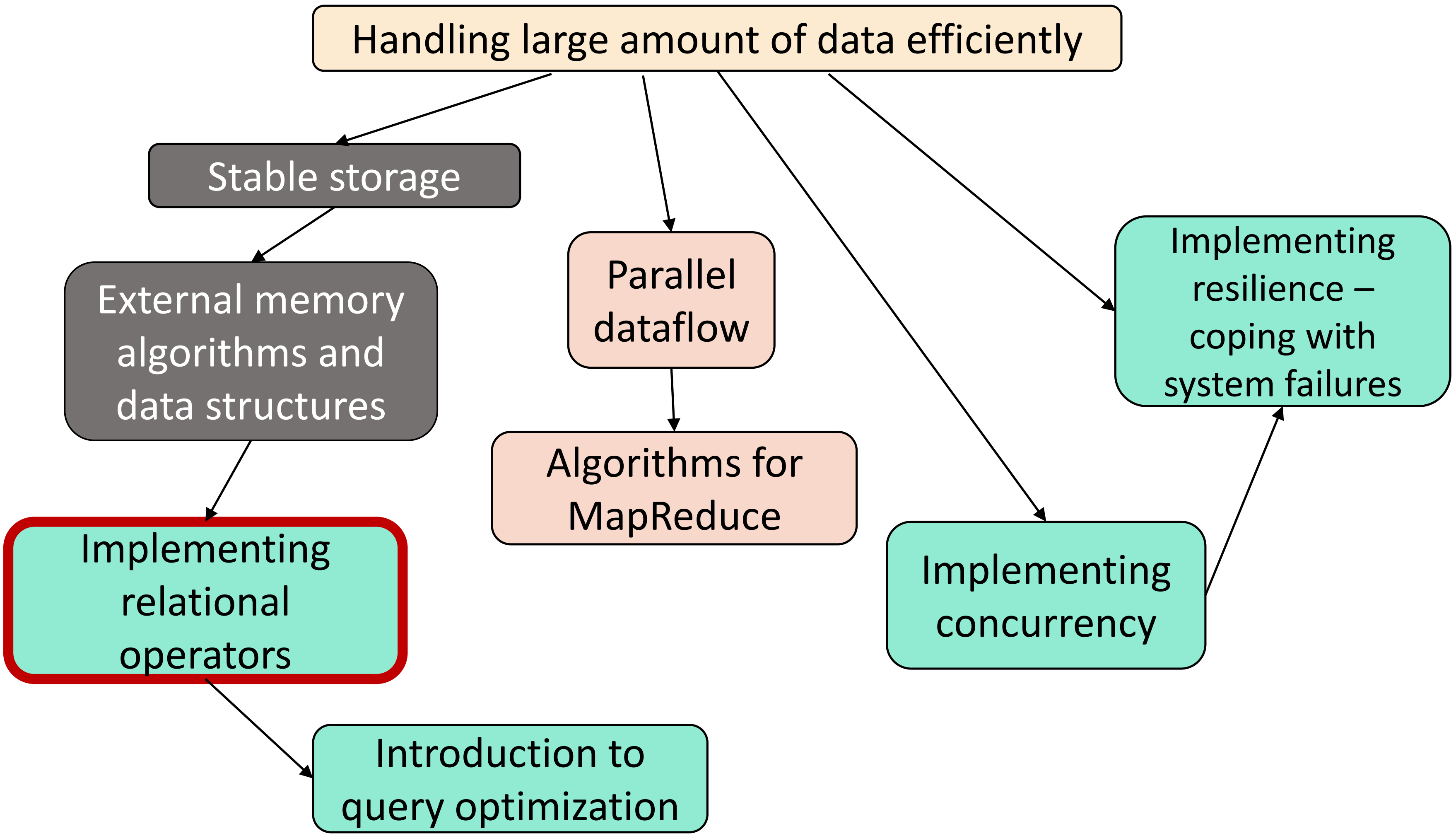
Roadmap


Handling large amount of data efficiently
Stable storage
External memory algorithms and data structures
Parallel dataflow
Algorithms for MapReduce
Implementing resilience – coping with system failures
Implementing concurrency
Implementing relational operators
Introduction to query optimization

Lecture 03.02.

Algorithms for query evaluation

Joins

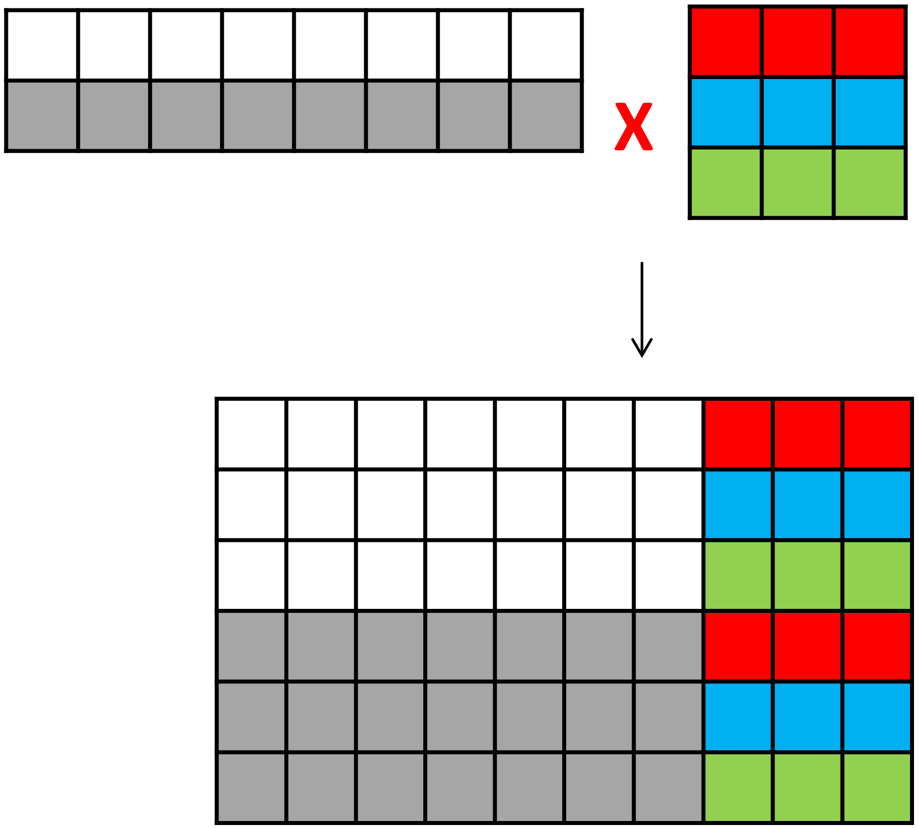
By Marina Barsky
Winter 2017, University of Toronto

Join operator: Cartesian product

1. Set of tuples ***rs*** that are formed by choosing the first part (***r***) to be any tuple of **R** and the second part (***s***) to be any tuple of **S**.

2. Schema for the resulting relation is the union of schemas for **R** and **S**.

3. If **R** and **S** happen to have some attributes in common, then prefix those attributes by the relation name.



T=R x S

Cartesian product (cross-product)

A	B
x	1
y	2

R

C	D	E
x	10	a
y	10	a
z	20	b
z	10	b

S

RxS:

A	B	C	D	E
x	1	x	10	a
x	1	y	10	a
x	1	z	20	b
x	1	z	10	b
y	2	x	10	a
y	2	y	10	a
y	2	z	20	b
y	2	z	10	b

```
SELECT *
FROM R, S
```

If there is no WHERE clause for 2 relations, it is probably a bug, as it will produce a Cartesian product (cross-product) – a huge relation of size T(R)*T(S)

Join: reminder

- ***Natural join*** ($\bowtie$) - a Cartesian product with equality condition on common attributes

  Example:

  - If R has schema $R(A, B, C, D)$, and if S has schema $S(E, B, D)$
  - Common attributes: B and D
  - Then:

$$\mathbf{R \bowtie S} = \pi_{A, B, C, D, E}\,[\sigma_{R.B = S.B \wedge R.D = S.D}\,(R \times S)]$$

- In SQL:

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

  SELECT * FROM R NATURAL JOIN S

Join: Example

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

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

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

R

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

S

A	D
3	7
2	2
2	3

A	B	C	D
2	3	4	2

Join: Example

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

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

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

R

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

S

A	D
3	7
2	2
2	3

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

Join: Example

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

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

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

R

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

S

A	D
3	7
2	2
2	3

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

Join: Example

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

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

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

R

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

S

A	D
3	7
2	2
2	3

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

Join: Example

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

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

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

R

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

S

A	D
3	7
2	2
2	3

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

Semantically: A Subset of the Cross Product

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

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

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

R

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

$\times$

S

A	D
3	7
2	2
2	3

Cross Product

…

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

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

Can we actually implement a join this way?

How do we evaluate the following query:

SELECT B,D

FROM R,S

WHERE R.A = “c” AND S.E = 2 AND R.C=S.C

```
SELECT B,D
FROM R,S
WHERE R.A = “c” AND S.E = 2 AND  R.C=S.C
```

R	A	B	C
	a	1	10
	b	1	20
	c	2	10
	d	2	35
	e	3	45

S	C	D	E
	10	x	2
	20	y	2
	30	z	2
	40	x	1
	50	y	3

Answer	B	D
	2	x

```
SELECT B,D
FROM R,S
WHERE R.A = "c" AND S.E = 2 AND R.C=S.C
```

Plan I

- Do Cartesian product (produce all pairs FROM R, S)
- Select tuples according to WHERE clause
- Do projection: select only columns of SELECT clause

```
SELECT B,D
FROM R,S
WHERE R.A = "c" AND S.E = 2 AND R.C=S.C
```

Product, select, project

RxS

Each row of R is coupled with each row of S

R.A	R.B	R.C	S.C	S.D	S.E
a	1	10	10	x	2
a	1	10	20	y	2
.					
.					
c	2	10	10	x	2
.					
.					

```
SELECT B,D
FROM R,S
WHERE R.A = "c" AND S.E = 2 AND R.C=S.C
```

Scan the resulting (huge!) product table and check conditions

RxS

R.A	R.B	R.C	S.C	S.D	S.E
a	1	10	10	x	2
a	1	10	20	y	2
.					
.					
c	2	10	10	x	2
.					
.					

Bingo!

Got one...

```
SELECT B,D
FROM R,S
WHERE R.A = "c" AND S.E = 2 AND R.C=S.C
```

Plan II

- Do selection on R
- Do selection on S
- Join results on attribute C
- Project B,D columns and place in the result

```
SELECT B,D
FROM R,S
WHERE R.A = “c” AND S.E = 2 AND R.C=S.C
```

Select, join, project

R

A	B	C
a	1	10
b	1	20
c	2	10
d	2	35
e	3	45

σ (R)

A	B	C
c	2	10

σ(S)

C	D	E
10	x	2
20	y	2
30	z	2

S

C	D	E
10	x	2
20	y	2
30	z	2
40	x	1
50	y	3

$\bowtie$

```
SELECT B,D
FROM R,S
WHERE R.A = "c" AND S.E = 2 AND R.C=S.C
```

Plan III

Use R.A and S.C Indexes

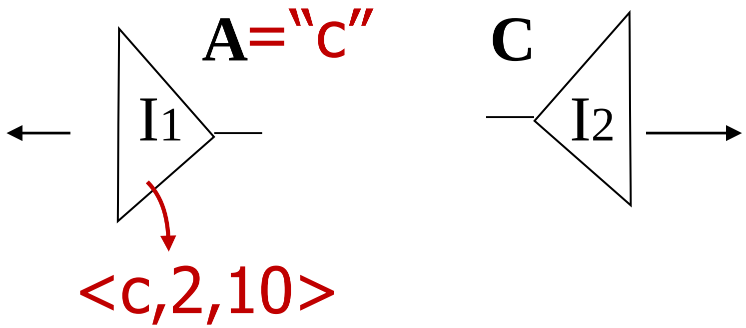
- Use R.A index to select R tuples with R.A = "c"
- For each R.C value found, use S.C index to find matching tuples from S
- Eliminate S tuples where $S.E \neq 2$
- In surviving R,S tuples, project B,D attributes and place in result

Select B,D
From R,S
Where R.A = “c” AND S.E = 2 AND R.C=S.C

Search, join, project

R

A	B	C
a	1	10
b	1	20
c	2	10
d	2	35
e	3	45



S

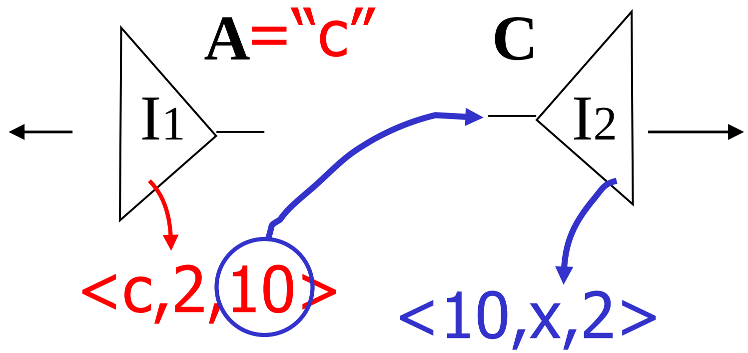
C	D	E
10	x	2
20	y	2
30	z	2
40	x	1
50	y	3

```
SELECT B,D
FROM R,S
WHERE R.A = "c" AND S.E = 2 AND R.C=S.C
```

Search, join, project

R

A	B	C
a	1	10
b	1	20
c	2	10
d	2	35
e	3	45



S

C	D	E
10	x	2
20	y	2
30	z	2
40	x	1
50	y	3

SELECT B,D

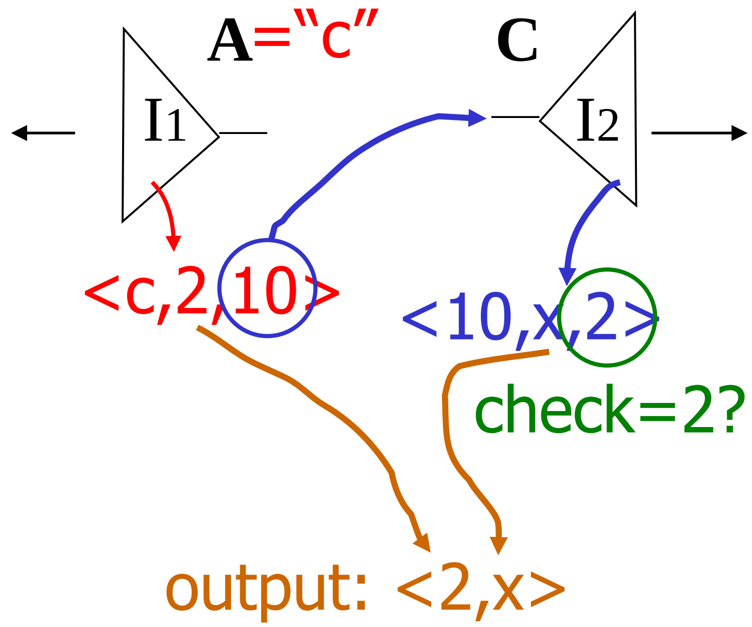
FROM R,S

WHERE R.A = “c” AND S.E = 2 AND R.C=S.C

Search, join, project

R

A	B	C
a	1	10
b	1	20
c	2	10
d	2	35
e	3	45



S

C	D	E
10	x	2
20	y	2
30	z	2
40	x	1
50	y	3

Join algorithm I: Nested Loop

Setup

- We write **R ⋈ *S*** to mean *join R and S by returning all tuple pairs where **all shared attributes** are equal*

- We write **R ⋈ *S* on *A*** to mean *join R and S by returning all tuple pairs where **attribute(s) A** are equal*

- For simplicity, we'll consider joins on **two tables** and with **equality constraints** ("equijoins")

However joins *can* merge > 2 tables, and some algorithms do support non-equality constraints!

