Disclaimer:
These slides are from a University of Wisconsin Database Course available online

What you will learn about in this section

1. Sort-Merge Join (SMJ)
2. Hash Join (HJ)
3. SMJ vs. HJ

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

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

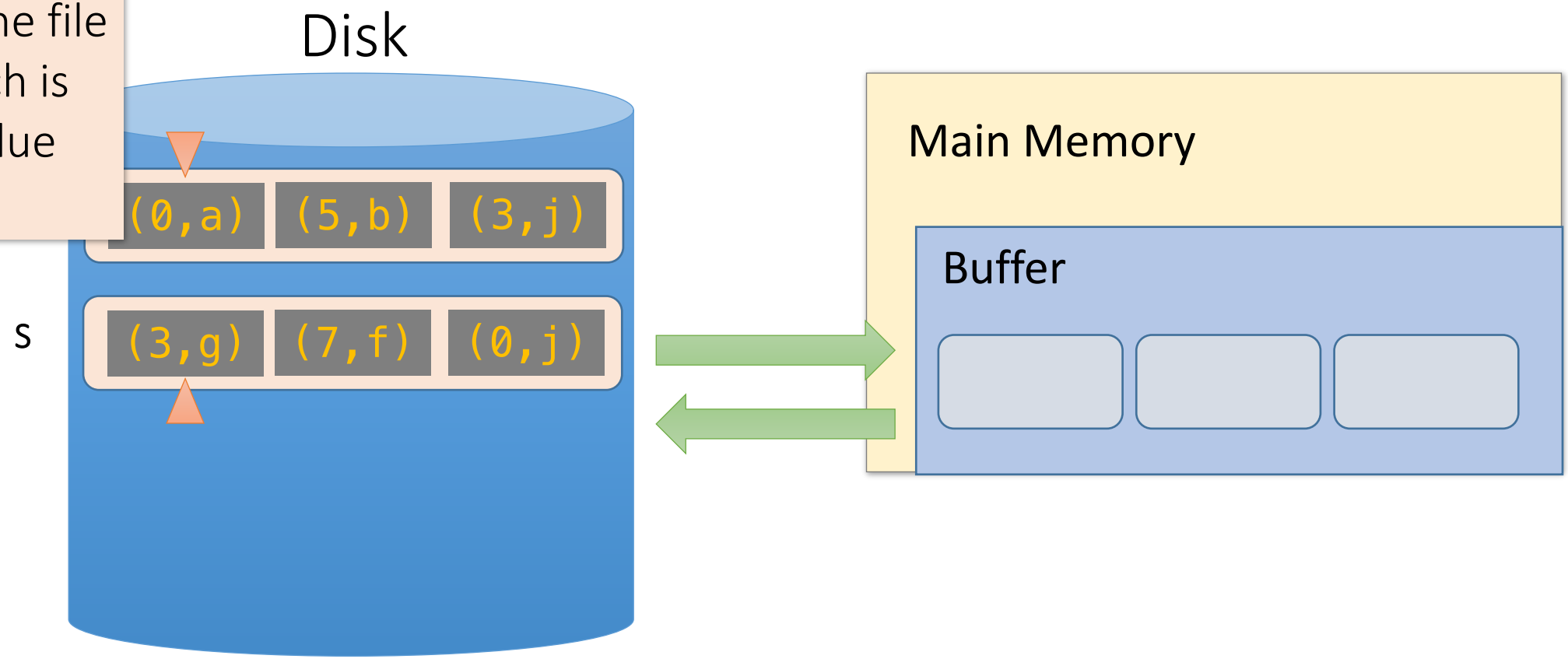
Note that we are only considering equality join conditions here

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

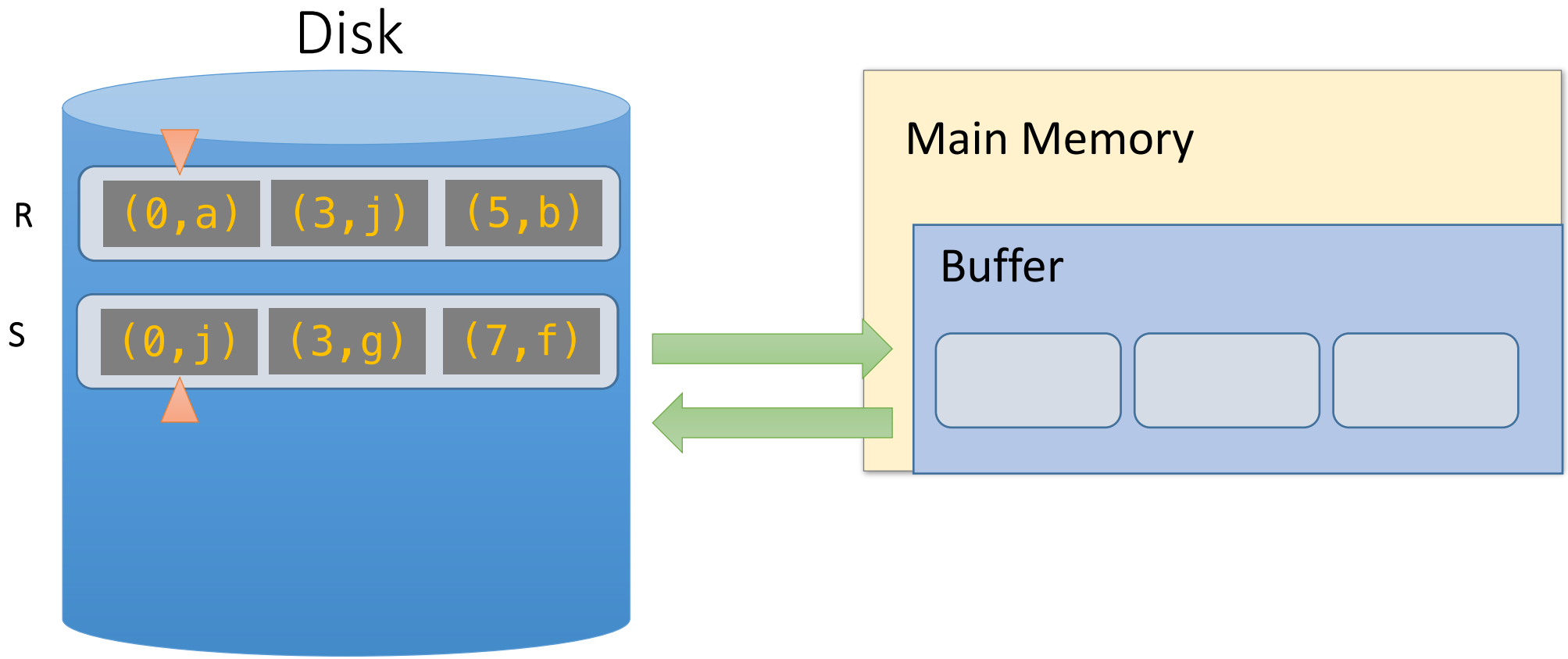
- For simplicity: Let each page be ***one tuple***, and let the first value be A

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



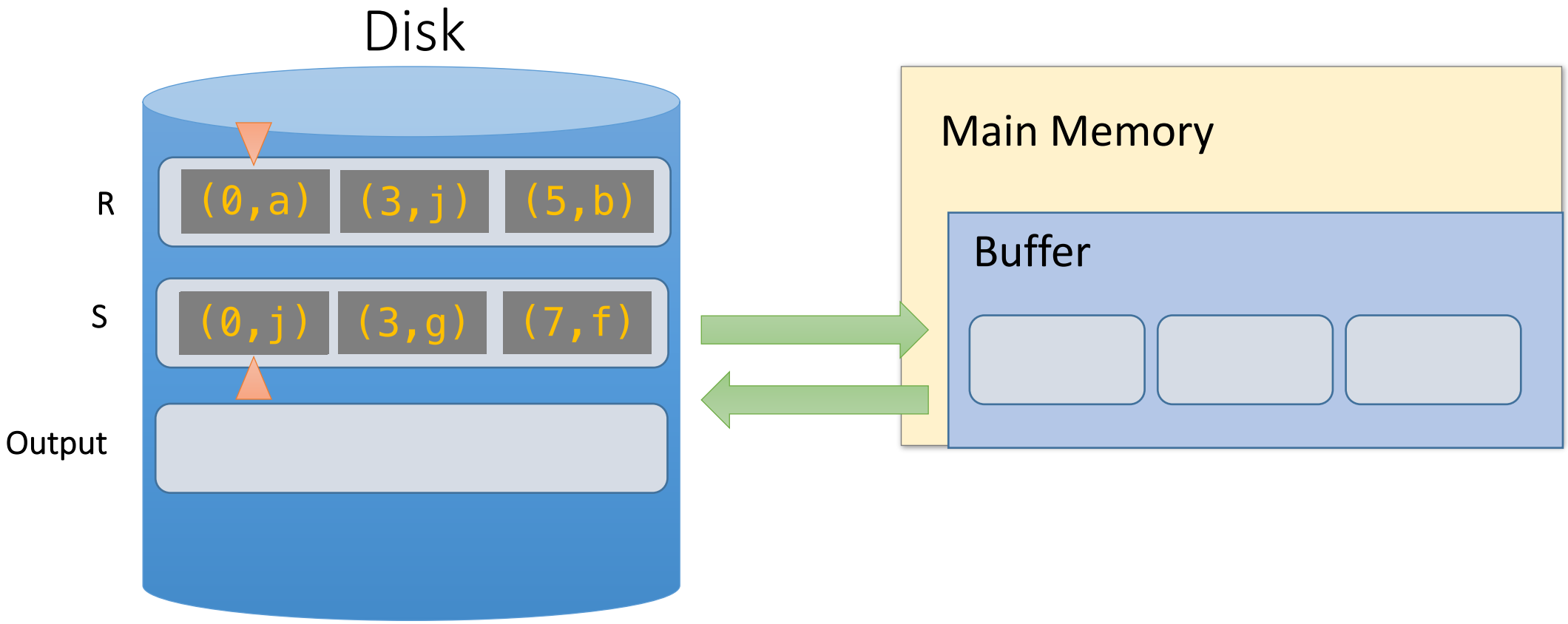
SMJ Example: $R \bowtie S$ on A with 3 page buffer

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



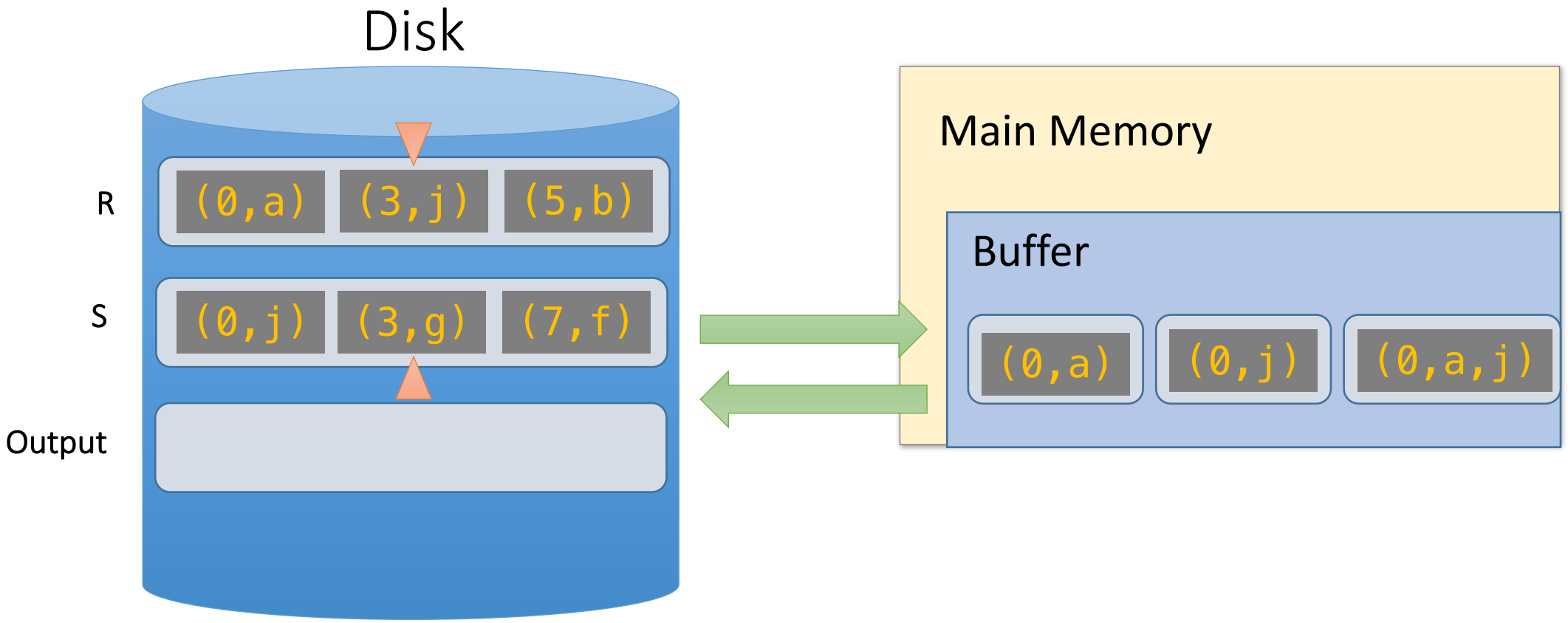
SMJ Example: $R \bowtie S$ on A with 3 page buffer

2. Scan and “merge” on join key!



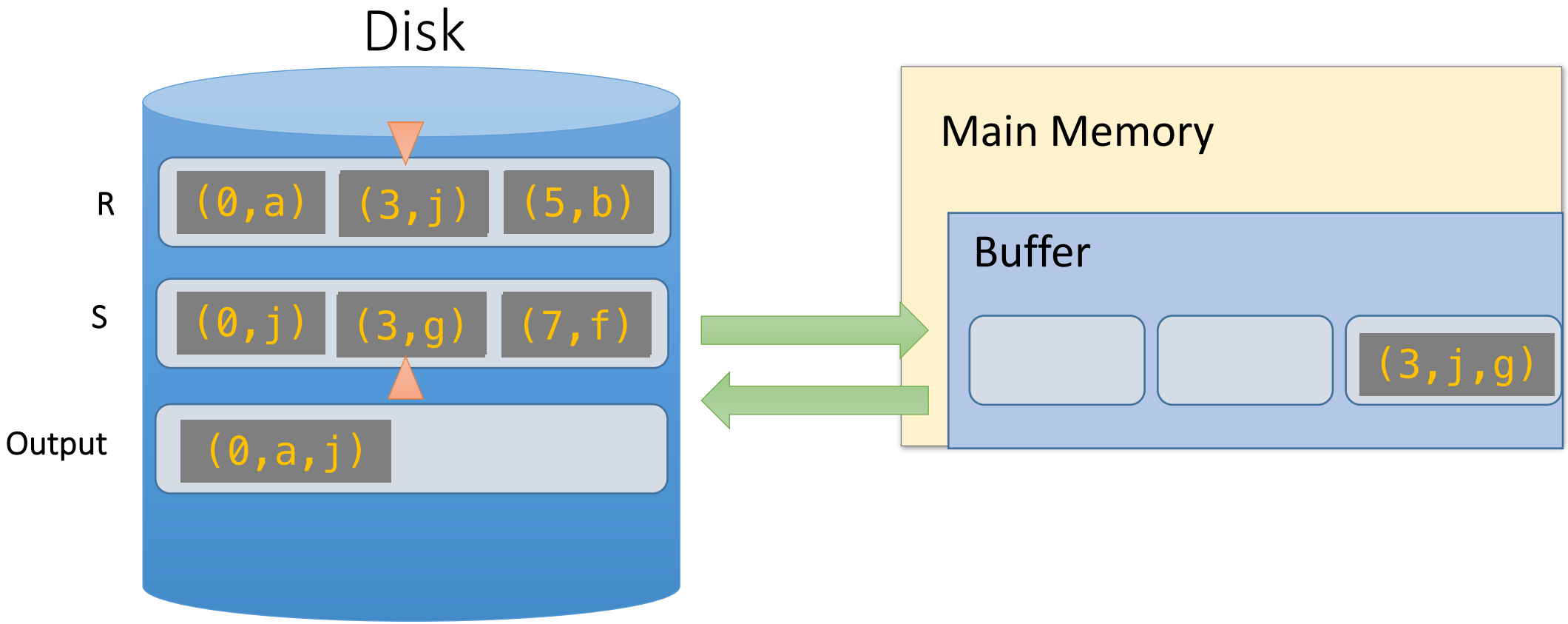
SMJ Example: $R \bowtie S$ on A with 3 page buffer

2. Scan and “merge” on join key!



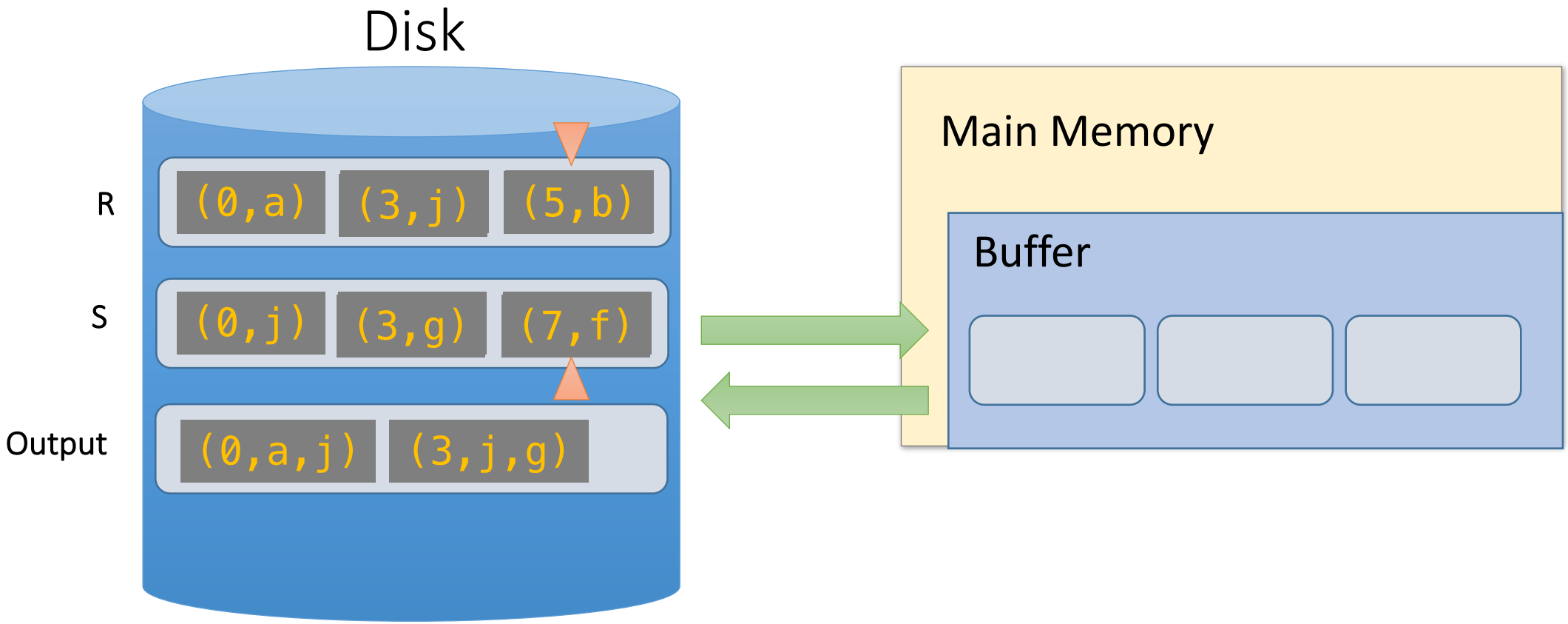
SMJ Example: $R \bowtie S$ on A with 3 page buffer

2. Scan and “merge” on join key!



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

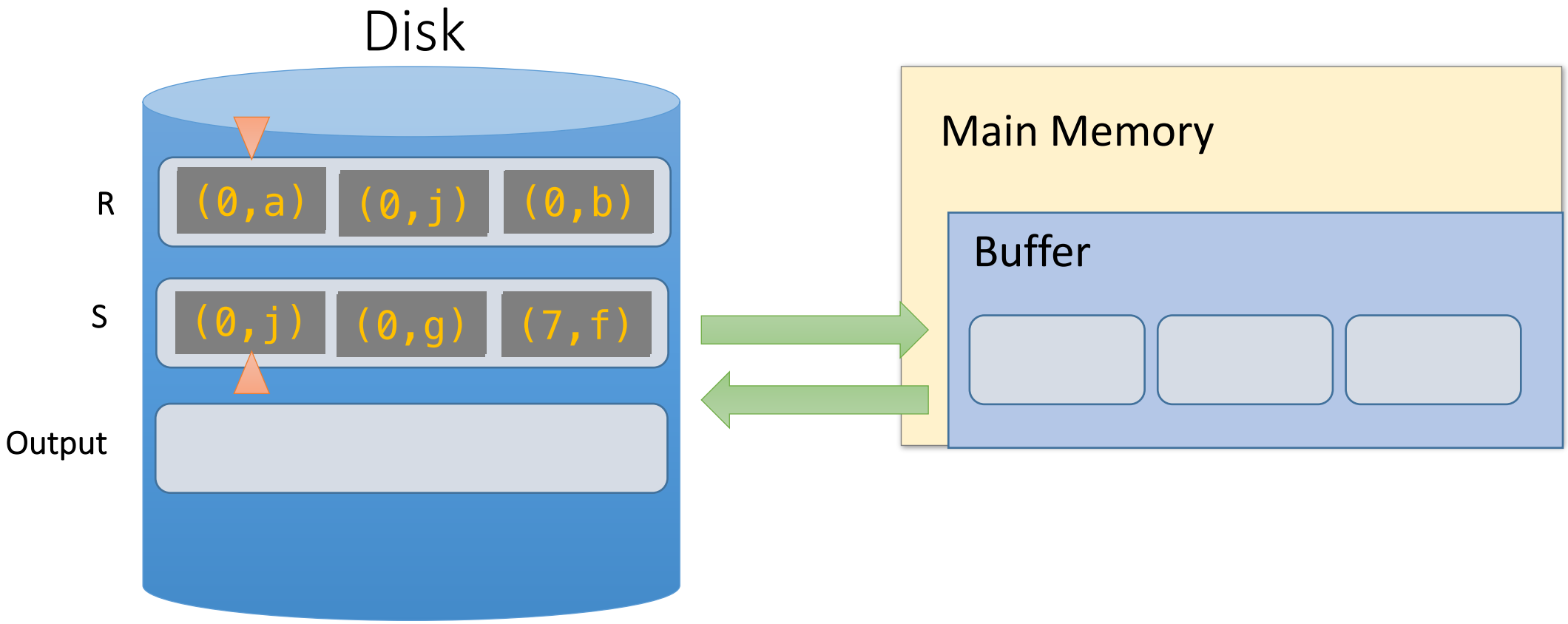
2. Done!



What happens with duplicate join keys?

