CS 6400 A

Database Systems Concepts and Design

Lecture 11

09/30/24

Announcement

- Project proposal due this Wednesday (Oct 2)

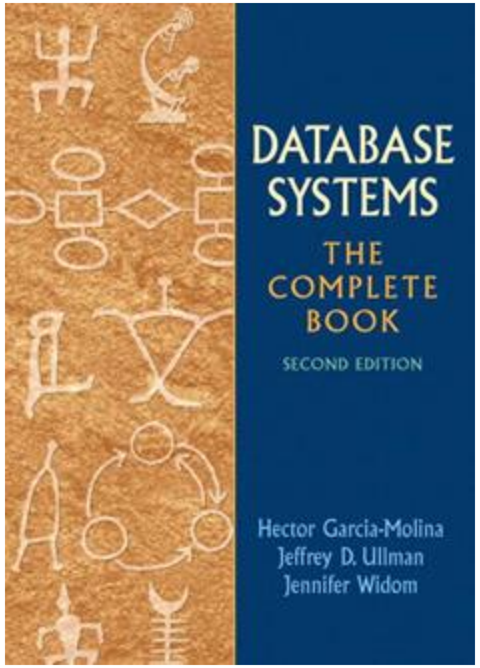
- Assignment 2 released today
  - Start early!!!
  - Due Oct 21

- Midterm
  - Answer will be released on canvas
  - Grades will be released on Wednesday

Reading Materials

Database Systems: The Complete Book (2nd edition)

- Chapter 15: Query Execution



Acknowledgement: The following slides have been adapted from CS145 (Intro to Big Data Systems) taught by Peter Bailis.

Agenda

RECAP: Joins

1. Nested Loop Join (NLJ)

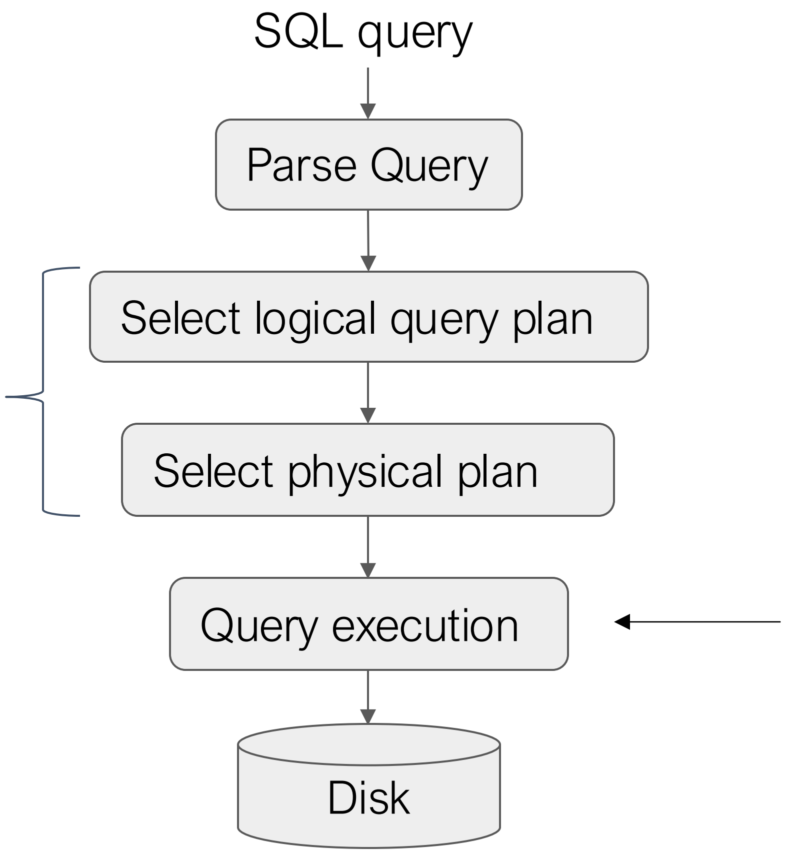
2. Sort-Merge Join (SMJ)

3. Hash Join (HJ)

RDBMS Architecture

How does a SQL engine work ?

Query optimization (next 2 lectures)



Query execution (this lecture): algorithms that manipulate the data of the database

We will use JOIN algorithms as an example

- Arguable one of the most computational expensive operations in relational databases

- As we will see, different implementations of JOINs can make a huge difference in performance.

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

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

```
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

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

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

```
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
2	3	4	3
2	5	2	2
2	5	2	3

Joins: Example

$\mathbf{R} \bowtie S$

```
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
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 $\mathrm{r} \in R, s \in S$ 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

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?

1. Nested Loop Joins

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 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")

Join can involve > 2 tables, and some algorithms do support non-equality constraints!

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

- Note also that we omit ceilings in calculations… good exercise to put back in!

Nested Loop Join (NLJ)

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

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

Nested Loop Join (NLJ)

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

```
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)
```

<u>Cost:</u>

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

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

```
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)

What would OUT be if our join condition is trivial (if TRUE)?

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

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)
```

<u>Cost:</u>

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

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!

IO-Aware Approach

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)
```

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)

Given B+1 pages of memory

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)
```

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

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)
```

Cost:

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

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)

