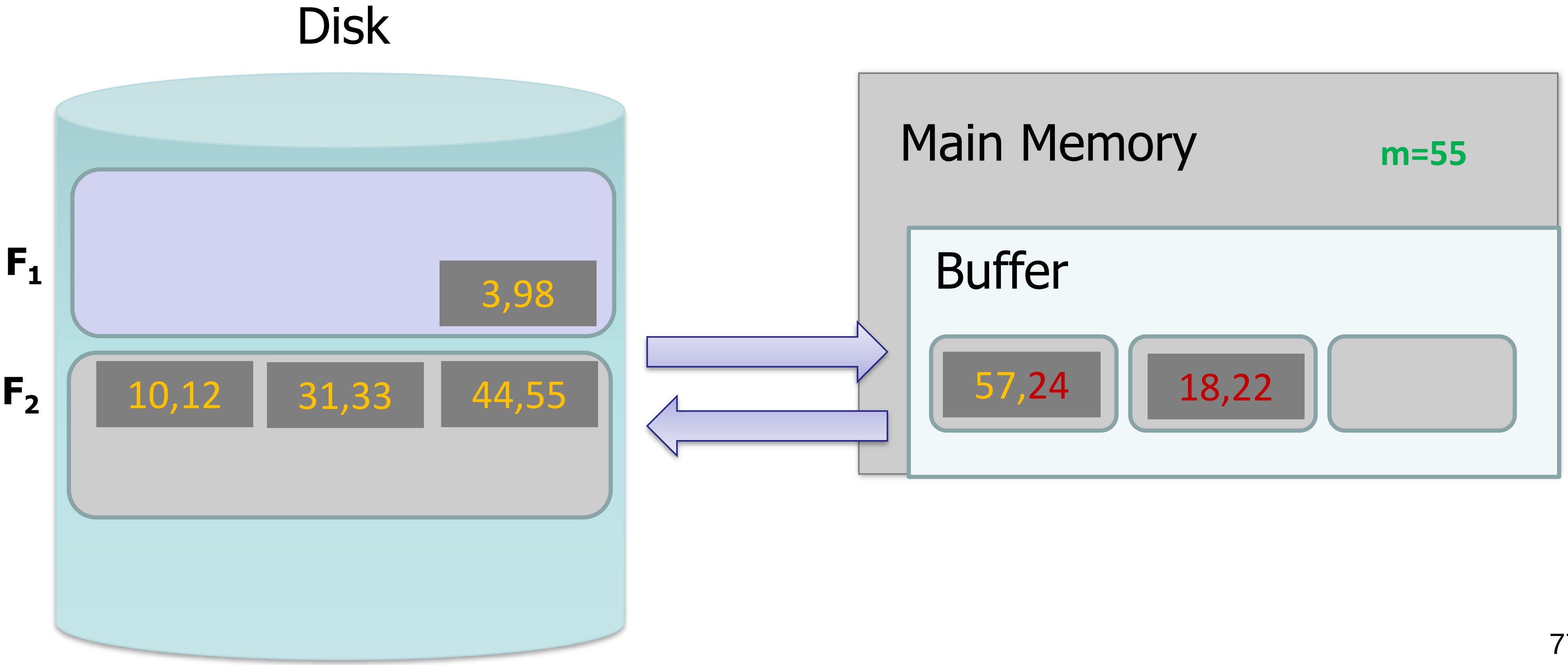
- Now, however, ***the smallest values are less than the largest (last) in the sorted run…***



We call these values ***frozen*** because we can't add them to this run…

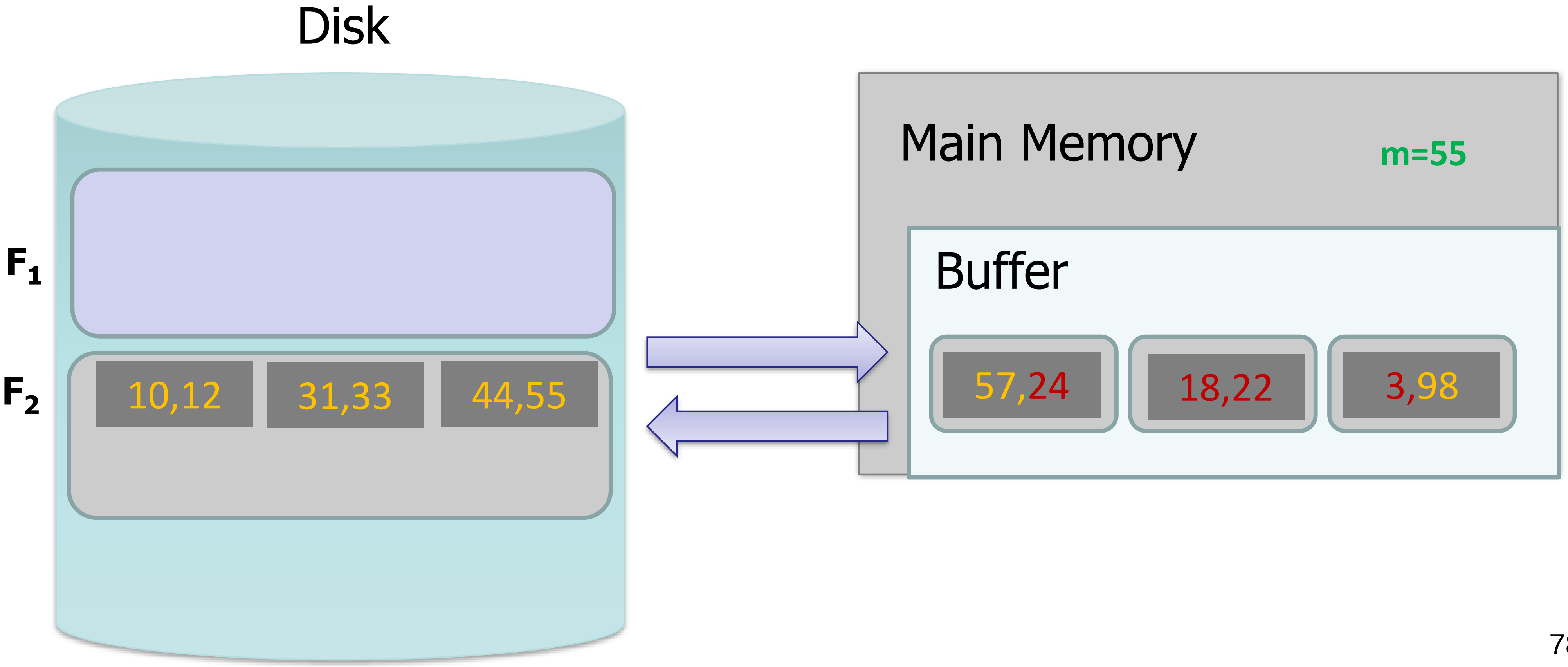
Repacking Example: 3 page buffer

- Now, however, ***the smallest values are less than the largest (last) in the sorted run…***



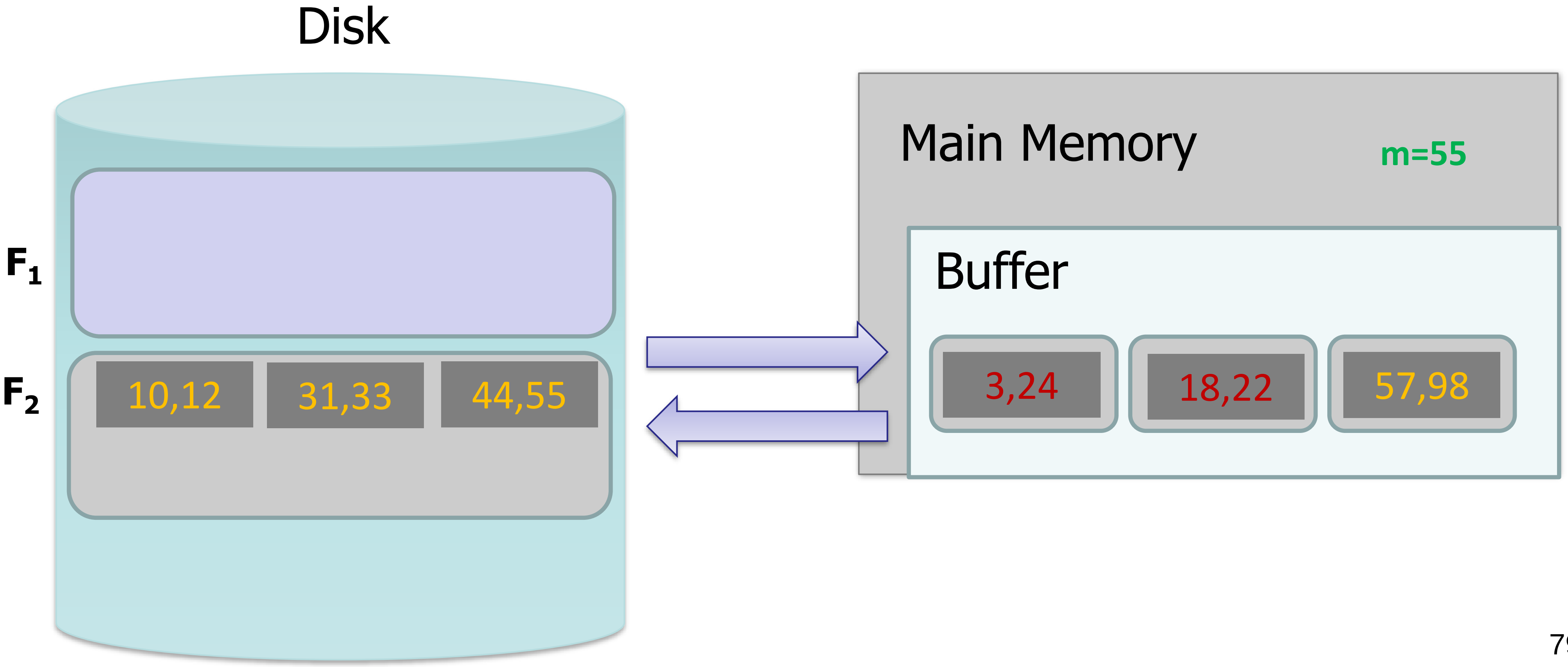
Repacking Example: 3 page buffer

- Now, however, ***the smallest values are less than the largest (last) in the sorted run…***