- Given a relation R, let:
  - **T(R)** = # of tuples in R
  - **B(R)** = # of blocks (pages) in R

Recall that we read / write entire pages with disk IO

Nested Loop Join (NLJ)

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

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

Nested Loop Join (NLJ)

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

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

Cost:

B(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 $\mathrm{R} \bowtie S\ on\ A$:

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

Cost:

B(R) + T(R)*B(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 $R \bowtie S \text{ on } A$:

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

Cost:

B(R) + T(R)*B(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 change the *if* statement!

Nested Loop Join (NLJ)

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

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

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

Cost:

B(R) + T(R)*B(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
4. **Output combined tuple if match**

Nested Loop Join (NLJ)

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

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

Cost:

B(R) + T(R)*B(S)

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

B(***S***) + T(***S***)*B(***R***)

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

Join algorithm IA: Block Nested Loop

IO-aware modification

Block Nested Loop Join (BNLJ)

Given M pages of memory

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

```
for each chunk c_R of R of size M-1:
  load c_R pages of R into mem
  for each p_s page of S:
    for each tuple s in p_s:
      for each tuple r in c_R
        if r[A] == s[A]:
          OUT (r,s)
```

Cost:

$B(R)$

1. **Load in M-1 pages of R at a time (leaving 1 page free for S)**

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

Block Nested Loop Join (BNLJ)

Given ***M*** pages of memory

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

```
for each chunk c_R of R of size M-1:
  load c_R pages of R into mem
  for each p_s page of S:
    for each tuple s in p_s:
      for each tuple r in c_R
        if r[A] == s[A]:
          OUT (r,s)
```

Cost:

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

1. Load in M-1 pages of R at a time (leaving 1 page free for S)
2. **For each (M-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 ***M*** pages of memory

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

```
for each chunk c_R of R of size M-1:
    load c_R pages of R into mem
    for each p_s page of S:
        for each tuple s in p_s:
            for each tuple r in c_R
                if r[A] == s[A]:
                    OUT (r,s)
```

Cost:

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

1. Load in M-1 pages of R at a time (leaving 1 page free for S)
2. For each (M-1)-page segment of R, load each page of S
3. **Check against the join conditions with all in-mem tuples**

BNLJ can also handle non-equality conditions

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 (M-1)-page segment of R***!
  - Still the full cross-product, but more done *in memory*

NLJ

B(R) + T(R)*B(S)

BNLJ

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

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

BNLJ vs. NLJ: Benefits of IO Aware

- Example:
  - B(R) = 500 pages
  - B(S) = 1000 pages
  - T (R) = 50,000 tuples
  - T (S) = 100,000 tuples
  - We have 11 pages of memory (M = 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**

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

Can we do better than Cross-Product?

Smarter than cross-products: from quadratic to nearly linear

- All joins that compute the ***full cross-product*** have some **quadratic** term
  - For example we saw:

NLJ: B(R) + T(R)B(S)

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

- Now we'll see some (nearly) linear joins:
  - ~ O(B(R) + B(S))

We get this gain by ***taking advantage of data structures and algorithms*** – for simplicity considering equality constraints ("equijoin") only!

Join algorithms II: Index Nested Loop

Index Nested Loop Join (INLJ)

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

```
Given index I on S.A:
  for r in R:
    s_L = index I(r[A])
    for s in s_L:
      OUT r,s
```

Cost:

$\mathrm{B(R) + T(R)*(TH_i + \boldsymbol{\mathit{SC(S,A)}})}$

where TH_i is the height of a B-tree and ***SC(S,A)*** is the IO cost to collect all values equal to r[A] in the index of S.A; assuming these fit on one page, ~ 3 is good est.

$\mathrm{B(R) + 3\,T(R)}$

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

INLJ - cost

Algorithm:
for each tuple *r* of *R*, lookup all tuples in *S* with key *r*[*Y*] and output their join with *r*.

- We want to compute $R(X,Y) \bowtie S(Y,Z)$ on Y
- Suppose there is an index on S[Y].

- Cost:
  - **B(R)** to read entire *R* once
  - Each tuple of *R* joins with SC(S,Y) = T(*S*)/V(*S*,*Y*) tuples of *S*, on average.
  - If *S* has a non-clustered index on *Y*:
    → I/O cost is $\mathbf{B(R) + T(R) \times (TH_i + T(S)/V(S,Y))}$
  - If *S* has a clustered index on *Y*:
    → I/O cost is $\mathbf{B(R) + T(R) \times (TH_i + B(S)/V(S,Y))}$

INLJ: cost example

$B(R) + T(R)*(TH_i + \mathit{SC(S,A)}\,)$

INLJ - clustered index on B[*Y*]:
B(R) + T(*R*) x (3 + B(*S*)/V(*S*,*Y*))

BNLJ:

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

- T(R) = 10,000, B(R) = 1000
- T(S) = 5000, B(S) = 500, V(S,Y) = 100
- M = 11

INLJ:

- To compute R(X,Y) ⋈ S(Y,Z) using a clustered index on S[Y]:
  1000 + 10,000*(3+500/100) = **81,000 I/O's**
- Even when the top level of B-tree is buffered:
  1000 + 10,000*(1+500/100) = **61,000 I/O's**

BNLJ:

- 1000 + 100*500 = **51,000 I/Os**

→ Use of index is not beneficial if selection cardinality is high (50 in this example)

Join using sorted indexes

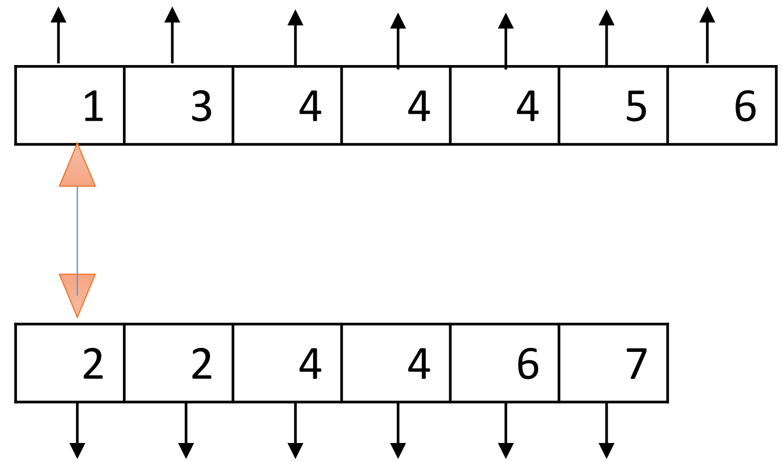
- We want to compute $R(X,Y) \bowtie S(Y,Z)$ on Y

- If both R and S have sorted (B-tree) index on *Y*, do a ***zigzag-join***:
  - We scan the leaves of both B-trees in order. In the best case, we use just **B(R) + B(S)** disk I/O's to read their indexes (if there are no matching values).

Zigzag Join - example

Leaves of B-tree index on R[Y]

Leaves of B-tree index on S[Y]

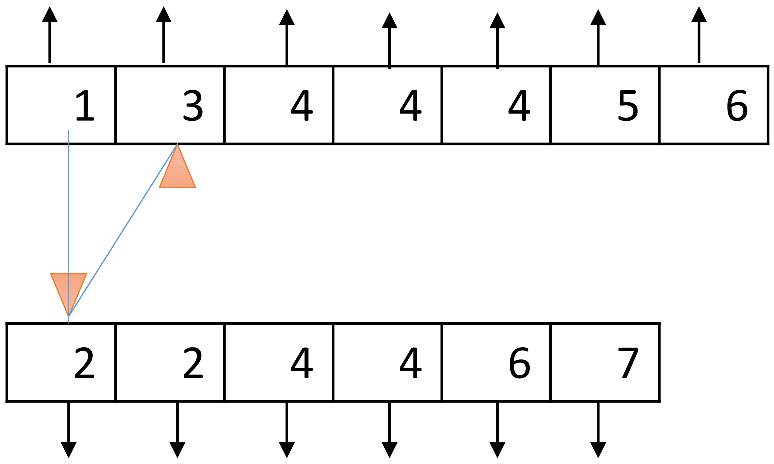


- Start with the 1 and 2. Since 1<2 skip 1 in R’s index.

Zigzag Join - example

Leaves of B-tree index on R[Y]

Leaves of B-tree index on S[Y]

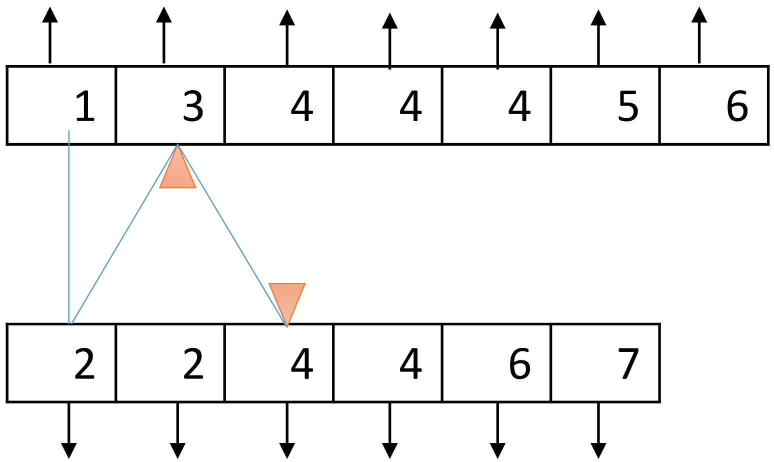


- Start with the 1 and 2. Since 1<2 skip 1 in R’s index.
- Since 2<3 skip the 2’s in S’s index.

Zigzag Join - example

Leaves of B-tree index on R[Y]

Leaves of B-tree index on S[Y]

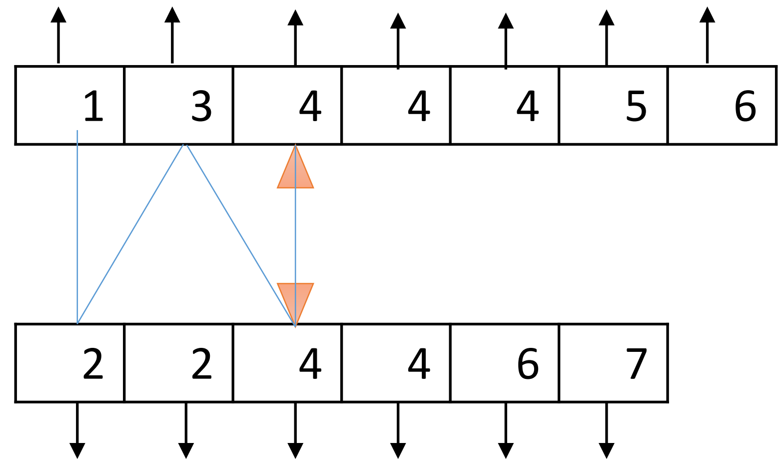


- Start with the 1 and 2. Since 1<2 skip 1 in R's index.
- Since 2<3 skip the 2's in S's index.

Zigzag Join - example

Leaves of B-tree index on R[Y]

Leaves of B-tree index on S[Y]

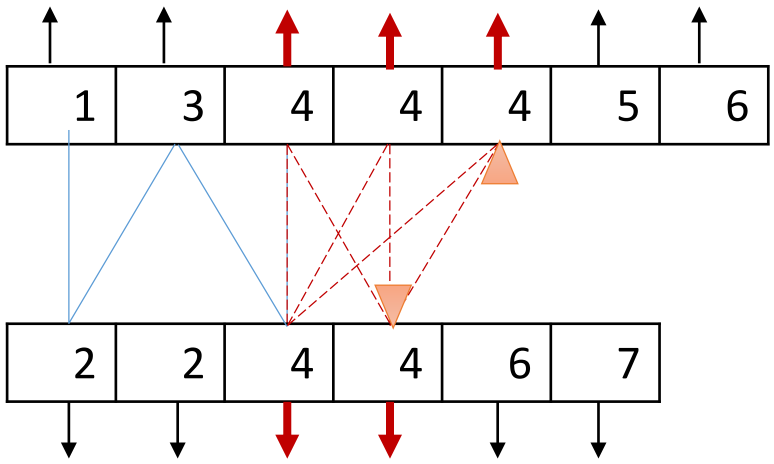


- Start with the 1 and 2. Since 1<2 skip 1 in R's index.
- Since 2<3 skip the 2's in S's index.
- Since 3<4 skip 3 in R.

