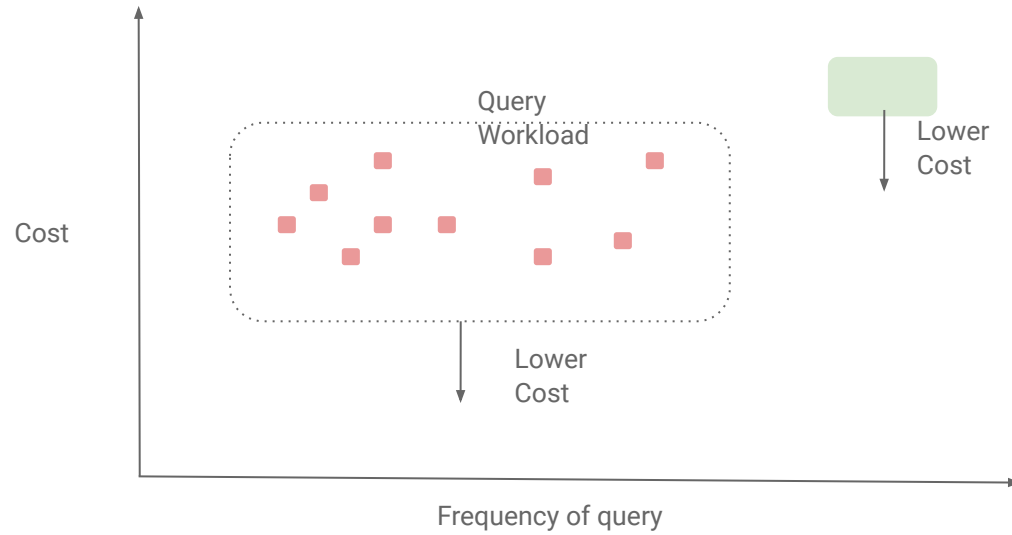




Optimization!

Optimizing

Queries and workloads



Workload = <Query, Frequency of query>

Example:

Basic SFW queries

Workload description

```
SELECT pname
FROM   Product
WHERE  year = ? AND category = ?
```

```
SELECT pname
FROM   Product
WHERE  year = ? AND Category = ?
AND    manufacturer = ?
```

Lower cost
(query and
update cost)

1. How to execute? Sort, Hash first ...?
2. Maintain indexes for Year? Category? Manufacturer?
3. For query, check multiple indexes?
4. What's cost of maintaining index?
5. Use multiple machines? ...

Intuition

Manufacturers likely most **Selective**.

Many more manufacturers than Categories. Maintain index, if this query happens a lot.

Optimization

Roadmap



Build Query Plans



Analyze Plans

1. For SFW, Joins queries
 - a. Sort? Hash? Count? Brute-force?
 - b. Pre-build an index? B+ tree, Hash?
2. What statistics can I keep to optimize?
 - a. E.g. Selectivity of columns, values

Cost in I/O, resources?
To query, maintain?



1. Nested Loop Joins

What you will
learn about in
this section

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



RECAP: Joins

Joins: Example

$R \bowtie S$

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

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

R

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

S

A	D
3	7
2	2
2	3



A	B	C	D
2	3	4	2

Joins: Example

$R \bowtie S$

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

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

R

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

S

A	D
3	7
2	2
2	3



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

Joins: Example

$R \bowtie S$

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

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

R

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

S

A	D
3	7
2	2
2	3



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

Joins: Example

$R \bowtie S$

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

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

R

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

S

A	D
3	7
2	2
2	3



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

Joins: Example

$R \bowtie S$

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

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

R

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

S

A	D
3	7
2	2
2	3



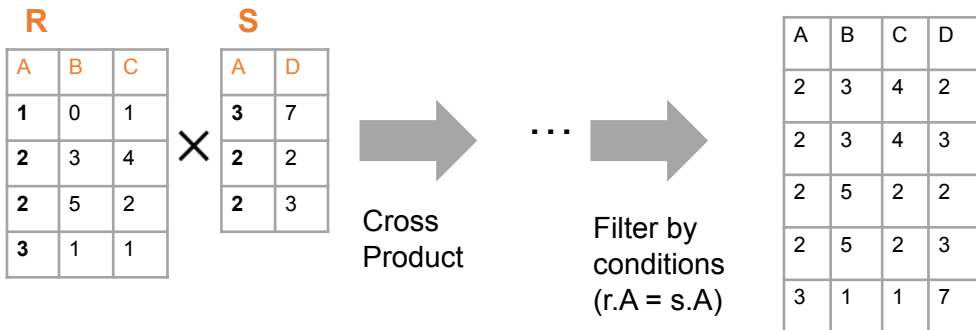
A	B	C	D
2	3	4	2
2	3	4	3
2	5	2	2
2	5	2	3
3	1	1	7

Semantically: A Subset of the Cross Product

$R \bowtie S$

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

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



Can we actually implement a join in this way?



Nested Loop Joins



Notes

We consider “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

We'll see lots of formulae from now

⇒ Hint: Focus on how it works. Much easier to derive from 1st principles (vs recalling formula soup)



Nested Loop Join (NLJ)

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


Nested Loop Join (NLJ)

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

Cost:

$P(R)$

1. Loop over the tuples in R

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

Nested Loop Join (NLJ)

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

Cost:

$$P(R) + T(R) \cdot P(S)$$

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

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

Nested Loop Join (NLJ)

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

Cost:

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

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

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

Nested Loop Join (NLJ)

Compute $R \bowtie S$ on A :

for r in R :

for s in S :

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

yield (r,s)

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

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

Cost:

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

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

Nested Loop Join (NLJ)

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

Cost:

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

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



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

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



IO-Aware Approach

Announcements

Next week

- “Putting it all together” for big systems

Upcoming talks

- Nov 9: [Girish Baliga] How Uber scales user/driver data with prestoDB
- Nov 9: [Michael Li] How Coinbase works with user data and blockchain

Ed post on talks this AM

- DataOps and ETL/ELT for a modern data stack
- Is data a 100b\$ or 1T\$ industry for the next decade?

Intuition:

Block Nested Loop Joins

Census (SSN, Address, ...)
TaxInfo (SSN, TaxPaid, ...)

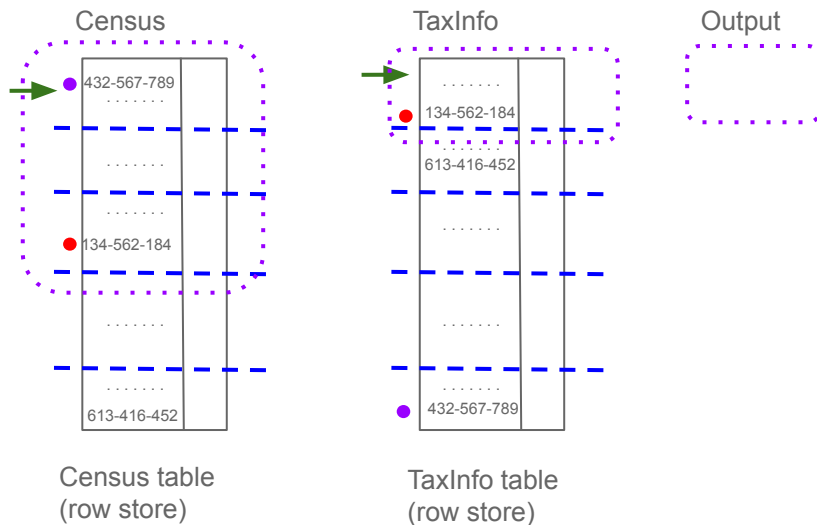
For 1 Billion people

Goal: Compute Census JOIN TaxInfo

Data stored in RowStores, 1000 tuples/page (million pages)

Given: $B + 1$ buffer space

Idea: Use $B-1$ pages for Census, 1 page each for TaxInfo and output



Steps: Repeat till done

Read $B-1$ pages from Census into Buffer

Read 1 page from TaxInfo

Partial Join into 1 output page

Block Nested Loop Join (BNLJ)

Compute $R \bowtie S$ on A :

for each $B-1$ pages pr of R :

for page ps of S :

for each tuple r in pr :

for each tuple s in ps :

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

yield (r,s)

Given $B+1$ pages of memory
(For $B \ll P(R), P(S)$)

Cost:

$P(R)$

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

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

Block Nested Loop Join (BNLJ)

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

Given $B+1$ pages of memory

Cost:

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

How many R pages to
read?
 $P(R)$

How many times is each
S page read?
 $P(R)/(B-1)$

1. Load in $B-1$ pages of R at a time (leaving 1 page each free for S & output)
2. For each $(B-1)$ -page segment of R , load each page of S

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

Block Nested Loop Join (BNLJ)

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

Cost:

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

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

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

BNLJ can also handle non-equality constraints

Block Nested Loop Join (BNLJ)

Given $B+1$ pages of memory

Cost:

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

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

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

BNLJ vs. NLJ: Benefits of IO Aware

In BNLJ, by loading larger chunks of R, we minimize the number of full *disk reads* of S

- We only read all of S from disk for every $(B-1)$ -page segment of R!
- Still the full cross-product, but more done only *in memory*

NLJ

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



BNLJ

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

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

BNLJ vs. NLJ: Benefits of IO Aware

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

*Ignoring OUT
here...*

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

Example NLJ vs. BNLJ: Steel Cage Match

Example: $P(R) = 1000$, $P(S) = 500$,
 100 tuples/page $\Rightarrow T(R) = 1000 \cdot 100$, $T(S) = 500 \cdot 100$

	B + 1 = 100 (i.e., B = 99)	B + 1 = 20 (i.e., Buffer B = 19)
NLJ	$(1000 + 1000 \cdot 100 \cdot 500 + \text{OUT})$ $\Rightarrow \text{IO} = \sim 5,001,000 + \text{OUT}$	$(1000 + 1000 \cdot 100 \cdot 500 + \text{OUT})$ $\Rightarrow \text{IO} = \sim 5,001,000 + \text{OUT}$
BNLJ	$(500 + 1000 \cdot 500 / (99 - 1))$ $\Rightarrow \text{IO} = \sim 5.6\text{K} + \text{OUT}$	$(500 + 1000 \cdot 500 / (19 - 1))$ $\Rightarrow \text{IO} = 28.2\text{K IOs} + \text{OUT}$

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

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




Smarter than Cross-Products

What you will
learn about in
this section

0. Index Joins

1. Sort-Merge Join

2. HashPartion Joins



Smarter than Cross-Products: From Quadratic to Nearly Linear

All joins computing the *full cross-product* have a quadratic term

- For example we saw:

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

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

Now we'll see some (nearly) linear joins:

- $\sim O(P(R) + P(S) + \text{OUT})$

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

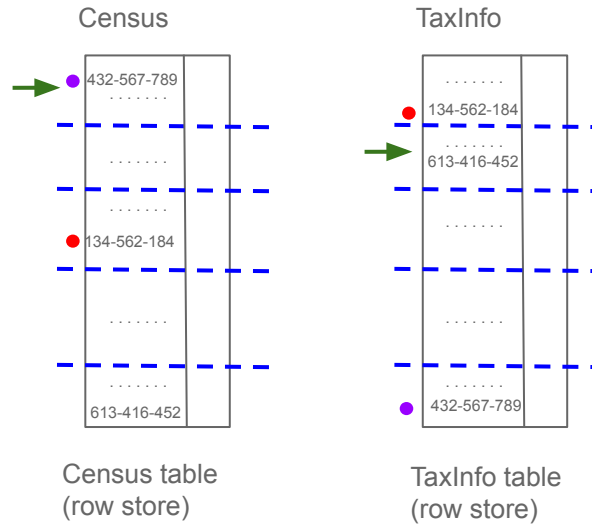
Example

Census (SSN, Address, ...)
TaxInfo (SSN, TaxPaid, ...)