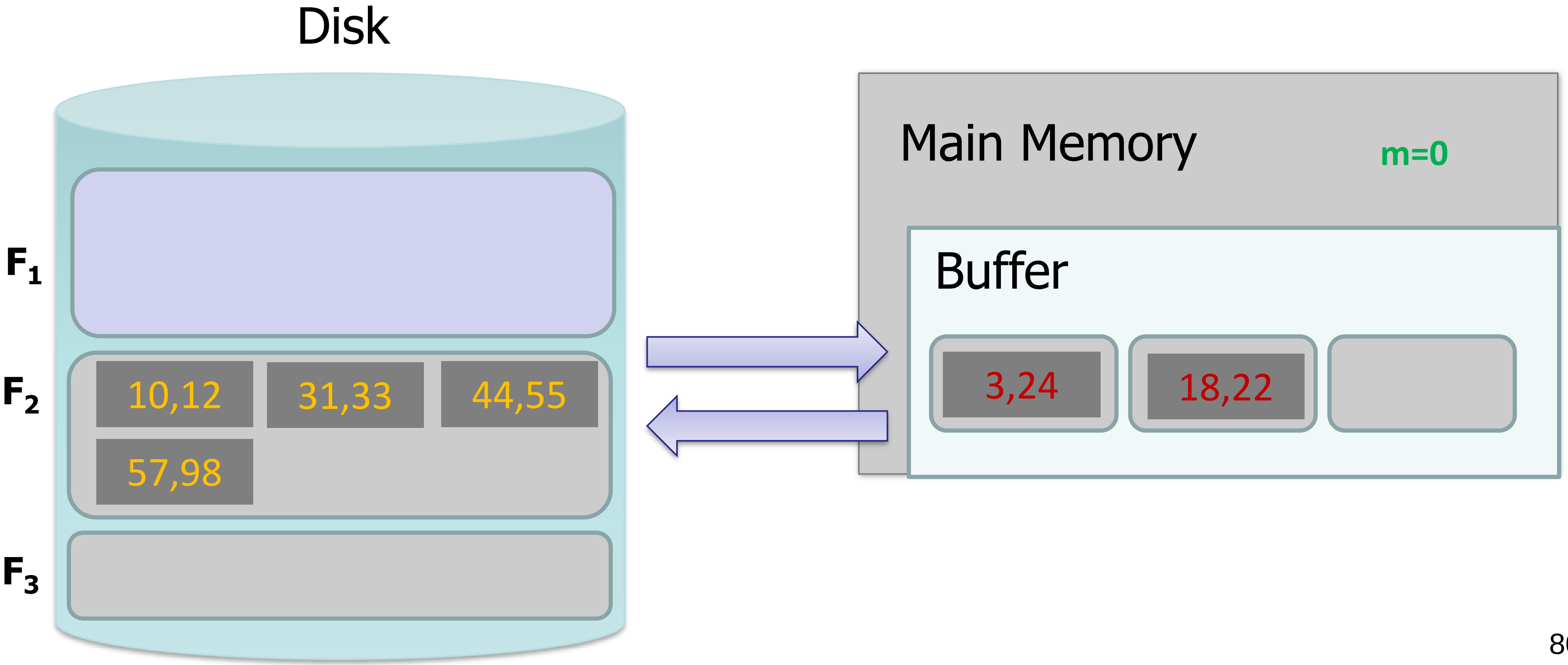
Repacking Example: 3 page buffer

- Now, however, ***the smallest values are less than the largest (last) in the sorted run…***



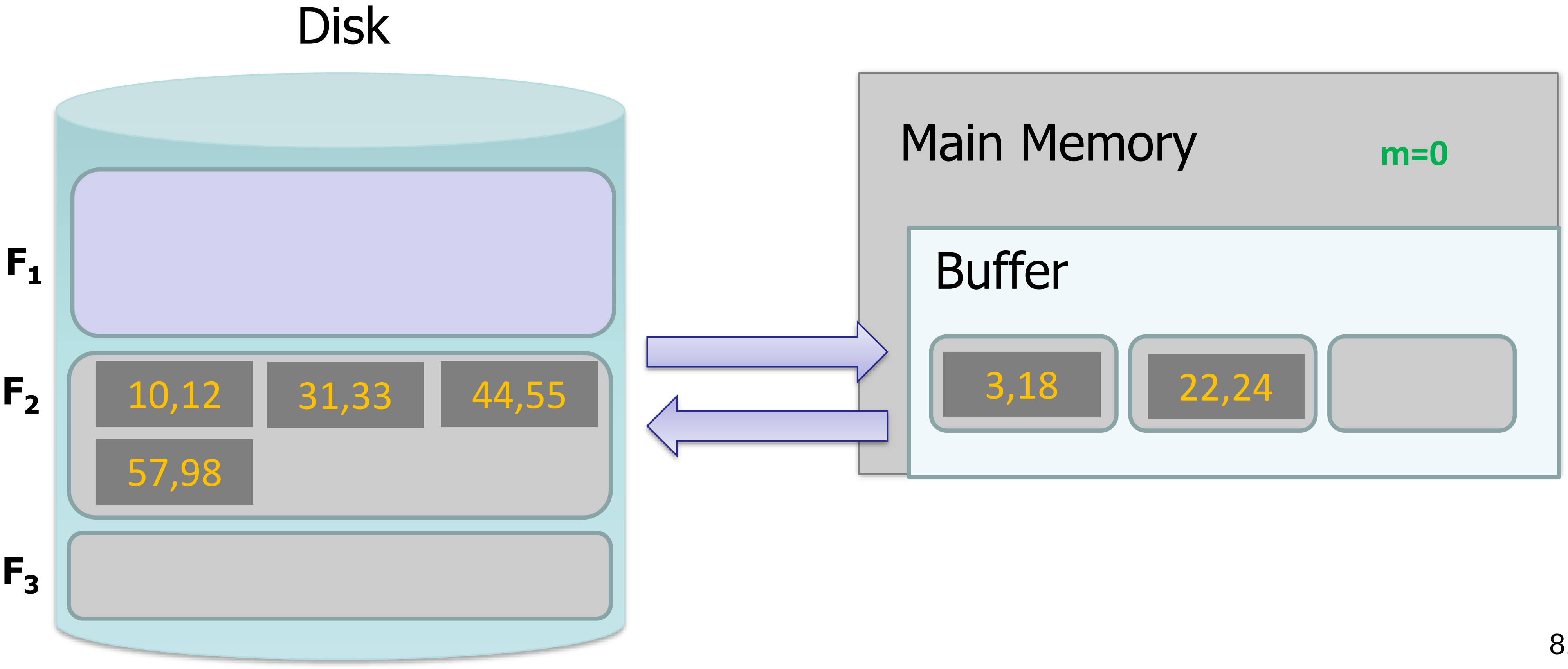
Repacking Example: 3 page buffer

- Once ***all buffer pages have a frozen value***, or input file is empty, start new run with the frozen values