Given B+1 pages of memory

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)
```

Cost:

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

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 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

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

→

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

Example:

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

Ignoring OUT here…

NLJ: Cost = 500 + **50,000*1000** = **50 Million IOs** ~= 140 hours

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

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:

NLJ $$\mathrm{P(R)} + \mathrm{T(R)P(S)} + \mathrm{OUT}$$

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

Now we'll see some (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 - 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:
  if 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, $\mathbf{L} \sim \mathbf{3}$ is good est.

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

2. Sort-Merge Join (SMJ)

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***

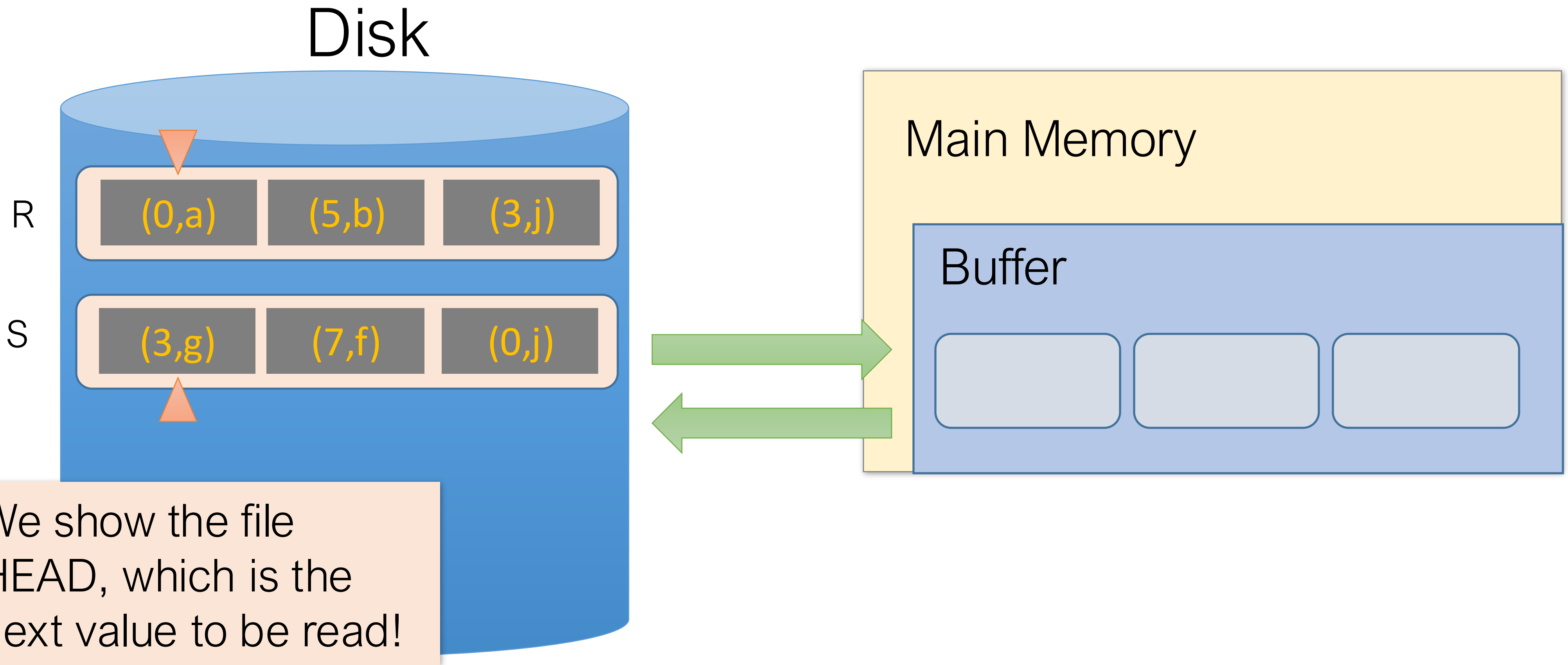
2. ***Scan*** sorted files and "merge"

3. *[May need to "backup"- see next subsection]*

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

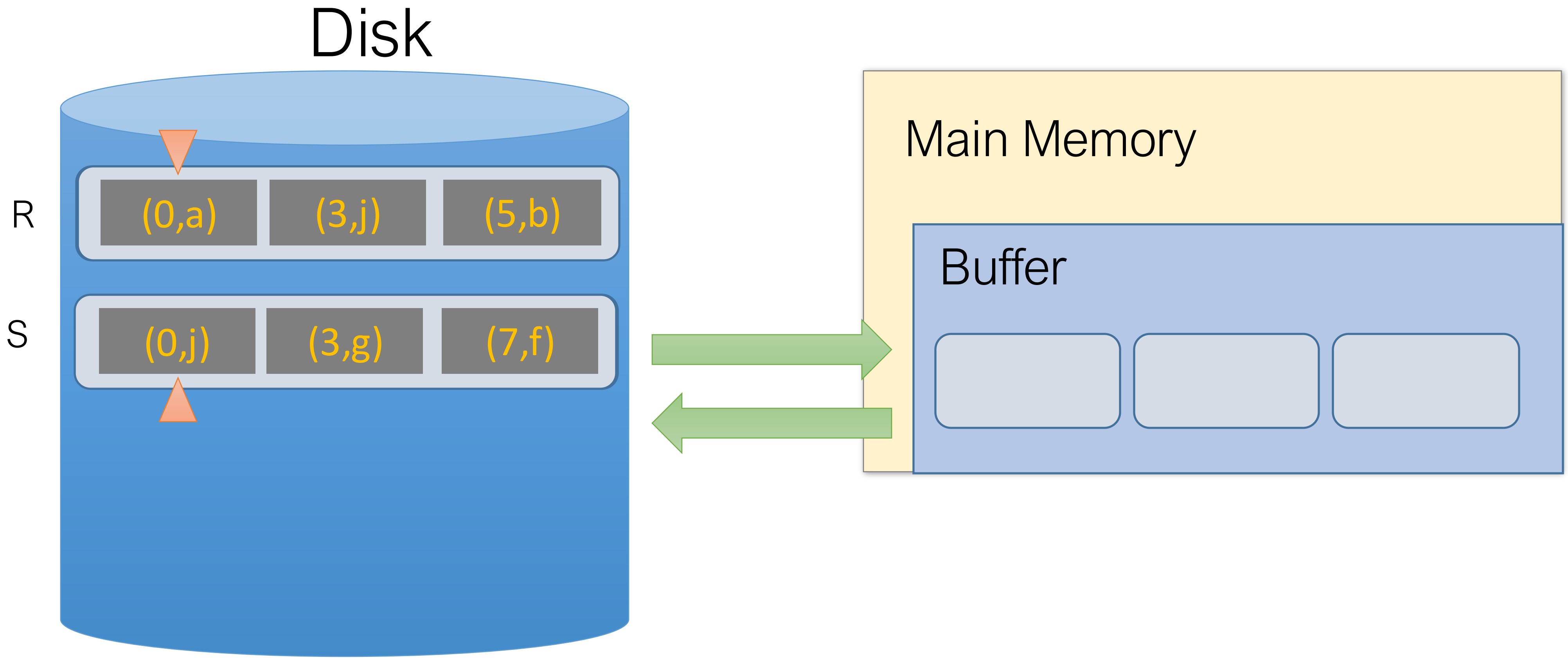
- For simplicity: Let each page be ***one tuple***, and let the first value be A



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

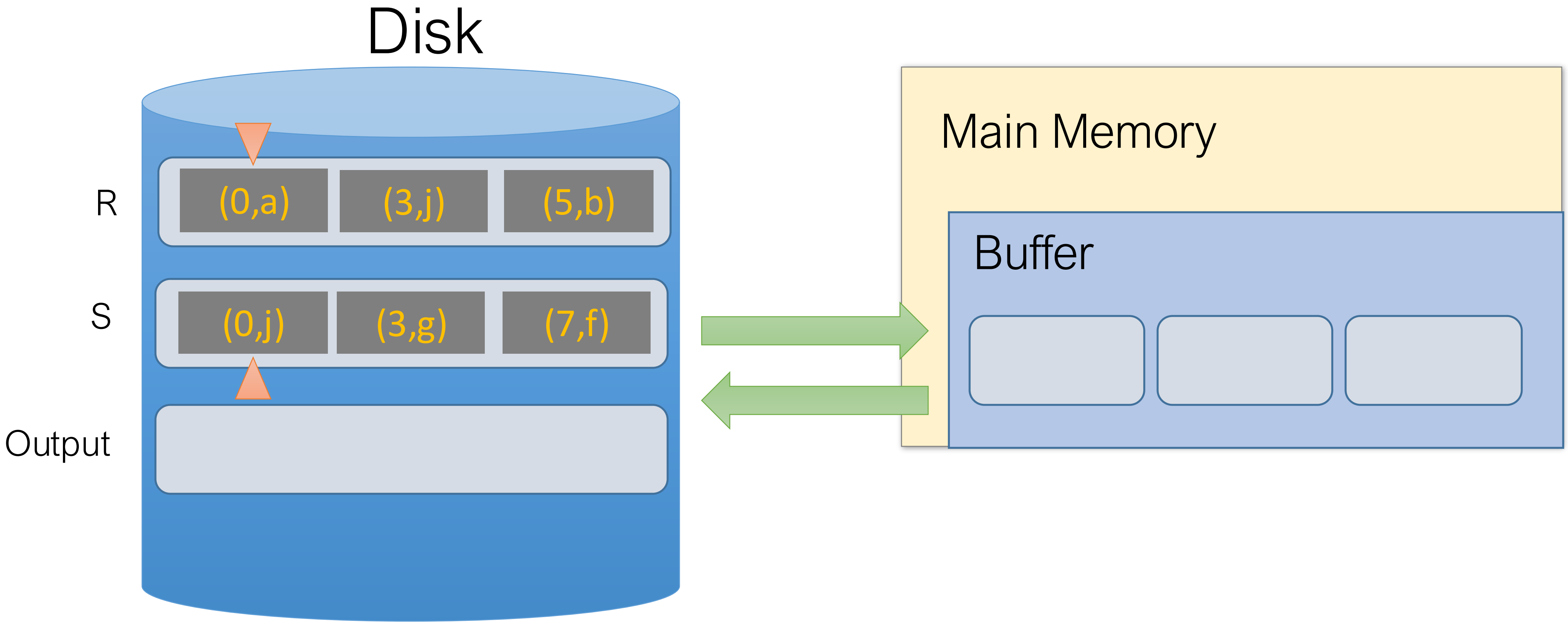
SMJ Example: $\mathrm{R} \bowtie S\ on\ A$ with 3 page buffer

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



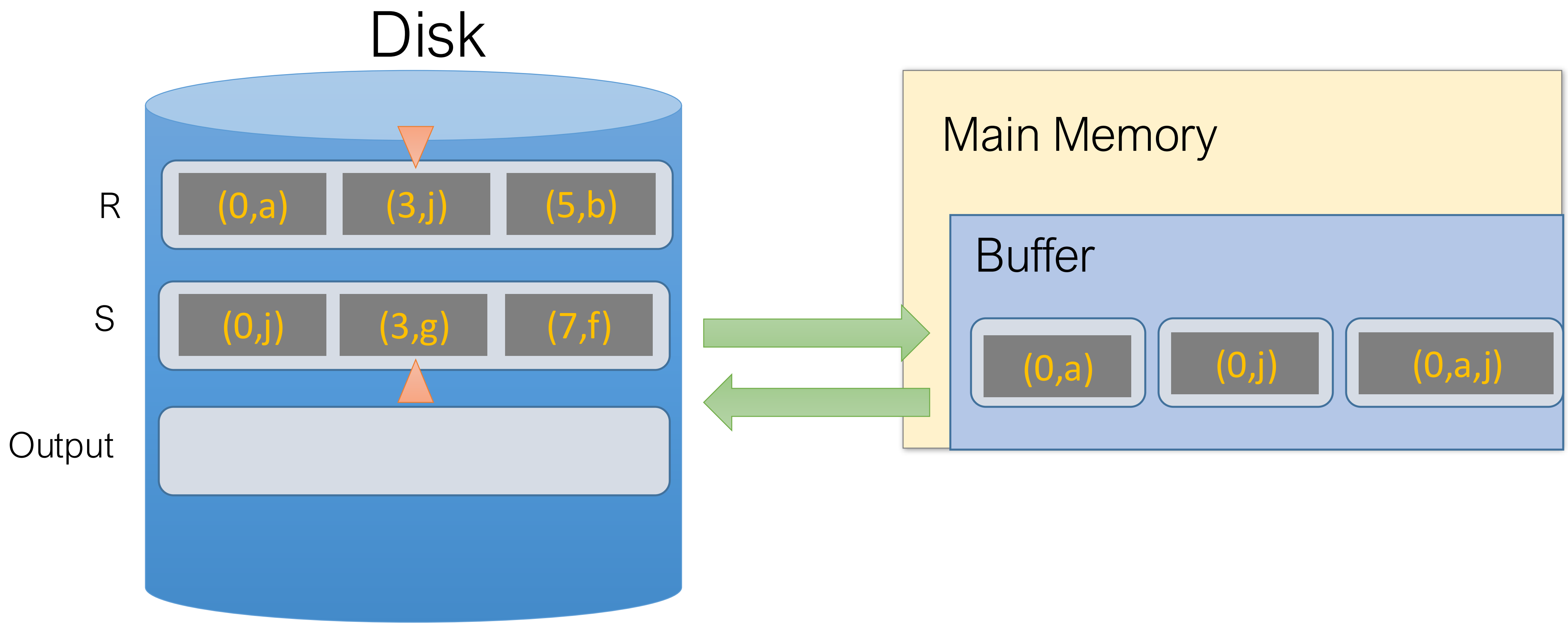
SMJ Example: $\mathrm{R} \bowtie S\ on\ A$ with 3 page buffer