For 1 Billion people

Goal: Compute Census JOIN TaxInfo

Data stored in RowStores, 1000 tuples/page (million pages)



BNLJ -- See all the extra work BNLJ is doing to JOIN for 432-567-789, ...

Index Nested Loop Join (INLJ)

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

Cost:

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

Where L is the IO cost to access each distinct value in index

Recall: L is usually small (e.g., 3-5)

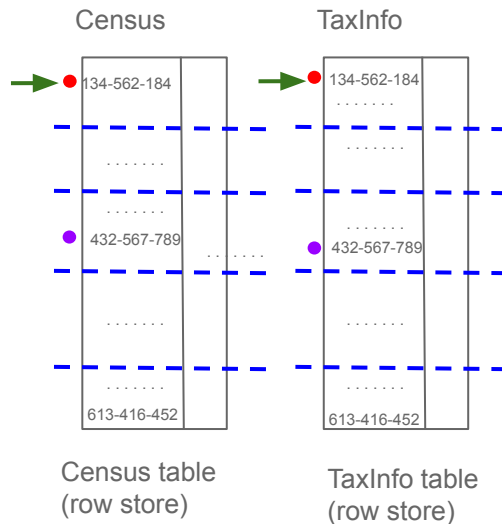
→ We can use an **index** (e.g. B+ Tree) to **avoid full cross-product!**

→ Much better than quadratic. But what if $T(R) = 1$ billion?

Pre-process data before JOINing

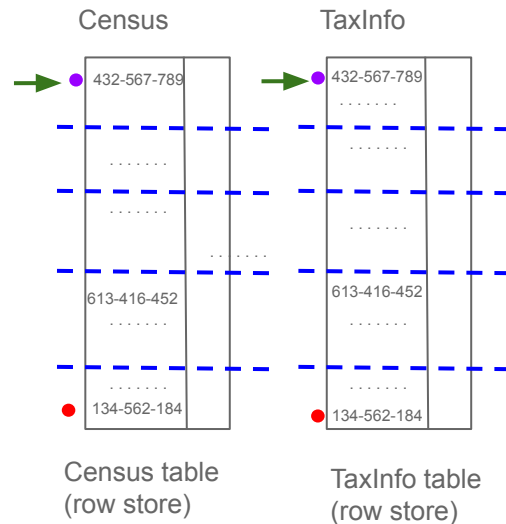
Preview of
doing better?

SortMergeJoin



- Sort(Census), Sort(TaxInfo) on SSN
- Merge sorted pages

HashPartitionJoin



- Hash(Census), Hash(TaxInfo) on SSN
- Merge partitioned pages



Speedy Joins: With Sorting and Hashing

- *Given enough memory*, SortMergeJoin and HashJoins cost

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

[For smaller RAM and “worse” data distribution,
 $\sim k * (P(R)+P(S)) + OUT$, for ‘small’ $k = 3 \dots 5$]

- Hash Joins are highly parallelizable
- Sort-Merge less sensitive to data skew and result is sorted

⇒ Big takeaway: IO-aware join algorithms

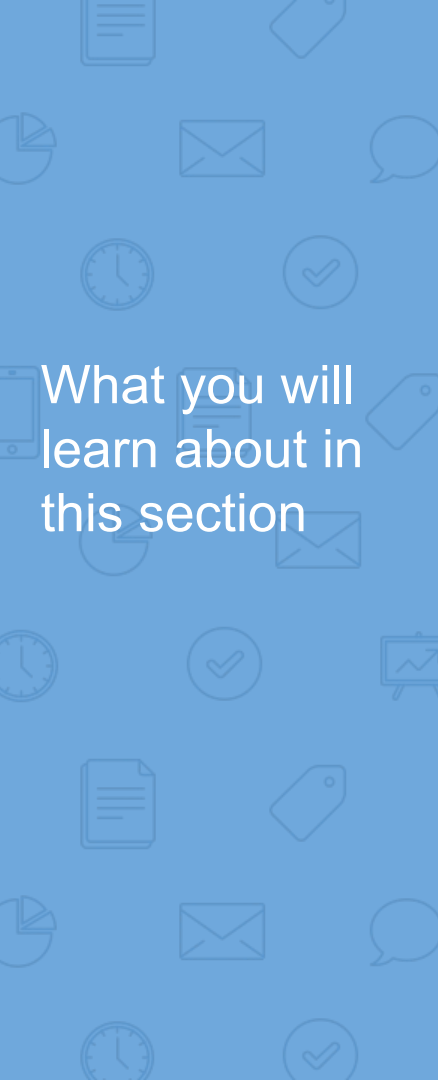
- Massive difference vs brute-force
- Nearly linear vs quadratic (or worse)

What you will
learn about in
this section

0. Intuition for smarter joins
1. Sort-Merge Join
2. HashPartition Joins



Sort-Merge Join (SMJ)

A blue vertical sidebar on the left side of the slide, featuring a repeating pattern of white line-art icons. The icons include a document, a tag, a pie chart, an envelope, a speech bubble, a clock, a checkmark, a smartphone, and a presentation board.

What you will
learn about in
this section

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

Sort Merge Join (SMJ)

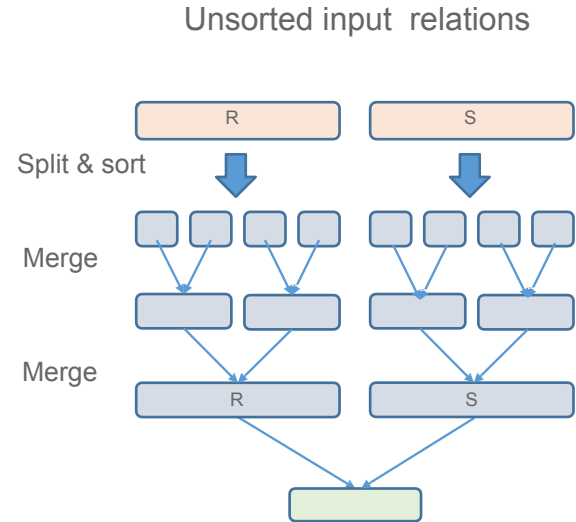
Goal: Execute $R \bowtie S$ on A

Key Idea:

Sort R and S [with external sort]
Merge-scan over them!

IO Cost:

- *Sort phase:* $\text{Sort}(R) + \text{Sort}(S)$
- *Merge / join phase:* $\sim P(R) + P(S) + \text{OUT}$

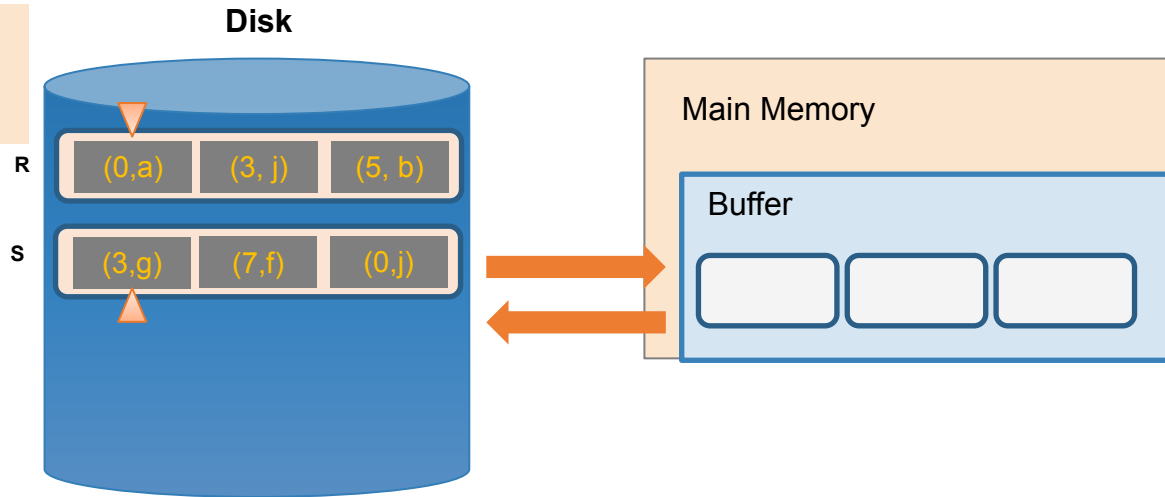


SMJ Example: $R \bowtie S$ with 3 page buffer

For simplicity: Let each page be **one tuple**. Let the **first column** be join key

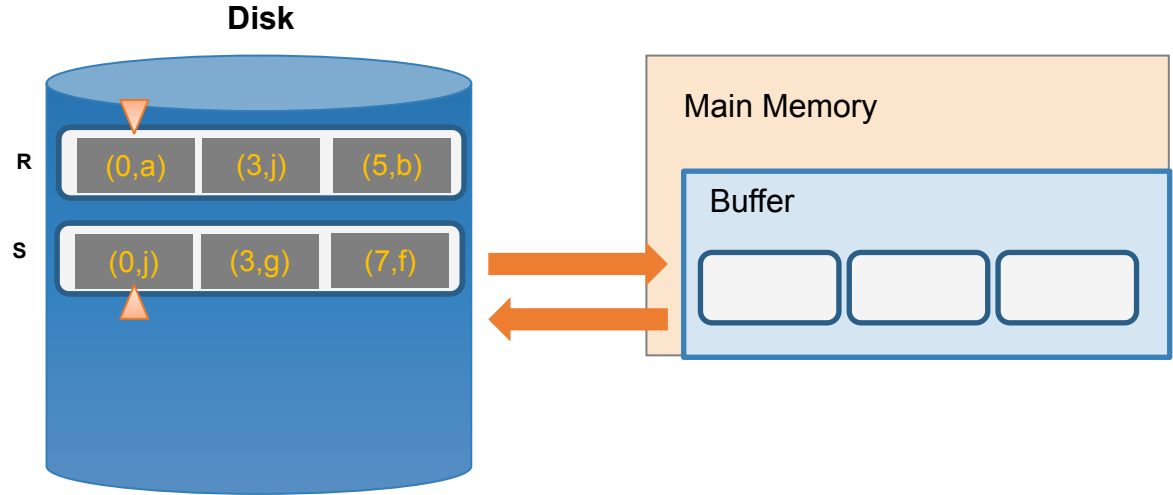
(E.g., R has 3 pages with 1 tuple each)

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



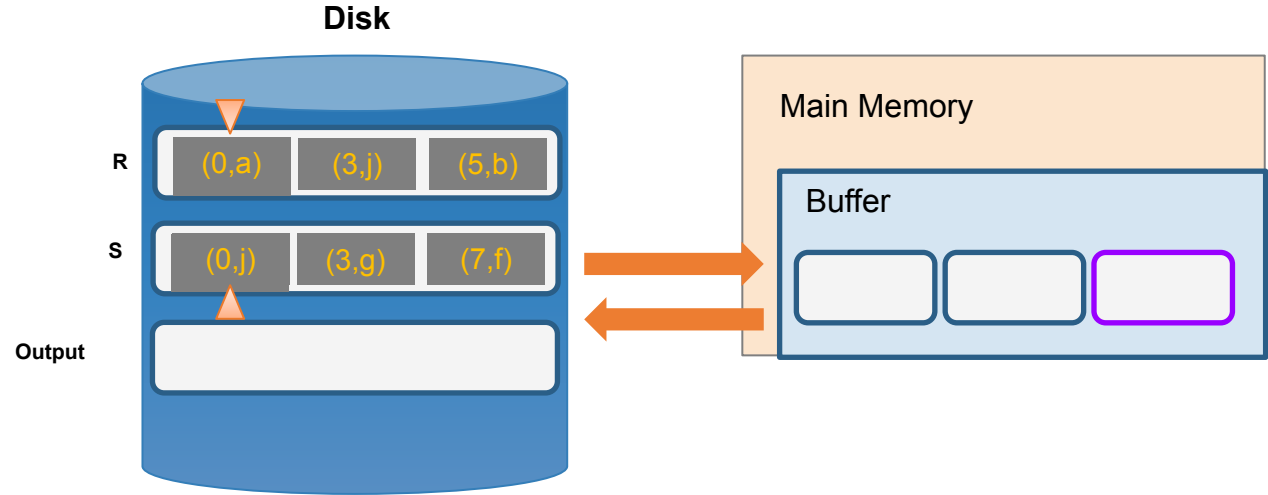
SMJ Example: $R \bowtie S$ with 3 page buffer

