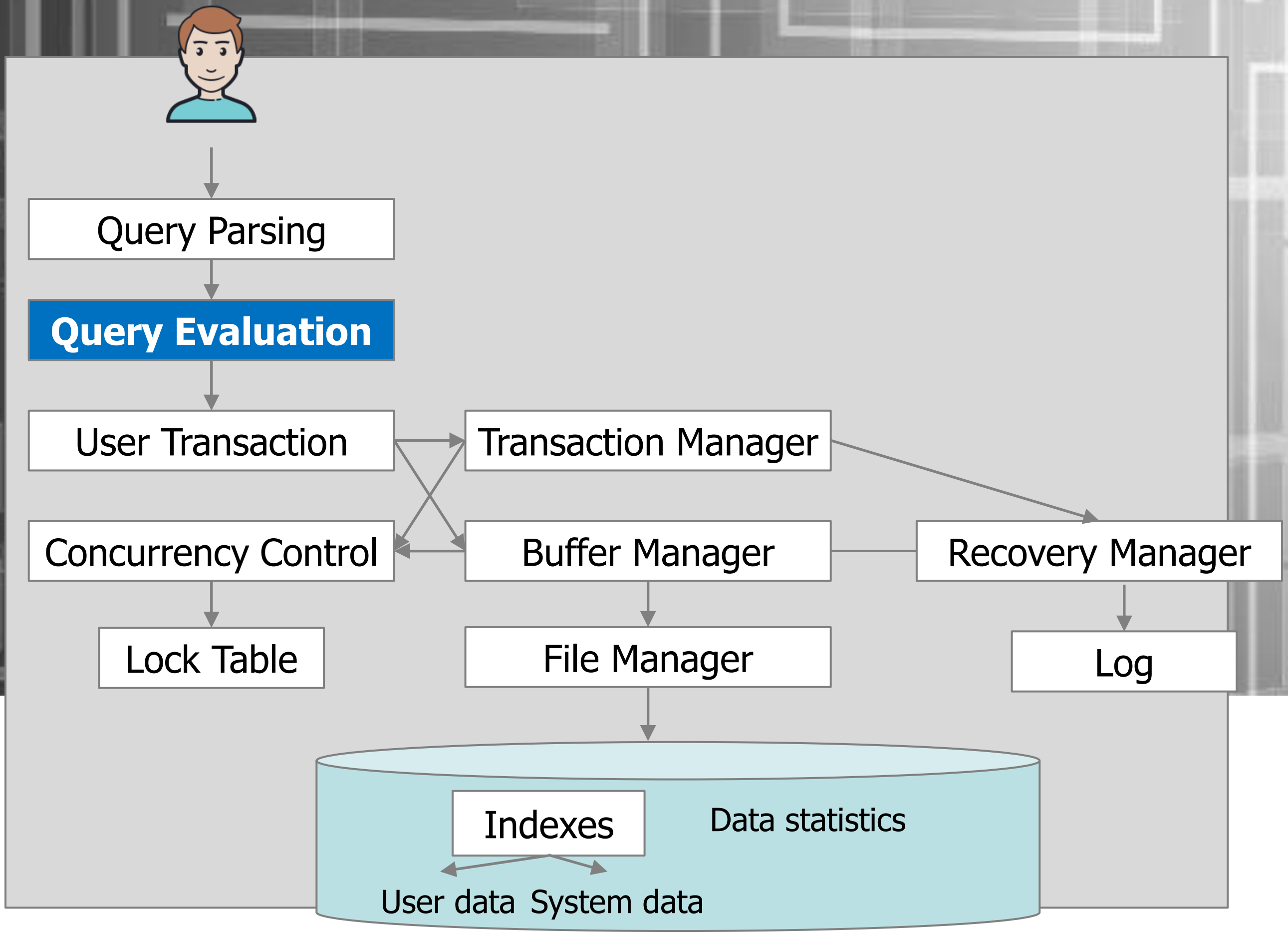
M146 Database Systems
Spring 2021

Georgia Koutrika

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

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***

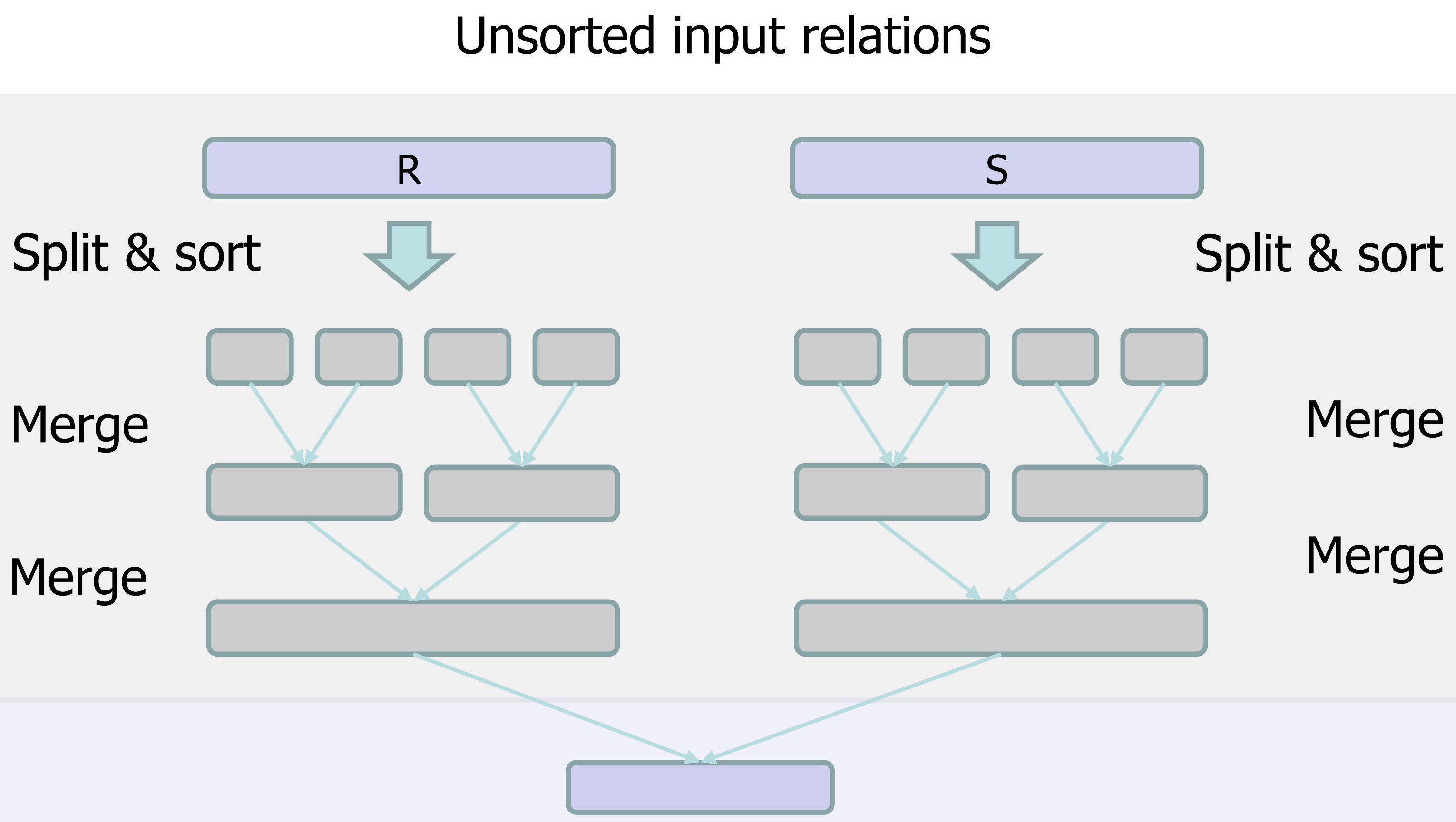
2. ***Scan*** sorted files and “merge”

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

SMJ

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

Sort Phase (Ext. Merge Sort)



Merge / Join Phase

Joined output file created!

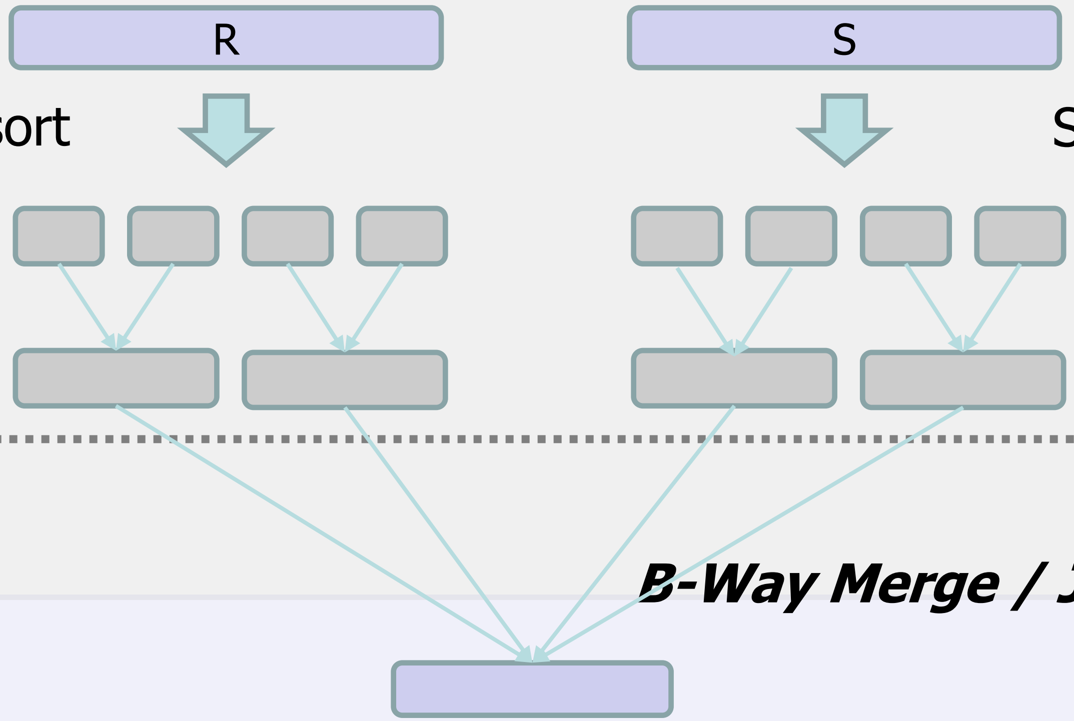
Simple SMJ Optimization

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, $\mathbf{max}\{\mathbf{P}(\mathbf{R}), \mathbf{P}(\boldsymbol{S})\} \leq \boldsymbol{B}^{\mathbf{2}}$ is an approximate sufficient condition**

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

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 **never** occur that x = y but $h_B(x) \mathrel{!=} h_B(y)$

- We hash on an attribute A, so our hash function $h_B(t)$ has the form $h_B(t.A)$.
  - **Collisions** may be more frequent.

Hash Join: High-level procedure

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

Note again that we are only considering equality constraints here

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

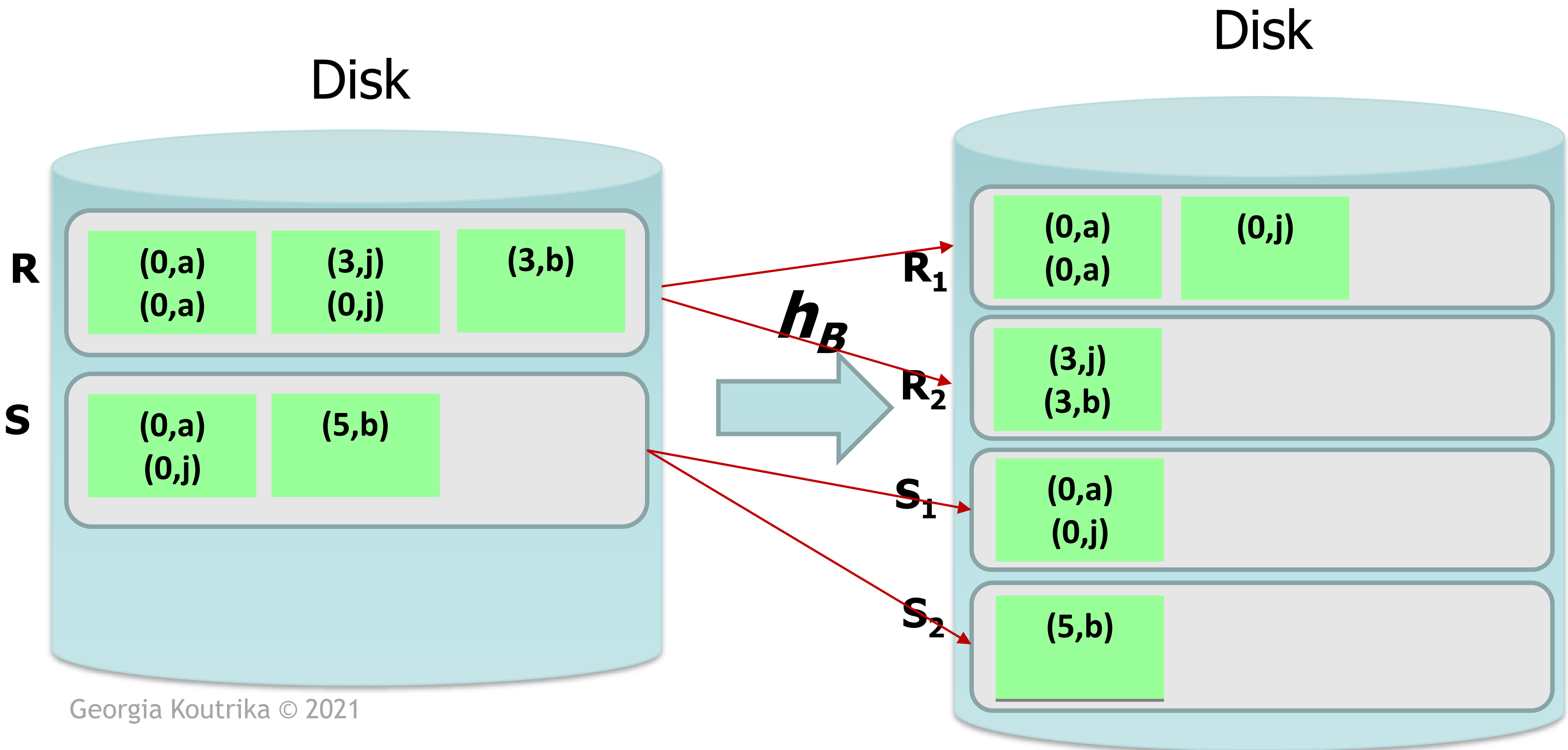
2. **Matching Phase:** Take pairs of buckets whose tuples have the same values for ***h***, and join these
   1. Use BNLJ here; or hash again → either way, operating on small partitions so fast!