2. Scan and “merge” on join key!



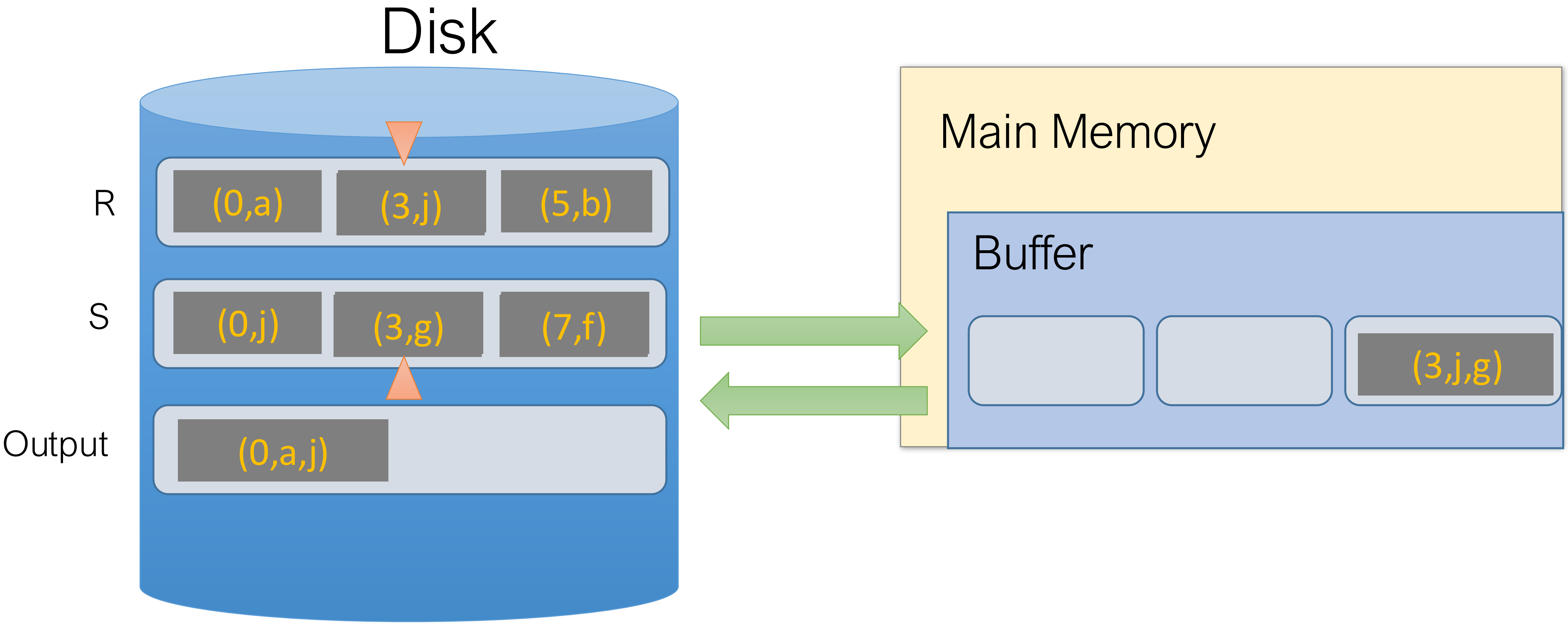
SMJ Example: $R \bowtie S \text{ on } A$ with 3 page buffer

2. Scan and "merge" on join key!



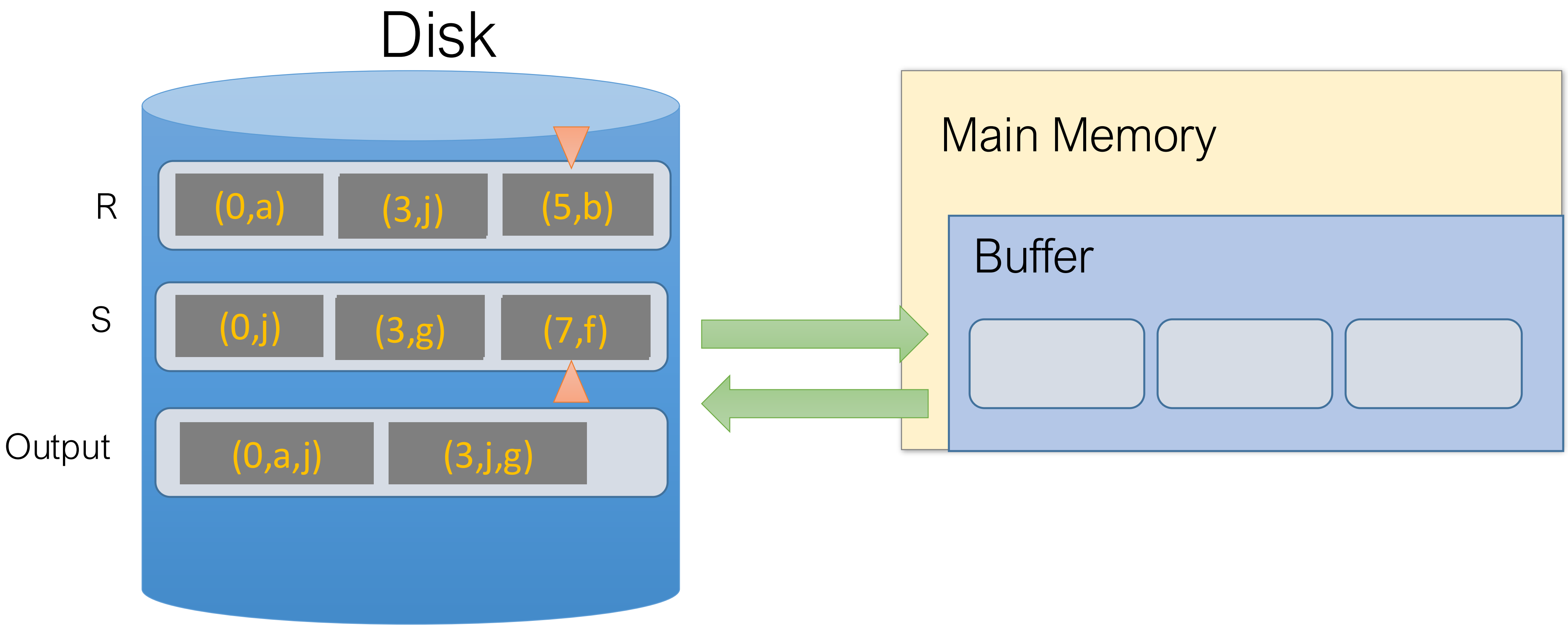
SMJ Example: $R \bowtie S$ *on* A with 3 page buffer

2. Scan and "merge" on join key!



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

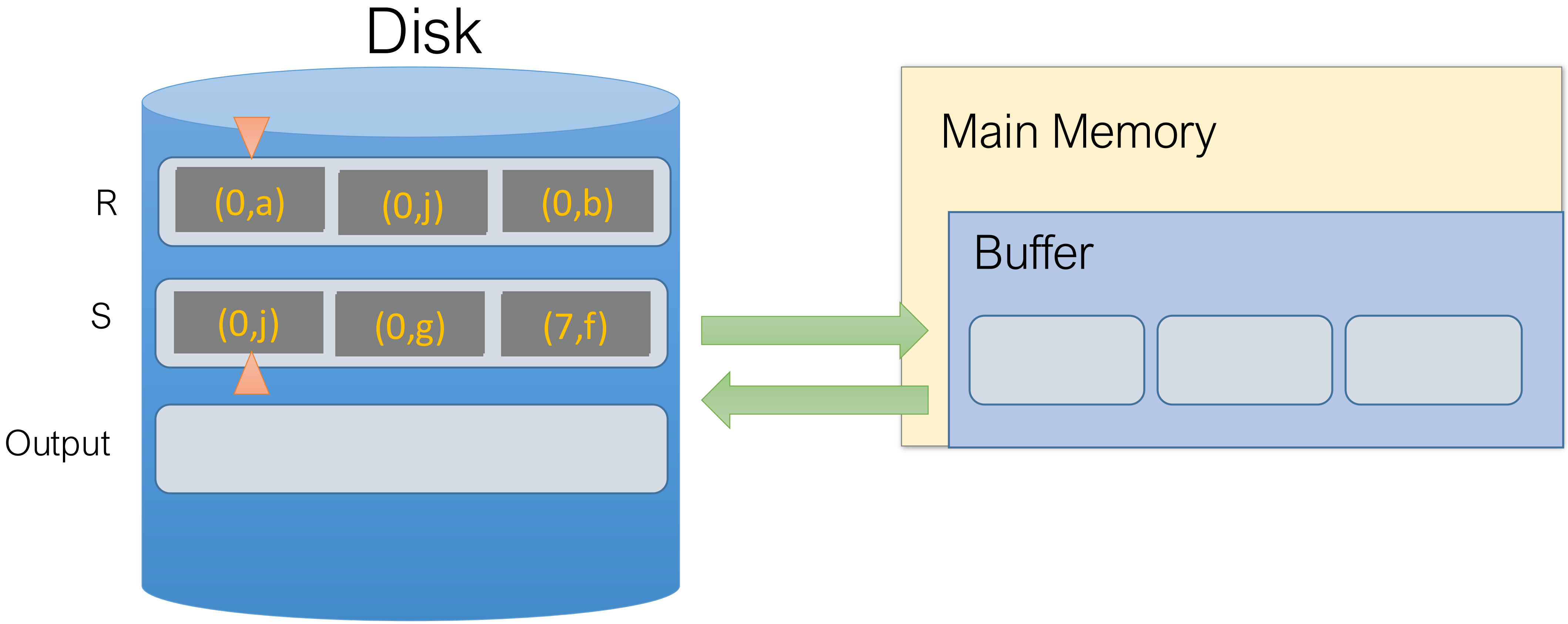
2. Done!



What happens with duplicate join keys?

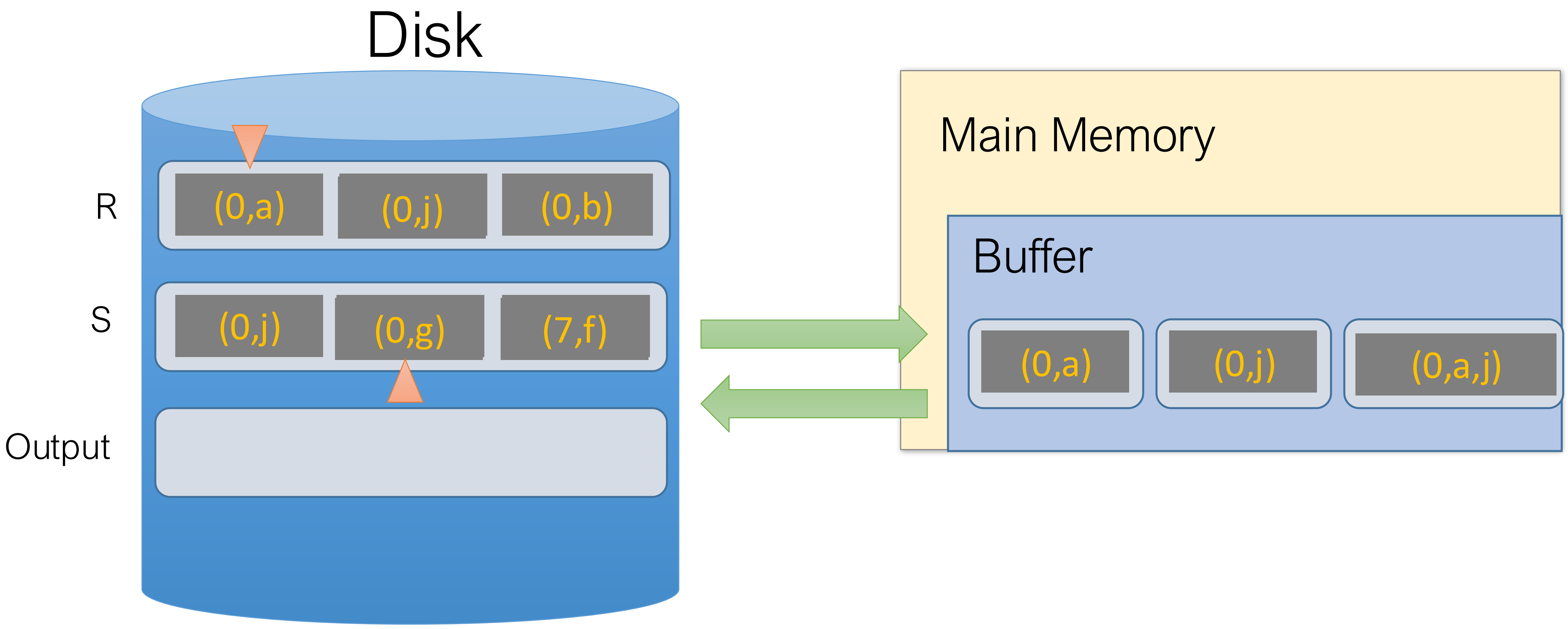
Multiple tuples with Same Join Key: “Backup”

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



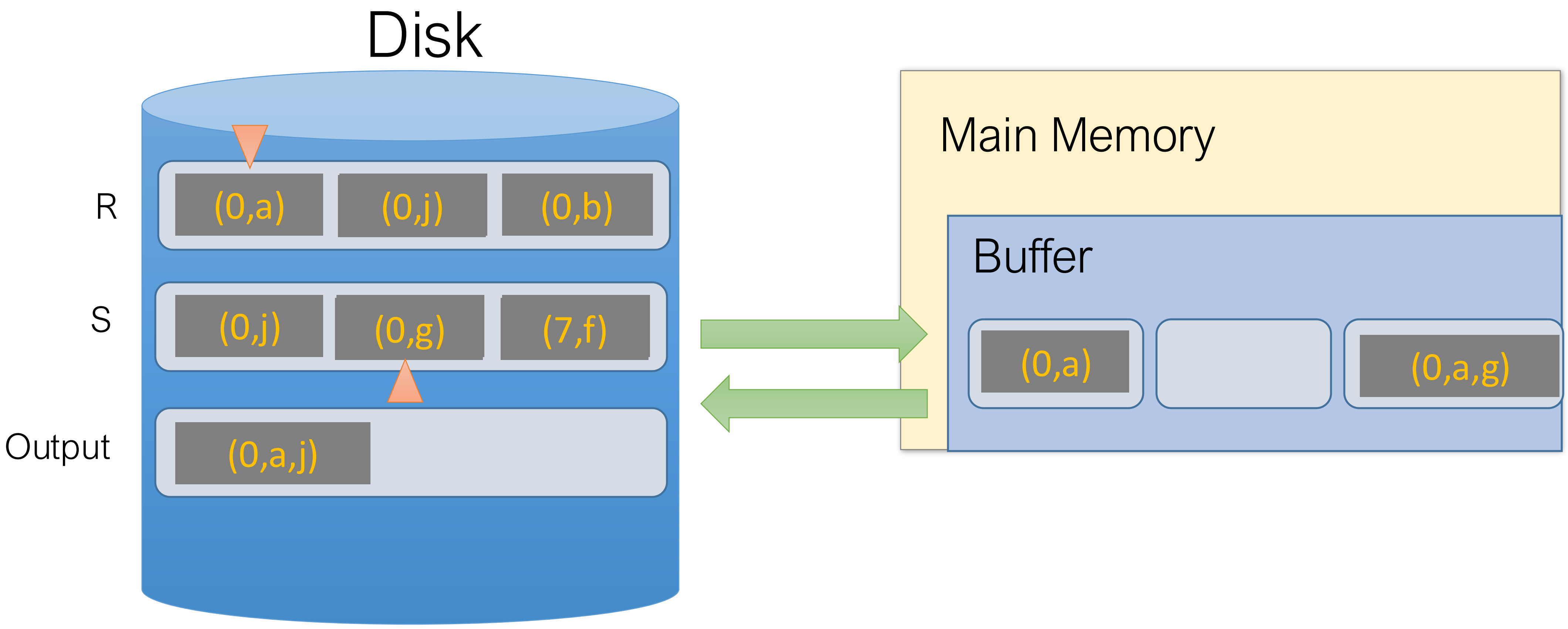
Multiple tuples with Same Join Key: “Backup”

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



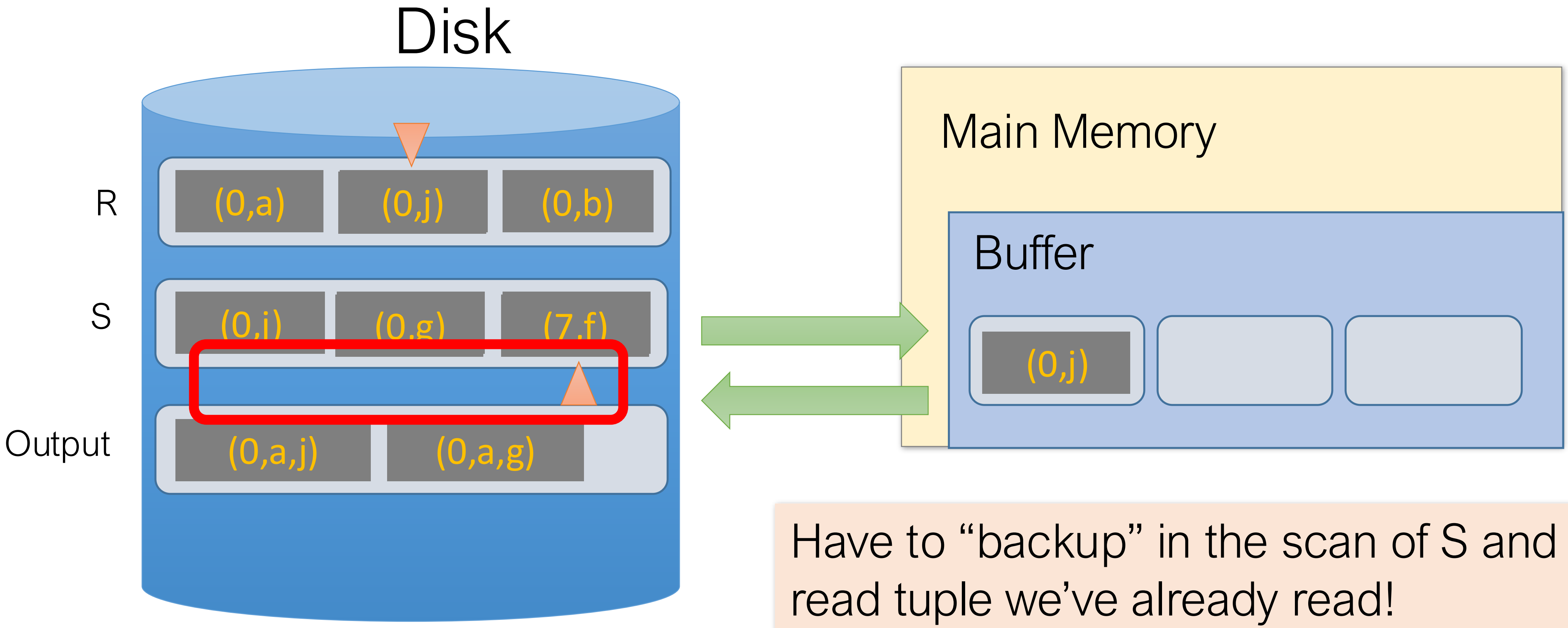
Multiple tuples with Same Join Key: "Backup"

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…



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…

What's the cost of sorting?

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**

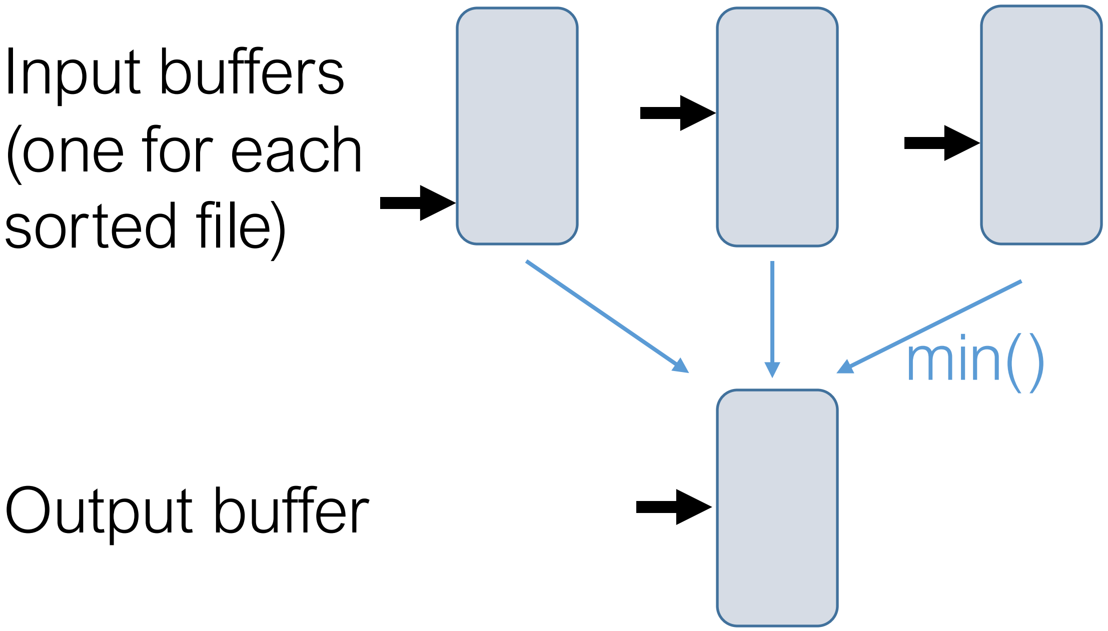
$$\sim \mathrm{Sort}(\mathrm{P}(\mathrm{R})) + \mathrm{Sort}(\mathrm{P}(\mathrm{S})) + \mathrm{P}(\mathrm{R}) + \mathrm{P}(\mathrm{S}) + \mathrm{OUT}$$

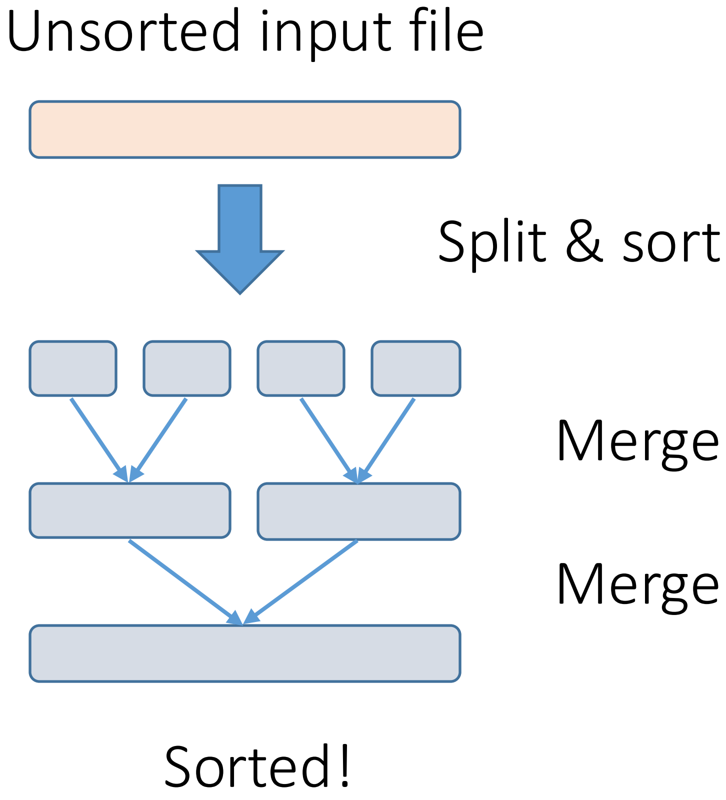
External Merge Sort

Given ***B+1*** buffer page

Phase 1. Split R into files small enough to sort in memory. Write sorted files to disk.

Phase 2. B-way merge of sorted files



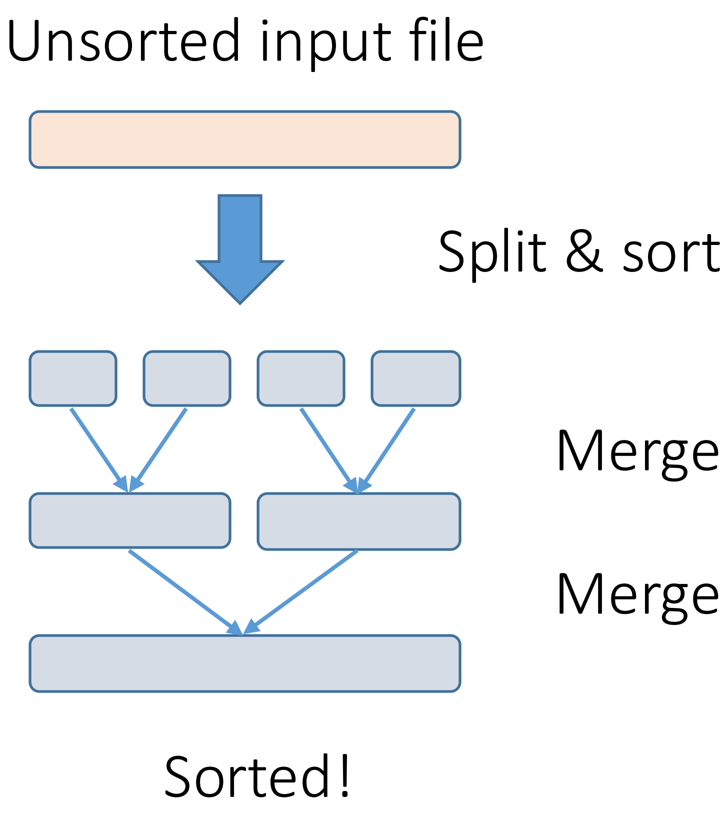


External Merge Sort

Given ***B+1*** buffer page

Phase 1. Split R into files small enough to sort in memory. Write sorted files to disk.

Phase 2. B-way merge of sorted files



IO costs:

- Phase 1: 1 Read and 1 Write per page = 2N IOs