1. Sort R and S (on 1st column)



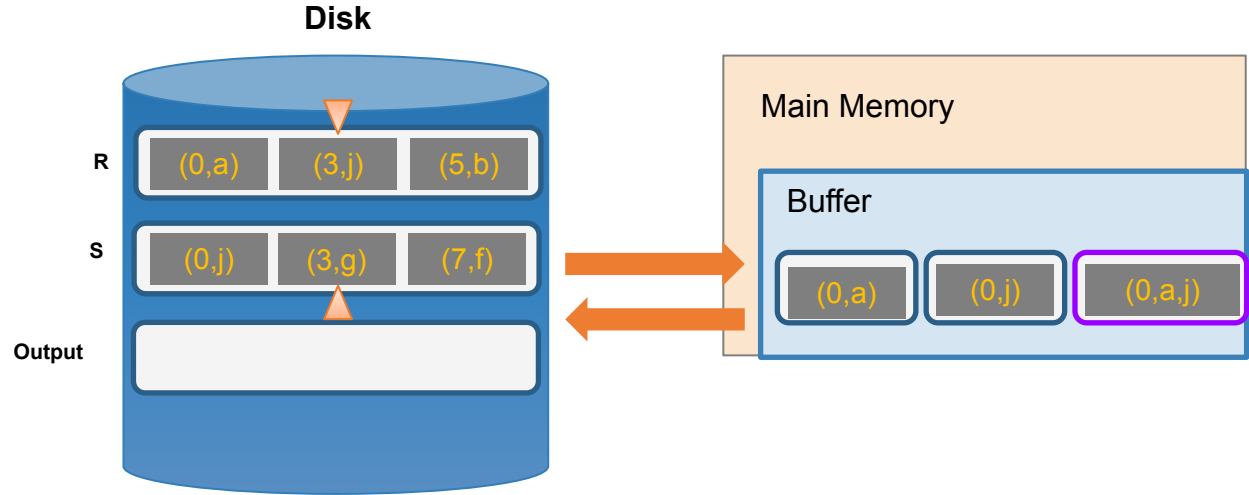
SMJ Example: $R \bowtie S$ with 3 page buffer

2. Scan and “merge” on join key!



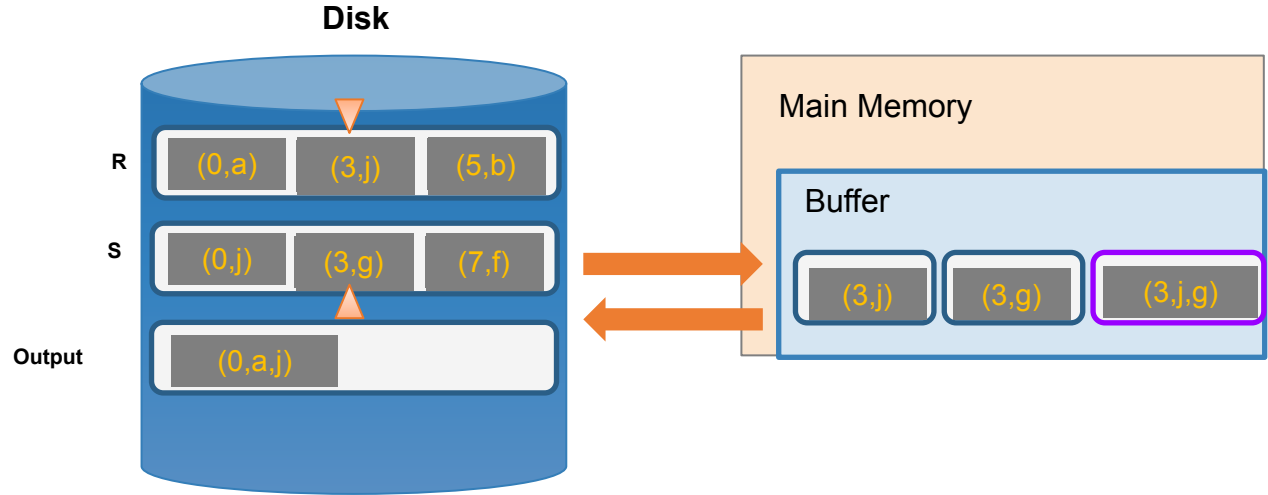
SMJ Example: $R \bowtie S$ with 3 page buffer

2. Scan and “merge” on join key!



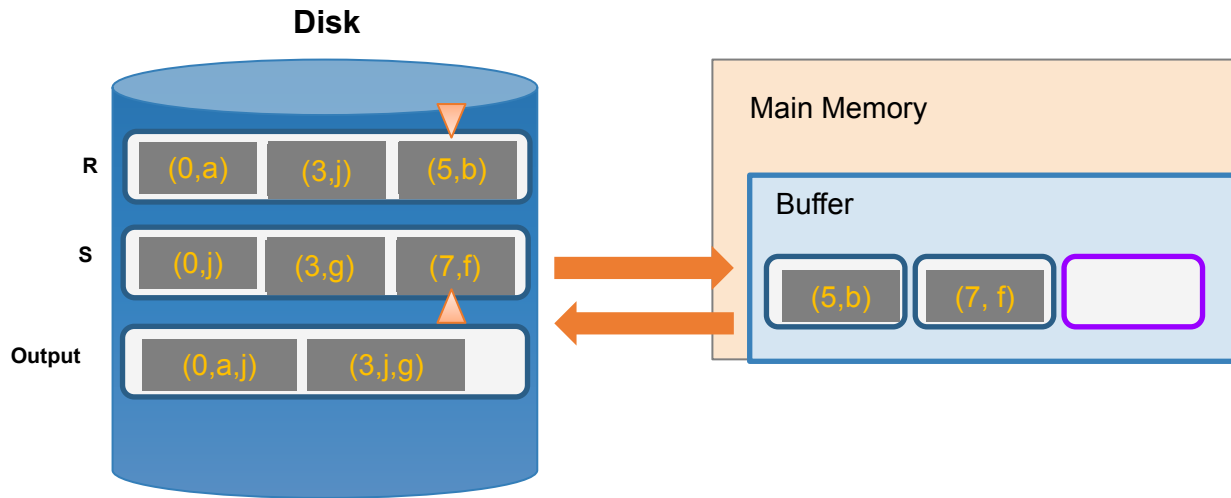
SMJ Example: $R \bowtie S$ with 3 page buffer

2. Scan and “merge” on join key!



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

2. Done!

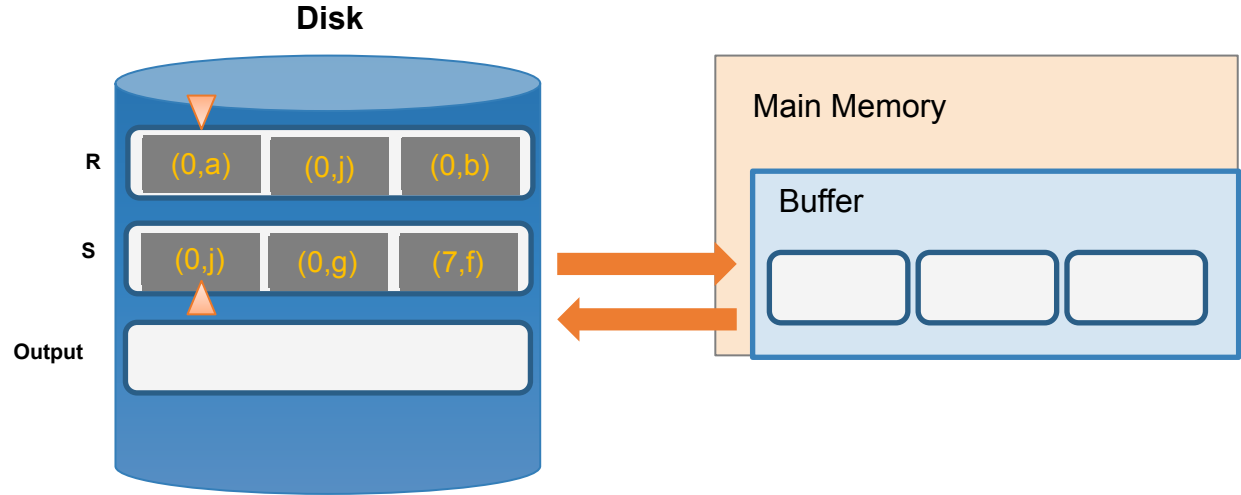


The header features a 3x10 grid of white line-art icons on a blue background. The icons include: a document, a tag, a puzzle piece, a magnifying glass, a smartphone, a document with lines, a tag, a puzzle piece, a magnifying glass, a smartphone, a document with lines, an envelope, a speech bubble, a target with an arrow, two interlocking gears, a pie chart, an envelope, a speech bubble, a target with an arrow, two interlocking gears, a pie chart, a checkmark in a circle, a presentation board with a line graph, a thumbs up, a lightbulb, a clock, a checkmark in a circle, a presentation board with a line graph, a thumbs up, a lightbulb, a clock, and a checkmark in a circle.

What happens with duplicate join keys?

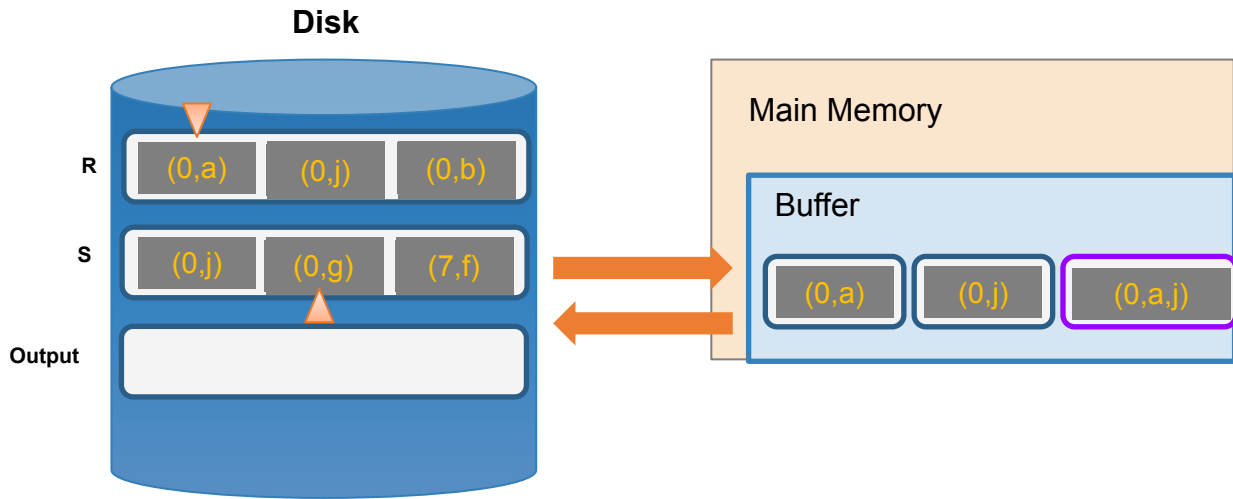
Multiple tuples with Same Join Key: “Backup”