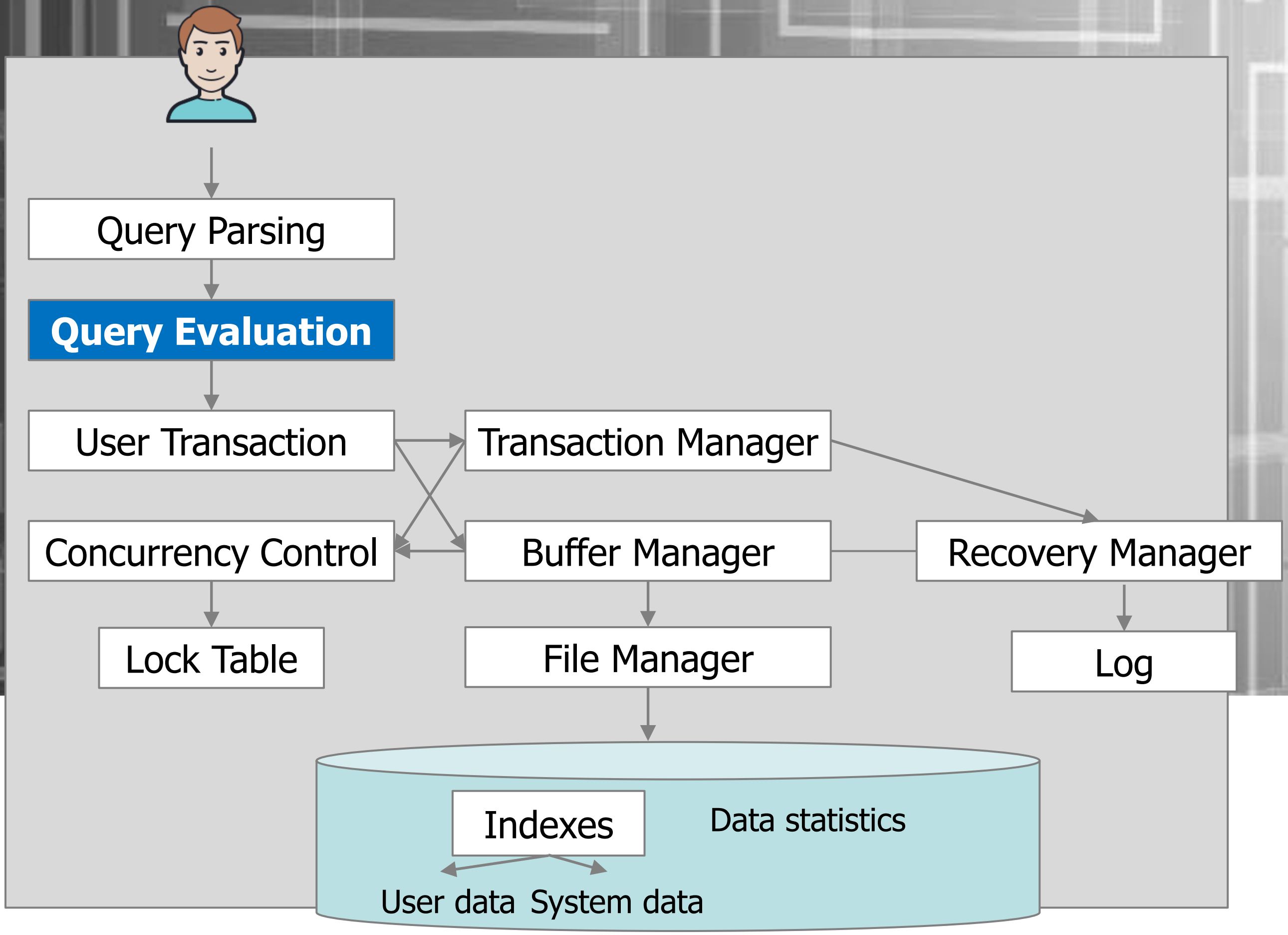
Repacking Example: 3 page buffer

- Once ***all buffer pages have a frozen value***, or input file is empty, start new run with the frozen values



QUERY EVALUATION

1. How to execute joins?



Nested Loop Join Methods: Recap

T(R) = # of tuples in R
P(R) = # of pages in R

Nested Loop Join (NLJ)

Compute R $\bowtie S$ on A:
 for r in R:
  for s in S:
   if r[A] == s[A]:
    yield (r,s)

P(R) + T(R)*P(S) + OUT

Block Nested Loop Join (BNLJ)

Compute R $\bowtie S$ on A:
 for each B-1 pages pr of R:
  for page ps of S:
   for each tuple r in pr:
    for each tuple s in ps:
     if r[A] == s[A]:
      yield (r,s)

$\mathrm{P}(R) + \frac{P(R)}{B-1} P(S)$ + OUT

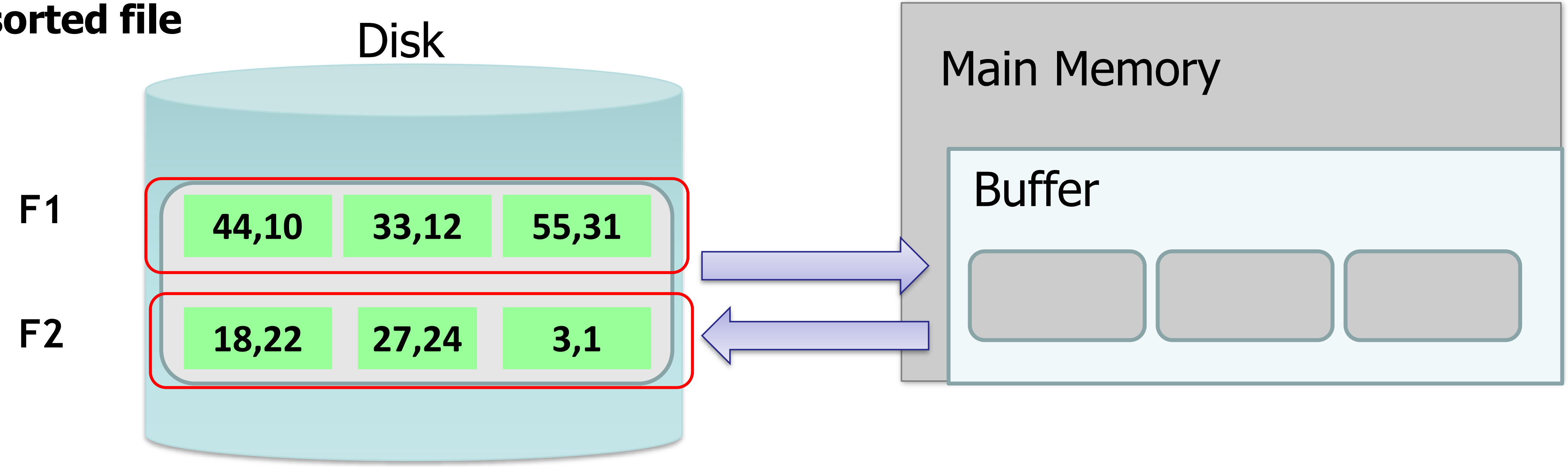
Index Nested Loop Join (INLJ)

Compute R $\bowtie S$ on A:
 Given index idx on S.A:
  for r in R:
   s in idx(r[A]):
    yield r,s

P(R) + T(R)****L**** + OUT

External Merge Sort Algorithm: Recap

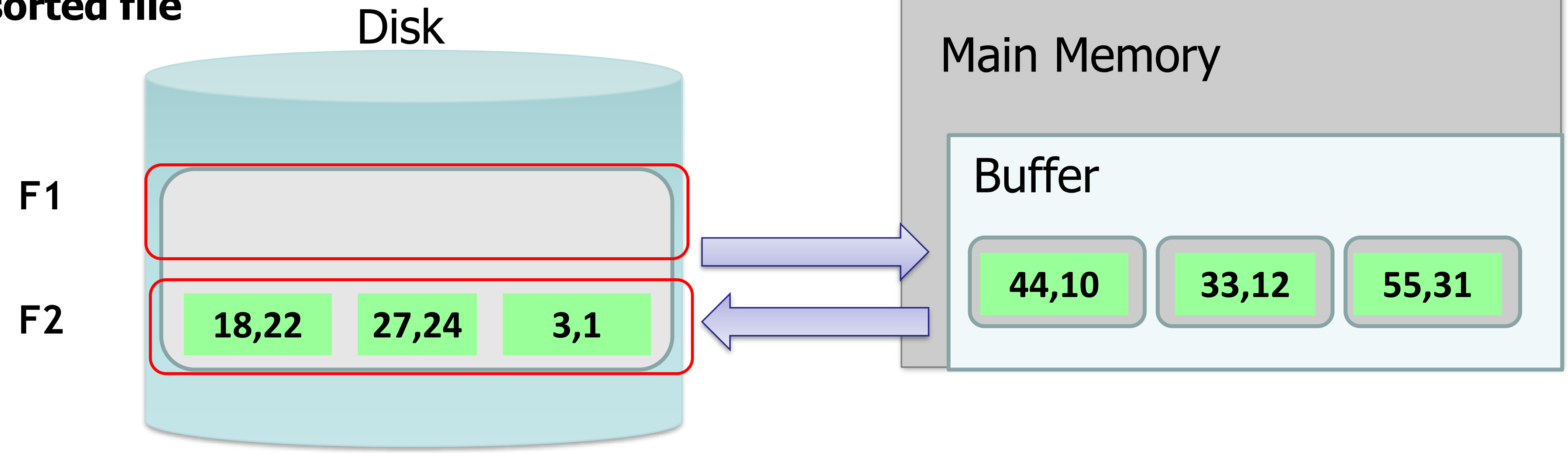
- **3 Buffer pages**
- **6-page file**
- **1 big unsorted file**