- Phase 2: 1 Read per page = N IOs

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 + 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>

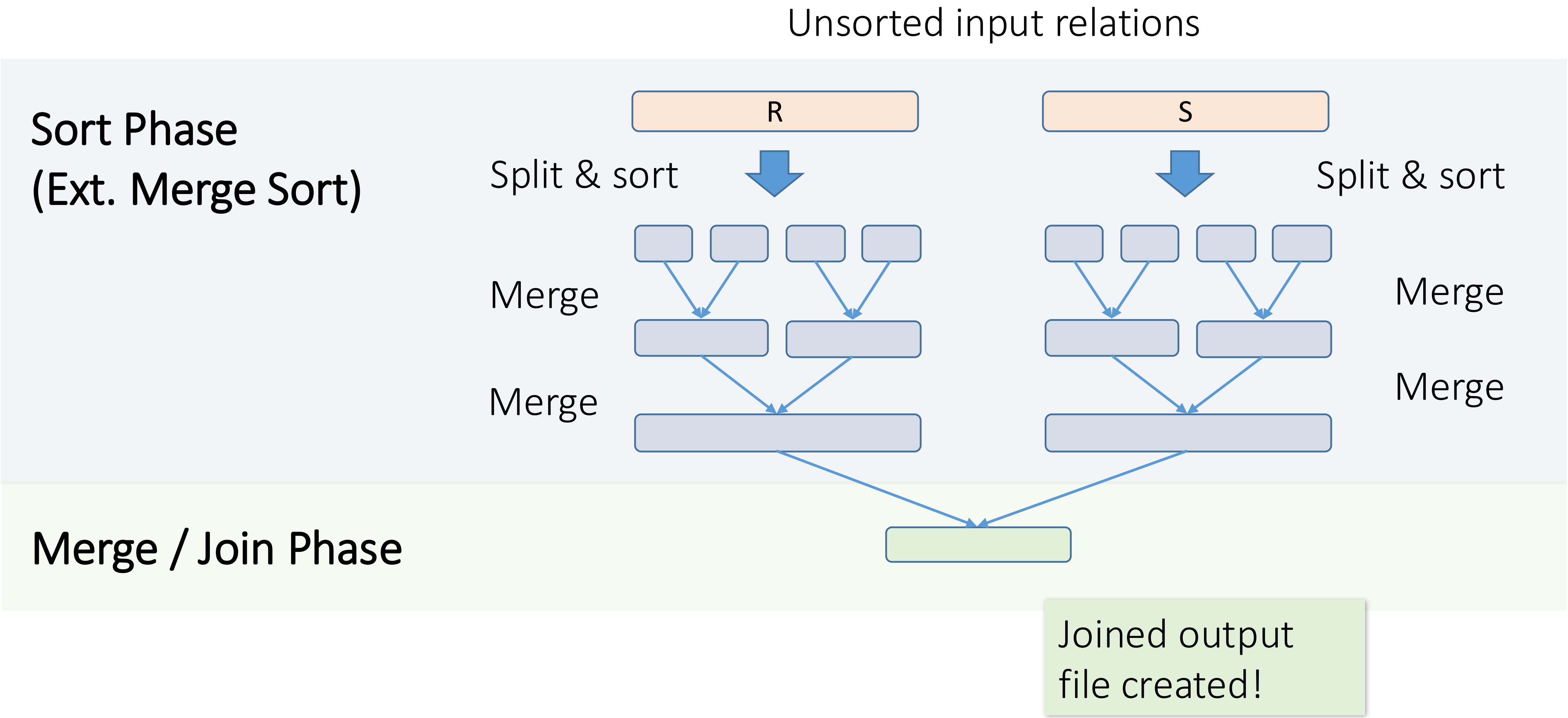
But, if we have 35 buffer pages?

- 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.

Un-Optimized SMJ

Given *B+1* buffer pages



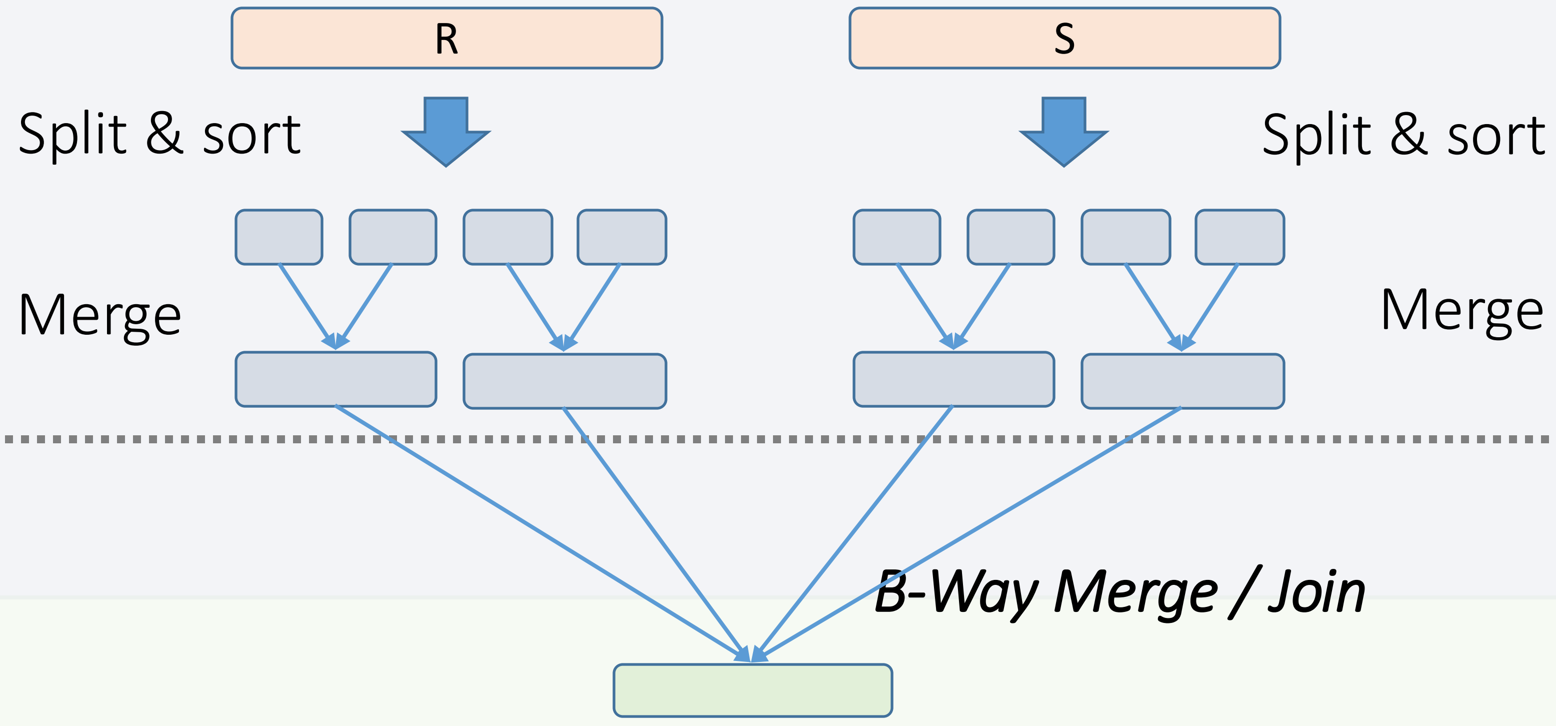
Simple SMJ Optimization

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

Unsorted input relations

Sort Phase
(Ext. Merge Sort)

<= B total sorted files



Merge / Join Phase

Joined output
file created!

Simple SMJ Optimization

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

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

We are saving two disk I/O's per block by combining the second phase of the sorts with the join itself. ***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?

- $\frac{P(R)+P(S)}{B} \leq B$
- Thus, $\mathbf{max}\{\mathbf{P(R)}, \mathbf{P(S)}\} \leq \boldsymbol{B^2}$ 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.

SMJ needs to sort **both** relations

- If $\max \{ P(R), P(S) \} < B^2$ then cost is $3(P(R)+P(S)) + OUT$

3. Hash Join (HJ)

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 != y but $h_B(x) = h_B(y)$
  - Note however that it will <u>**never**</u> occur that x = y but $h_B(x) \neq h_B(y)$
- We hash on an attribute A, so our has function is $h_B(t)$ has the form $h_B(t.A)$.
  - **Collisions** may be more frequent.

Hash Join: High-level procedure

Given *B+1* buffer pages

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

1. **Partition Phase:** Using one (shared) hash function h_B per pass partition R *and* S into *B* buckets.