Zigzag Join - example

Leaves of B-tree index on R[Y]

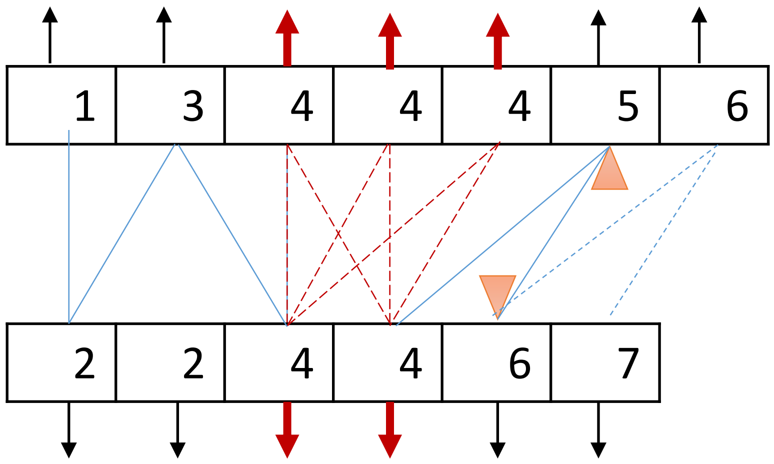
Leaves of B-tree index on S[Y]



- Start with the 1 and 2. Since 1<2 skip 1 in R’s index.
- Since 2<3 skip the 2’s in S’s index.
- Since 3<4 skip 3 in R.
- Join 4’s (retrieve records).

Zigzag Join - example

Leaves of B-tree index on R[Y]



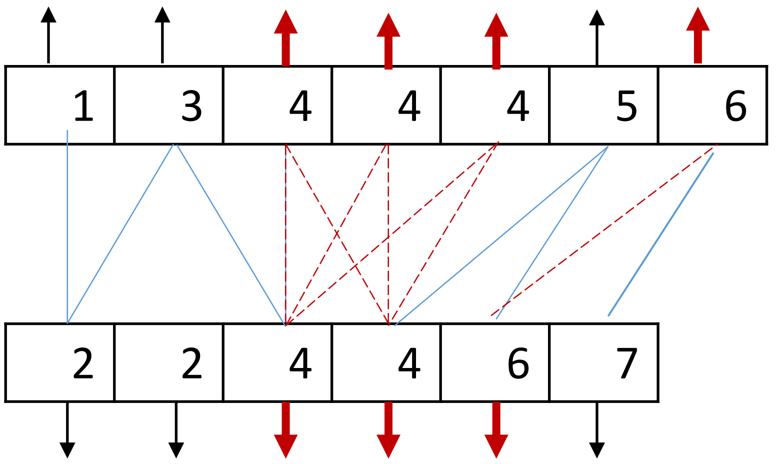
Leaves of B-tree index on S[Y]

- Start with the 1 and 2. Since 1<2 skip 1 in R's index.
- Since 2<3 skip the 2's in S's index.
- Since 3<4 skip 3 in R.
- Join 4's (retrieve records).
- …

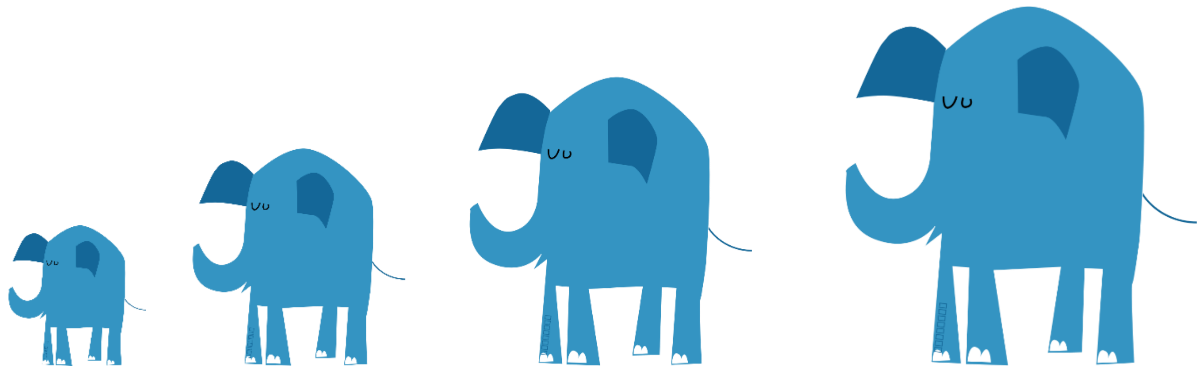
Zigzag Join

Leaves of B-tree index on R[Y]

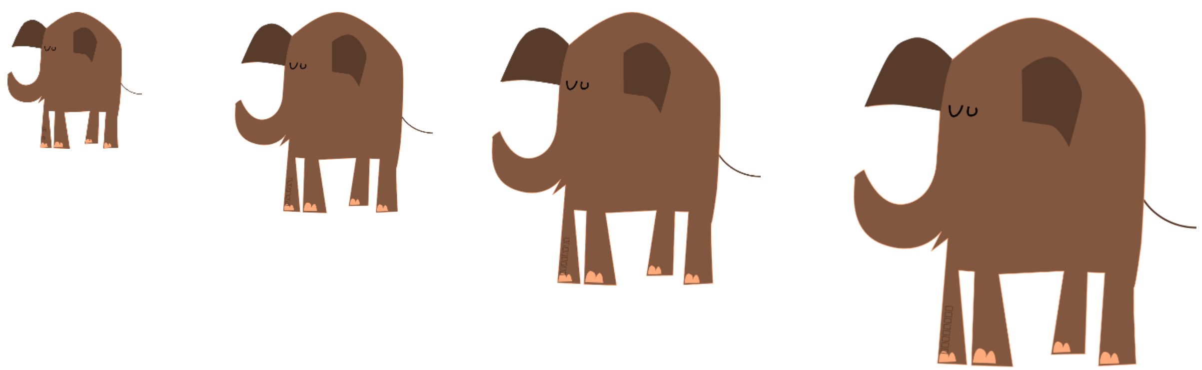
Leaves of B-tree index on S[Y]



- We jump back and forth between the indexes finding Y-values that they share in common.
- Tuples from R with Y-value that don't appear in S need never be retrieved, and similarly tuples of S whose Y-value doesn't appear in R need never be retrieved.
- The worst-case cost (clustered indexes, R < S):
- **B(R) + B(S) + B(R) * B(S) / V (S, a)**



Join algorithm III: Sort-Merge Join (SMJ)



Sort Merge Join (SMJ): Basic Procedure

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

Note that we are only considering equality join conditions here

1. Sort R, S on A using ***external merge sort***

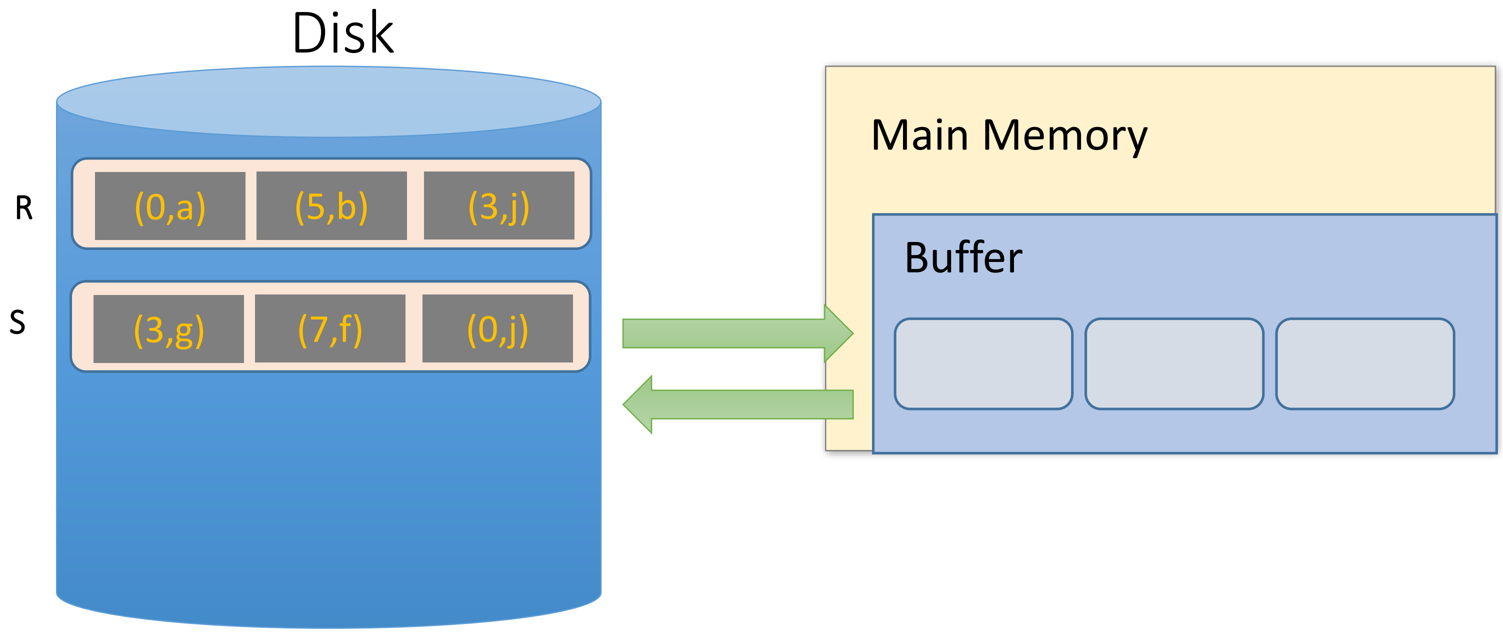
2. Scan sorted files and "merge"

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

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

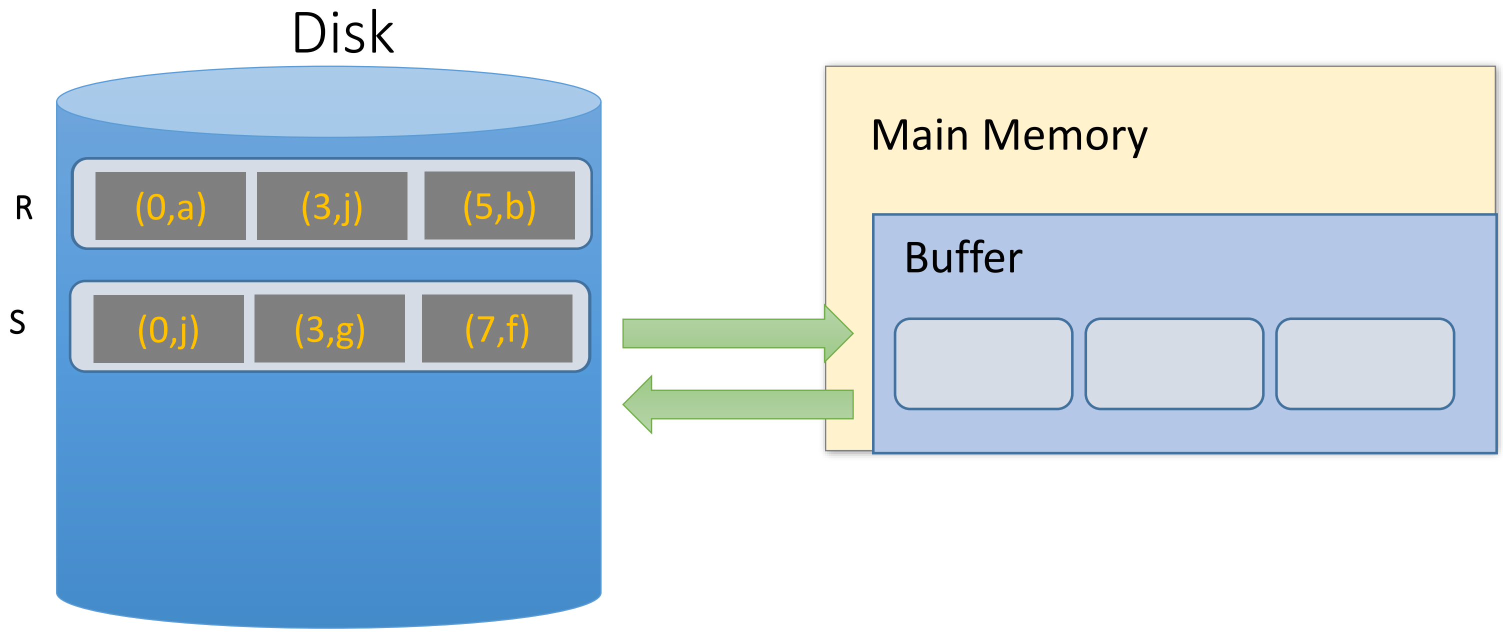
SMJ Example: $\mathrm{R} \bowtie S \; on \; A$ with 3-page buffer

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



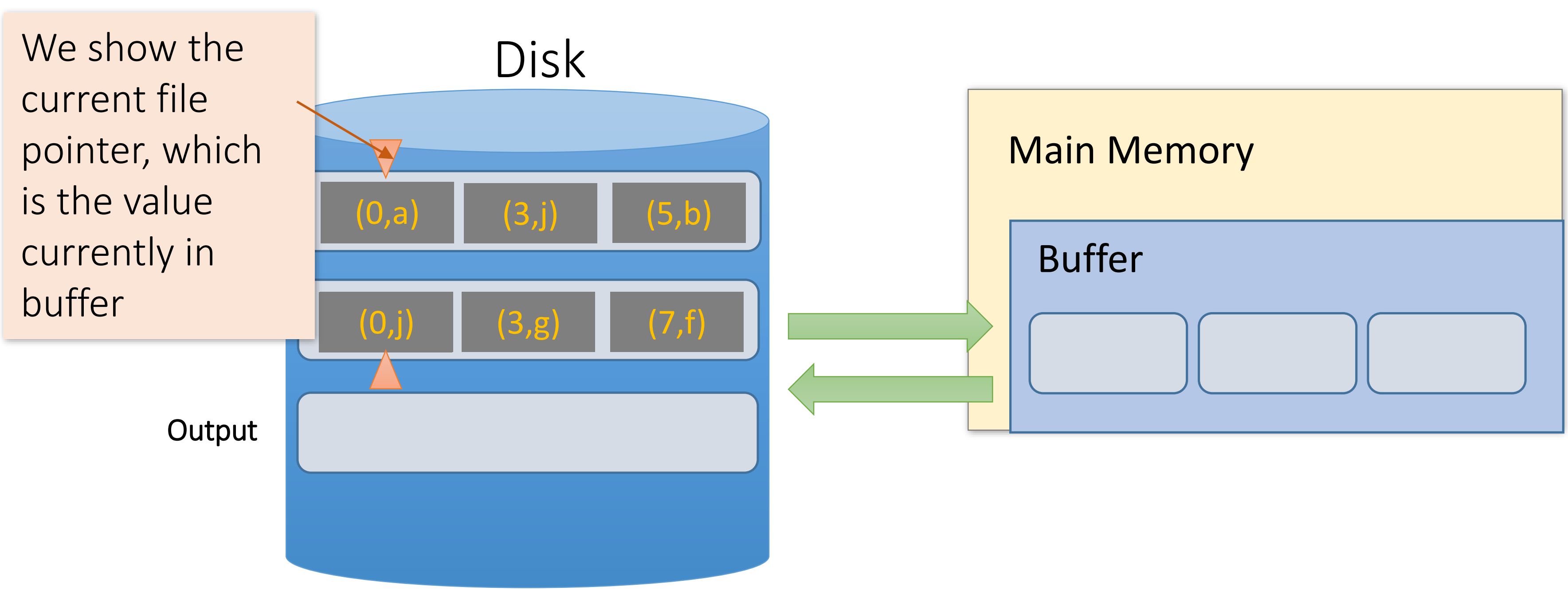
SMJ Example: $\mathrm{R} \bowtie S\ on\ A$ with 3-page buffer