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

External Merge Sort Algorithm: Recap

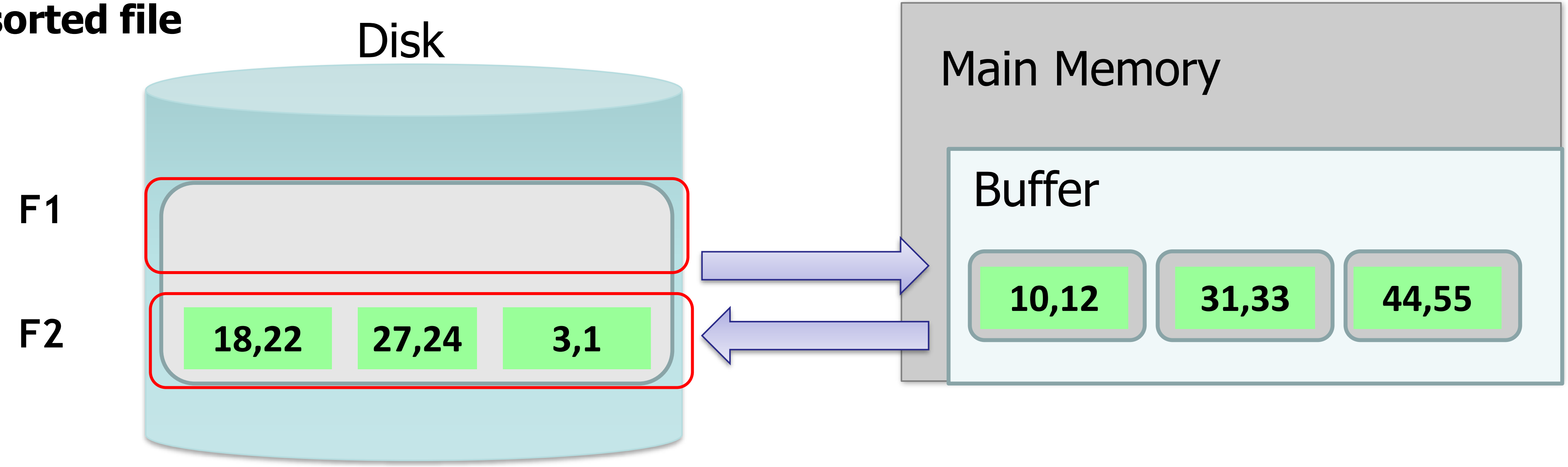
- **3 Buffer pages**
- **6-page file**
- **1 big unsorted file**



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

External Merge Sort Algorithm: Recap

- **3 Buffer pages**
- **6-page file**
- **1 big unsorted file**



1. **Split** into chunks small enough to sort in memory
2. **Sort** in memory

Do the same for F2

3. **Run** the **external merge** algorithm

More on Join Methods

1. Sort-Merge Join (SMJ)

2. Hash Join (HJ)

Sort Merge Join (SMJ): Basic Procedure

To compute $\mathrm{R} \bowtie S \ on \ A$:

Note that we are only considering equality join conditions here

1. Sort R, S on A using ***external merge sort***

2. ***Scan*** sorted files and "merge"

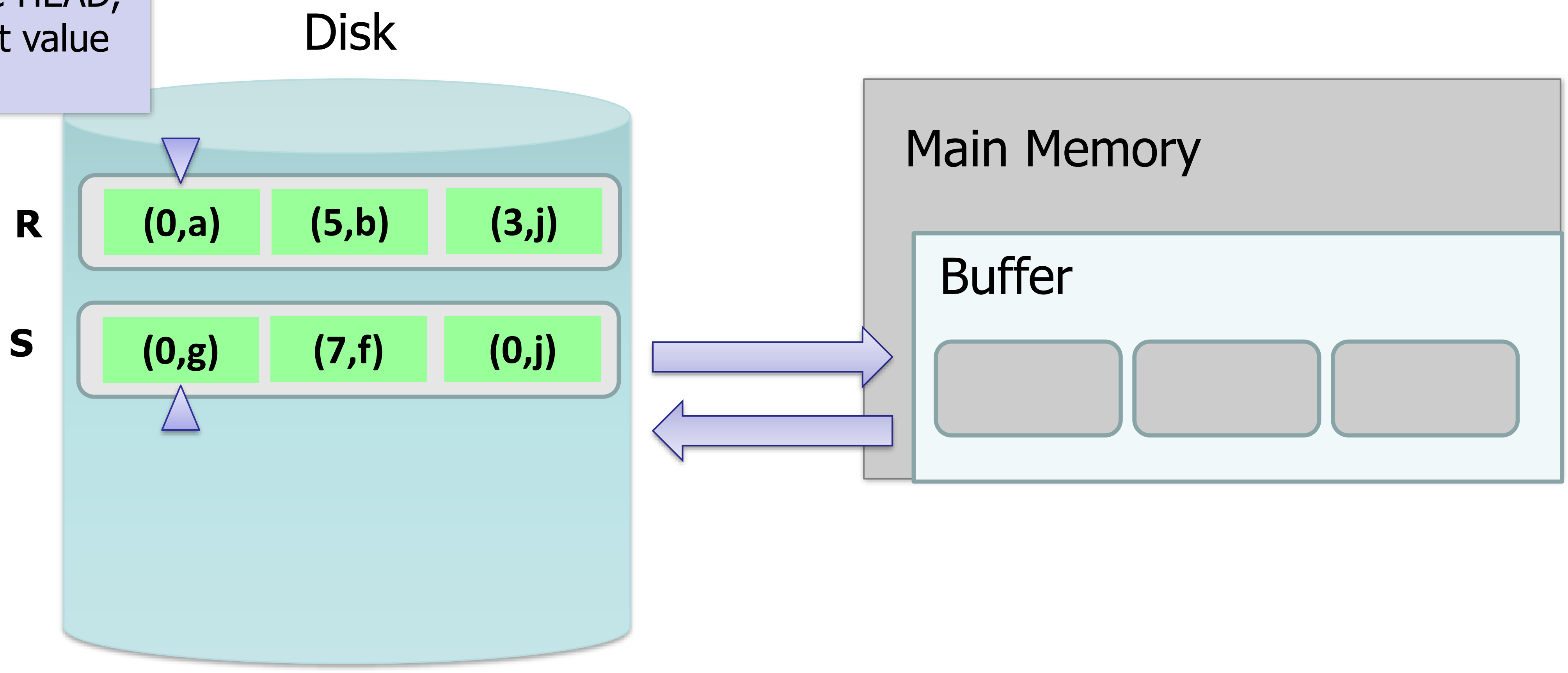
Note that if R, S are already sorted on A, SMJ will be awesome!

SMJ Example

$R \bowtie S \text{ on } A$ with 3 page buffer

- For simplicity: Let each page be ***one tuple***, and let the first value be A

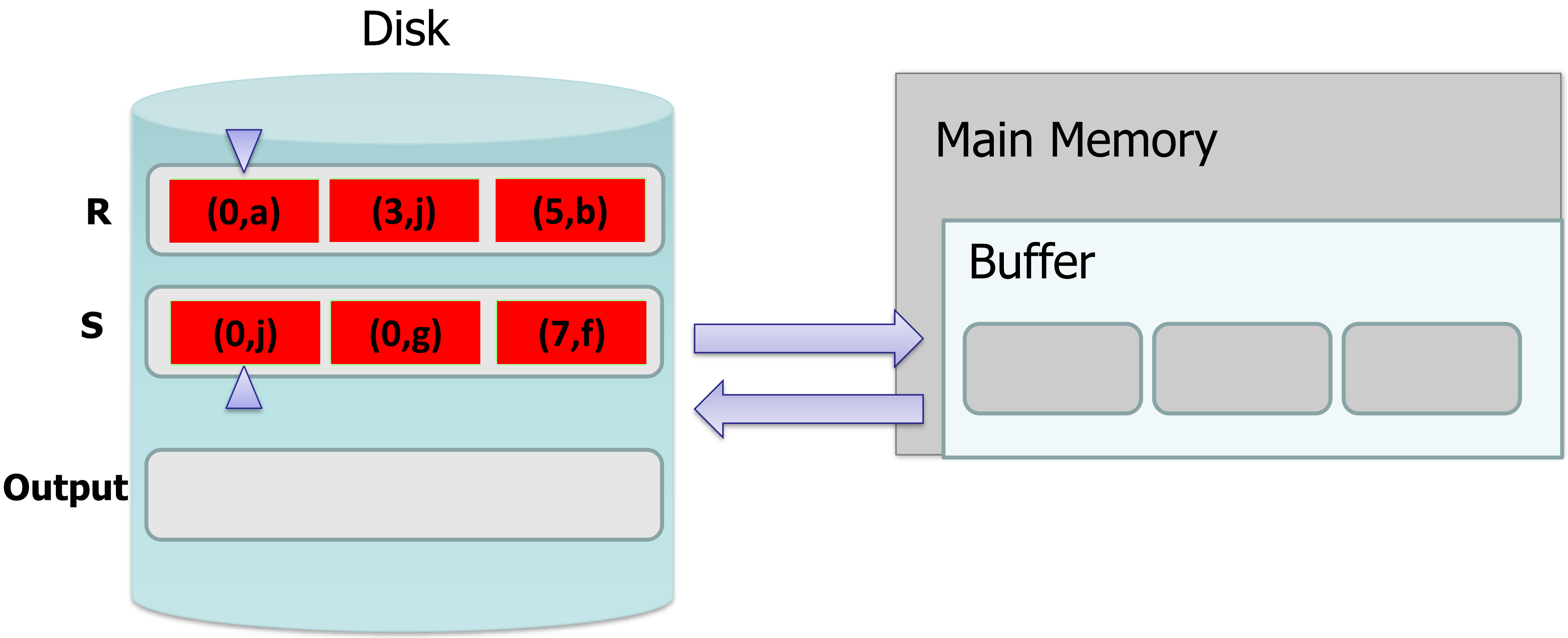
We show the file HEAD, which is the next value to be read!