   - Each phase creates B more buckets that are a factor of B smaller.
   - Repeatedly partition with a new hash function
   - Stop when all buckets for one relation are smaller than B-1 pages

Each pass takes $2(\mathrm{P(R)} + \mathrm{P(S)})$

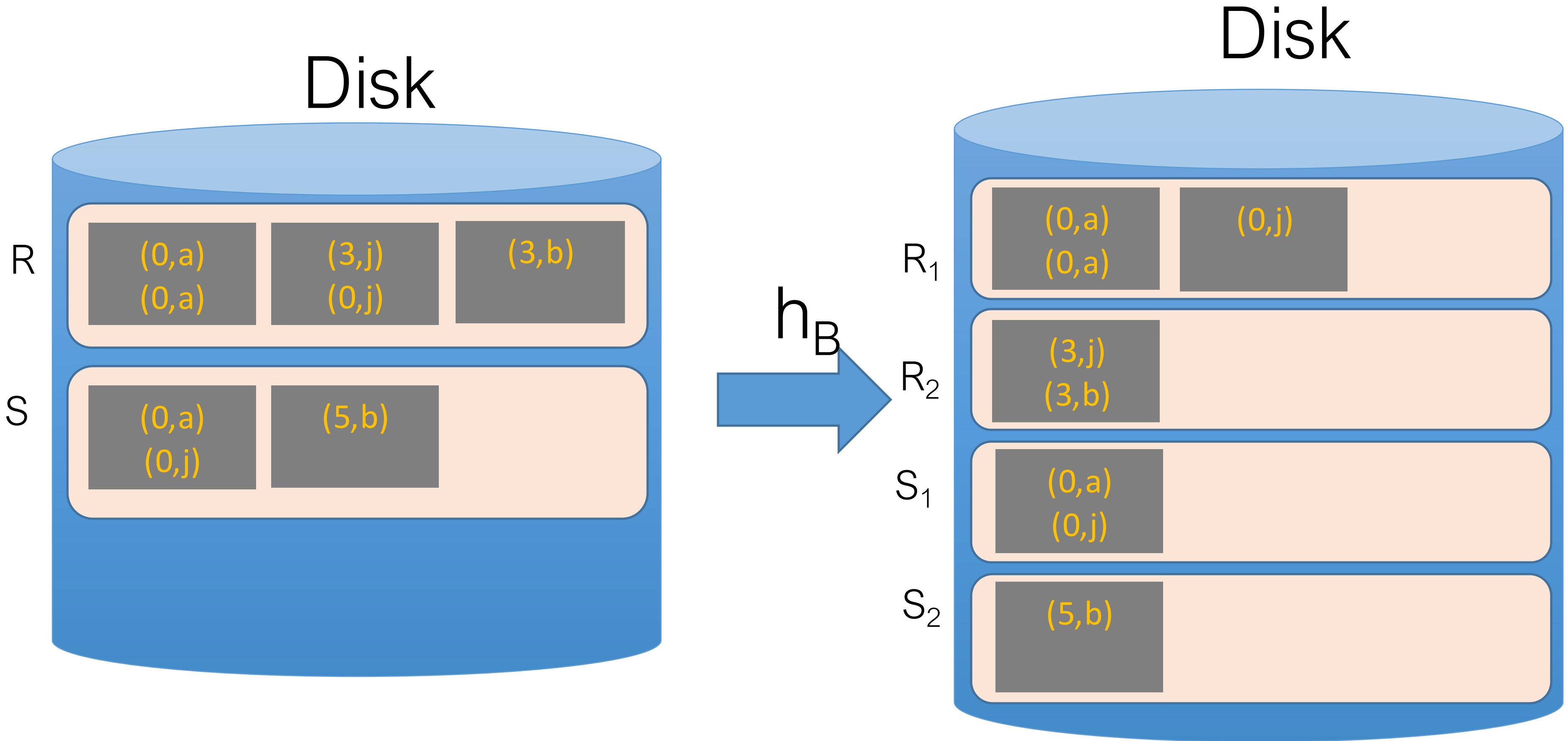
2. **Matching Phase:** Take pairs of buckets whose tuples have the same values for *h*, and join these
   - Use BNLJ here for each matching pair.

$\mathrm{P(R)} + \mathrm{P(S)} + \mathrm{OUT}$

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

Hash Join: High-level procedure

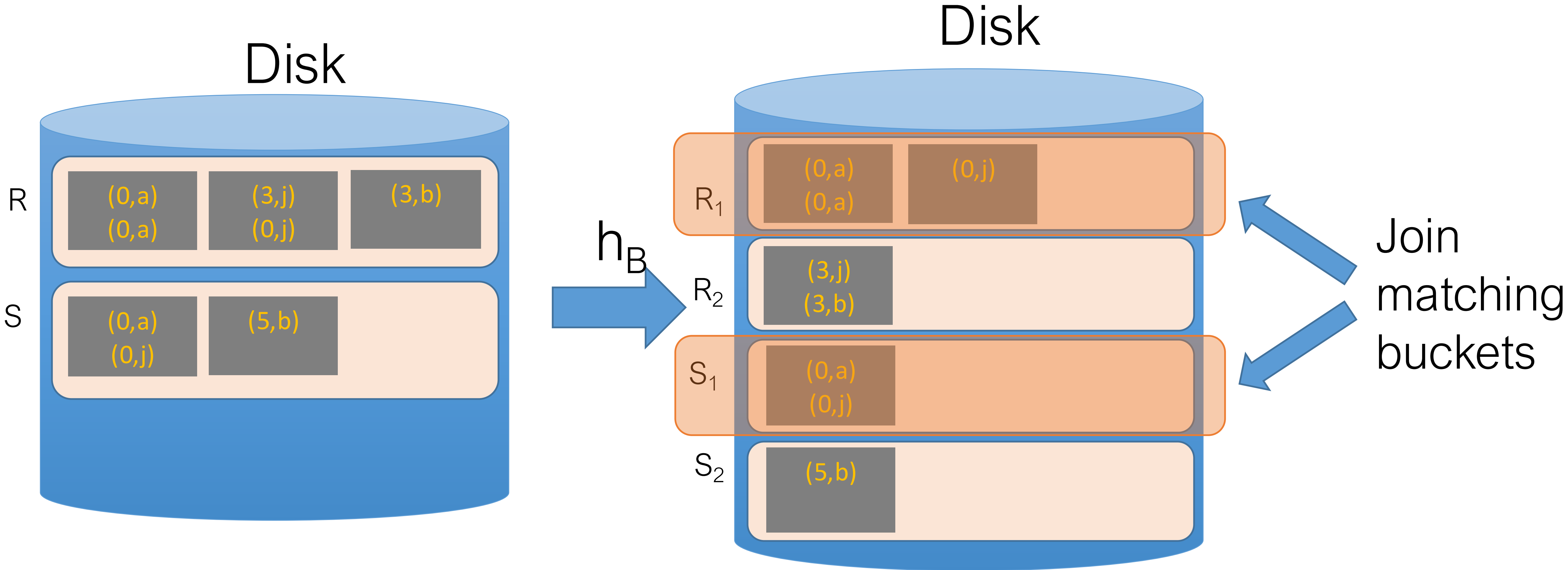
1. Partition Phase: Using one (shared) hash function h_B, partition R *and* S into ***B*** buckets



Suppose each pages has two tuples (one per row)

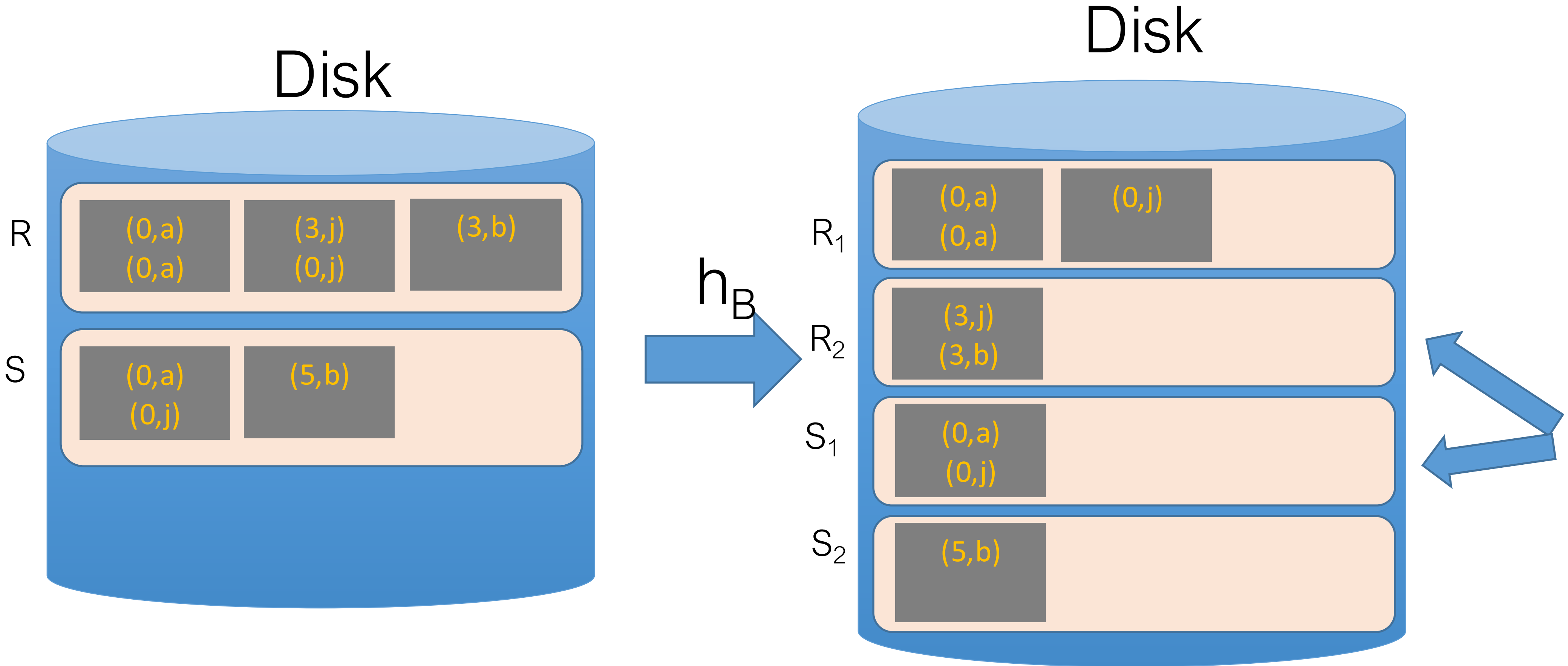
Hash Join: High-level procedure

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



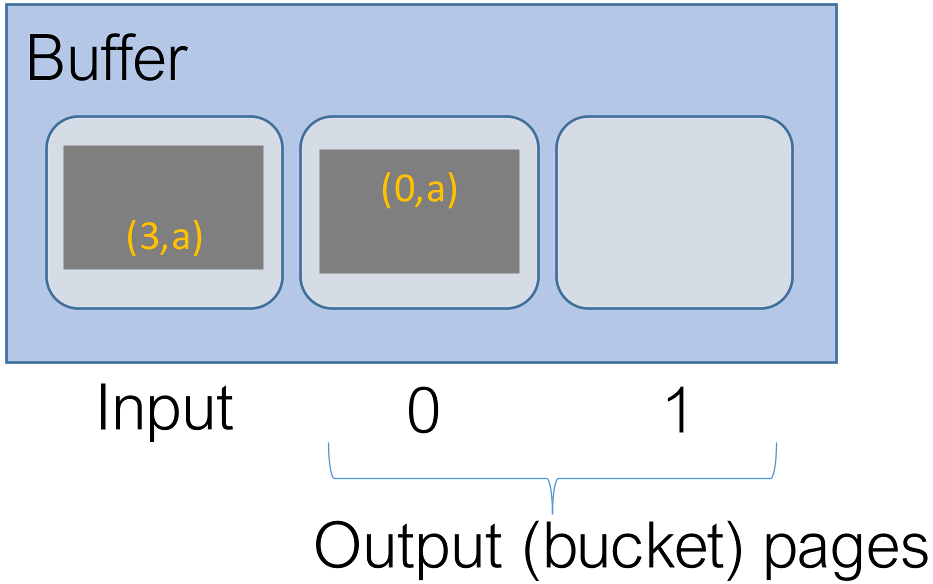
Don't have to join the others! E.g. (S_1 and R_2)!

Hash Join Phase 1: Partitioning

Goal: For each relation, partition relation into **buckets** such that if $h_B(t.A) = h_B(t'.A)$ 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



How big *do we want* the resulting buckets?

Given B+1 buffer pages

Ideally, our buckets would be of size $\leq \boldsymbol{B-1}$ **pages**

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),* $\boldsymbol{P(R) \leq B-1}$!

- And more generally, being able to fit bucket in memory is advantageous

Recall for BNLJ:

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

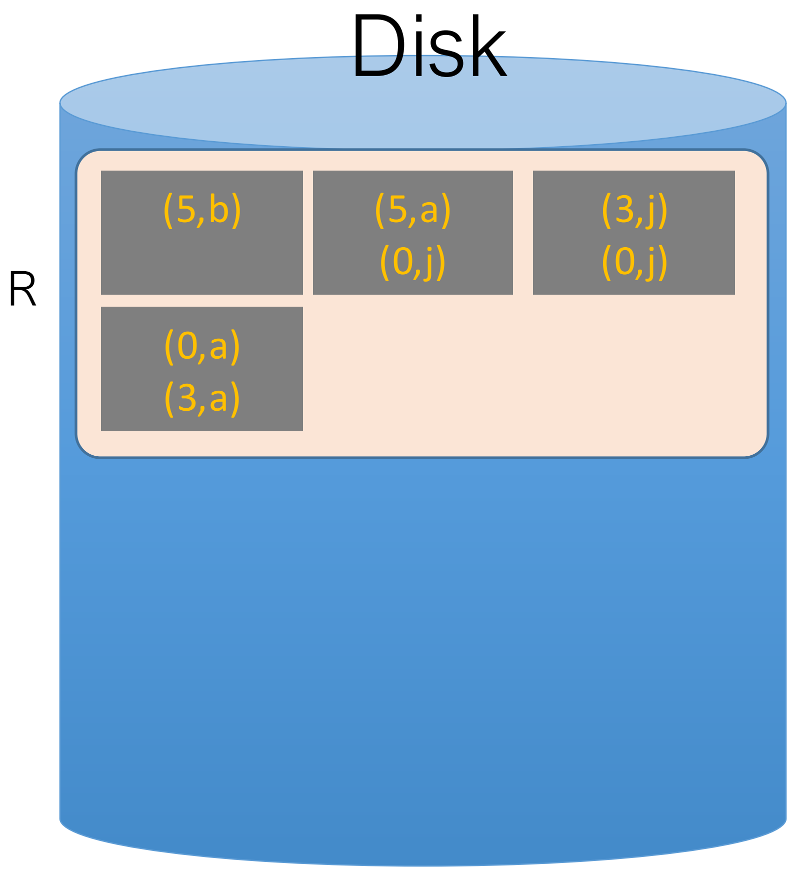
- We can keep partitioning buckets until they are $\leq \boldsymbol{B-1}$ **pages**
  - Using a new hash key which will split them…

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)