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



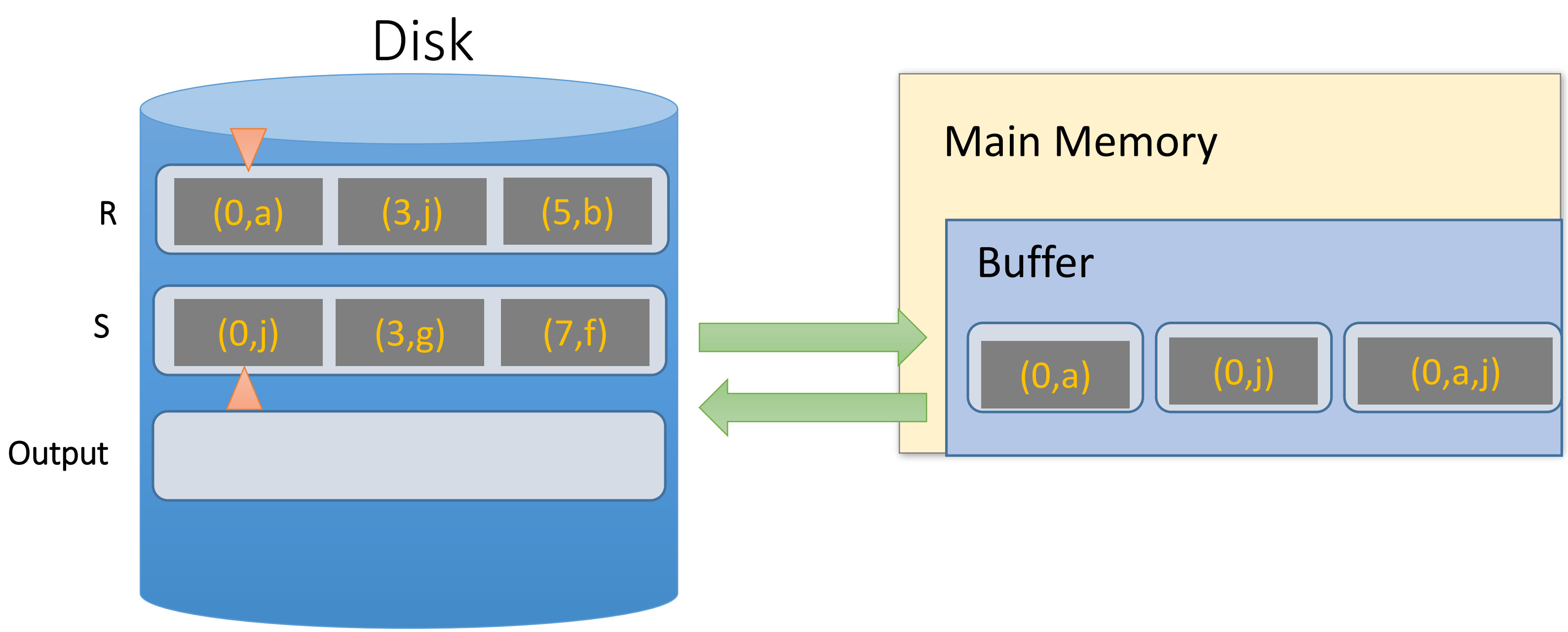
SMJ Example: $\mathrm{R} \bowtie S\ on\ A$ with 3-page buffer

2. Scan and “merge” on join key!



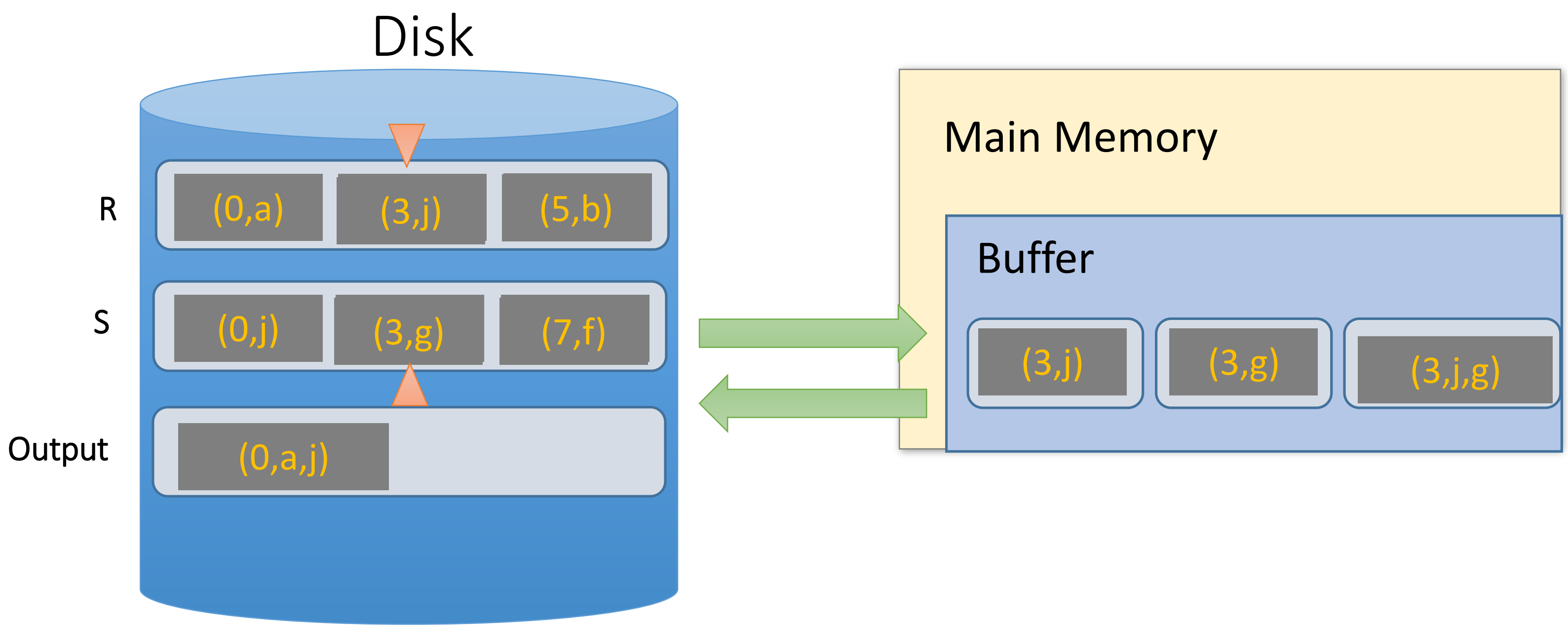
SMJ Example: $\mathrm{R} \bowtie S\ on\ A$ with 3-page buffer

2. Scan and “merge” on join key!



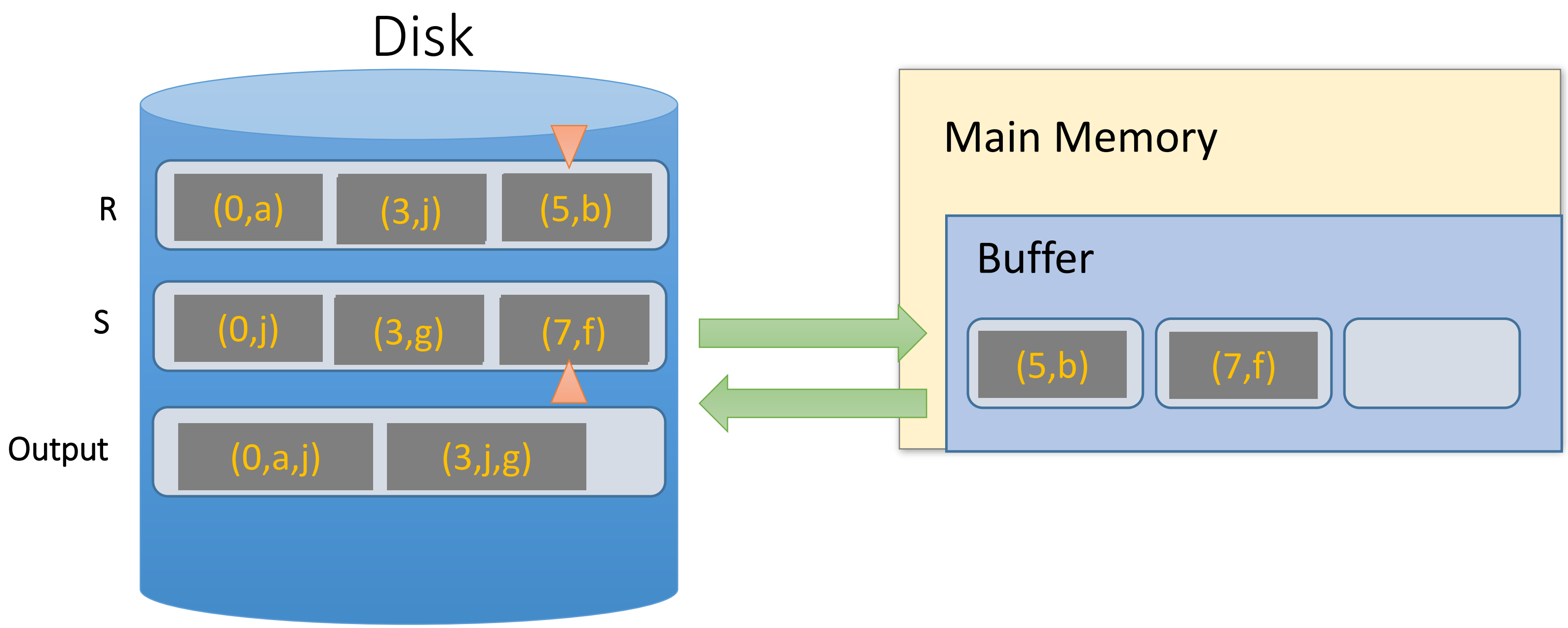
SMJ Example: $\mathrm{R} \bowtie S \ on \ A$ with 3-page buffer

2. Scan and “merge” on join key!



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

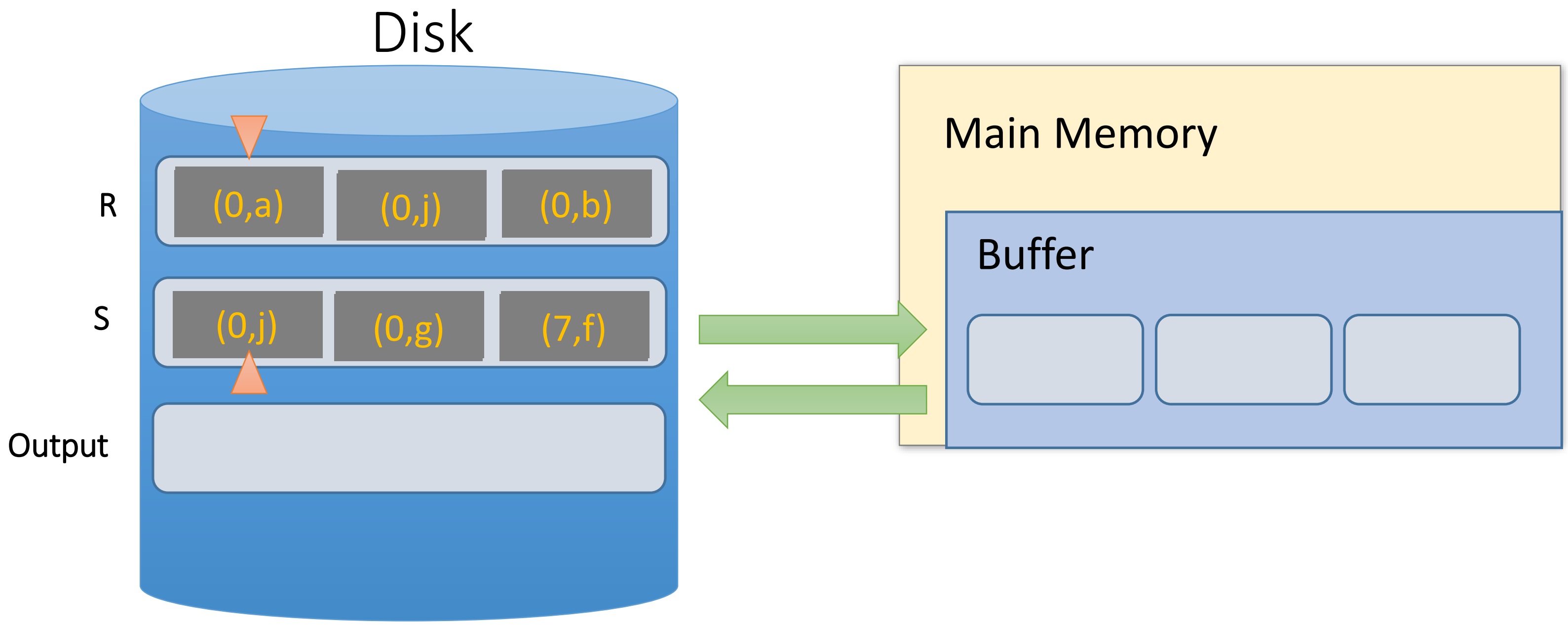
2. Done!