SMJ Example

$R \bowtie S\ on\ A$ with 3 page buffer

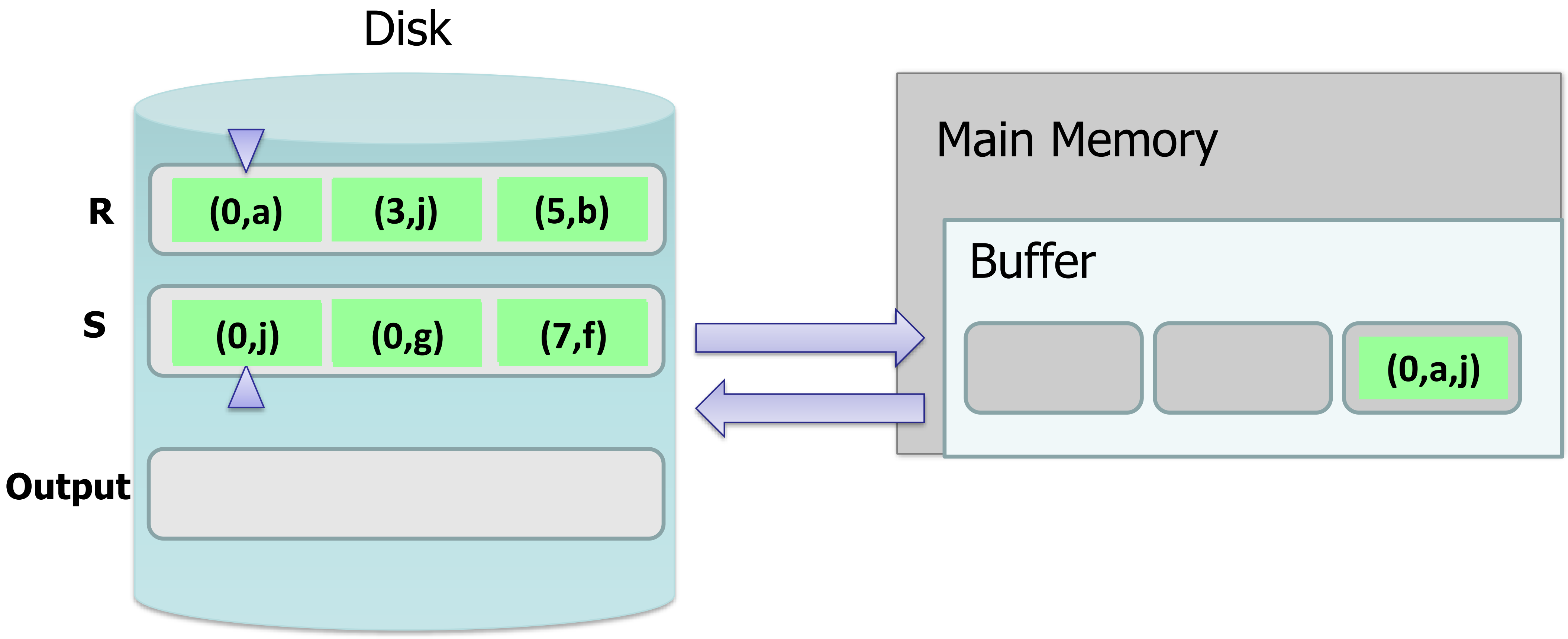
1. Sort the relations R, S on the join key (first value)



SMJ Example

R $\bowtie S$ on A with 3 page buffer

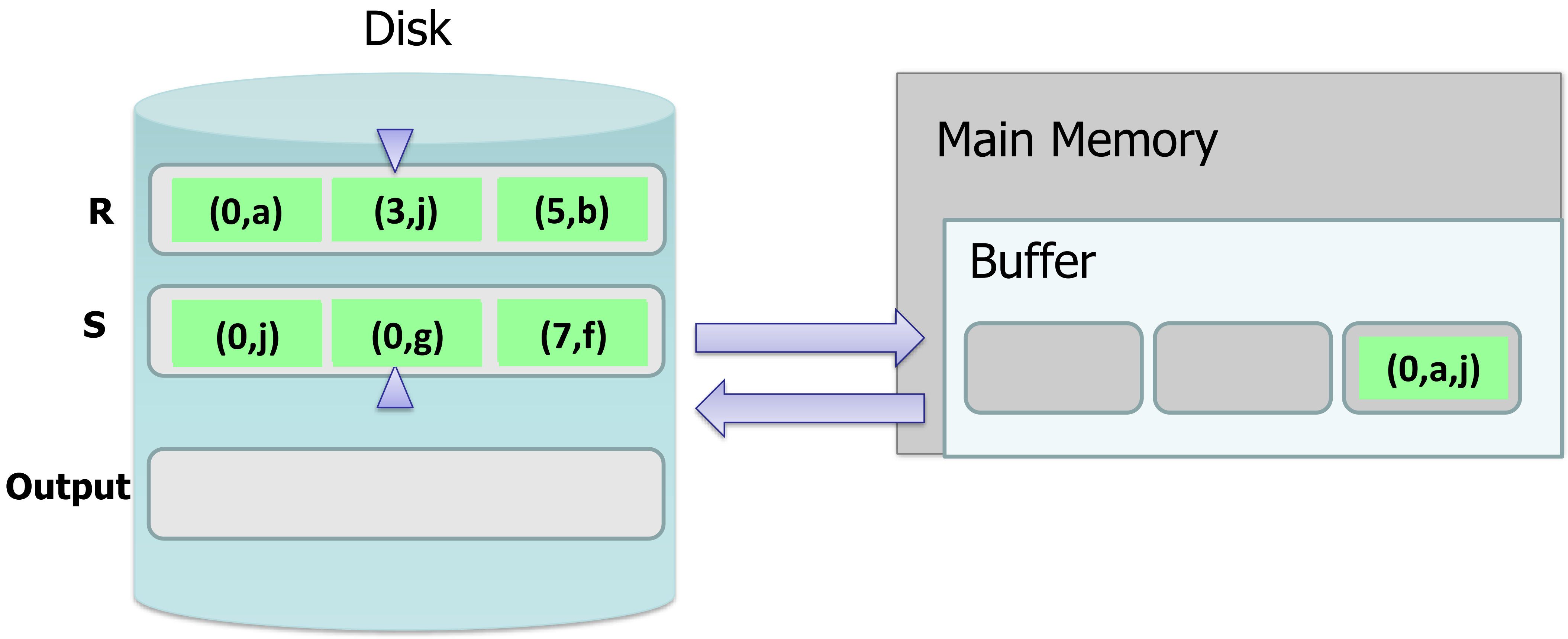
2. Scan and “merge” on join key!



SMJ Example

$\mathrm{R} \bowtie S \ on \ A$ with 3 page buffer

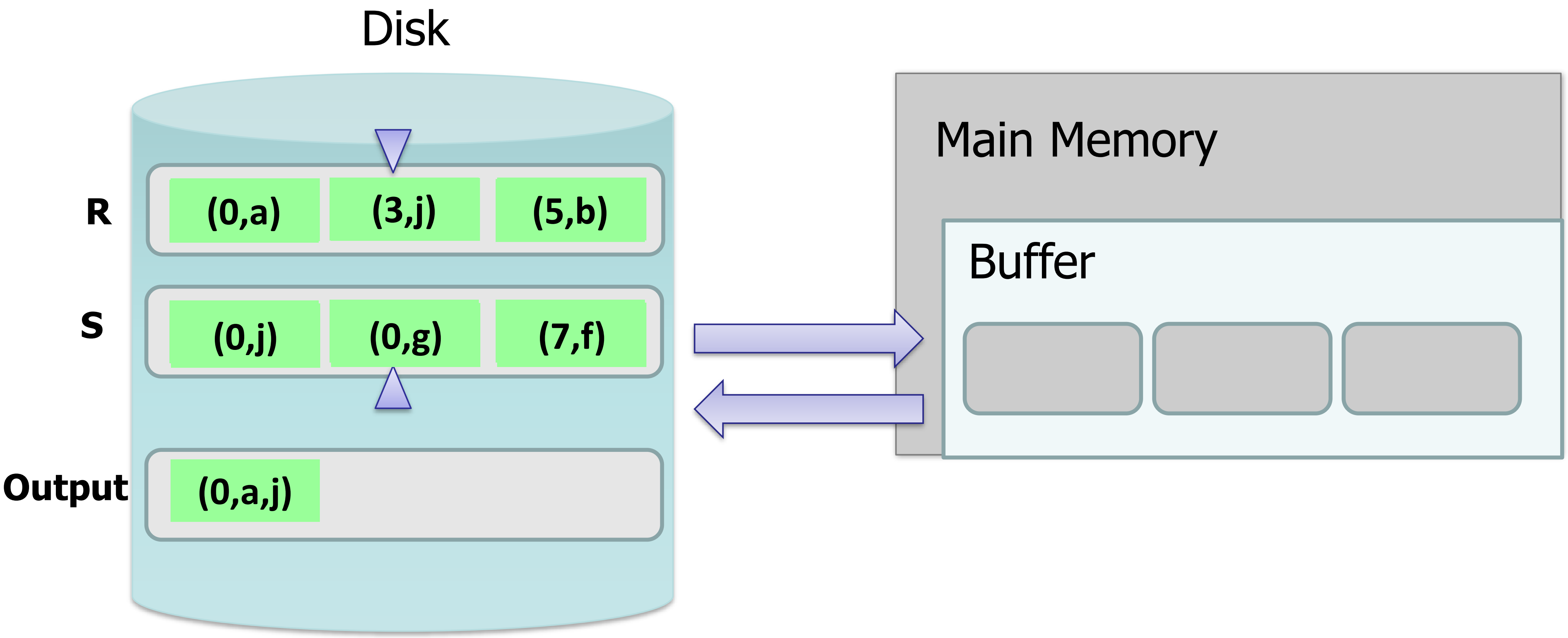
2. Scan and “merge” on join key!