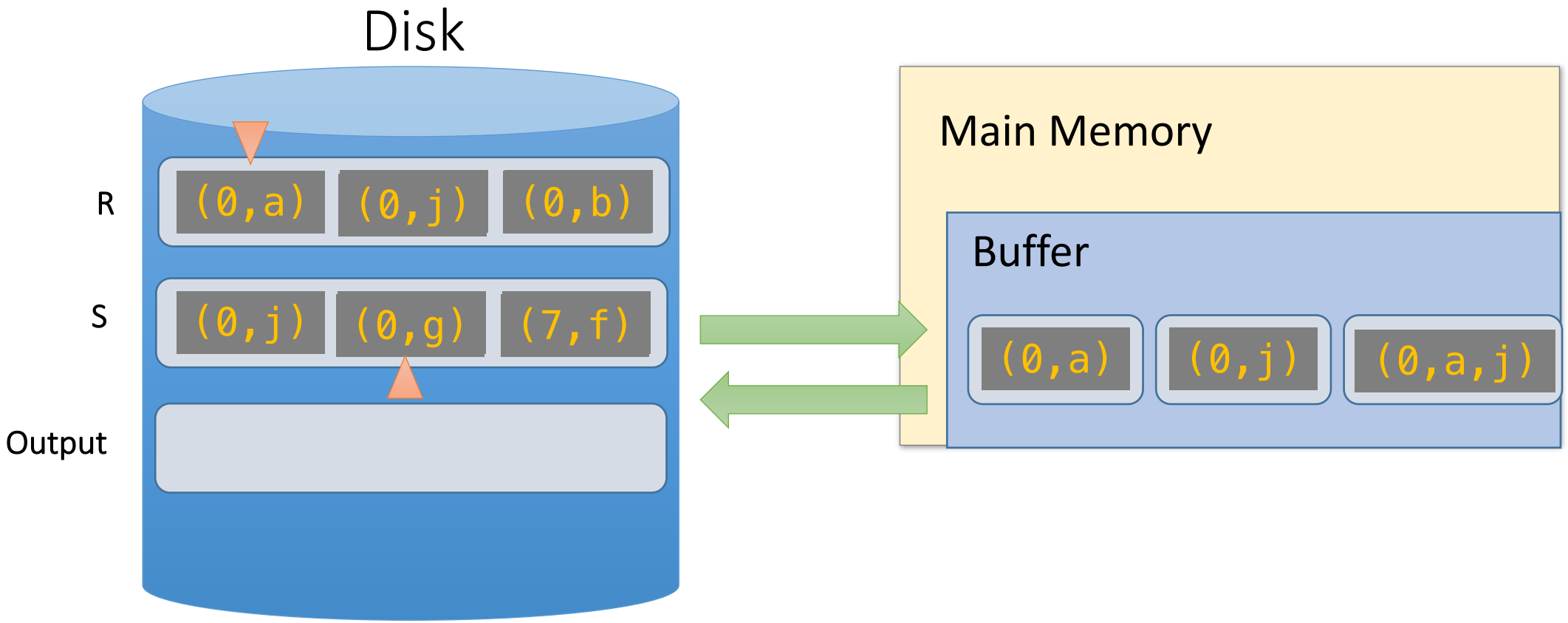
Multiple tuples with Same Join Key: “Backup”

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



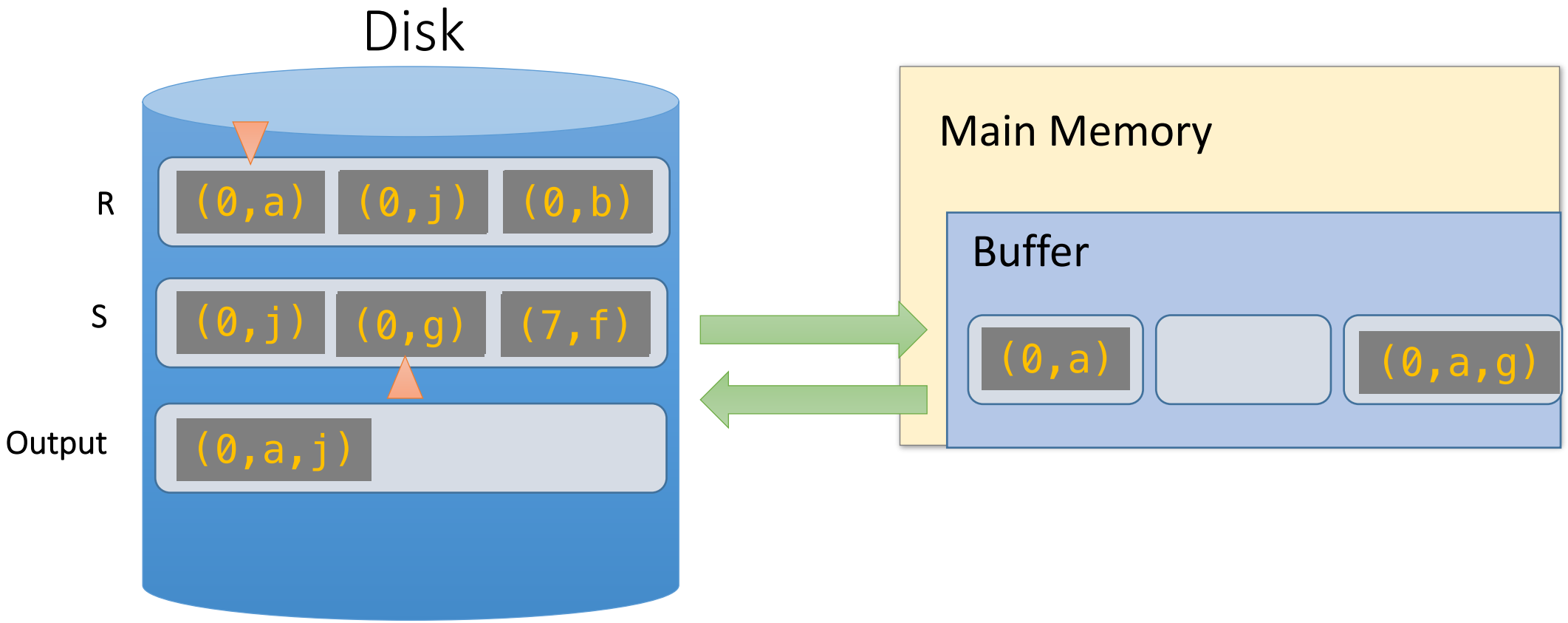
Multiple tuples with Same Join Key: “Backup”

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



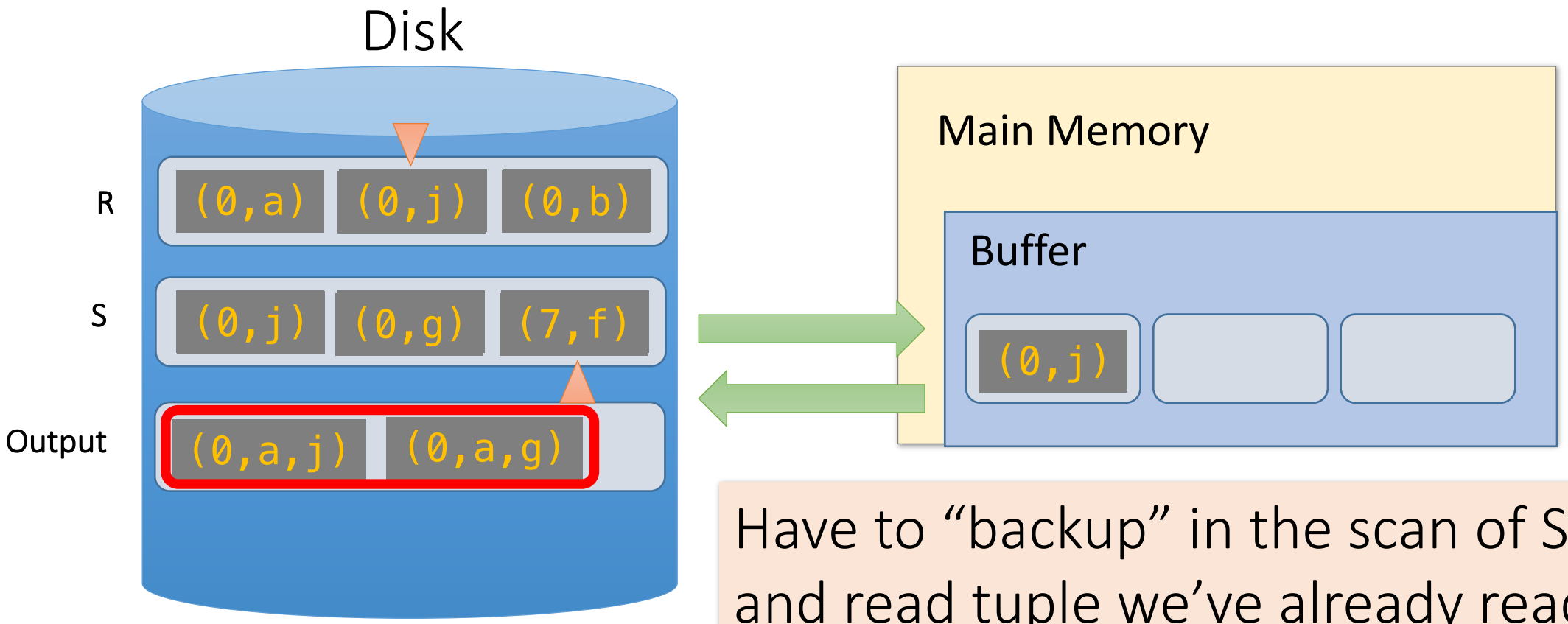
Multiple tuples with Same Join Key: “Backup”

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…



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)
  - Can "zig-zag" (see animation)

SMJ: Total cost

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

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

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

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

SMJ vs. BNLJ: Steel Cage Match

- 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 = **6,000 IOs**
  - Merge phase 1000 + 500 = 1,500 IOs
  - **= 7,500 IOs + OUT**

What is BNLJ?

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

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

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

A Simple Optimization: Merges Merged!

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

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

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

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

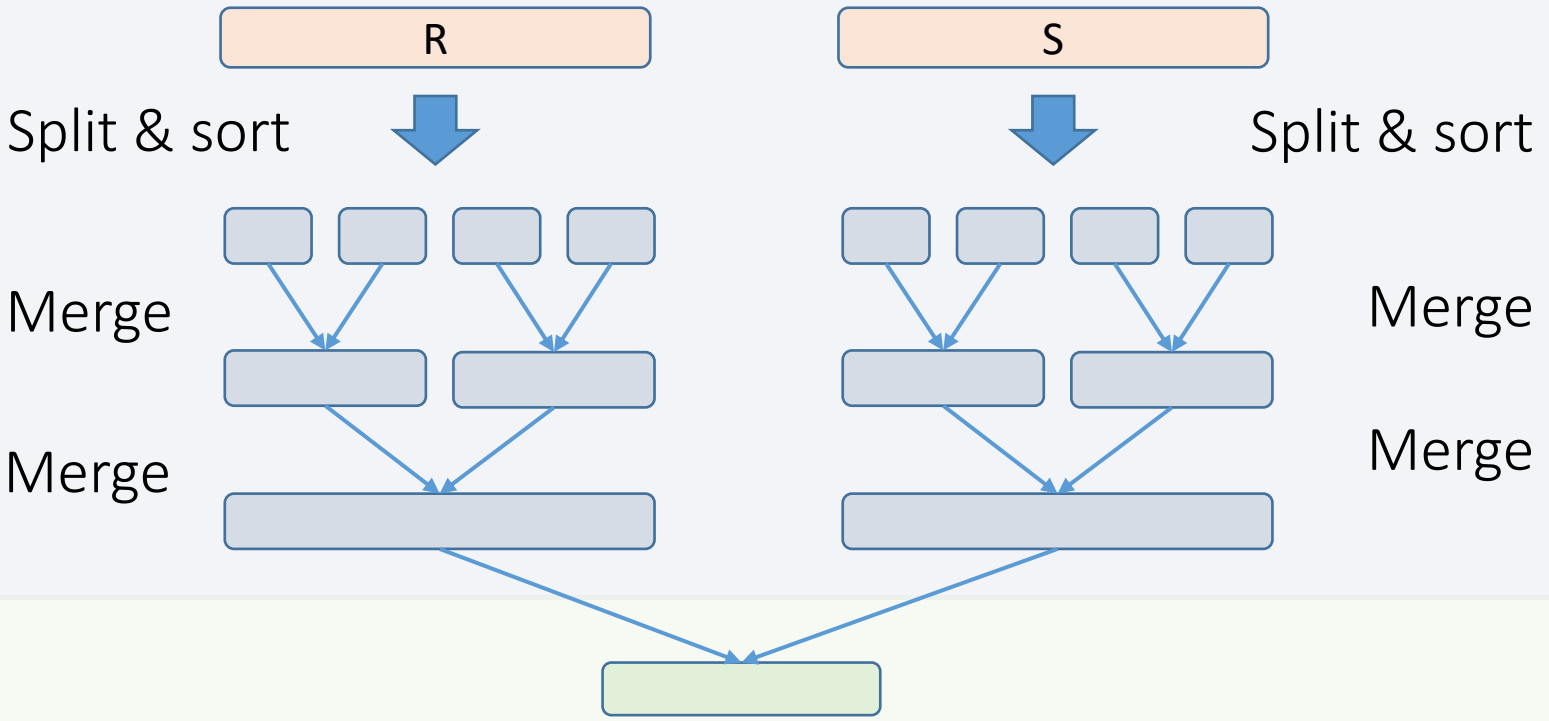
  - OR, we could combine them: do **one** B-way merge and complete the join!

Un-Optimized SMJ

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

Unsorted input relations

Sort Phase
(Ext. Merge Sort)

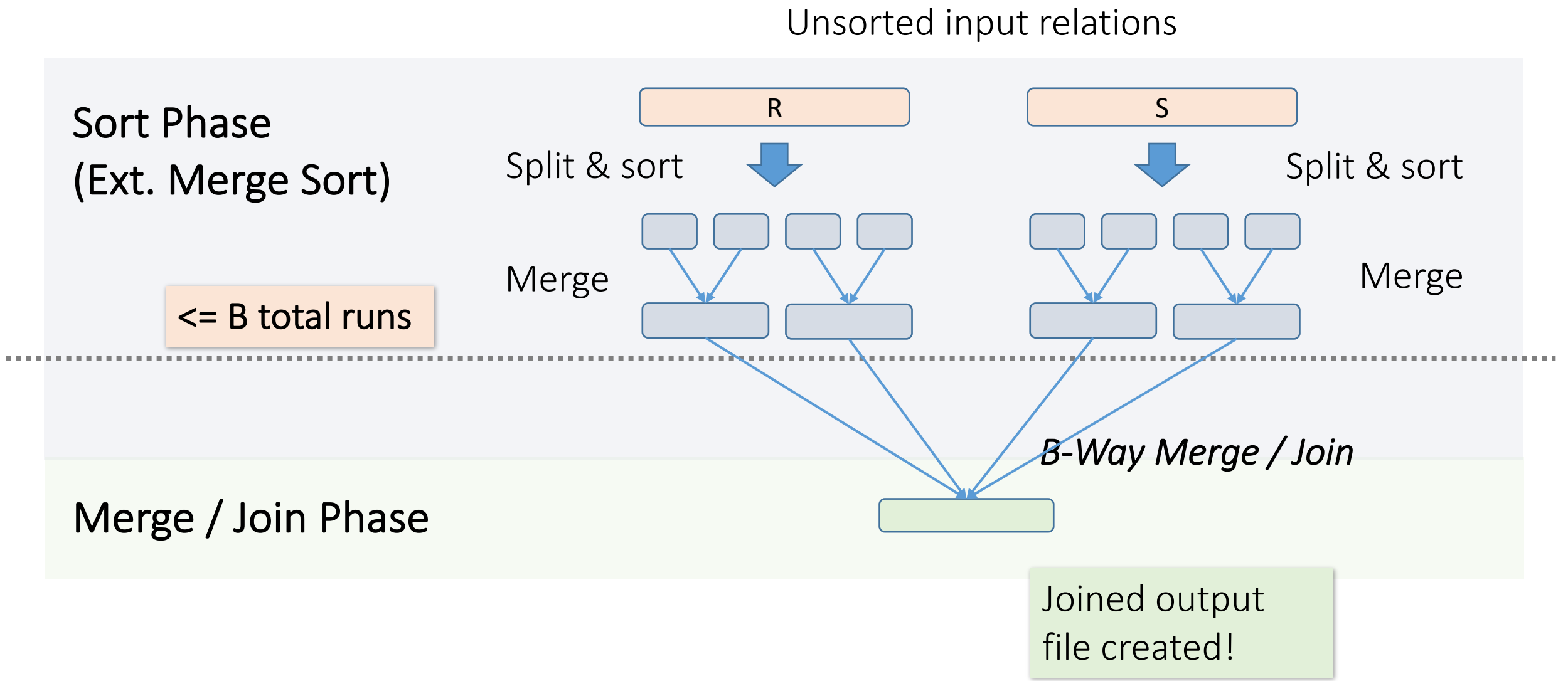


Merge / Join Phase

Joined output file created!

Simple SMJ Optimization

Given *B+1* buffer pages



Simple SMJ Optimization

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

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

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

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

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 != y but $h_B(x) = h_B(y)$
  - Note however that it will **never** occur that x = y but $h_B(x)$!= $h_B(y)$

- 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

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

Note again that we are only considering equality constraints here

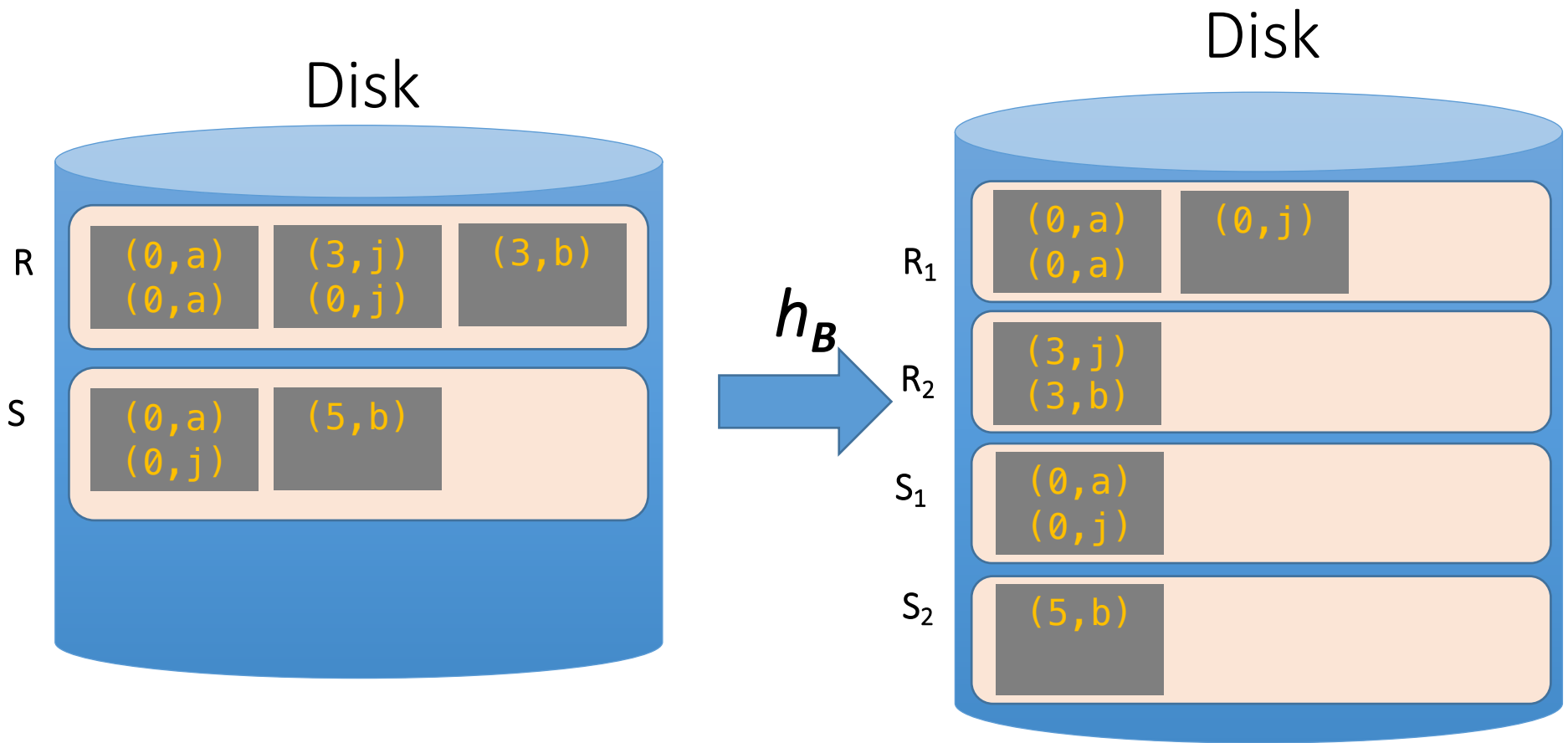
1. **Partition Phase:** Using one (shared) hash function $\boldsymbol{h_B}$, partition R *and* S into ***B*** buckets

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

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

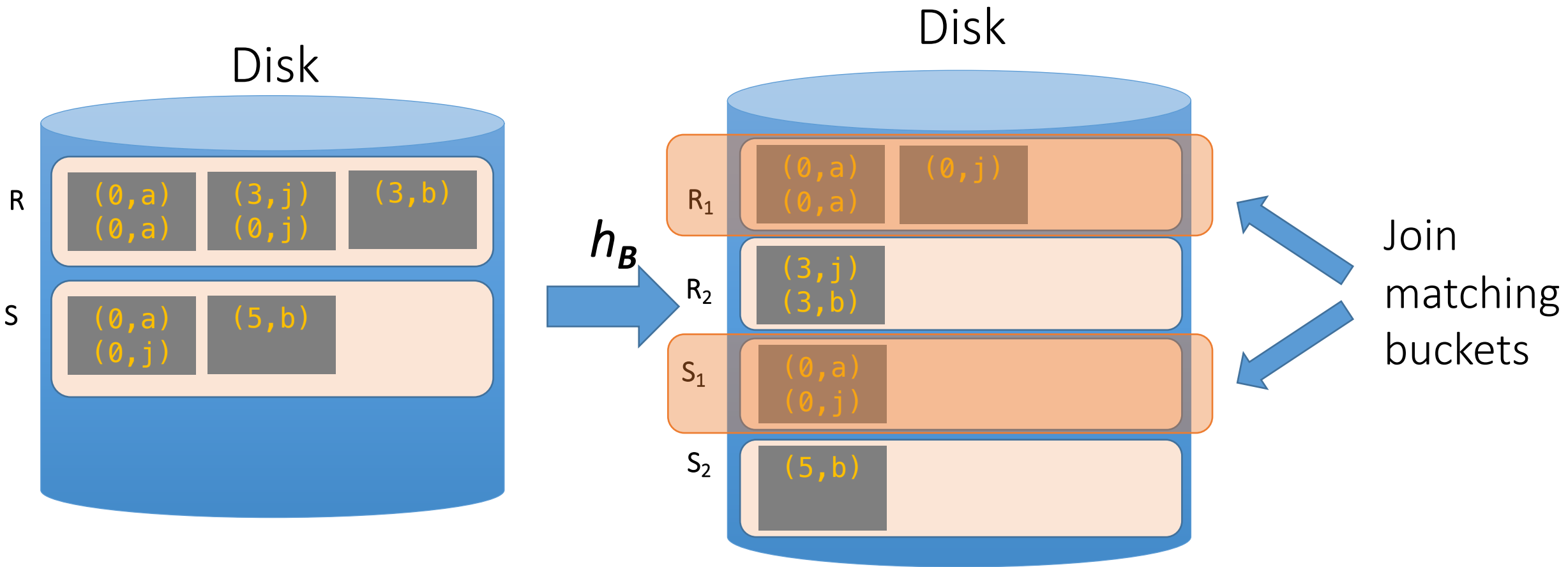


Note our new convention: pages each have two tuples (one per row)