We ***decompose*** the problem using $\boldsymbol{h_B}$, then complete the join

Hash Join: High-level procedure

Assume each page has two tuples (one per row)

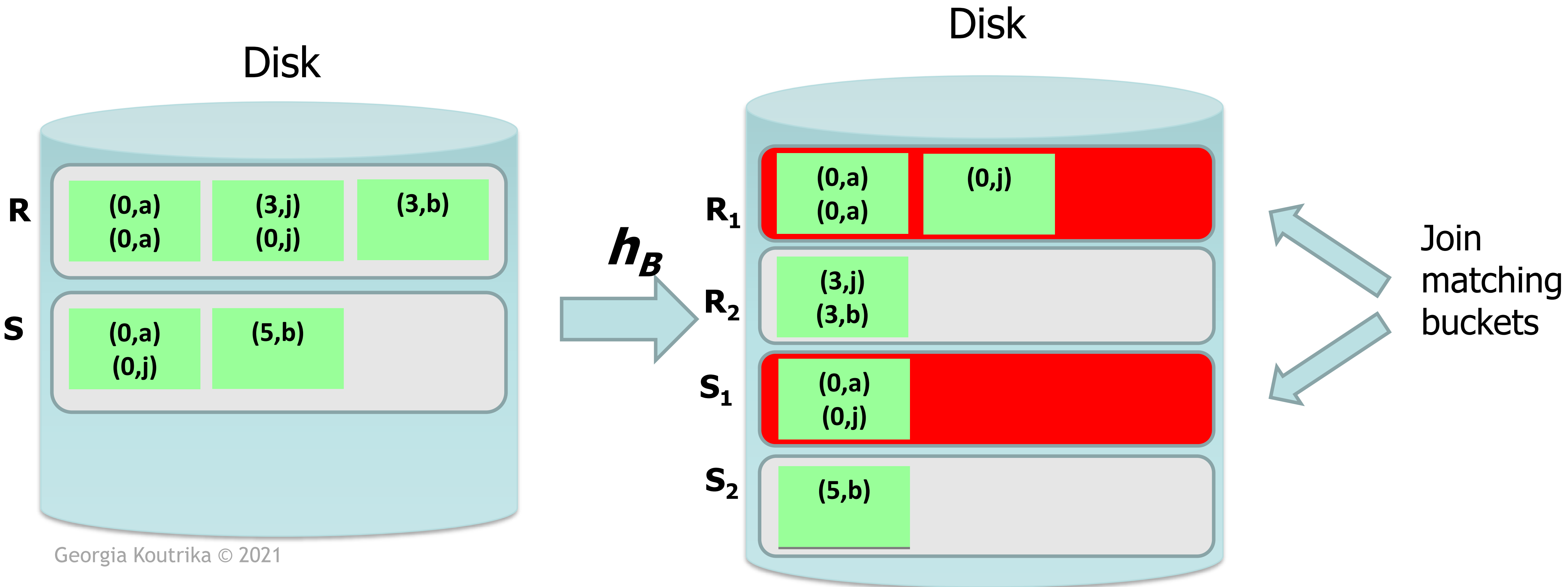
1. **Partition Phase:** Using one (shared) hash function h_B, partition R *and* S into ***B*** buckets



More detail in a second…

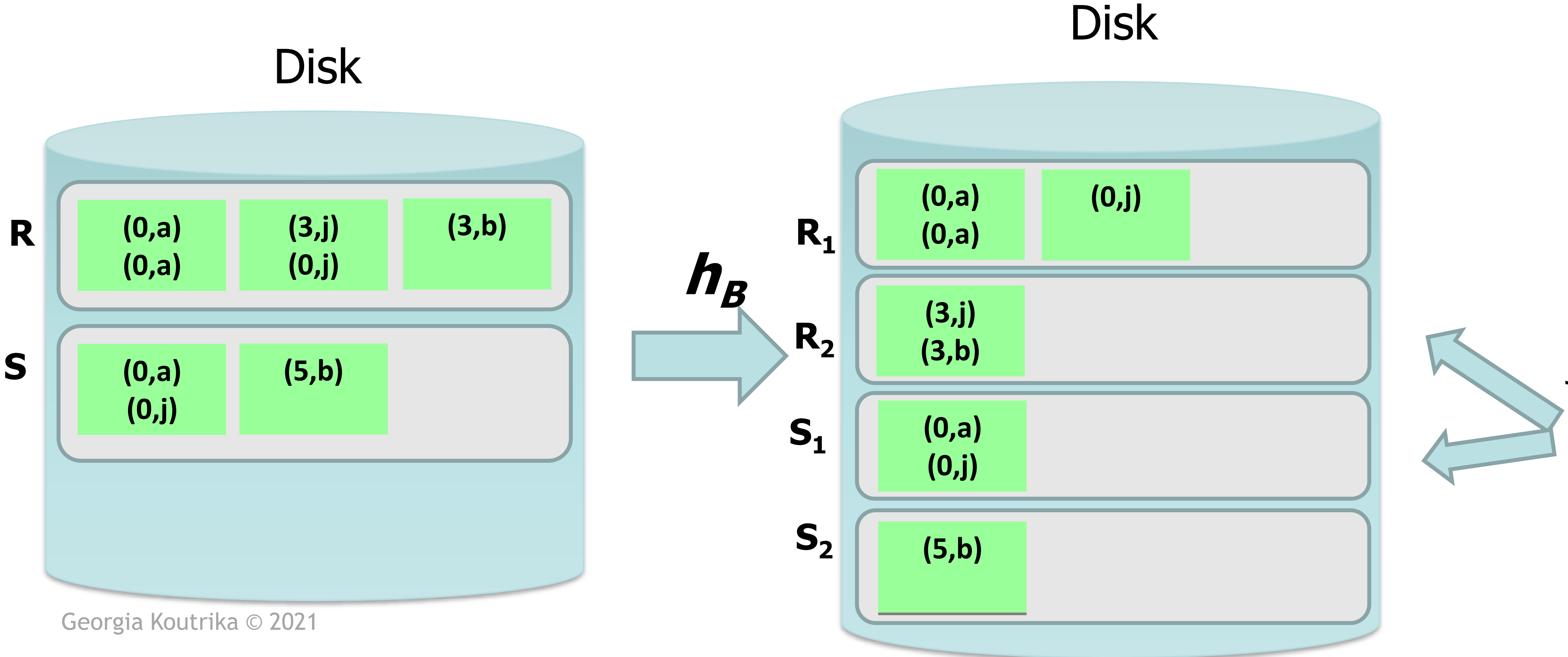
Hash Join: High-level procedure

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



Hash Join: High-level procedure

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



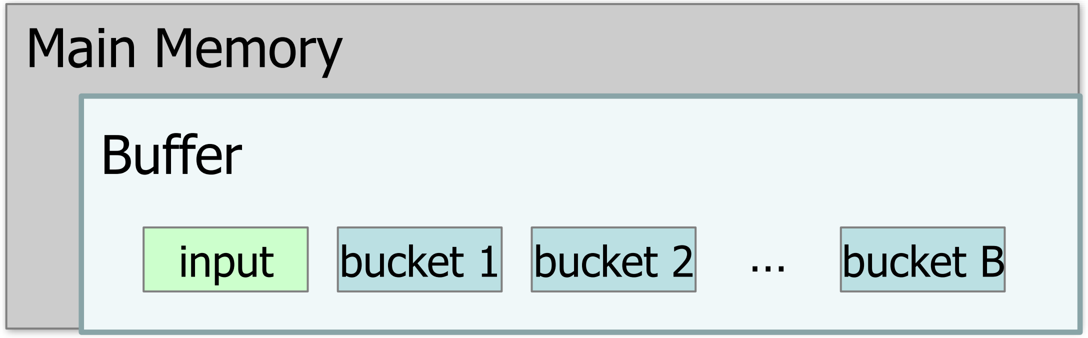
Don't have to join the others! E.g. (S_1 and R_2)!

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:

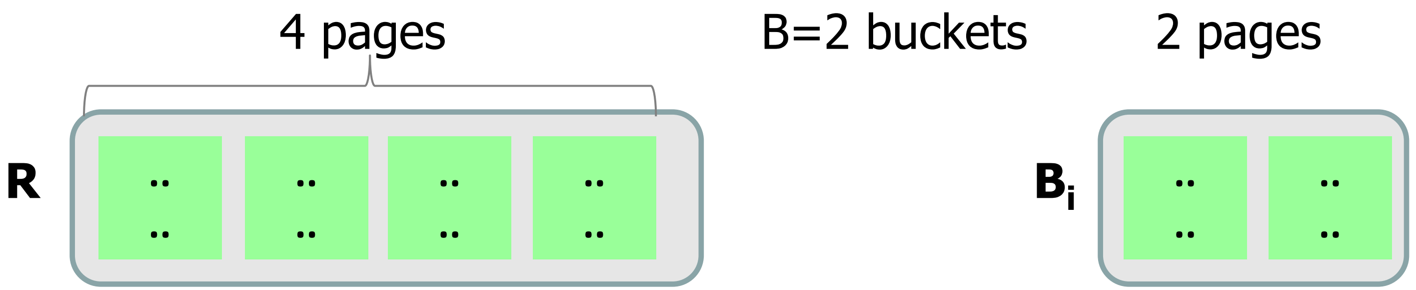
- **We use B buffer pages for output (one for each bucket), and 1 for input**
  - The “dual” of sorting.
  - For each tuple t in input, copy to buffer page for $h_B(t.A)$
  - When a bucket page fills up, flush to disk.



How big are the resulting buckets?

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

- Given **N input pages, we partition into B buckets:**
  - → Ideally our buckets are each of size **~ N/B pages**

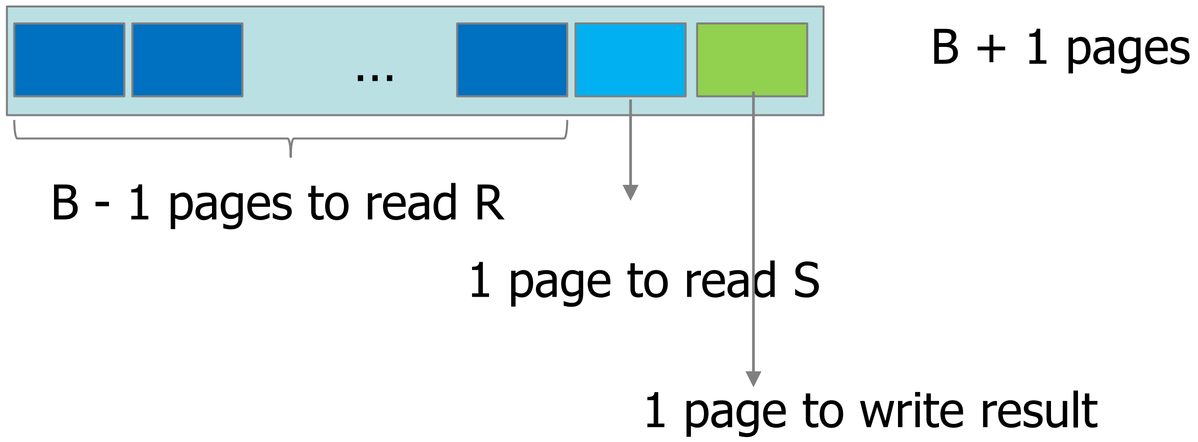


- What happens if there are **hash collisions**?
  - Buckets could be > N/B
  - **We'll do several passes…**

- What happens if there are **duplicate join keys?**
  - Nothing we can do here… could have some **skew** in size of the buckets

RECALL: Block Nested Loop Join (BNLJ)