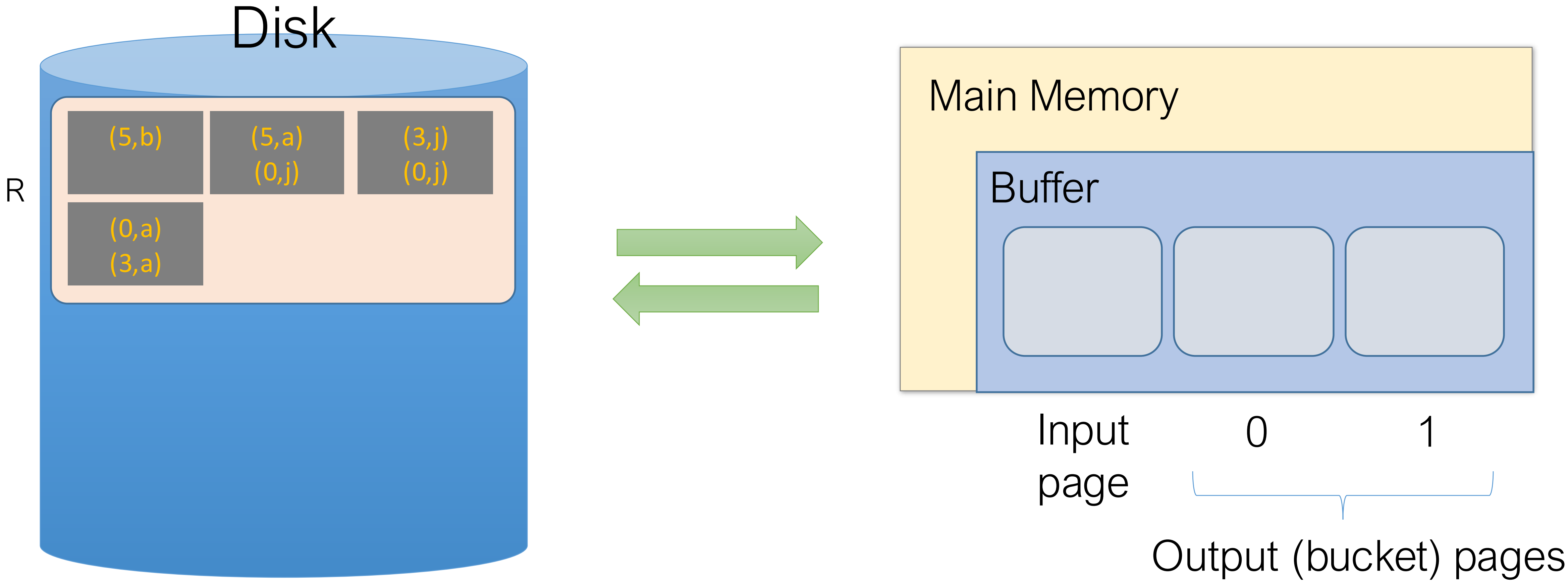
For simplicity, we'll look at partitioning one of the two relations- we just do the same for the other relation!

Recall: our goal will be to get B = 2 buckets of size <= B-1 → 1 page each

Hash Join Phase 1: Partitioning

Given B+1 = 3 buffer pages

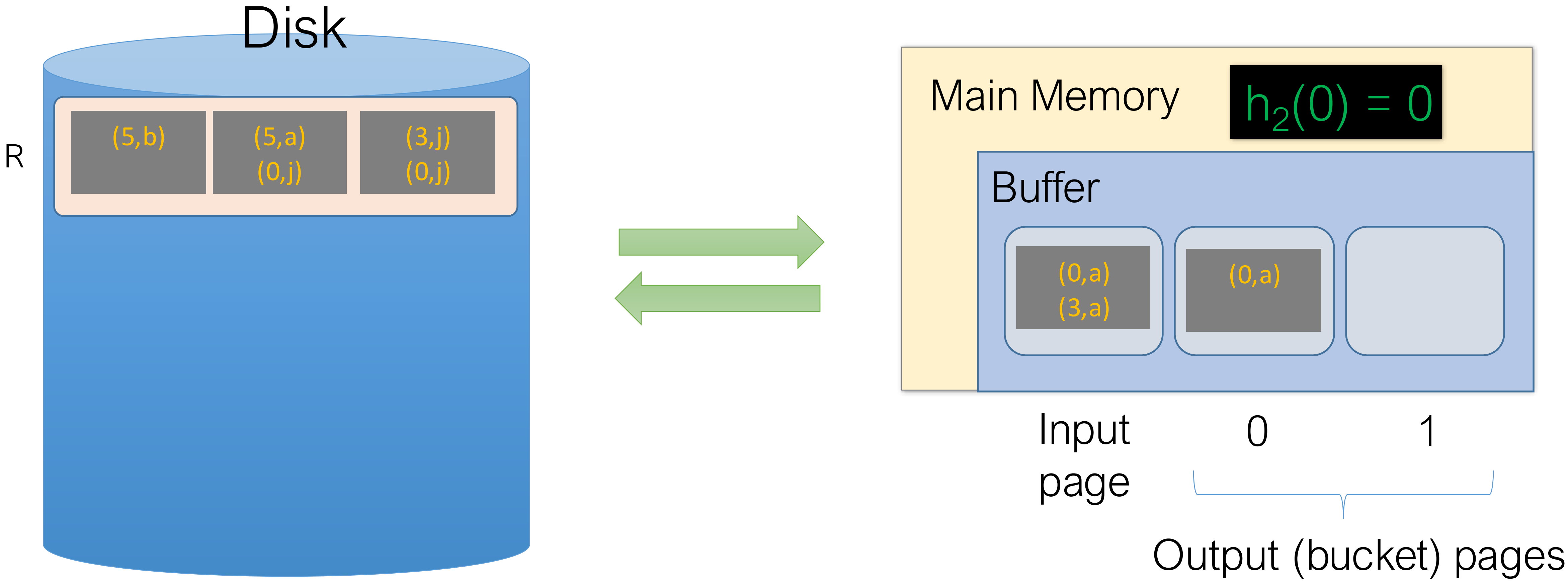
1. We read pages from R into the “input” page of the buffer…



Hash Join Phase 1: Partitioning

Given B+1 = 3 buffer pages

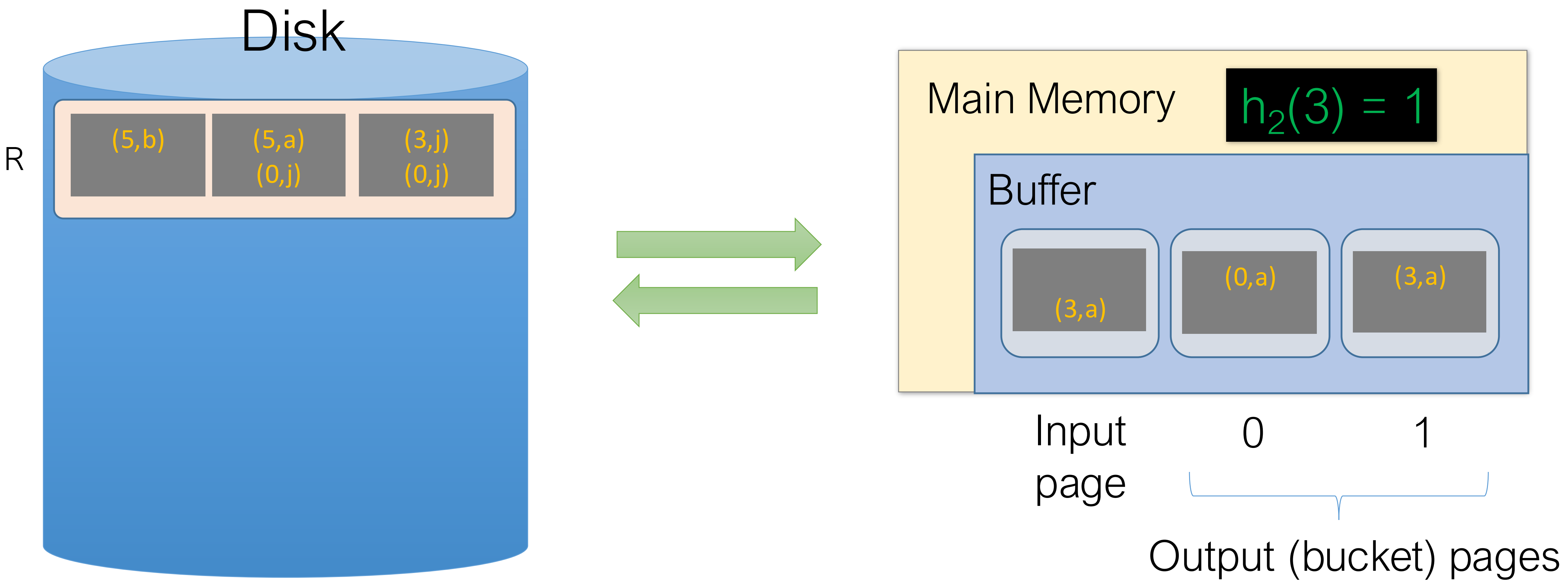
2. Then we use **hash function h_2** to sort into the buckets, which each have one page in the buffer



Hash Join Phase 1: Partitioning

Given B+1 = 3 buffer pages

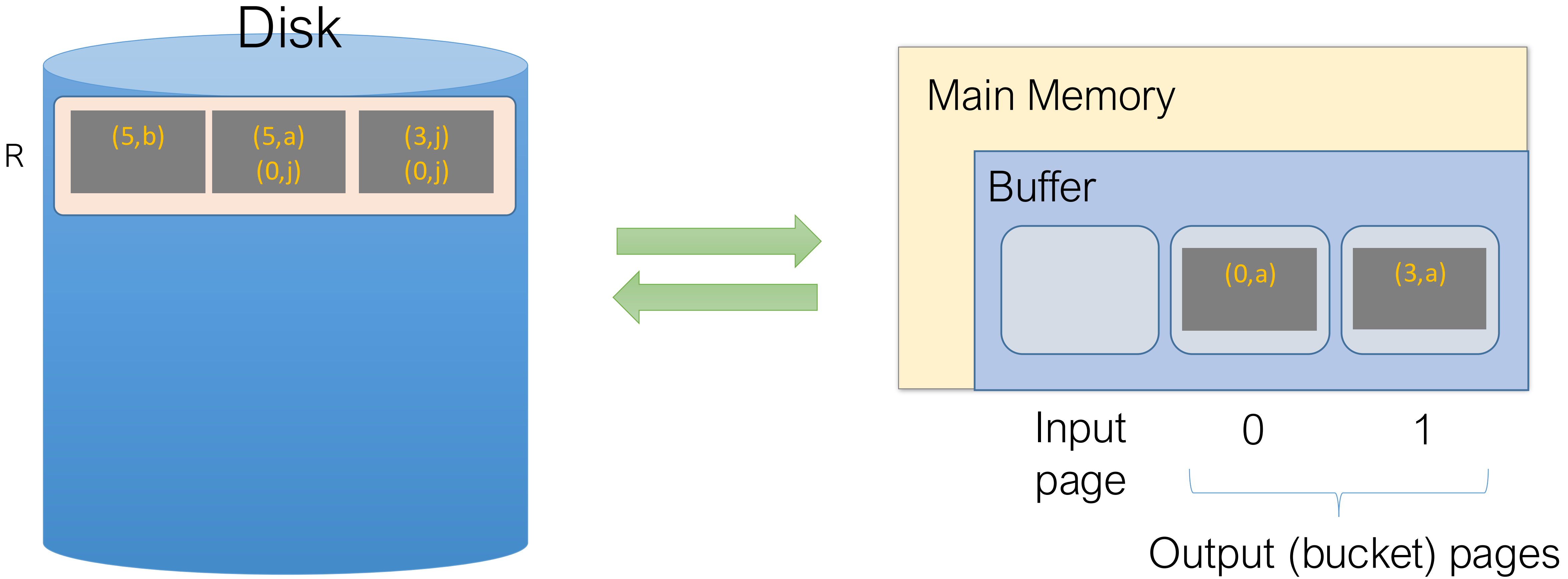
2. Then we use hash function h_2 to sort into the buckets, which each have one page in the buffer



Hash Join Phase 1: Partitioning

Given B+1 = 3 buffer pages

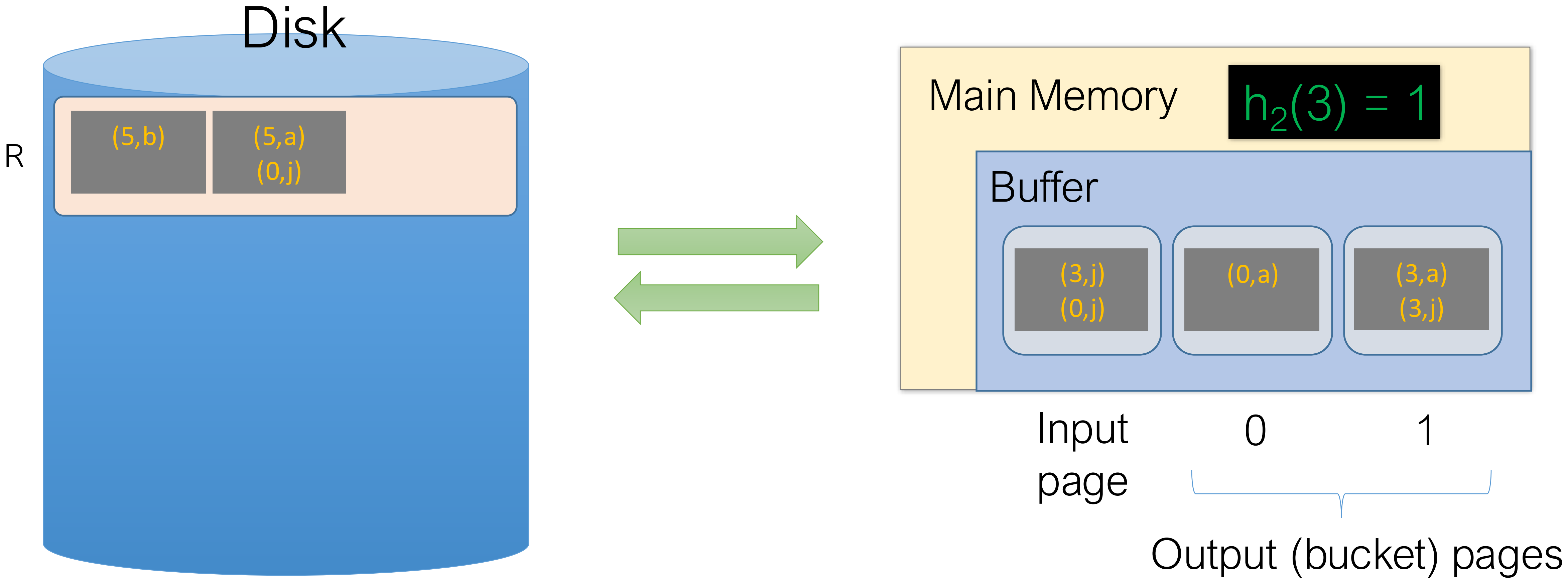
3. We repeat until the buffer bucket pages are full…



Hash Join Phase 1: Partitioning

Given B+1 = 3 buffer pages

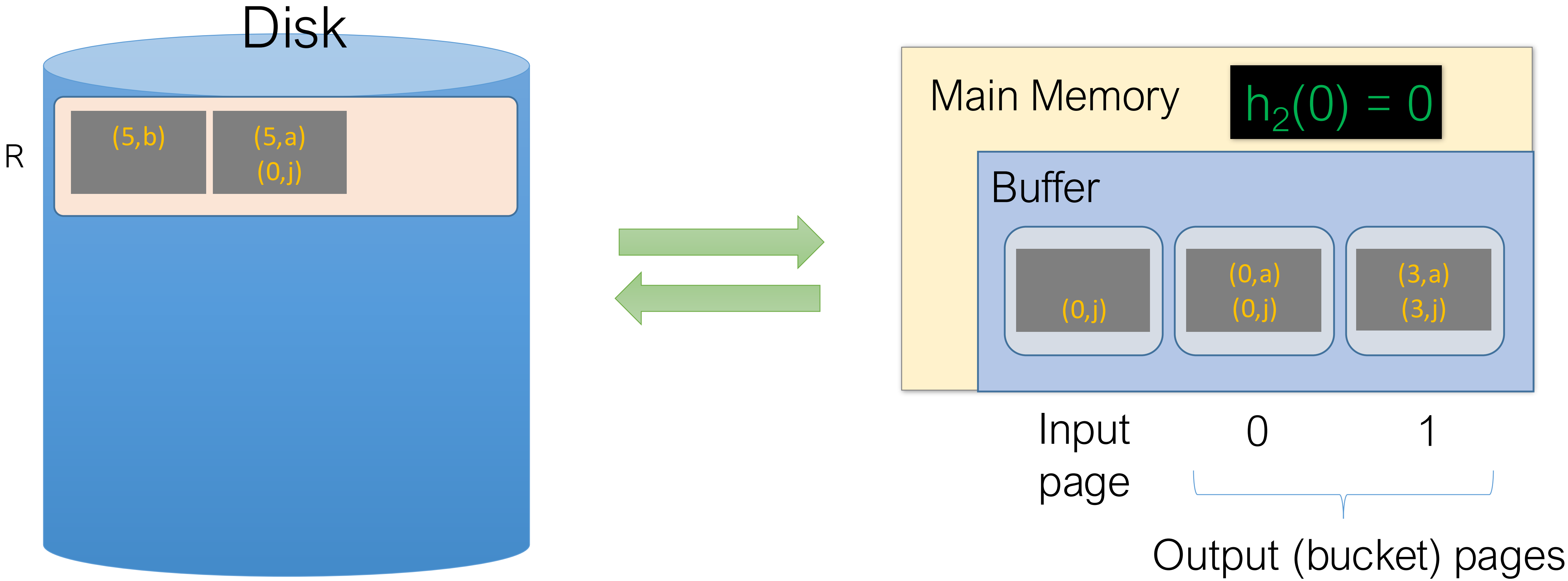
3. We repeat until the buffer bucket pages are full…



Hash Join Phase 1: Partitioning

Given B+1 = 3 buffer pages

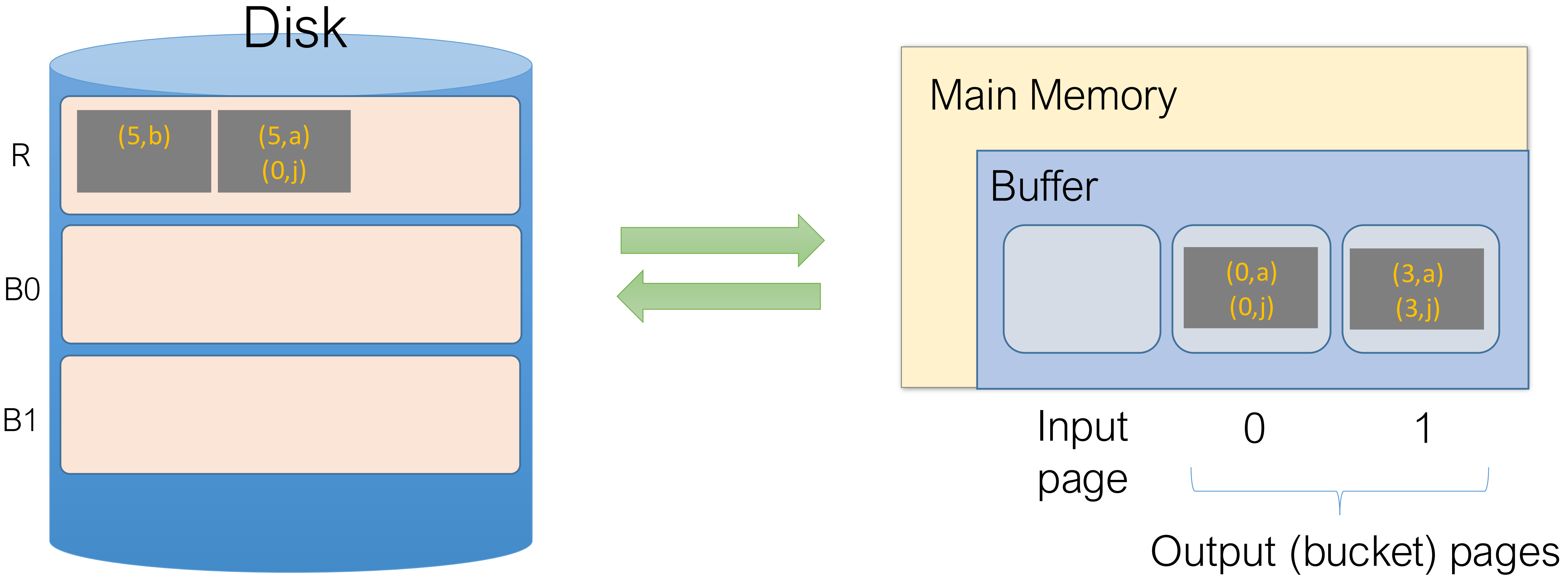
3. We repeat until the buffer bucket pages are full…



Hash Join Phase 1: Partitioning

Given B+1 = 3 buffer pages

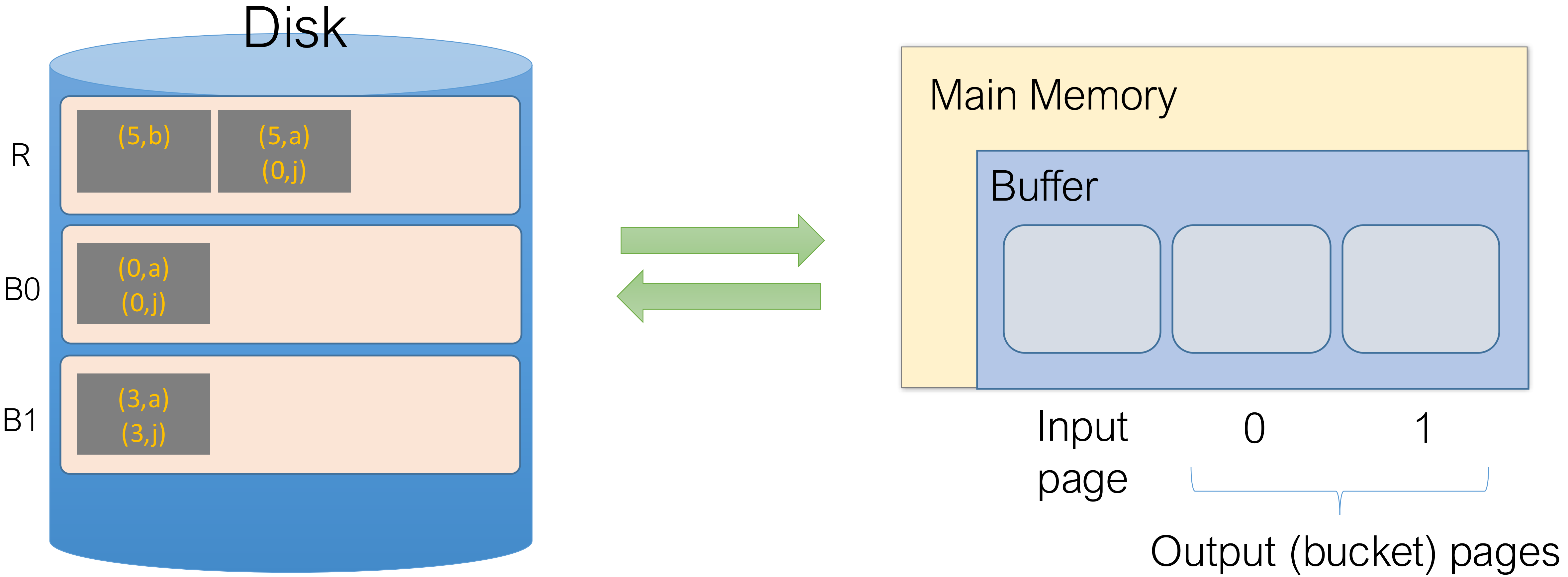
3. We repeat until the buffer bucket pages are full… then flush to disk



Hash Join Phase 1: Partitioning

Given B+1 = 3 buffer pages

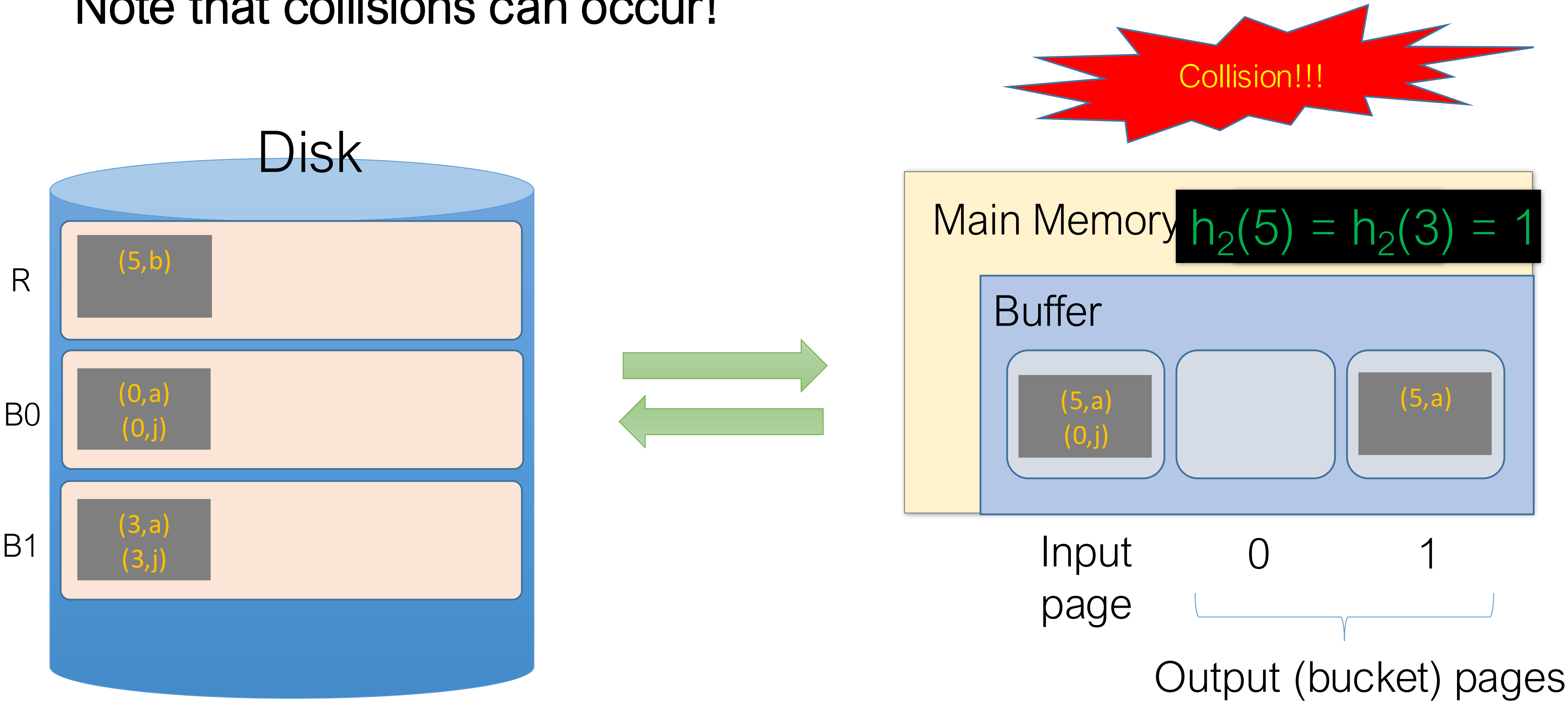
3. We repeat until the buffer bucket pages are full… then flush to disk



Hash Join Phase 1: Partitioning

Given B+1 = 3 buffer pages

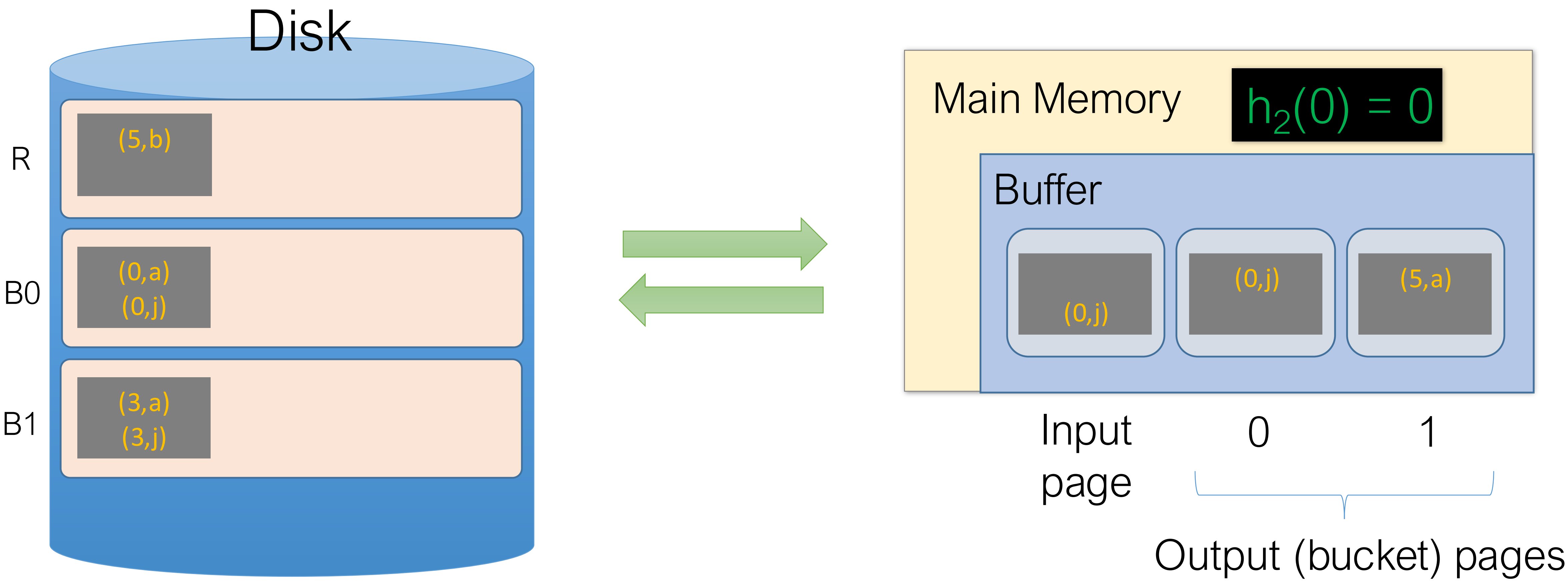