SMJ Example

$R \bowtie S \text{ on } A$ with 3 page buffer

2. Scan and "merge" on join key!

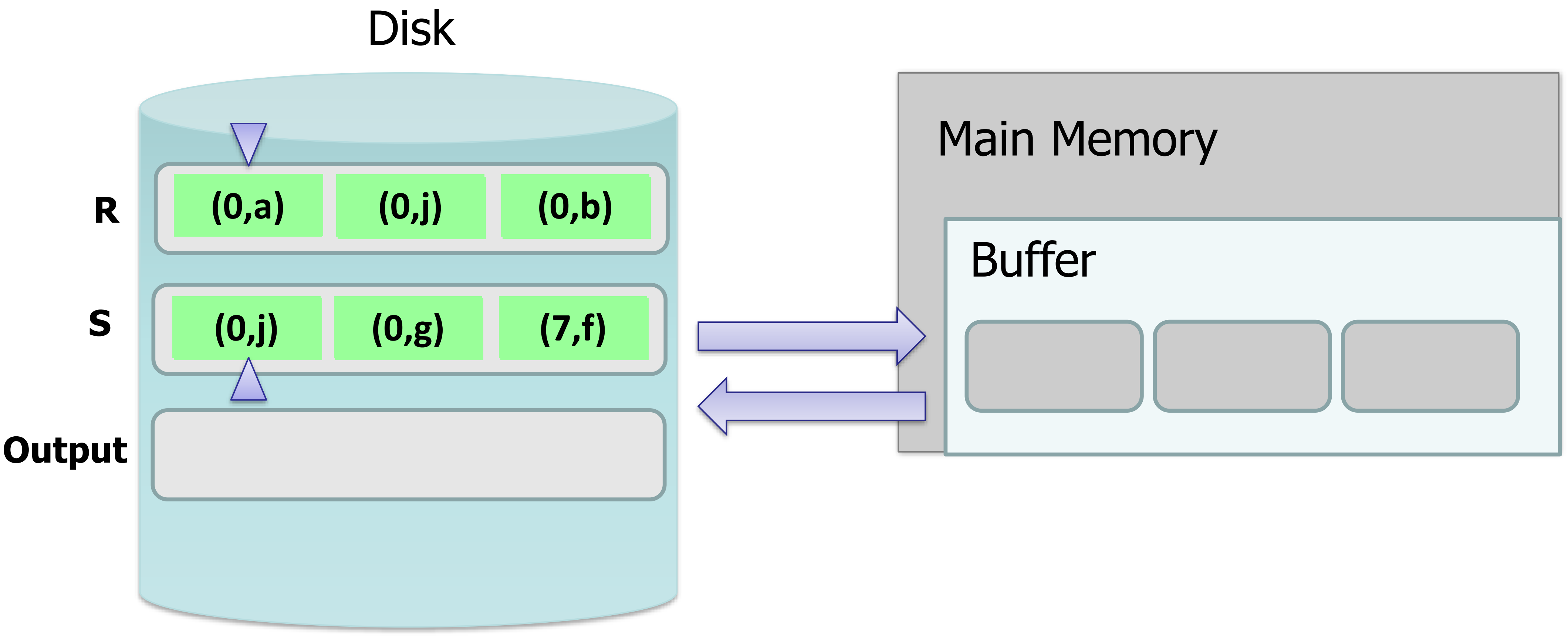


What happens with duplicate join keys?

Multiple tuples with Same Join Key

"Backup"

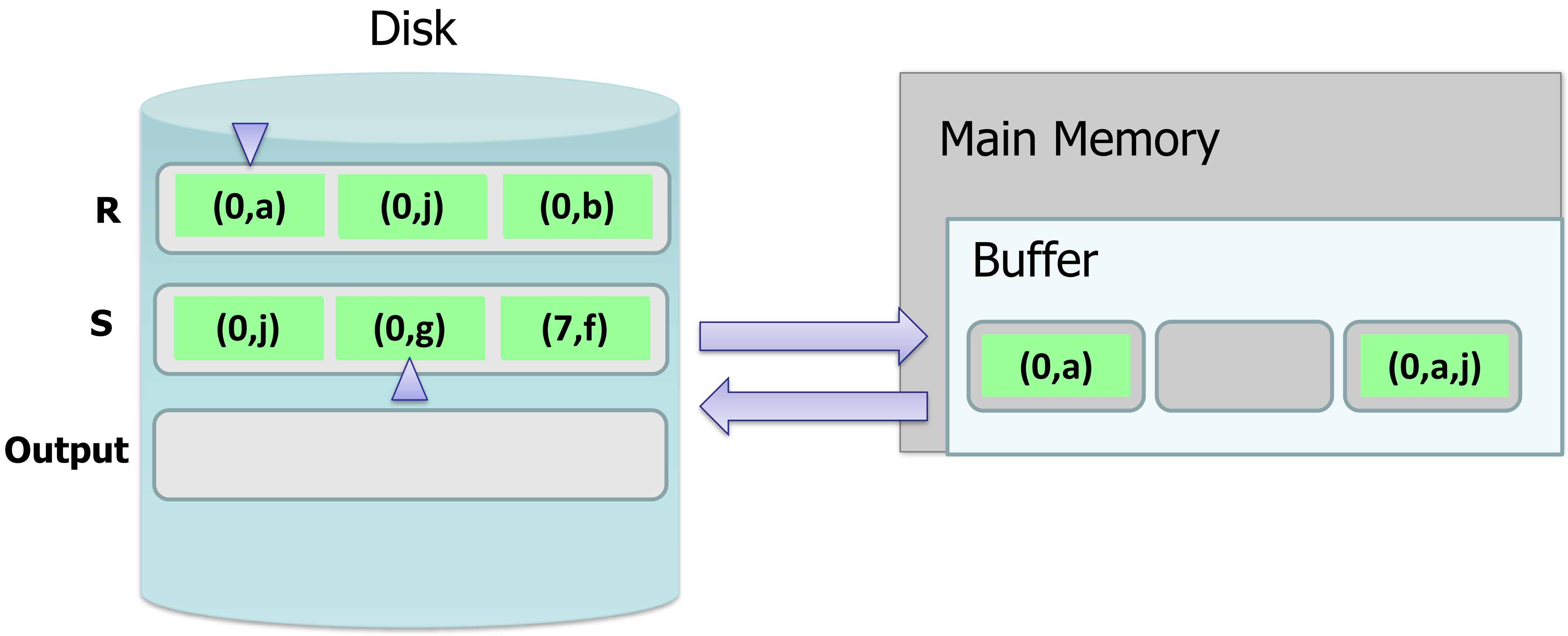
1. Start with sorted relations, and begin scan / merge…



Multiple tuples with Same Join Key

"Backup"

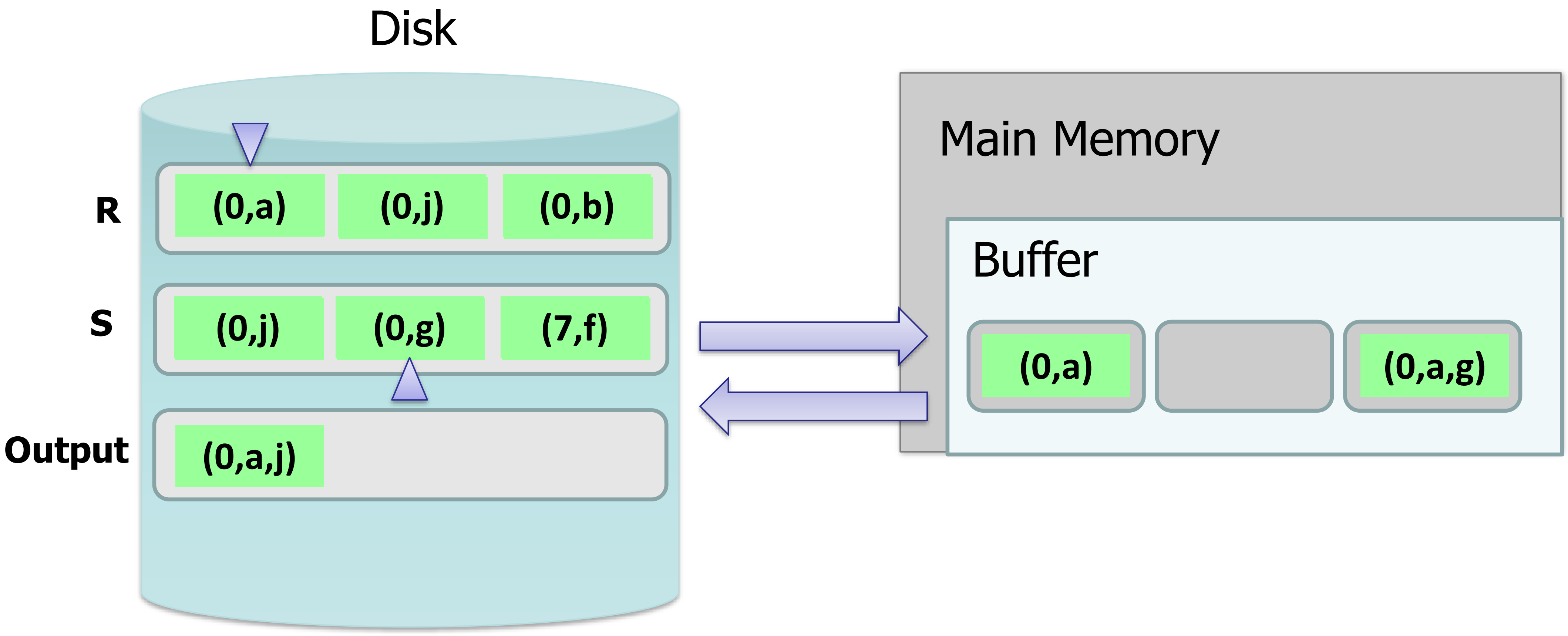
1. Start with sorted relations, and begin scan / merge…