Buffer



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)
```

Hash Join Phase 2: Join

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

How big *do we want* the resulting buckets?

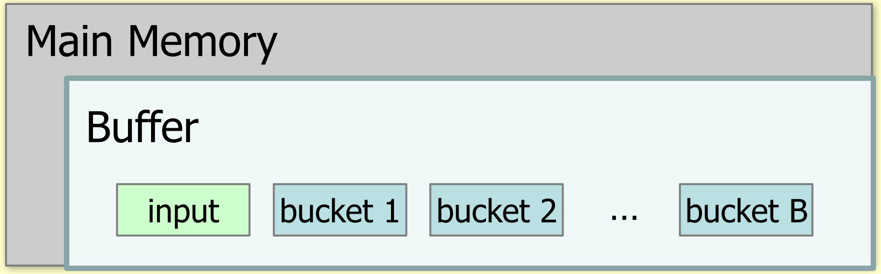
- Recall: If we want to join a bucket from R and one from S, we can do BNLJ **in linear time** if $\boldsymbol{P(R) \leq B - 1}$!
  - And more generally, being able to fit bucket in memory is advantageous
- Ideally, our buckets would be of size $\boldsymbol{\leq B - 1}$ **pages**
  - ***1*** for input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

Recall for BNLJ:

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

Let's see how we do the partitioning then!

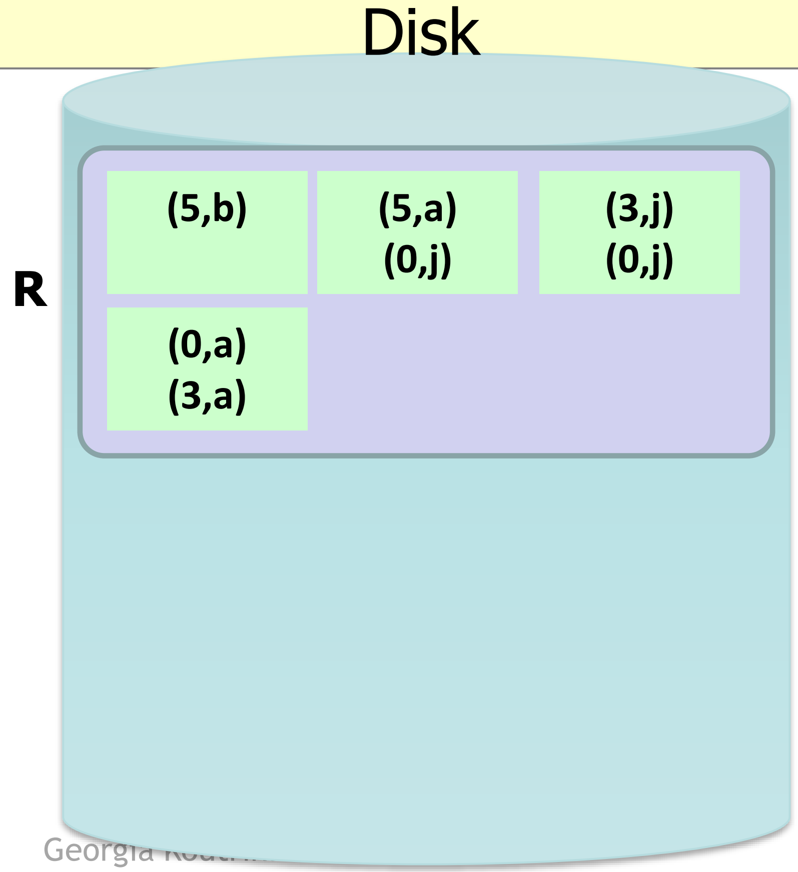
Hash Join Phase 1: Partitioning

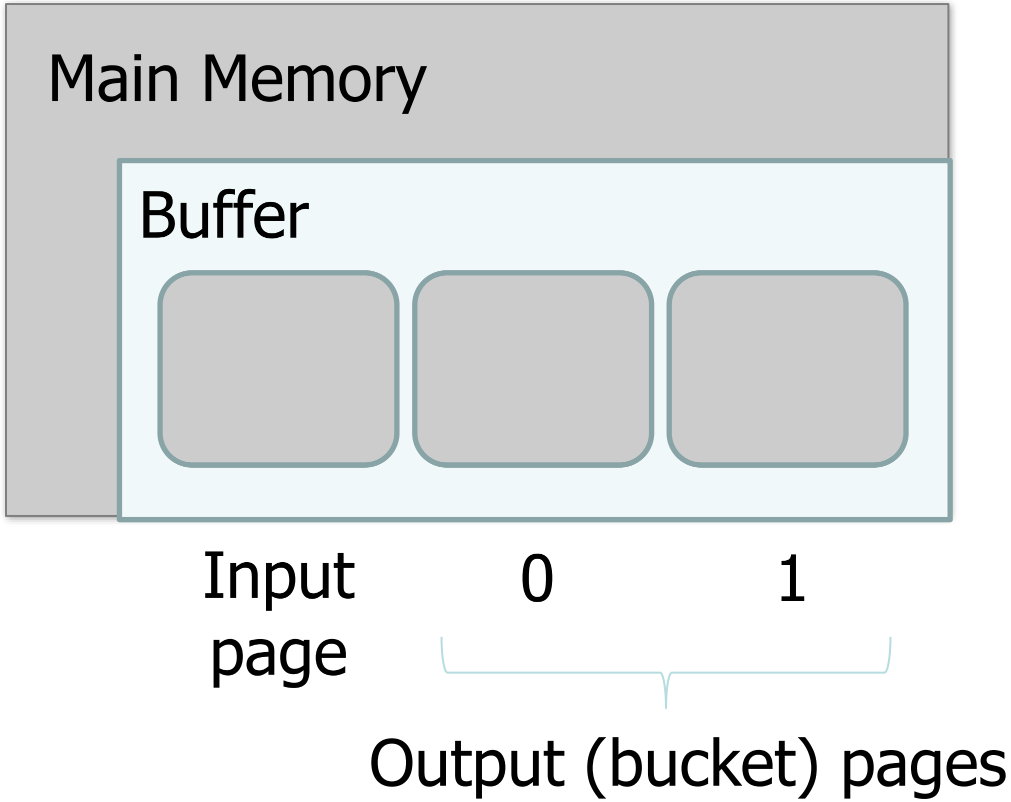


Given ***B+1 = 3*** buffer pages

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