More detail in a second…

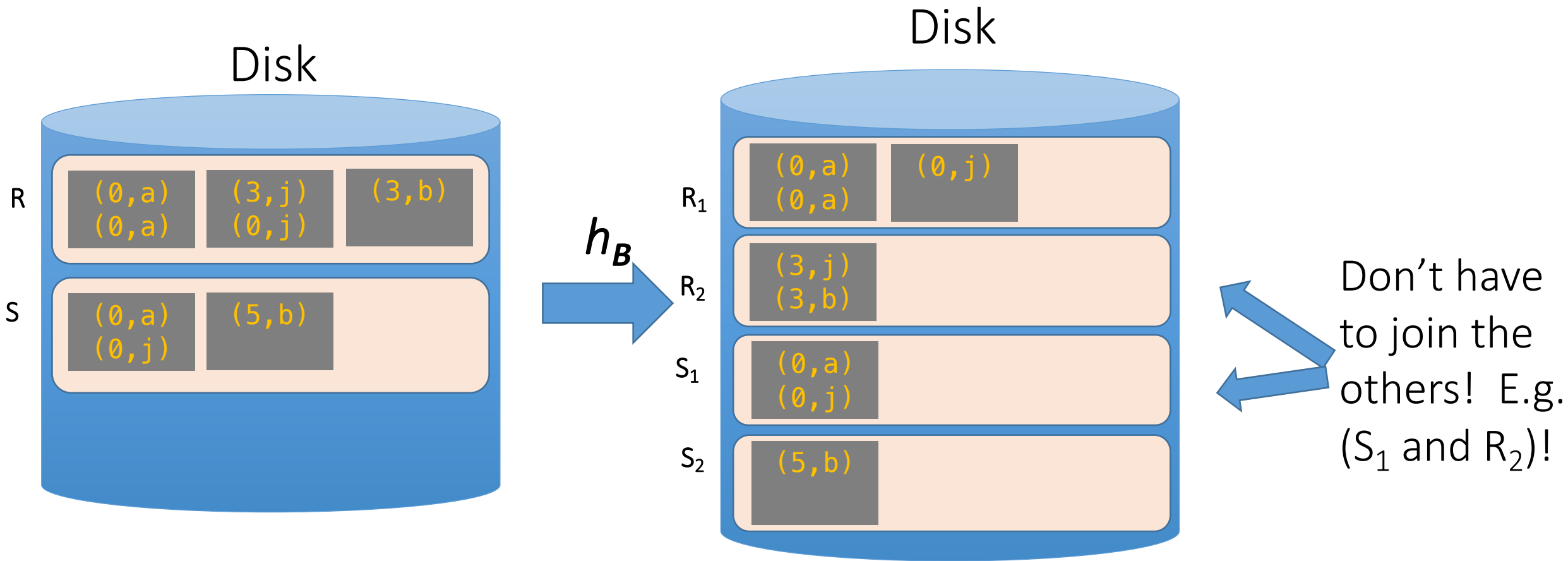
Hash Join: High-level procedure

2. Matching Phase: Take pairs of buckets whose tuples have the same values for h_B, and join 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 $h_B(t.A) = h_B(t'.A)$ 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 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

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**
  - ***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),* $\boldsymbol{P(R) \leq B-1}$!
  - And more generally, being able to fit bucket in memory is advantageous

Recall for BNLJ:

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

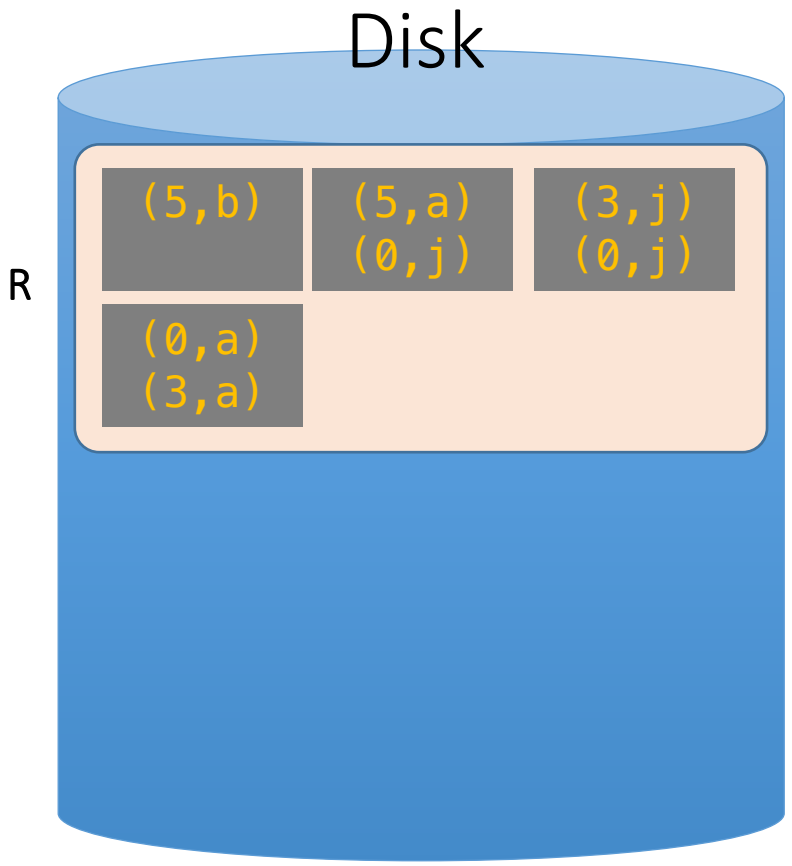
- We can keep partitioning buckets that are > B-1 pages, until they are $\leq \boldsymbol{B-1}$ **pages**
  - Using a new hash key which will split them…

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)



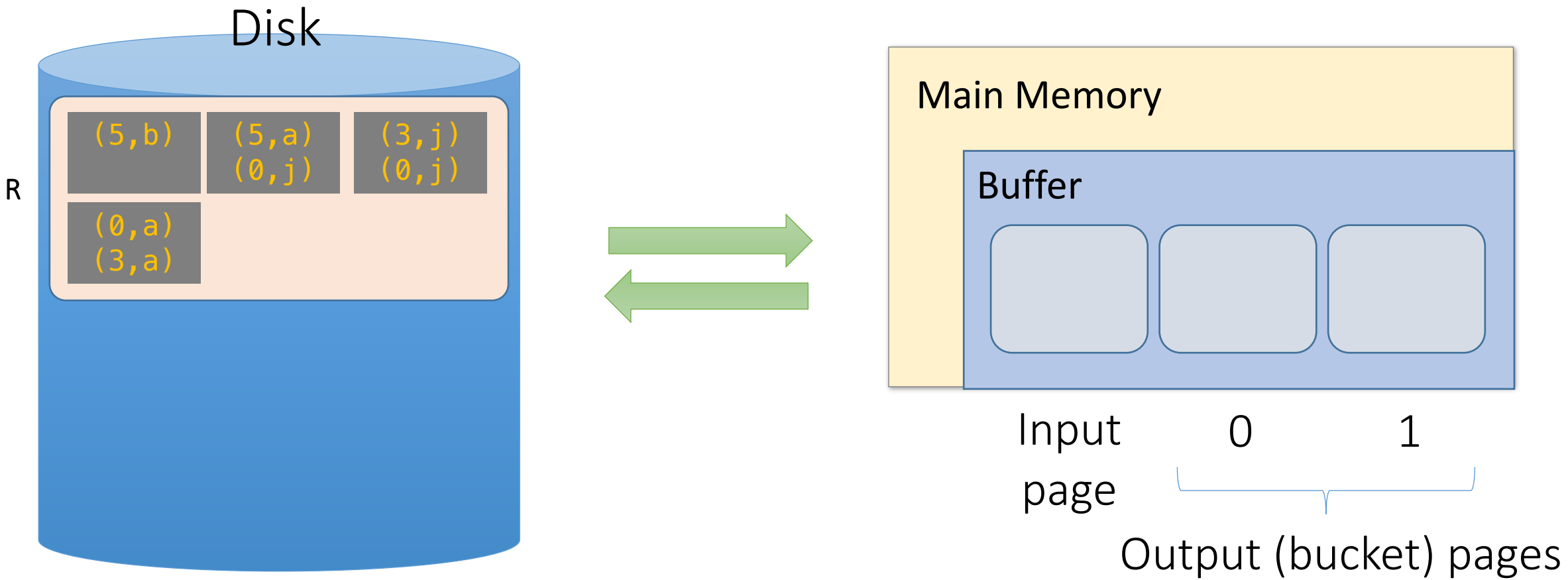
For simplicity, we'll look at partitioning one of the two relations- we just do the same for the other relation!

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

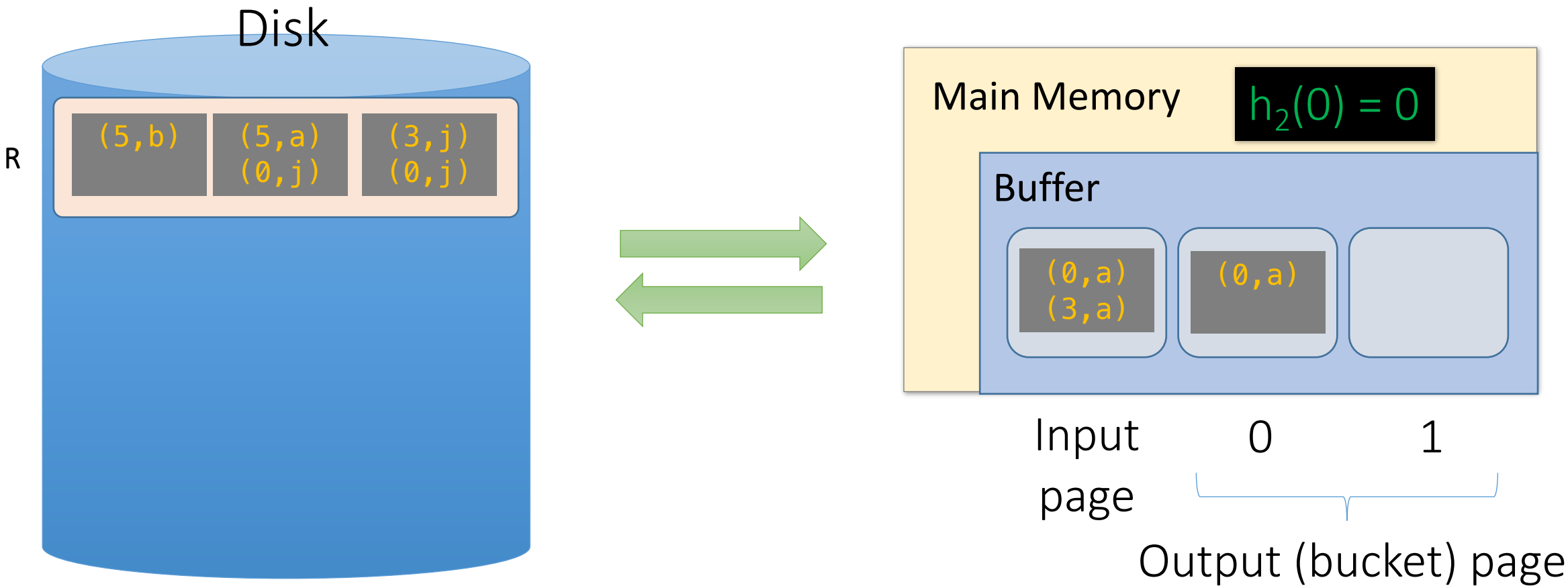
1. We read pages from R into the “input” page of the buffer…



Hash Join Phase 1: Partitioning

Given *B+1 = 3* buffer pages

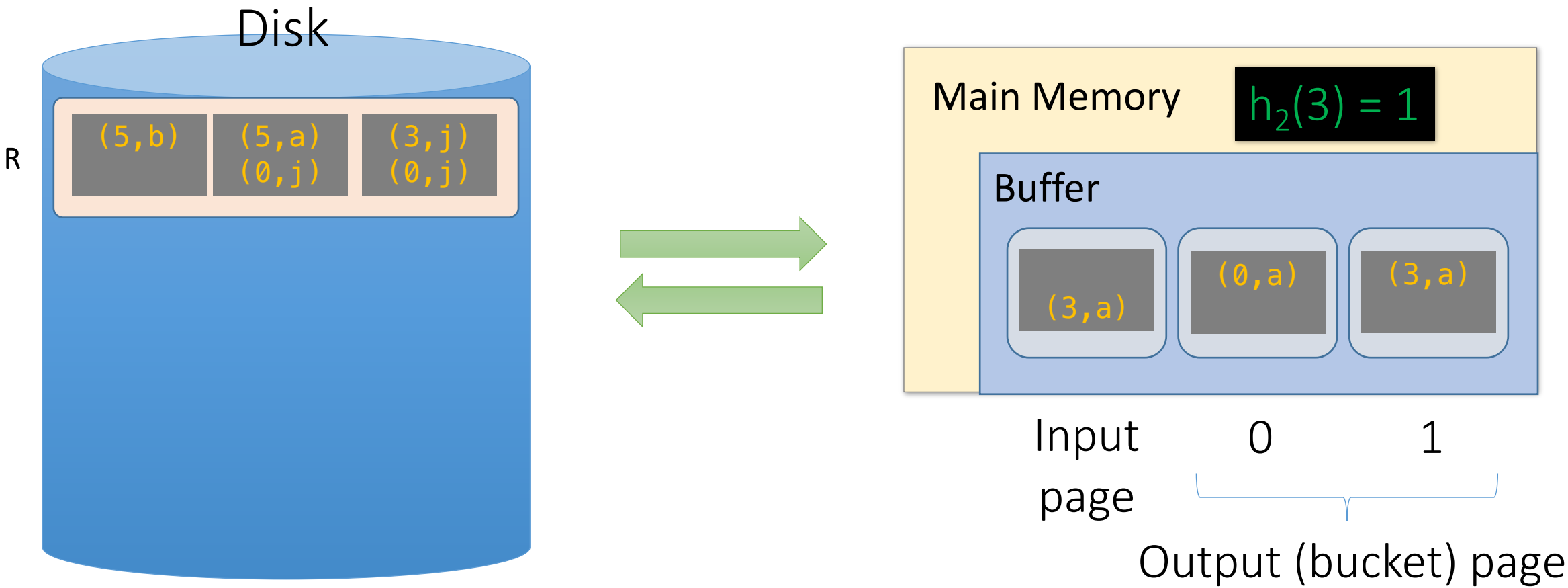
2. Then we use **hash function h_2** to sort into the buckets, which each have one page in the buffer



Hash Join Phase 1: Partitioning

Given *B+1 = 3* buffer pages

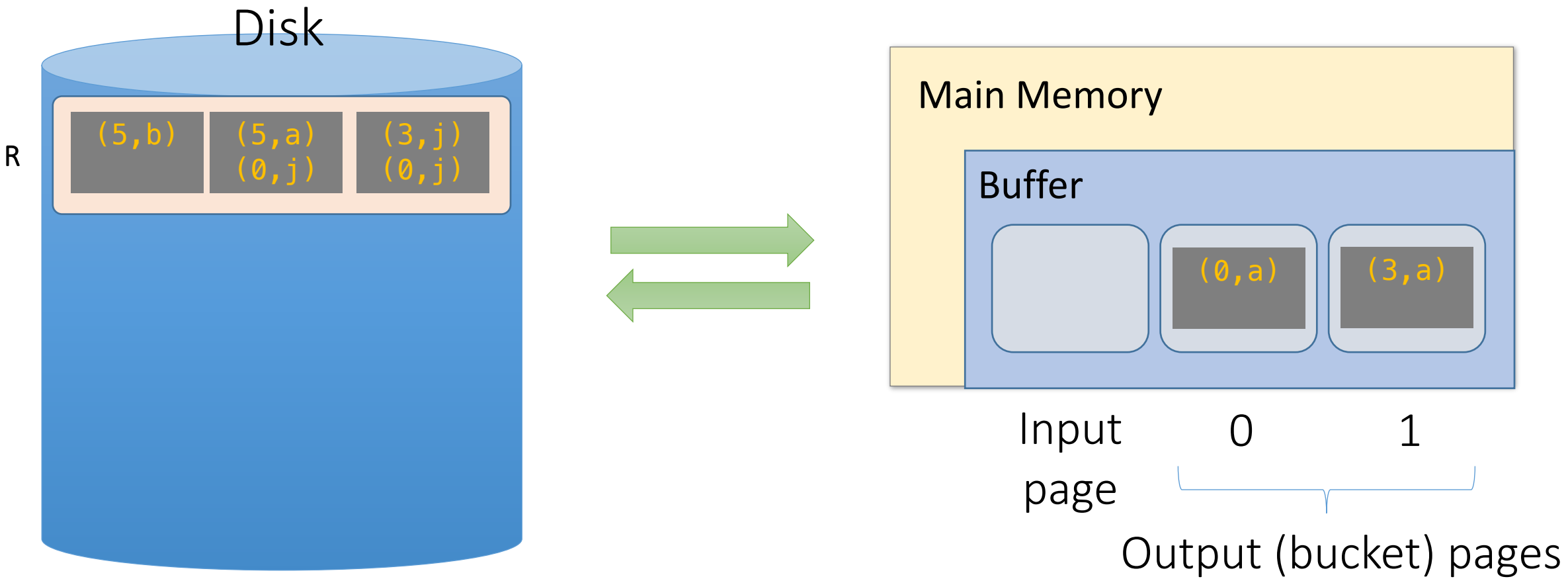
2. Then we use **hash function h_2** to sort into the buckets, which each have one page in the buffer



Hash Join Phase 1: Partitioning

Given *B+1 = 3* buffer pages

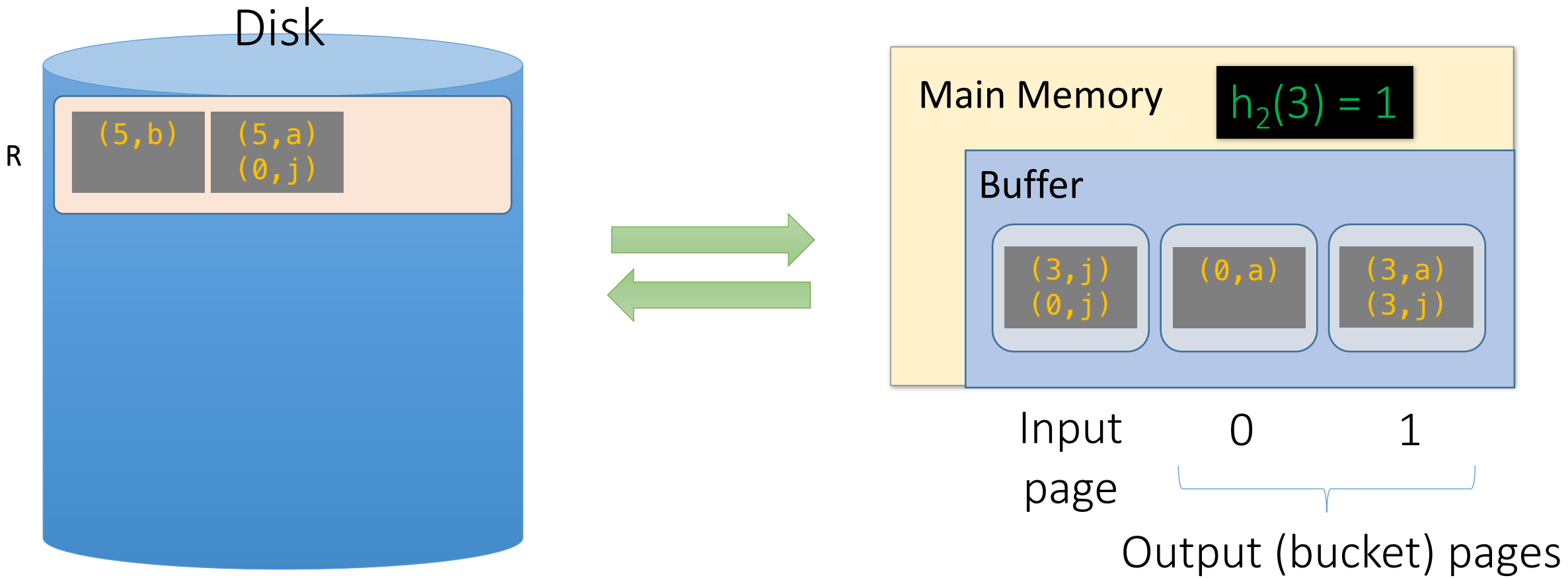
3. We repeat until the buffer bucket pages are full…



Hash Join Phase 1: Partitioning

Given *B+1 = 3* buffer pages

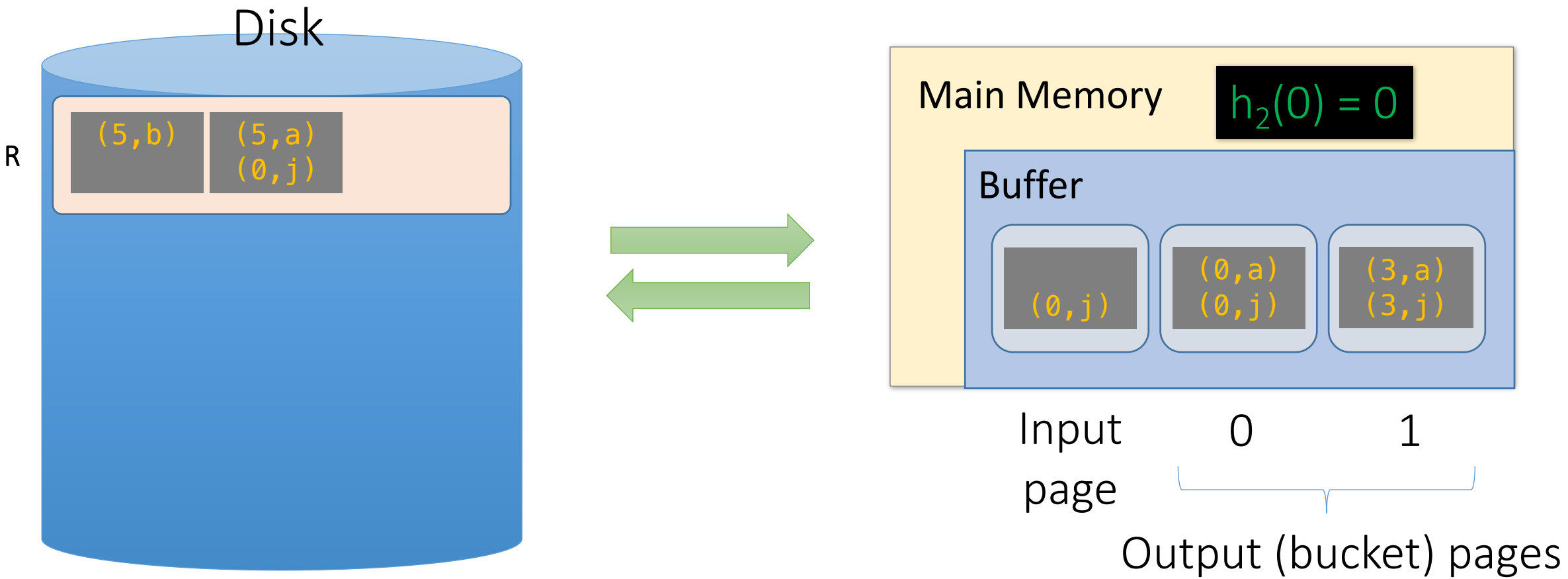
3. We repeat until the buffer bucket pages are full…