Note that collisions can occur!



Hash Join Phase 1: Partitioning

Given B+1 = 3 buffer pages

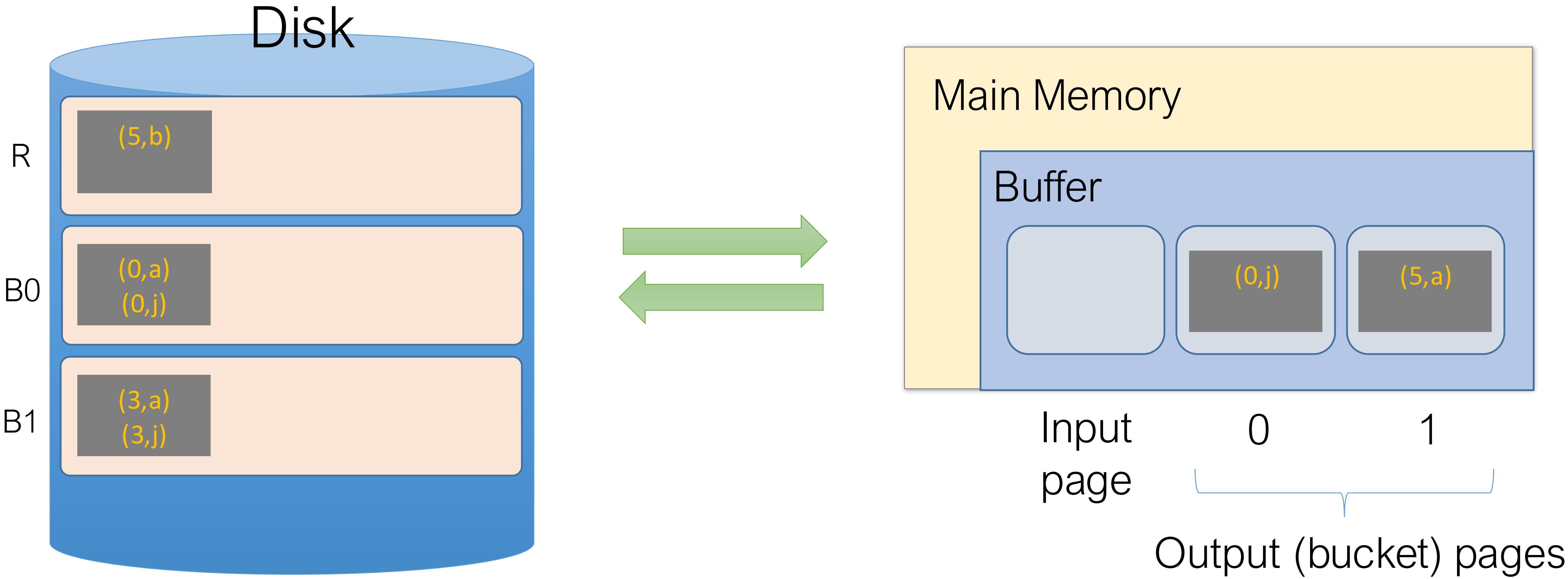
Finish this pass…



Hash Join Phase 1: Partitioning

Given B+1 = 3 buffer pages

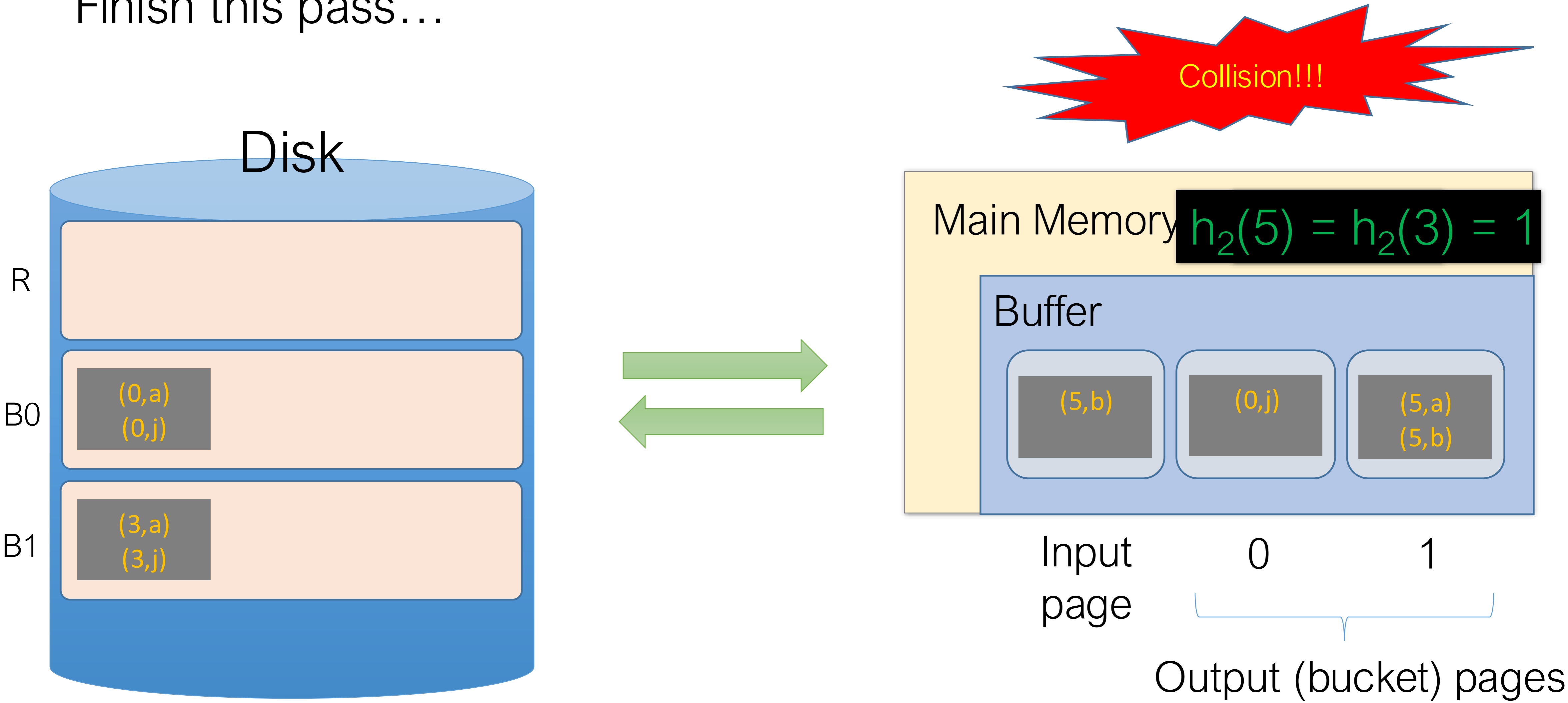
Finish this pass…



Hash Join Phase 1: Partitioning

Given B+1 = 3 buffer pages

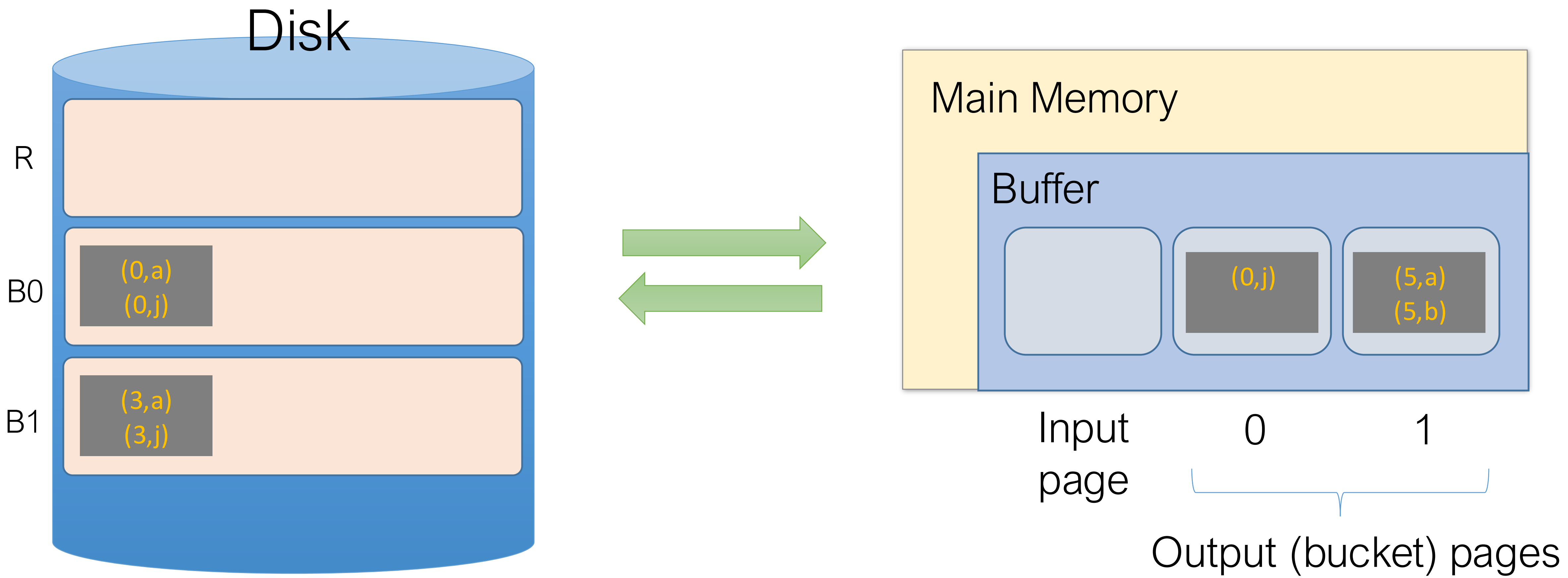
Finish this pass…



Hash Join Phase 1: Partitioning

Given B+1 = 3 buffer pages

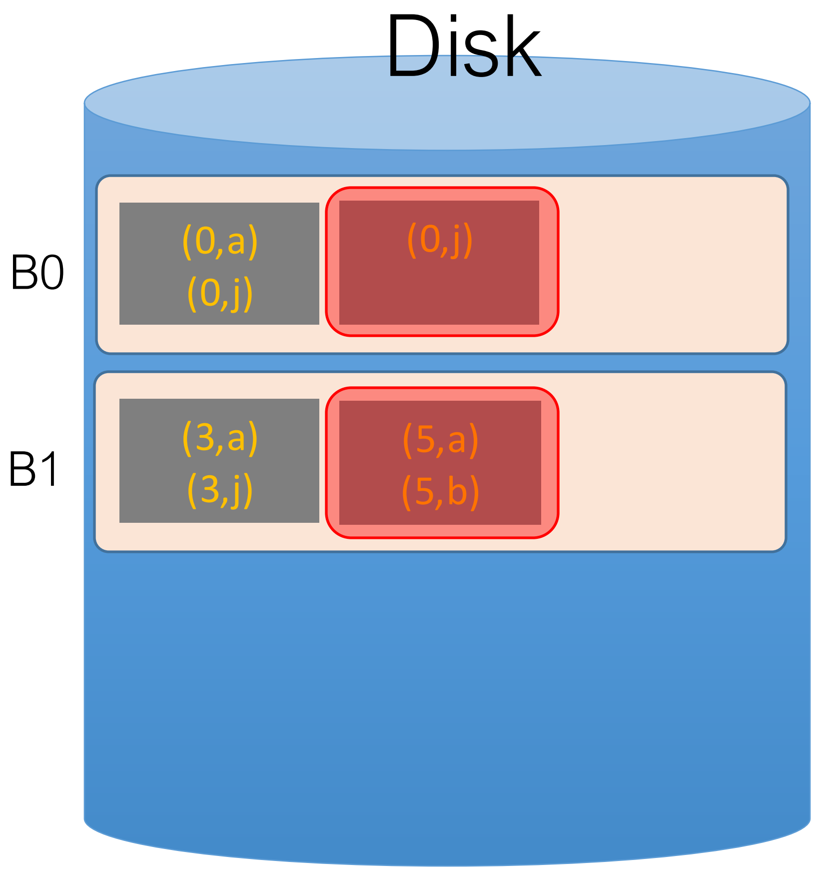
Finish this pass…



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:

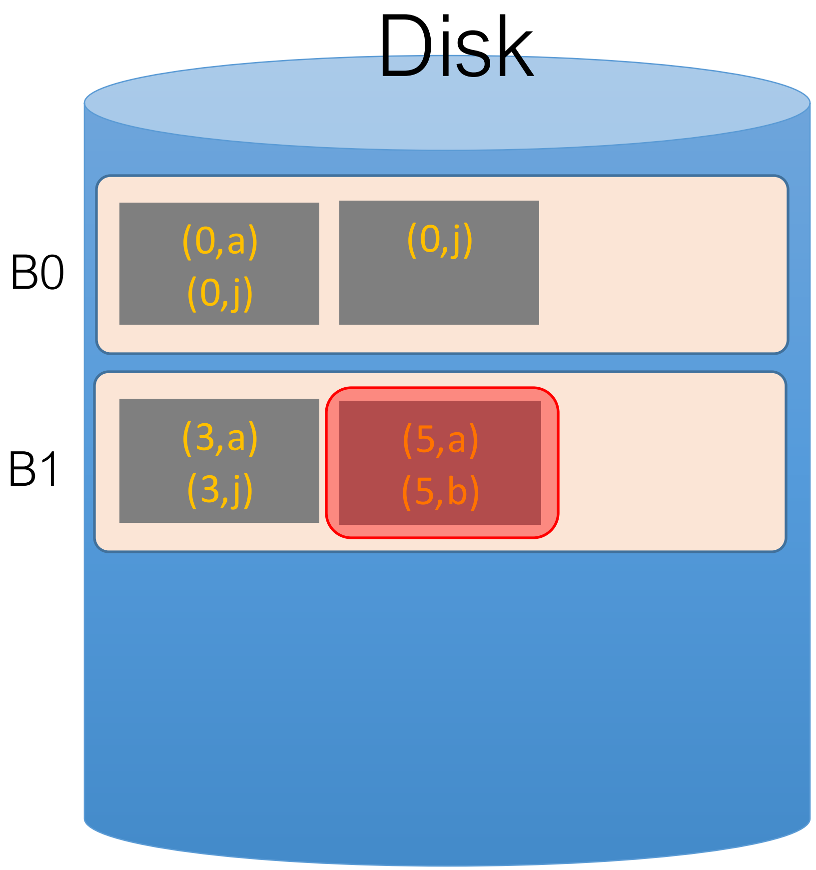


(1) Duplicate join keys

(2) Hash collisions

Hash Join Phase 1: Partitioning

Given B+1 = 3 buffer pages



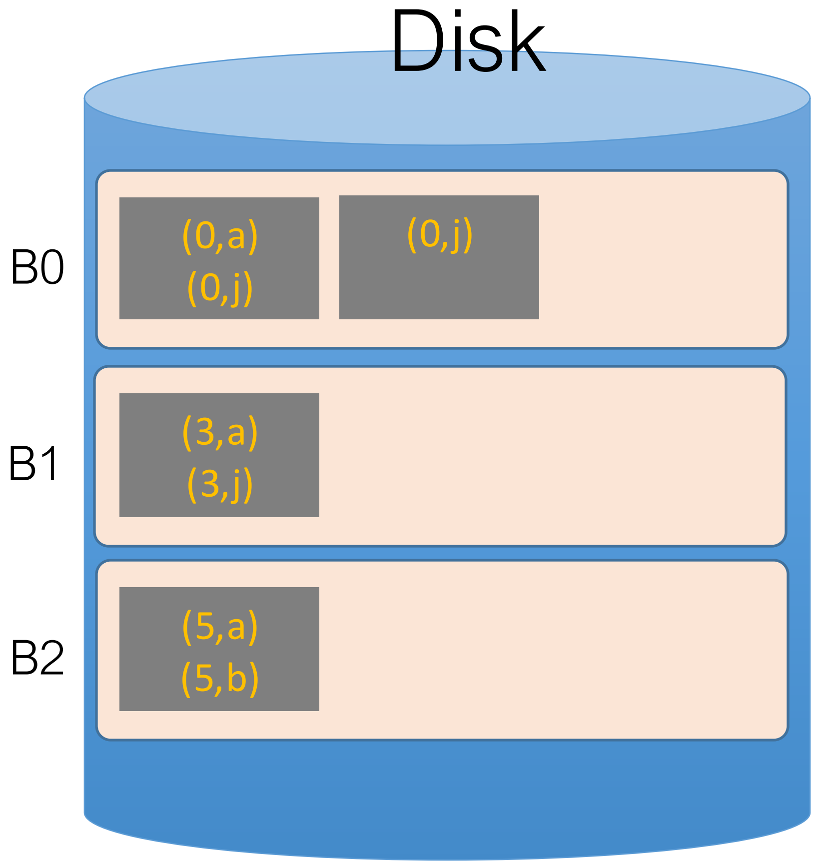
To take care of larger buckets caused by (2) hash collisions, we can just do another pass!

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

$$h'_2(3) \mathrel{!=} h'_2(5)$$

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!

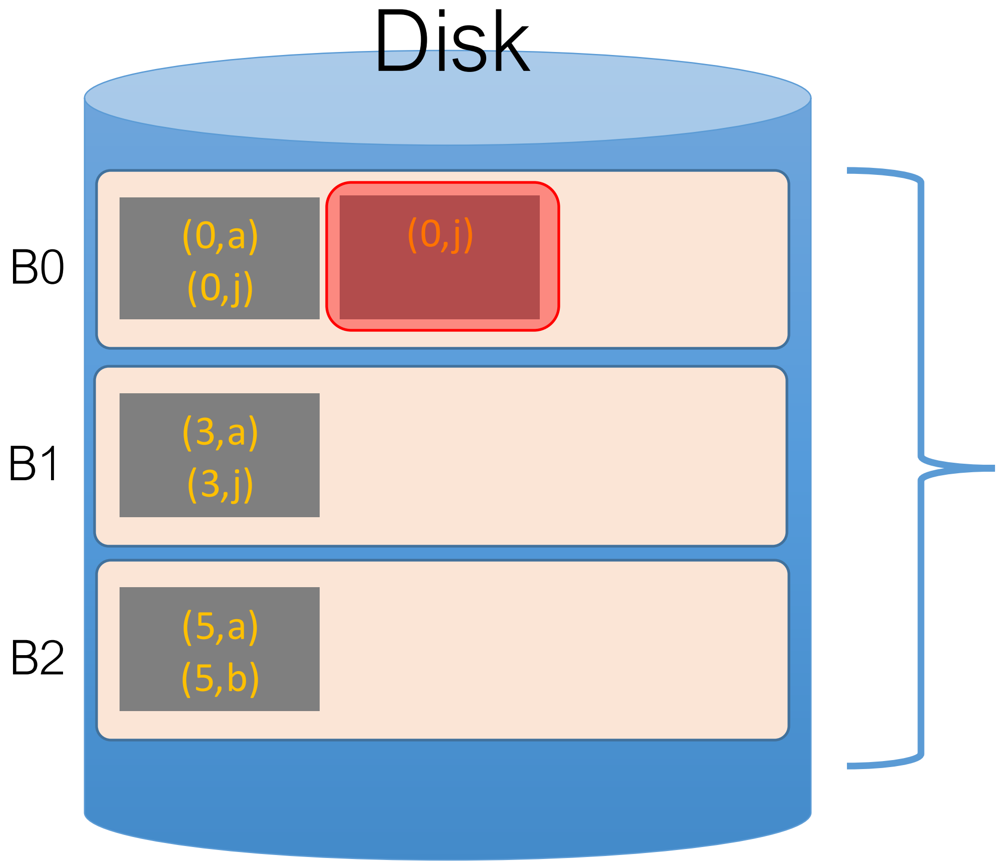
Do another pass with a different hash function, h'_2, ideally such that:

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

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.

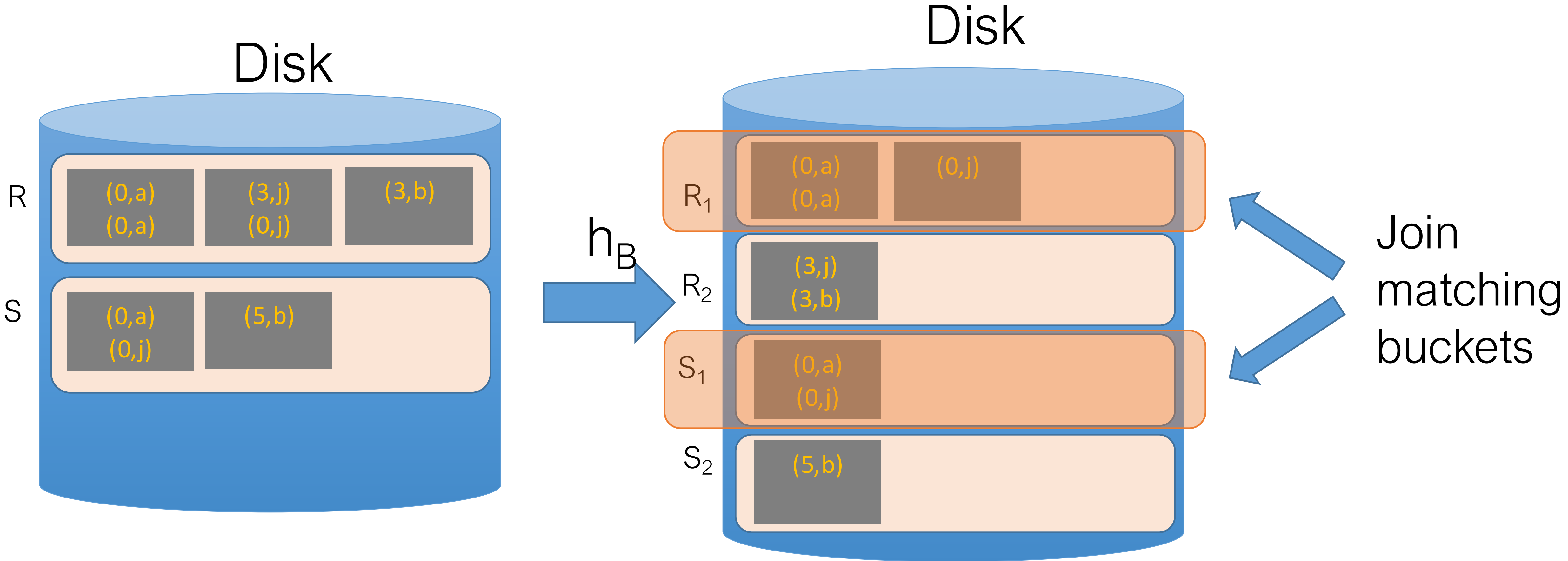


We call this unevenness in the bucket size <u>skew</u>

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 → 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

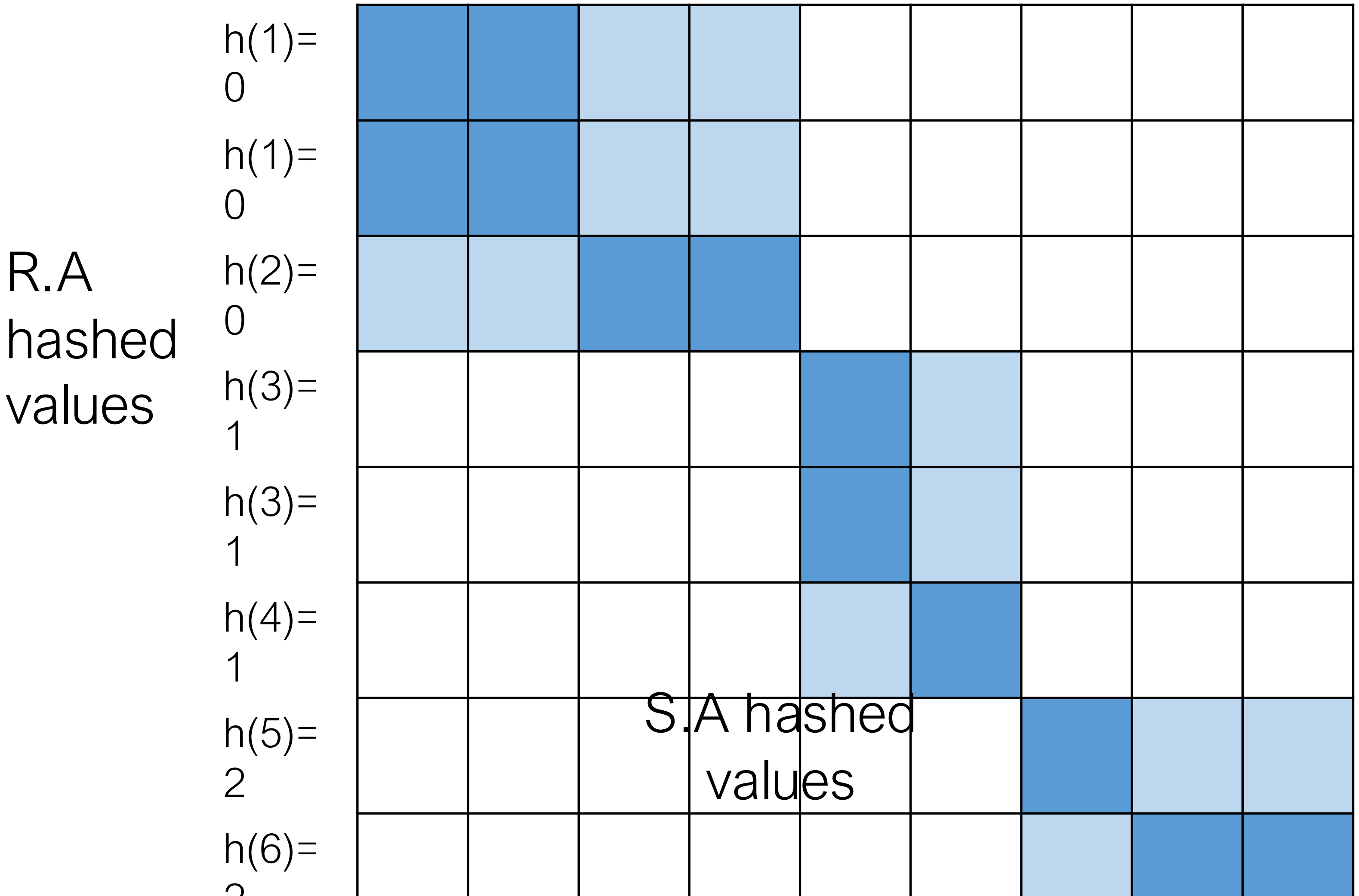
- If our buckets are $\sim \boldsymbol{B-1}$ **pages,** can join each such pair using BNLJ *in linear time*; recall (with P(R) = B-1):

<u>BNLJ Cost:</u> $\mathrm{P}(R) + \frac{P(R)P(S)}{B-1} = P(R) + \frac{(B-1)P(S)}{B-1}$ = P(R) + P(S)