Our goal (for the join) will be to get ***B = 2 buckets*** of size <= ***B-1 → 1 page each***

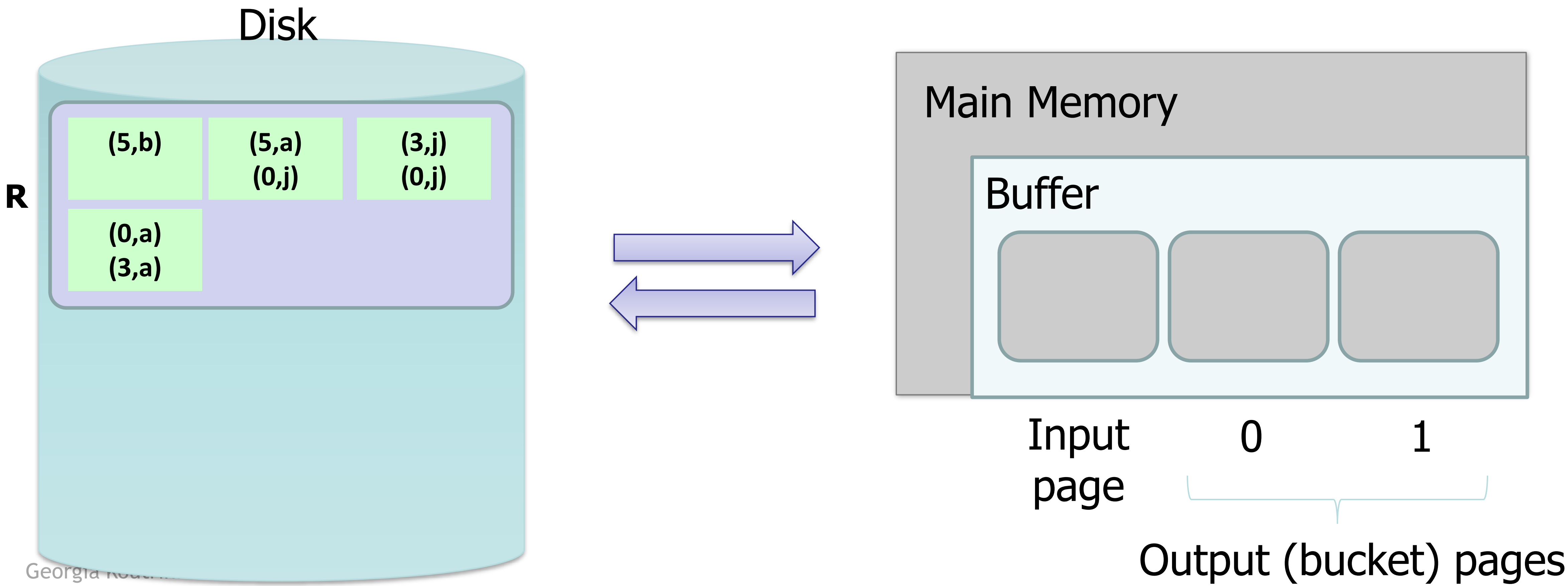




Hash Join Phase 1: Partitioning

Given ***B+1 = 3*** buffer pages

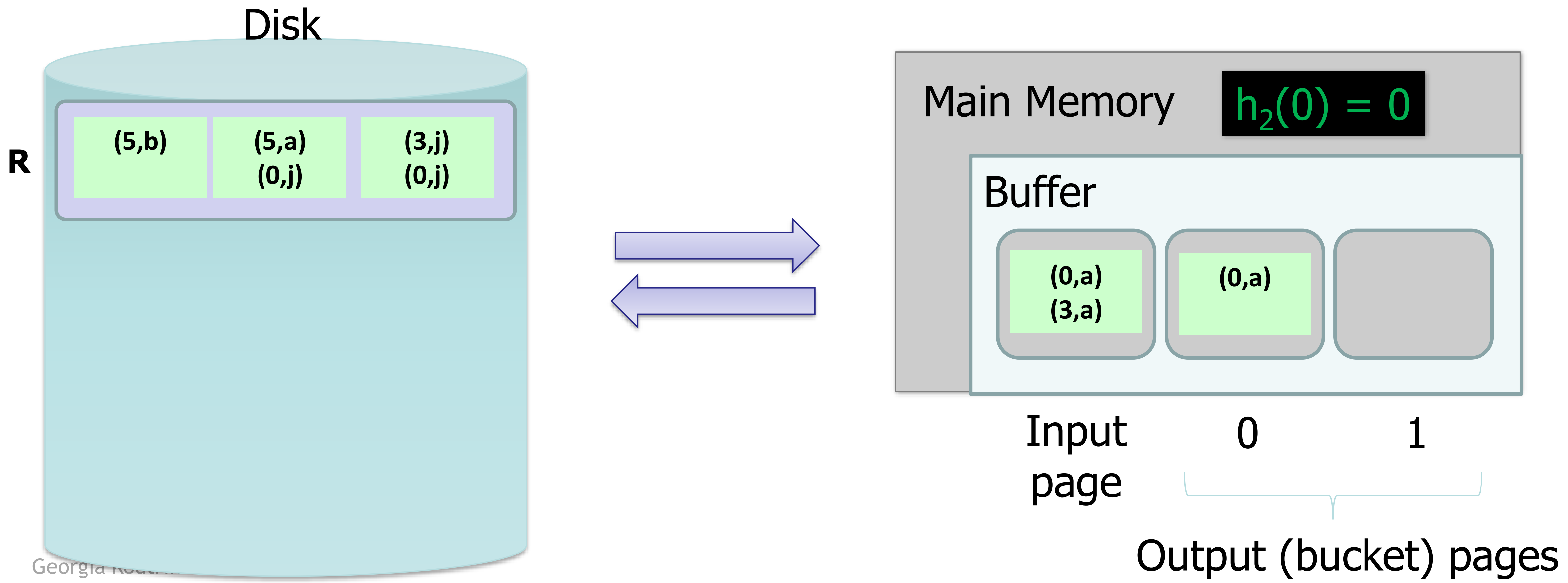
1. We read pages from R into the “input” page of the buffer…



Hash Join Phase 1: Partitioning

Given $B+1 = 3$ buffer pages

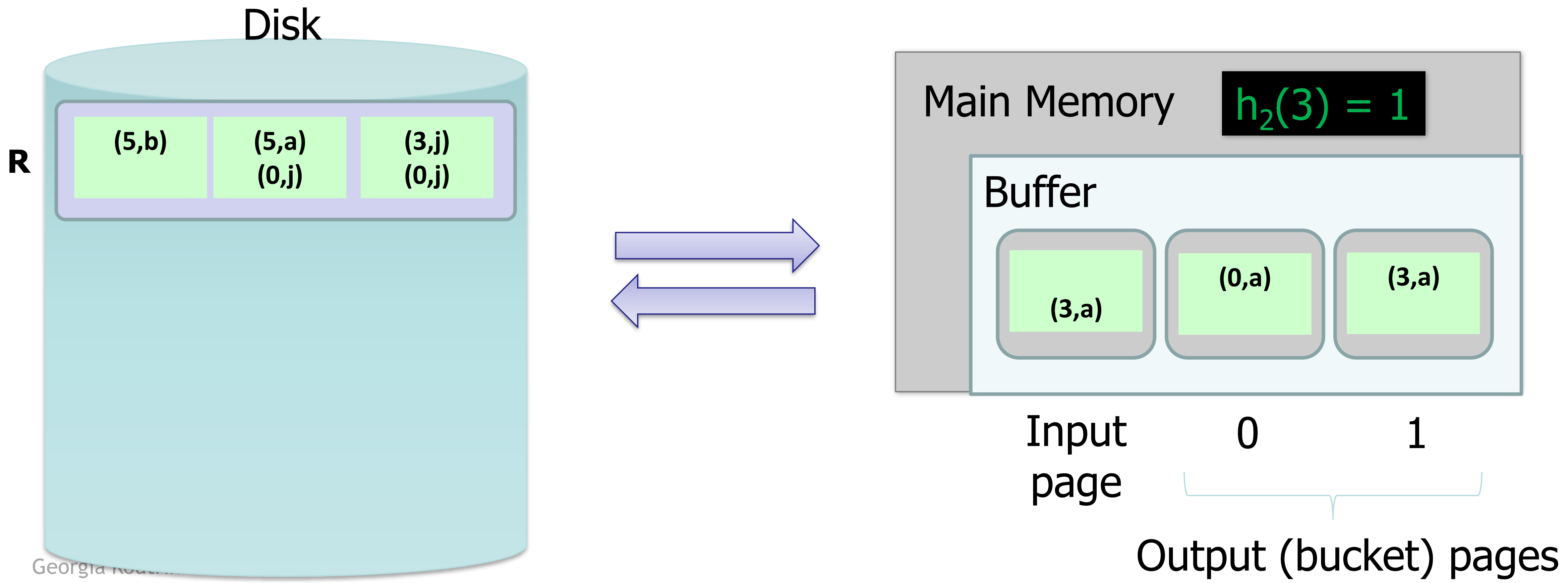
2. Then we use **hash function h_2** to sort into the buckets, which each have one page in the buffer



Hash Join Phase 1: Partitioning

Given ***B+1 = 3*** buffer pages

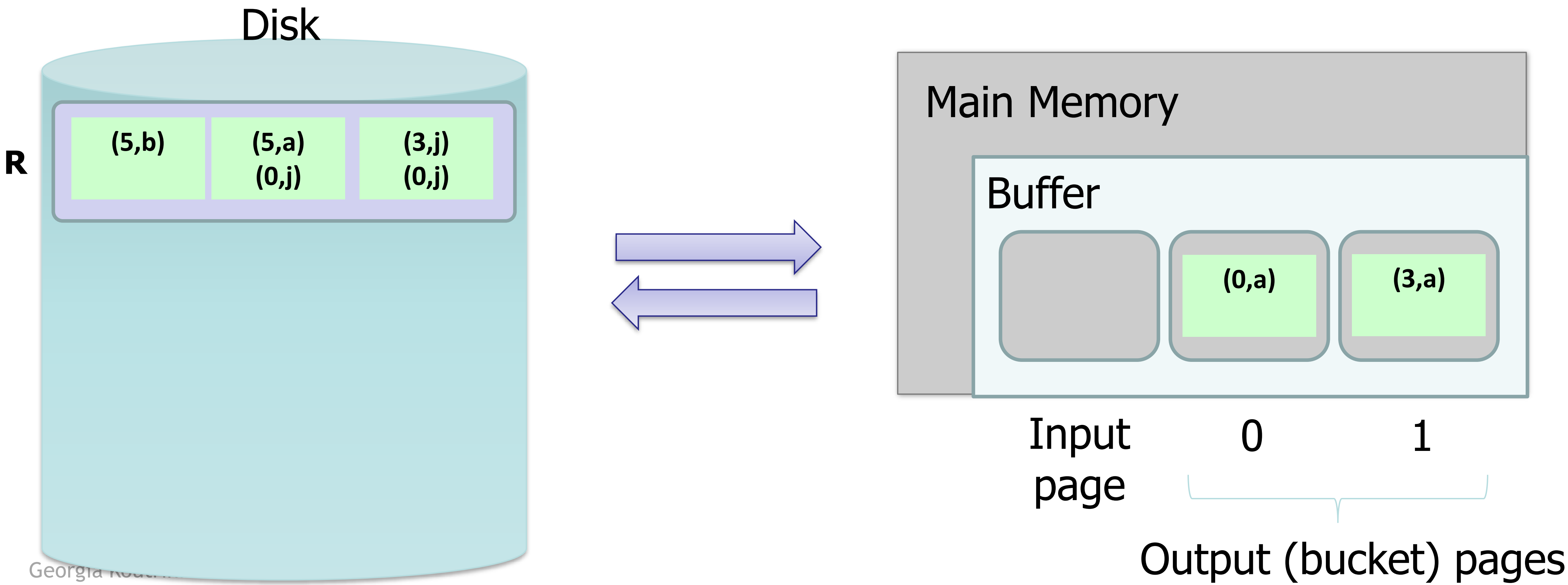
2. Then we use **hash function h_2** to sort into the buckets, which each have one page in the buffer



Hash Join Phase 1: Partitioning

Given ***B+1 = 3*** buffer pages

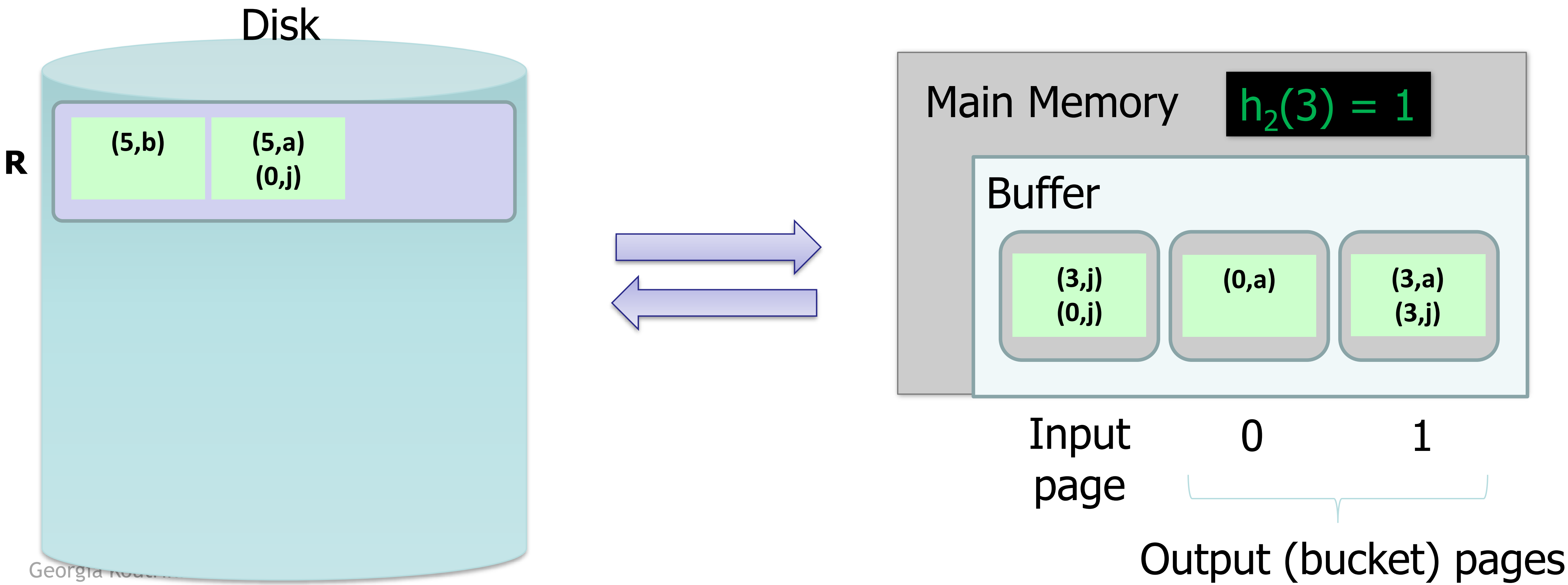
3. We repeat until the buffer bucket pages are full…



Hash Join Phase 1: Partitioning

Given ***B+1 = 3*** buffer pages