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



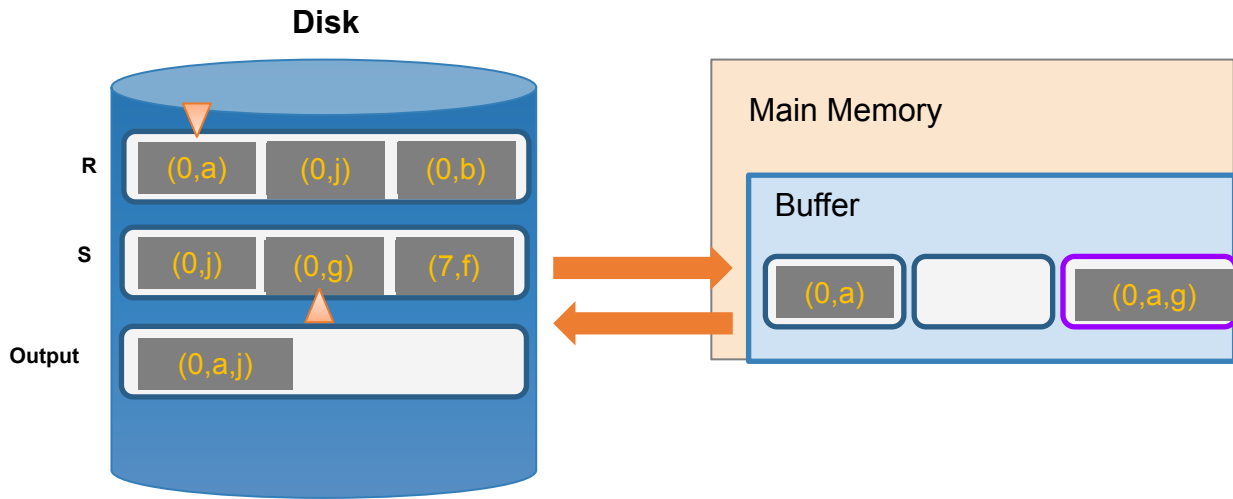
Multiple tuples with Same Join Key: “Backup”

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



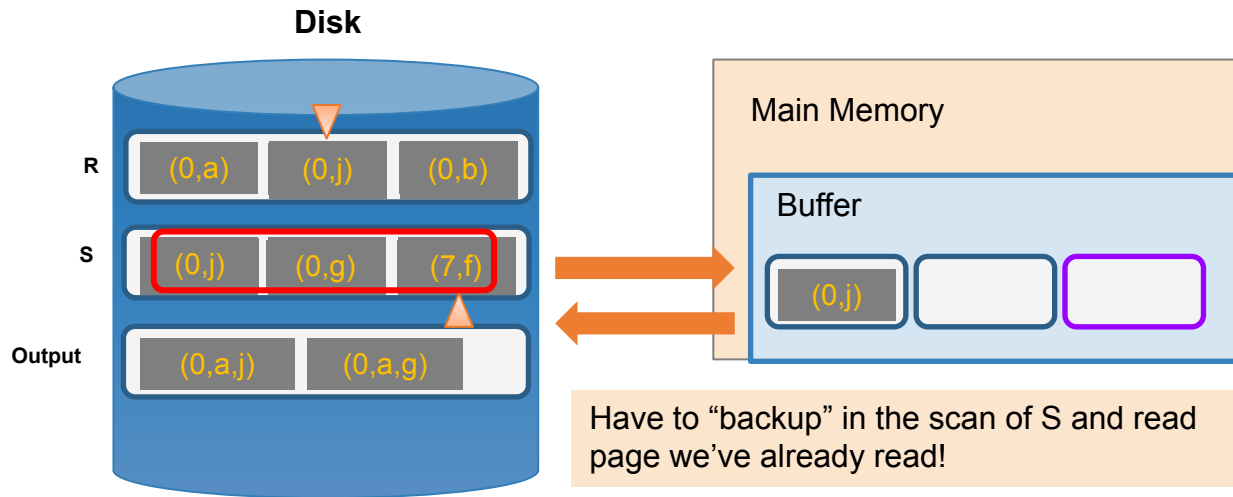
Multiple tuples with Same Join Key: “Backup”

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



Multiple tuples with Same Join Key: “Backup”

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



Backup

- At best, no backup \rightarrow scan takes **$P(R) + P(S)$** reads
 - For ex: if no duplicate values in join attribute
- At worst (e.g. full backup each time), scan could take **$P(R) * P(S)$** reads!
 - For ex: if *all* duplicate values in join attribute, i.e. all tuples in R and S have the same value for the join attribute
 - Roughly: For each page of R, we'll *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 try other algorithms

SMJ: Total cost

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

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

Num passes

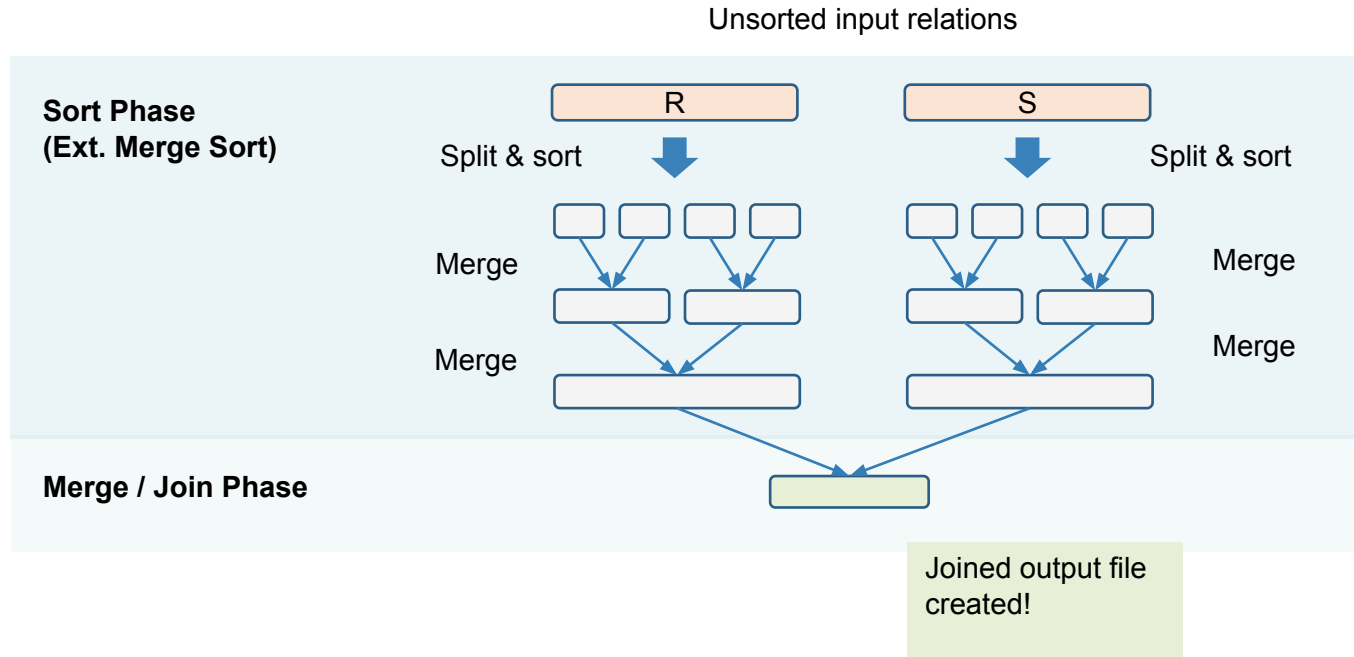


$$\text{Recall: } \text{Sort}(N) \approx 2N \left(\left\lceil \log_B \frac{N}{2(B+1)} \right\rceil + 1 \right)$$

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

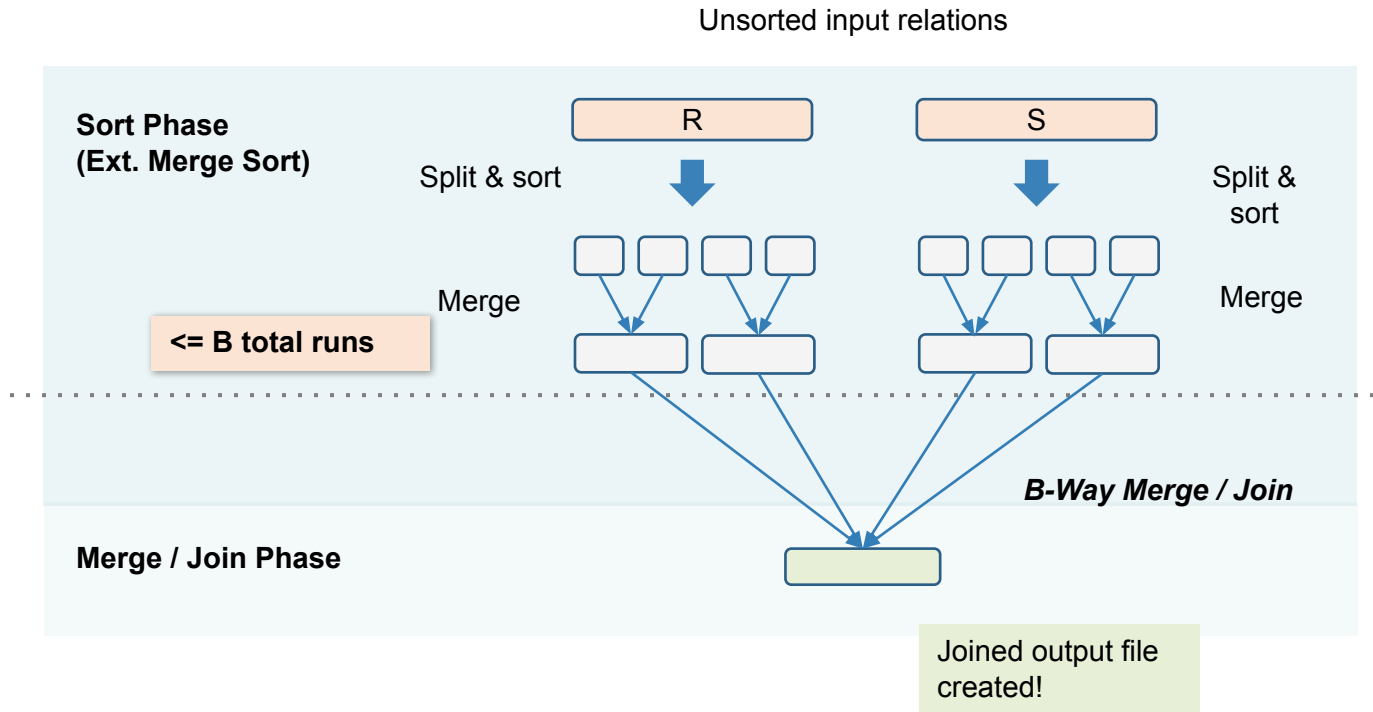
Un-Optimized SMJ

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



Simple SMJ Optimization

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



A close-up photograph of a person's hand holding a blue pen, poised to write on a white sheet of paper. The hand is wearing a grey, textured sweater. The background is blurred, showing a wooden desk and a laptop screen.

