

CSC 261/461 – Database Systems

Lecture 19

Spring 2018

Sections to Study

Chapter 18

- 18.4.1 (J3 and J4)
- 18.4.4
- Section 19.1

Chapter 19

- Very Important: 19.1.2. We have gone through the material but you must study this section for better understanding.
If must understand each and every step in Figure 19.2 is performed. Study the last paragraph: Summary of Heuristics for Algebraic Optimization.
- 19.3.2
- 19.3.3
- 19.4 (S1 to S6)
- 19.5 (J1-J4) (Important. We cover the exact same topic using easier representation.)

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??**
 - What about if we need to sort **1TB** of data with **1GB** of RAM...

A classic problem in computer science!

So how do we sort big files?

1. Split into chunks small enough to **sort in memory** (*“runs”*)
2. **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!

Before we proceed..

- Question: Assume each page/block can contain 100 records. You have a file containing 80 unsorted records. What is the IO cost of sorting this file.

A. $80 * \log_2 80$ IO

B. 1 IO

C. 2 IO

D. $\log_2 80$ IO

Before we proceed..

- Question: Assume each page/block can contain 100 records. You have a file containing 80 unsorted records. What is the IO cost of sorting this file.

A. $80 * \log_2 80$ IO

B. 1 IO

C. 2 IO

D. $\log_2 80$ IO

EXTERNAL MERGE & SORT

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 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:** $2(M+N)$

Key (Simple) Idea

To find an element that is no larger than all elements in two lists, one only needs to compare minimum elements from each list.

If:

$$A_1 \leq A_2 \leq \dots \leq A_N$$
$$B_1 \leq B_2 \leq \dots \leq B_M$$

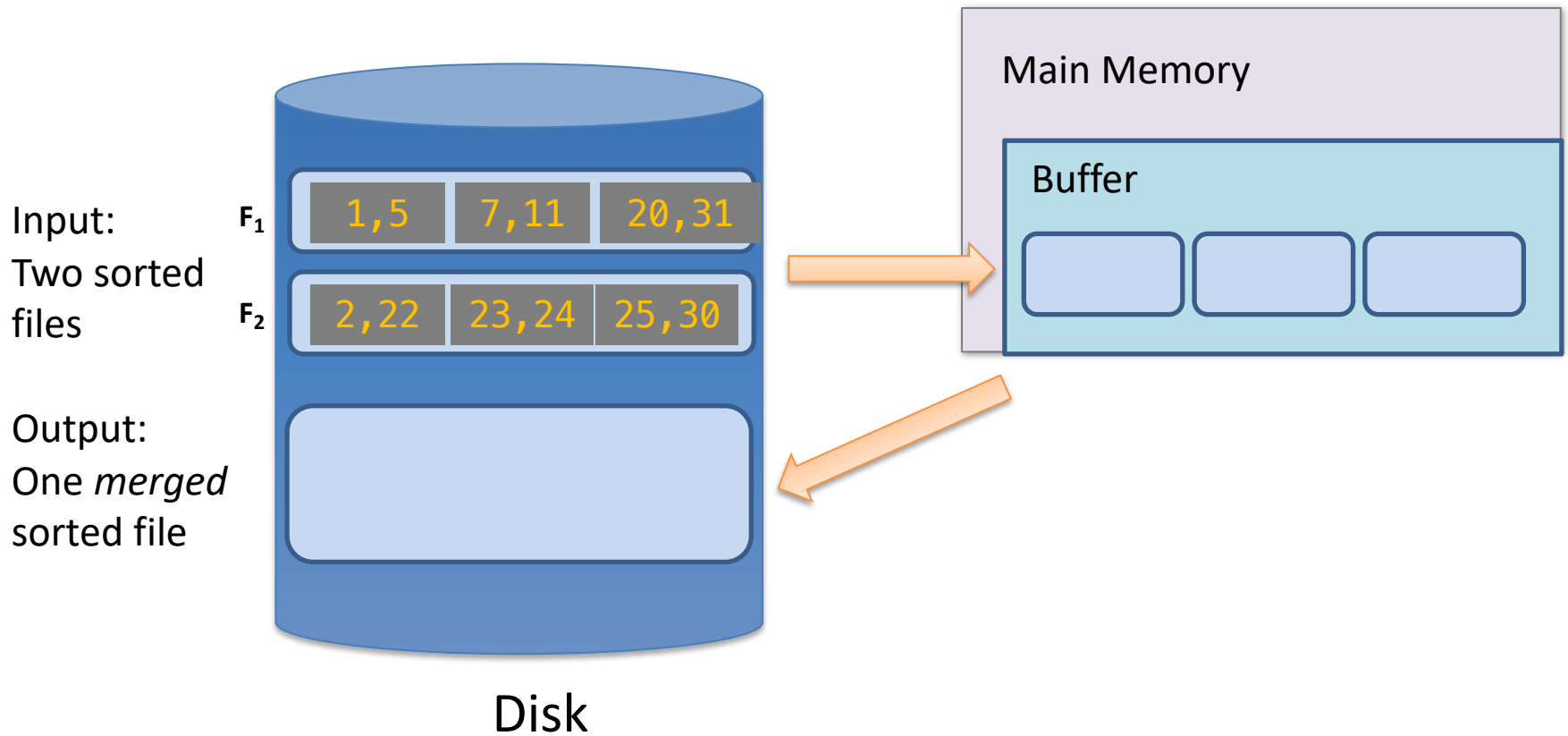
Then:

$$\text{Min}(A_1, B_1) \leq A_i$$

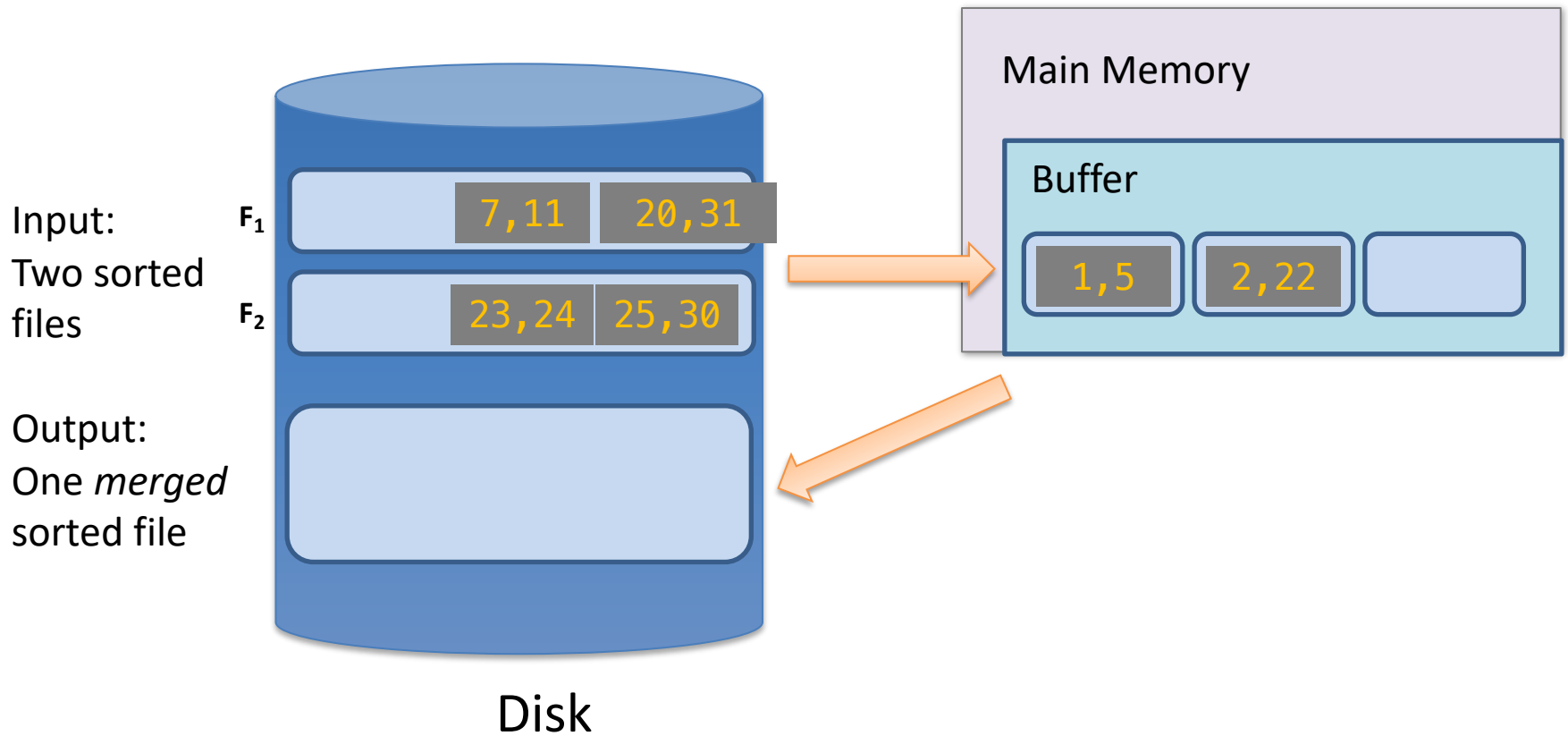
$$\text{Min}(A_1, B_1) \leq B_j$$

for $i=1\dots N$ and $j=1\dots M$

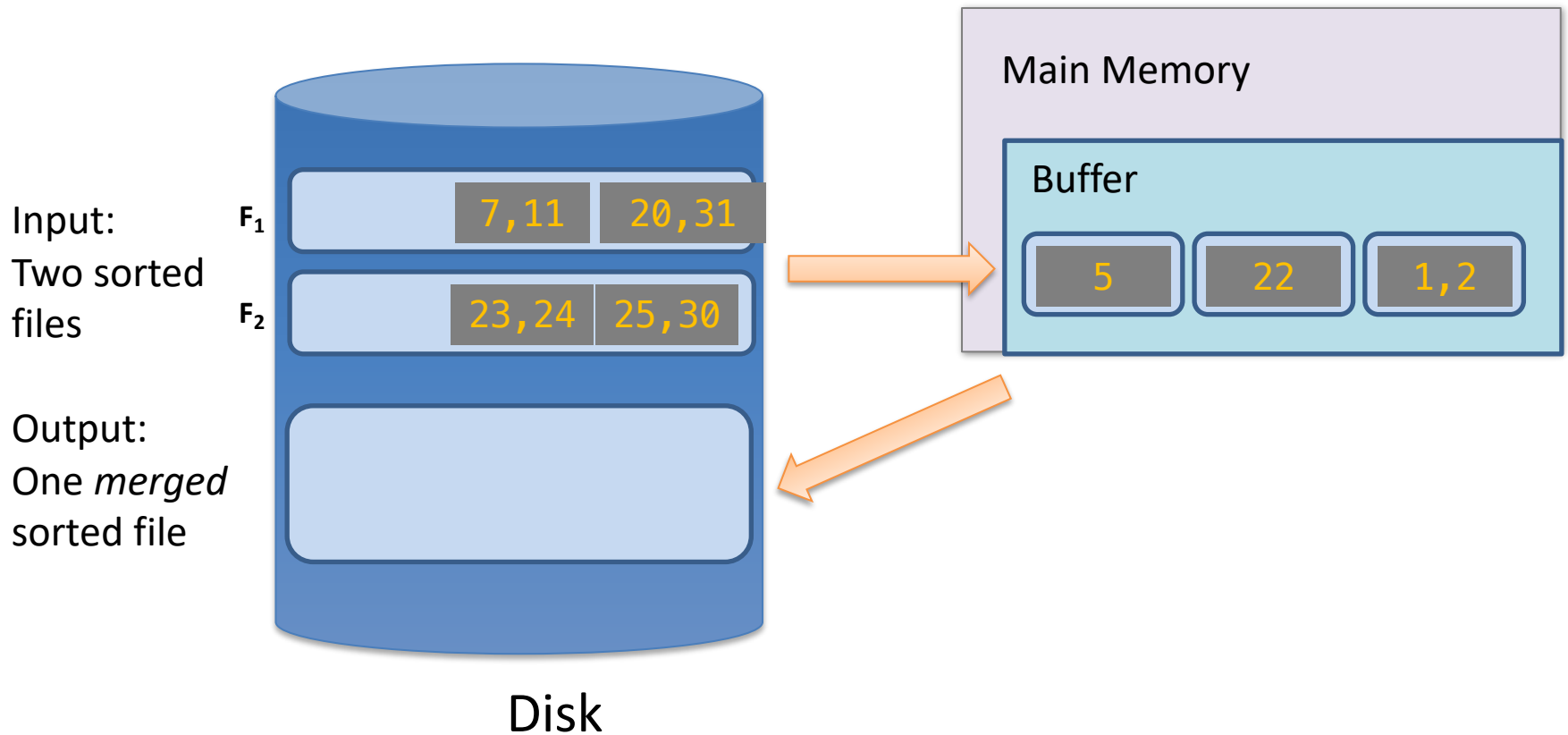
External Merge Algorithm



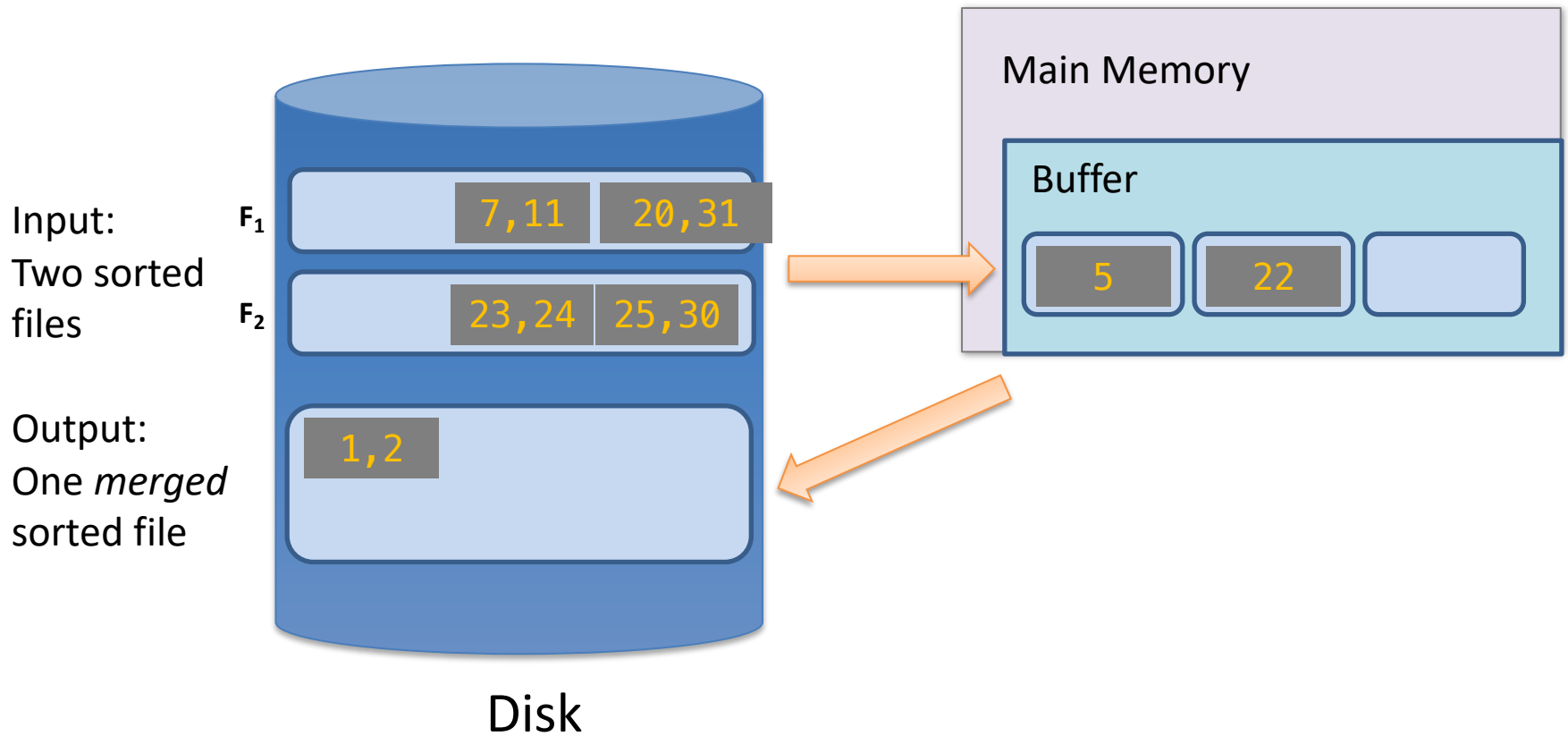
External Merge Algorithm



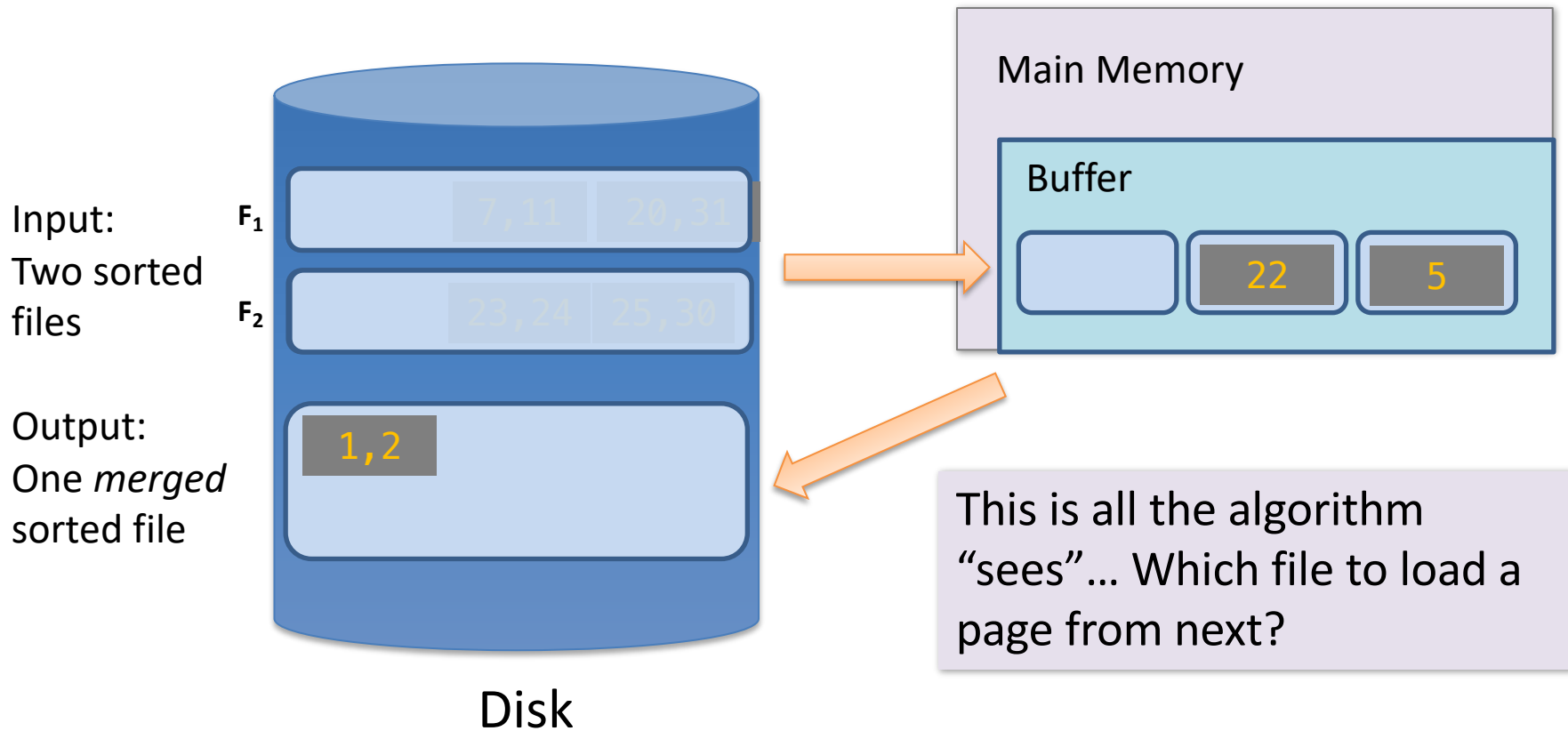
External Merge Algorithm



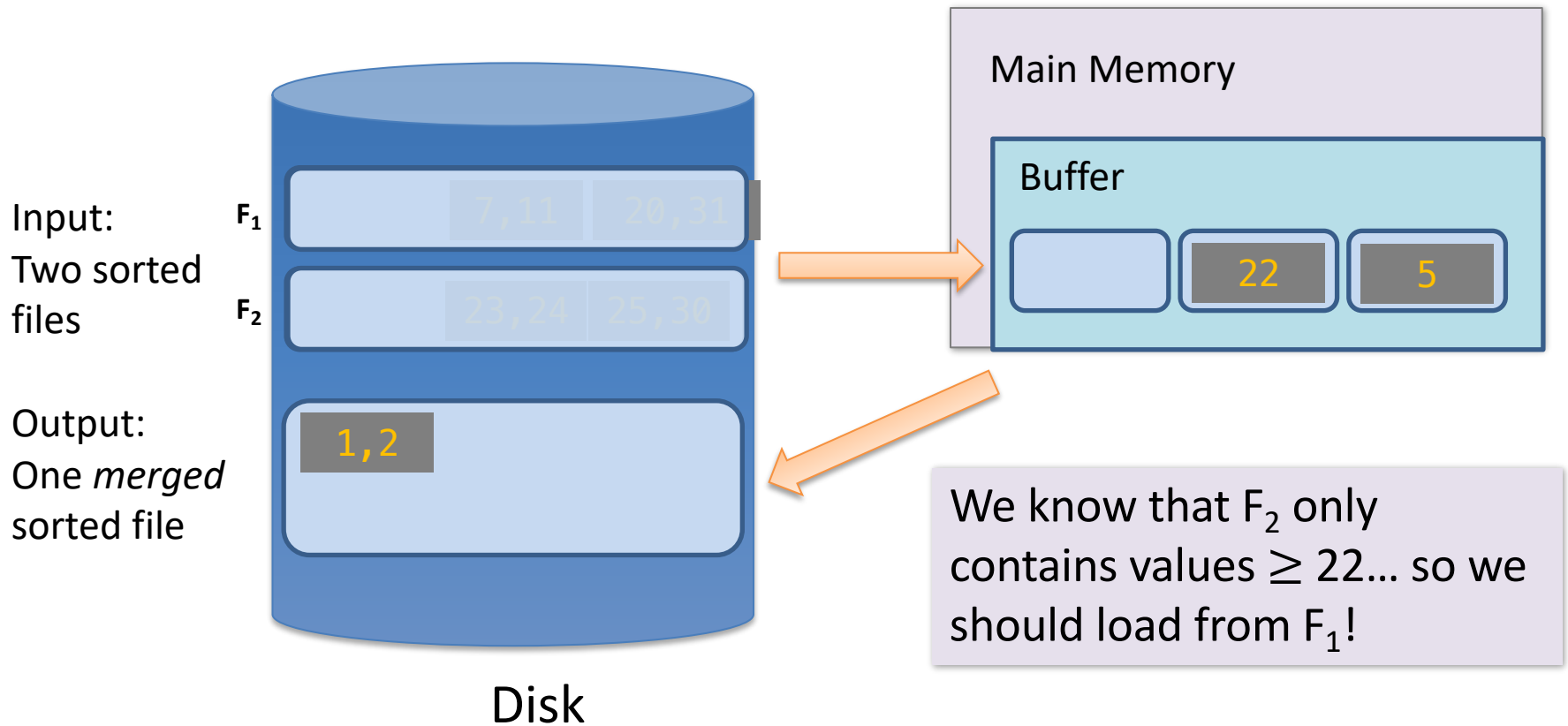
External Merge Algorithm



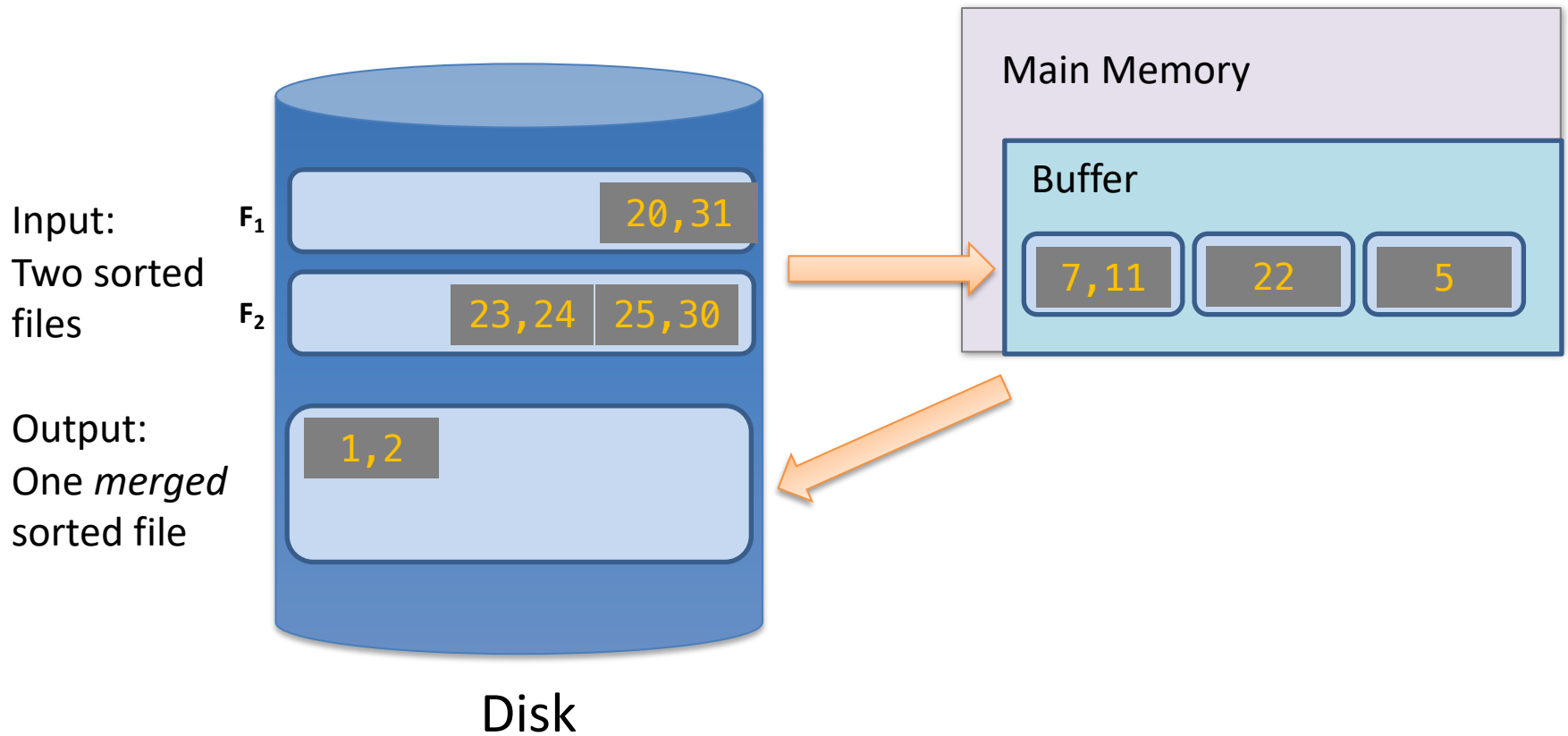
External Merge Algorithm



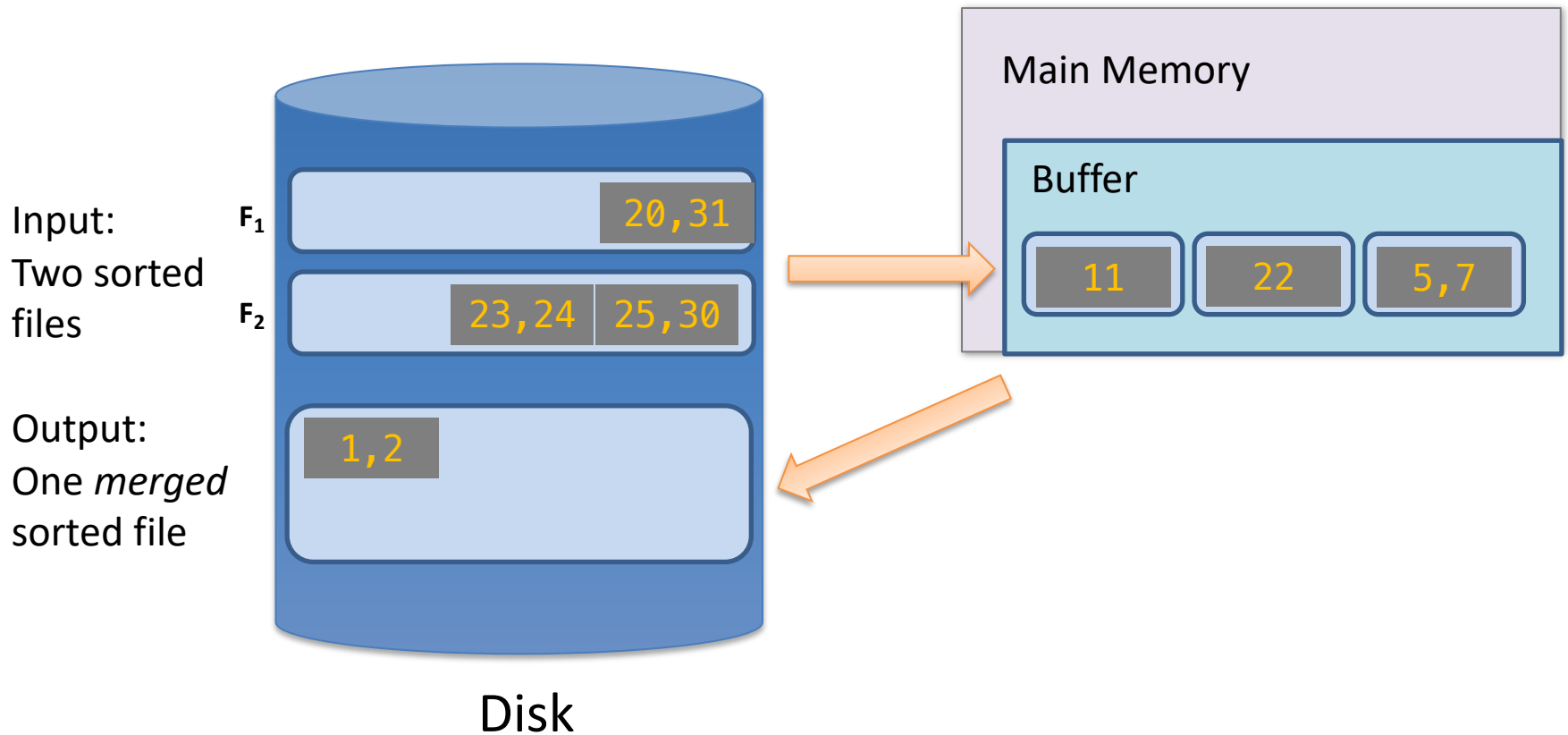
External Merge Algorithm



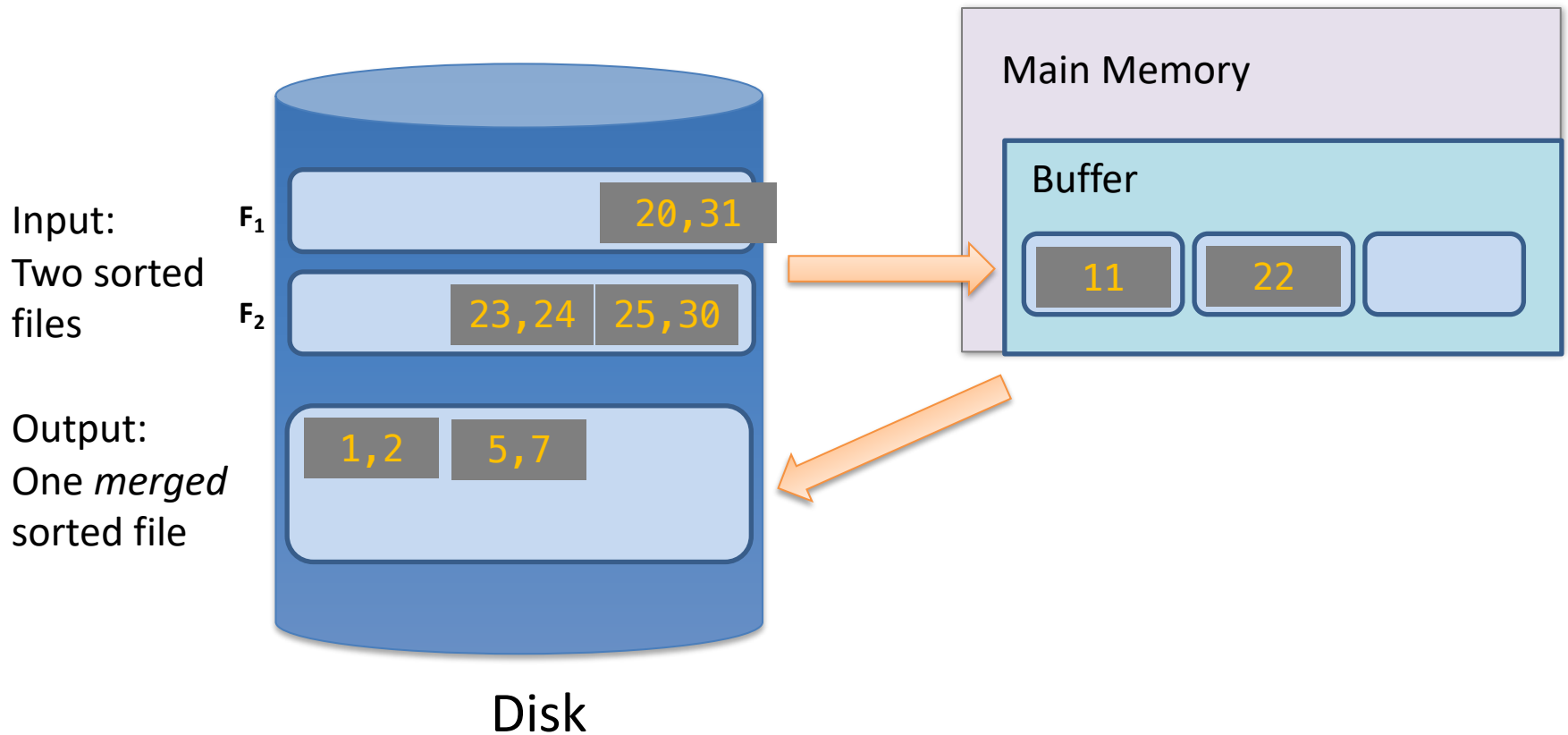
External Merge Algorithm



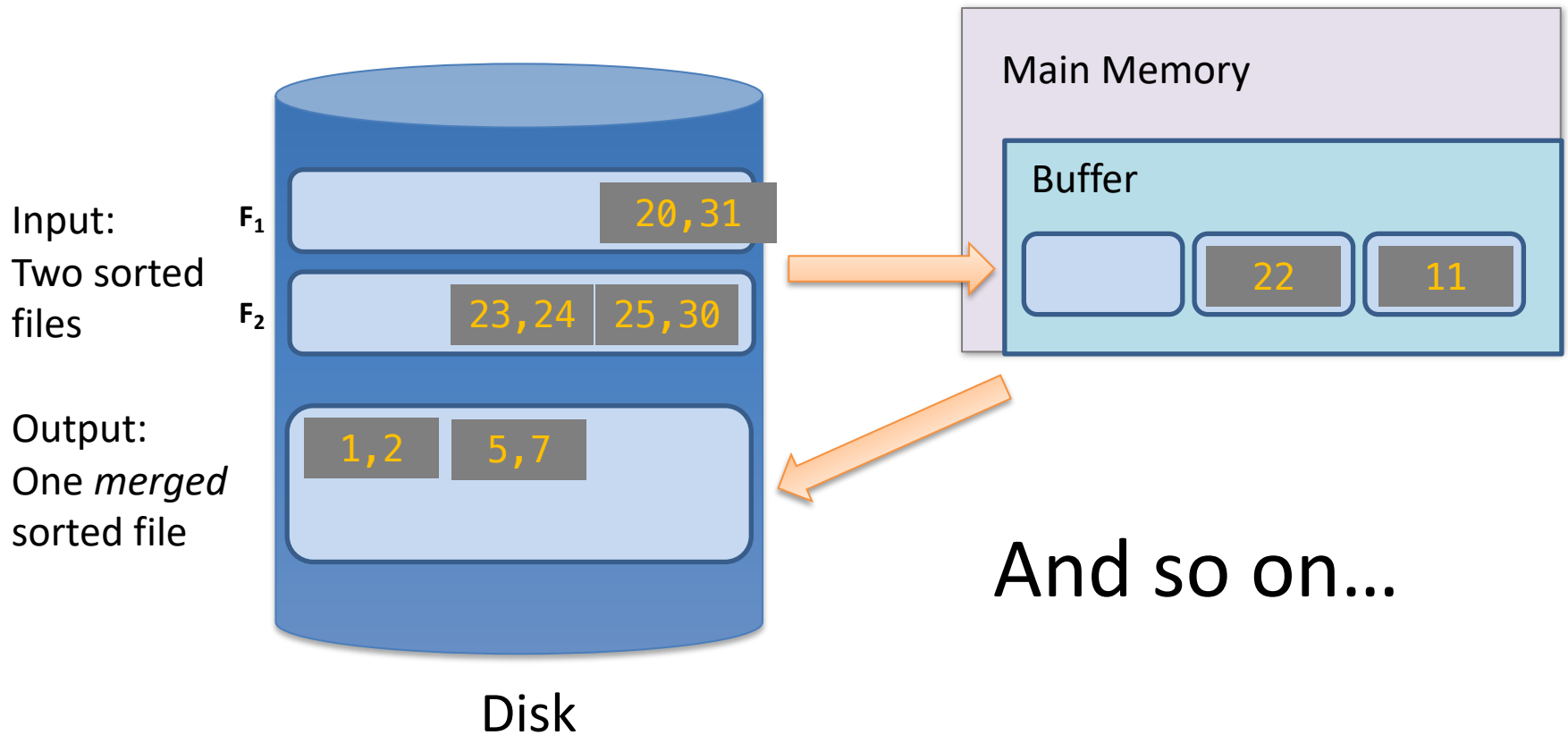
External Merge Algorithm



External Merge Algorithm



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

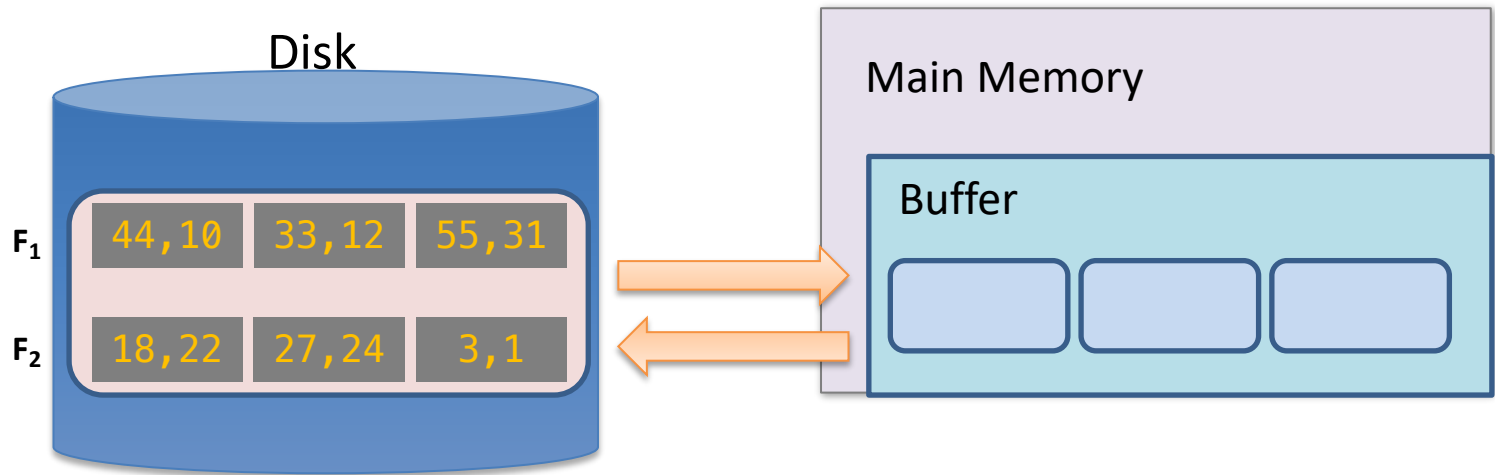
Each page is read once, written once

External Merge Sort Algorithm

Example:

- 3 Buffer pages
- 6-page file

Orange file
= unsorted



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

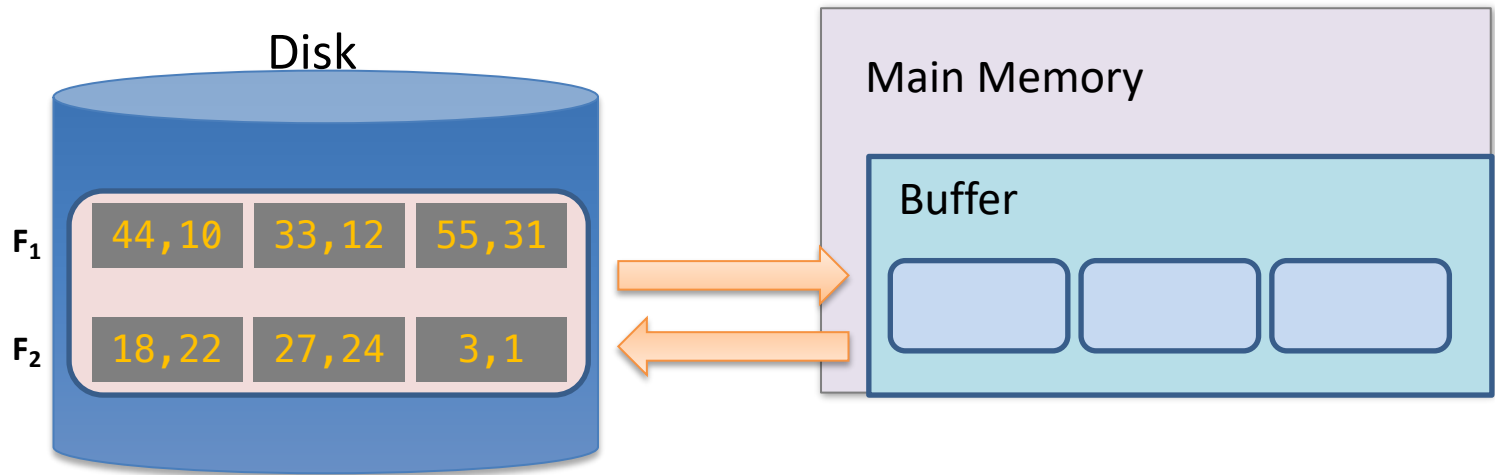
EXTERNAL MERGE SORT (BEFORE MERGE)

External Merge Sort Algorithm

Example:

- 3 Buffer pages
- 6-page file

Orange file
= unsorted



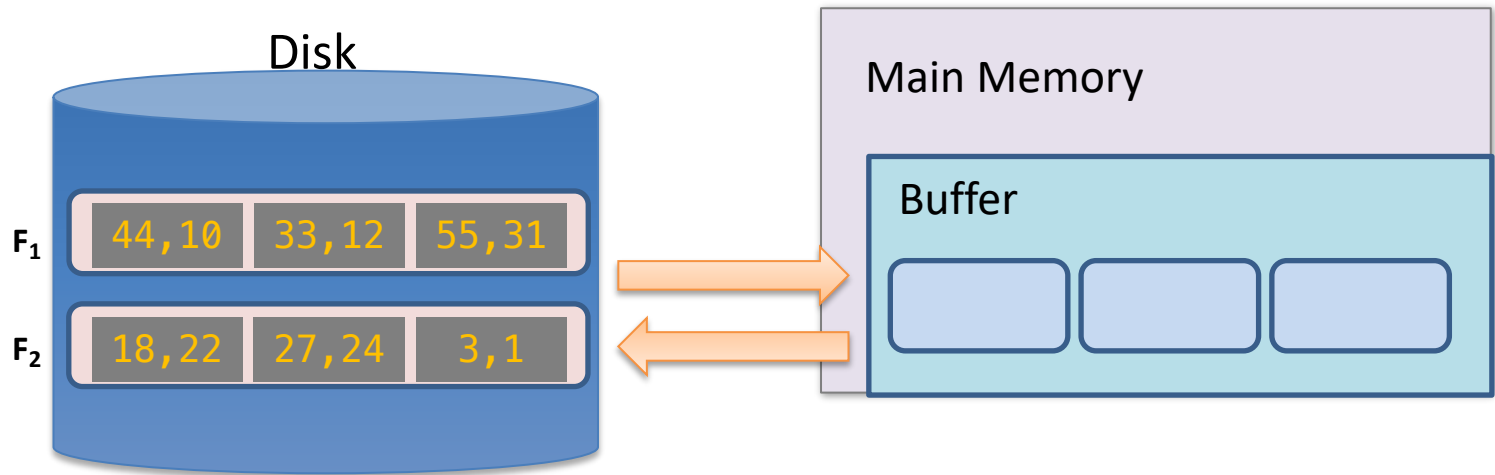
1. Split into chunks small enough to **sort in memory**

External Merge Sort Algorithm

Example:

- 3 Buffer pages
- 6-page file

Orange file
= unsorted



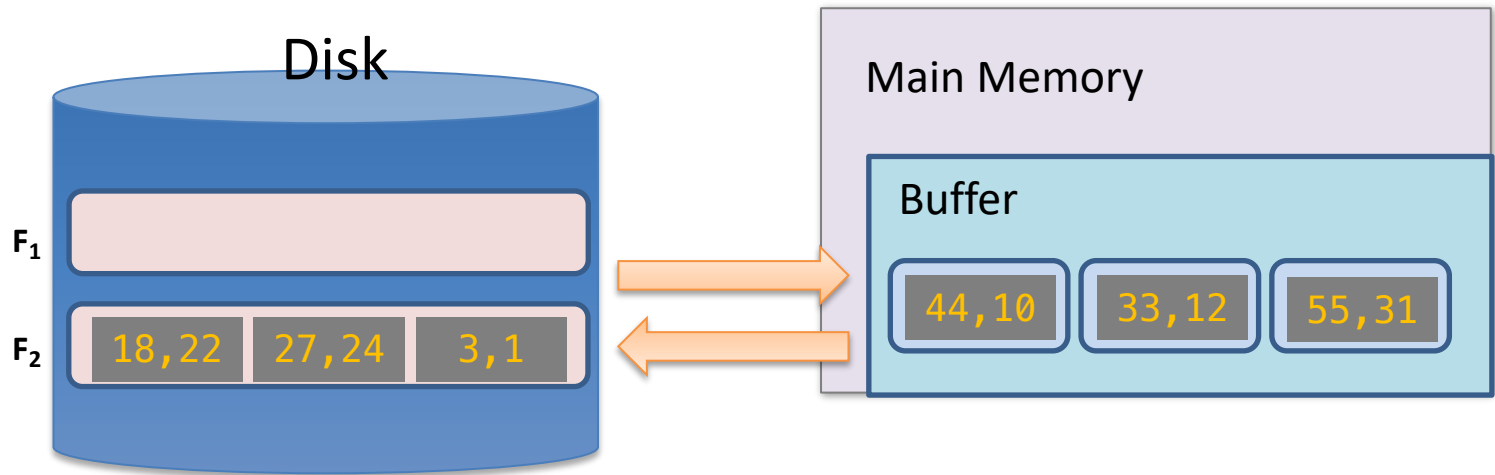
1. Split into chunks small enough to **sort in memory**

External Merge Sort Algorithm

Example:

- 3 Buffer pages
- 6-page file

Orange file
= unsorted



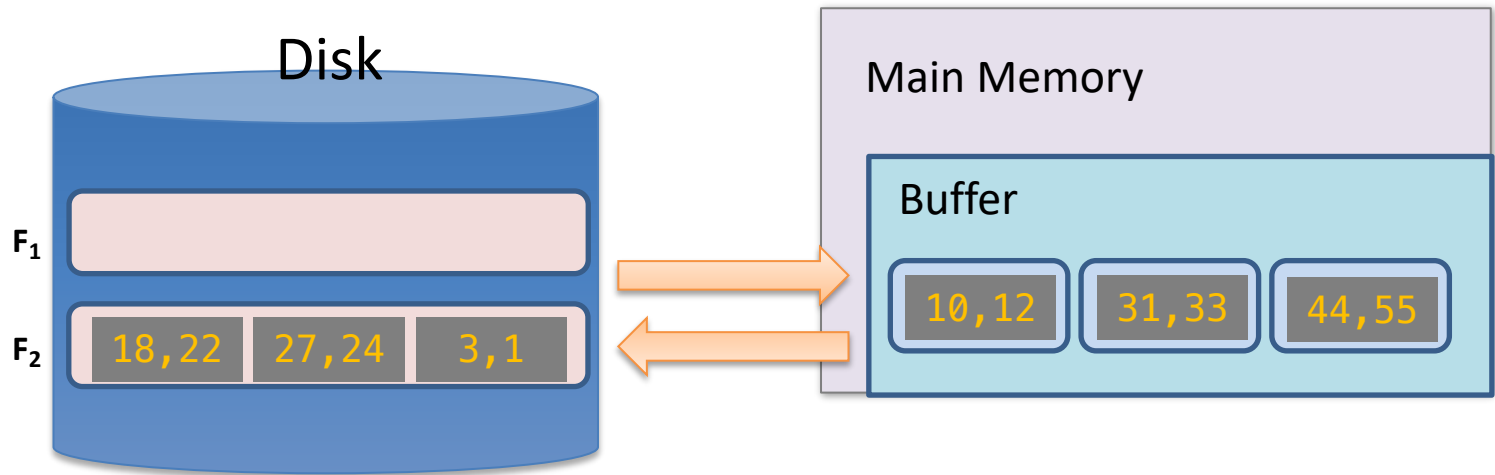
1. Split into chunks small enough to **sort in memory**

External Merge Sort Algorithm

Example:

- 3 Buffer pages
- 6-page file

Orange file
= unsorted



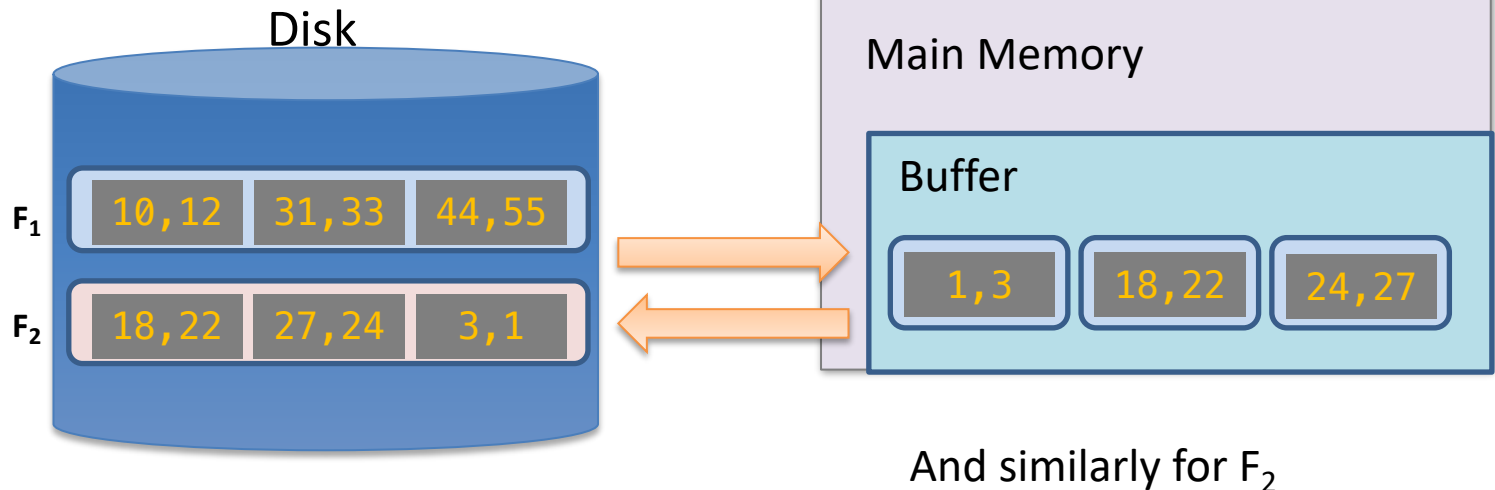
1. Split into chunks small enough to **sort in memory**

External Merge Sort Algorithm

Example:

- 3 Buffer pages
- 6-page file

Each sorted file is called a *run*

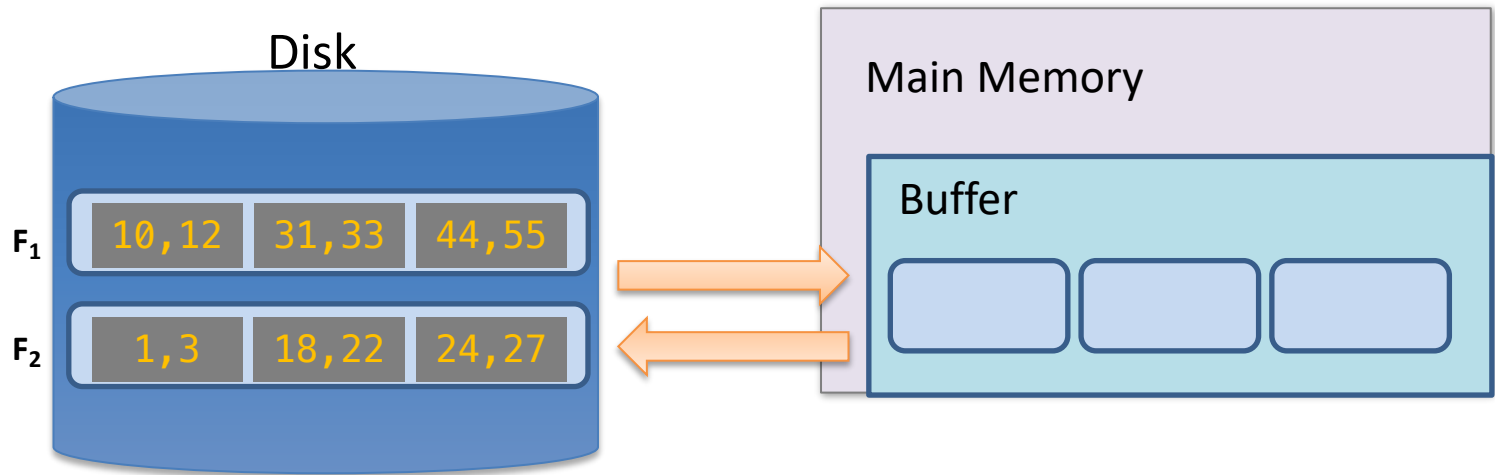


1. Split into chunks small enough to **sort in memory**

External Merge Sort Algorithm

Example:

- 3 Buffer pages
- 6-page file



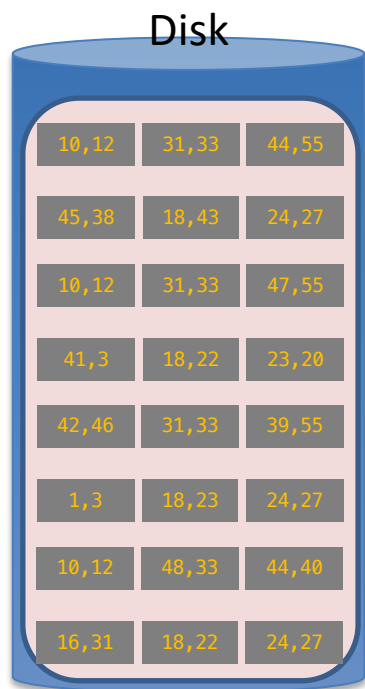
2. Now just run the **external merge** algorithm & we're done!

Calculating IO Cost

For 3 buffer pages, 6 page file:

1. Split into two 3-page files and sort in memory
 1. = 1 R + 1 W for each file = $2 \times (3 + 3) = 12$ IO operations
2. Merge each pair of sorted chunks *using the external merge algorithm*
 1. = $2 \times (3 + 3) = 12$ IO operations
(if this explanation makes more sense: Reading contents of all pages only once writing them only once --- so, $6 + 6 = 12$ IO)
3. Total cost = 24 IO

Running External Merge Sort on Larger Files

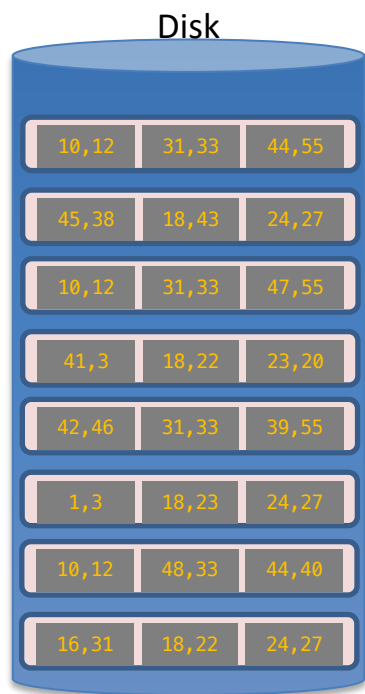


Assume we still only have 3 buffer pages (*Buffer not pictured*)

Disk

10, 12	31, 33	44, 55
45, 38	18, 43	24, 27
10, 12	31, 33	47, 55
41, 3	18, 22	23, 20
42, 46	31, 33	39, 55
1, 3	18, 23	24, 27
10, 12	48, 33	44, 40
16, 31	18, 22	24, 27

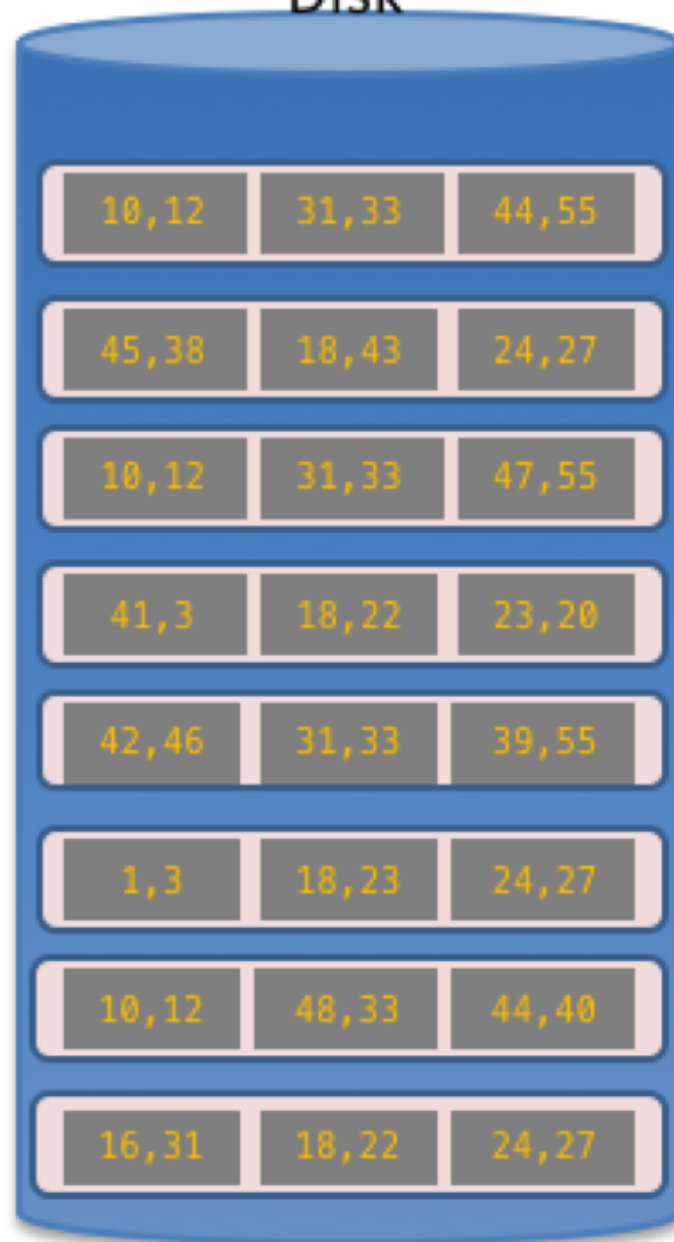
Running External Merge Sort on Larger Files



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

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

Disk



Running External Merge Sort on Larger Files

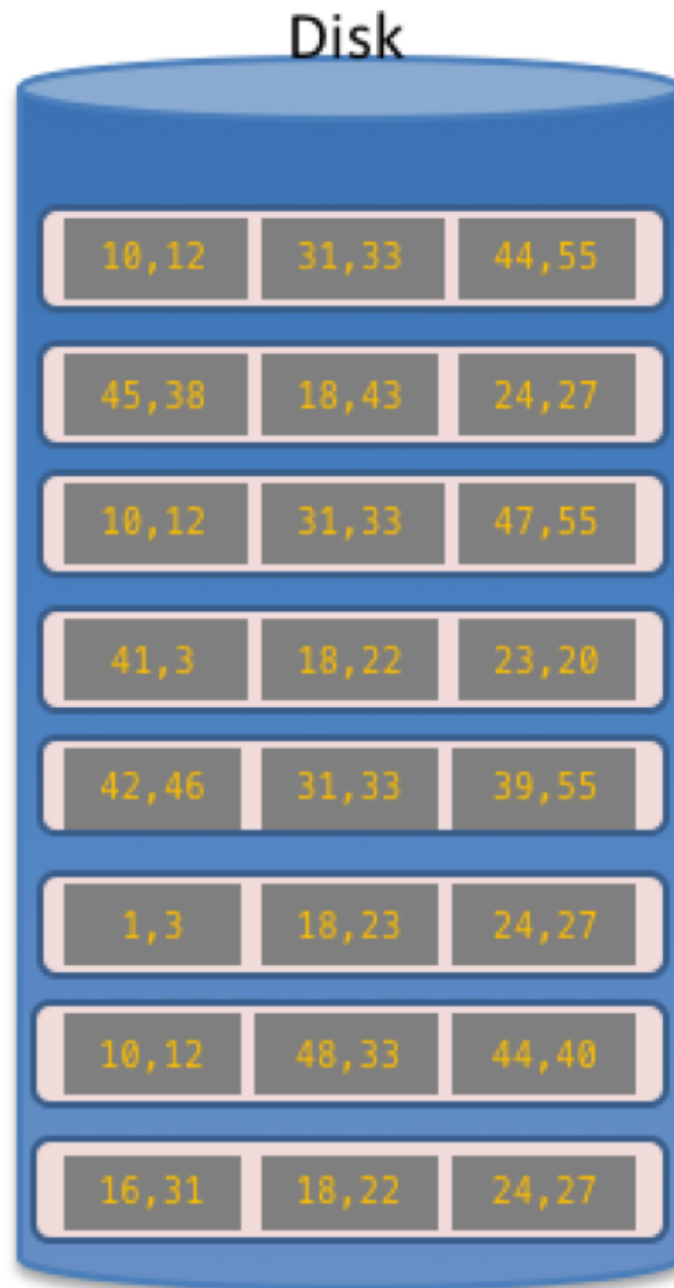


1. Split into files small enough to sort in buffer... and sort

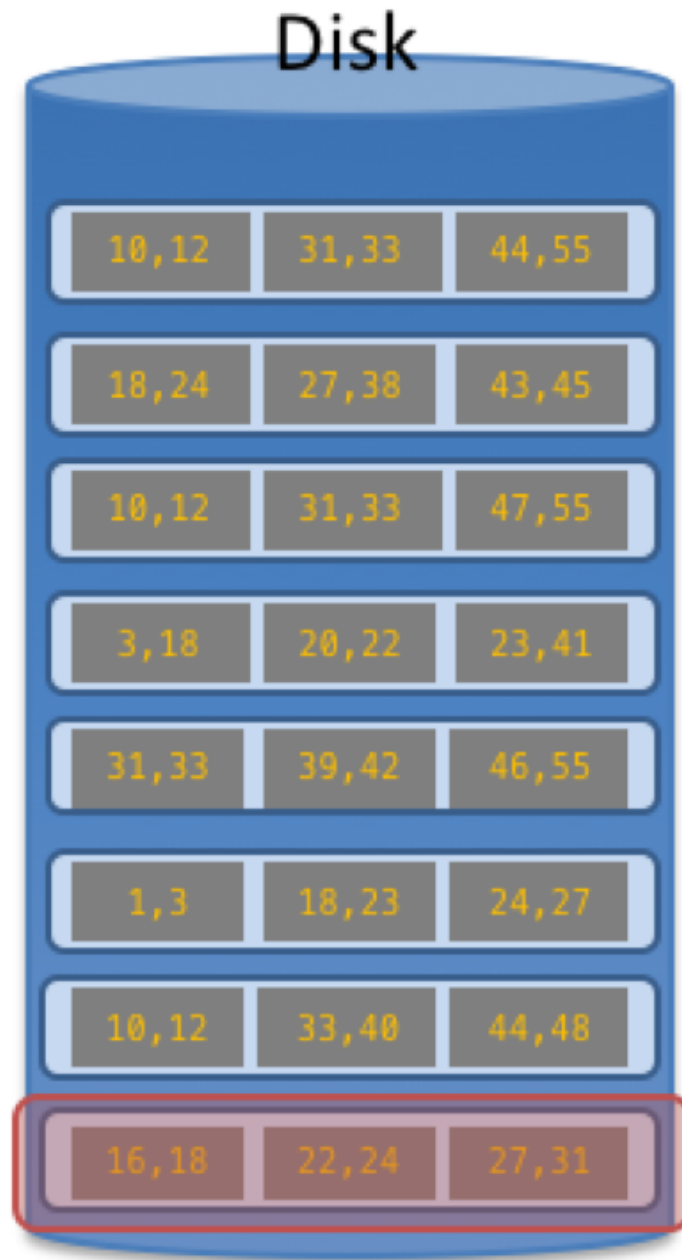
Call each of these sorted files a ***run***

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

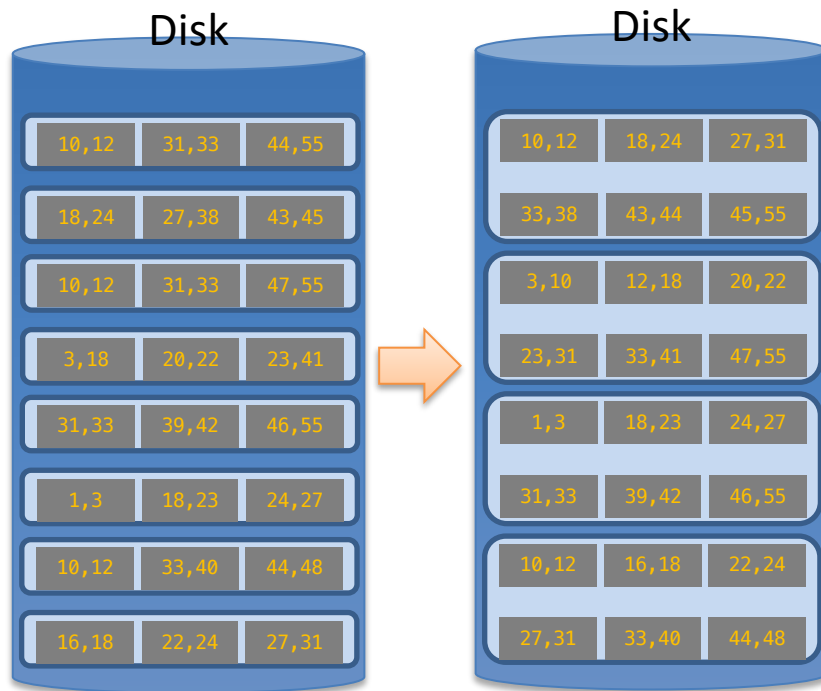
From



To



Running External Merge Sort on Larger Files



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

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

Disk

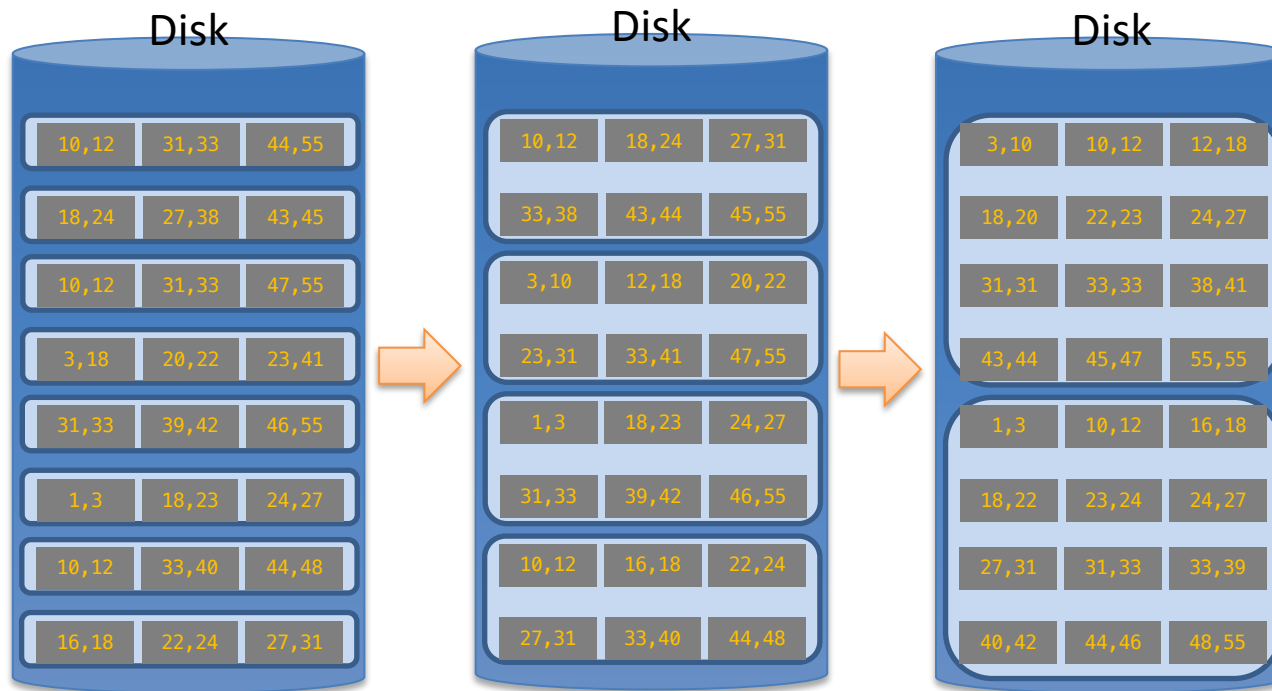
10,12	31,33	44,55
18,24	27,38	43,45
10,12	31,33	47,55
3,18	20,22	23,41
31,33	39,42	46,55
1,3	18,23	24,27
10,12	33,40	44,48
16,18	22,24	27,31



Disk

10,12	18,24	27,31
33,38	43,44	45,55
3,10	12,18	20,22
23,31	33,41	47,55
1,3	18,23	24,27
31,33	39,42	46,55
10,12	16,18	22,24
27,31	33,40	44,48

Running External Merge Sort on Larger Files




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

3. And repeat...

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

Disk



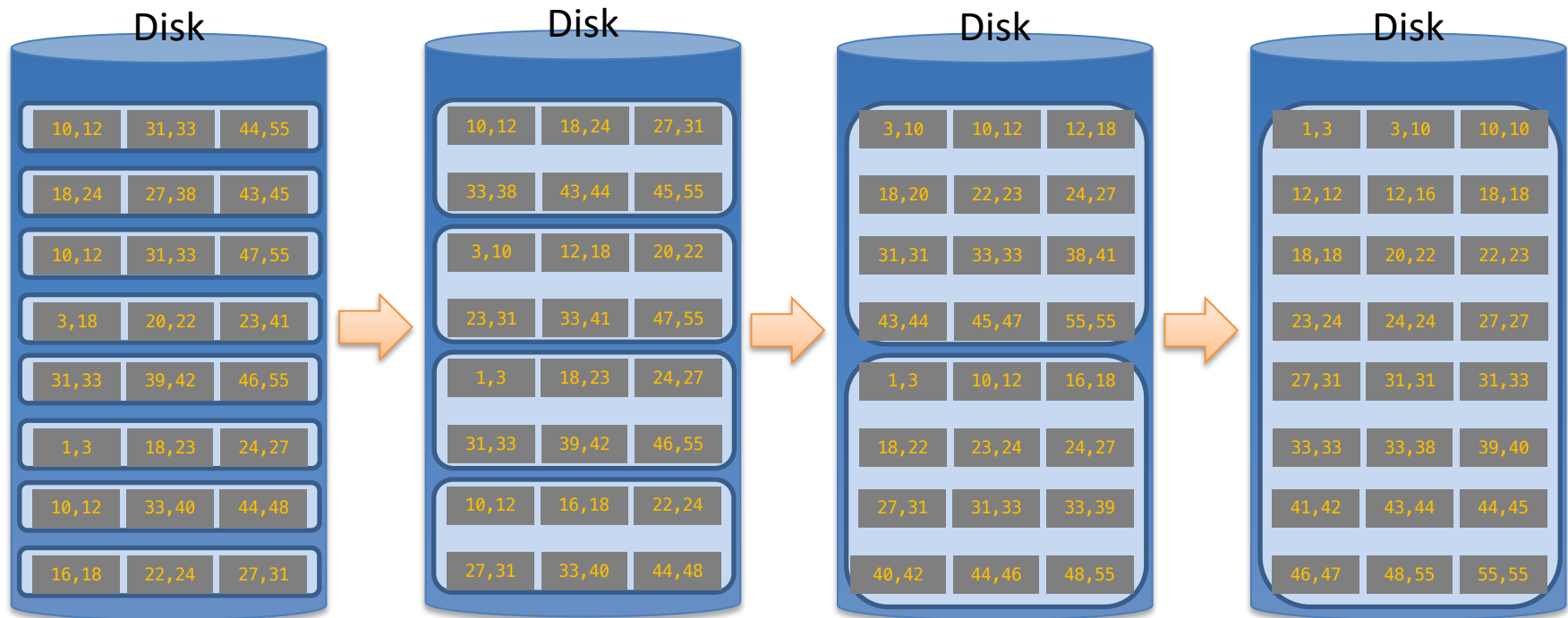
10,12	18,24	27,31
33,38	43,44	45,55
3,10	12,18	20,22
23,31	33,41	47,55
1,3	18,23	24,27
31,33	39,42	46,55
10,12	16,18	22,24
27,31	33,40	44,48

Disk




3,10	10,12	12,18
18,20	22,23	24,27
31,31	33,33	38,41
43,44	45,47	55,55
1,3	10,12	16,18
18,22	23,24	24,27
27,31	31,33	33,39
40,42	44,46	48,55

Running External Merge Sort on Larger Files



4. And repeat!

Disk



3, 10	10, 12	12, 18
18, 20	22, 23	24, 27
31, 31	33, 33	38, 41
43, 44	45, 47	55, 55
1, 3	10, 12	16, 18
18, 22	23, 24	24, 27
27, 31	31, 33	33, 39
40, 42	44, 46	48, 55



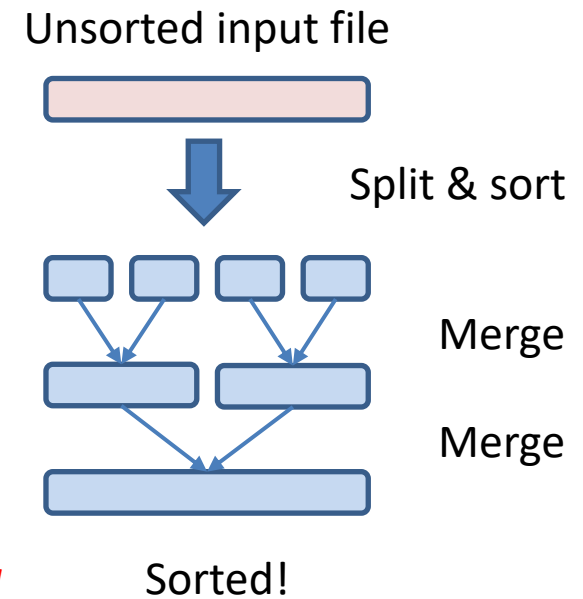
Disk

1, 3	3, 10	10, 10
12, 12	12, 16	18, 18
18, 18	20, 22	22, 23
23, 24	24, 24	27, 27
27, 31	31, 31	31, 33
33, 33	33, 38	39, 40
41, 42	43, 44	44, 45
46, 47	48, 55	55, 55

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 & writing out all the pages = **$2N$ IO**



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

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

IO Cost:

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



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

Starting with runs
of length 1

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

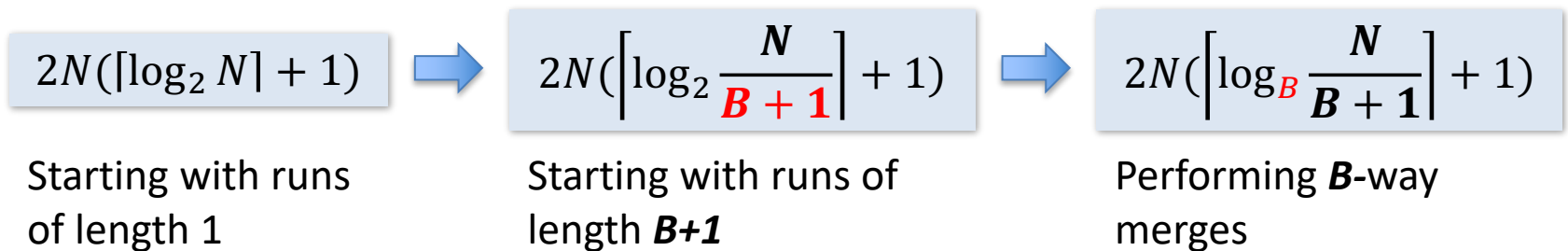
Using B+1 buffer pages to reduce # of passes

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)!