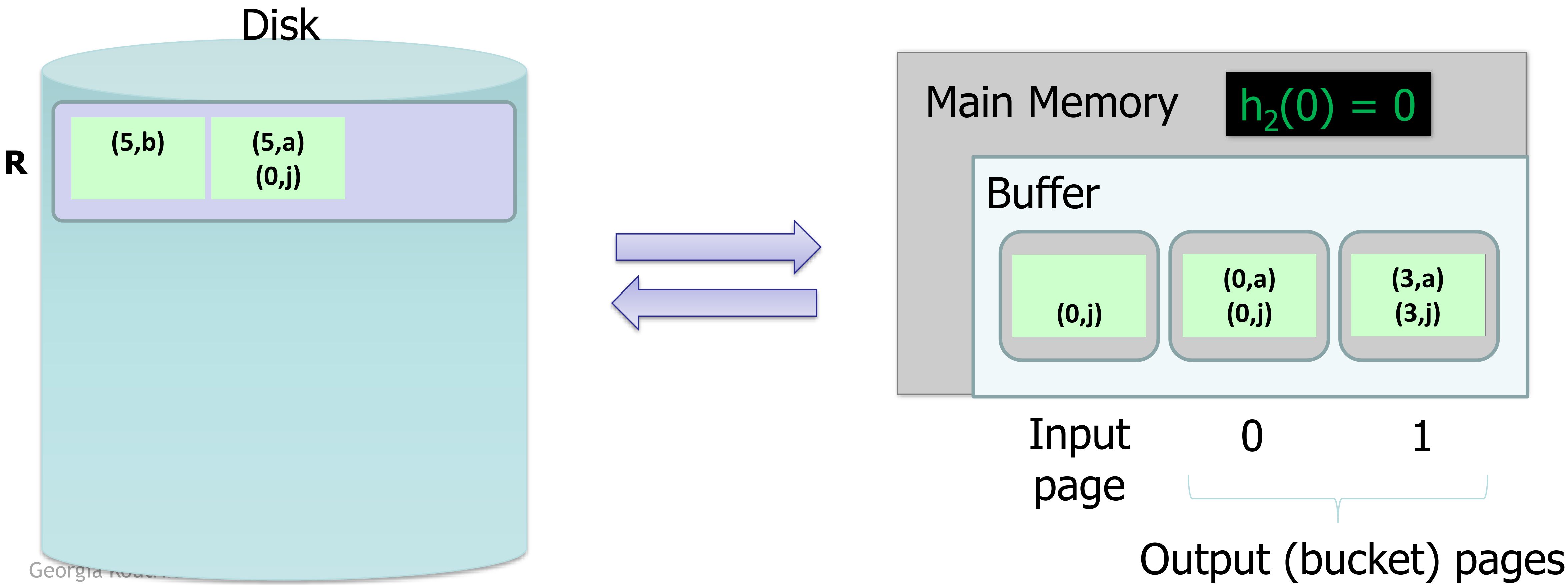
3. We repeat until the buffer bucket pages are full…



Hash Join Phase 1: Partitioning

Given ***B+1 = 3*** buffer pages

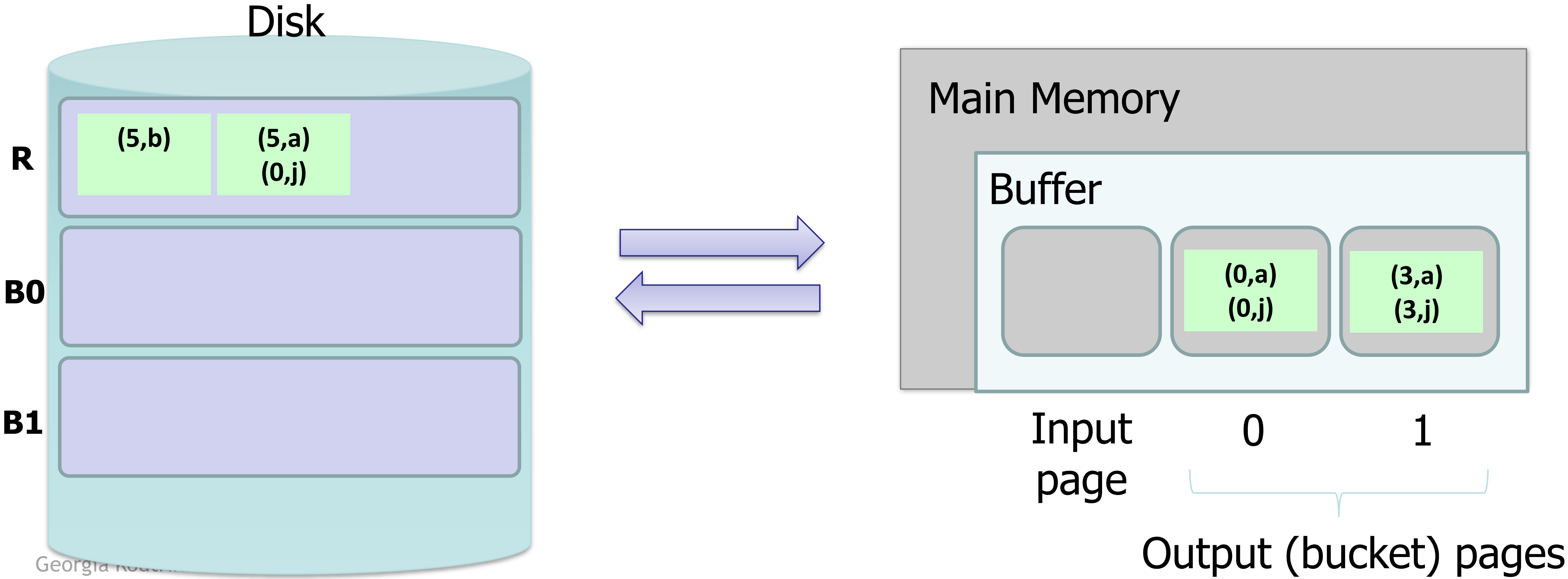
3. We repeat until the buffer bucket pages are full...



Hash Join Phase 1: Partitioning

Given ***B+1 = 3*** buffer pages

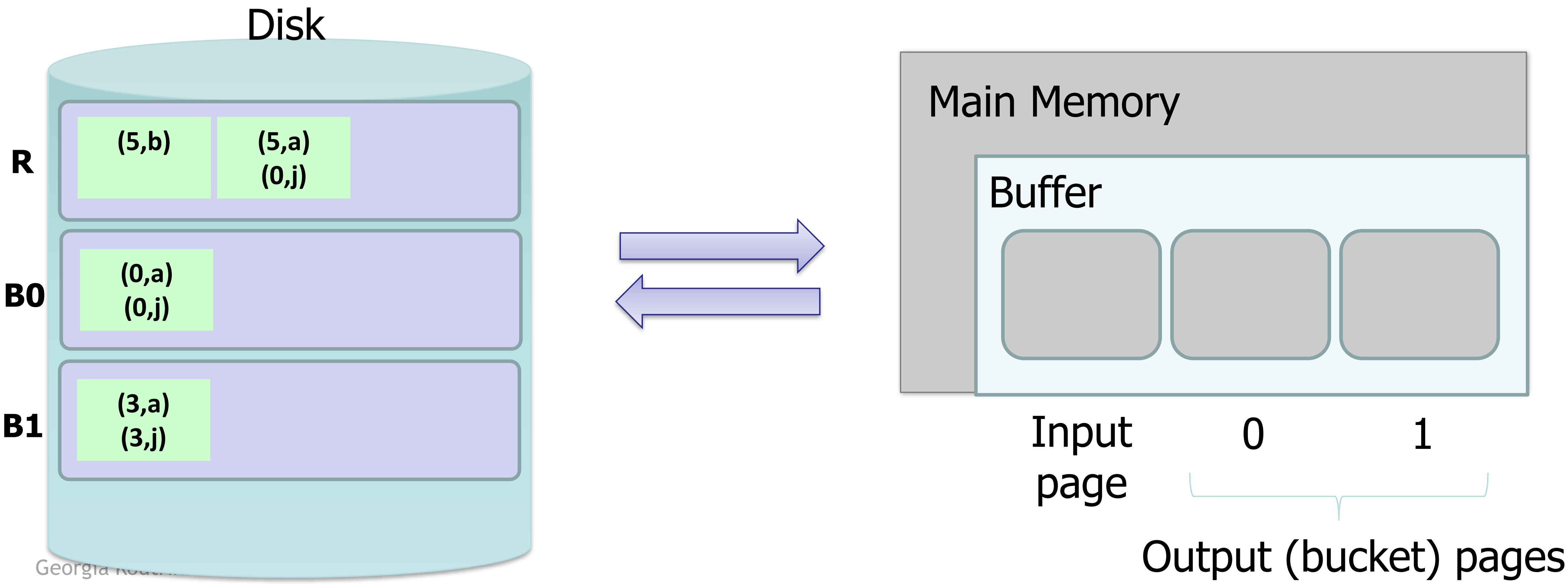
3. We repeat until the buffer bucket pages are full… then flush to disk



Hash Join Phase 1: Partitioning

Given ***B+1 = 3*** buffer pages

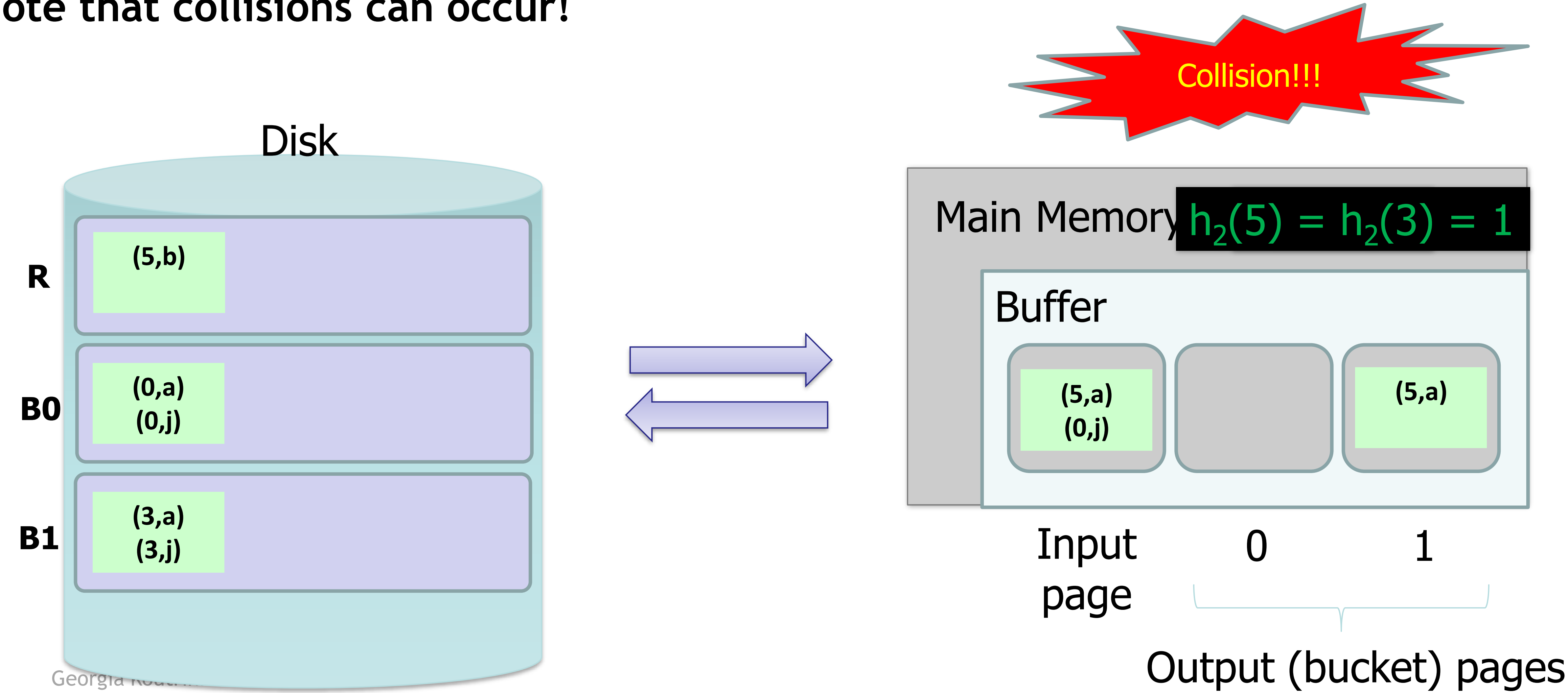
3. We repeat until the buffer bucket pages are full… then flush to disk



Hash Join Phase 1: Partitioning

Given $B+1 = 3$ buffer pages

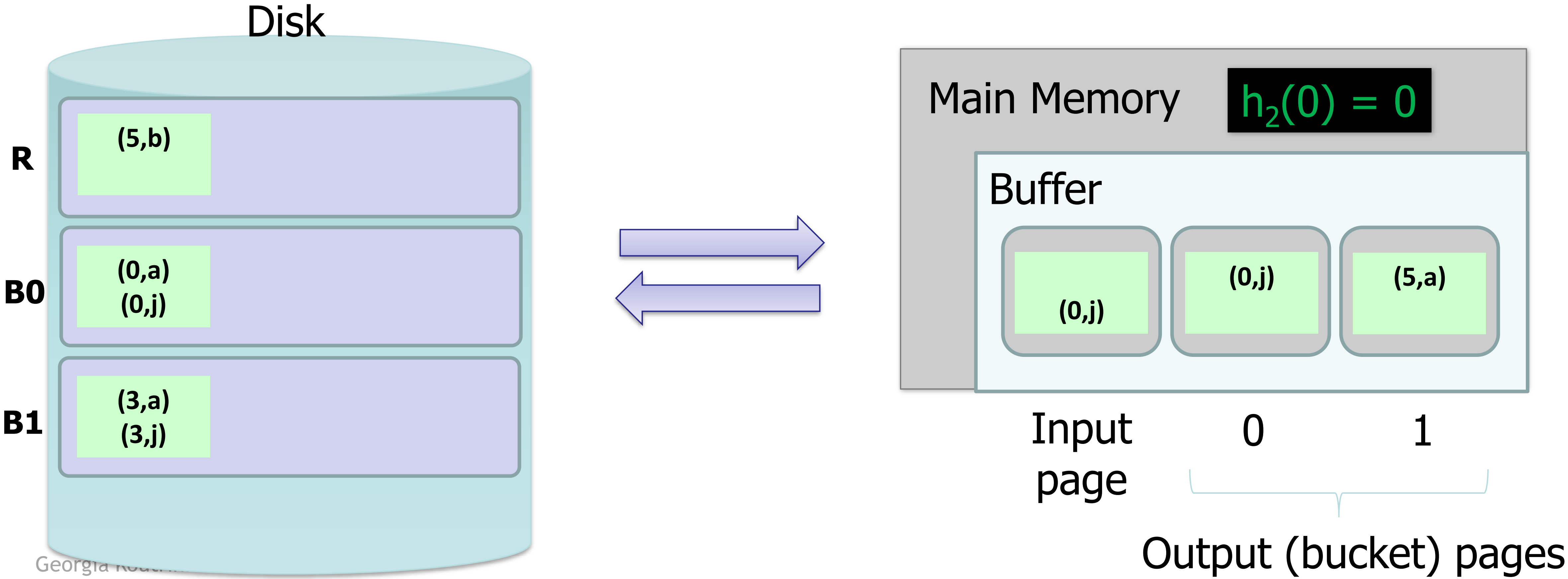
Note that collisions can occur!



Hash Join Phase 1: Partitioning

Given ***B+1 = 3*** buffer pages

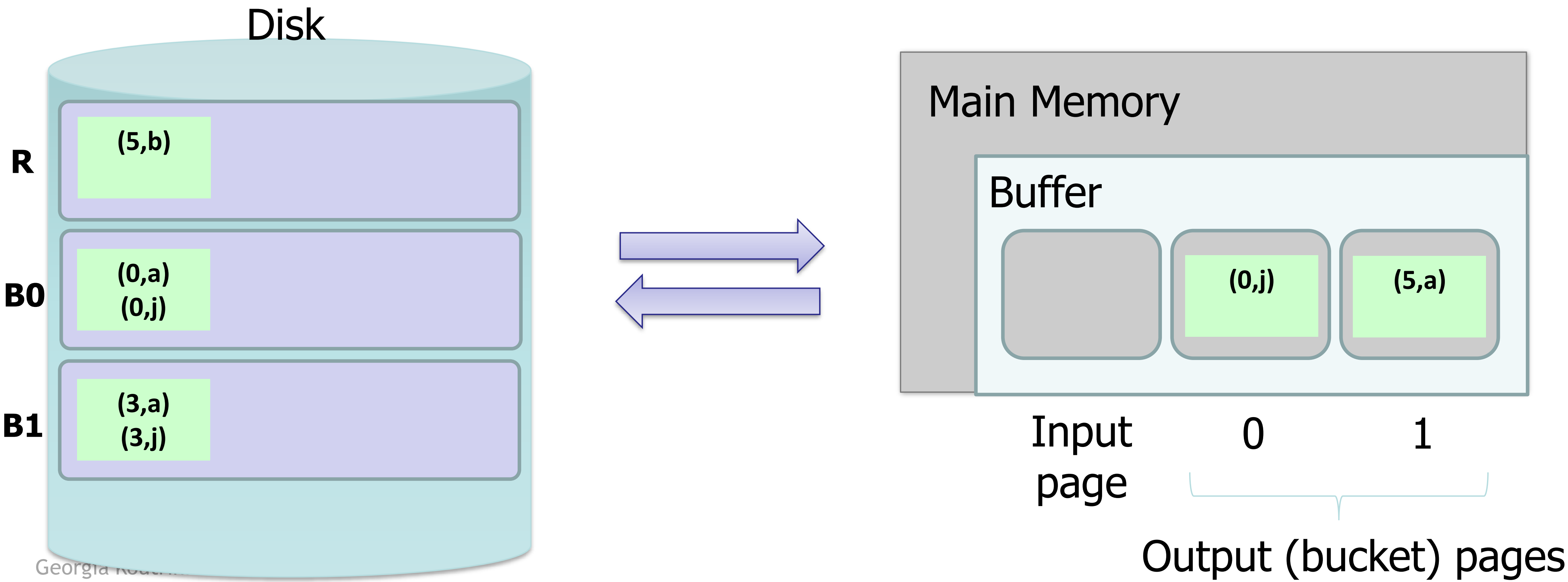
Finish this pass…



Hash Join Phase 1: Partitioning

Given ***B+1 = 3*** buffer pages