Multiple tuples with Same Join Key

"Backup"

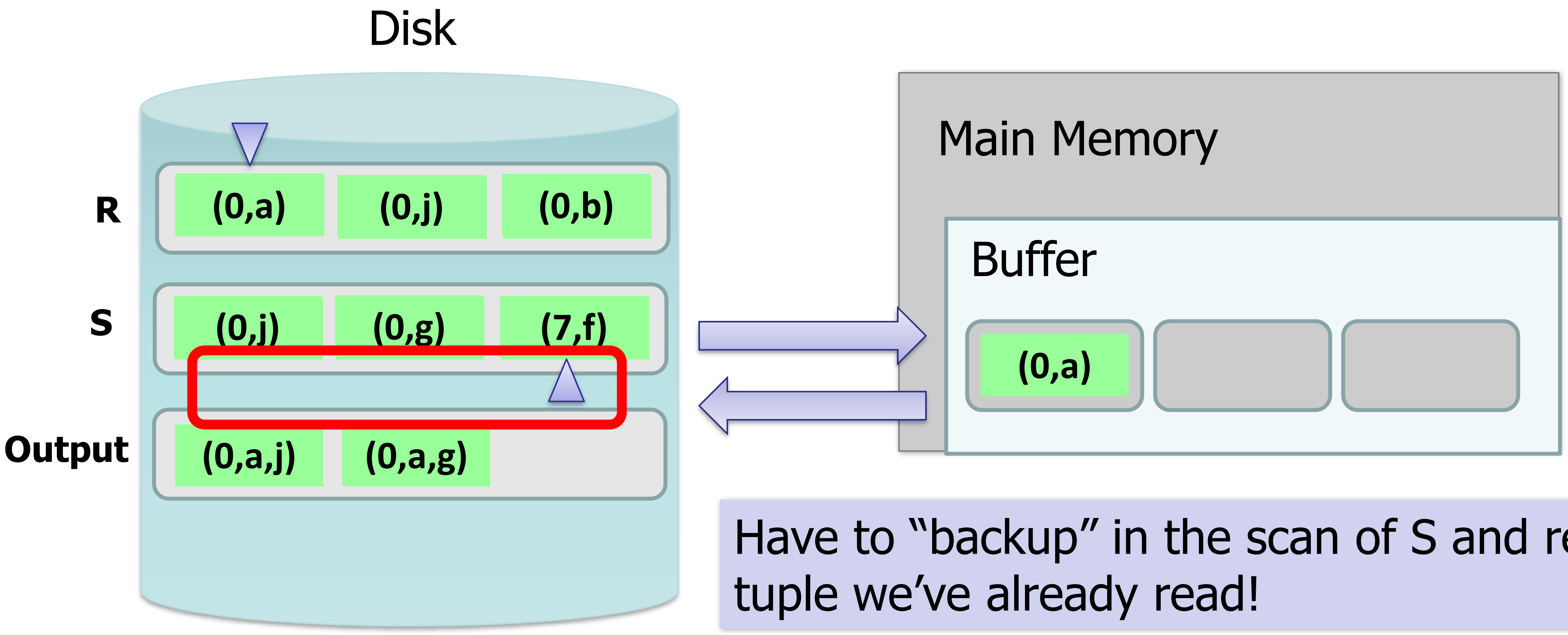
1. Start with sorted relations, and begin scan / merge…



Multiple tuples with Same Join Key

"Backup"

1. Start with sorted relations, and begin scan / merge…

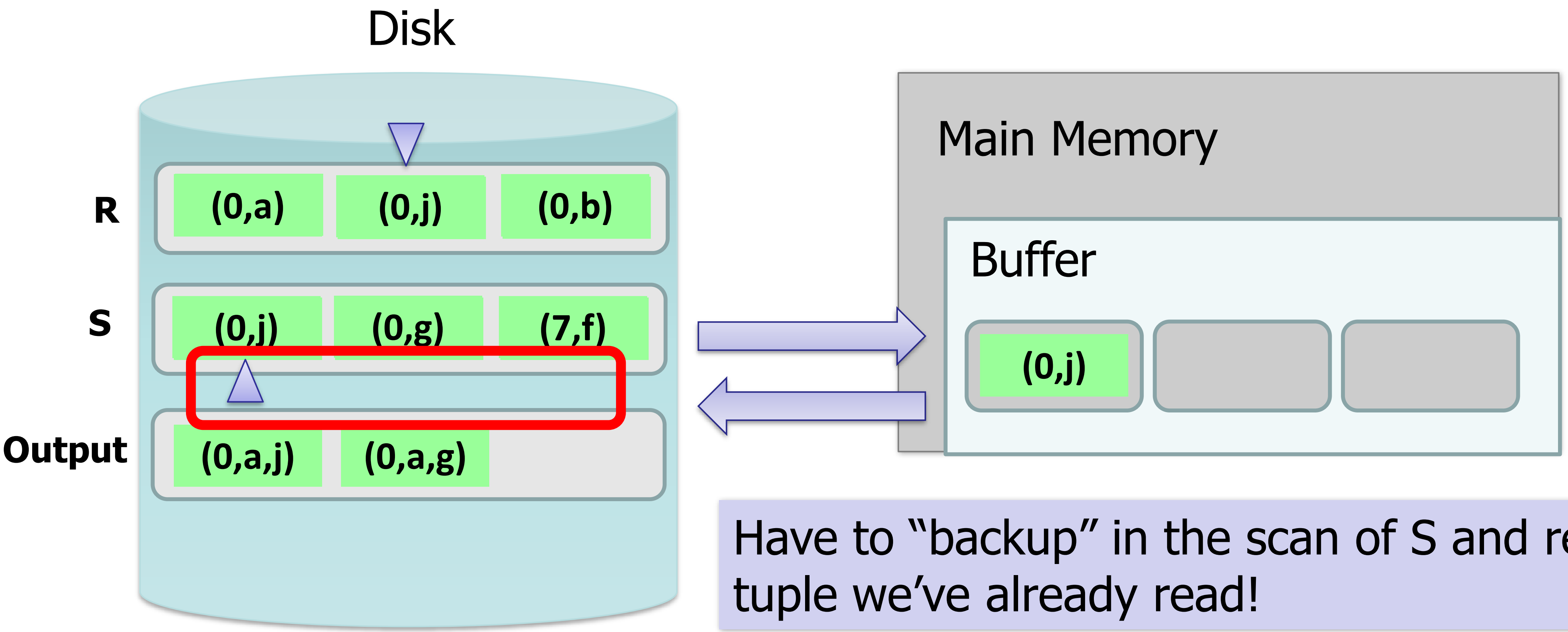


Have to "backup" in the scan of S and read tuple we've already read!

Multiple tuples with Same Join Key

"Backup"

1. Start with sorted relations, and begin scan / merge…



Have to "backup" in the scan of S and read tuple we've already read!

Backup

- At best, no backup → scan takes **P(R)** + **P(S)** reads
  - For ex: if no duplicate values in join attribute
- At worst (e.g. full backup each time), scan could take **P(R)** * **P(S)** reads!
  - For ex: if *all* duplicate values in join attribute, i.e. all tuples in R and S have the same value for the join attribute
  - Roughly: For each page of R, we'll have to *back up* and read each page of S…
- Often not that bad however, plus we can:
  - Leave more data in buffer (for larger buffers)

SMJ: Total cost

- Cost of SMJ is **cost of sorting** R and S…
- Plus the **cost of scanning**: ~P(R)+P(S)
  - Because of *backup*: in worst case P(R)*P(S); but this would be very unlikely
- Plus the **cost of writing out**: ~P(R)+P(S) but in worst case T(R)*T(S)

~ Sort(P(R)) + Sort(P(S)) + P(R) + P(S) + OUT

Recall: Sort(N) $\approx 2N\left(\left\lceil \log_B \frac{\boldsymbol{N}}{\boldsymbol{2(B+1)}} \right\rceil + 1\right)$

Note: this is using repacking, where we estimate that we can create initial runs of length ~2(B+1)

SMJ vs. BNLJ

Sort(P(R)) + Sort(P(S)) + P(R) + P(S) + OUT **SMJ**

$\mathrm{P}(R) + \frac{P(R)}{B-1} P(S)$ + OUT **BNLJ**

- If we have B+1=100 buffer pages, P(R) = 1000 pages and P(S) = 500 pages:
  - Sort both in two passes: 2 * 2 * 1000 + 2 * 2 * 500 = **6,000 IOs**
  - Merge phase 1000 + 500 = 1,500 IOs
  - **<u>= 7,500 IOs + OUT</u>**

What is BNLJ?