IO Cost:



What Next!

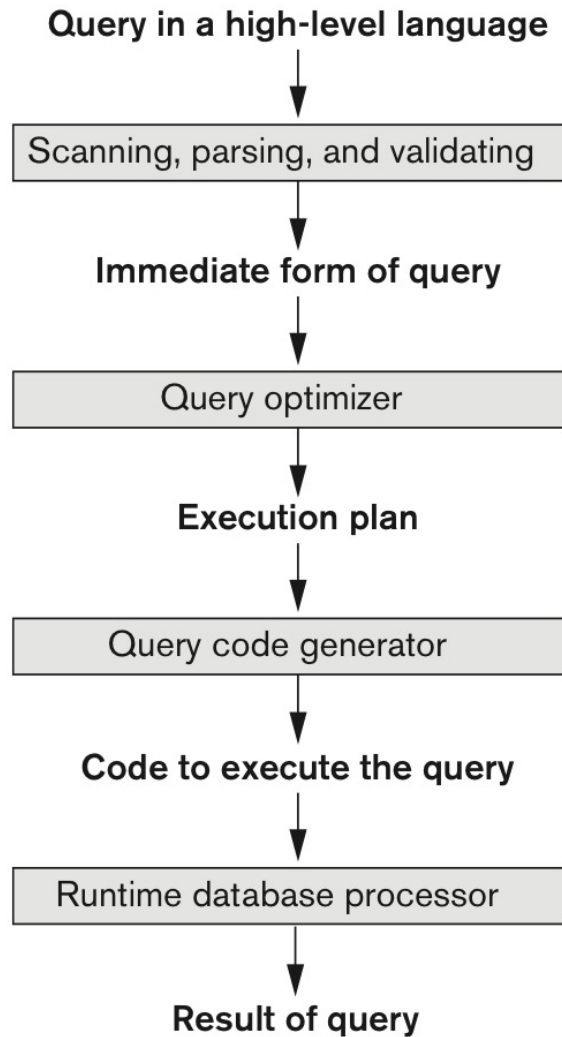
- Query Processing (Chapter 18)
- Query Optimization (Chapter 19)

QUERY PROCESSING

Steps in Query Processing

- Scanning
- Parsing
- Validation
- Query Tree Creation
- Query Optimization (Query planning)
- Code generation (to execute the plan)
- Running the query code

Steps in Query Processing



Code can be:

Executed directly (interpreted mode)

Stored and executed later whenever needed (compiled mode)

SQL Queries

- SQL Queries are decomposed into **Query blocks**:
 - Select...From...Where...Group By...Having
- Translate **Query blocks** into **Relational Algebraic expression**
- Remember, SQL includes aggregate operators:
 - MIN, MAX, SUM, COUNT etc.
 - Part of the extended algebra
 - Let's go back to Chapter 8 (Section 8.4.2)

Aggregate Functions and Grouping (Relational Algebra)

- Aggregate function: \mathcal{F}
- $\langle \text{grouping attributes} \rangle \mathcal{F} \langle \text{function list} \rangle (R)$

Dno \mathcal{F} COUNT Ssn, AVERAGE Salary(EMPLOYEE).

Dno	Count_ssn	Average_salary
5	4	33250
4	3	31000
1	1	55000

Semijoin (\bowtie)

- $R \bowtie S = \Pi_{A_1, \dots, A_n} (R \bowtie S)$
- Where A_1, \dots, A_n are the attributes in R
- Example:
 - Employee \bowtie Dependents

Students(sid, sname, gpa)
People(ssn, pname, address)

SQL:

```
SELECT DISTINCT  
  sid, sname, gpa  
FROM  
  Students, People  
WHERE  
  sname = pname;
```

OR

```
SELECT DISTINCT  
  sid, sname, gpa  
FROM  
  Students  
WHERE  
  sname IN  
  (SELECT pname FROM People);
```



RA:

Students \bowtie People

Algorithm for External Sorting

- We have already covered

Algorithm for Select Operation

- Read Section 18.3 (18.3.1 , 18.3.2, 18.3.3, 18.3.4)
- Mostly covers searching:
 - 1. Linear Search
 - 2. Binary Search
 - 3. Indexing
 - 4. Hashing
 - 5. B+ Tree
- (Skip bitmap index and functional index if you want)

Algorithm for Join Operation

- The most time consuming operation

What you will learn about in this section

1. Nested Loop Join (NLJ)
2. Block Nested Loop Join (BNLJ)
3. Index Nested Loop Join (INLJ)
4. Sorted-Merge Join
5. Hash Join

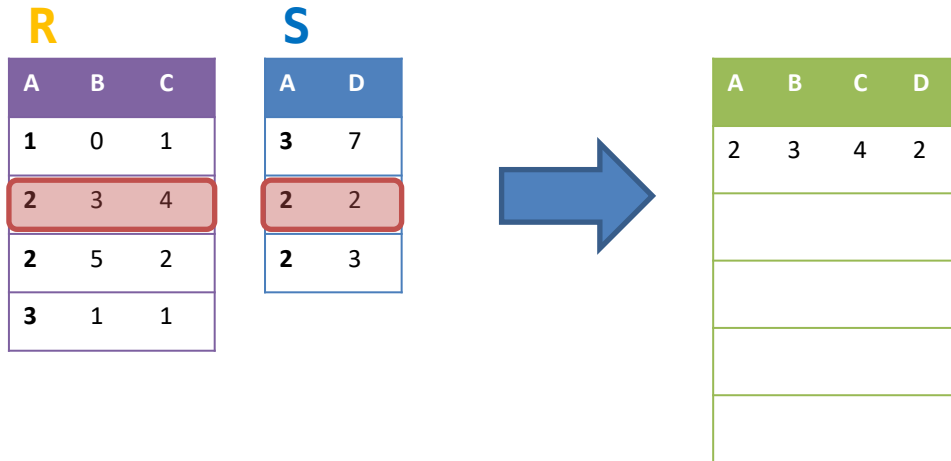
RECAP: Joins

Joins: Example

$R \bowtie S$

```
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 $r.A = s.A$



Joins: Example

$R \bowtie S$

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

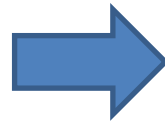
Example: Returns all pairs of tuples $r \in R, s \in S$ such that $r.A = s.A$

R

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

S

A	D
3	7
2	2
2	3



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

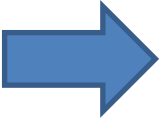
Joins: Example

$R \bowtie S$

```
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			S	
A	B	C	A	D
1	0	1	3	7
2	3	4	2	2
2	5	2	2	3
3	1	1		



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

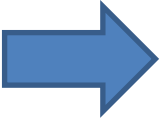
Joins: Example

$R \bowtie S$

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

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

R			S	
A	B	C	A	D
1	0	1	3	7
2	3	4	2	2
2	5	2	2	3
3	1	1		



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

Joins: Example

$R \bowtie S$

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

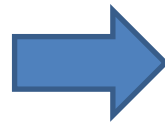
Example: Returns all pairs of tuples $r \in R, s \in S$ such that $r.A = s.A$

R

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

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

$R \bowtie S$

```
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 $r.A = s.A$

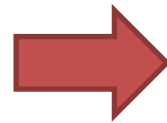
R

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

×

S

A	D
3	7
2	2
2	3



Cross
Product

...



Filter by
conditions
($r.A = s.A$)

A	B	C	D
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 $\mathbf{R} \bowtie \mathbf{S}$ to mean *join R and S by returning all tuple pairs where **all shared attributes** are equal*
- We write $\mathbf{R} \bowtie \mathbf{S} \text{ on } \mathbf{A}$ 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 again 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
- Note also that we omit ceilings in calculations... good exercise to put back in!

Recall that we read / write entire pages with disk IO

Nested Loop Join (NLJ)

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

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)
```

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)

```
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

Have to read ***all of S*** from disk for ***every tuple 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  $(r, s)$ 
```

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)

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

Cost:

$$P(R) + T(R) * P(S) + \text{OUT}$$

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)

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

Cost:

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

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!

Block Nested Loop Join (BNLJ)

Block Nested Loop Join (BNLJ)

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

Cost:

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  $(r, s)$ 
```

$P(R)$

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  $R \bowtie S$  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

Cost:

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

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

Cost:

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

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**

```
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  $(r,s)$ 
```

BNLJ can also handle non-equality constraints

Block Nested Loop Join (BNLJ)

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

Cost:

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

```
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  $(r, s)$ 
```

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

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

$$P(R) + T(R) * P(S) + \text{OUT}$$



BNLJ

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

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

BNLJ vs. NLJ: Benefits of IO Aware

- Example:
 - R: 500 pages
 - S: 1000 pages
 - 100 tuples / page
 - We have 12 pages of memory ($B = 11$)
- NLJ: Cost = $500 + 50,000 * 1000 = 50 \text{ Million IOs} \approx \underline{140 \text{ hours}}$
- BNLJ: Cost = $500 + \frac{500 * 1000}{10} = 50 \text{ Thousand IOs} \approx \underline{0.14 \text{ hours}}$

*Ignoring OUT
here...*

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

Smarter than Cross-Products

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: $\text{NLJ } P(R) + \mathbf{T(R)P(S)} + \text{OUT}$

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

- Now we'll see some (nearly) linear joins:
 - $\sim O(P(R) + P(S) + \mathbf{OUT})$, 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 $R \bowtie S$ on A :

Given index idx on $S.A$:

```
for r in R:
    s in idx(r[A]):
        yield r,s
```

Cost:

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

where L is the IO cost to access all the distinct values in the index; assuming these fit on one page, $L \sim 3$ is good est.

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

Sort-Merge Join (SMJ)

What you will learn about in this section

1. Sort-Merge Join
2. “Backup” & Total Cost
3. Optimizations

Sort Merge Join (SMJ): Basic Procedure

To compute $R \bowtie S$ on A :

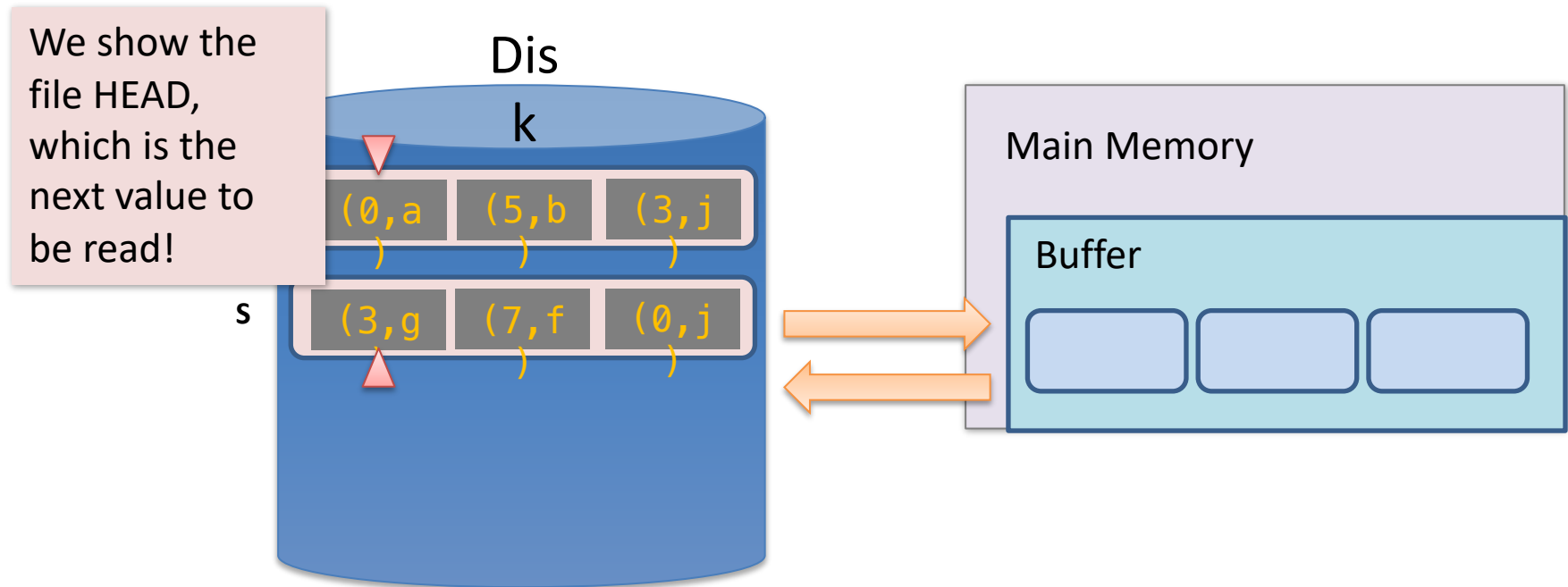
1. Sort R, S on A using ***external merge sort***
2. ***Scan*** sorted files and “merge”
3. *[May need to “backup”- see next subsection]*

Note that we are only considering equality join conditions here

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

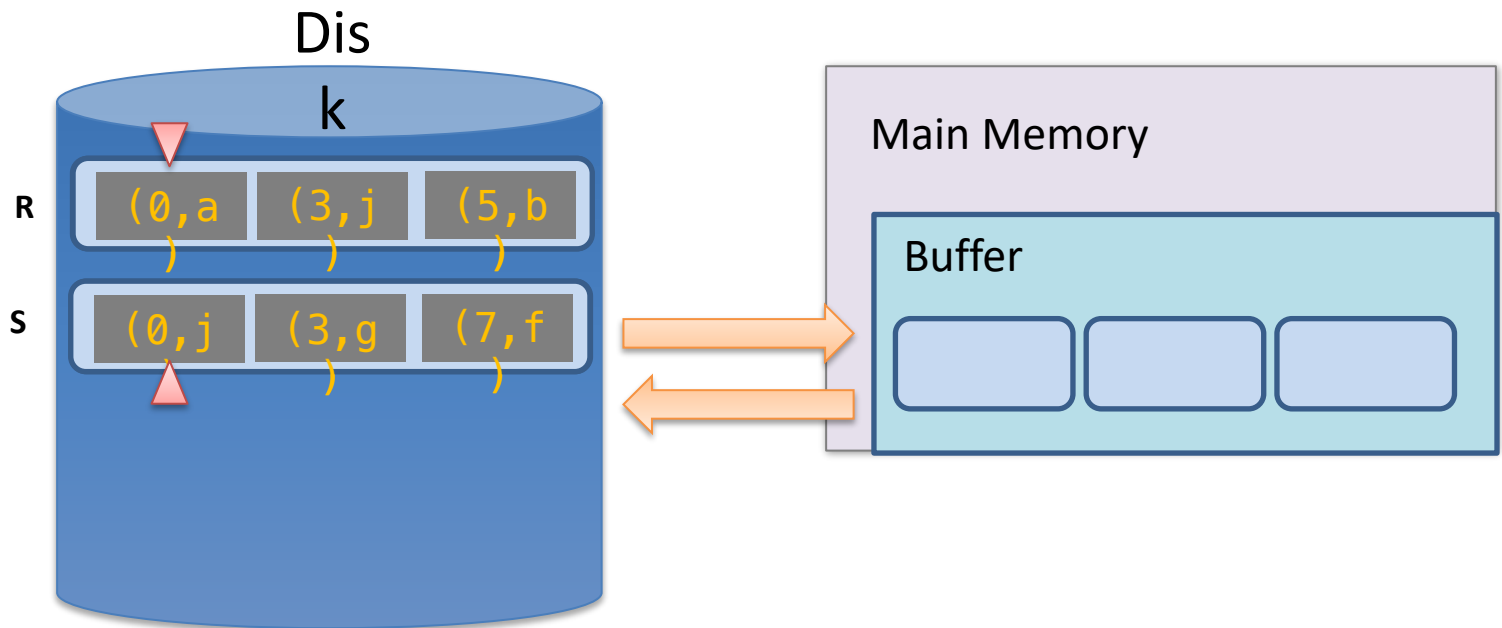
SMJ Example: $R \bowtie S$ on A with 3 page buffer

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



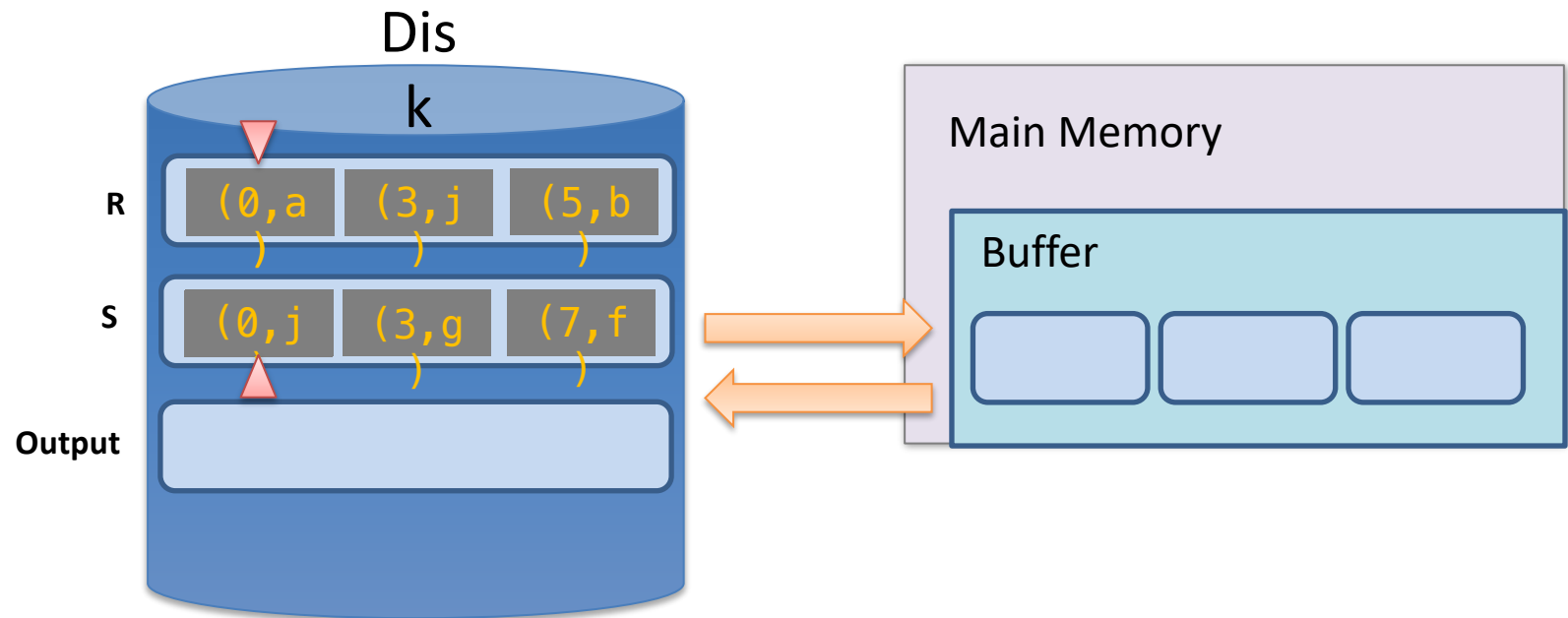
SMJ Example: $R \bowtie S$ on A with 3 page buffer

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



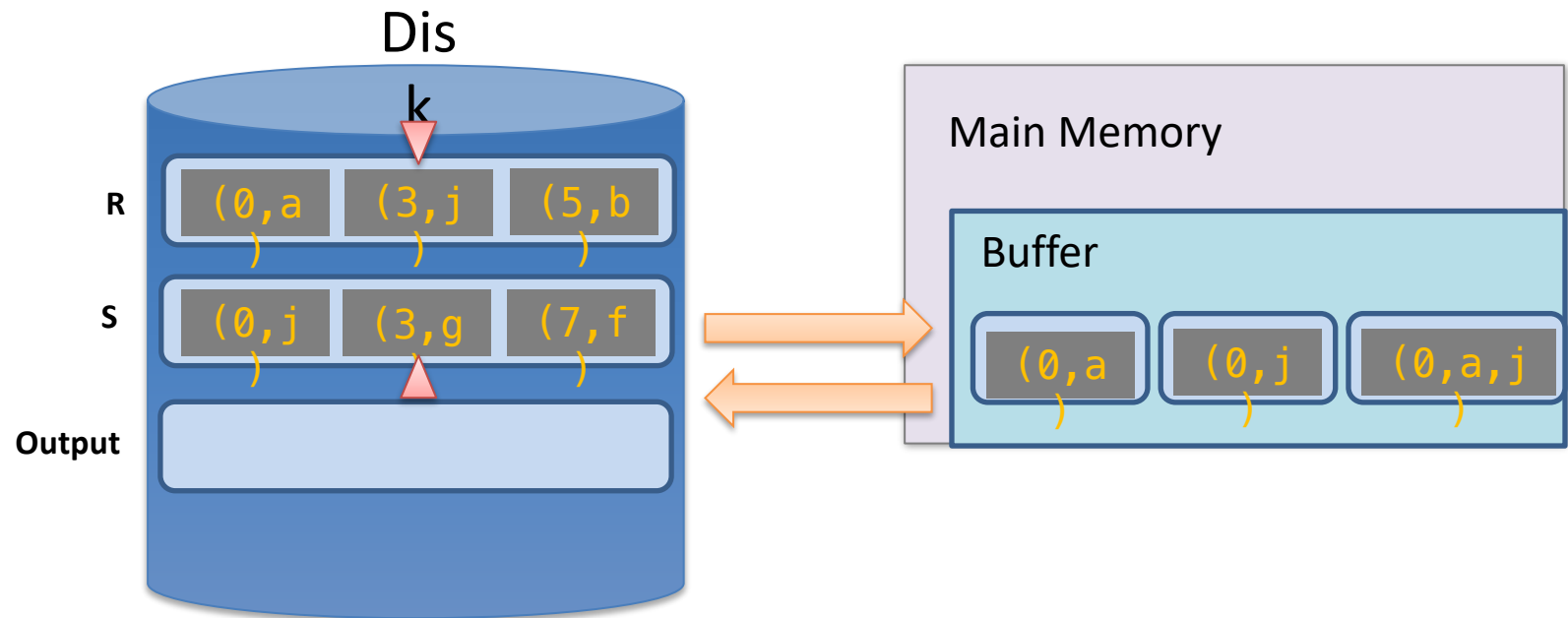
SMJ Example: $R \bowtie S$ on A with 3 page buffer

2. Scan and “merge” on join key!



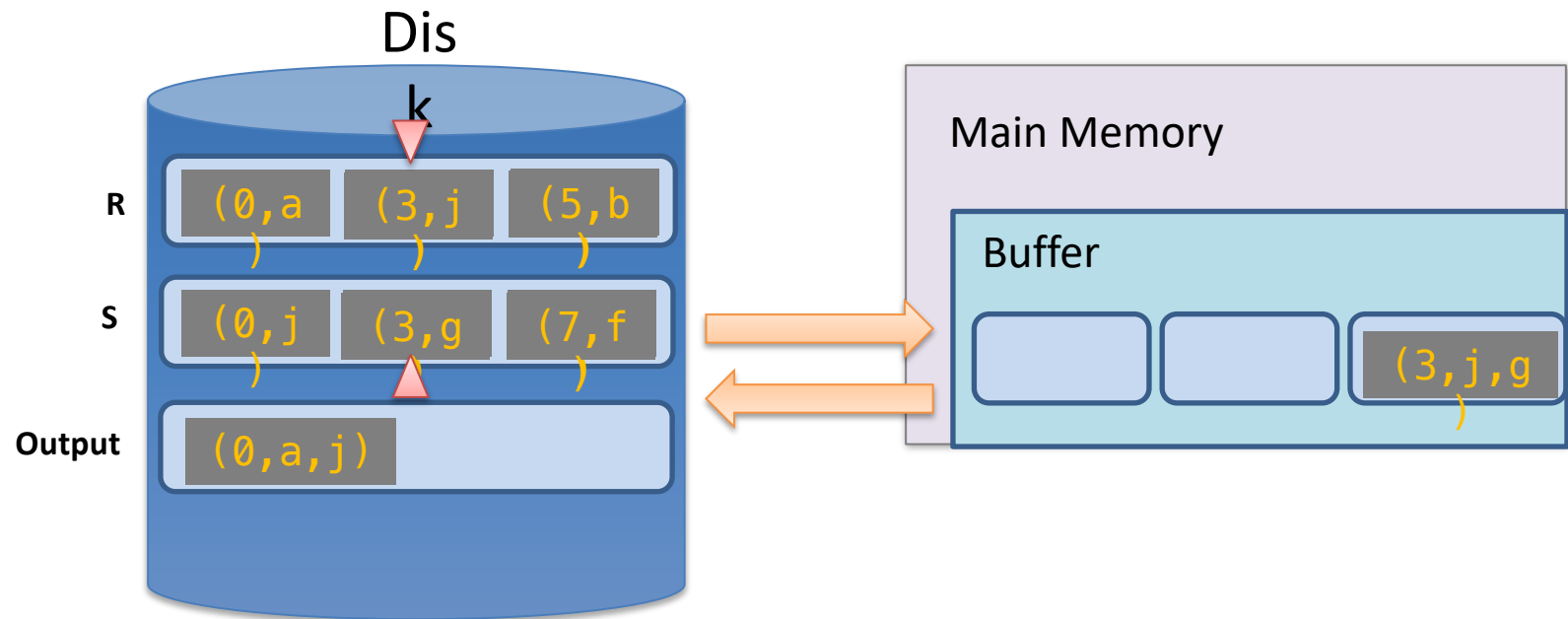
SMJ Example: $R \bowtie S$ on A with 3 page buffer

2. Scan and “merge” on join key!



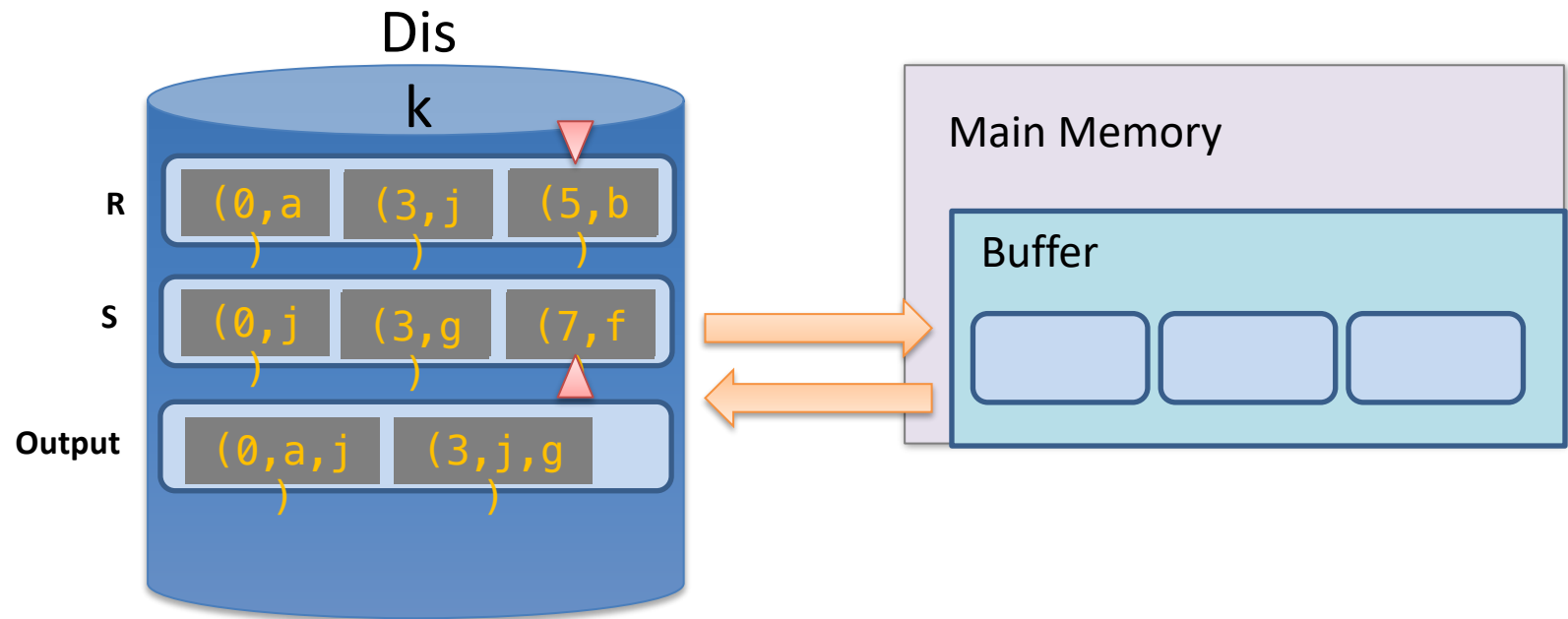
SMJ Example: $R \bowtie S$ on A with 3 page buffer

2. Scan and “merge” on join key!



SMJ Example: $R \bowtie S$ on A with 3 page buffer

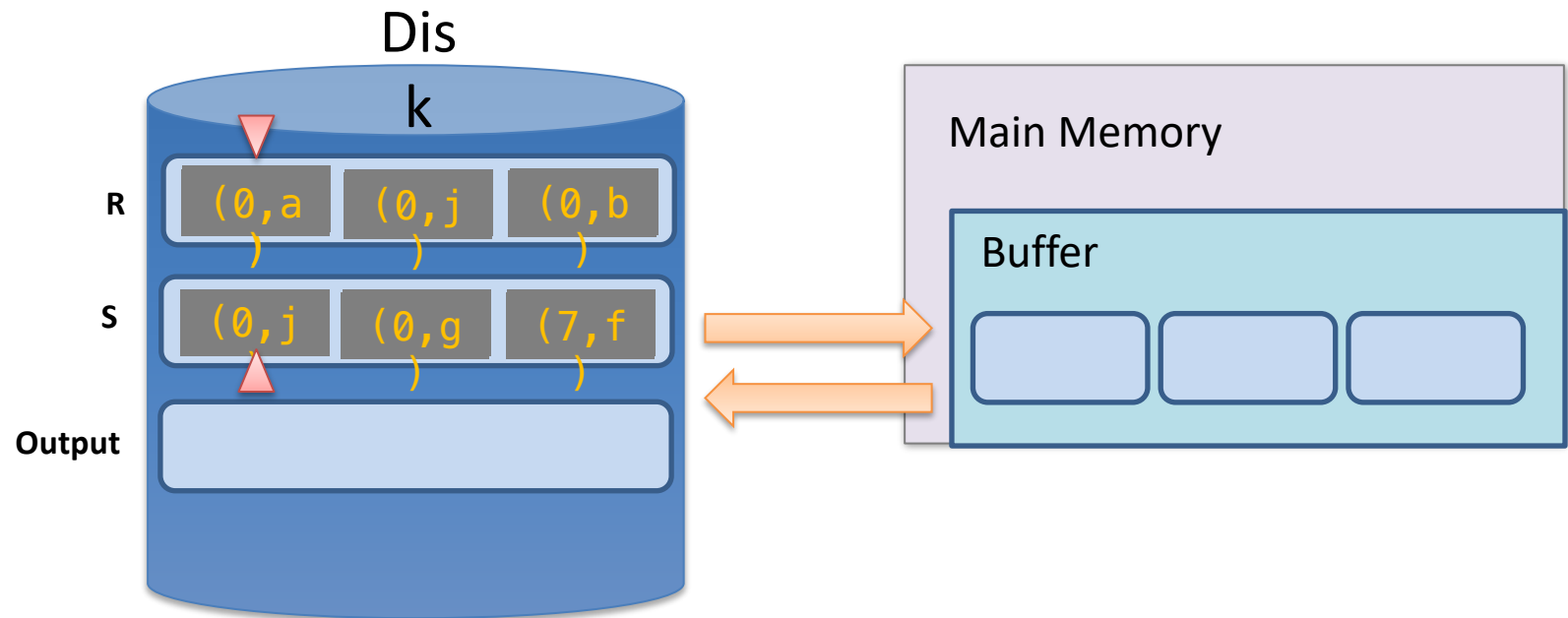
2. Done!



What happens with duplicate join keys?

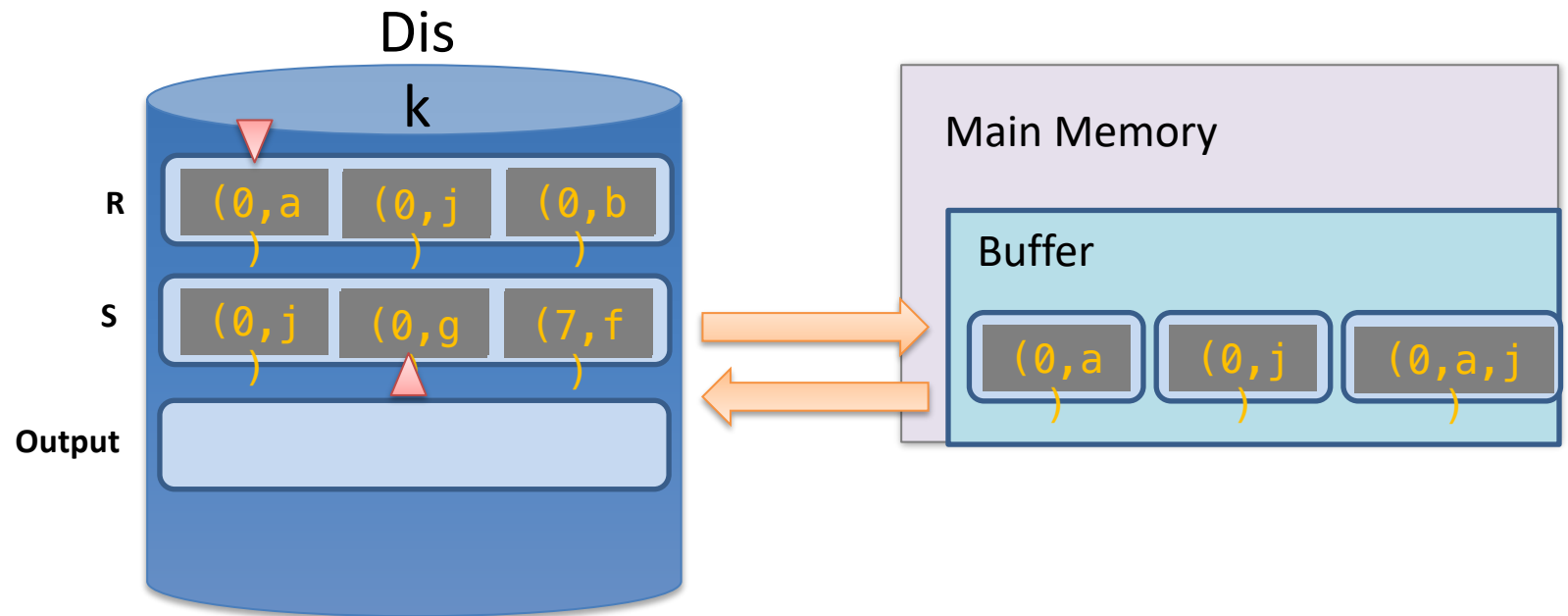
Multiple tuples with Same Join Key: “Backup”

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



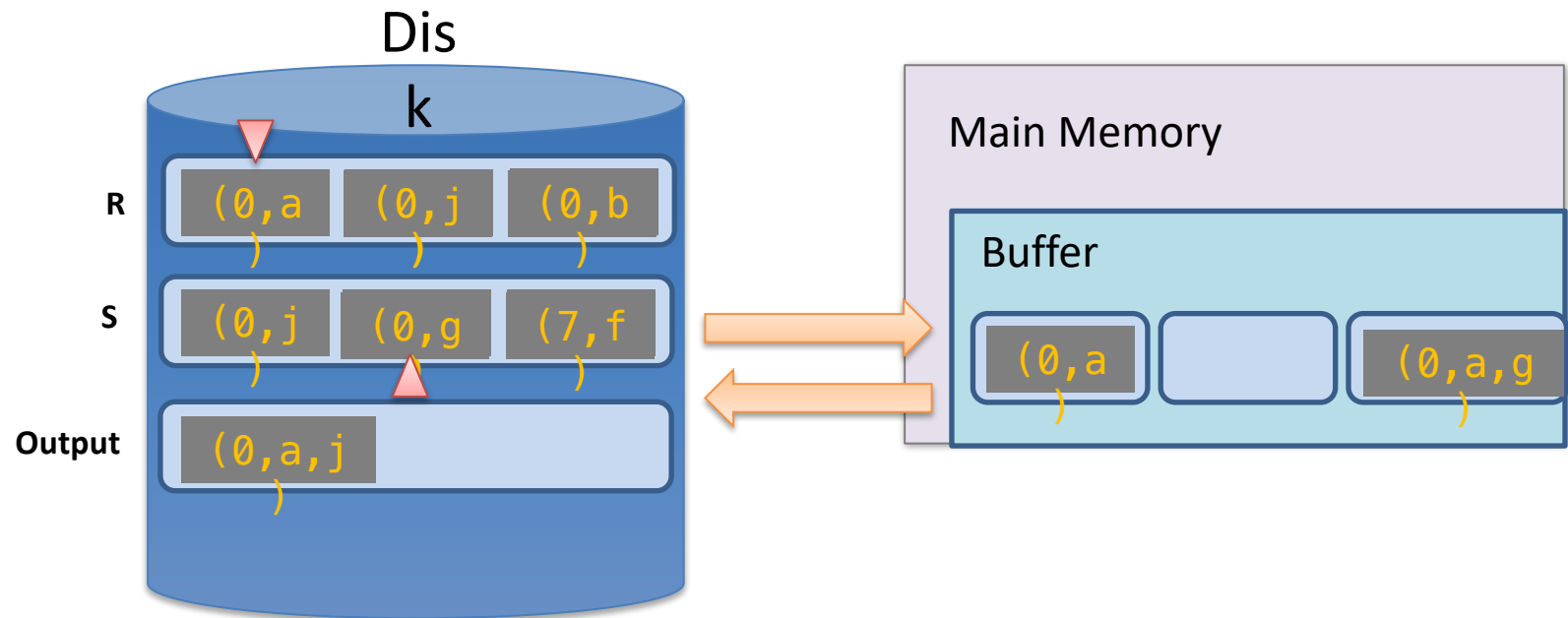
Multiple tuples with Same Join Key: “Backup”

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



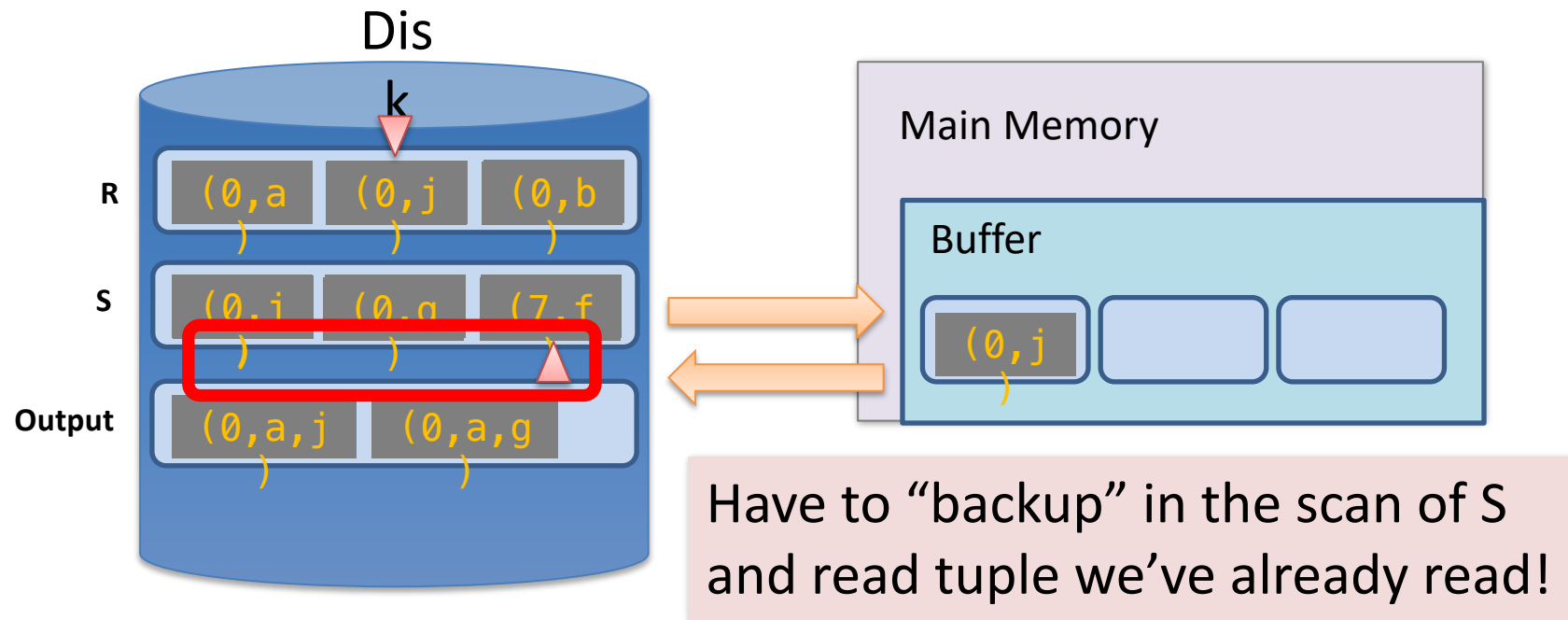
Multiple tuples with Same Join Key: “Backup”

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...



Backup

- At best, no backup \rightarrow 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

SMJ: Total cost

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

$$\begin{aligned} &\sim \text{Sort}(P(R)) + \text{Sort}(P(S)) \\ &+ P(R) + P(S) + \text{OUT} \end{aligned}$$

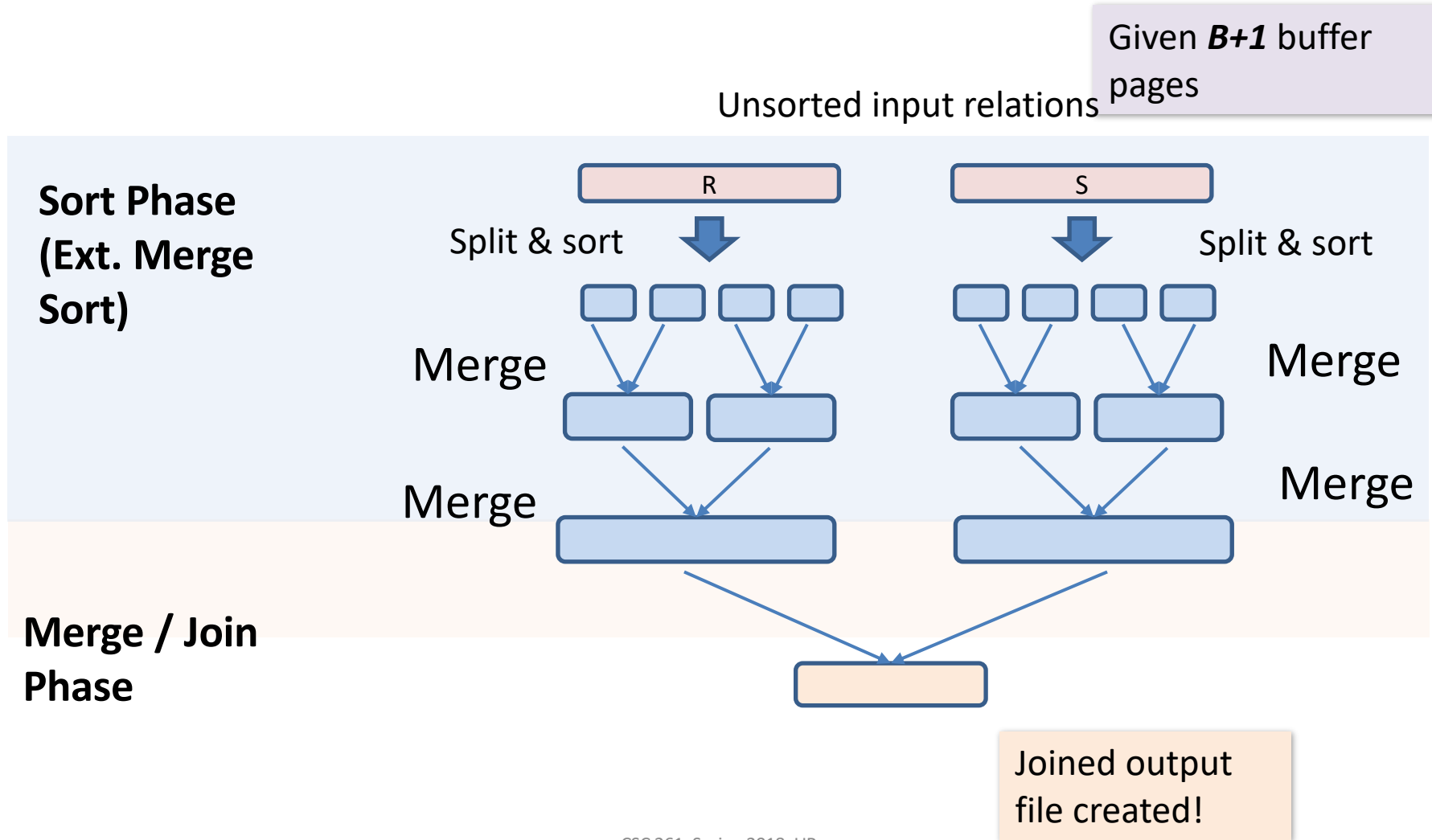
SMJ vs. BNLJ

- If we have 100 buffer pages, $P(R) = 1000$ pages and $P(S) = 500$ pages:
 - Sort both in two passes: $2 * 2 * 1000 + 2 * 2 * 500 = \mathbf{6,000 \text{ IOs}}$
 - Merge phase $1000 + 500 = 1,500 \text{ IOs}$
 - **$= 7,500 \text{ IOs} + \text{OUT}$**

What is BNLJ?

- $500 + 1000 * \left\lceil \frac{500}{98} \right\rceil = \mathbf{\underline{6,500 \text{ IOs} + \text{OUT}}}$
- But, if we have 35 buffer pages?
 - Sort Merge has same behavior (still 2 passes)