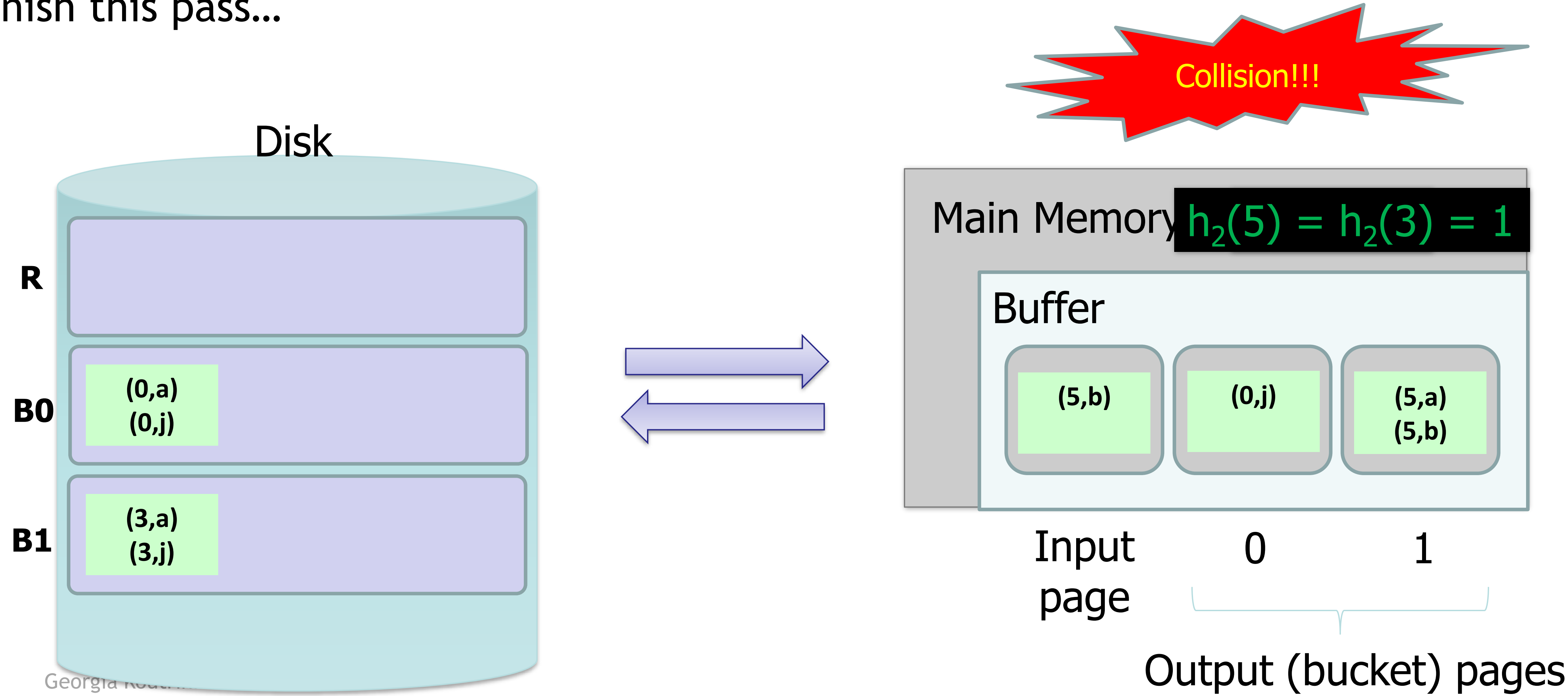
Finish this pass…



Hash Join Phase 1: Partitioning

Given ***B+1 = 3*** buffer pages

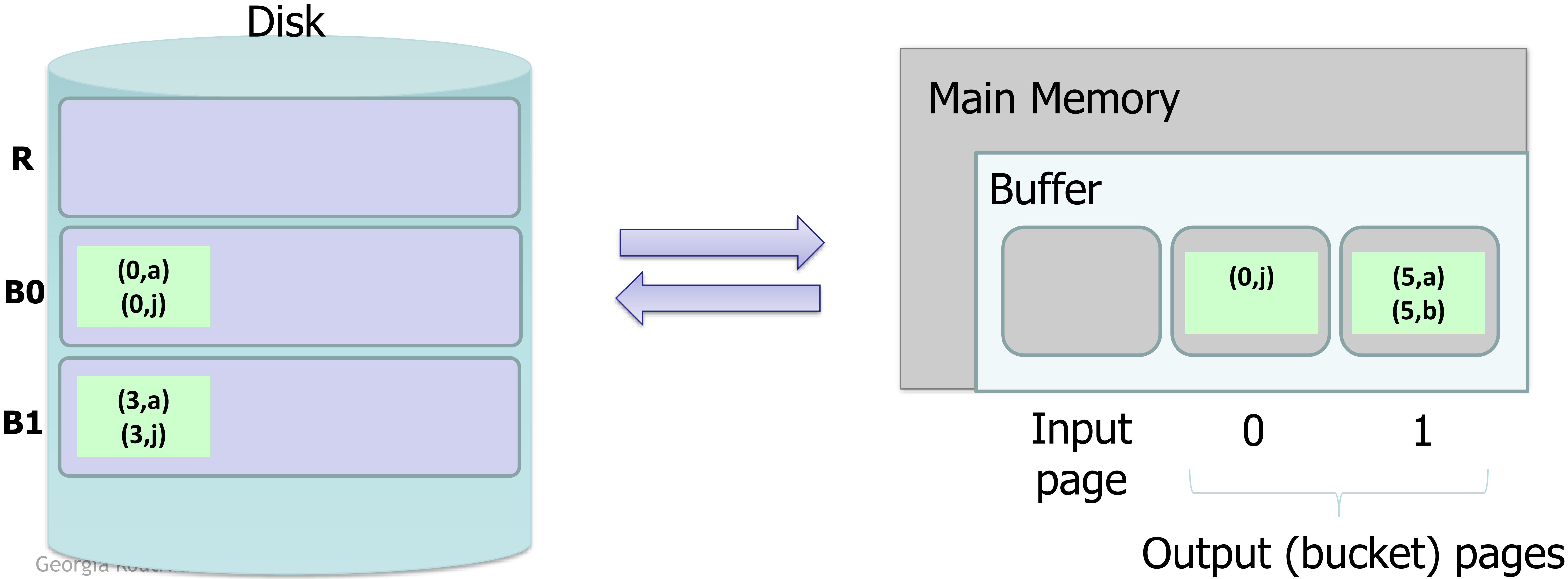
Finish this pass…



Hash Join Phase 1: Partitioning

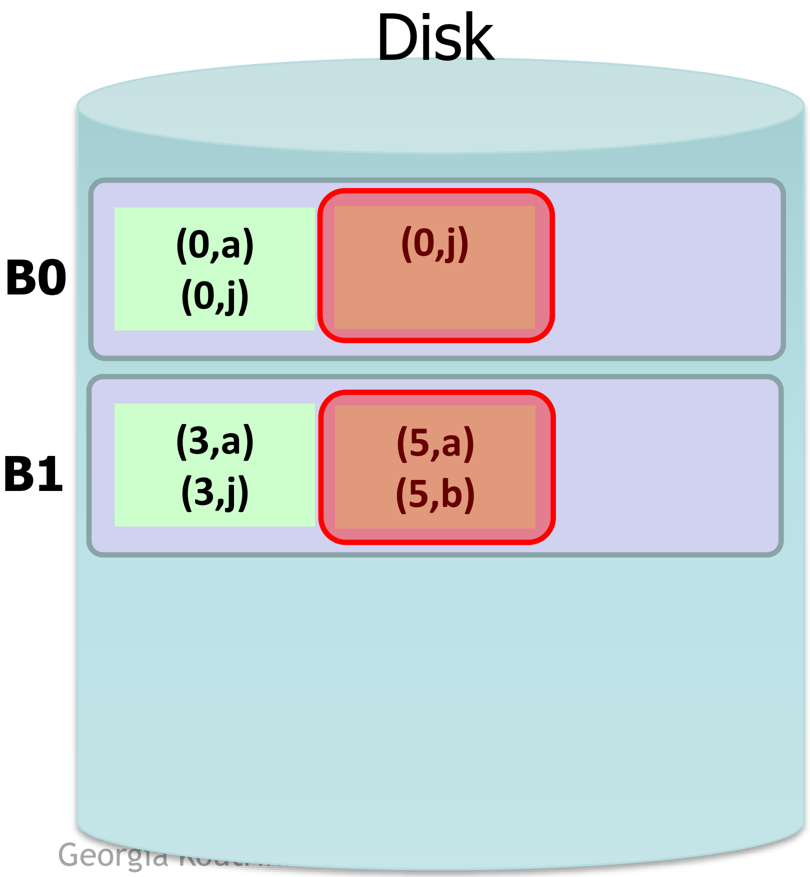
Given ***B+1 = 3*** buffer pages

Finish this pass…



Hash Join Phase 1: Partitioning

Given ***B+1 = 3*** buffer pages



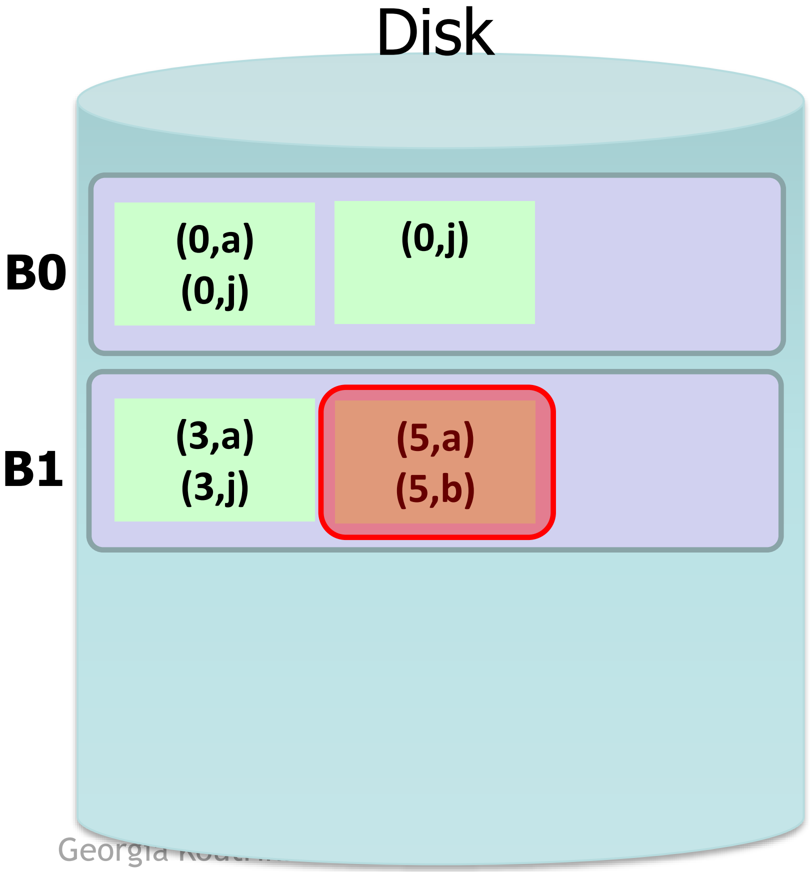
We wanted buckets of size ***B-1 = 1…***
however we got larger ones due to:

(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!

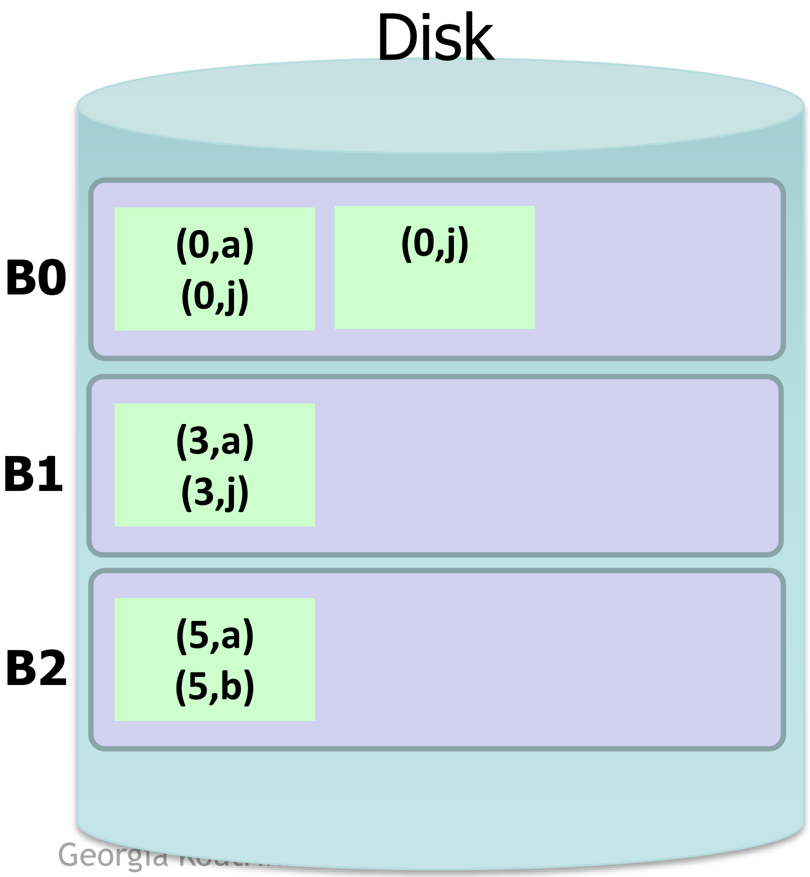
What hash function should we use?

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

$$h'_2(3) \;!=\; h'_2(5)$$

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?

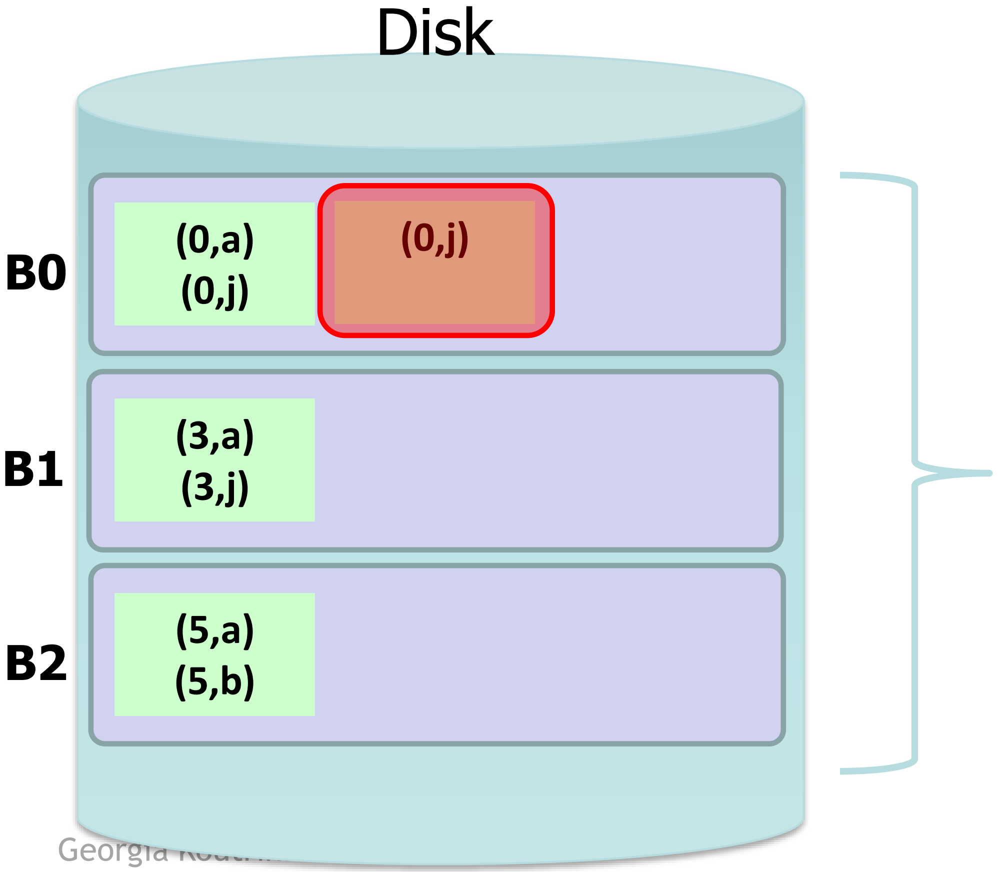
Do another pass with a different hash function, h'_2, ideally such that:

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