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

Example SMJ Number of passes

Consider $P(R) = 1000$, $P(S) = 500$

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

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

Case1: Num passes for R (for $B+1 = 100$) =

$$K = \lceil \log_{99} 1000/(2*100) \rceil + 1 = 2$$

Case2: Num passes for R (for $B+1 = 20$)

$$K = \lceil \log_{19} 1000/(2*20) \rceil + 1 = 3$$

(Repeat for S, and you get $k = 2$ and 3)

Reminder: More Buffer? Fewer passes for Sorting

Example SMJ vs. BNLJ: Steel Cage Match

Consider $P(R) = 1000$, $P(S) = 500$

	Buffer = 100 (i.e., $B+1=100$)	Buffer = 20 (i.e., $B+1=20$)
SMJ $\sim \text{Sort}(P(R)) + \text{Sort}(P(S)) + P(R) + P(S) + \text{OUT}$	(Sort R and S in $k=2$ passes: $2*1000*k + 2*500*k = 6000$ Merge: $1000 + 500 = 1500$ IOs) $\Rightarrow \text{IO} = 7500 \text{ IOs} + \text{OUT}$	(Sort R and S in $k=3$ passes: $2*1000*k + 2*500*k = 9000$ Merge: $1000 + 500: 1500$ IOs) $\Rightarrow \text{IO} = 10,500 \text{ IOs} + \text{OUT}$
$P(R) + \frac{P(R)}{B-1}P(S) + \text{OUT}$	BNLJ $(500 + 1000*500/(99-1))$ $\Rightarrow \text{IO} = \sim 5.6\text{K} + \text{OUT}$	$(500 + 1000*500/(19-1))$ $\Rightarrow \text{IO} = 28.2\text{K IOs} + \text{OUT}$

SMJ is \sim linear vs. BNLJ is quadratic...
 But it's all about the memory.

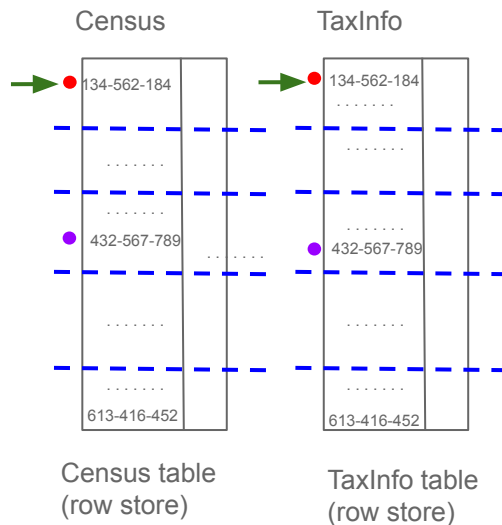
What you will
learn about in
this section

0. Intuition for smarter joins
1. Sort-Merge Join
2. HashPartition Joins

Pre-process data before JOINing

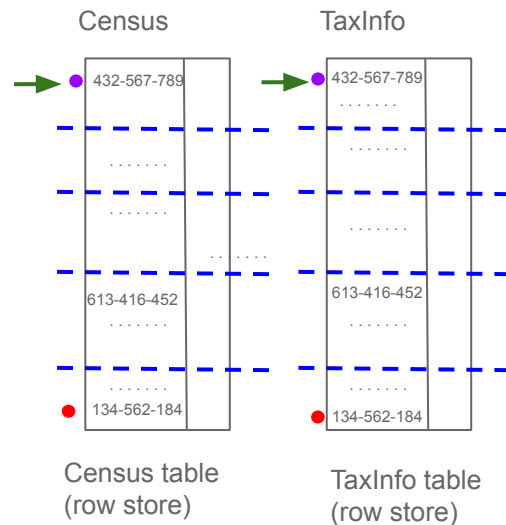
Preview of
smarter joins

SortMergeJoin



- Sort(Census), Sort(TaxInfo) on SSN
- Merge sorted pages

HashPartitionJoin



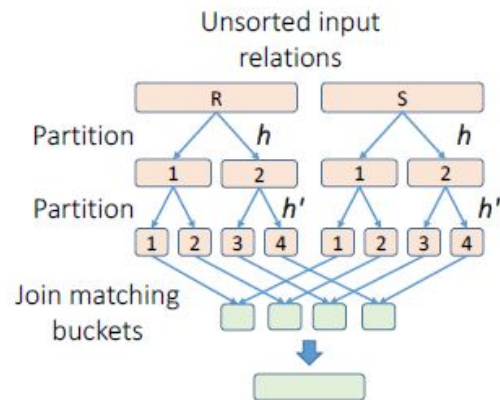
- Hash(Census), Hash(TaxInfo) on SSN
- Merge partitioned pages



Hash Join (HJ) or Hash Partition Join (HPJ)

Hash Join

- **Goal:** Execute $R \bowtie S$ on A
- **Key Idea:**
 - Partition R and S into buckets by hashing the join attribute
 - Join the pairs of (small) matching buckets!





HPJ Phase 1: Hash 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

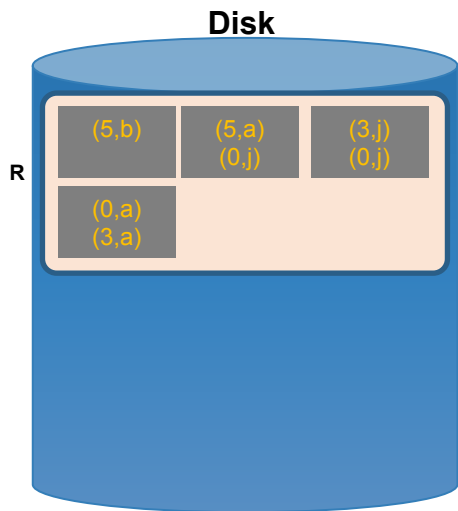
Given $B+1$ buffer pages, we partition into B buckets:

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

HPJ Phase 1: Partitioning

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)

Given $B+1 = 3$ buffer pages



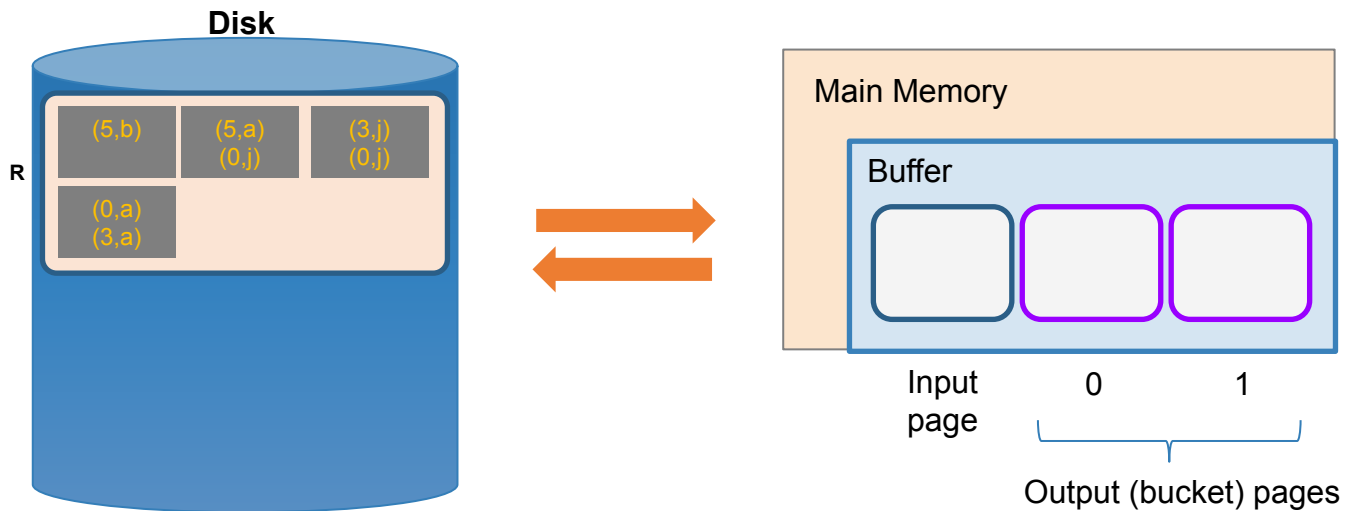
For simplicity, we'll look at partitioning one of the two relations- we just do the same for the other relation!