Hash Join Phase 1: Partitioning

Given *B+1 = 3* buffer pages

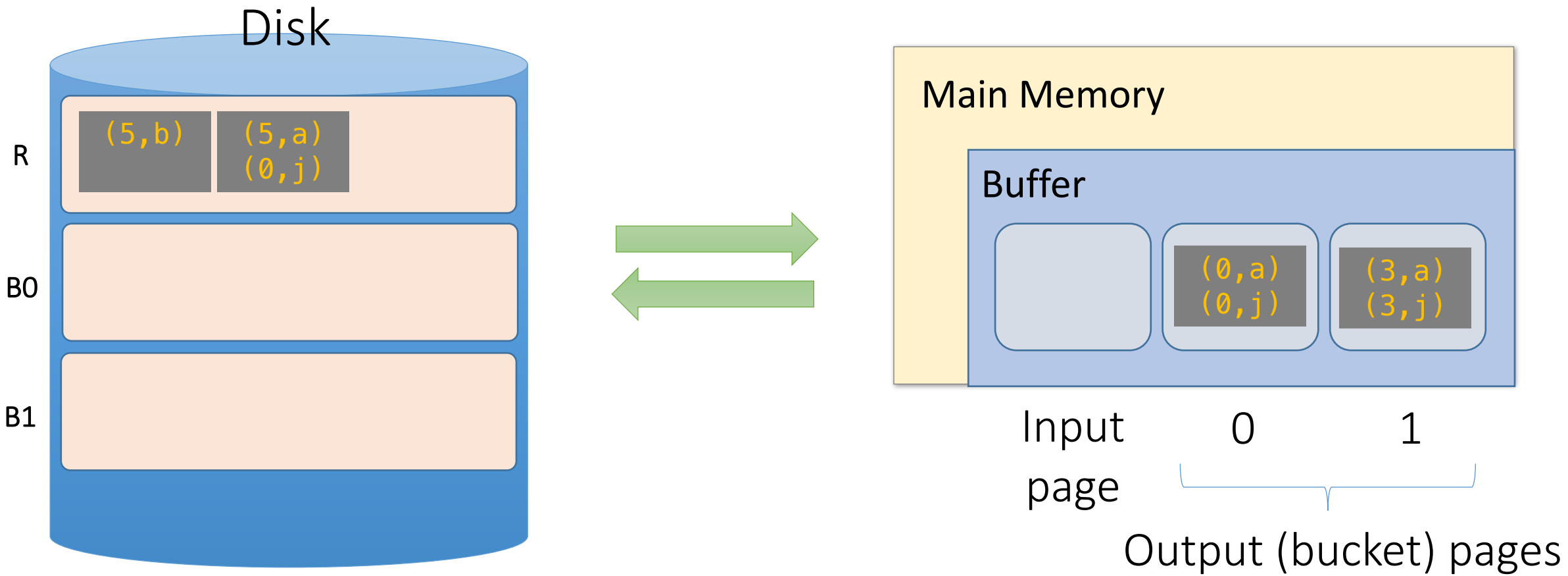
3. We repeat until the buffer bucket pages are full…



Hash Join Phase 1: Partitioning

Given *B+1 = 3* buffer pages

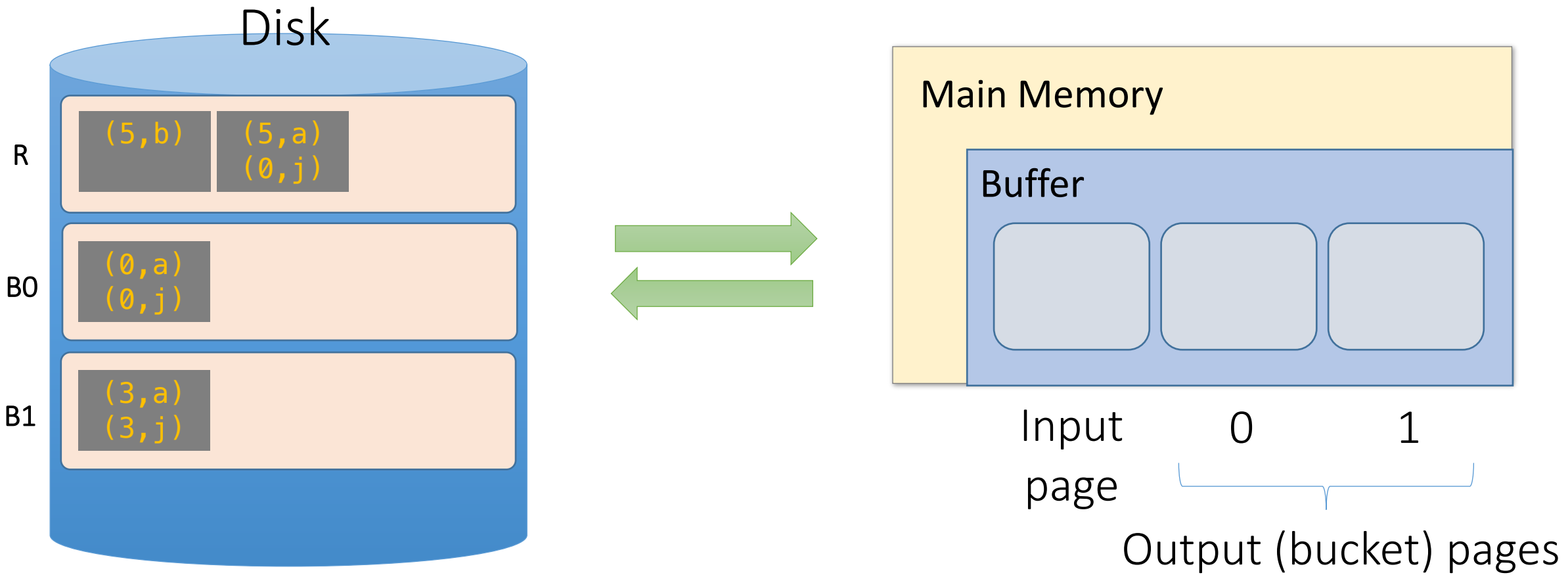
3. We repeat until the buffer bucket pages are full… then flush to disk



Hash Join Phase 1: Partitioning

Given *B+1 = 3* buffer pages

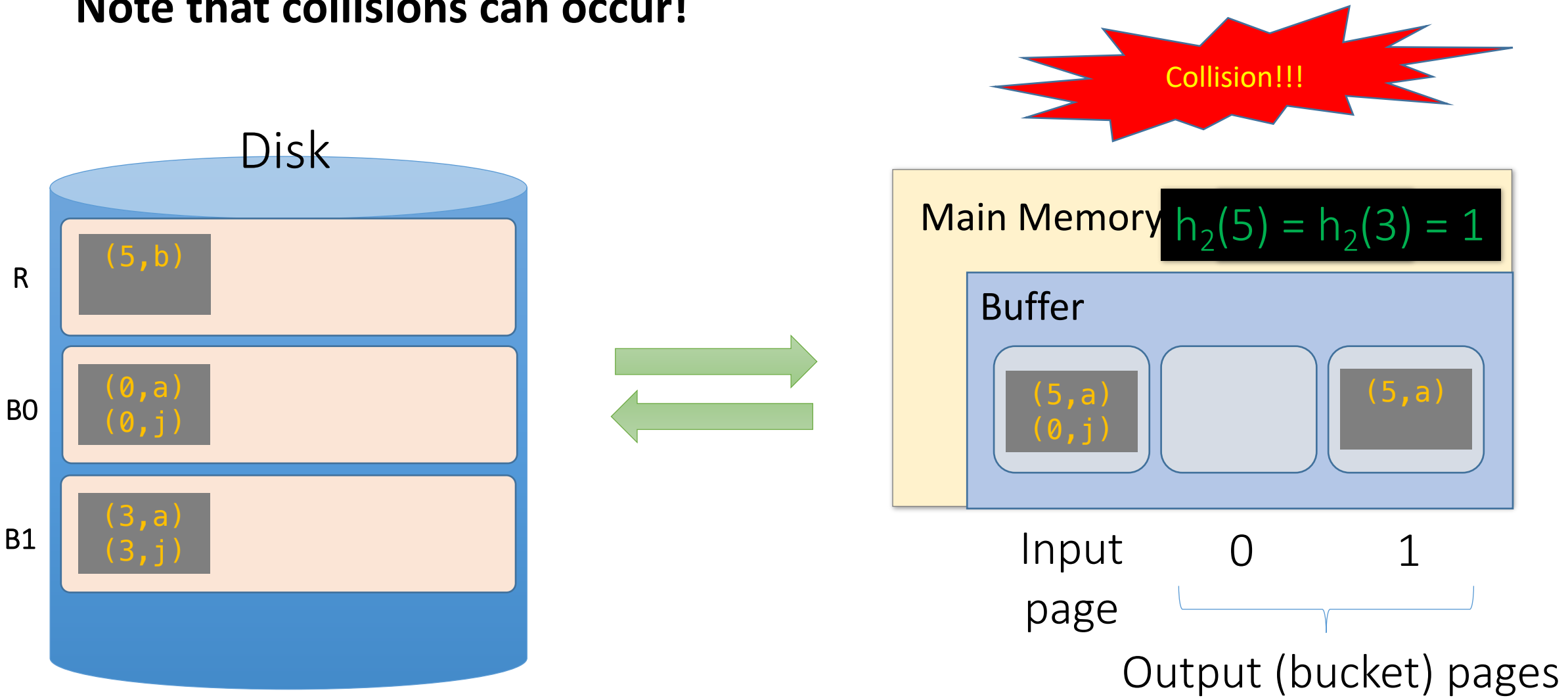
3. We repeat until the buffer bucket pages are full… then flush to disk



Hash Join Phase 1: Partitioning

Given *B+1 = 3* buffer pages

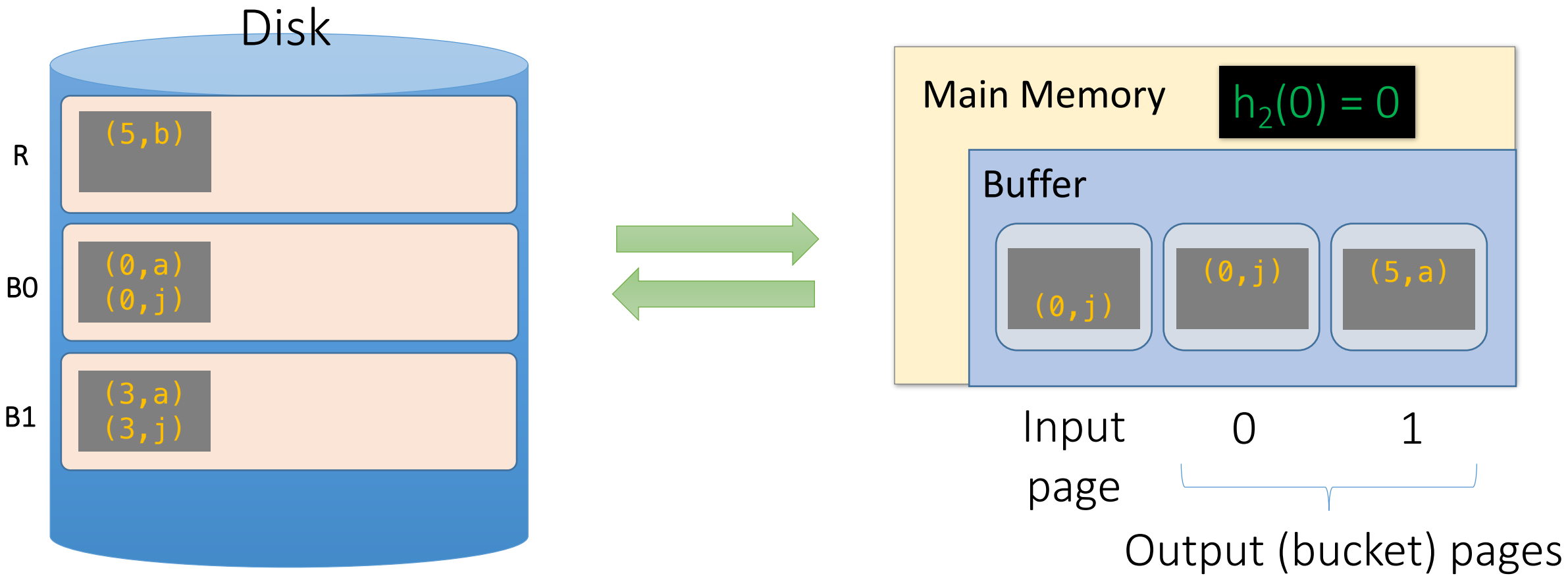
Note that collisions can occur!



Hash Join Phase 1: Partitioning

Given *B+1 = 3* buffer pages

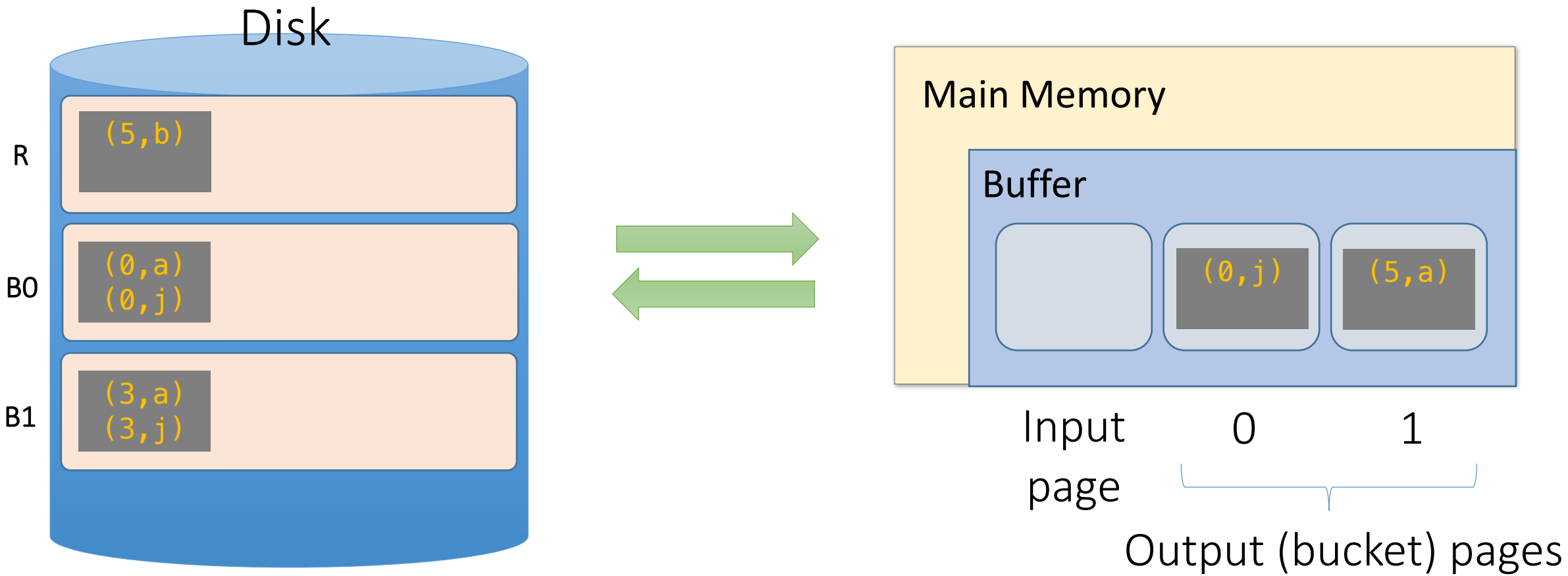
Finish this pass…



Hash Join Phase 1: Partitioning

Given *B+1 = 3* buffer pages

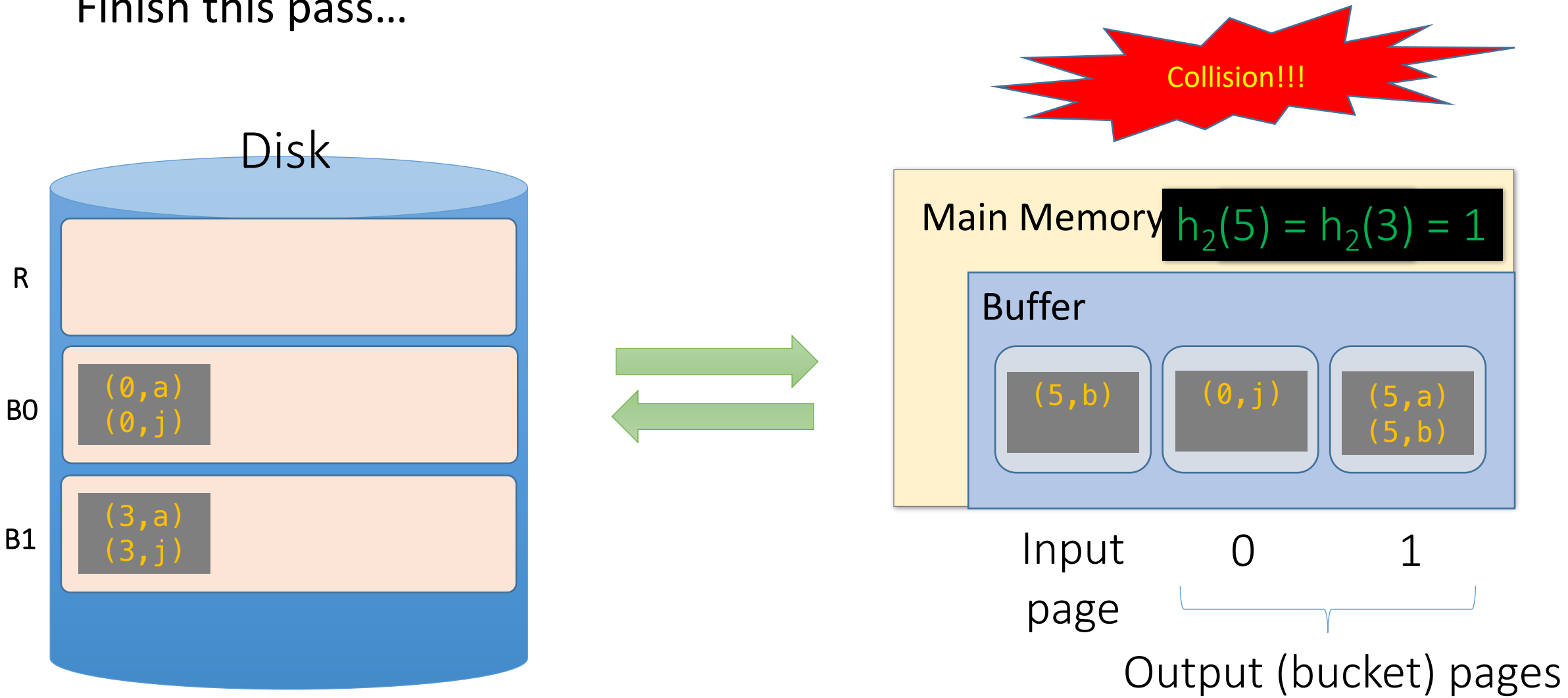
Finish this pass…



Hash Join Phase 1: Partitioning

Given *B+1 = 3* buffer pages

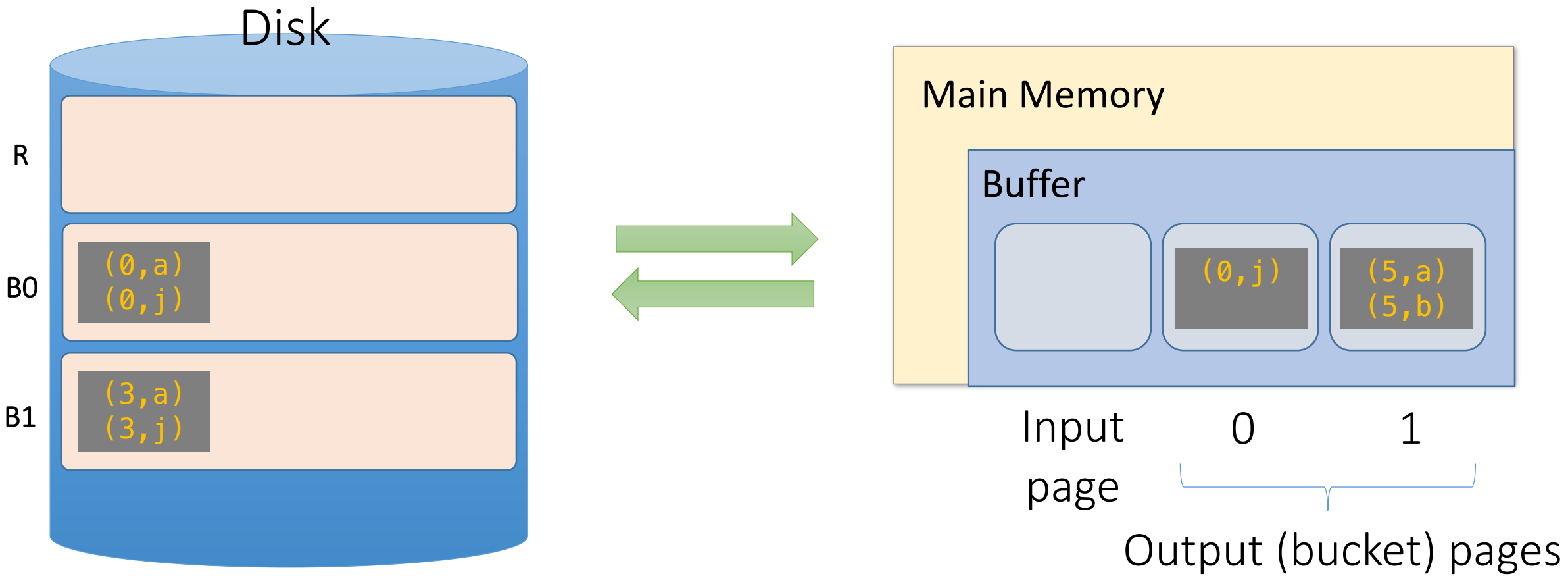
Finish this pass…



Hash Join Phase 1: Partitioning

Given *B+1 = 3* buffer pages

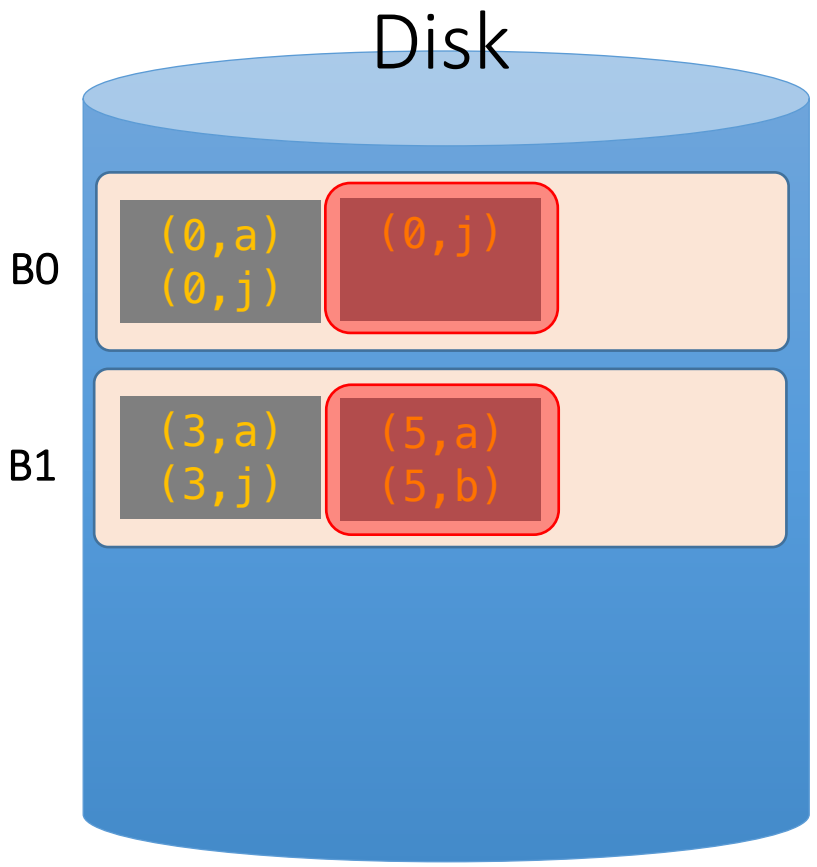
Finish this pass…



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:

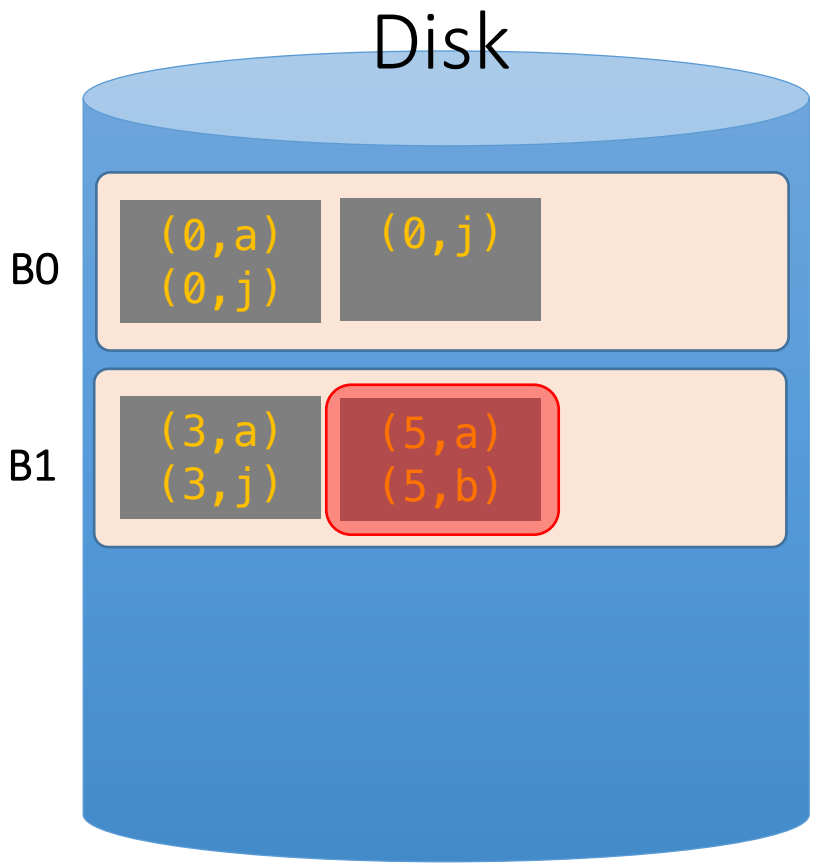


(1) Duplicate join keys

(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!