What happens if join keys have many duplicates?

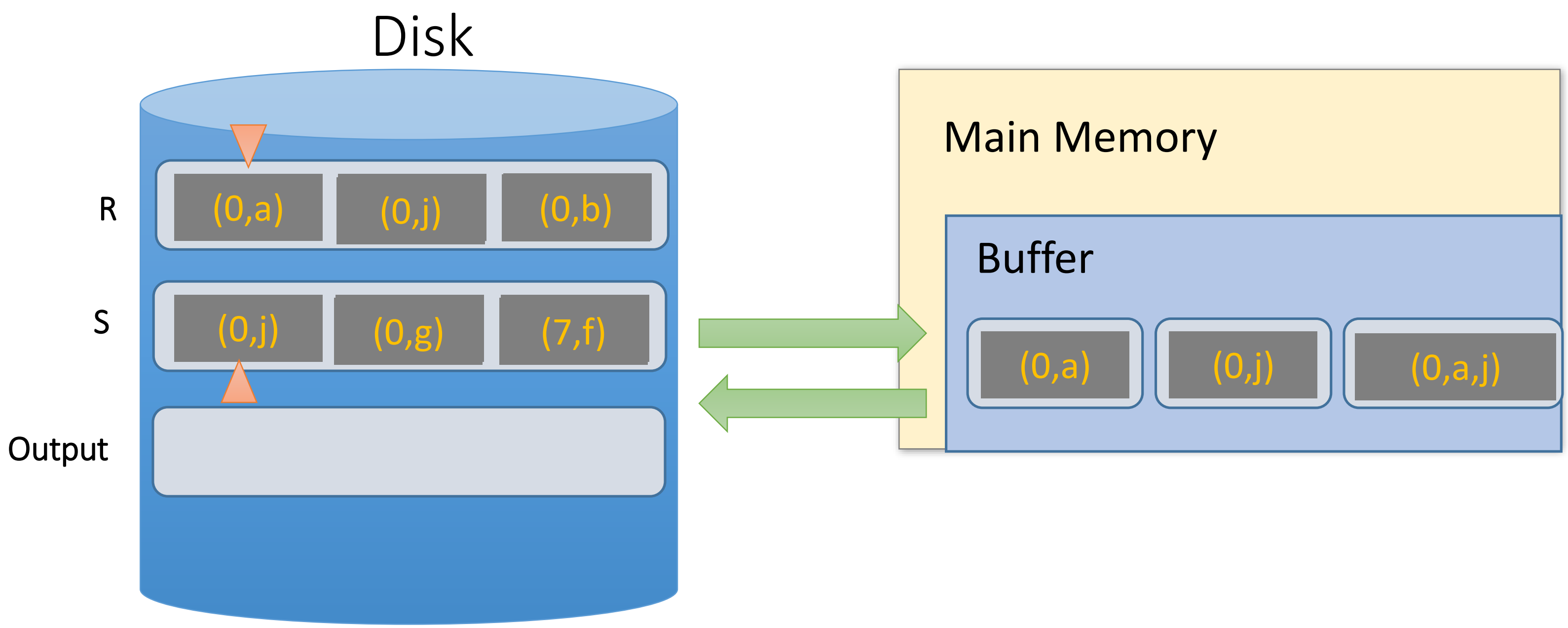
Multiple tuples with same join key: "backup"

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



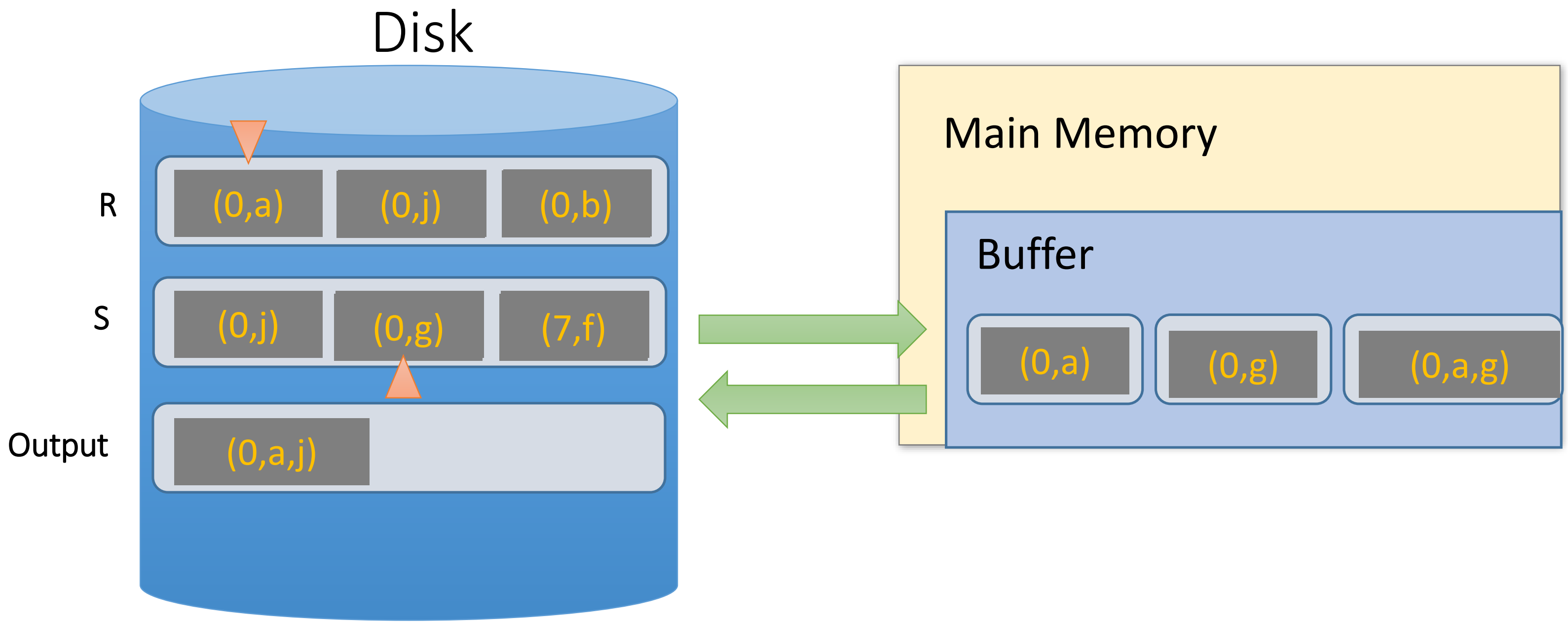
Multiple tuples with same join key: "backup"

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



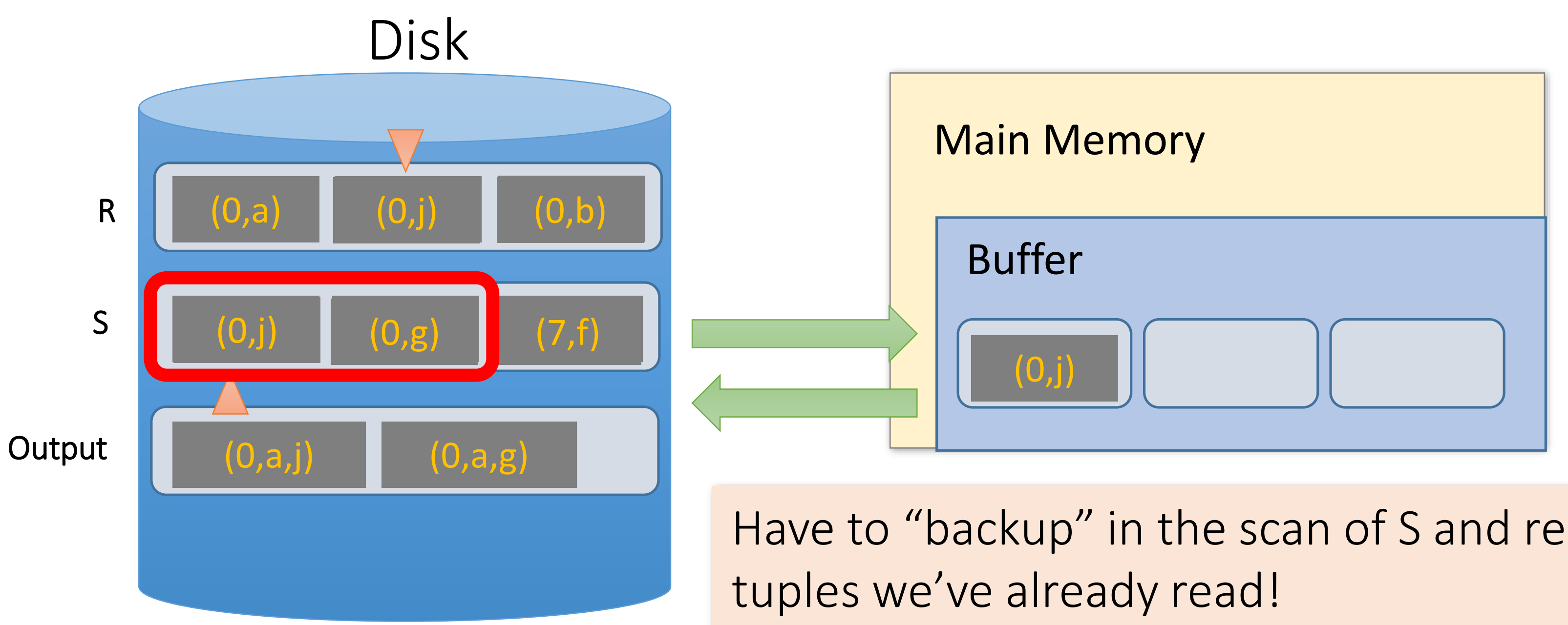
Multiple tuples with same join key: "backup"

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 tuples we’ve already read!

SMJ: cost of a final scan

- At best, no backup → final scan takes **B(R) + B(S)** reads
  - For ex.: if no duplicate values in join attribute

- At worst (e.g. full backup each time), scan could take **B(R) * B(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…
  - Not a very realistic scenario

SMJ: Total cost

- Cost of SMJ is **cost of sorting** R and S and writing temporary sorted files: **4B(R) + 4B(S)**

- Plus the **cost of scanning**: **~B(R)+B(S)**
  - Because of *backup*: in worst case B(R)*B(S); but this would be very unlikely

5B(R) + 5B(S)

SMJ cost: example

BNLJ:
$$\mathbf{B(R) + \frac{B(R)}{M-1} B(S)}$$

SMJ:
5*B(R) + 5*B(*S*)

- We have 101 buffer pages,
- B(R) = 1000, and B(S) = 500 pages:
- **SMJ**:
  - Sort both in two passes: 4* 1000 + 4* 500 = **6,000 IOs**
  - Merge-join phase 1000 + 500 = 1,500 IOs
  - **= 7,500 IOs**
- What with **BNLJ**?
  - 500 + 1000* $\left\lceil \frac{500}{100} \right\rceil$ = **5,500 IOs**
- But, if we have 26 buffer pages?
  - **SMJ** has same behavior (still 2 passes): **= 7,500 IOs**
  - **BNLJ**? ***25,500 IOs!***

SMJ is ~ linear vs. BNLJ is quadratic…

A simple optimization for SMJ: join during sort

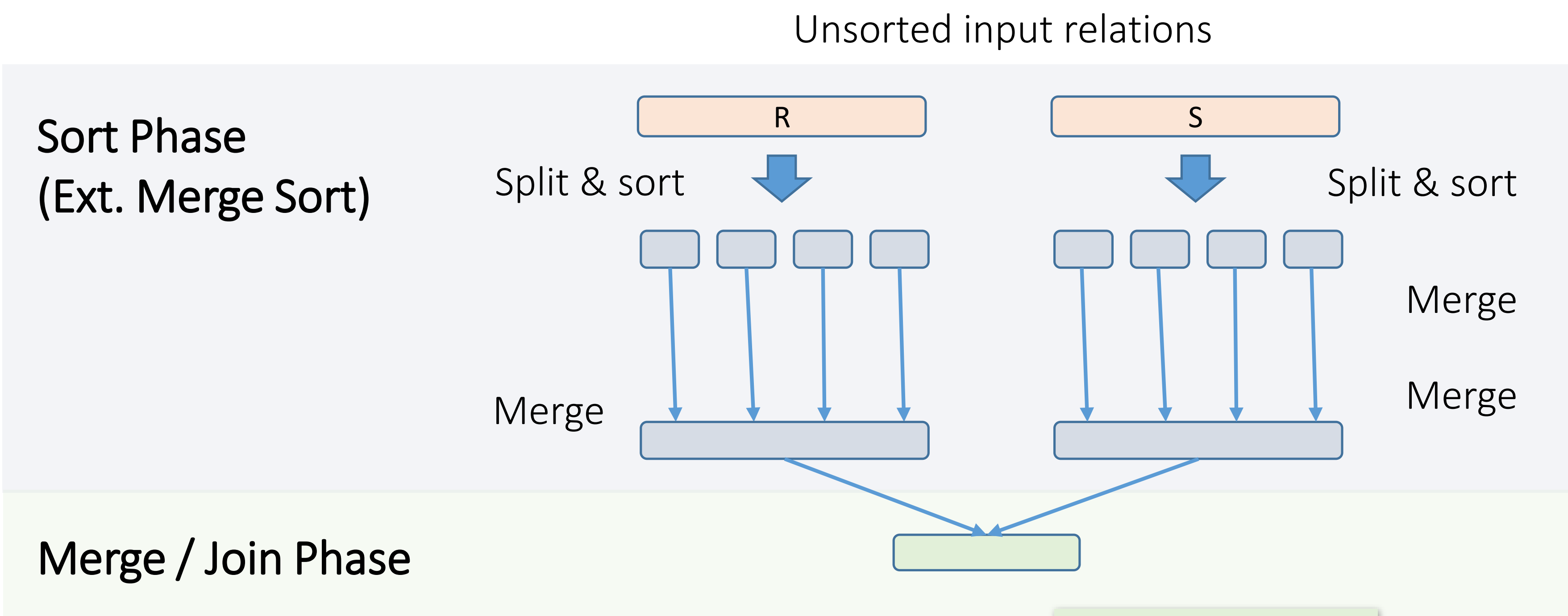
Given ***M*** buffer pages

- SMJ is composed of a 2PMMS ***sort*** and a ***join of sorted tuples***

- During the ***2PMMS***, if R and S have <= (***M - 1)*** (sorted) runs in total:
  - We could do two separate 2PMMS merges (for each of R & S) at this point, complete the sort phase, and start the join phase…
  - OR, we could combine them: do **one** (M - 1)-way merge simultaneously for R and S and complete the join!