 - BNLJ? **$15,500 \text{ IOs} + \text{OUT!}$**

Basic SMJ



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.

4. HASH JOIN (HJ)

What you will learn about in this section

1. Hash Join
2. Memory requirements

Recall: Hashing

- **Magic of hashing:**
 - A hash function h_B maps into $[0, B-1]$
 - And maps nearly uniformly
- A hash **collision** is when $x \neq y$ but $h_B(x) = h_B(y)$
 - Note however that it will never occur that $x = y$ but $h_B(x) \neq h_B(y)$

Hash Join: High-level procedure

To compute $R \bowtie S$ on A :

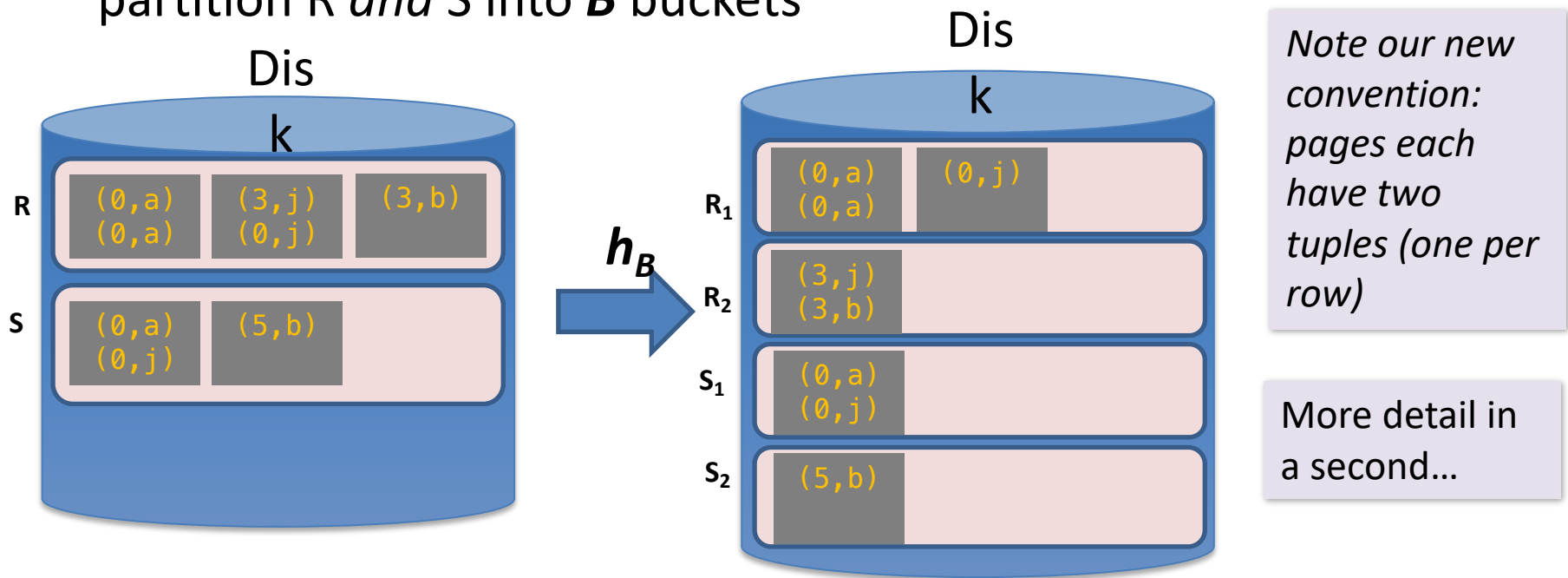
Note again that we are only considering equality constraints here

- 1. Partition Phase:** Using one (shared) hash function h_B , partition R and S into B buckets
- 2. Matching Phase:** Take pairs of buckets whose tuples have the same values for h , and join these
 1. Use BNLJ here; or hash again \rightarrow either way, operating on small partitions so fast!

We **decompose** the problem using h_B , then complete the join

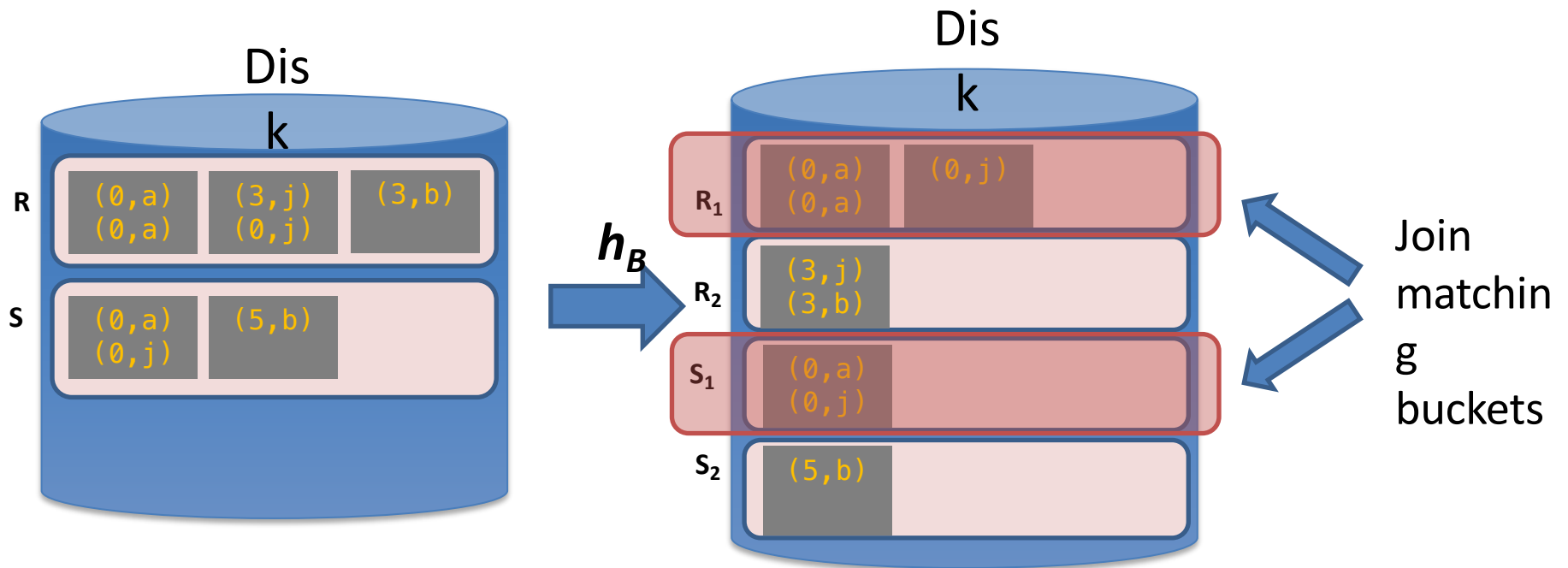
Hash Join: High-level procedure

1. Partition Phase: Using one (shared) hash function h_B , partition R and S into B buckets



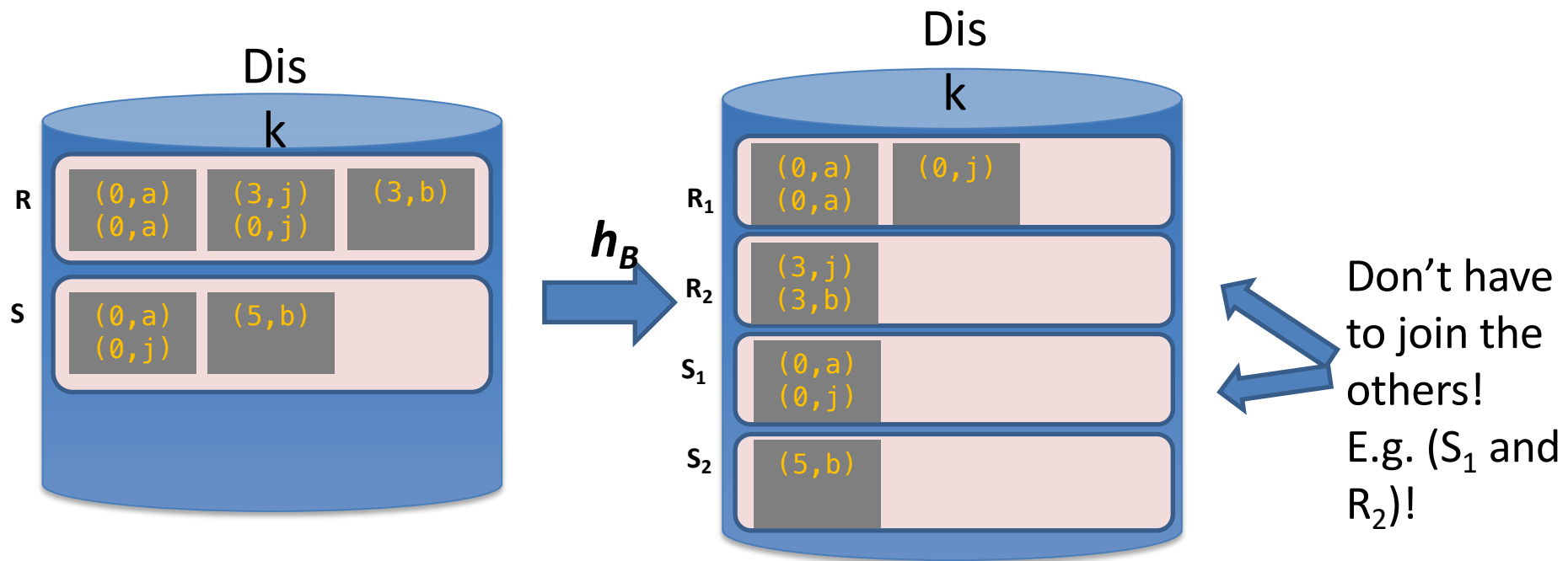
Hash Join: High-level procedure

2. Matching Phase: Take pairs of buckets whose tuples have the same values for h_B , and join these



Hash Join: High-level procedure

2. Matching Phase: Take pairs of buckets whose tuples have the same values for h_B , and join these



Hash Join Phase 1: Partitioning

Goal: For each relation, partition relation into **buckets** such that if $h_B(t_i.A) = h_B(t_j.A)$ they are in the same bucket

Given $B+1$ buffer pages, we partition into B buckets:

- We use B buffer pages for output (one for each bucket), and 1 for input
 - For each tuple t in input, copy to buffer page for $h_B(t.A)$
 - When page fills up, flush to disk.

How big are the resulting buckets?

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

- Given **N input pages, we partition into B buckets:**
- \rightarrow Ideally our buckets are each of size $\sim N/B$ pages

How big *do we want* the resulting buckets?

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

- Ideally, our buckets would be of size $\leq B - 1$ pages
 - **1** for input page, **1** for output page, **$B-1$** for each bucket
- Recall: If we want to join a bucket from R and one from S, we can do BNLJ **in linear time** if for *one of them (wlog say R)*, **$P(R) \leq B - 1$** !
 - And more generally, being able to fit bucket in memory is advantageous
- We can keep partitioning buckets that are $> B-1$ pages, until they are $\leq B - 1$ pages
 - Using a new hash key which will split them...

Recall for BNLJ:

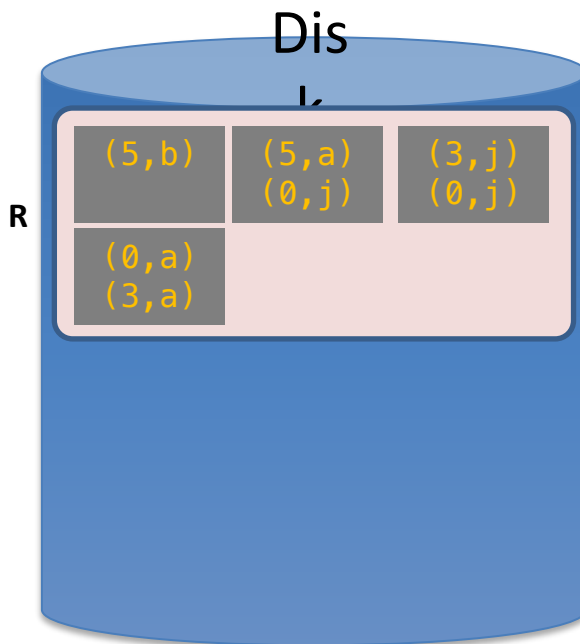
$$P(R) + \frac{P(R)P(S)}{B - 1}$$

We'll call each of these a "pass" again...

Hash Join Phase 1: Partitioning

Given $B+1 = 3$ buffer pages