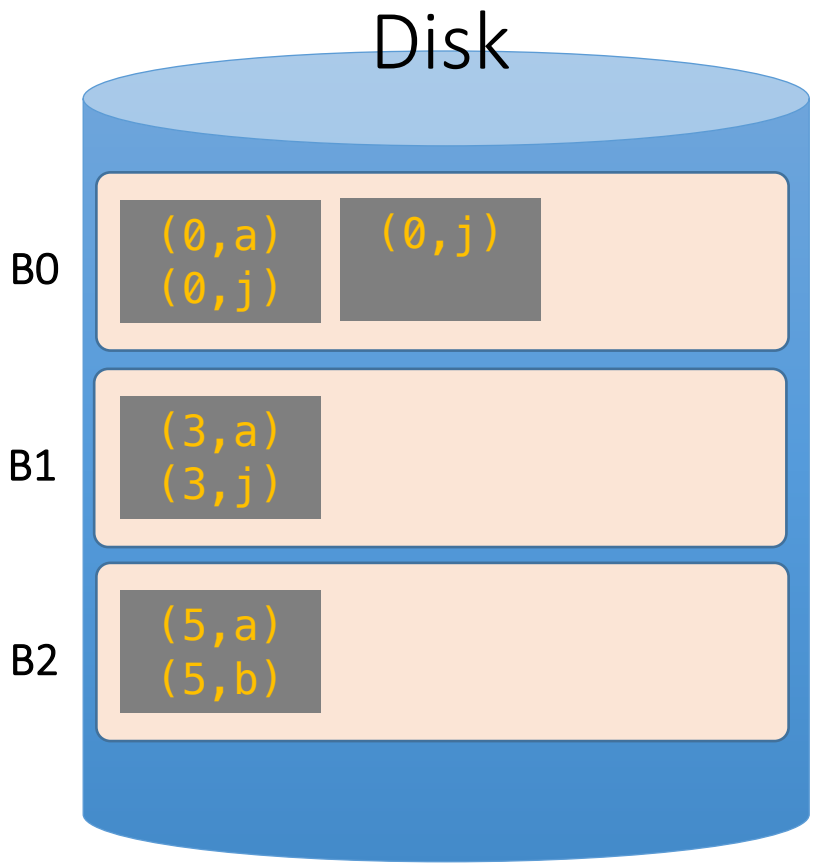
What hash function should we use?

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



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

What hash function should we use?

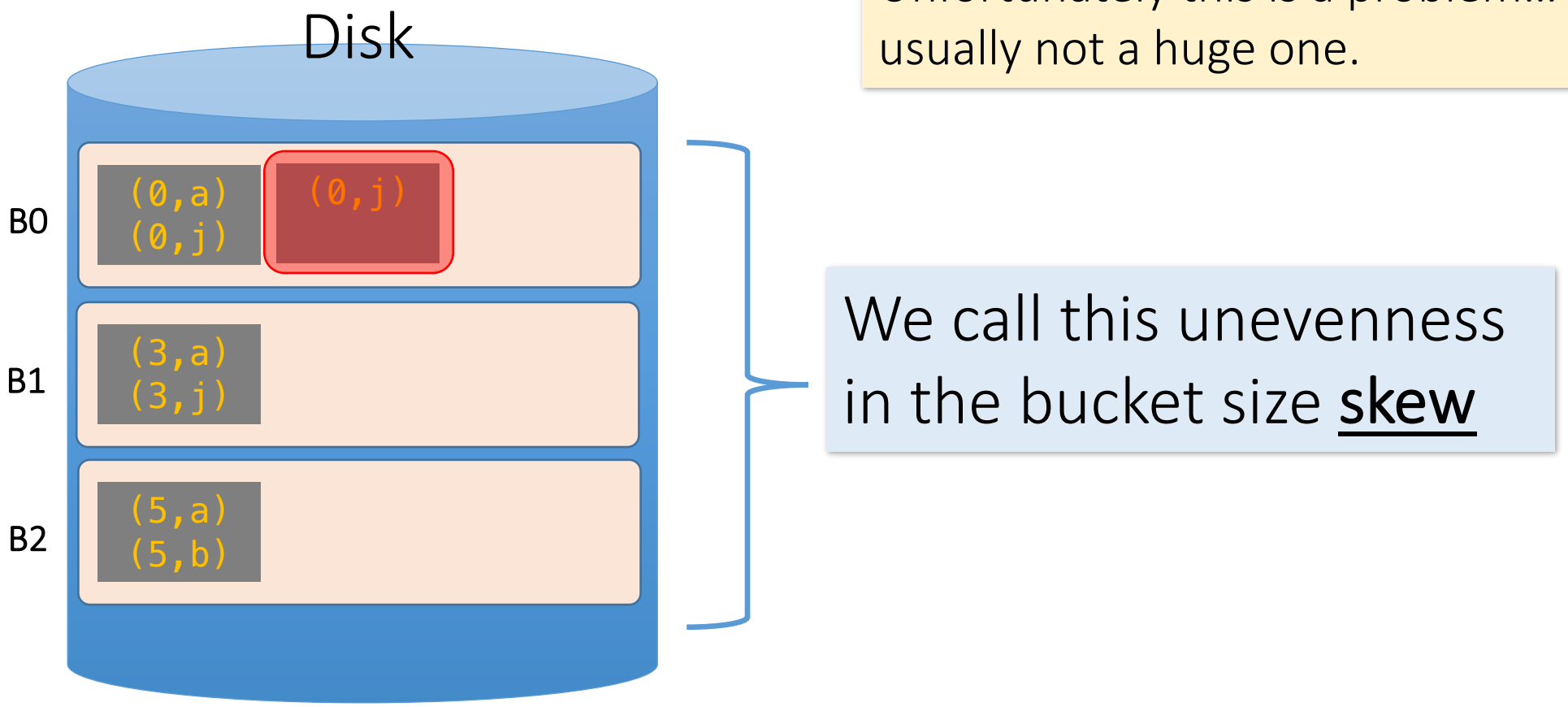
Do another pass with a different hash function, h'_2, ideally such that:

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

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.

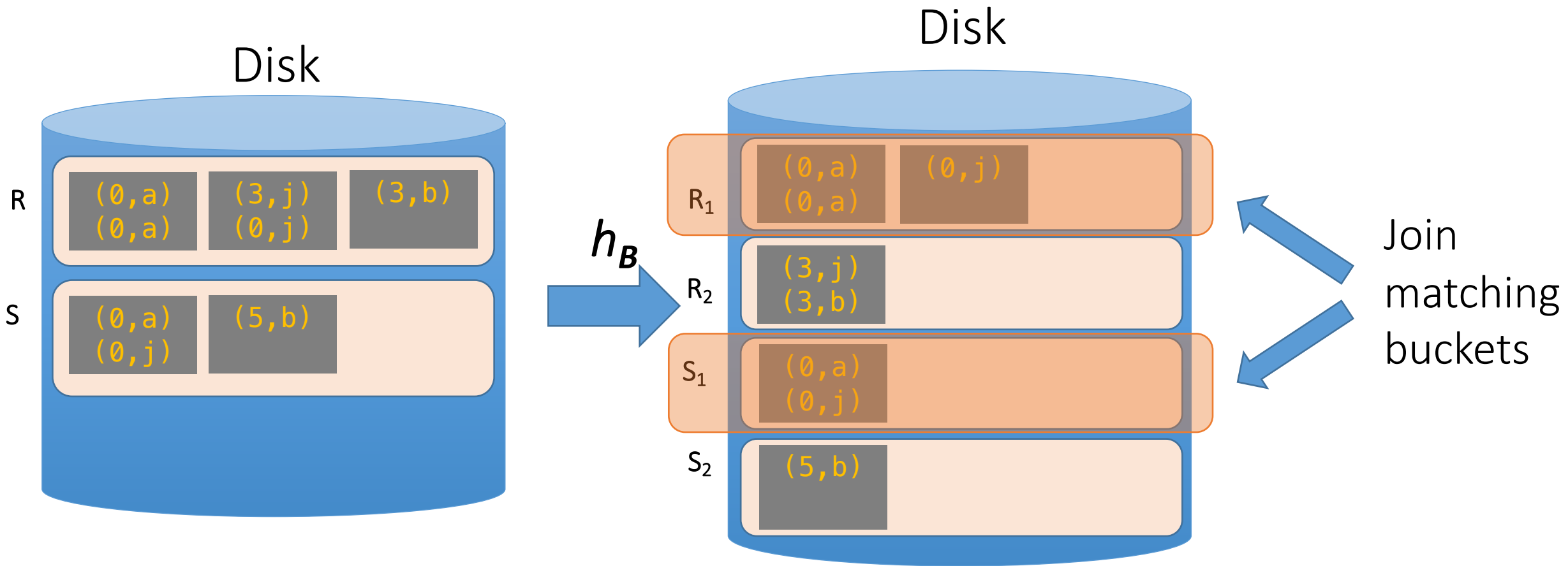


We call this unevenness in the bucket size **skew**

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

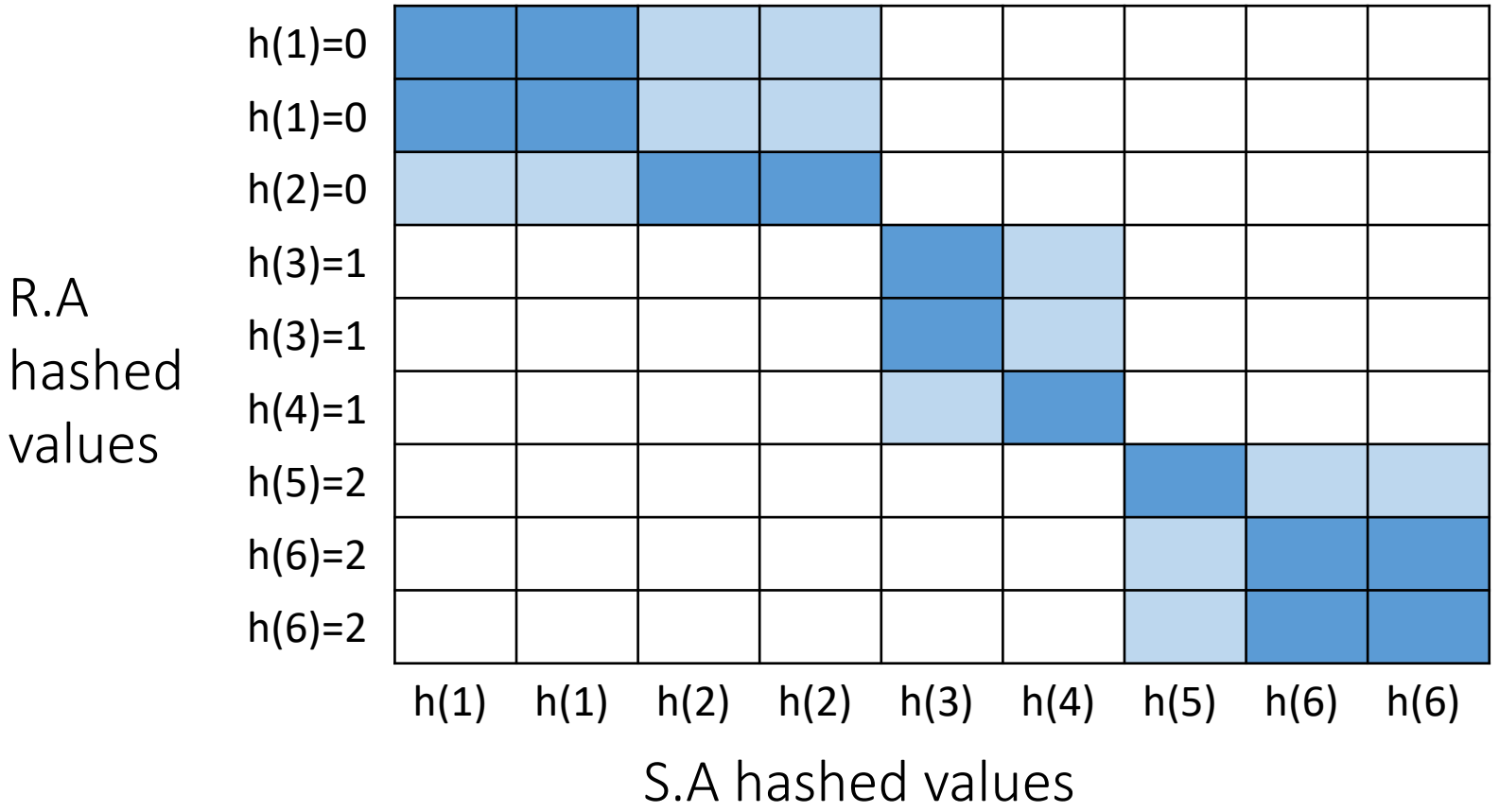
- Note that since x = y → h(x) = h(y), we only need to consider pairs of buckets (one from R, one from S) that have the same hash function value

- 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):

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

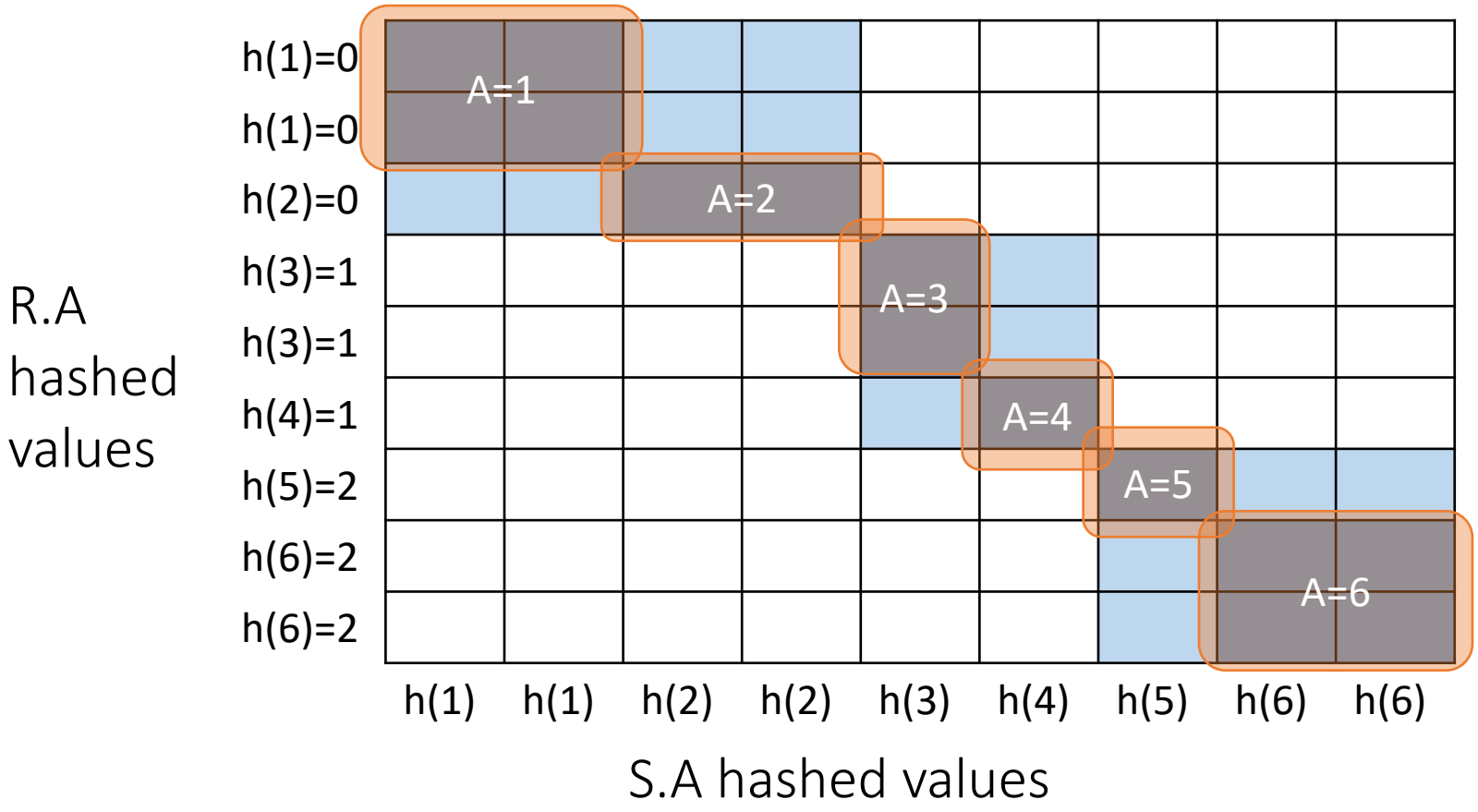
Joining the pairs of buckets is linear!
(As long as smaller bucket <= B-1 pages)