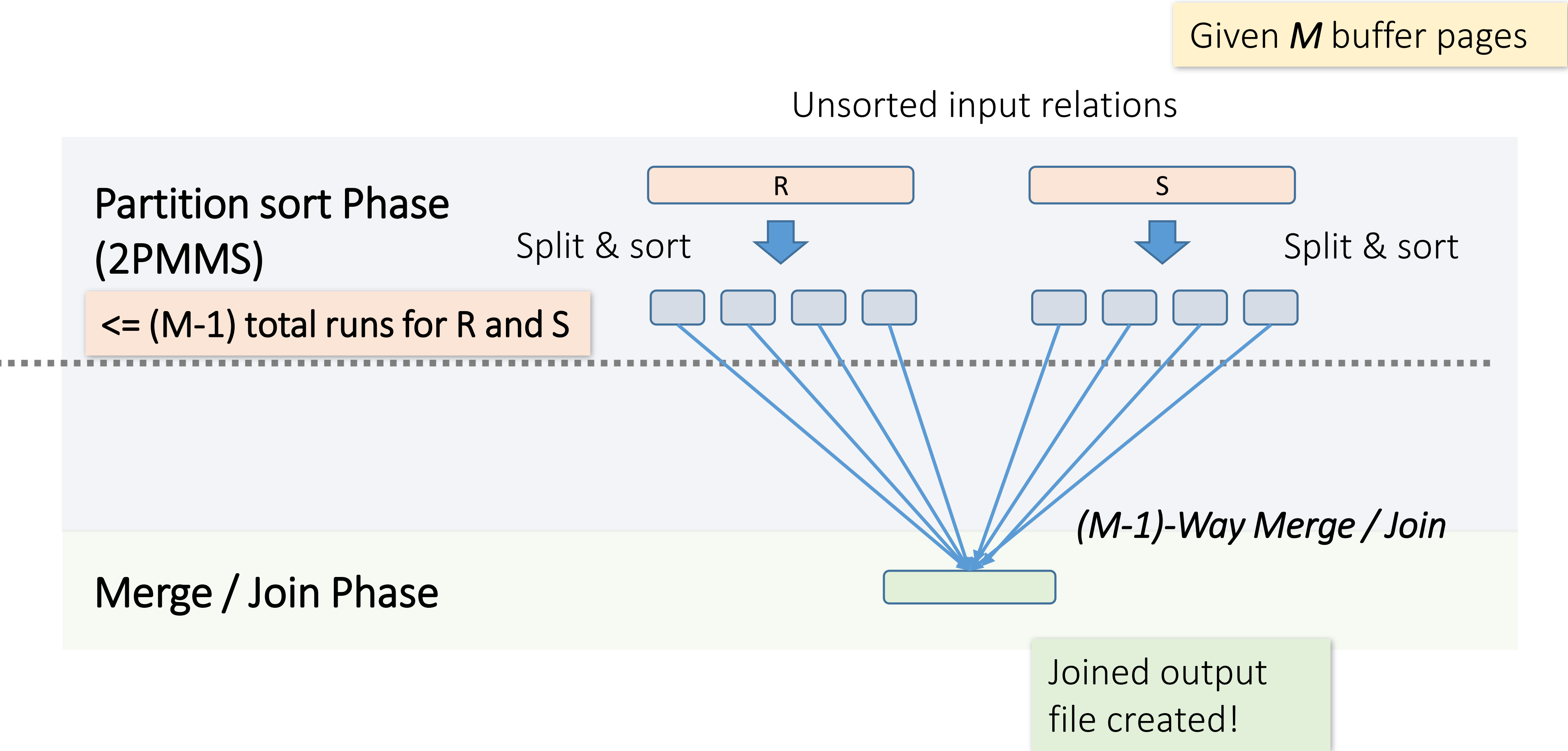
Un-Optimized SMJ

Given ***M*** buffer pages



Joined output file created!

Simple SMJ Optimization



Optimized SMJ: memory requirements

Given ***M*** buffer pages

- If we can initially split R and S into total M-1 runs, each run **of length <= M**, then we only need ***3(B(R) + B(S))*** for SMJ!
  - 2 Read/Write per page to sort runs in memory, 1 Read per page to (M-1)-way merge / join!
- How much memory for this to happen?
  - $\frac{B(R)+B(S)}{M-1} \leq M \Rightarrow \sim \mathrm{B(R)} + \mathrm{B(S)} \leq M^2$
- **Thus, $\mathbf{M} \geq \mathbf{sqrt}\ (\mathbf{B(R)} + \mathbf{B(S)})$ is an approximate sufficient condition for this algorithm**

If the sum of R,S has <= M^2 pages, then SMJ costs 3(B(R)+B(S))!

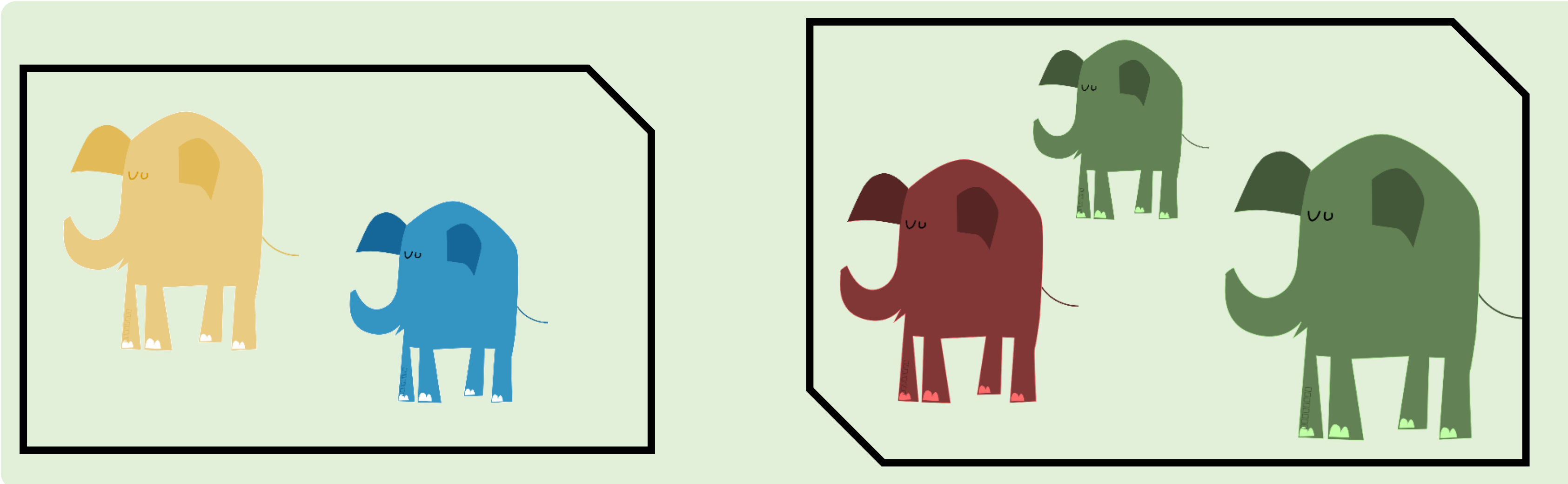
Takeaway points from SMJ

If input already sorted on join key, skip the sorts

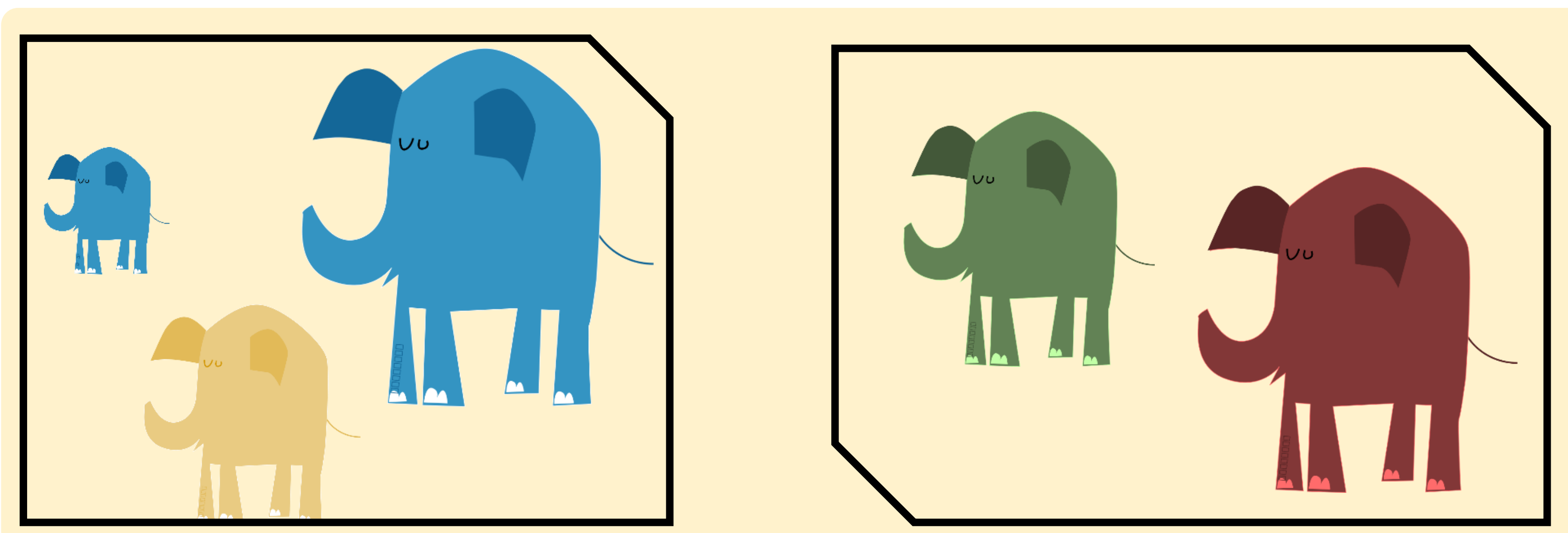
- SMJ is basically linear
- Nasty but unlikely case: too many duplicate join keys

SMJ needs to sort **both** relations

- If $B(R) + B(S) <= M^2$ then cost is ***3(B(R) + B(S))***



Join algorithm IV: Hash Join (HJ)



Recall: Hashing

- **Magic of hashing**:
  - A hash function h_M maps into [0,M-1]
  - And maps nearly uniformly

- A hash **collision** is when x != y but $h_M(x) = h_M(y)$
  - Note however that it will **never** occur that

  x = y but $h_M(x) \neq h_M(y)$

- We hash on attribute A, so our hash function $h_M(t)$ has the form $h_M(t.A)$.
  - **Collisions** may be more frequent, as we have much more tuples than the buckets

Hash Join: High-level

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

Note again that we are only considering equality join condition here

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

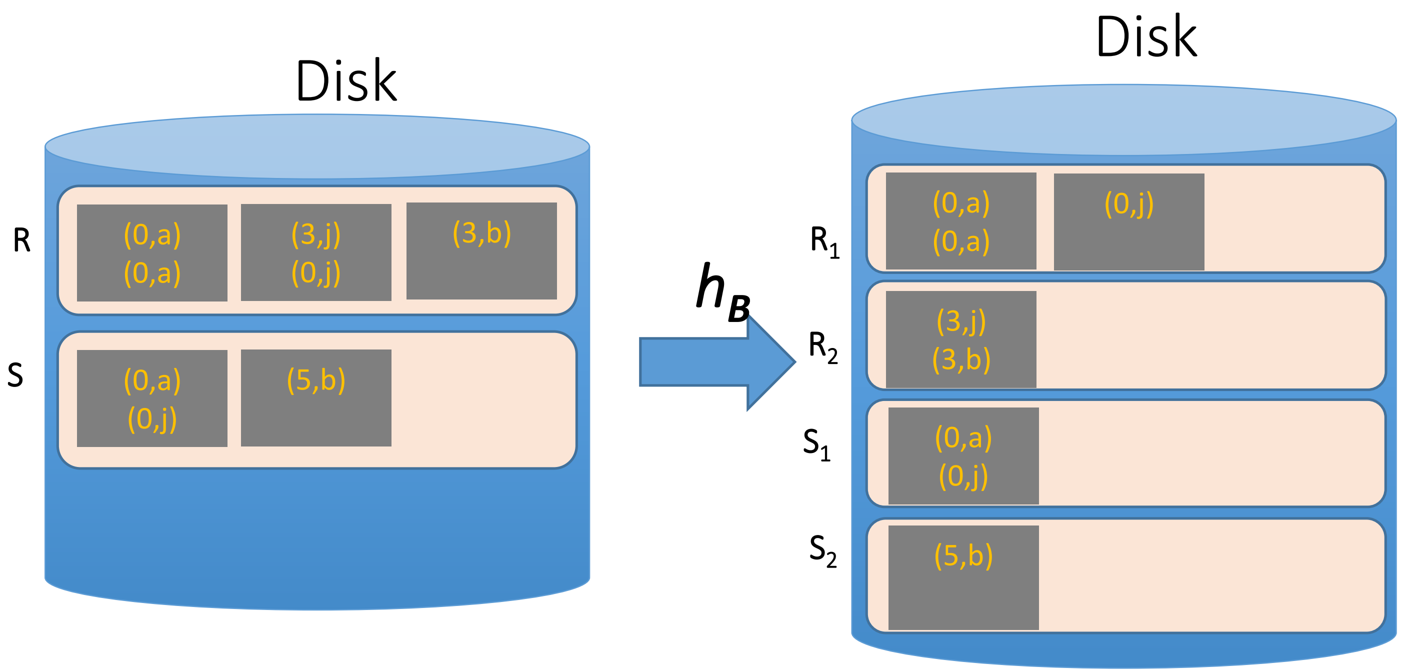
2. **Matching Phase:** Take pairs of buckets whose tuples have the same values for ***h***, and join these