We partition into $B = 2$ buckets using hash function h_2 so that we can have one buffer page for each partition (and one for input)



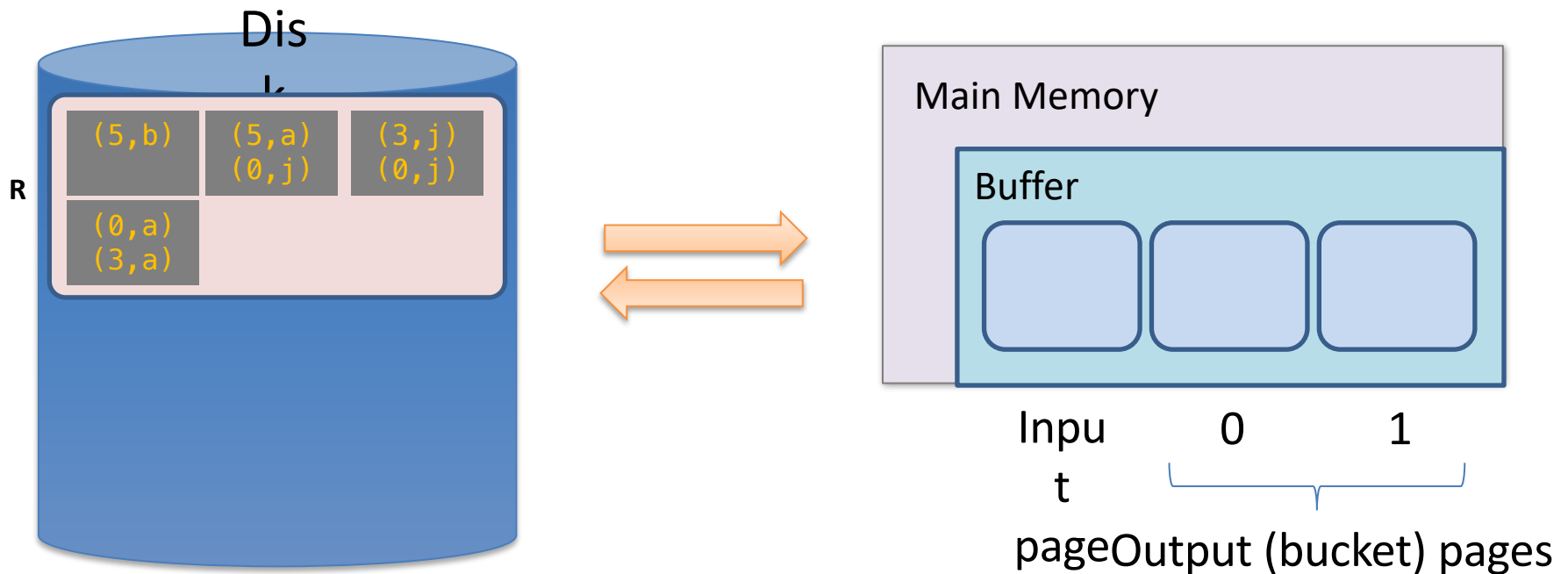
For simplicity, we'll look at partitioning one of the two relations- we just do the same for the other relation!

Recall: our goal will be to get $B = 2$ ***buckets*** of size $\leq B-1 \rightarrow 1$ ***page each***

Hash Join Phase 1: Partitioning

Given $B+1 = 3$ buffer pages

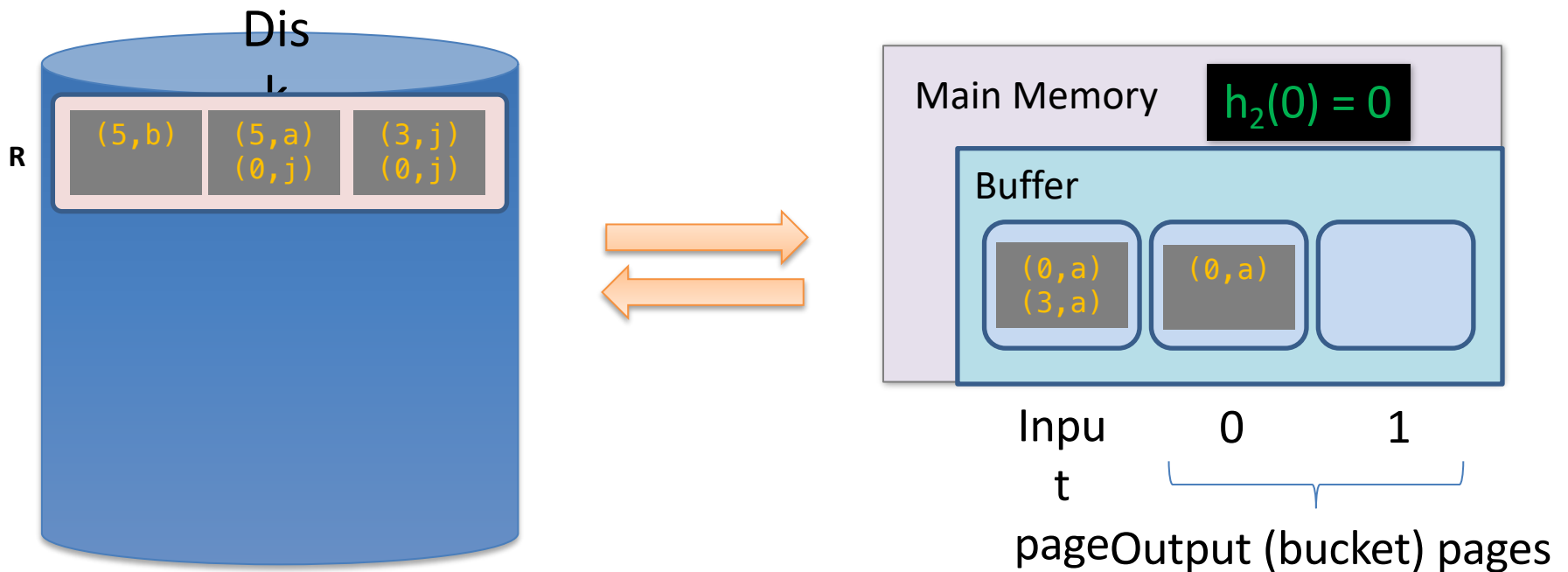
1. We read pages from R into the “input” page of the buffer...



Hash Join Phase 1: Partitioning

Given $B+1 = 3$ buffer pages

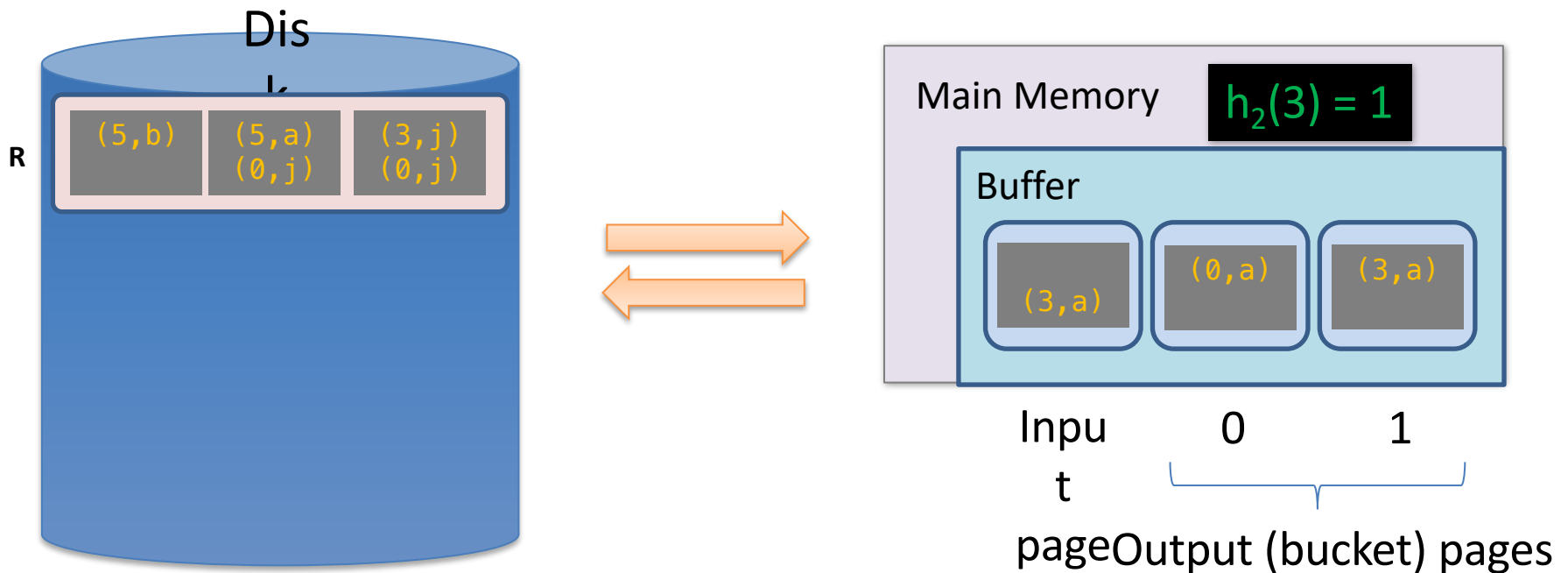
2. Then we use **hash function h_2** to sort into the buckets, which each have one page in the buffer



Hash Join Phase 1: Partitioning

Given $B+1 = 3$ buffer pages

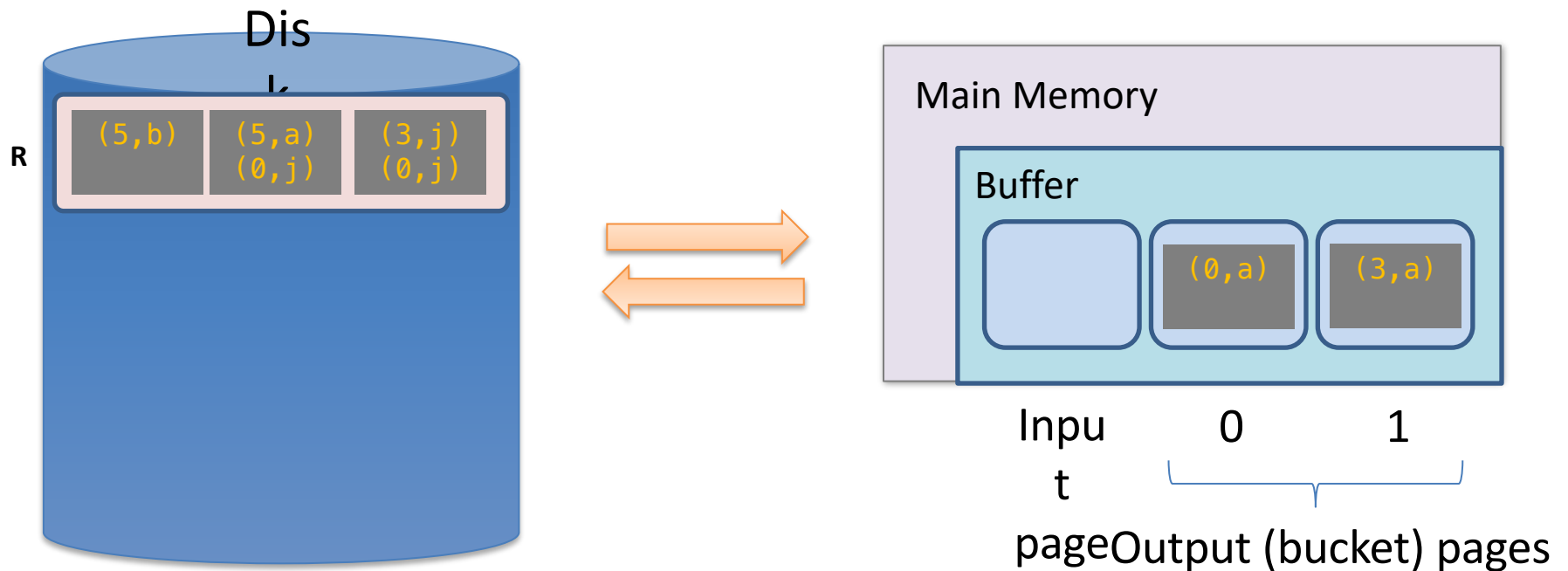
2. Then we use **hash function h_2** to sort into the buckets, which each have one page in the buffer



Hash Join Phase 1: Partitioning

Given $B+1 = 3$ buffer pages

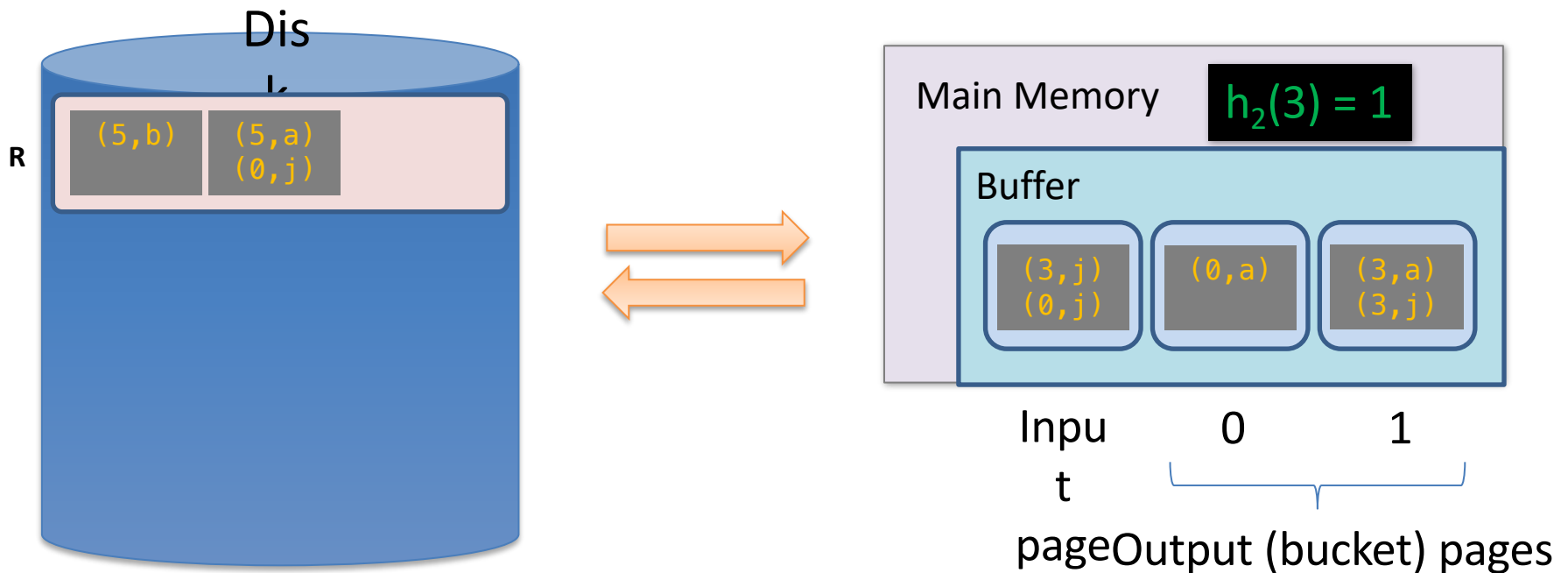
3. We repeat until the buffer bucket pages are full...



Hash Join Phase 1: Partitioning

Given $B+1 = 3$ buffer pages

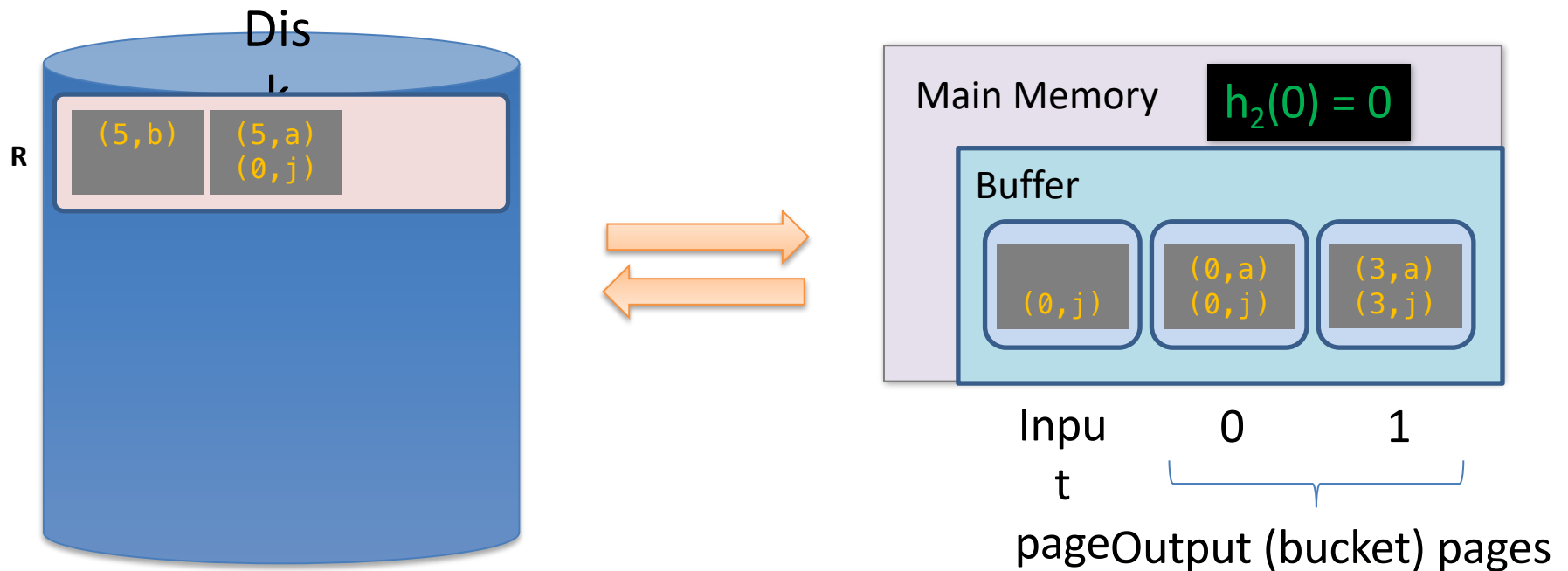
3. We repeat until the buffer bucket pages are full...



Hash Join Phase 1: Partitioning

Given $B+1 = 3$ buffer pages

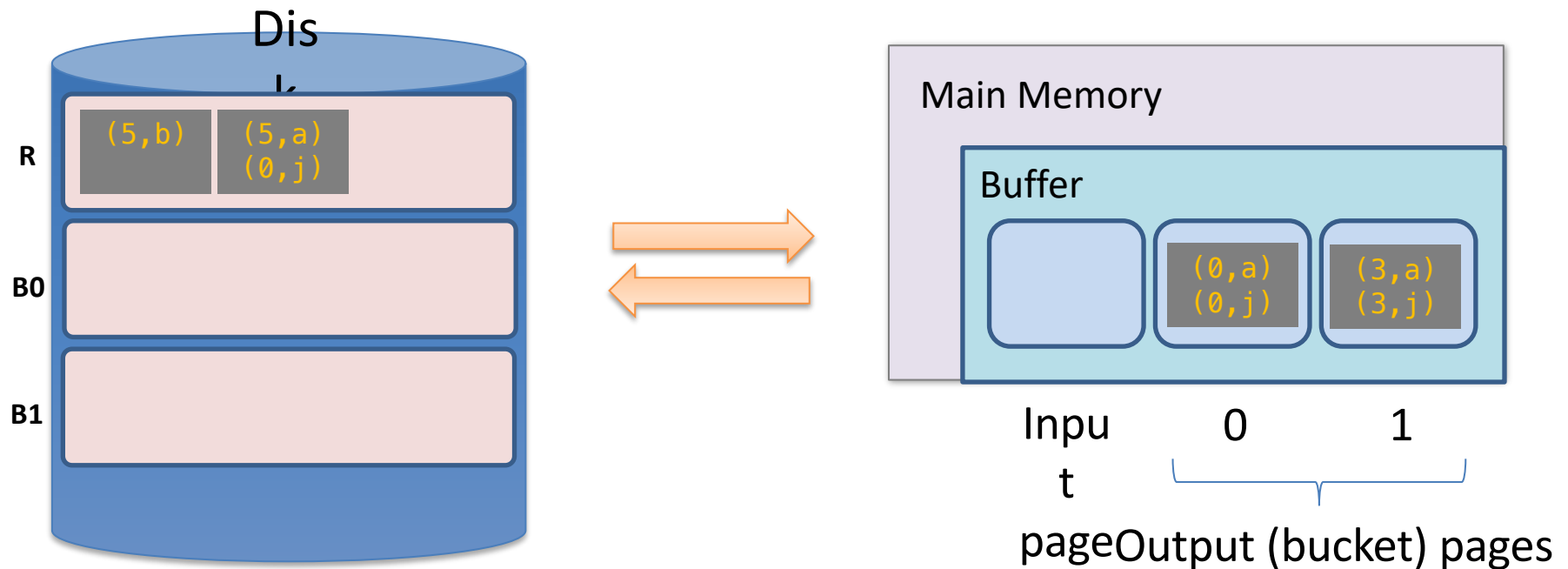
3. We repeat until the buffer bucket pages are full...



Hash Join Phase 1: Partitioning

Given $B+1 = 3$ buffer pages

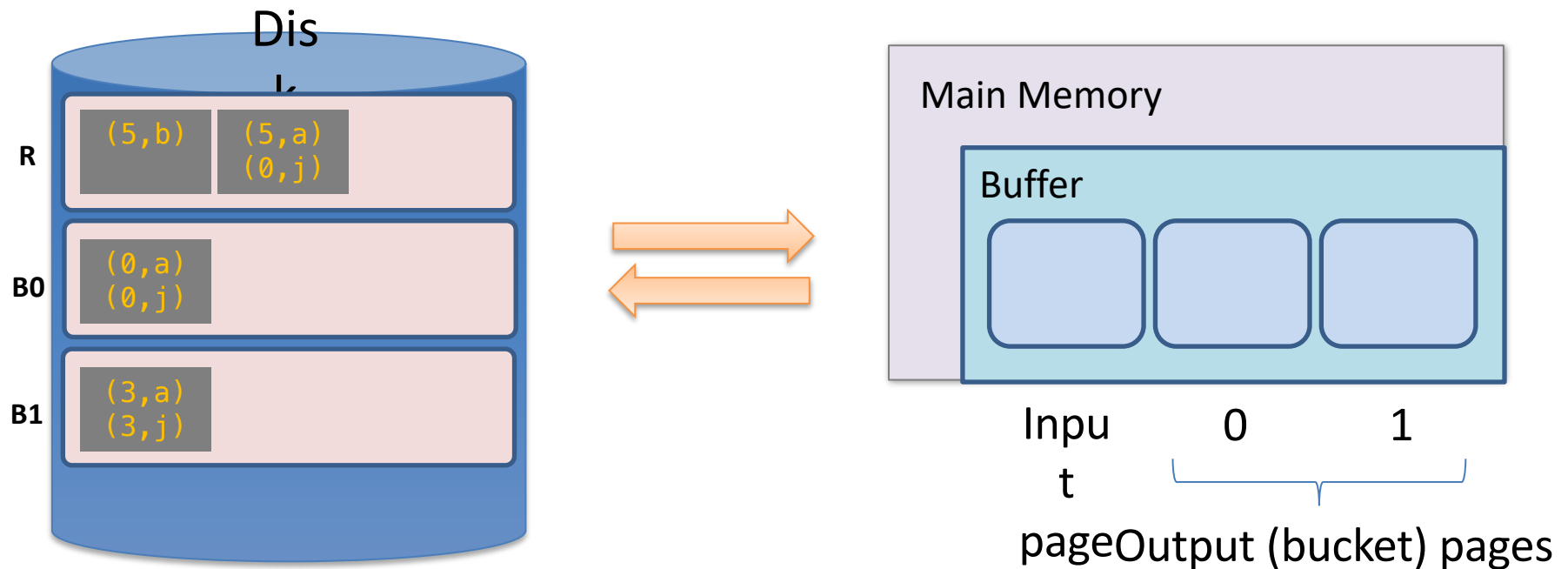
3. We repeat until the buffer bucket pages are full... then flush to disk



Hash Join Phase 1: Partitioning

Given $B+1 = 3$ buffer pages

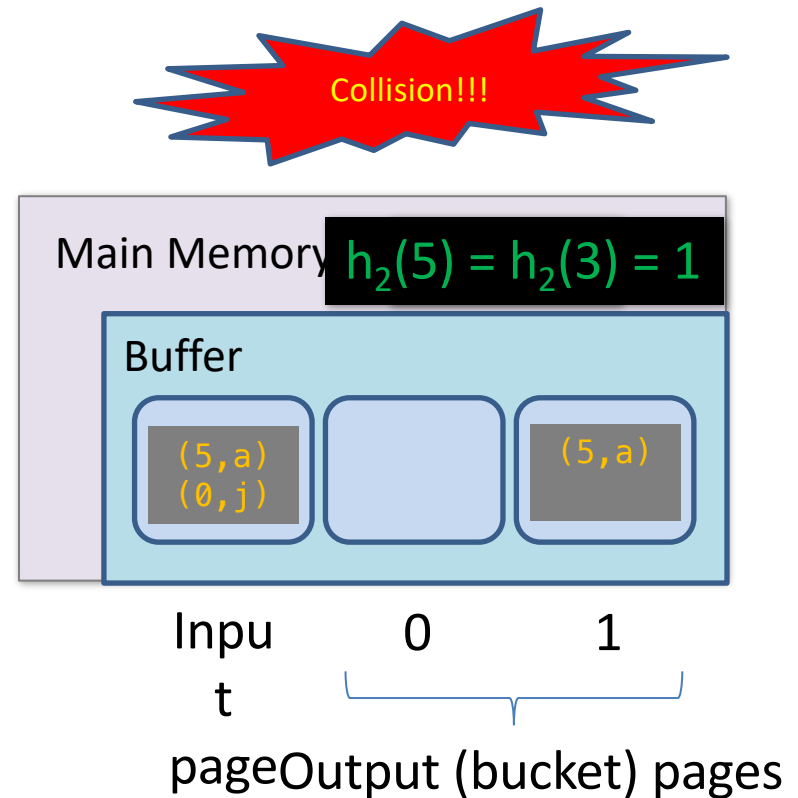
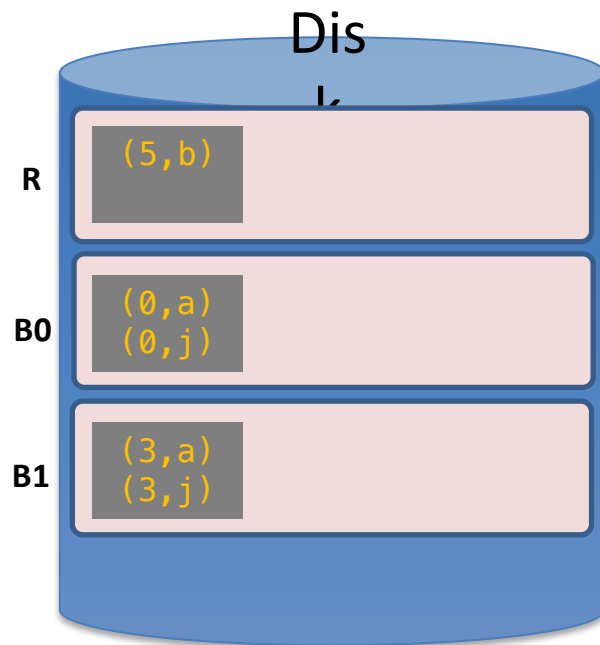
3. We repeat until the buffer bucket pages are full... then flush to disk



Hash Join Phase 1: Partitioning

Given $B+1 = 3$ buffer pages

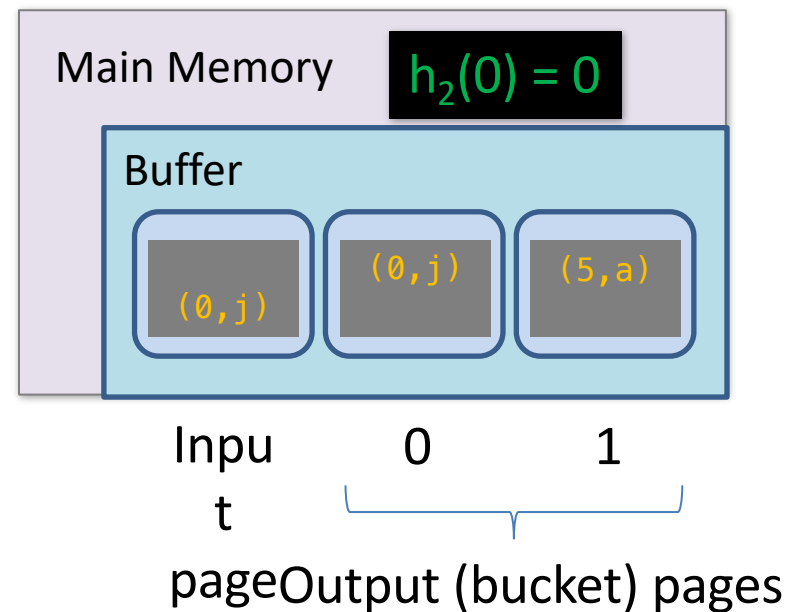
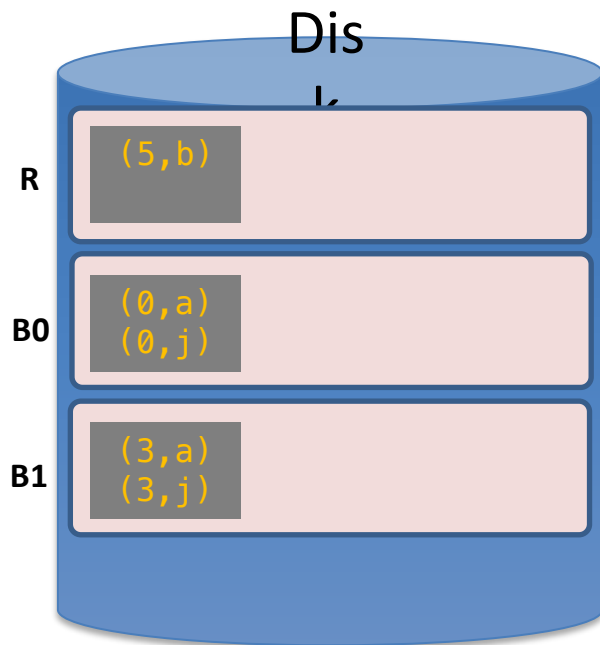
Note that collisions can occur!



Hash Join Phase 1: Partitioning

Given $B+1 = 3$ buffer pages

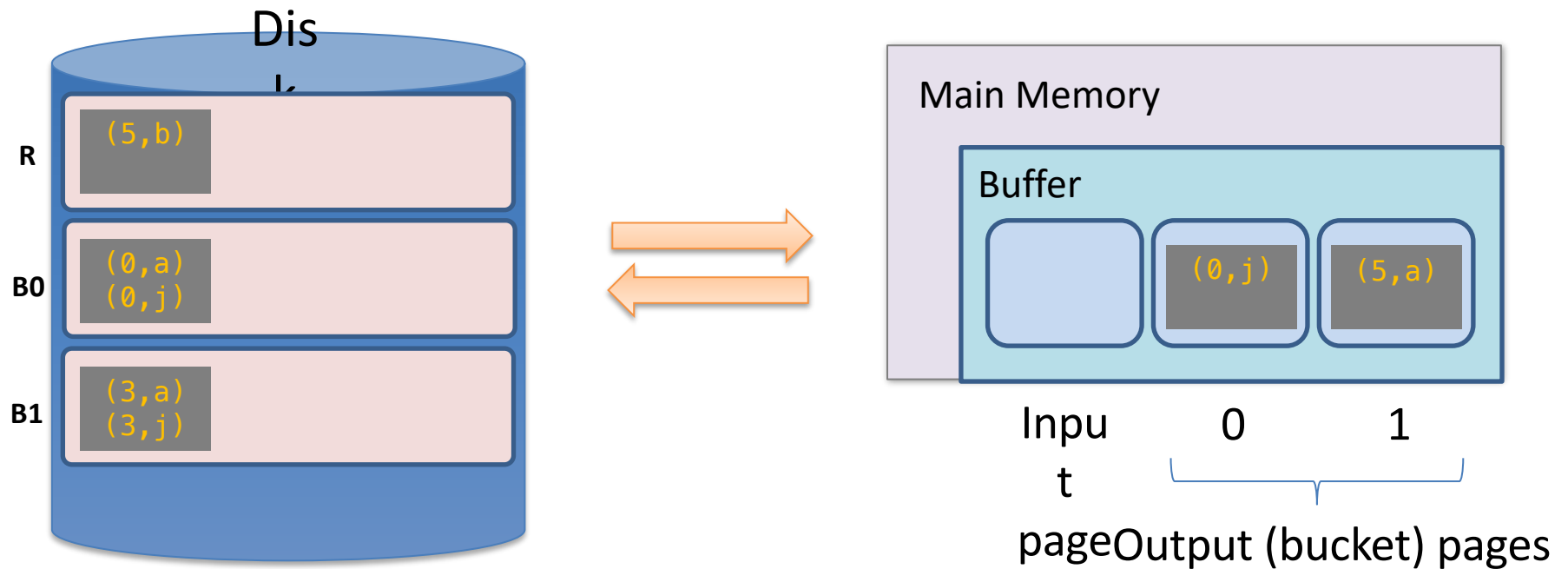
Finish this pass...



Hash Join Phase 1: Partitioning

Given $B+1 = 3$ buffer pages

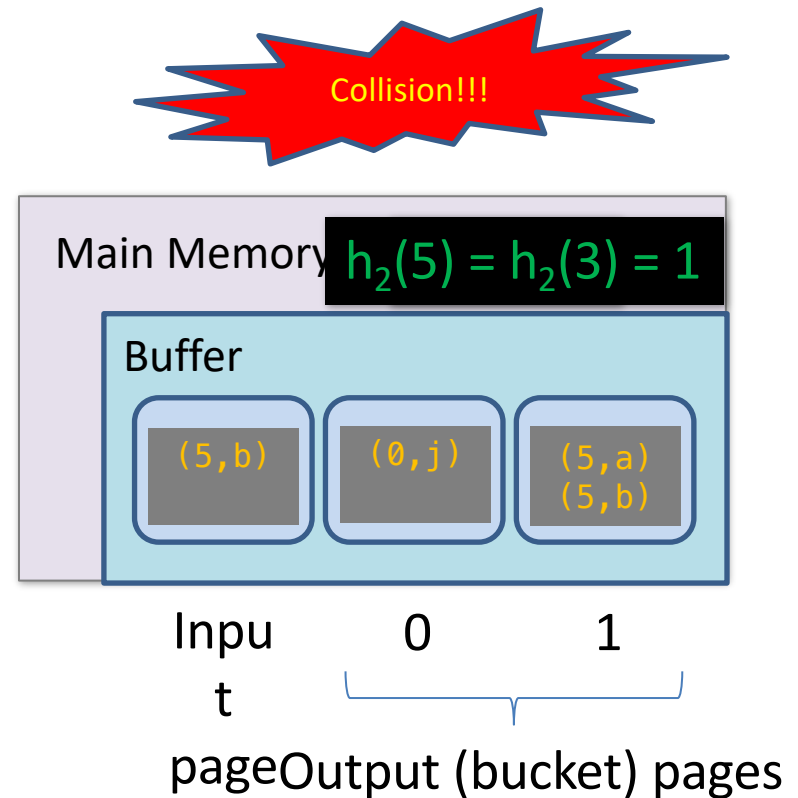
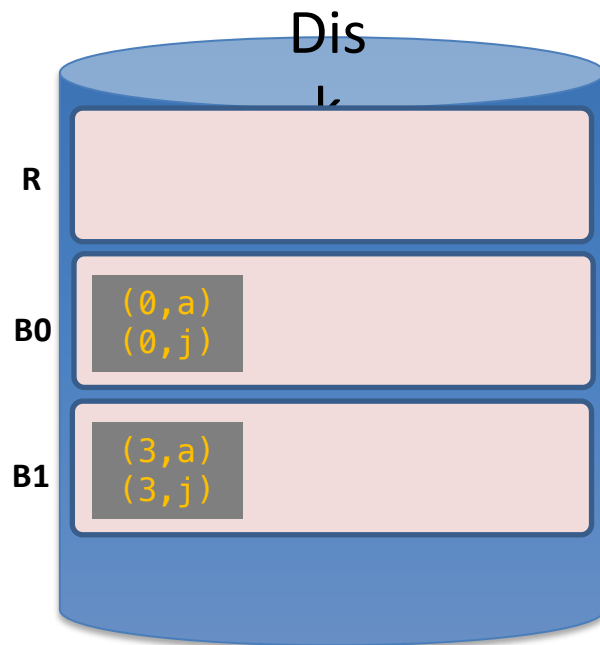
Finish this pass...



Hash Join Phase 1: Partitioning

Given $B+1 = 3$ buffer pages

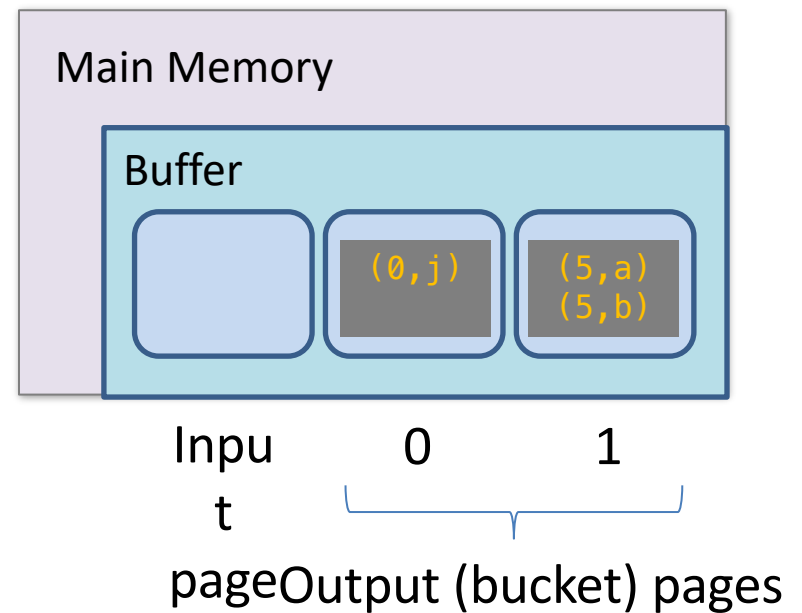
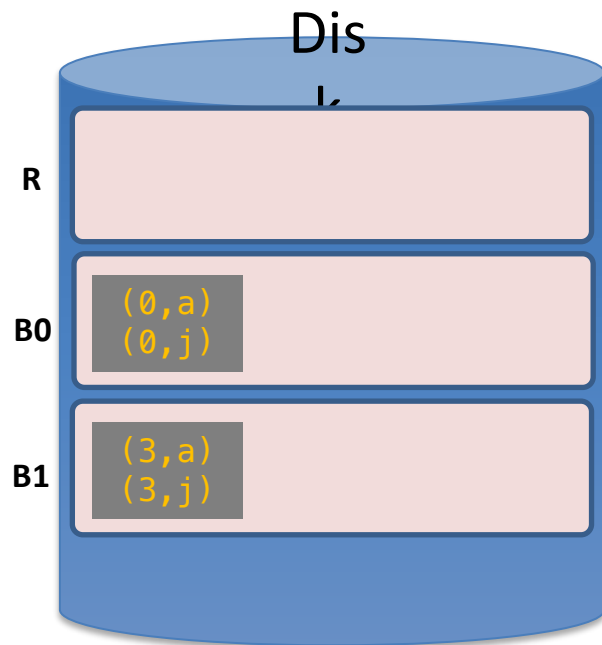
Finish this pass...



Hash Join Phase 1: Partitioning

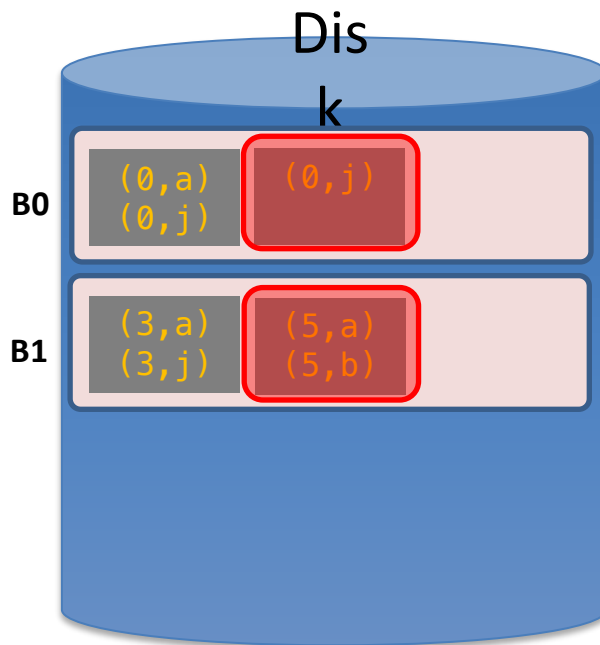
Given $B+1 = 3$ buffer pages

Finish this pass...



Hash Join Phase 1: Partitioning

Given $B+1 = 3$ buffer pages



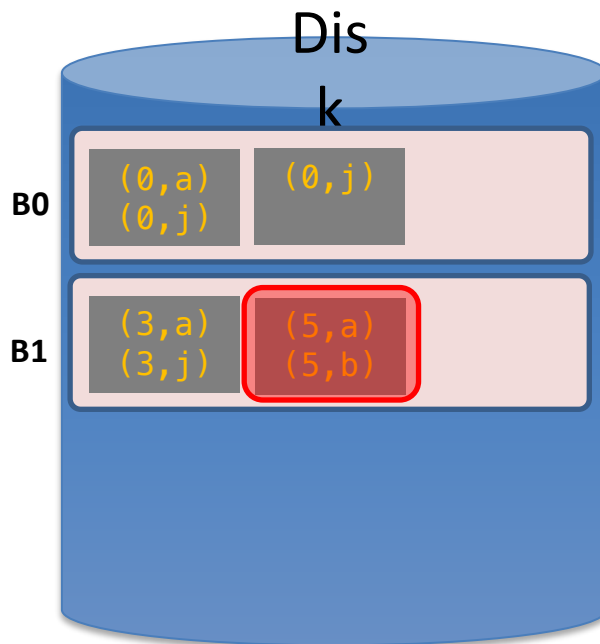
We wanted buckets of size $B-1 = 1...$ *however we got larger ones due to:*

(1) Duplicate join keys

(2) Hash collisions