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.

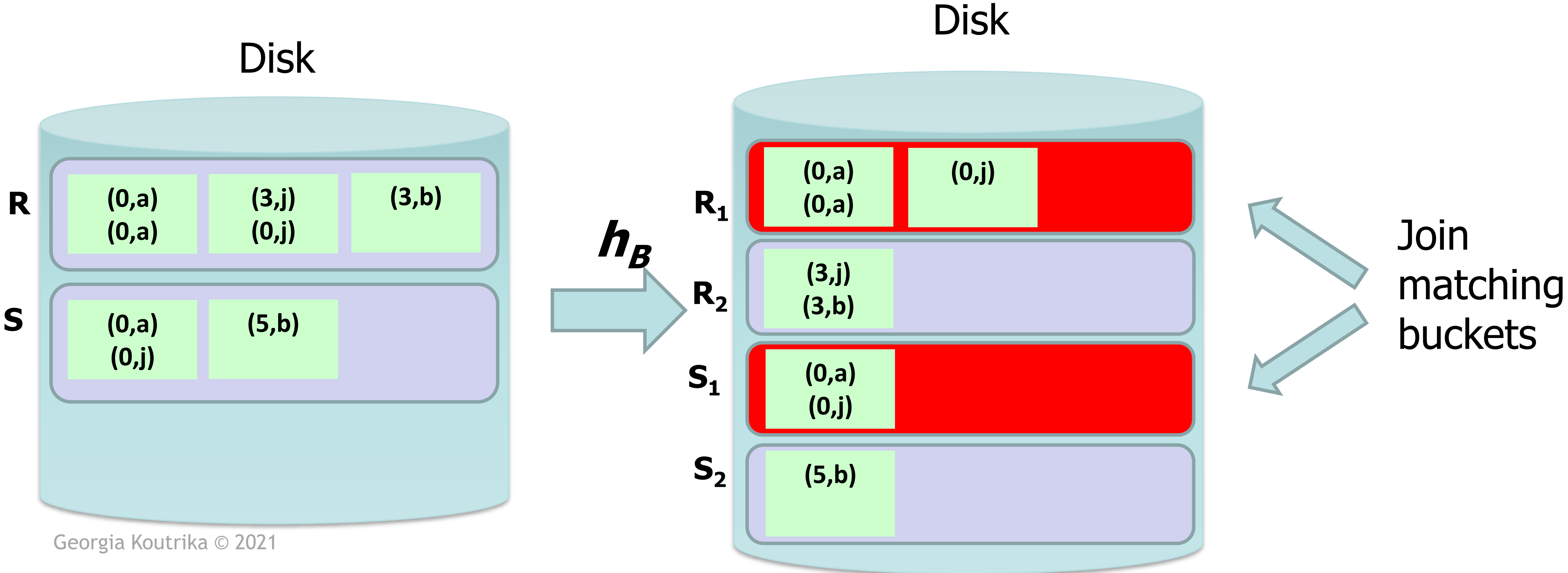


We call this unevenness in the bucket size **<u>skew</u>**

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

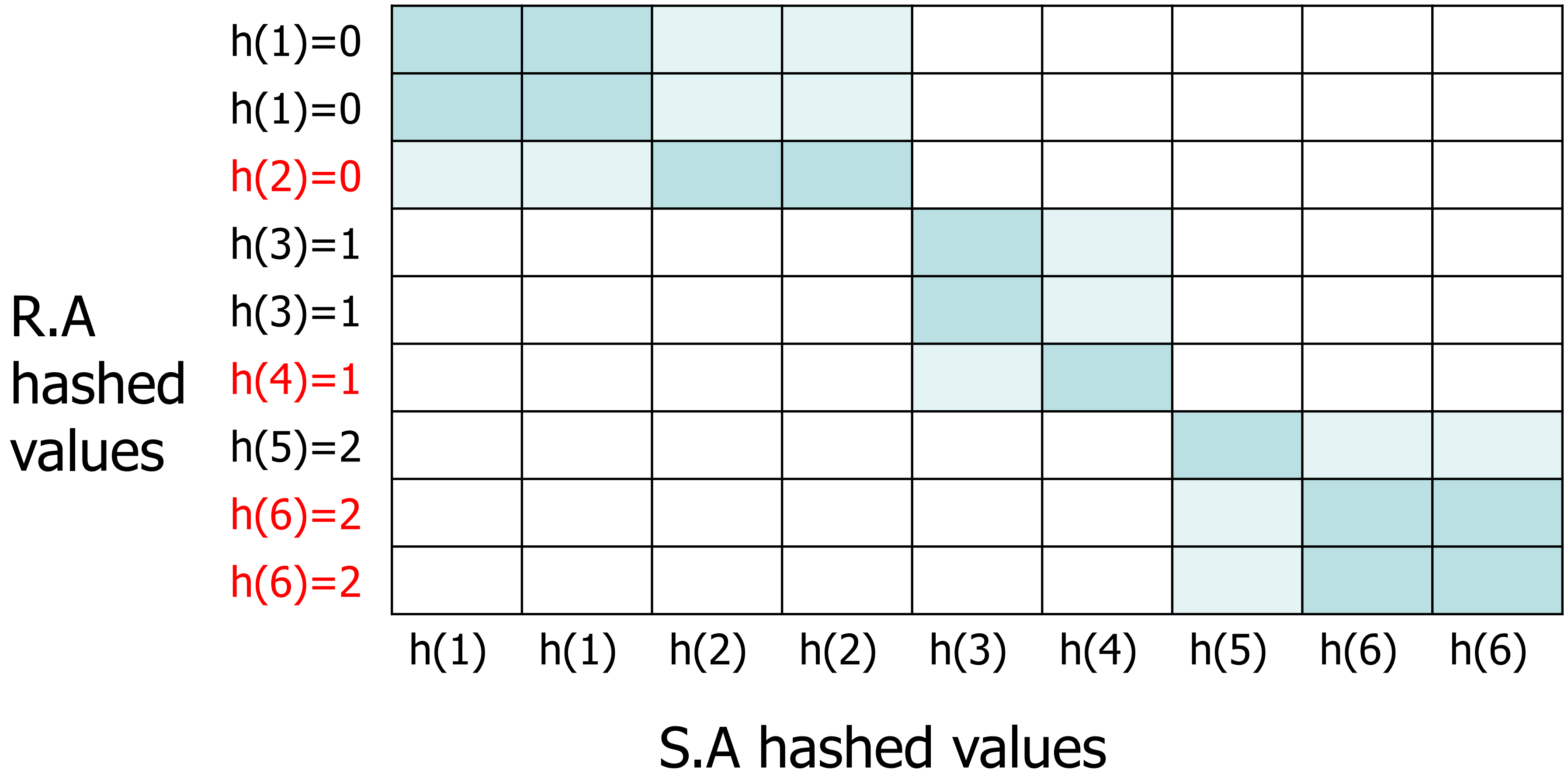
- If our buckets are **$\sim B - 1$ pages**, can join each such pair using BNLJ ***in linear time***; recall (with P(R) = B-1):

<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)

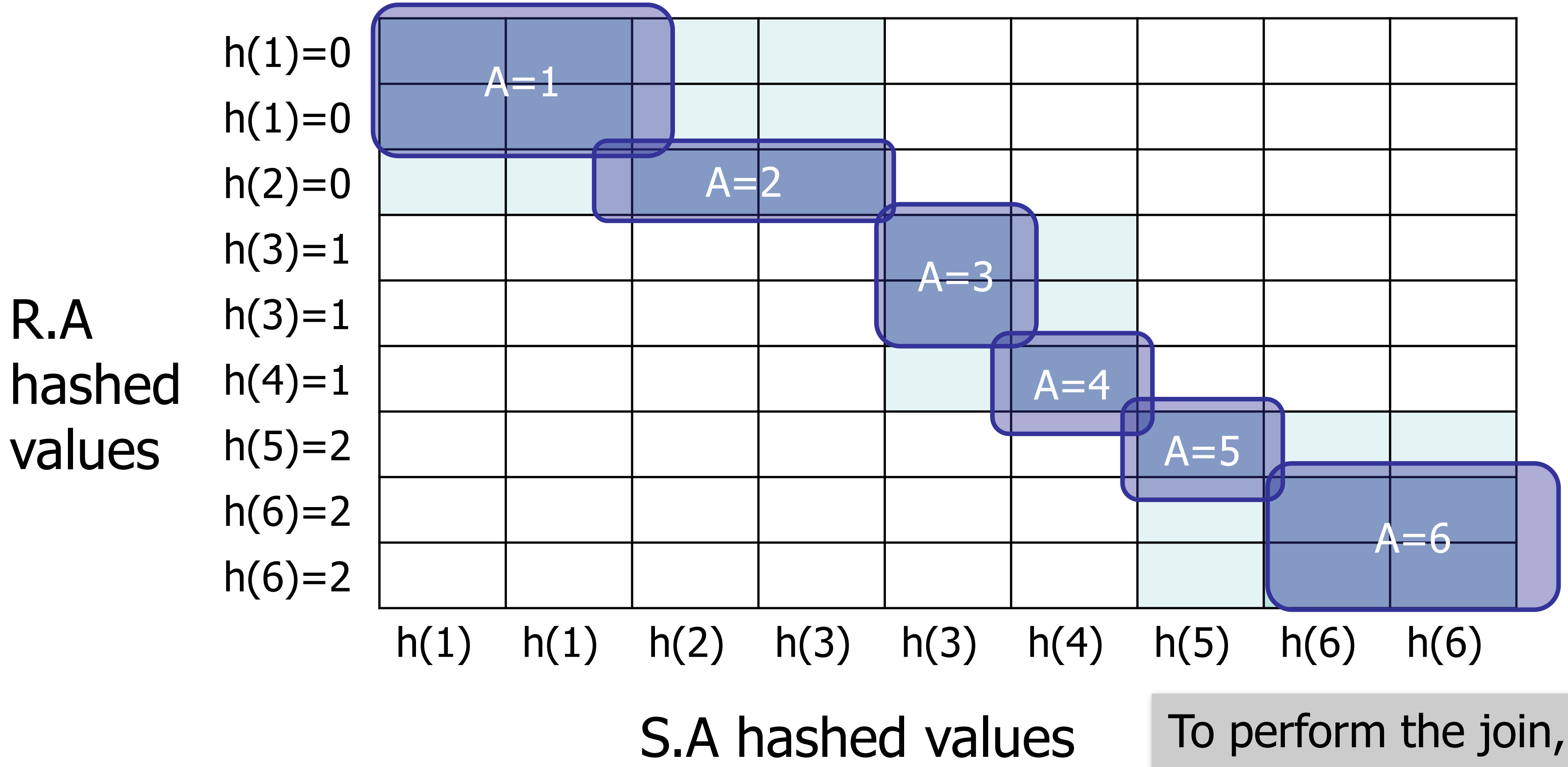
Joining the pairs of buckets is linear!
(As long as smaller bucket <= B-1 pages)

Hash Join Phase 2: Matching

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



Hash Join Phase 2: Matching

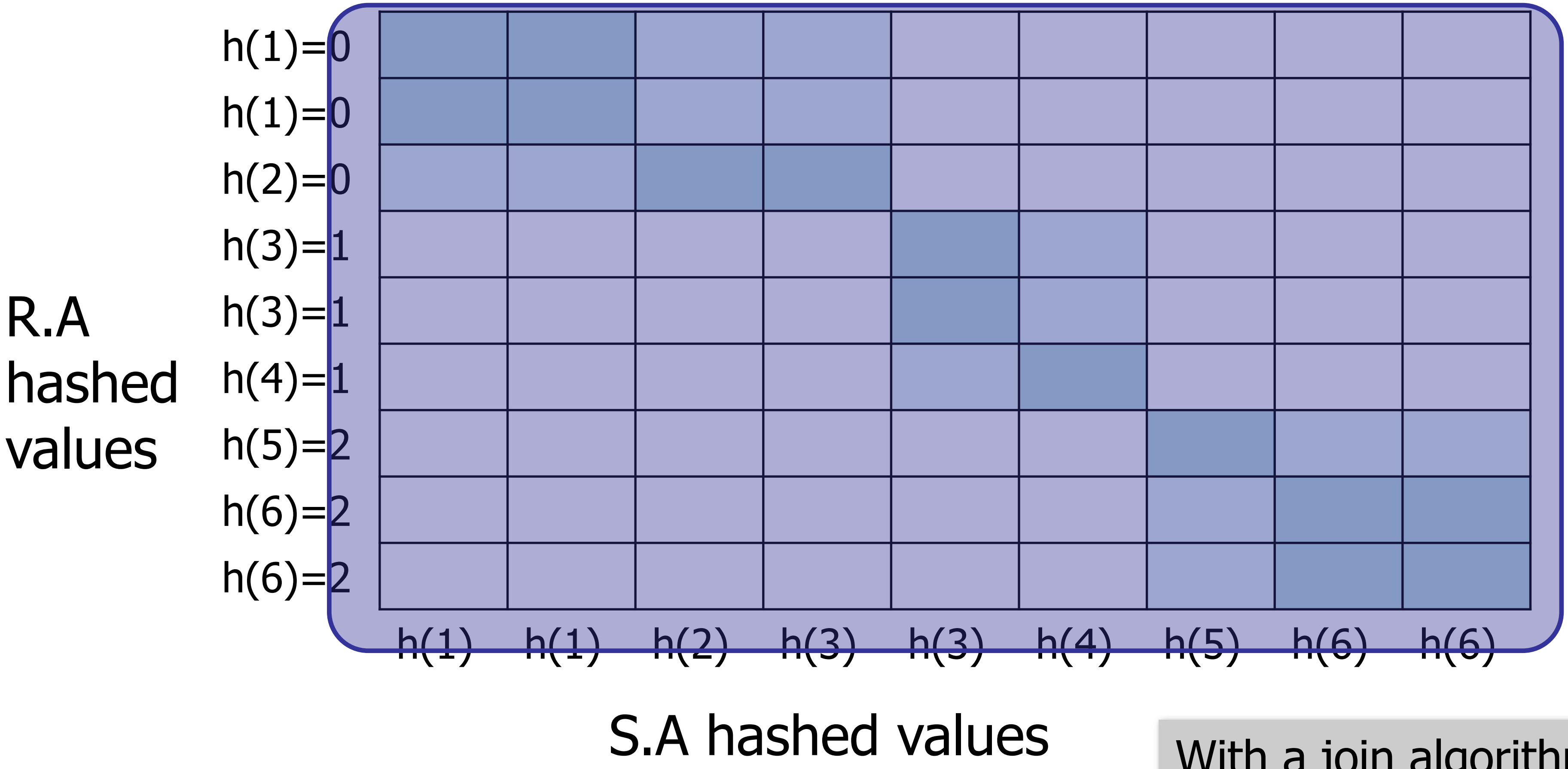


$R \bowtie S \text{ on } A$

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

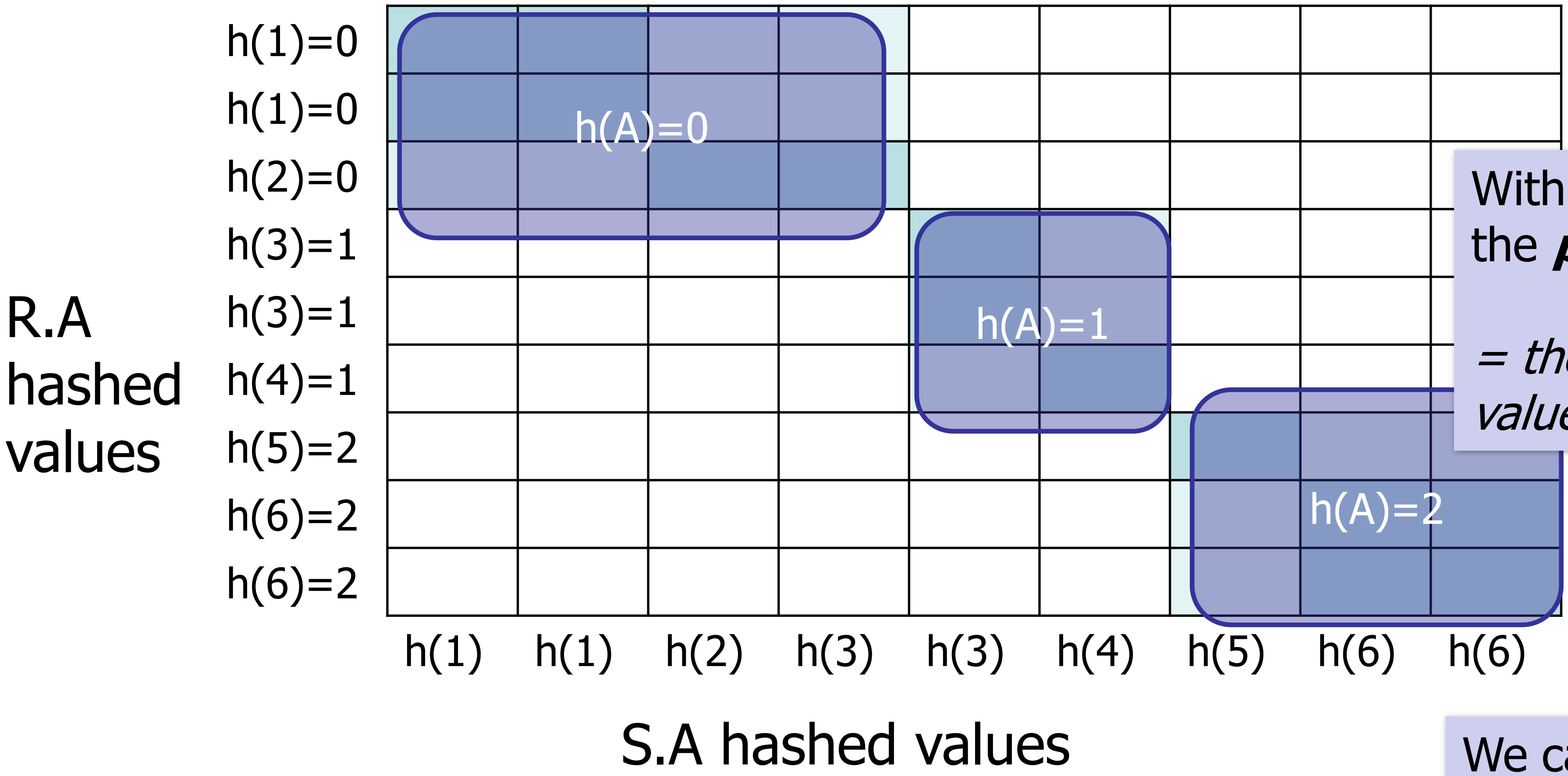
$R \bowtie S \ on \ A$



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