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

HJ: high-level

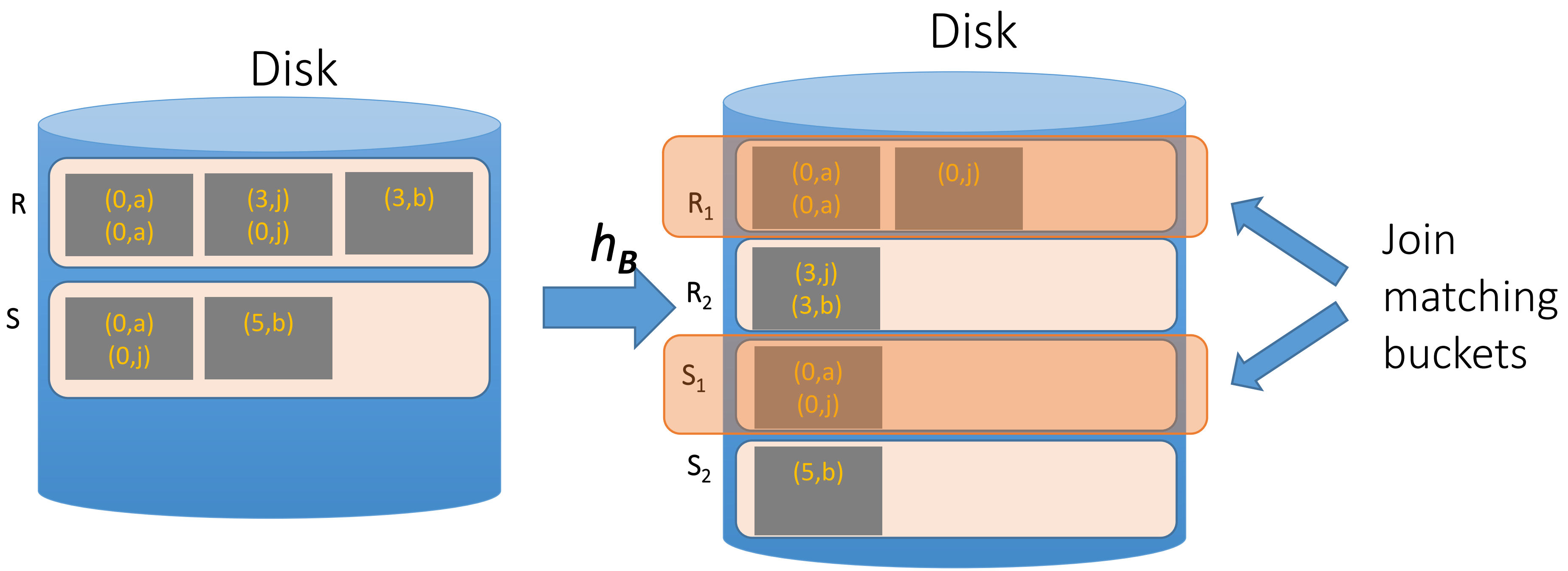
Note our new convention: pages each have two tuples (one per row)

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



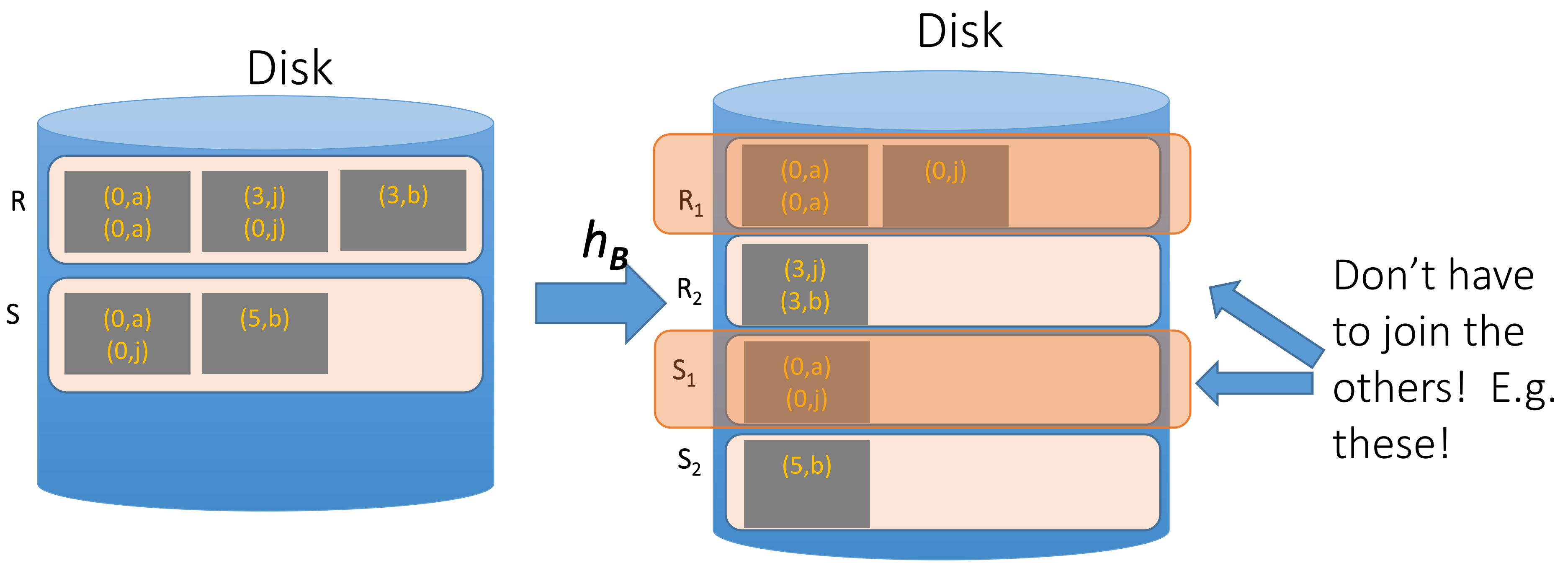
HJ: high-level

2. Matching Phase: Take pairs of buckets whose tuples have the same values for $\boldsymbol{h_M}$, and join these



HJ: high-level

2. Matching Phase: Take pairs of buckets whose tuples have the same values for $\boldsymbol{h_M}$, and join these



Hash Join phase 1: partitioning

Given ***M*** buffer pages

Goal: For each relation, partition relation into **buckets** such that if $h_M(t.A) = h_M(t'.A)$ they are in the same bucket

Given M buffer pages, we partition into M-1 buckets:

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

How big are the resulting buckets?

Given ***M*** buffer pages

- Given **B blocks of R, we partition into M-1 buckets**:
  - → Ideally our buckets are each of equal size **~ B/M pages**

- What happens if there are many **hash collisions**?
  - Some buckets could be > B/M
- What happens if there are multiple **duplicate join keys?**
  - Nothing we can do here… could have some **skew** in size of the buckets

How big at most *do we want* the resulting buckets?

Given $\boldsymbol{M}$ buffer pages

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

- Recall: If we want to join a bucket R_i from R and one from S, we can do BNLJ **in linear time** if for *one of them (say R_i),* $\boldsymbol{B(Ri) \leq M-1}$!

Recall for BNLJ:

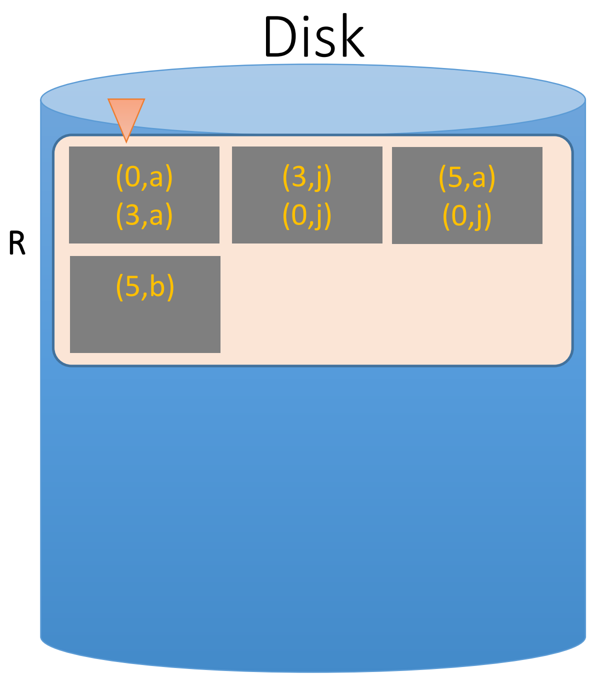
$$B(R) + \underbrace{\frac{B(R)B(S)}{M-1}}_{=1}$$

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

Hash Join Phase 1: Example

Given *M = 3* buffer pages

We partition into ***M-1 = 2*** buckets **using hash function h_2** so that we can have one buffer page for each partition (and one for input)



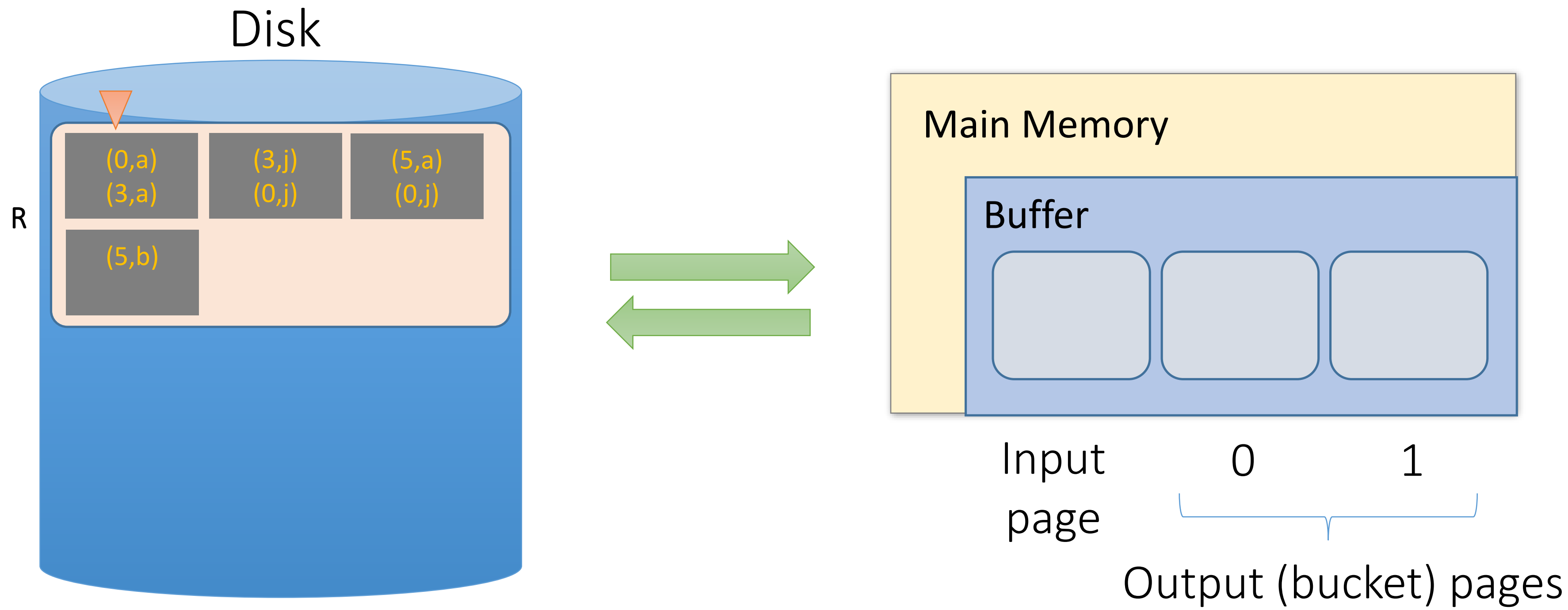
For simplicity, we'll look at partitioning one of the two relations - we just do the same for the other relation!

Recall: our goal will be to get *M - 1 = 2 buckets* of size <= *M -1* → *2 pages each*

Hash Join Phase 1: Example

Given *M = 3* buffer pages

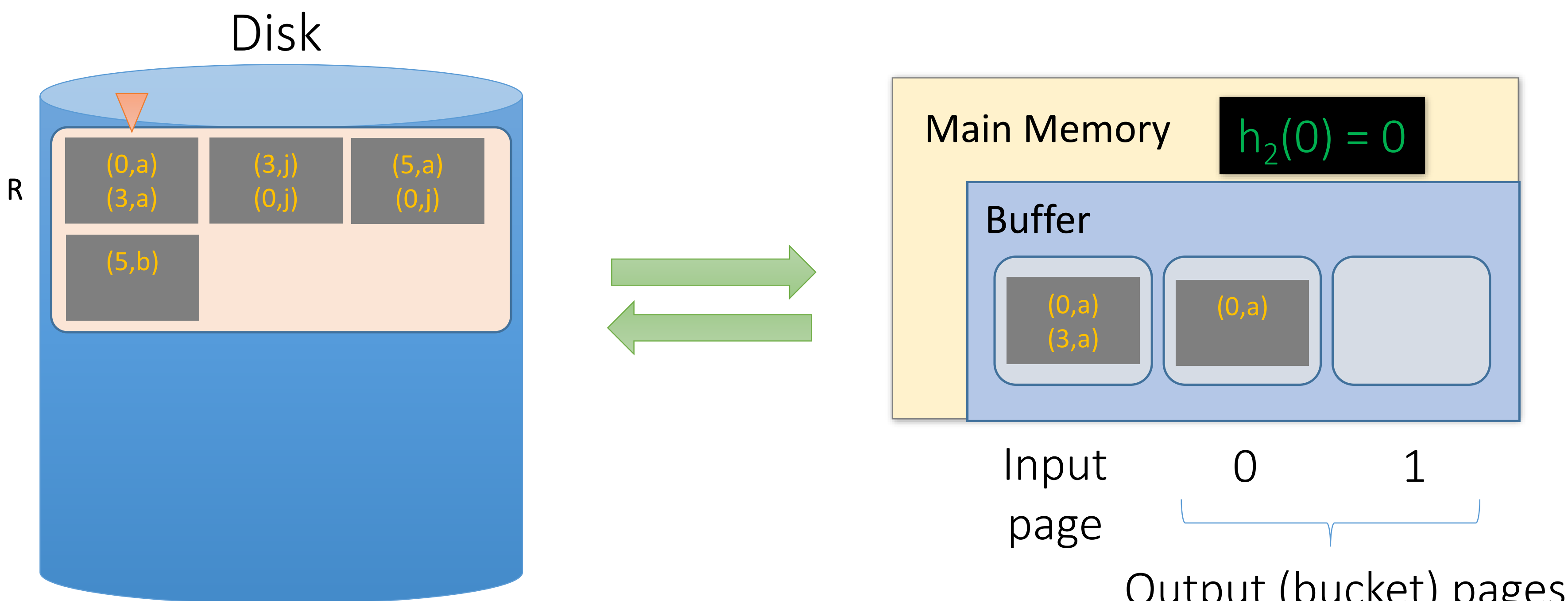
1. We read pages from R into the “input” page of the buffer…