  - 500 + 1000* $\left\lceil \frac{500}{98} \right\rceil$ = **<u>6,500 IOs + OUT</u>**

- But, if we have 35 buffer pages?
  - Sort Merge has same behavior (still 2 passes)
  - BNLJ? ***<u>15,500 IOs + OUT!</u>***

SMJ is ~ linear vs. BNLJ is quadratic…
But it's all about the memory.

A Simple Optimization: Merges Merged!

Given ***B+1*** buffer pages

- SMJ is composed of a ***sort phase*** and a ***merge phase***

- During the ***sort phase***, run passes of external merge sort on R and S
  - Suppose at some point, R and S have <= ***B*** (sorted) runs in total

  - We could do two merges (for each of R & S) at this point, complete the sort phase, and start the merge phase…

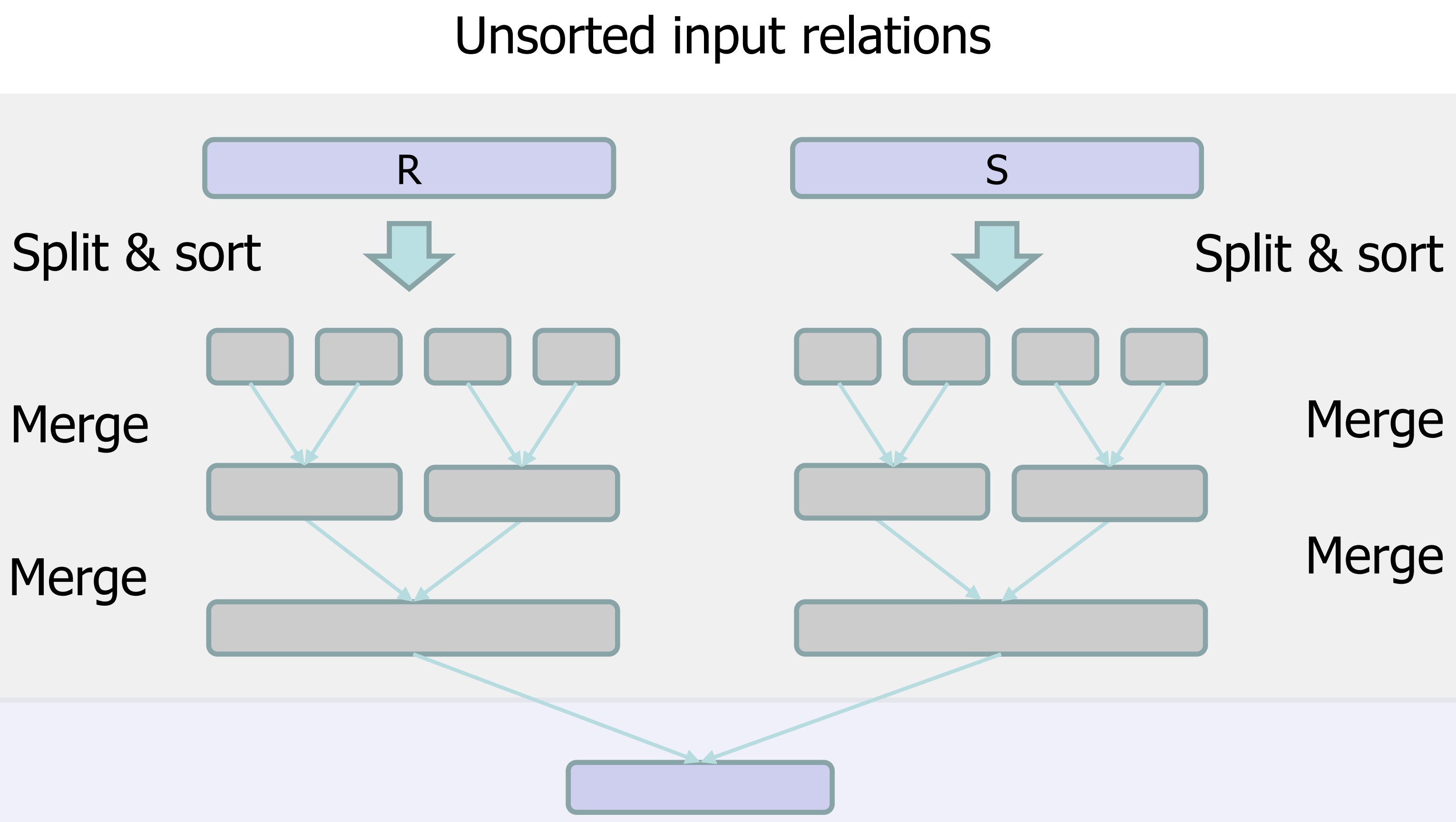
  - OR, we could combine them: do **one** B-way merge and complete the join!

Un-Optimized SMJ

Given ***B+1*** buffer pages

Sort Phase (Ext. Merge Sort)

Merge / Join Phase



Joined output file created!

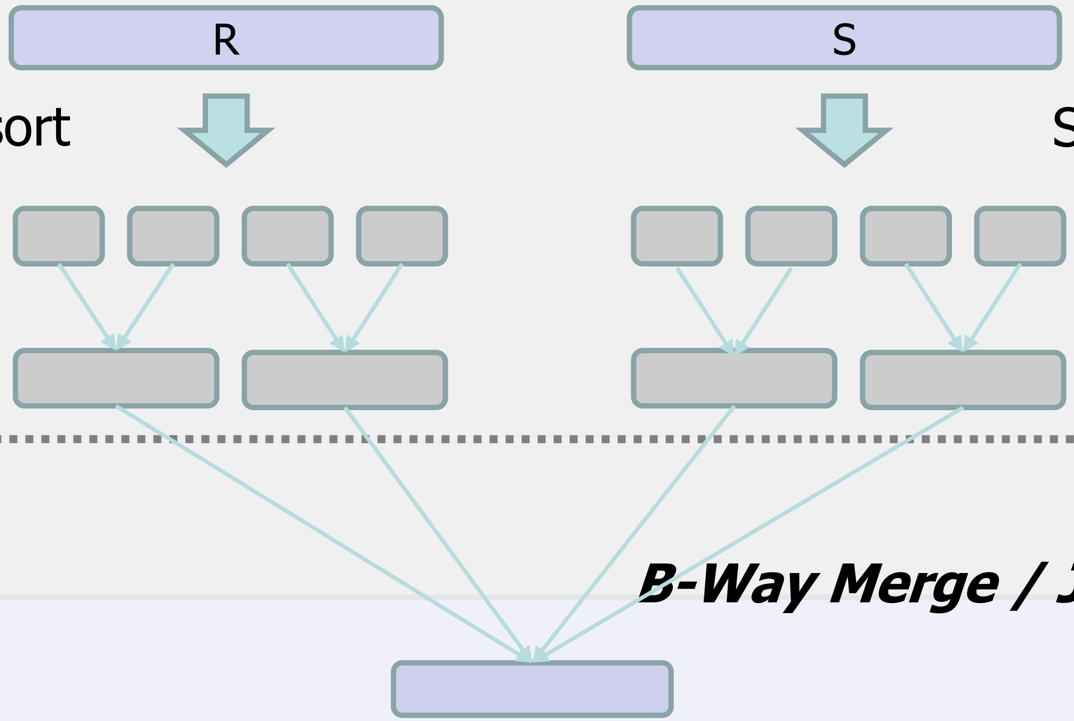
Simple SMJ Optimization

Given ***B+1*** buffer pages

Unsorted input relations

Sort Phase (Ext. Merge Sort)

<= B total runs



Merge / Join Phase

Joined output file created!

Simple SMJ Optimization

Given ***B+1*** buffer pages

- Now, on this last pass, we only do P(R) + P(S) IOs to complete the join!

- If we can initially split R and S into **B total runs each of length approx. <= 2(B+1)**, *assuming repacking lets us create initial runs of ~2(B+1)*- then we only need ***3(P(R) + P(S)) + OUT*** for SMJ!
  - 2 R/W per page to sort runs in memory, 1 R per page to B-way merge / join!

- How much memory for this to happen?
  - $\frac{P(R)+P(S)}{B} \leq 2(B+1) \Rightarrow \sim \mathrm{P(R)} + \mathrm{P(S)} \leq 2B^2$
  - **Thus,** $\mathbf{max\{P(R), P(S)\}} \leq \boldsymbol{B^2}$ **is an approximate sufficient condition**

If the larger of R,S has <= B^2 pages, then SMJ costs **3(P(R)+P(S)) + OUT**!

Takeaway points from SMJ

If input already sorted on join key, skip the sorts.

– SMJ is basically linear.
– Nasty but unlikely case: Many duplicate join keys.