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

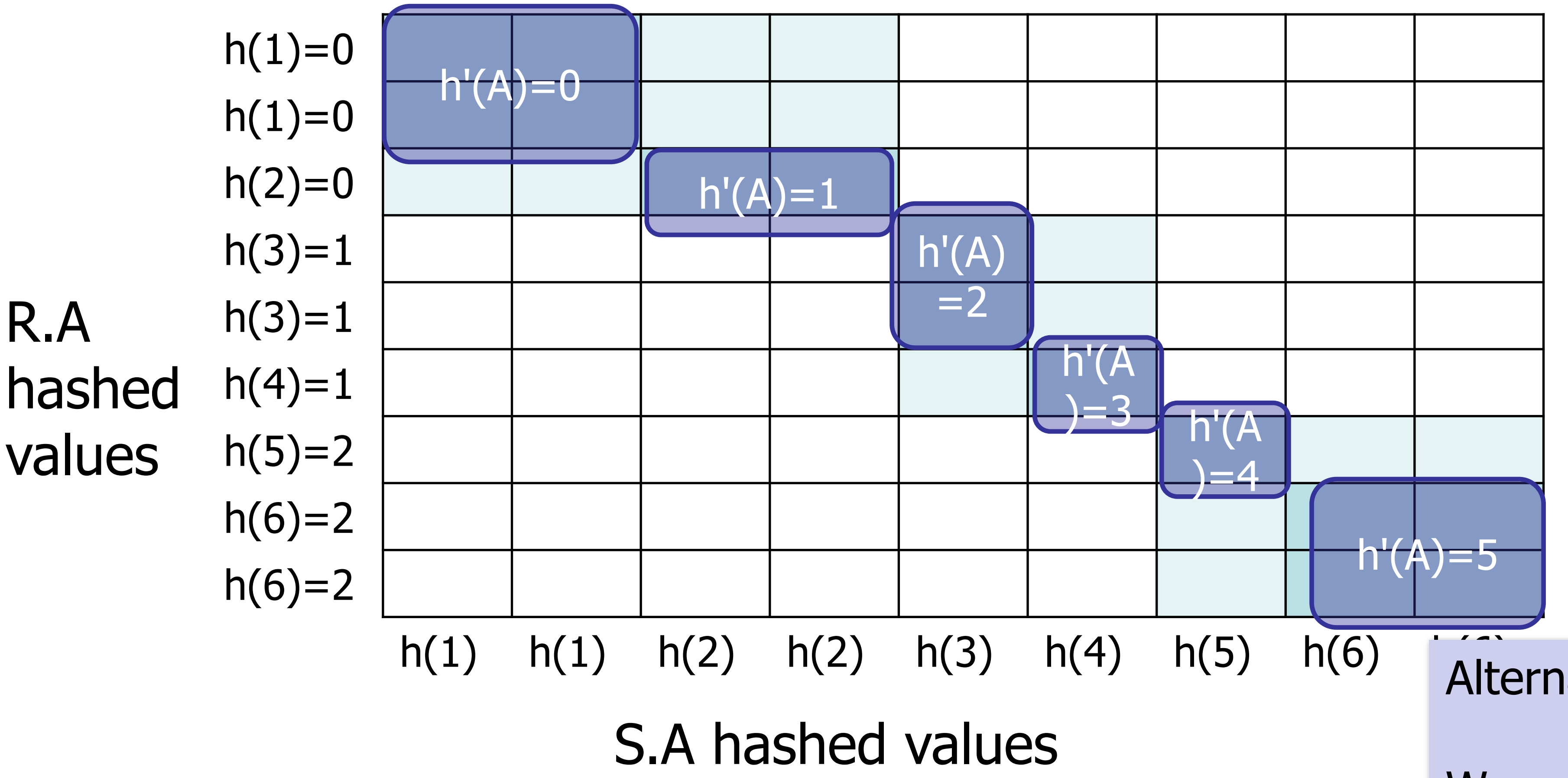


With HJ, we only explore the ***purple*** regions

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

We can apply BNLJ to each of these regions

Hash Join Phase 2: Matching



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

Alternative to applying BNLJ:

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

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:

$$\sim 3(P(R)+P(S)) + \mathit{OUT}$$

- ***"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?

Summary

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

Nested Loop Join (NLJ) P(R) + T(R)*P(S) + OUT

Block Nested Loop Join (BNLJ) $\mathrm{P}(R) + \frac{P(R)}{B-1} P(S)$ + OUT

Index Nested Loop Join (INLJ) P(R) + T(R)*L + OUT

Sort Merge Join (SMJ)

Hash Join (HJ) 3(P(R)+P(S)) + OUT!

With lot of memory

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!