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

HPJ Phase 1: Partitioning

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

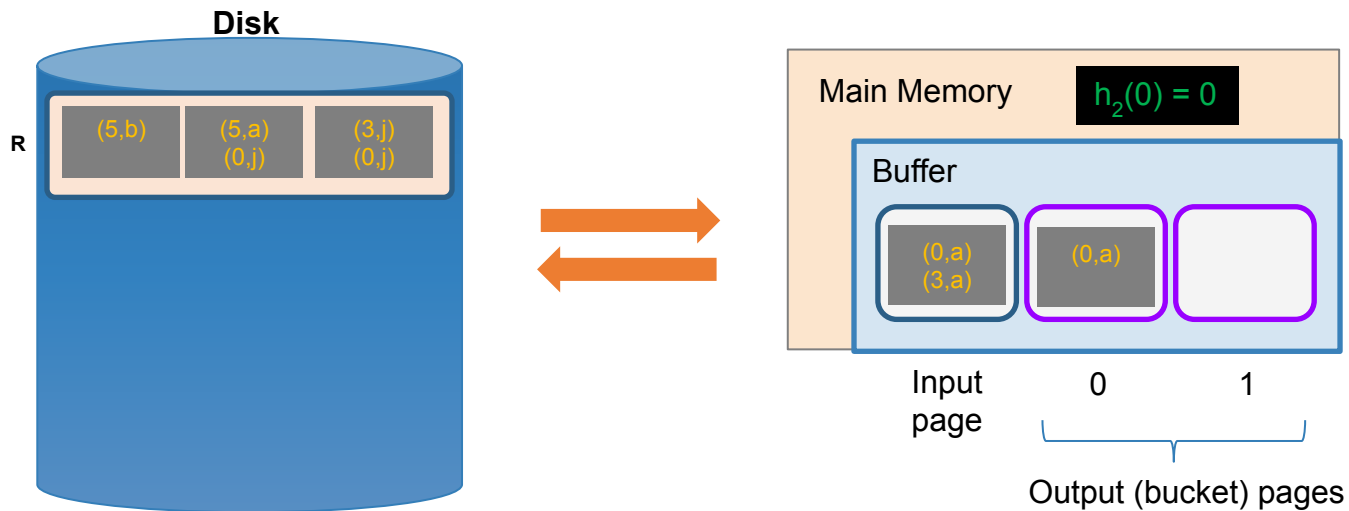
Given $B+1 = 3$
buffer pages



HPJ Phase 1: Partitioning

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

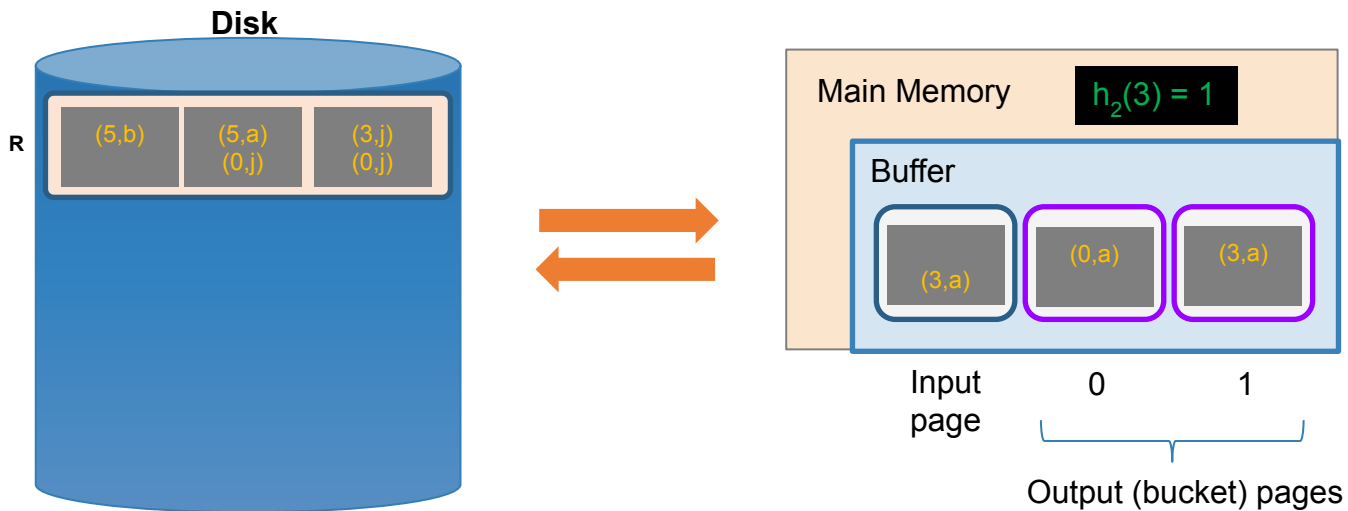
Given $B+1 = 3$
buffer pages



HPJ Phase 1: Partitioning

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

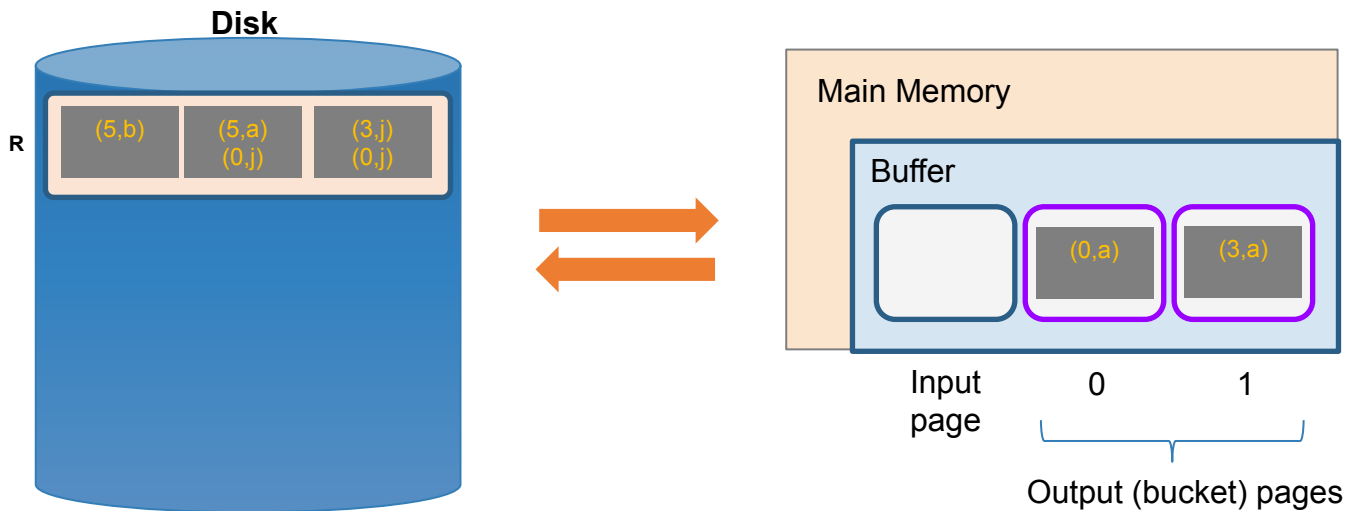
Given $B+1 = 3$
buffer pages



HPJ Phase 1: Partitioning

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

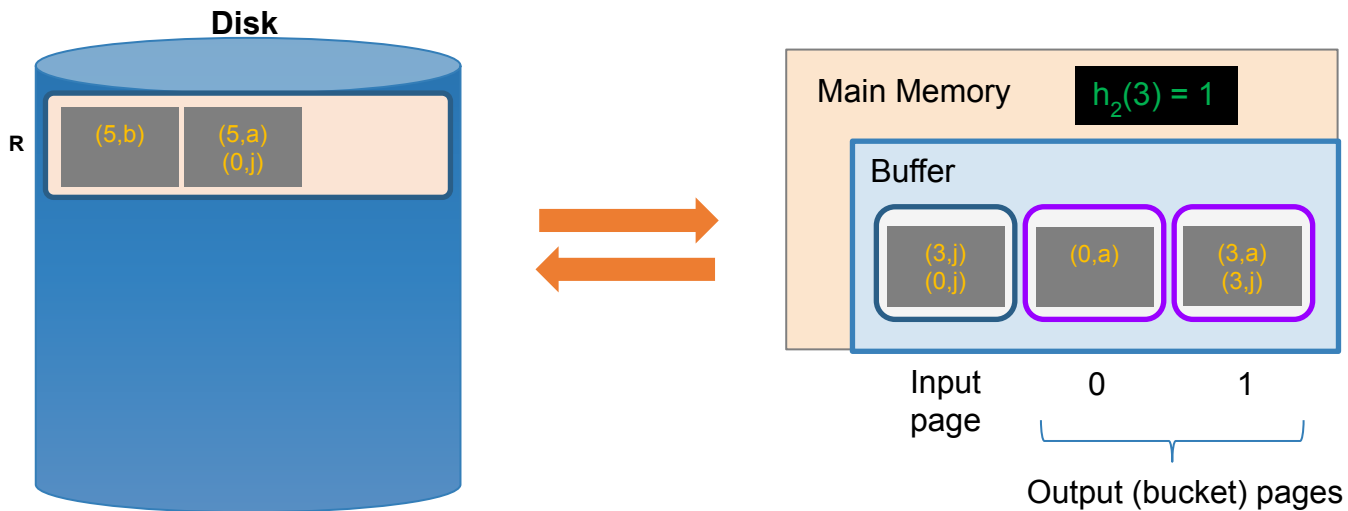
Given $B+1 = 3$
buffer pages



HPJ Phase 1: Partitioning

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

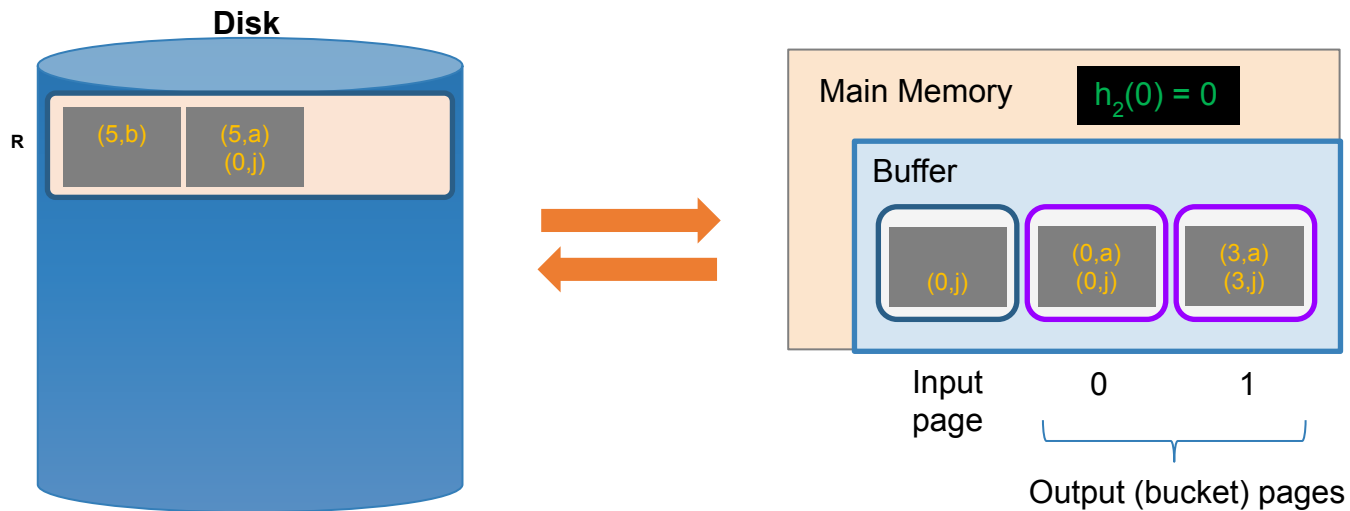
Given $B+1 = 3$
buffer pages