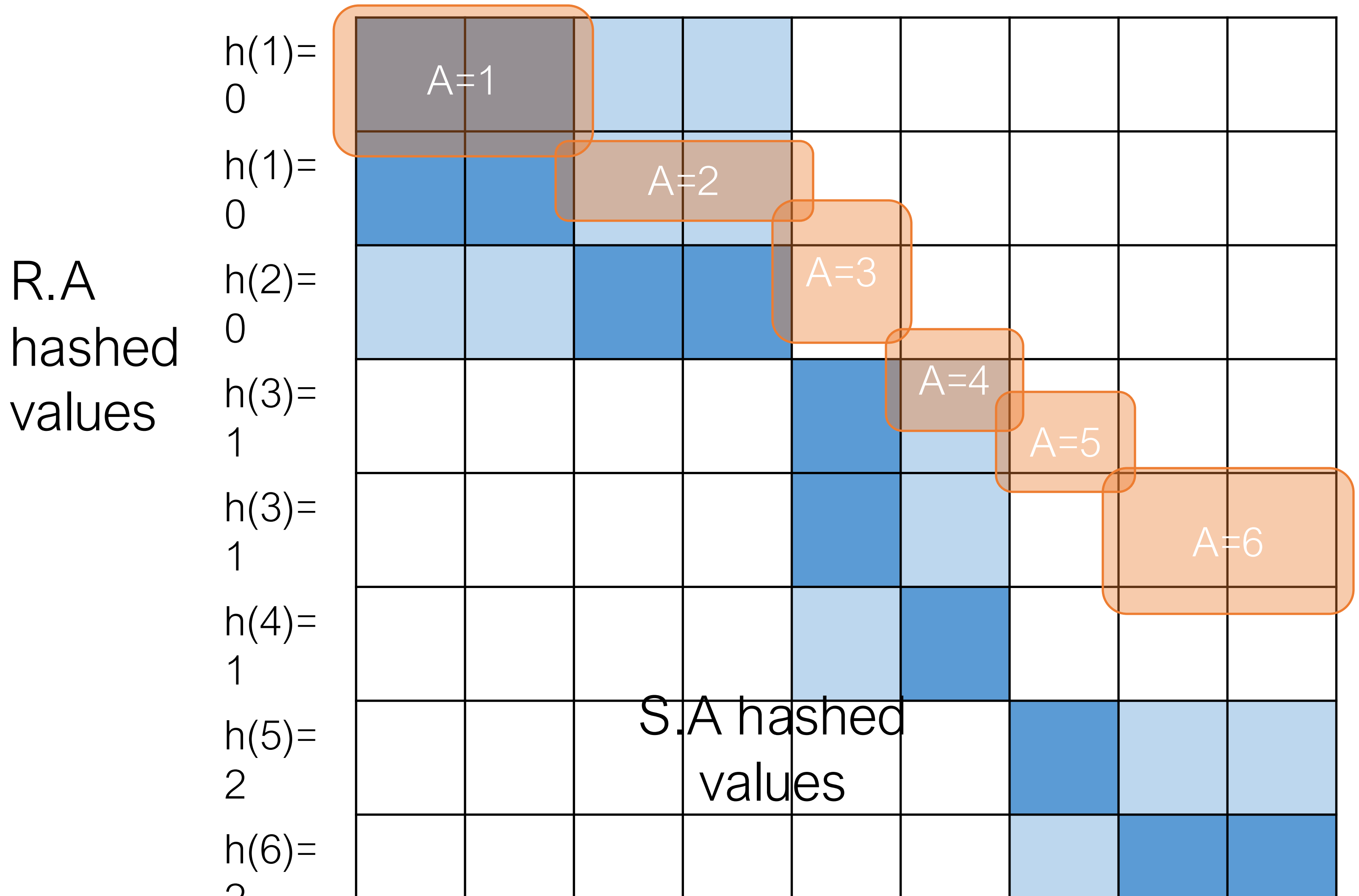
Joining the pairs of buckets is linear!
(As long as smaller bucket <= B-1 pages)

Hash Join Phase 2: Matching

$R \bowtie S \ on\ A$



Hash Join Phase 2: Matching

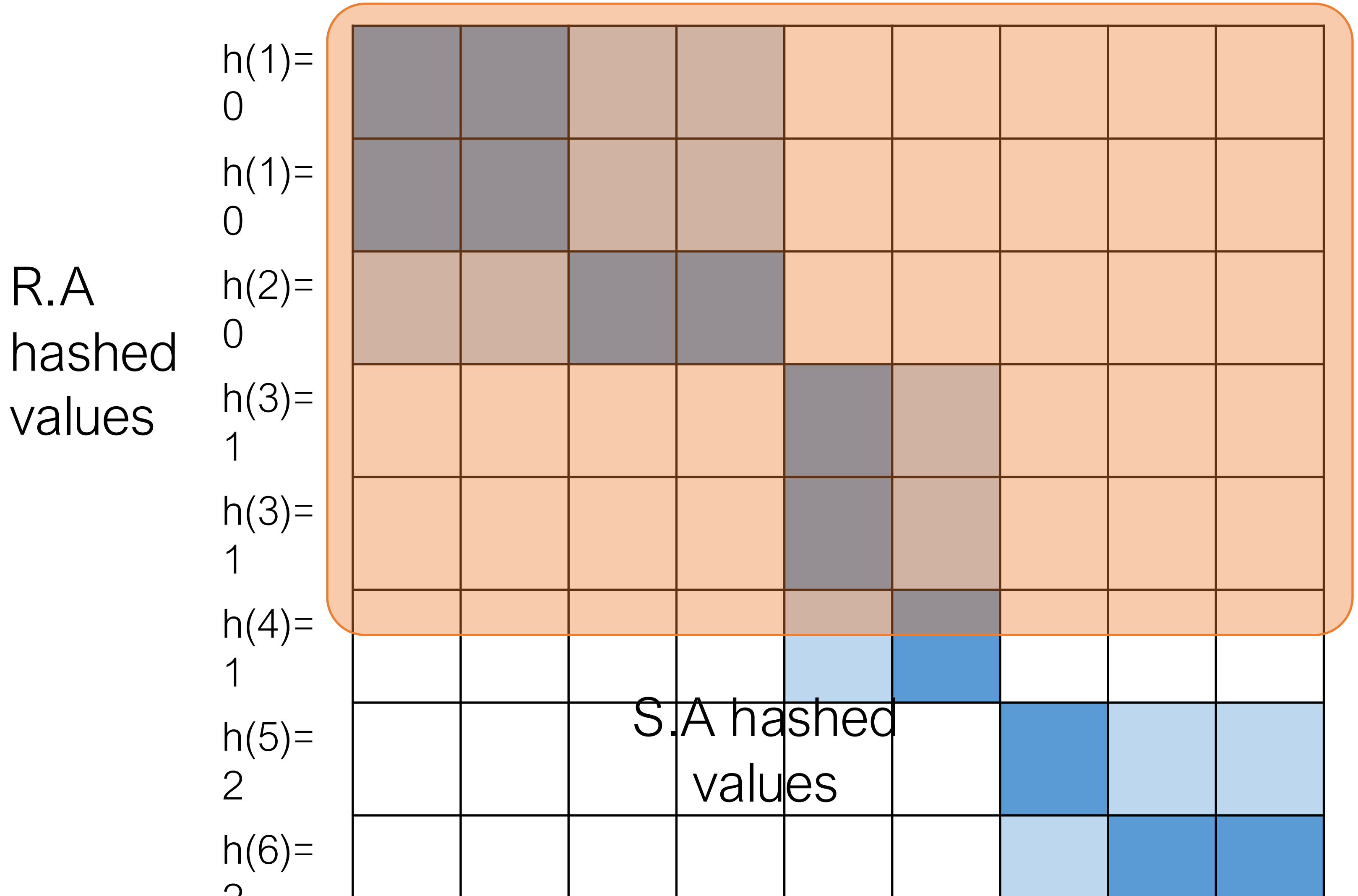


$R \bowtie S \text{ on } A$

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

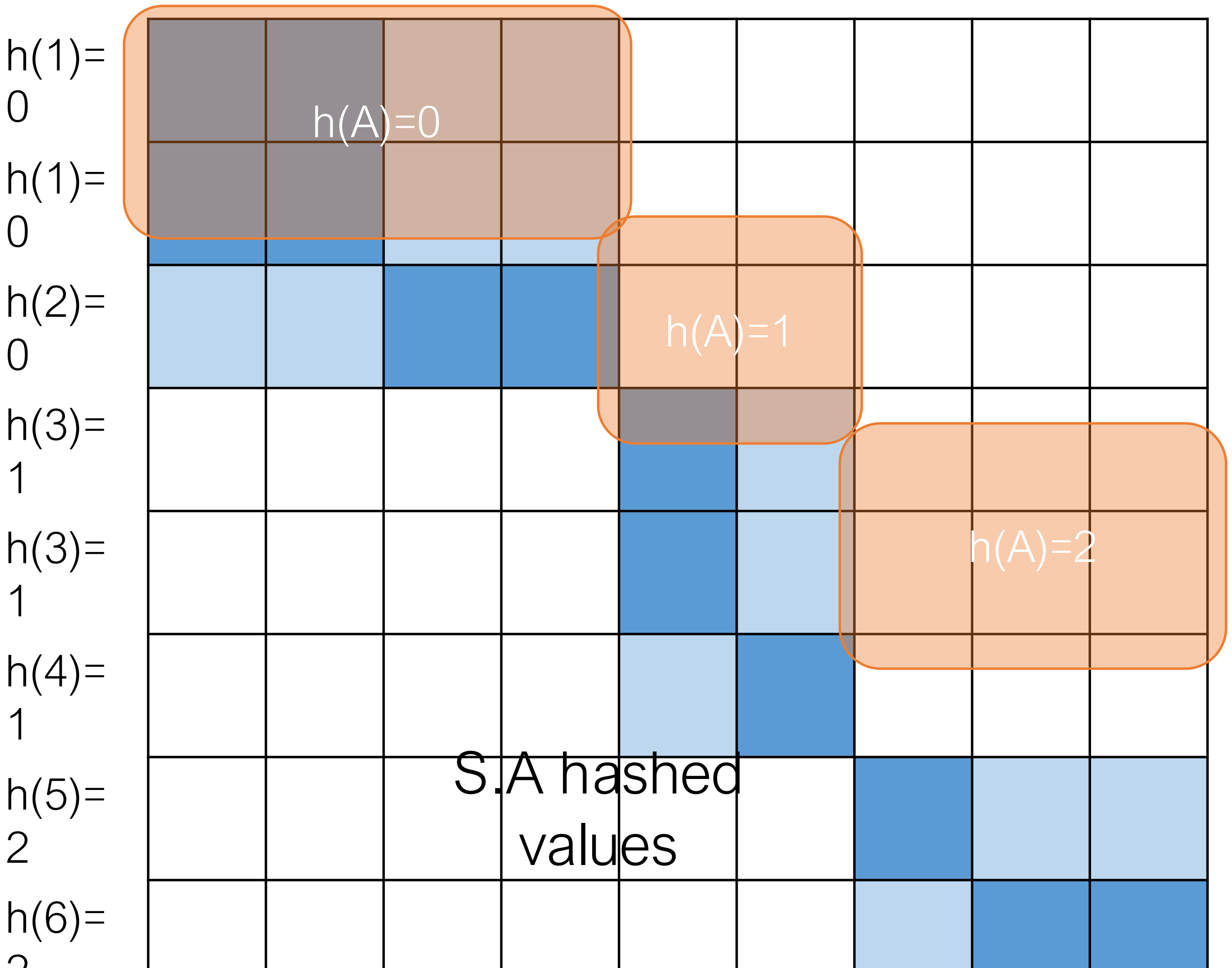


$R \bowtie S \text{ on } A$

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.A hashed values



$R \bowtie S$ 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

How much memory do we need for HJ?

- Given B+1 buffer pages

+ WLOG: Assume P(R) <= P(S)

- Suppose (reasonably) that we can partition into B buckets in 2 passes:
  - For R, we get B buckets of size ~P(R)/B
  - To join these buckets in linear time, we need these buckets to fit in B-1 pages, so we have:

$$B - 1 \geq \frac{P(R)}{B} \Rightarrow \sim \boldsymbol{B^2} \geq \boldsymbol{P(R)}$$

Quadratic relationship between smaller relation's size & memory!

Hash Join Summary

- *Given enough buffer pages as on previous slide…*
  - **Partitioning** requires reading + writing each page of R,S
    - → 2(P(R)+P(S)) IOs
  - **Matching** (with BNLJ) requires reading each page of R,S
    - → 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 ~3(P(R)+P(S)) + OUT IOs!

Sort-Merge v. Hash Join

Given enough memory, both SMJ and HJ have performance:

$$\sim 3(P(R)+P(S)) + OUT$$

"Enough" memory =

- SMJ: $B^2 > \max\{P(R), P(S)\}$
- HJ: $B^2 > \min\{P(R), P(S)\}$

Hash Join superior if relation sizes differ greatly. Why?

Further Comparisons of Hash and Sort Joins

- Hash Joins are highly parallelizable.

- Sort-Merge less sensitive to data skew and result is sorted