Hash Join Phase 2: Matching



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

Hash Join Phase 2: Matching

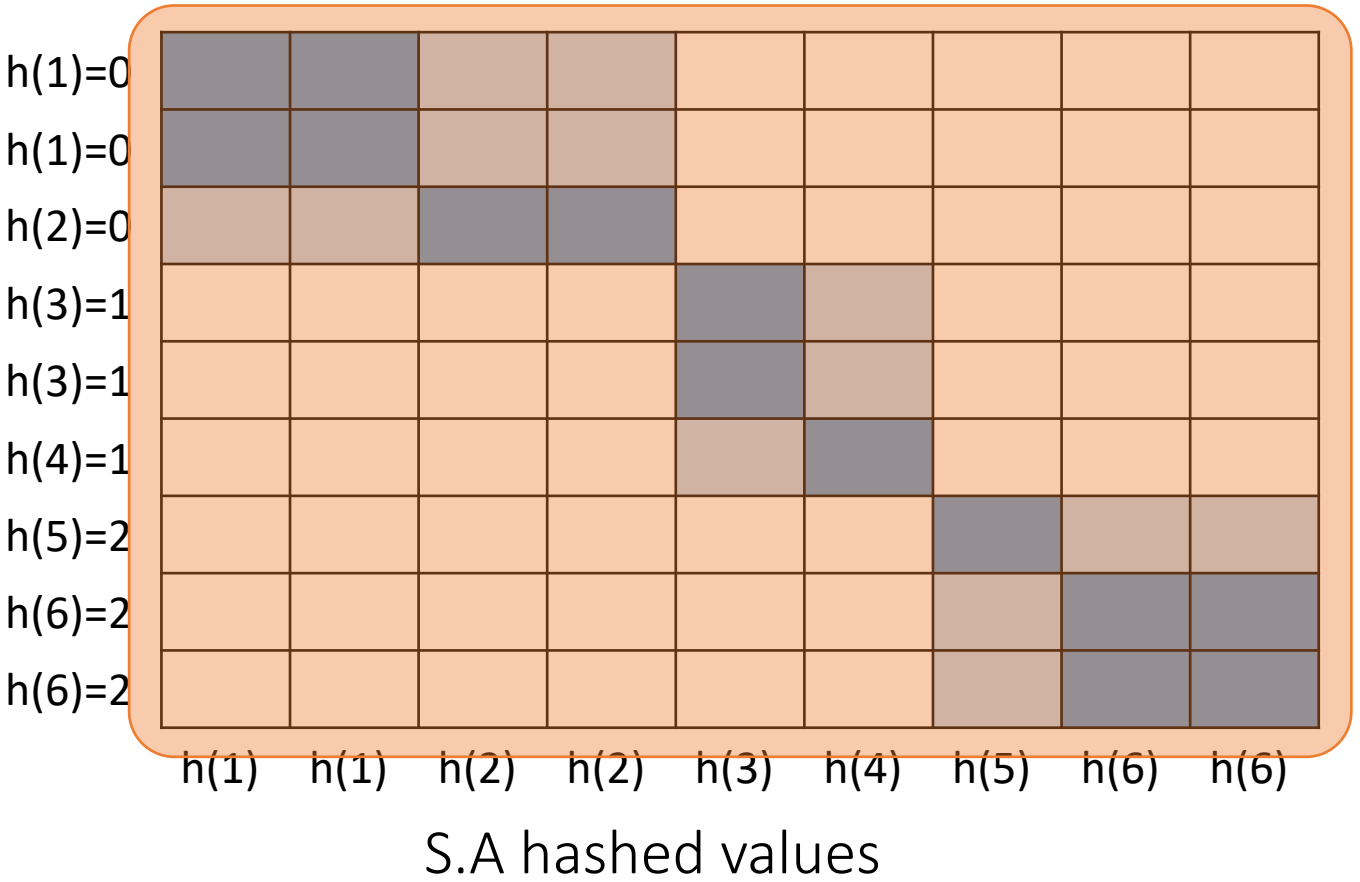


$R \bowtie S \; on \; A$

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

= the tuples with same values of the join key A

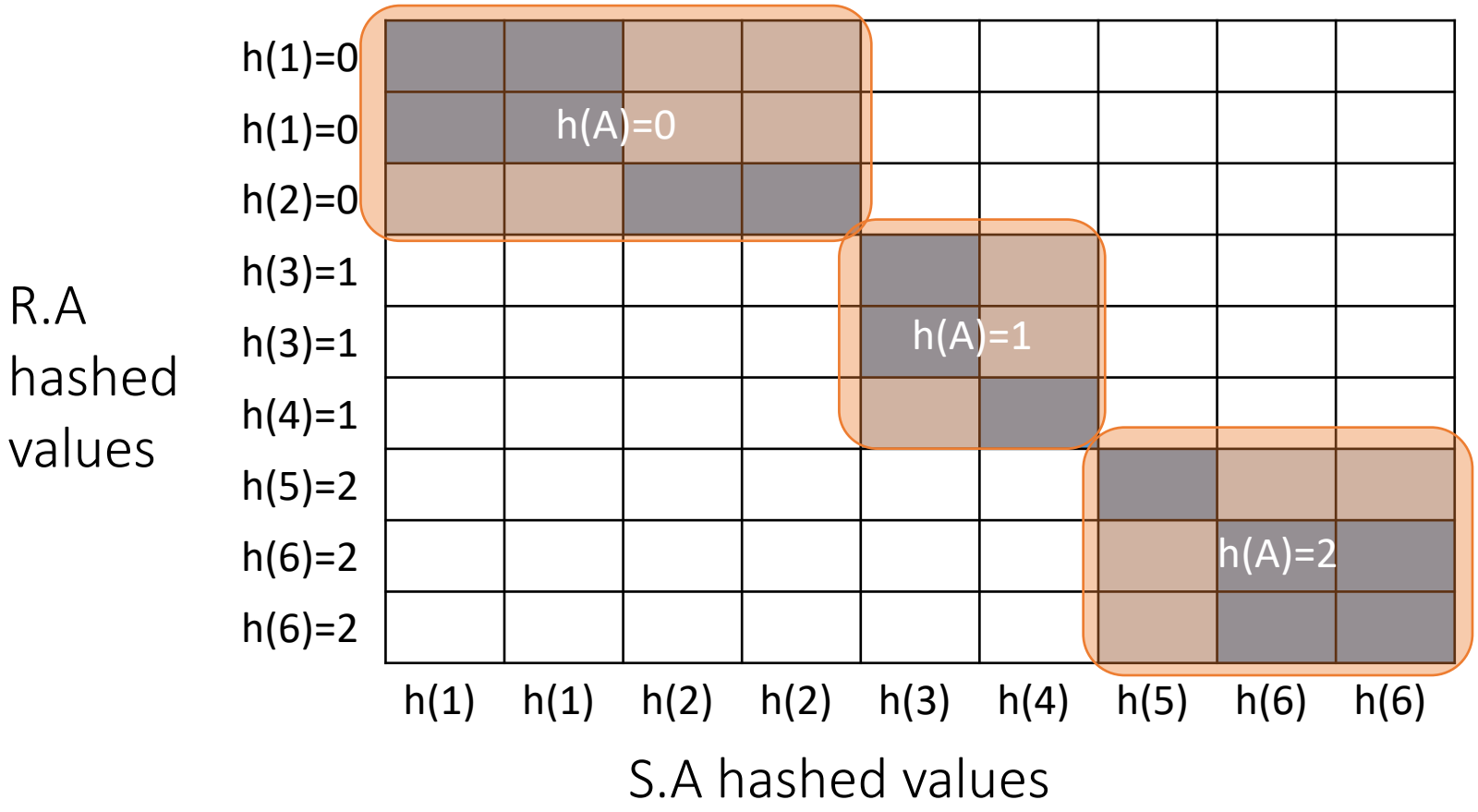
Hash Join Phase 2: Matching



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

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



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

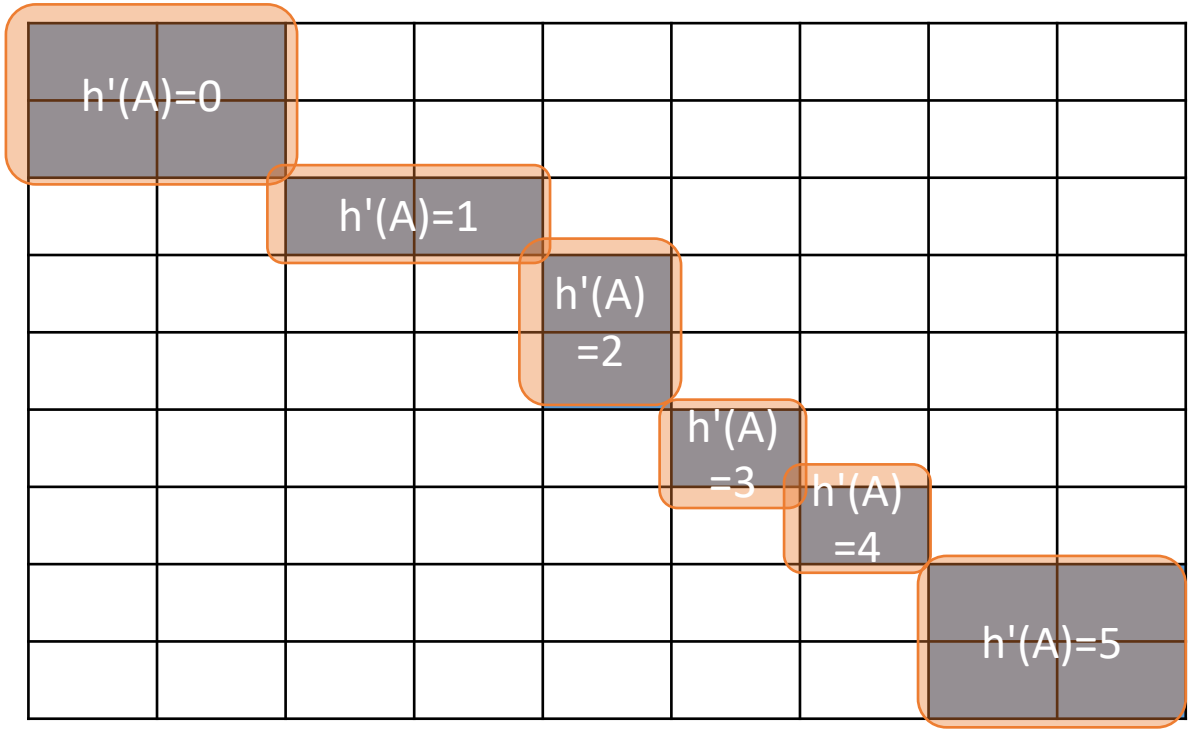
With HJ, we only explore the ***blue*** regions

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

We can apply BNLJ to each of these regions

Hash Join Phase 2: Matching

R.A hashed values



S.A hashed values

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

An alternative to applying BNLJ:

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

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!

SMJ vs. HJ

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

Summary

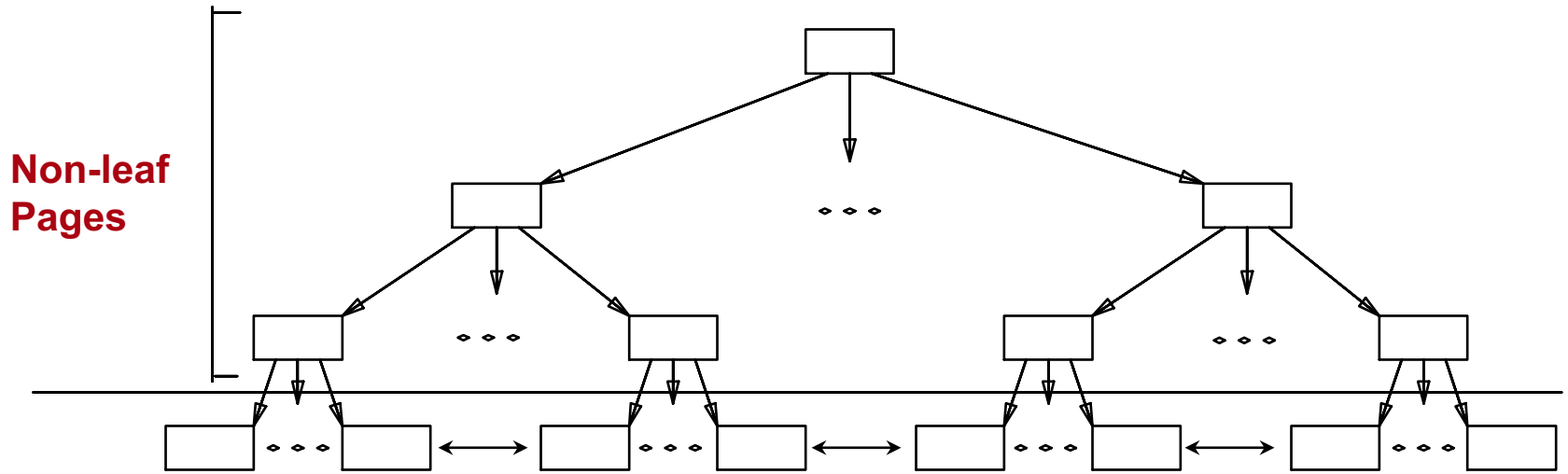
- Saw IO-aware join algorithms
  - Massive difference
- Memory sizes key in hash versus sort join
  - Hash Join = Little dog (depends on smaller relation)
- Skew is also a major factor

B+ Tree

B+ Trees

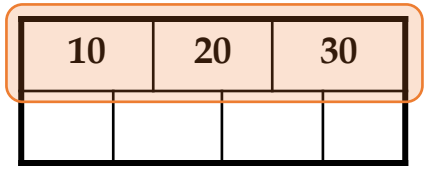
- Search trees
  - B does not mean binary!
- Idea in B Trees:
  - make 1 node = 1 physical page
  - Balanced, height adjusted tree (not the B either)
- Idea in B+ Trees:
  - Make leaves into a linked list (for range queries)

B+ Tree Index



- Leaf pages contain data entries, and are chained (prev & next)
- Non-leaf pages have data entries

B+ Tree Basics



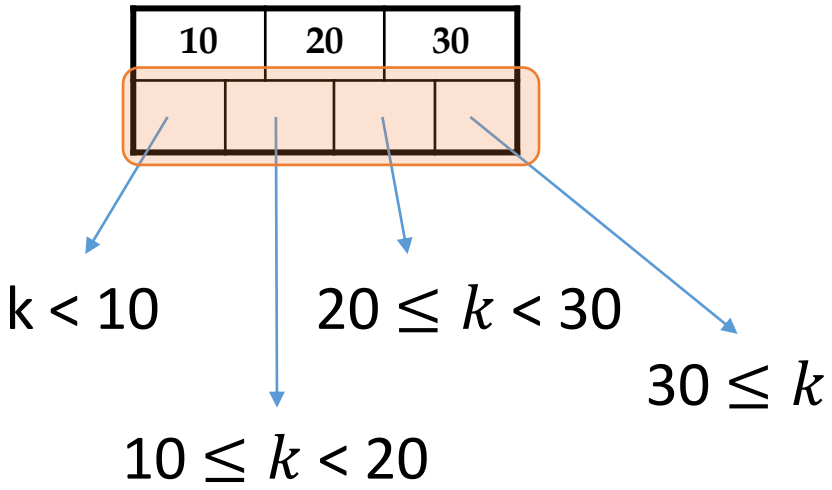
Parameter ***d*** = the order

Each *non-leaf ("interior")* ***node*** has d $\leq m \leq$ 2d ***entries***

- ***Minimum 50% occupancy***

Root ***node*** has 1 $\leq m \leq$ 2d ***entries***

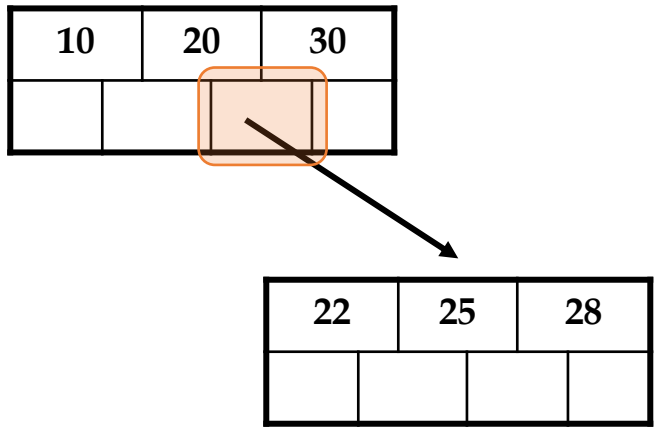
B+ Tree Basics



The *n* entries in a node define *n+1* ranges

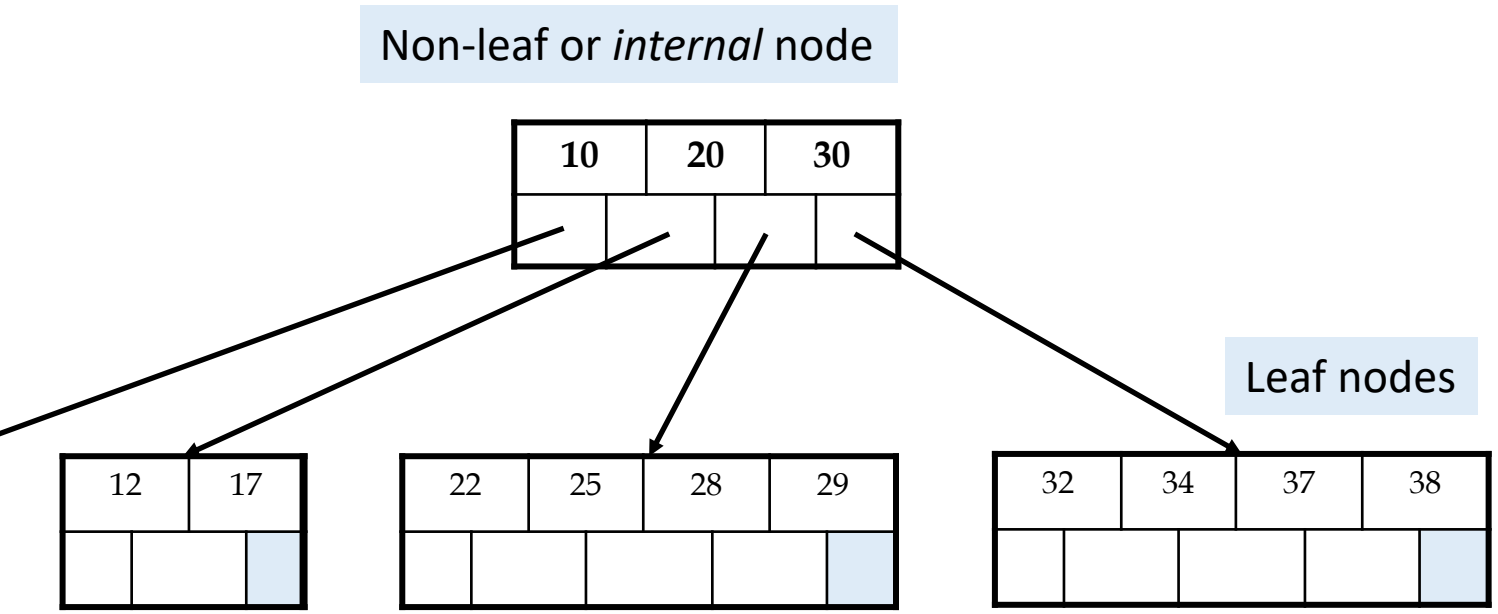
B+ Tree Basics

Non-leaf or *internal* node



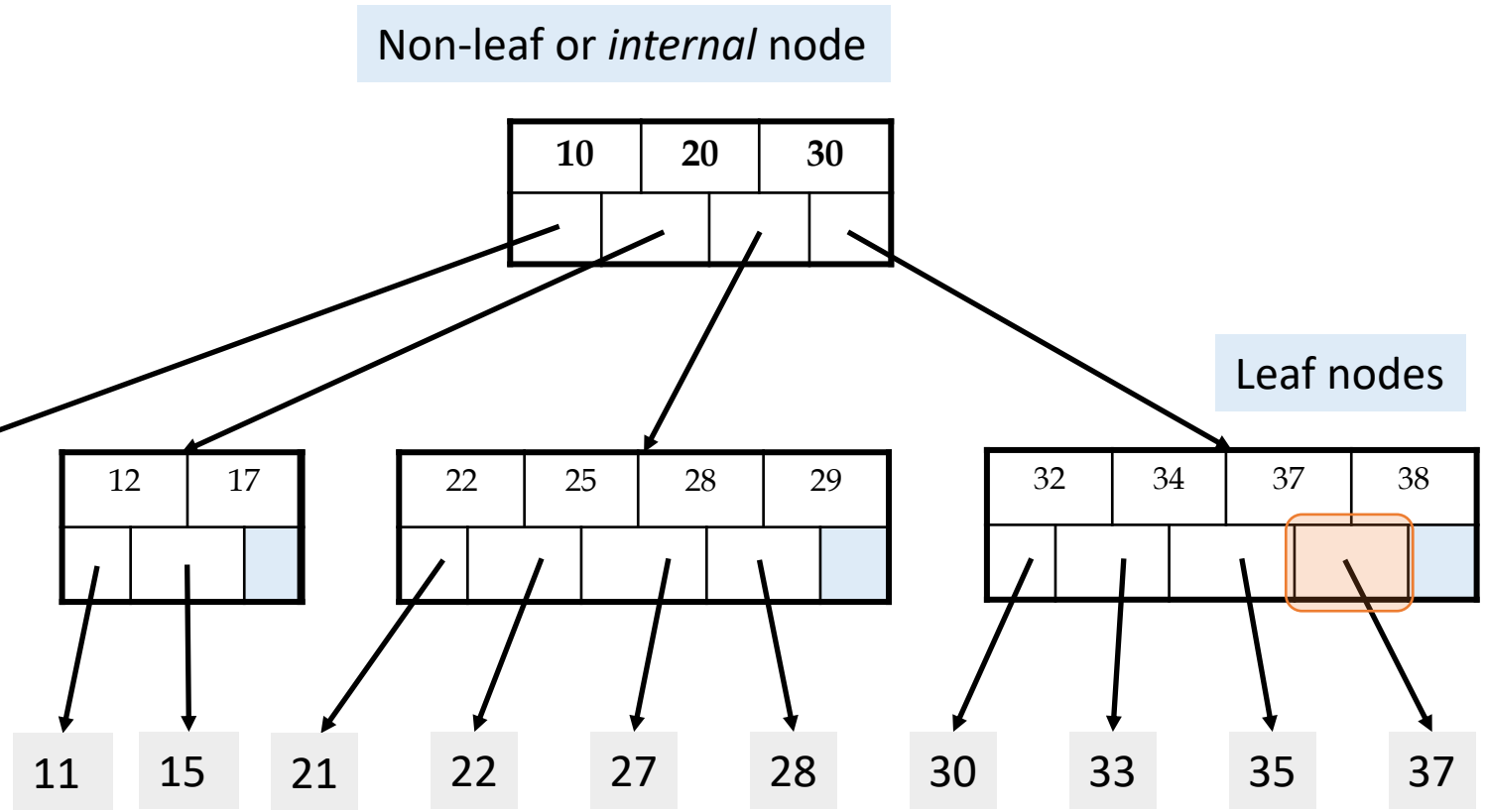
For each range, in a *non-leaf* node, there is a **pointer** to another node with entries in that range

B+ Tree Basics



Leaf nodes also have between *d* and *2d* entries, and are different in that:

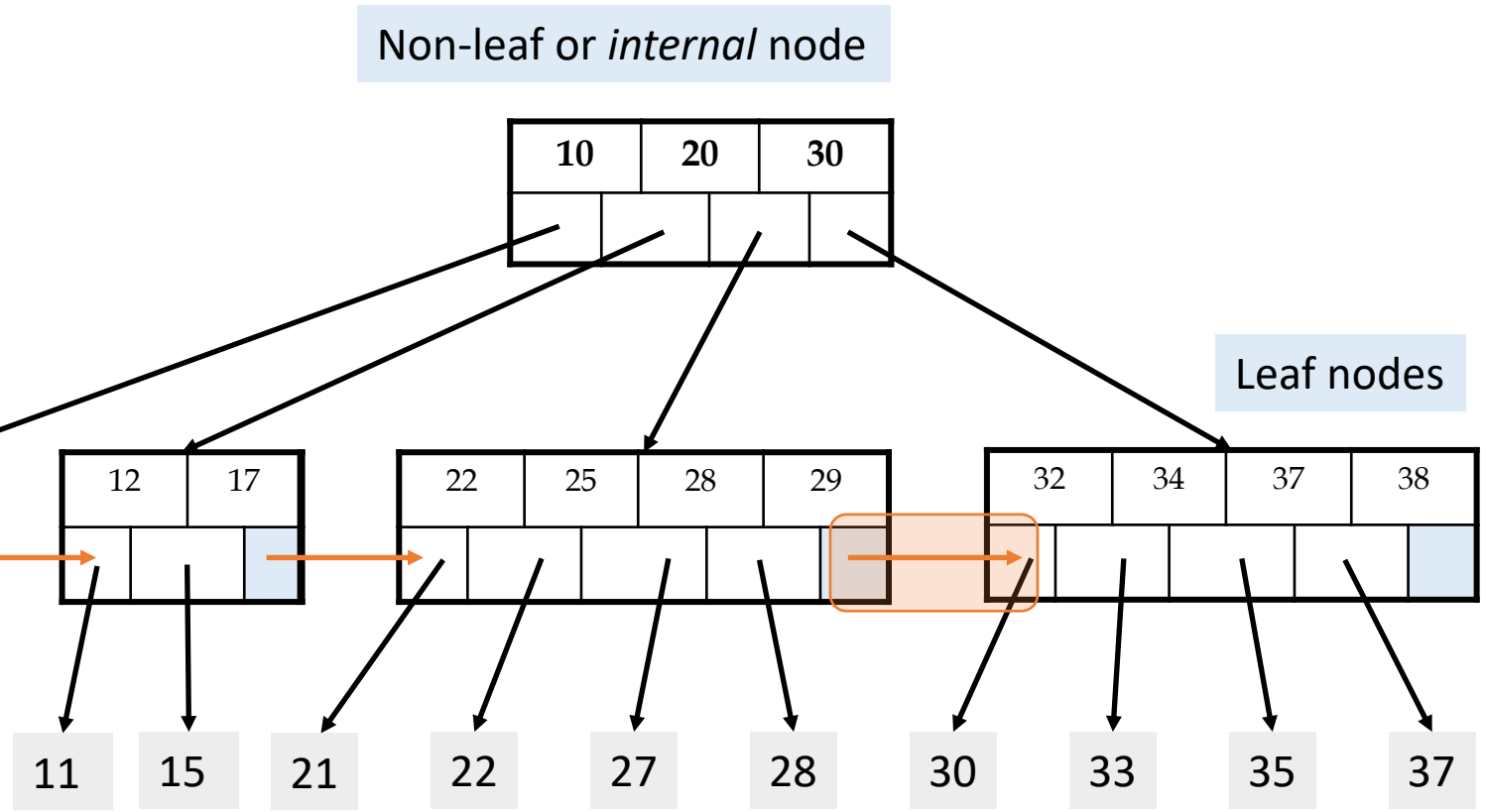
B+ Tree Basics



Leaf nodes also have between *d* and *2d* entries, and are different in that:

Their entry slots contain pointers to data records

B+ Tree Basics

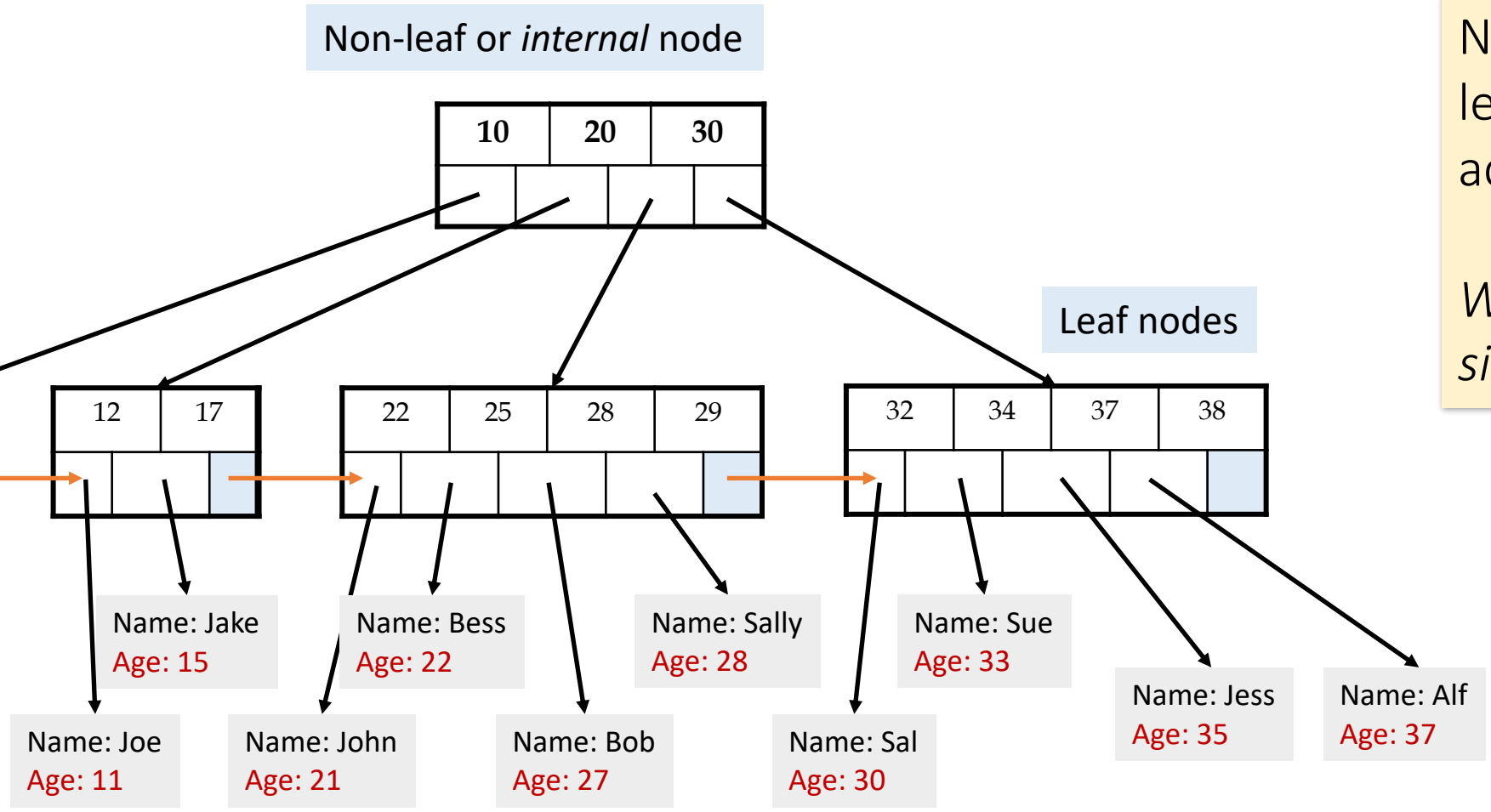


Leaf nodes also have between *d* and *2d* entries, and are different in that:

Their entry slots contain pointers to data records

They contain a pointer to the next leaf node as well, ***for faster sequential traversal***

B+ Tree Basics

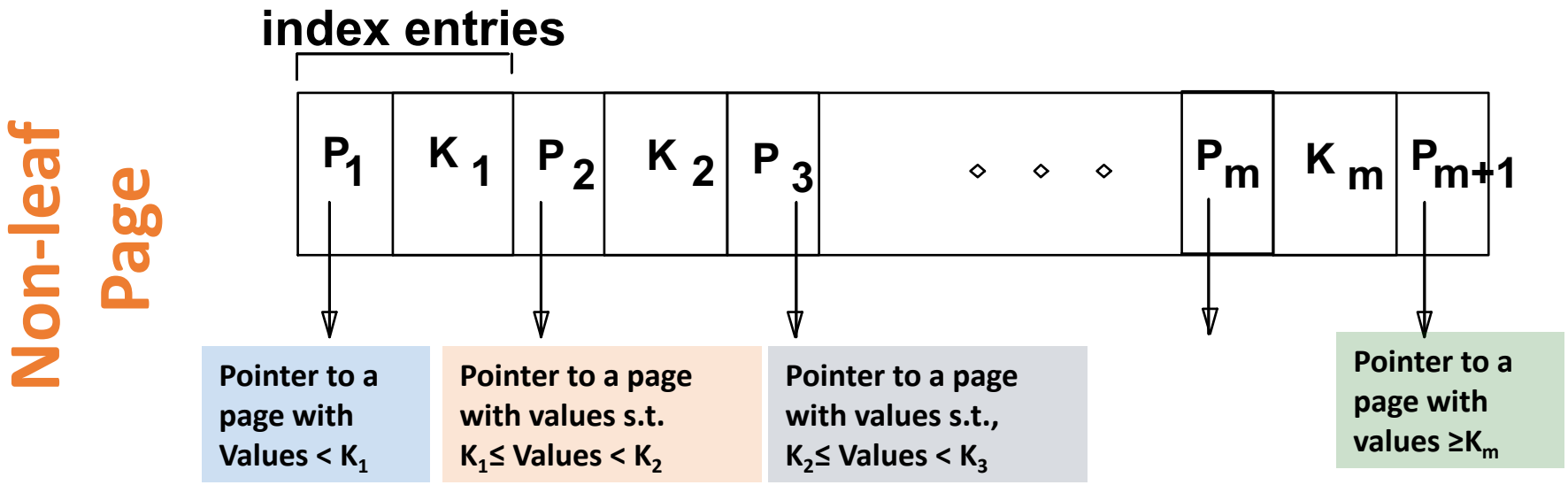


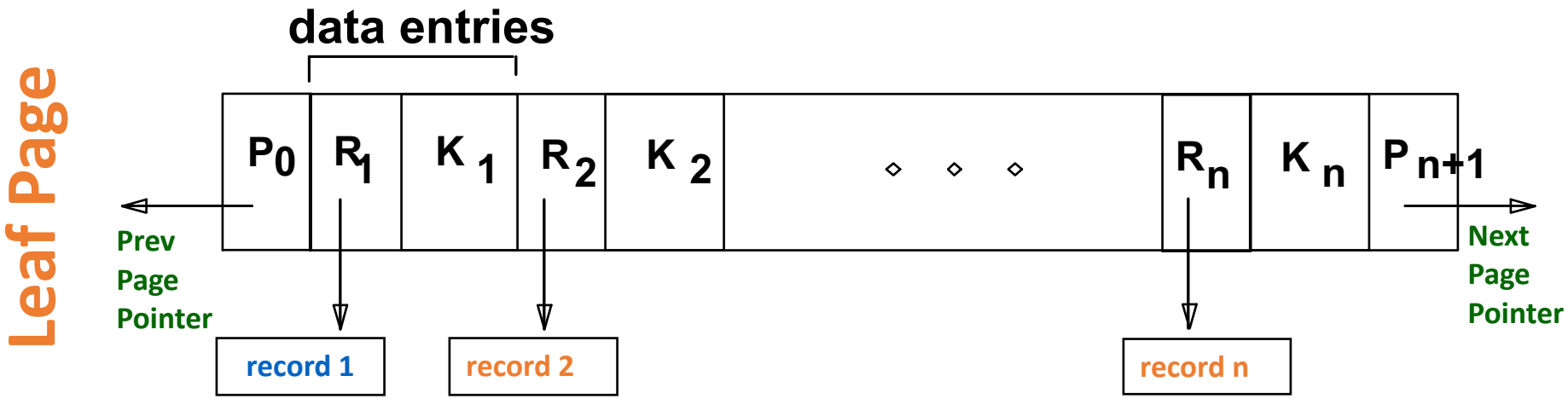
Note that the pointers at the leaf level will be to the actual data records (rows).

We might truncate these for simpler display (as before)…

Height = 1

B+ Tree Page Format





B+ Tree operations

A B+ tree supports the following operations:

- equality search
- range search
- insert
- delete
- bulk loading

Searching a B+ Tree

- For exact key values:
  - Start at the root
  - Proceed down, to the leaf

```
SELECT name
FROM   people
WHERE  age = 25
```

- For range queries:
  - As above
  - *Then sequential traversal*

```
SELECT name
FROM   people
WHERE  20 <= age
  AND  age <= 30
```

B+ Tree: Search

- start from root
- examine index entries in non-leaf nodes to find the correct child
- traverse down the tree until a leaf node is reached
- non-leaf nodes can be searched using a binary or a linear search

B+ Tree Exact Search Animation

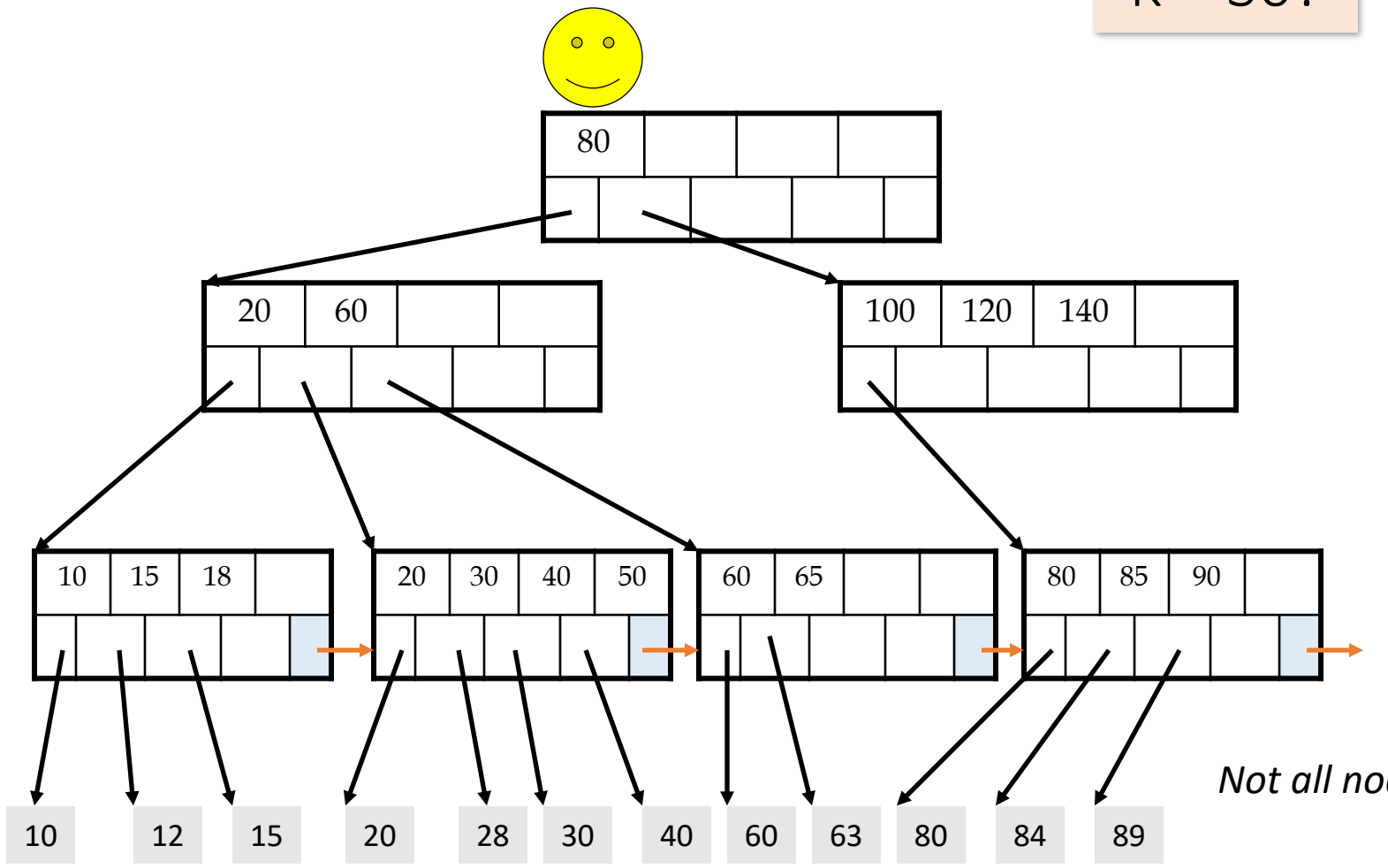
K = 30?

30 < 80

30 in [20,60)

30 in [30,40)

To the data!



Not all nodes pictured

B+ Tree Range Search Animation

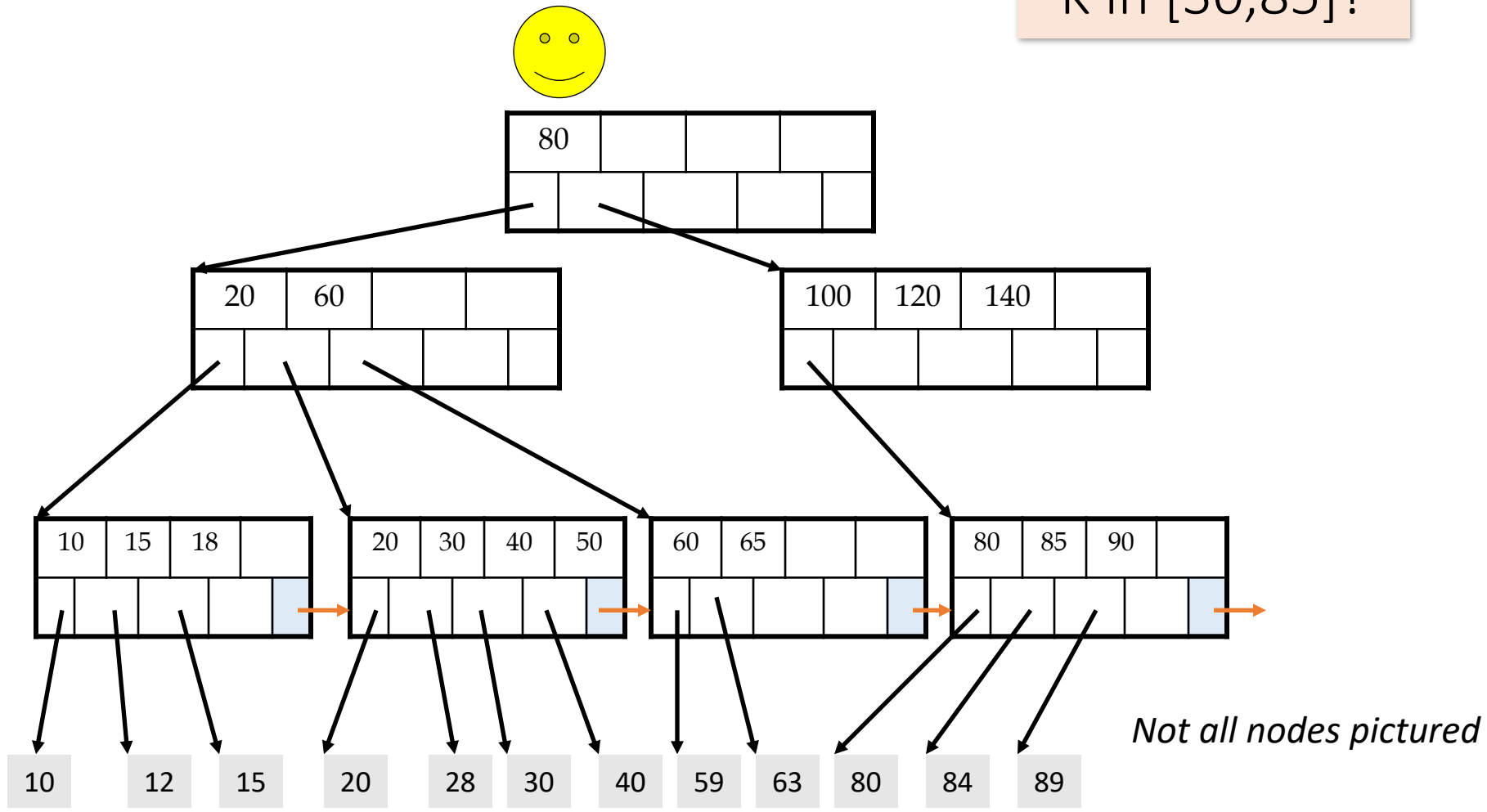
K in [30,85]?

30 < 80

30 in [20,60)

30 in [30,40)

To the data!



B+ Tree Design

- How large is ***d***?

- Example:
  - Key size = 4 bytes
  - Pointer size = 8 bytes
  - Block size = 4096 bytes

NB: Oracle allows 64K = 2^16 byte blocks
→ d <= 2730

- We want each *node* to fit on a single *block/page*
  - 2d x 4 + (2d+1) x 8 <= 4096 → ***d <= 170***

B+ Tree: High Fanout = Smaller & Lower IO

- As compared to e.g. binary search trees, B+ Trees have **high *fanout*** (***between d+1 and 2d+1***)
- This means that the **depth of the tree is small** → getting to any element requires very few IO operations!
  - Also can often store most or all of the B+ Tree in main memory!
- A TiB = 2^{40} Bytes. What is the height of a B+ Tree (with fill-factor = 1) that indexes it (with 64K pages)?
  - $(2*2730 + 1)^h = 2^{40}$ → ***h = 4***

The fanout is defined as the number of pointers to child nodes coming out of a node

Note that fanout is dynamic- we'll often assume it's constant just to come up with approximate eqns!

The known universe contains ~10^{80} particles… what is the height of a B+ Tree that indexes these?

B+ Trees in Practice

- Typical order: d=100. Typical fill-factor: 67%.
  - average fanout = 133
- Typical capacities:
  - Height 4: 133^4 = 312,900,700 records
  - Height 3: 133^3 = 2,352,637 records
- Top levels of tree sit *in the buffer pool*:
  - Level 1 = 1 page = 8 Kbytes
  - Level 2 = 133 pages = 1 Mbyte
  - Level 3 = 17,689 pages = 133 MBytes

Fill-factor is the percent of available slots in the B+ Tree that are filled; is usually < 1 to leave slack for (quicker) insertions

Typically, only pay for one IO!

Simple Cost Model for Search

- Let:
  - ***f*** = fanout, which is **in [d+1, 2d+1]** ***(we'll assume it's constant for our cost model…)***
  - ***N*** = the total number of *pages* we need to index
  - ***F*** = fill-factor (usually ~= 2/3)

- Our B+ Tree needs to have room to index ***N / F*** pages!
  - We have the fill factor in order to leave some open slots for faster insertions

- What height (*h*) does our B+ Tree need to be?
  - h=1 → Just the root node- room to index f pages
  - h=2 → f leaf nodes- room to index f^2 pages
  - h=3 → f^2 leaf nodes- room to index f^3 pages
  - …
  - h → f^{h-1} leaf nodes- room to index f^h pages!

→ We need a B+ Tree of height h = $\left\lceil \log_f \frac{N}{F} \right\rceil$!

Simple Cost Model for Search

- Note that if we have ***B*** available buffer pages, by the same logic:
  - We can store $\boldsymbol{L_B}$ levels of the B+ Tree in memory
  - where $\boldsymbol{L_B}$ ***is the number of levels such that the sum of all the levels' nodes fit in the buffer:***
    - $B \geq 1 + f + \cdots + f^{L_B - 1} = \sum_{l=0}^{L_B - 1} f^l$

- In summary: to do exact search:
  - We read in one page per level of the tree
  - However, levels that we can fit in buffer are free!
  - Finally we read in the actual record

IO Cost: $\left\lceil \log_f \frac{N}{F} \right\rceil - L_B + 1$

where $B \geq \sum_{l=0}^{L_B - 1} f^l$

Simple Cost Model for Search

- To do range search, we just follow the horizontal pointers

- The IO cost is that of loading additional leaf nodes we need to access + the IO cost of loading each ***page*** of the results- we phrase this as "Cost(OUT)"

IO Cost: $\left\lceil \log_f \frac{N}{F} \right\rceil - L_B + Cost(OUT)$

where $B \geq \sum_{l=0}^{L_B - 1} f^l$

Clustered vs. Unclustered Index

An index is ***clustered*** if the underlying data is ordered in the same way as the index's data entries.

Clustered vs. Unclustered Index

- Recall that for a disk with block access, **sequential IO is much faster than random IO**

- For exact search, no difference between clustered / unclustered

- For range search over R values: difference between **1 random IO + R sequential IO**, and **R random IO**:
  - A random IO costs ~ 10ms (sequential much much faster)
  - For R = 100,000 records- **difference between ~10ms and ~17min!**

Hash Indexes

Hash Index

- A hash index is a collection of buckets
  - bucket = primary page plus overflow pages
  - buckets contain one or more data entries

- uses a hash function **h**
  - ***h(r)*** = bucket in which (data entry for) record *r* belongs

Hash Index

- A hash index is:
  - good for equality search
  - not so good for range search (use tree indexes instead)

- Types of hash indexes:
  - **Static** hashing
  - **Extendible** hashing (dynamic)
  - Linear hashing (dynamic) – not covered in the course

Operations on Hash Indexes

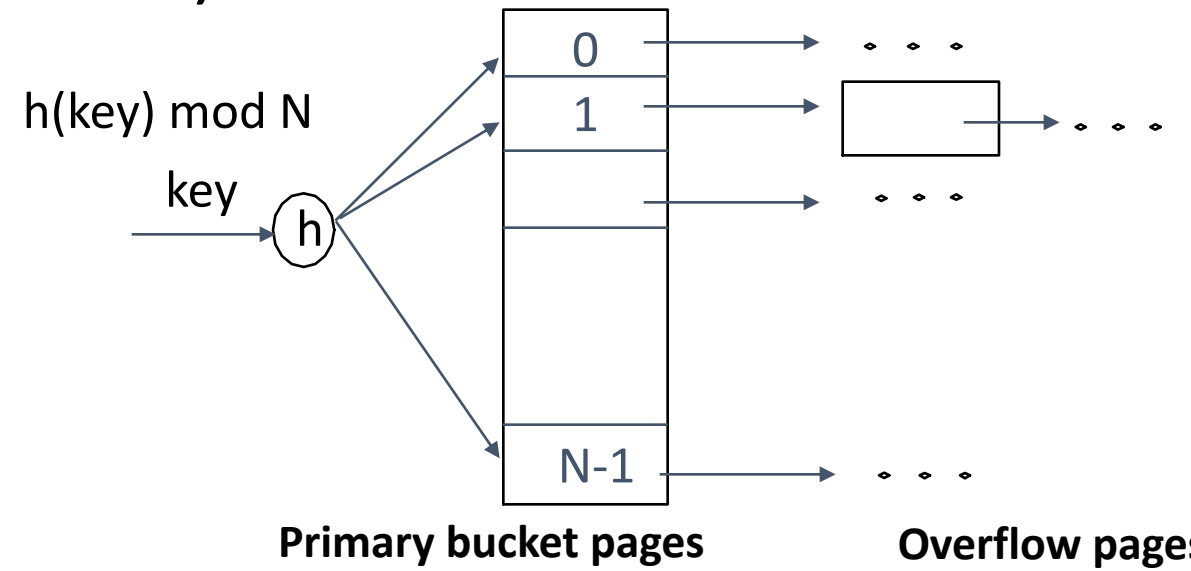
- **Equality search**
  - apply the hash function on the search key to locate the appropriate bucket
  - search through the primary page (plus overflow pages) to find the record(s)
- **Deletion**
  - find the appropriate bucket, delete the record
- **Insertion**
  - find the appropriate bucket, insert the record
  - if there is no space, create a new overflow page

Hash Functions

- An *ideal* hash function must be uniform: each bucket is assigned the same number of key values
- A *bad* hash function maps all search key values to the same bucket
- Examples of good hash functions:
  - ***h****(k) = a * k + b,* where *a* and *b* are constants
  - a random function

Static Hashing

- # primary bucket pages fixed, allocated sequentially, never de-allocated; overflow pages if needed.
- **h(*k*) mod N** = bucket to which data entry with key *k* belongs. (N = # of buckets)