Hash Join Phase 1: Partitioning

Given $B+1 = 3$ buffer pages



To take care of larger buckets caused by (2) hash collisions, we can just do another pass!

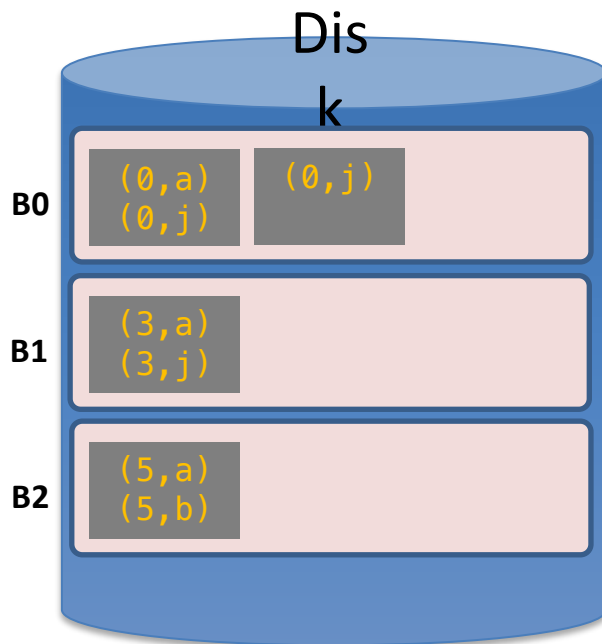
What hash function should we use?

Do another pass with a different hash function, h'_2 , ideally such that:

$$h'_2(3) \neq h'_2(5)$$

Hash Join Phase 1: Partitioning

Given $B+1 = 3$ buffer pages



To take care of larger buckets caused by (2) hash collisions, we can just do another pass!

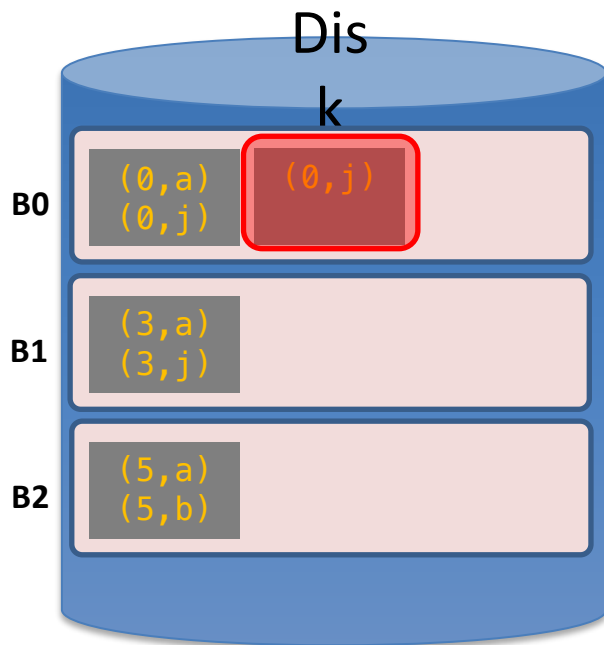
What hash function should we use?

Do another pass with a different hash function, h'_2 , ideally such that:

$$h'_2(3) \neq h'_2(5)$$

Hash Join Phase 1: Partitioning

Given $B+1 = 3$ buffer pages



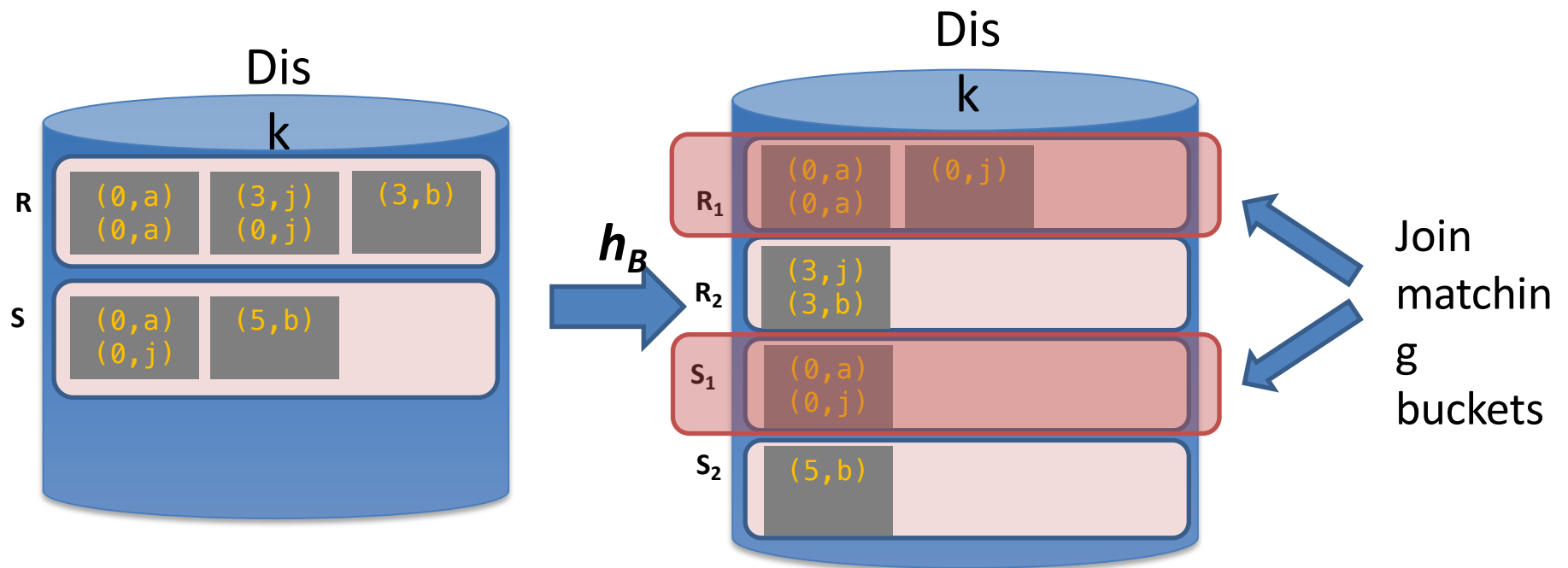
What about duplicate join keys?
Unfortunately this is a problem...
but usually not a huge one.

We call this unevenness
in the bucket size skew

Now that we have partitioned R and S...

Hash Join Phase 2: Matching

- Now, we just join pairs of buckets from R and S that have the same hash value to complete the join!



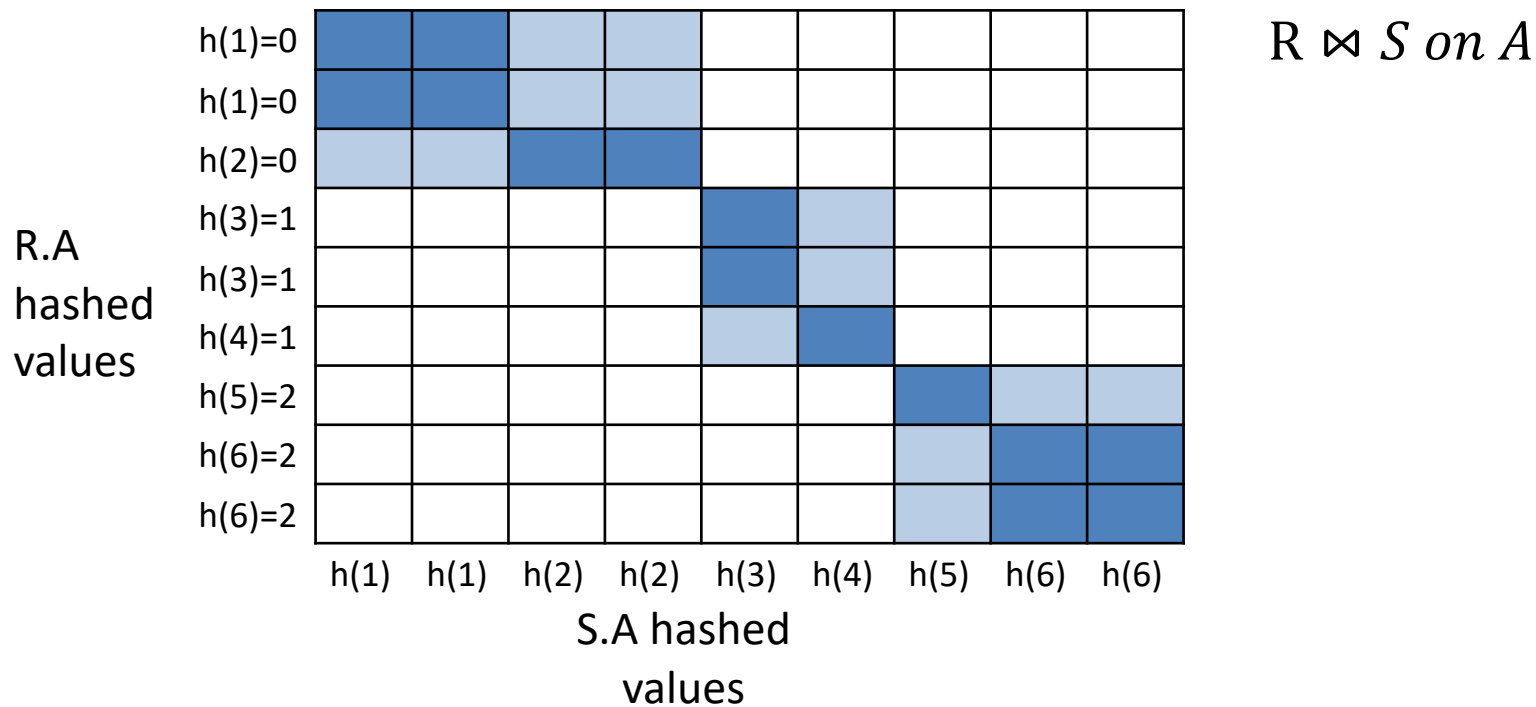
Hash Join Phase 2: Matching

- Note that since $x = y \rightarrow h(x) = h(y)$, we only need to consider pairs of buckets (one from R, one from S) that have the same hash function value
- If our buckets are $\sim B - 1$ pages, can join each such pair using BNLJ *in linear time*; recall (with $P(R) = B-1$):

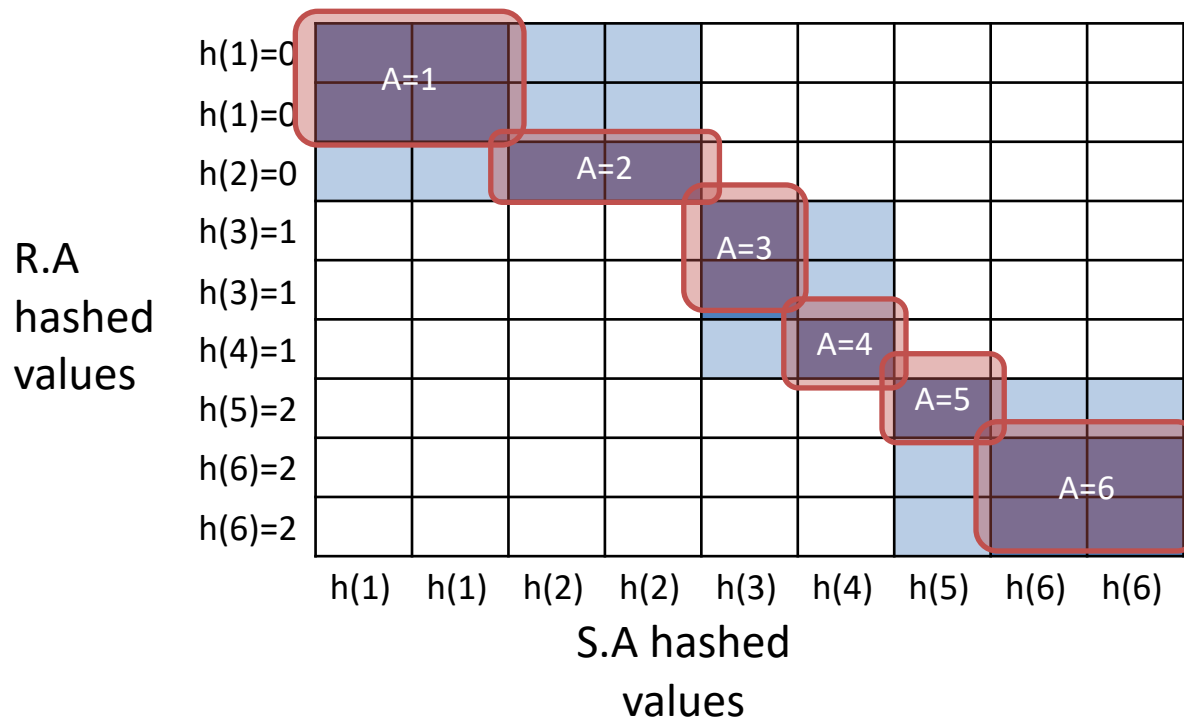
$$\text{BNLJ Cost: } P(R) + \frac{P(R)P(S)}{B-1} = P(R) + \frac{(B-1)P(S)}{B-1} = P(R) + P(S)$$

Joining the pairs of buckets is linear!
(As long as smaller bucket $\leq B-1$ pages)

Hash Join Phase 2: Matching



Hash Join Phase 2: Matching

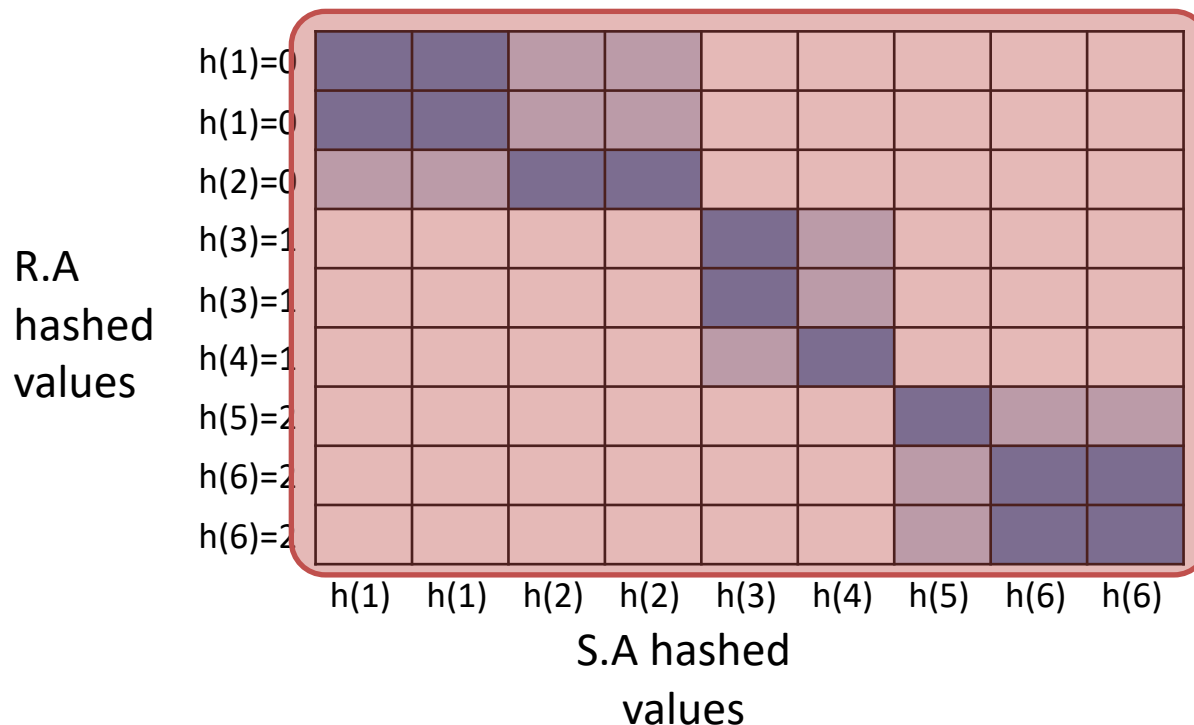


$R \bowtie S \text{ on } A$

To perform the join, we ideally just need to explore the dark blue regions

= the tuples with same values of the join key A

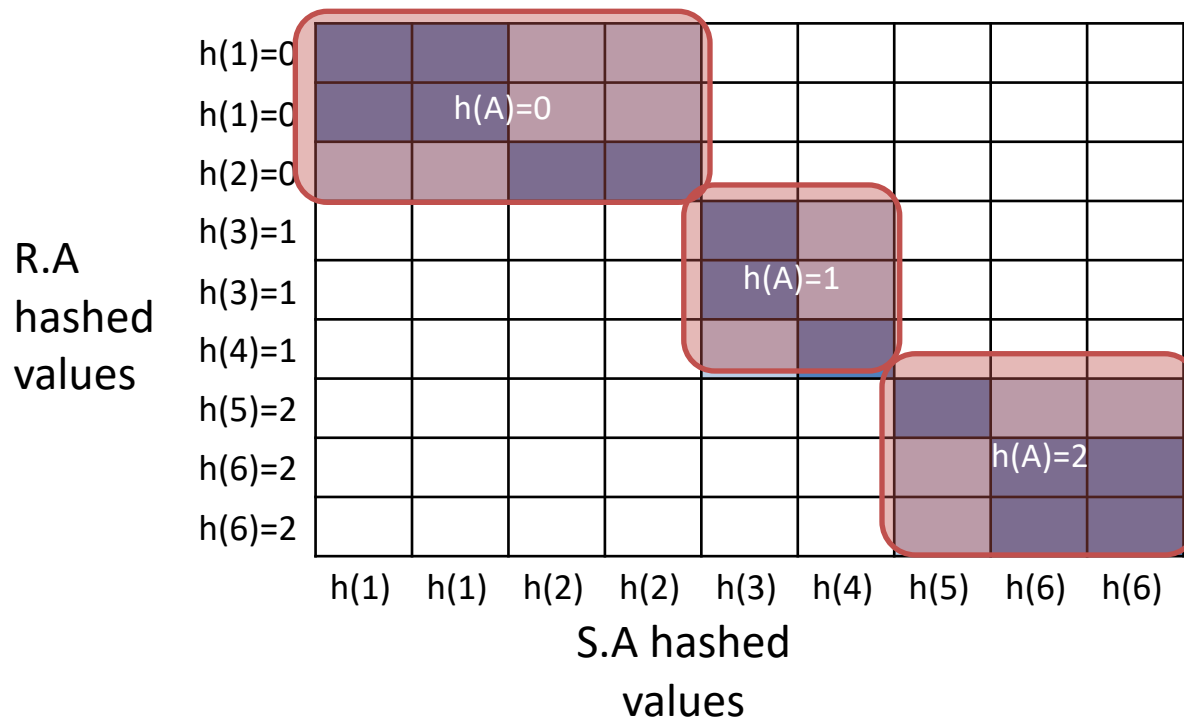
Hash Join Phase 2: Matching



$R \bowtie S$ on A

With a join algorithm like BNLJ that doesn't take advantage of equijoin structure, we'd have to explore this **whole grid!**

Hash Join Phase 2: Matching



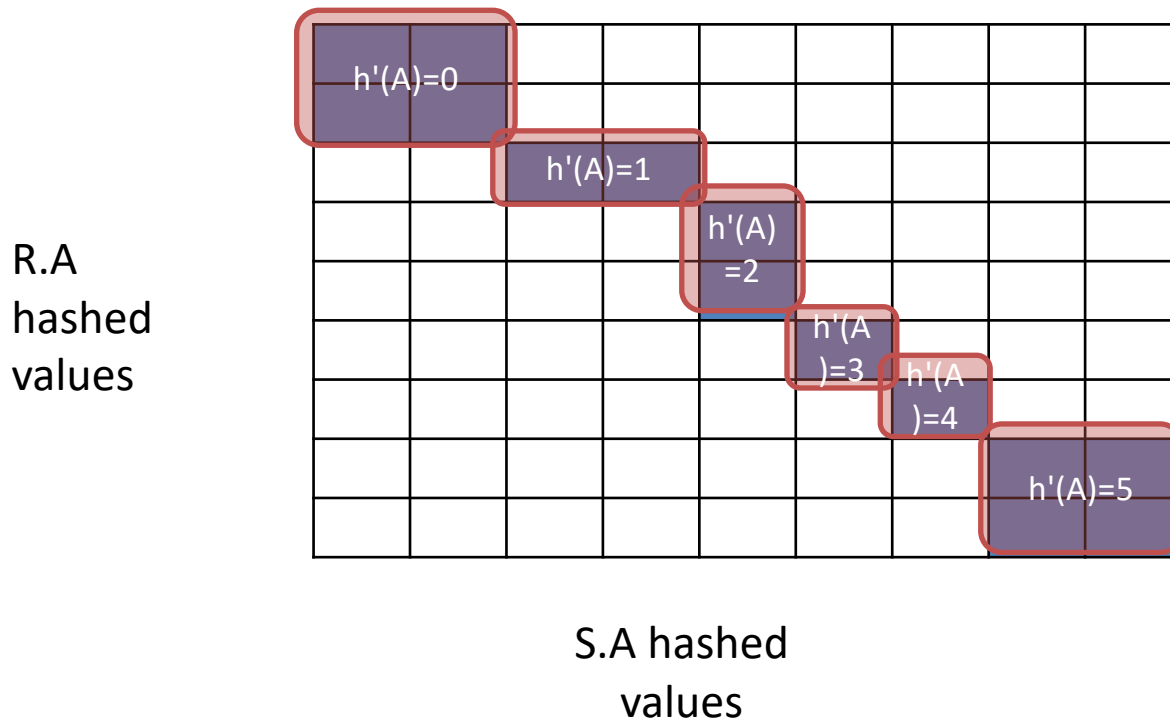
$R \bowtie S$ on A

With HJ, we only explore the **blue** regions

= the tuples with same values of $h(A)$!

We can apply BNLJ to each of these regions

Hash Join Phase 2: Matching



$R \bowtie S$ on A

An alternative to
applying BNLJ:

We could also hash
again, and keep
doing passes in
memory to reduce
further!

Hash Join Summary

- **Partitioning** requires reading + writing each page of R,S
 - $\rightarrow 2(P(R)+P(S))$ IOs
- **Matching** (with BNLJ) requires reading each page of R,S
 - $\rightarrow P(R) + P(S)$ IOs
- **Writing out results** could be as bad as $P(R)*P(S)$... but probably closer to $P(R)+P(S)$

HJ takes $\sim 3(P(R)+P(S)) + OUT$ IOs!