HPJ Phase 1: Partitioning

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

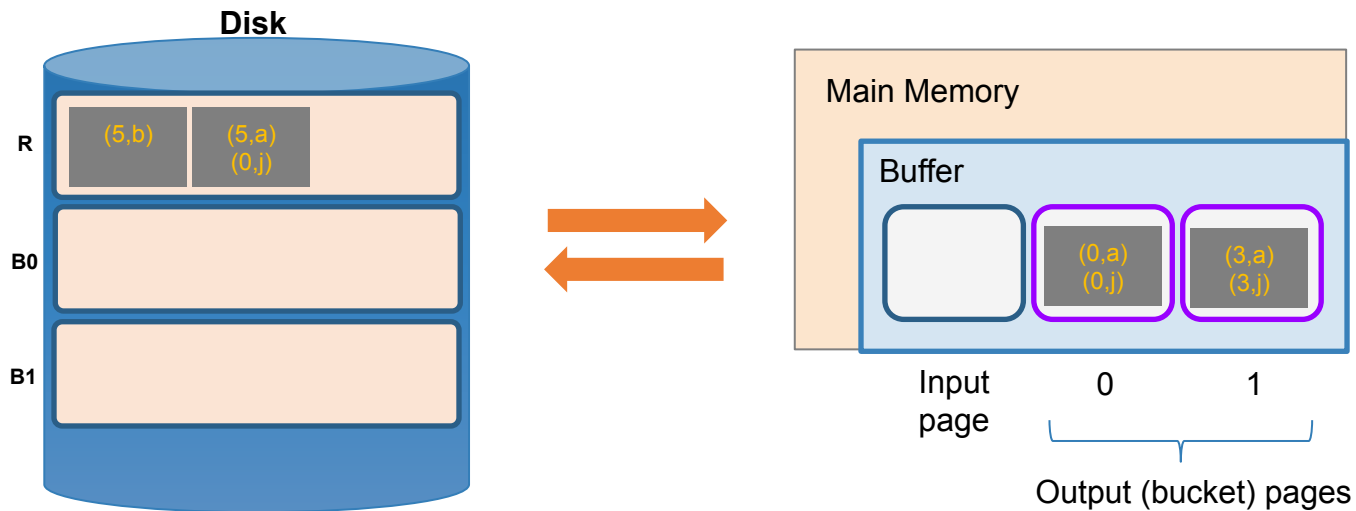
Given $B+1 = 3$
buffer pages



HPJ Phase 1: Partitioning

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

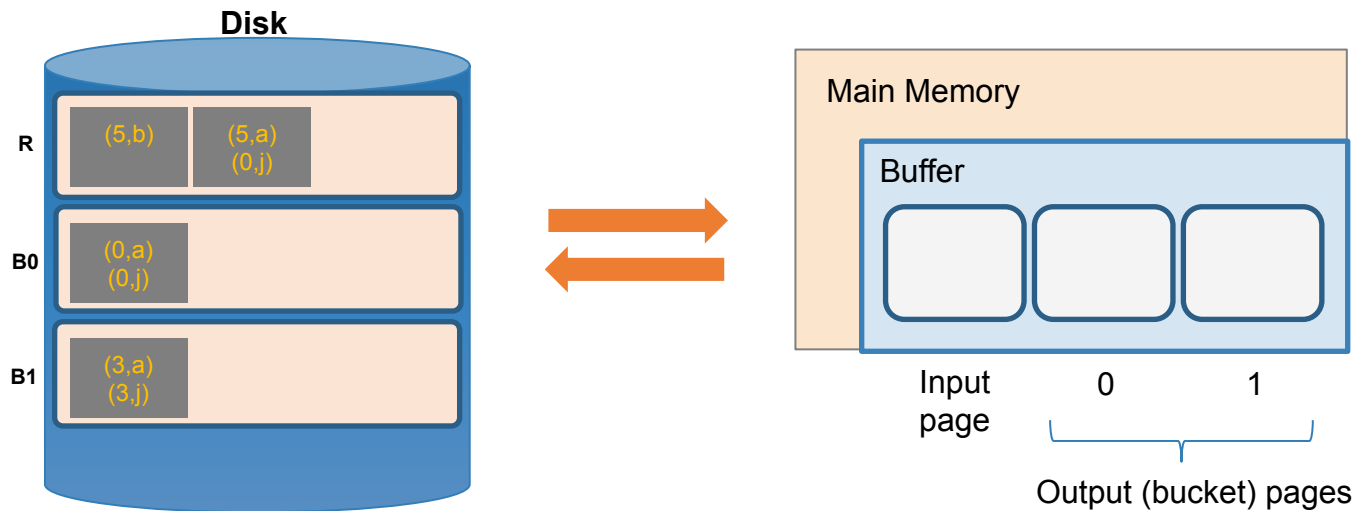
Given $B+1 = 3$
buffer pages



HPJ Phase 1: Partitioning

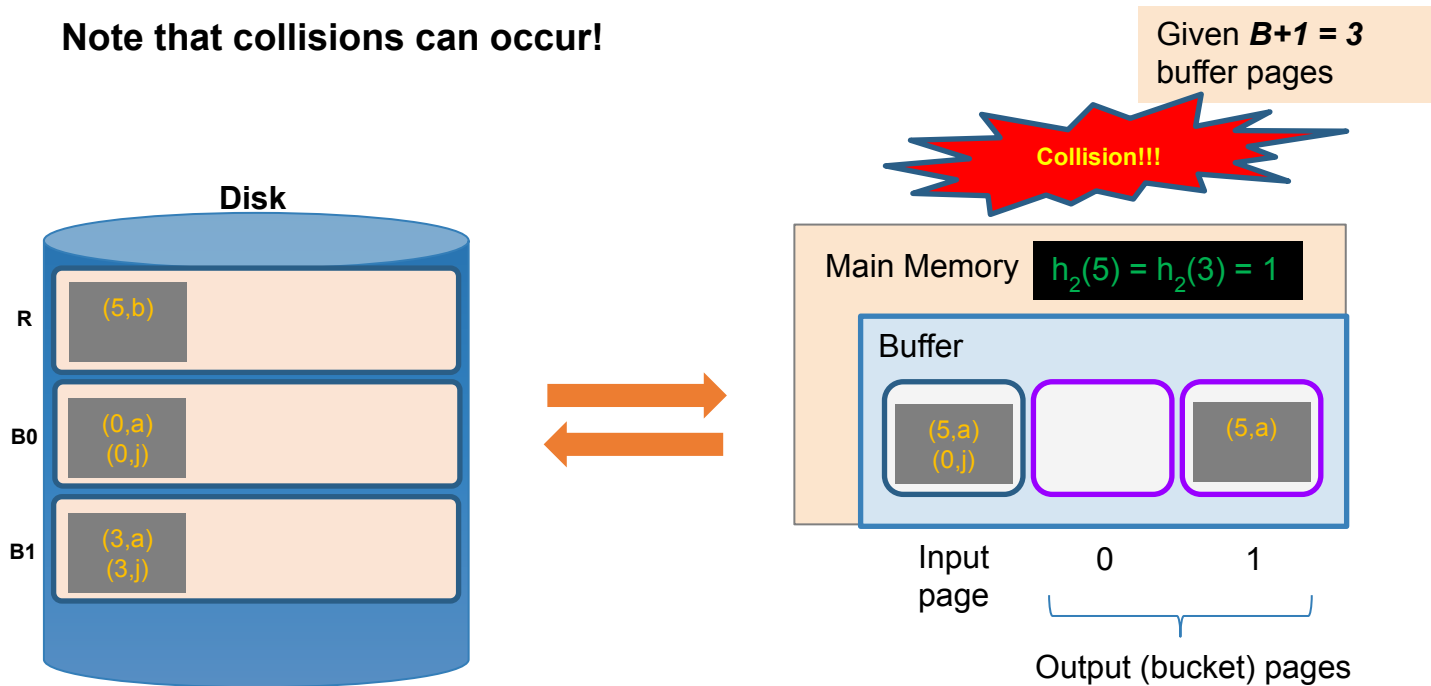
3. We repeat until the buffer bucket pages are full... then flush to disk

Given $B+1 = 3$
buffer pages



HPJ Phase 1: Partitioning

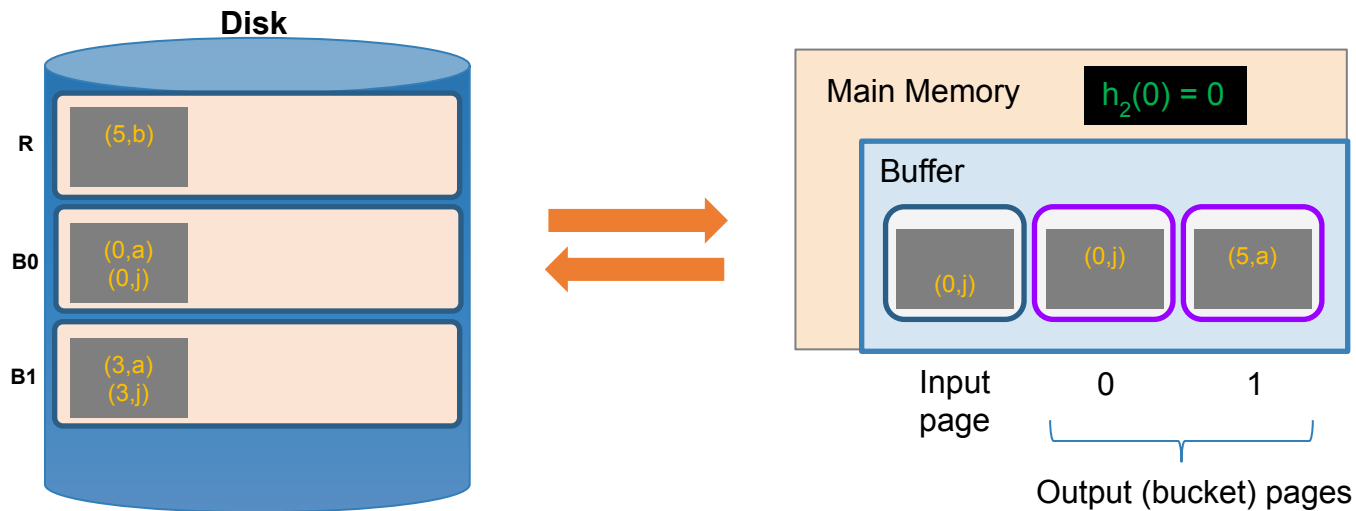
Note that collisions can occur!



HPJ Phase 1: Partitioning

Finish this pass...

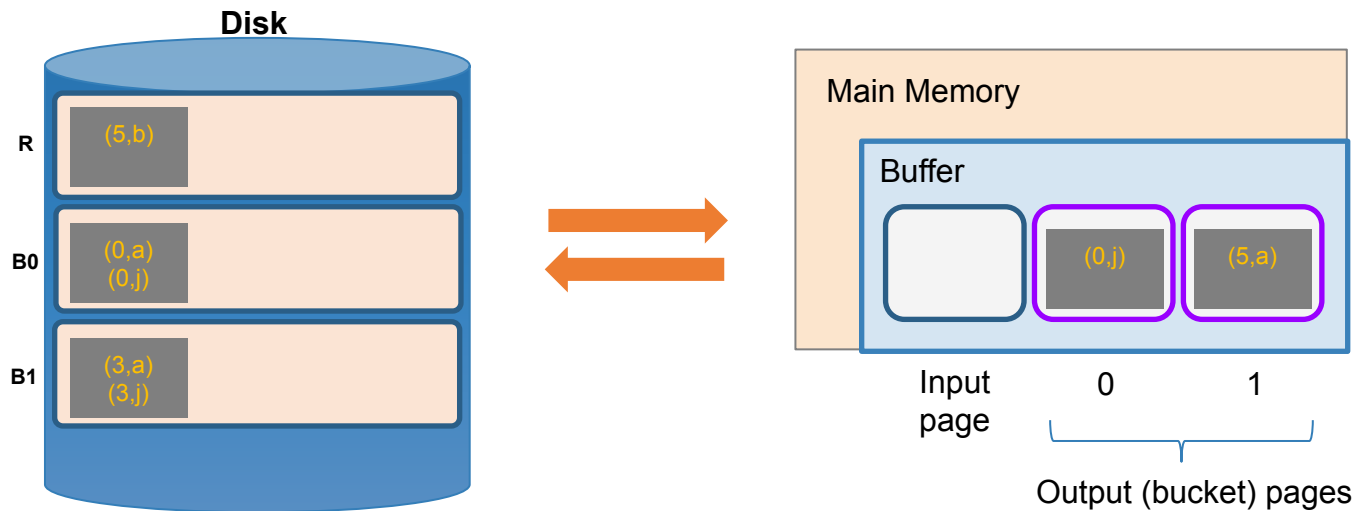
Given $B+1 = 3$
buffer pages



HPJ Phase 1: Partitioning

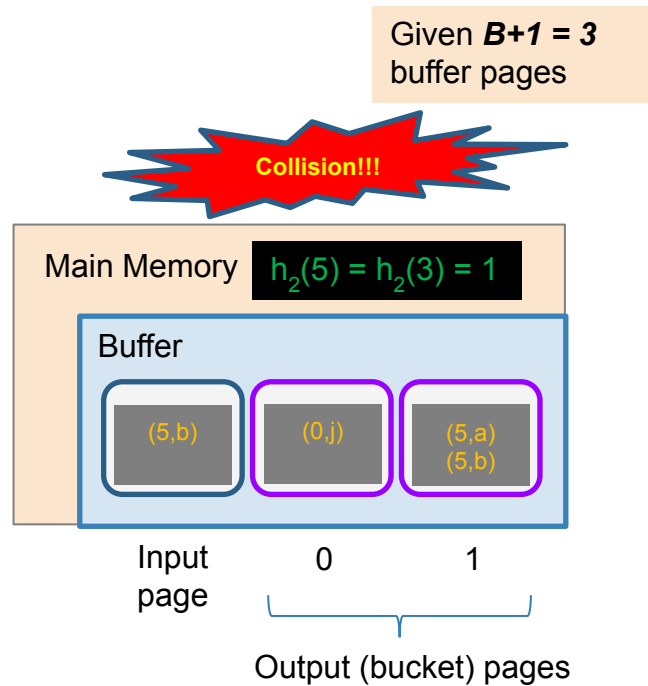
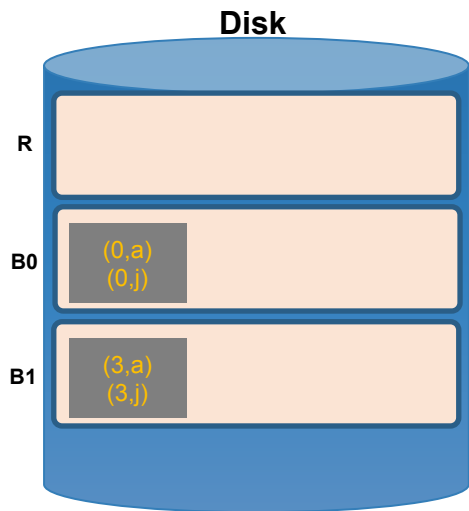
Finish this pass...