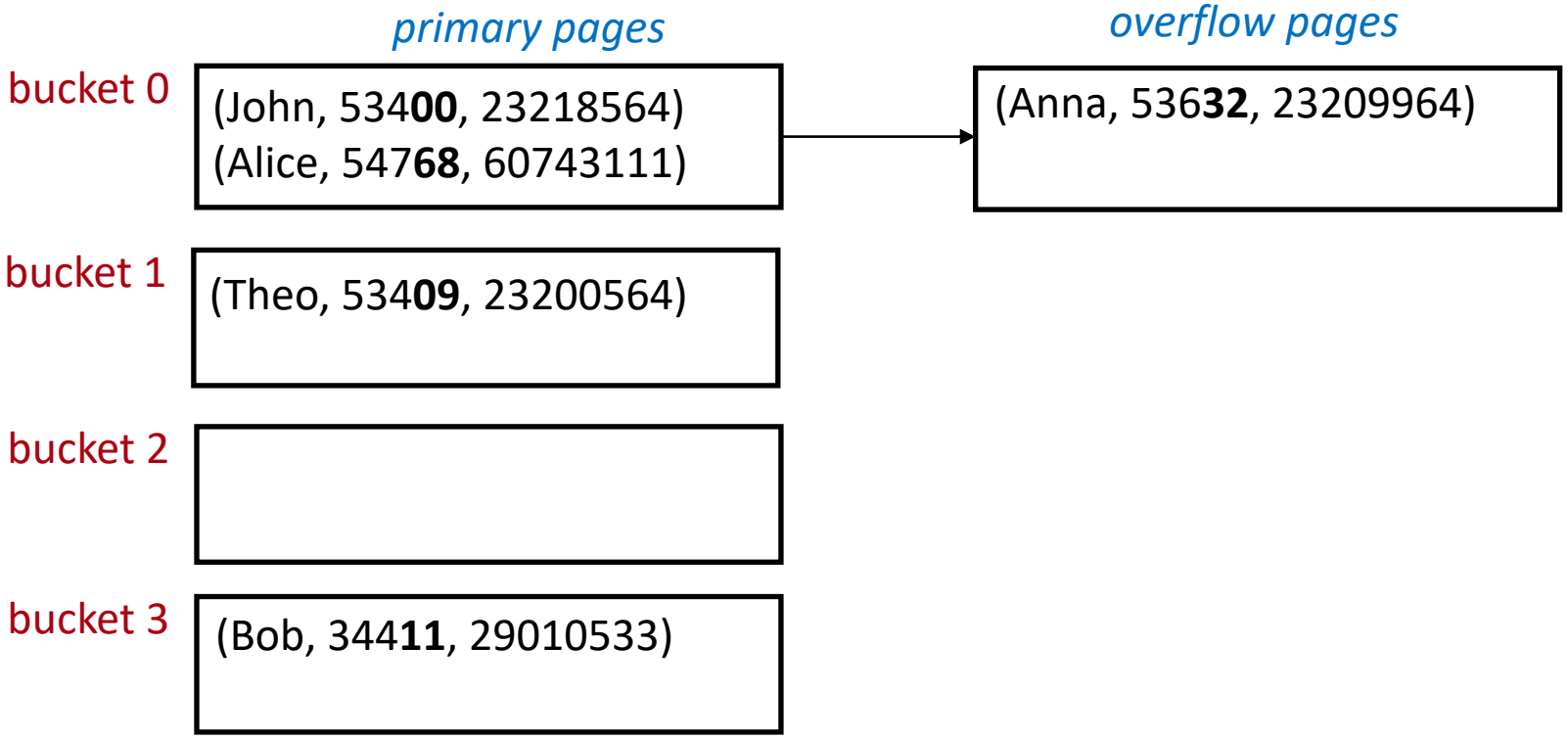


Static Hashing: Example

`Person``(name,zipcode,phone)`

- *search key*: zipcode
- *hash function* ***h***: last 2 digits

- 4 buckets
- each bucket has 2 data entries (full record)



Hash Functions

- An *ideal* hash function must be uniform: each bucket is assigned the same number of key values
- A *bad* hash function maps all search key values to the same bucket
- Examples of good hash functions:
  - ***h****(k) = a * k + b,* where *a* and *b* are constants
  - a random function

Bucket Overflow

- Bucket *overflow* can occur because of
  - insufficient number of buckets
  - *skew* in distribution of records
    - many records have the same search-key value
    - the hash function results in a non-uniform distribution of key values
- Bucket overflow is handled using *overflow buckets*

Problems of Static Hashing

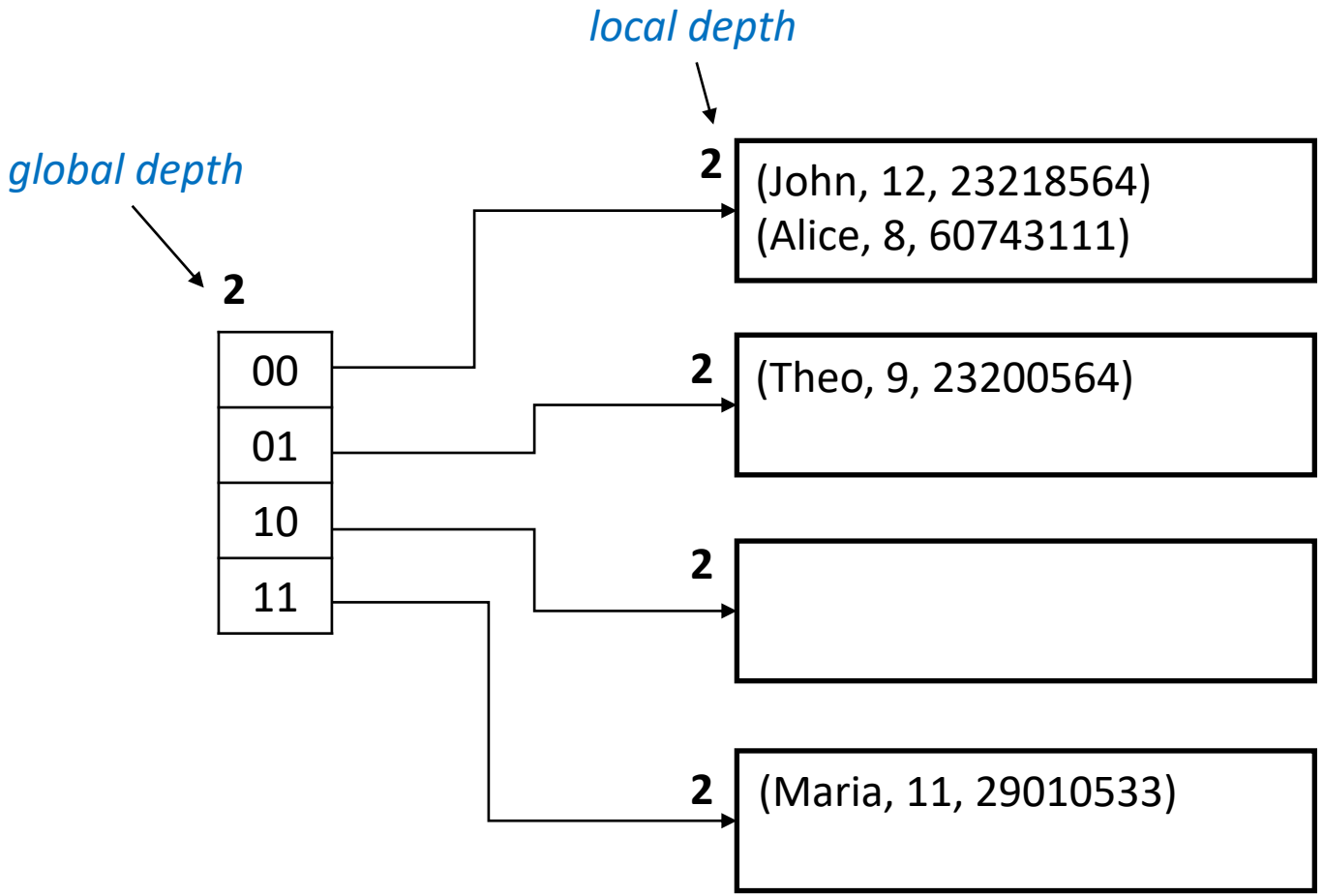
- In static hashing, there is a **fixed** number of buckets in the index
- Issues with this:
  - if the database grows, the number of buckets will be too small: long overflow chains degrade performance
  - if the database shrinks, space is wasted
  - reorganizing the index is expensive and can block query execution

Extendible Hashing

- **Extendible hashing** is a type of *dynamic* hashing
- It keeps a directory of pointers to buckets
- On overflow, it reorganizes the index by doubling the directory (and not the number of buckets)

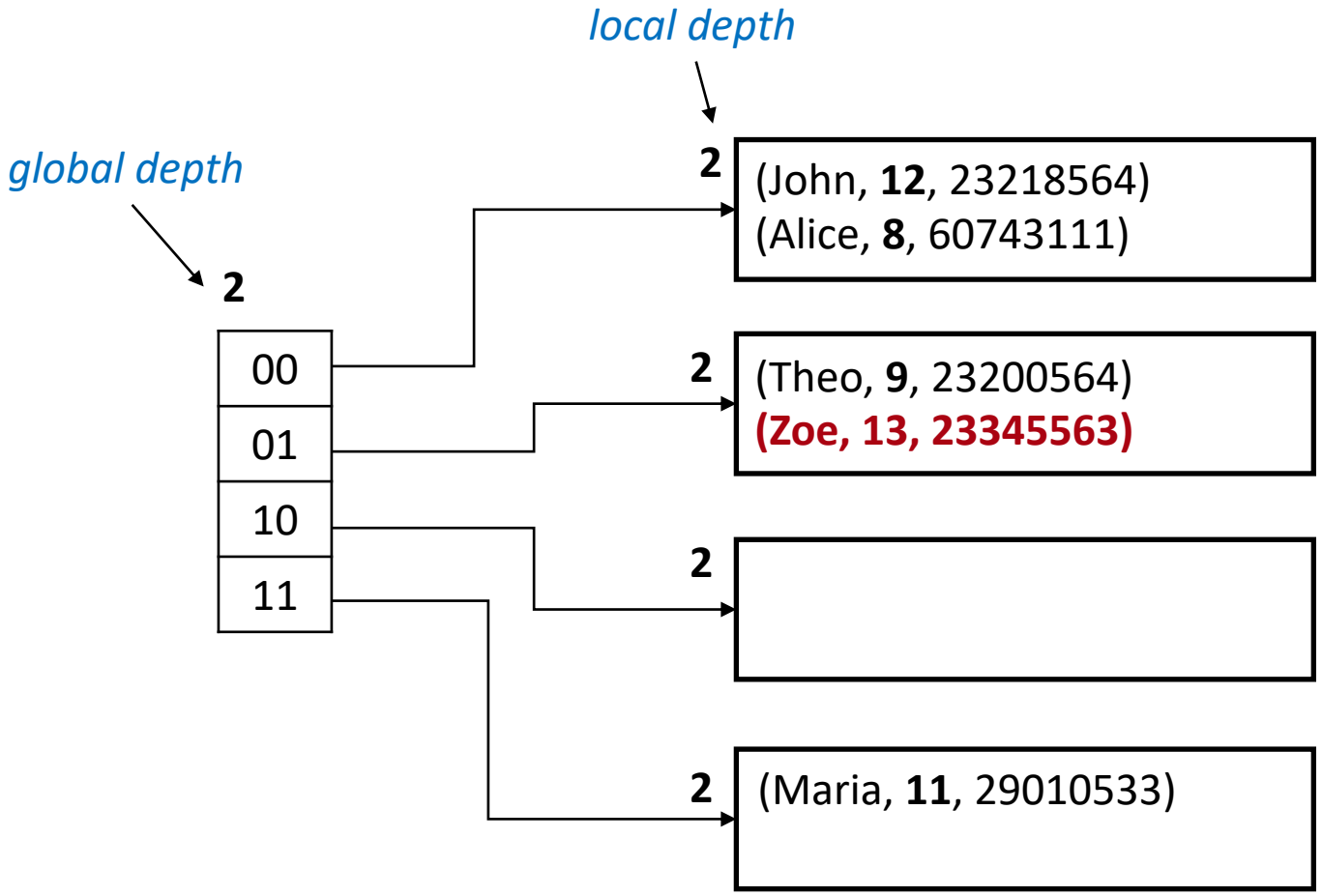
Extendible Hashing

To search, use the last **2** digits of the **binary** form of the search key value



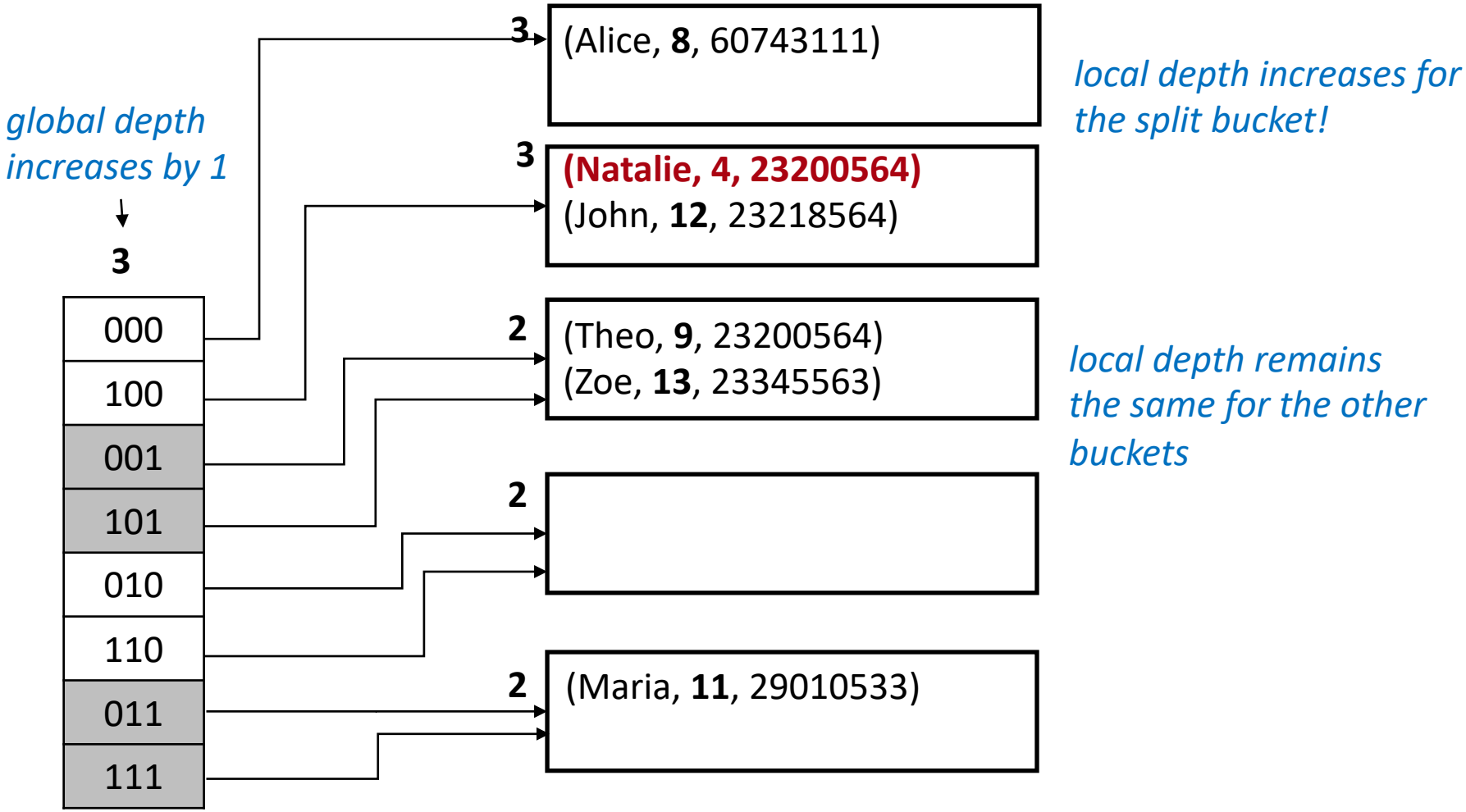
Extendible Hashing: Insert

If there is space in the bucket, simply add the record



Extendible Hashing: Insert

If the bucket is full, split the bucket and redistribute the entries



Extendible Hashing: Delete

- Locate the bucket of the record and remove it
- If the bucket becomes empty, it can be removed (and update the directory)
- Two buckets can also be coalesced together if the sum of the entries fit in a single bucket
- Decreasing the size of the directory can also be done, but it is expensive

More on Extendible Hashing

- How many disk accesses for equality search?
  - One if directory fits in memory, else two
- Directory grows in spurts, and, if the distribution of hash values is skewed, the directory can grow very large
- We may need overflow pages when multiple entries have the same hash

Bitmap indexes

Motivation

Consider the following table:

```
CREATE TABLE Tweets (
   uniqueMsgID INTEGER,        -- unique message id
   tstamp      TIMESTAMP,      -- when was the tweet posted
   uid         INTEGER,        -- unique id of the user
   msg         VARCHAR (140),  -- the actual message
   zip         INTEGER,        -- zipcode when posted
   retweet     BOOLEAN         -- retweeted?
   );
```

Consider the following query, Q1:

```
SELECT * FROM Tweets
WHERE uid = 145;
```

And, the following query, Q2:

```
SELECT * FROM Tweets
WHERE zip BETWEEN 53000 AND 54999
```

Speed-up queries using a B+-tree for the uid and the zip values.

Motivation

Consider the following table:

```
CREATE TABLE Tweets (
    uniqueMsgID INTEGER,        -- unique message id
    tstamp      TIMESTAMP,      -- when was the tweet posted
    uid         INTEGER,        -- unique id of the user
    msg         VARCHAR (140),  -- the actual message
    zip         INTEGER,        -- zipcode when posted
    retweet     BOOLEAN         -- retweeted?
    );
```

In a B+-tree, how many bytes do we use for each record?

At least key + rid,
so key-size+rid-size

Can we do better, i.e. an index with lower storage overhead? Especially for attributes with small domain cardinalities?

Bit-based indices: Two flavors

a) Bitmap indices and

b) Bitslice indices

Bitmap Indices

- Consider building an index to answer equality queries on the retweet attribute
- Issues with building a B-tree:
  - Three distinct values: True, False, NULL
  - Lots of duplicates for each distinct value
  - Sort of an odd B-tree with three long rid lists
- Bitmap Index: Build three bitmap arrays (stored on disk), one for each value.
  - The i^{th} bit in each bitmap correspond to the i^{th} tuple

    (need to map i^{th} position to a rid)

Bitmap Example

Table (stored in a heapfile)

uniqueMsgID	…	zip	retweet
1		11324	Y
2		53705	Y
3		53706	N
4		53705	NULL
5		90210	N
…	…	…	…
1,0000,000,000		53705	Y

Bitmap index on "retweet"

R-Yes	R-No	R-Null
1	0	0
1	0	0
0	1	0
0	0	1
0	1	0
…	…	…
1	0	0

```
SELECT * FROM Tweets WHERE retweet = 'N'
```

1. Scan the R-No Bitmap file
2. For each bit set to 1, compute the tuple #
3. Fetch the tuple # (s)

Critical Issue

- Need an efficient way to compute a bit position
  - Layout the bitmap in page id order.
- Need an efficient way to map a bit position to a reco
  How?
  1. If you fix the # records per page in the heapfile
  2. And lay the pages out so that page #s are sequential an
     increasing
  3. Then can construct rid (page-id, slot#)
     - page-id = Bit-position / #records-per-page
     - slot# = Bit-position % #records-per-page

Implications of #1?

With variable length records, have to set the limit based on the size of the largest record, which may result in under-filled pages.

Other Queries

Table (stored in a heapfile)

uniqueMsgID	…	zip	retweet
1		11324	Y
2		53705	Y
3		53706	N
4		53705	NULL
5		90210	N
…	…	…	…
1,0000,000,000		53705	Y

Bitmap index on "retweet"

R-Yes	R-No	R-Null
1	0	0
1	0	0
0	1	0
0	0	1
0	1	0
…	…	…
1	0	0

```
SELECT COUNT(*) FROM Tweets WHERE retweet = 'N'
```

```
SELECT * FROM Tweets WHERE retweet IS NOT NULL
```

2. Storing a bitmap index

Storing the Bitmap index

- One bitmap for each value, and one for Nulls
- Need to store each bitmap
- Simple method: 1 file for each bitmap
- Can compress the bitmap!

Index size? #tuples * (cardinality of the domain + 1) *bits*

When is a bitmap index more space efficient than a B+-tree?

#distinct values < data entry size in the B+-tree

3. Bit-sliced Index

Bit-sliced Index: Motivation

(Re)consider the following table:

```
CREATE TABLE Tweets (
    uniqueMsgID INTEGER,        -- unique message id
    tstamp      TIMESTAMP,      -- when was the tweet posted
    uid         INTEGER,        -- unique id of the user
    msg         VARCHAR (140),  -- the actual message
    zip         INTEGER,        -- zipcode when posted
    retweet     BOOLEAN         -- retweeted?
    );
```

```
SELECT * FROM Tweets WHERE zip = 53706
```

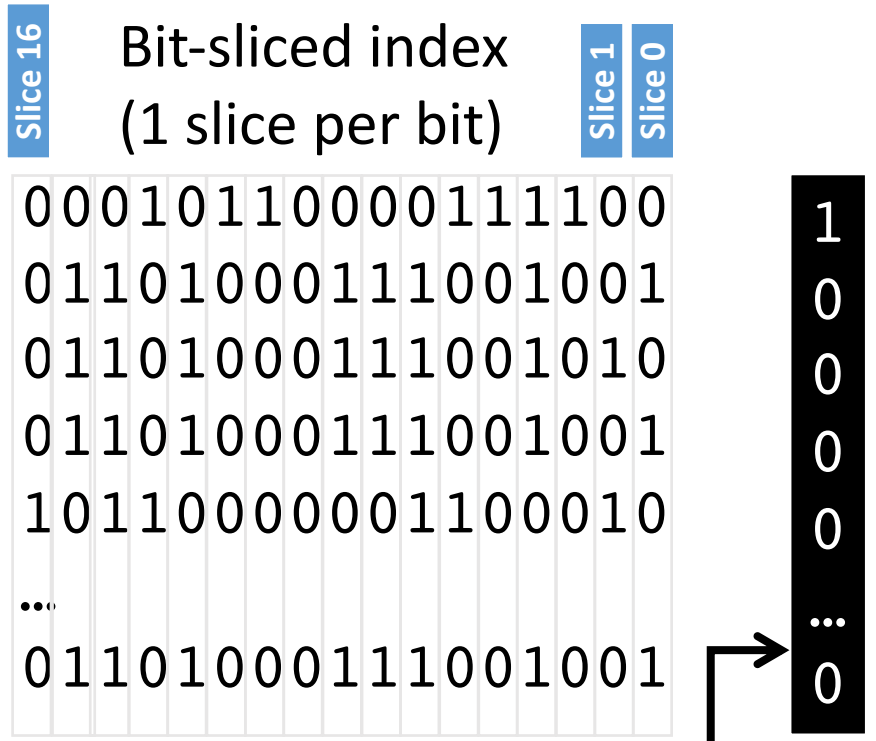
Would we build a bitmap index on zipcode?

Bit-sliced index

Why do we have 17 bits for zipcode?

Table

uniqueMsgID	…	zip	retweet
1		11324	Y
2		53705	Y
3		53706	N
4		53705	NULL
5		90210	N
…	…	…	…
1,0000,000,000		53705	Y



Query evaluation: Walk through each slice constructing a **result bitmap**

e.g. zip ≤ 11324, skip entries that have 1 in the first three slices (16, 15, 14)

(Null bitmap is not shown)

Bitslice Indices

- Can also do aggregates with Bitslice indices
  - E.g. SUM(attr): Add bit-slice by bit-slice.

    First, count the number of 1s in the **slice17,** and multiply the count by 2^{17}

    Then, count the number of 1s in the **slice16**, and multiply the count by …

- Store each slice using methods like what you have for a bitmap.
  - Note once again can use compression

Bitmap v/s Bitslice

- Bitmaps better for low cardinality domains
- Bitslice better for high cardinality domains
- Generally easier to “do the math” with bitmap indices