Given $B+1 = 3$
buffer pages



HPJ Phase 1: Partitioning

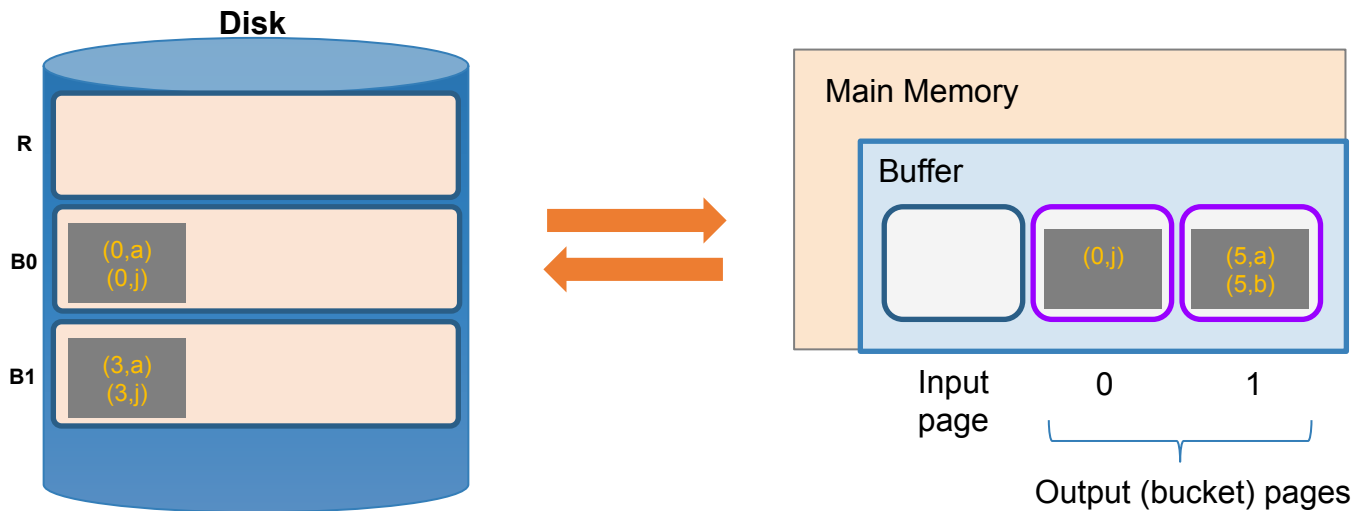
Finish this pass...



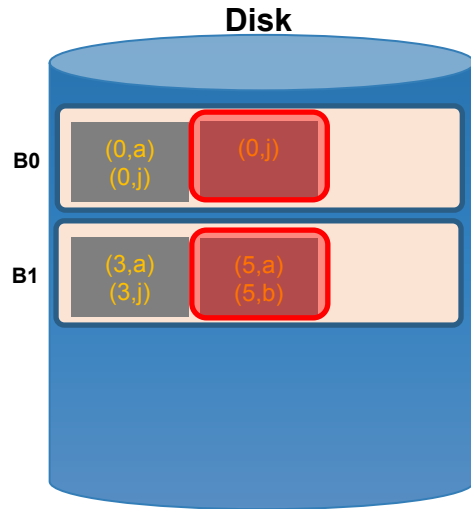
HPJ Phase 1: Partitioning

Finish this pass...

Given $B+1 = 3$
buffer pages



HPJ Phase 1: Partitioning



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

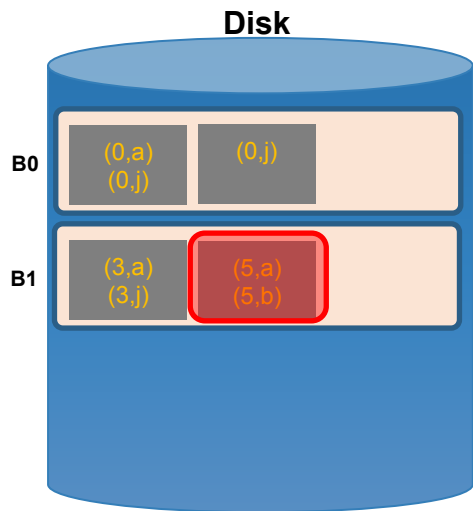
(1) Duplicate join keys

(2) Hash collisions

Given $B+1 = 3$
buffer pages

HPJ Phase 1: Partitioning

Given $B+1 = 3$
buffer pages



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

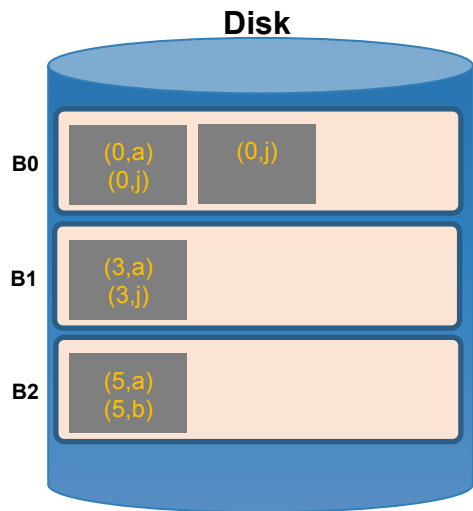
What hash function should we use?

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

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

HPJ Phase 1: Partitioning

Given $B+1 = 3$
buffer pages



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

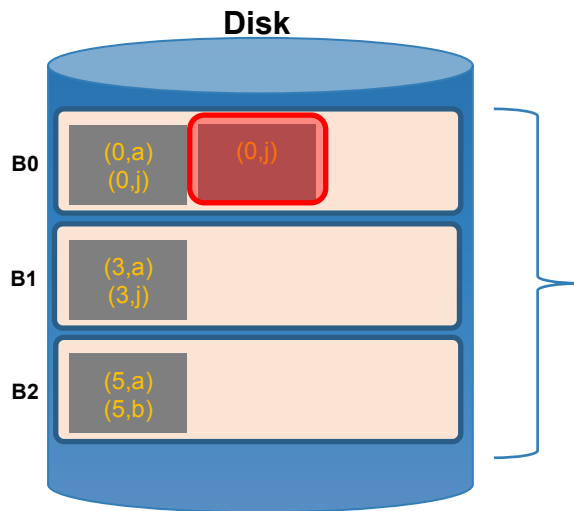
What hash function should we use?

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

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

HPJ 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.

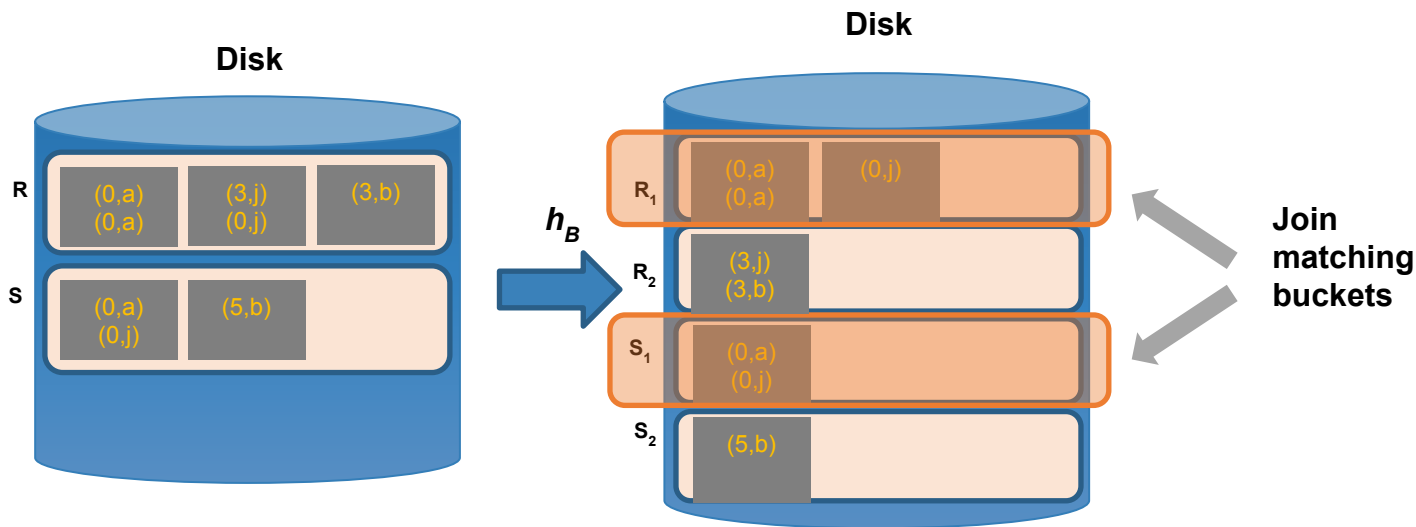
We call this unevenness in the bucket size skew

The header features a 3x10 grid of white line-art icons on a blue background. The icons include: a document, a tag, a puzzle piece, a magnifying glass, a smartphone, a document with lines, a tag, a puzzle piece, a magnifying glass, a smartphone, a document with lines, an envelope, a speech bubble, a target with an arrow, two interlocking gears, a pie chart, an envelope, a speech bubble, a target with an arrow, two interlocking gears, a pie chart, a checkmark in a circle, a presentation board with a line graph, a thumbs up, a lightbulb, a clock, a checkmark in a circle, a presentation board with a line graph, a thumbs up, a lightbulb, a clock, and a checkmark in a circle.

**Now that we have
partitioned R and S...**

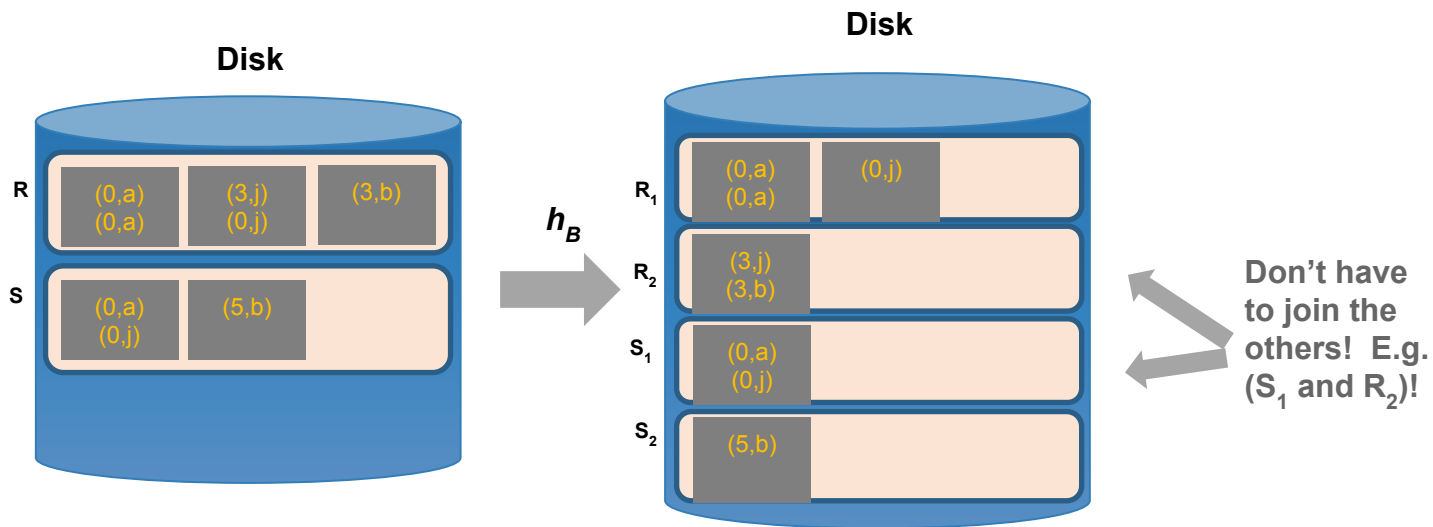
HPJ Phase 2: Partition Join

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



HPJ: High-level procedure

2. Per-Partition Join: JOIN tuples in same partition





HPJ Summary

Given enough buffer pages...

- **Hash Partition** requires reading + writing each page of R,S
 - $\rightarrow 2(P(R)+P(S))$ IOs
- **Partition Join** (with BNLJ) requires reading each page of R,S
 - $\rightarrow P(R) + P(S)$ IOs

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



SMJ vs HPJ Joins Summary

- ***Given enough memory***, both SMJ and HJ have performance:

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

[For smaller RAM and “worse” data distribution,
 $\sim k*(P(R)+P(S)) + OUT$, for small $k = 3...5$]

- Hash Joins are highly parallelizable
 - Sort-Merge less sensitive to data skew and result is sorted
- ⇒ Big takeaway: IO-aware join algorithms
- Massive difference vs brute-force
 - Nearly linear vs quadratic (or worse)