Hash Join Phase 1: Example

Given ***M = 3*** buffer pages

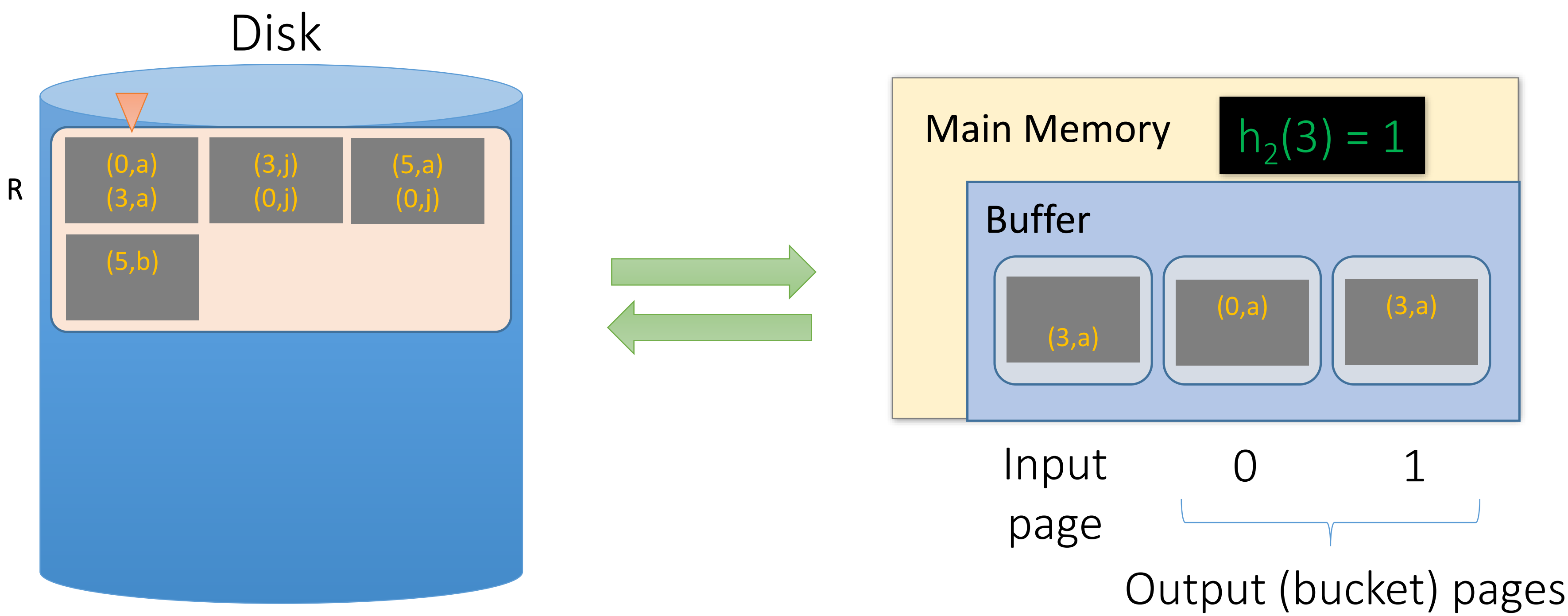
2. Then we use **hash function h_2** to find the output bucket, which each has one page in the buffer



Hash Join Phase 1: Example

Given *M = 3* buffer pages

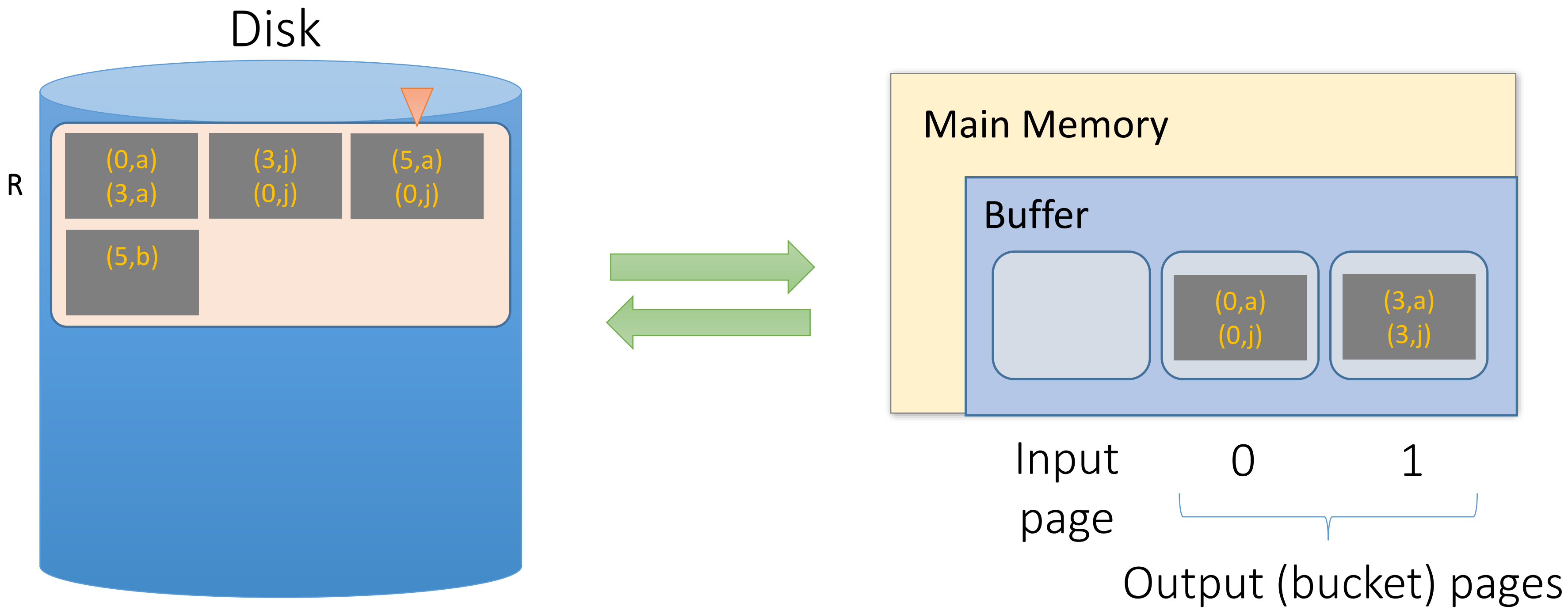
2. Then we use **hash function h_2** to find the output bucket, which each has one page in the buffer



Hash Join Phase 1: Example

Given $M = 3$ buffer pages

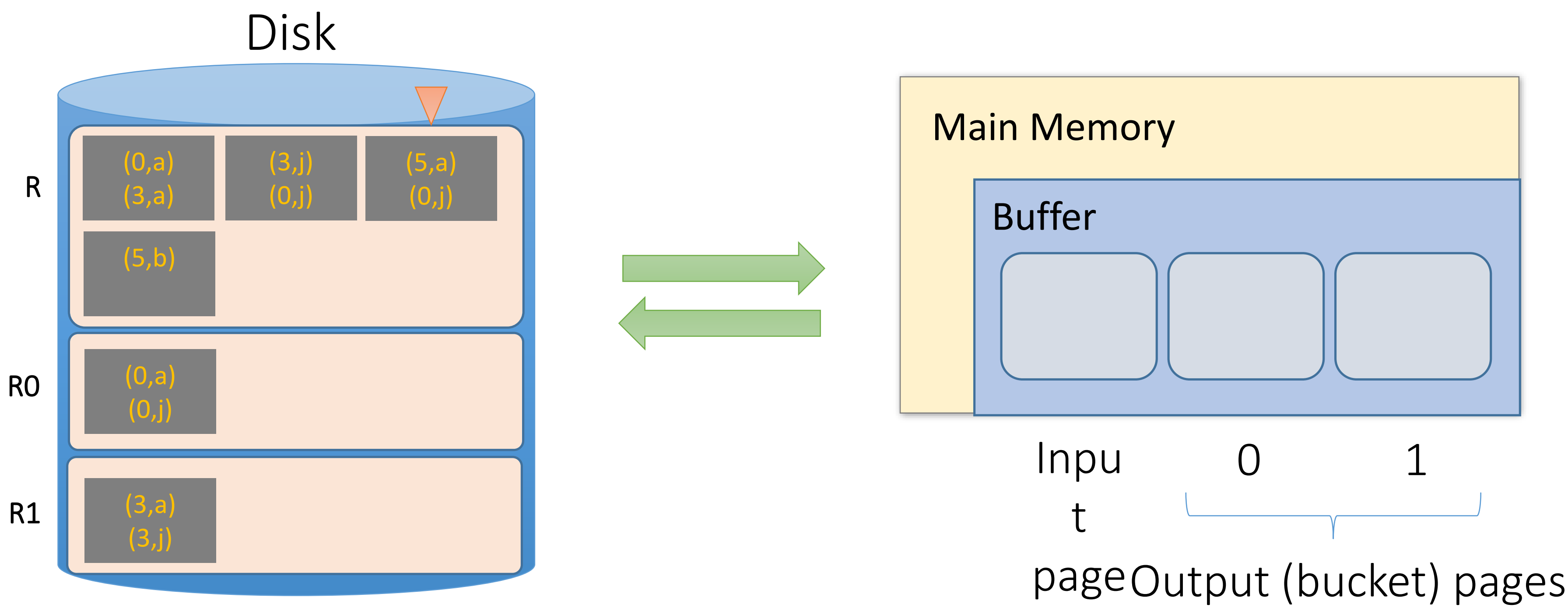
3. We repeat until the buffer bucket pages are full…



Hash Join Phase 1: Example

Given $M = 3$ buffer pages

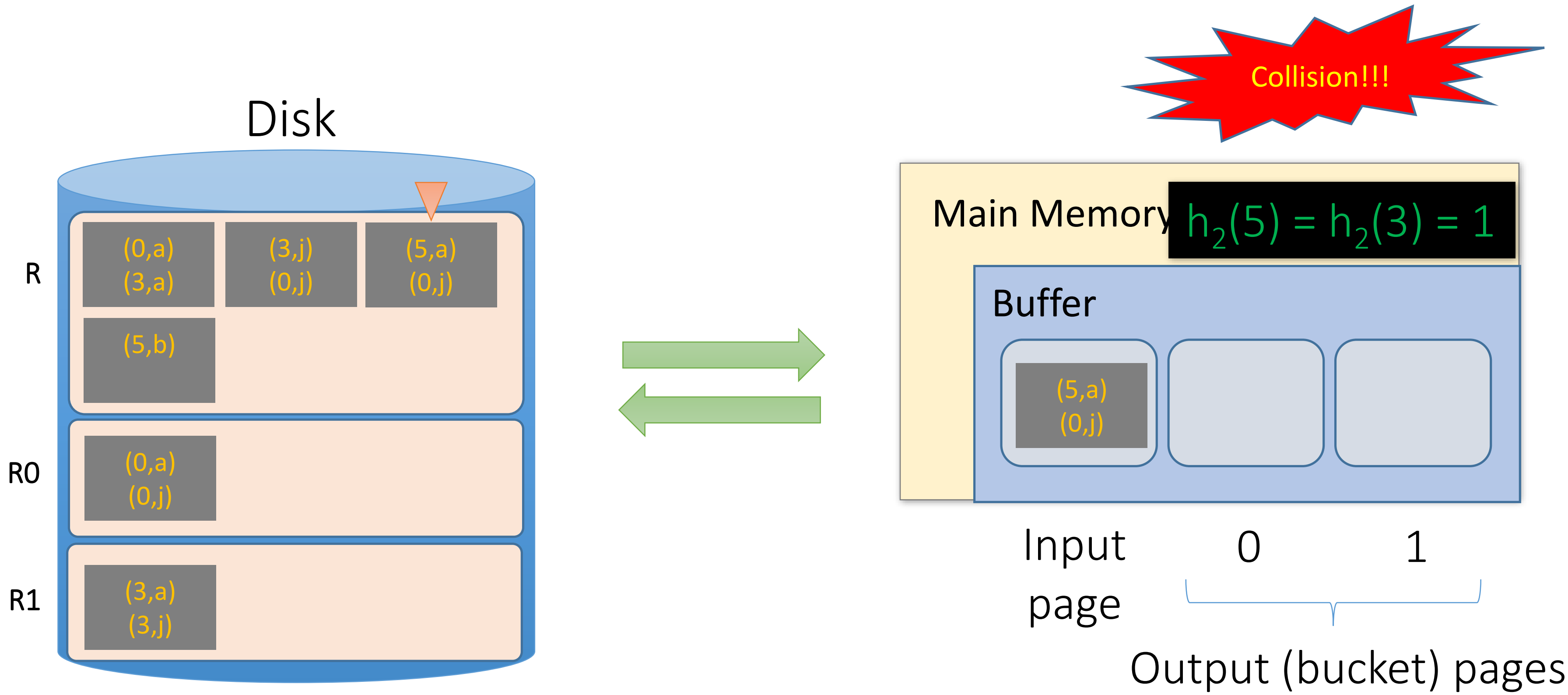
3. We repeat until the buffer bucket pages are full… then flush to disk



Hash Join Phase 1: Example

Given *M = 3* buffer pages

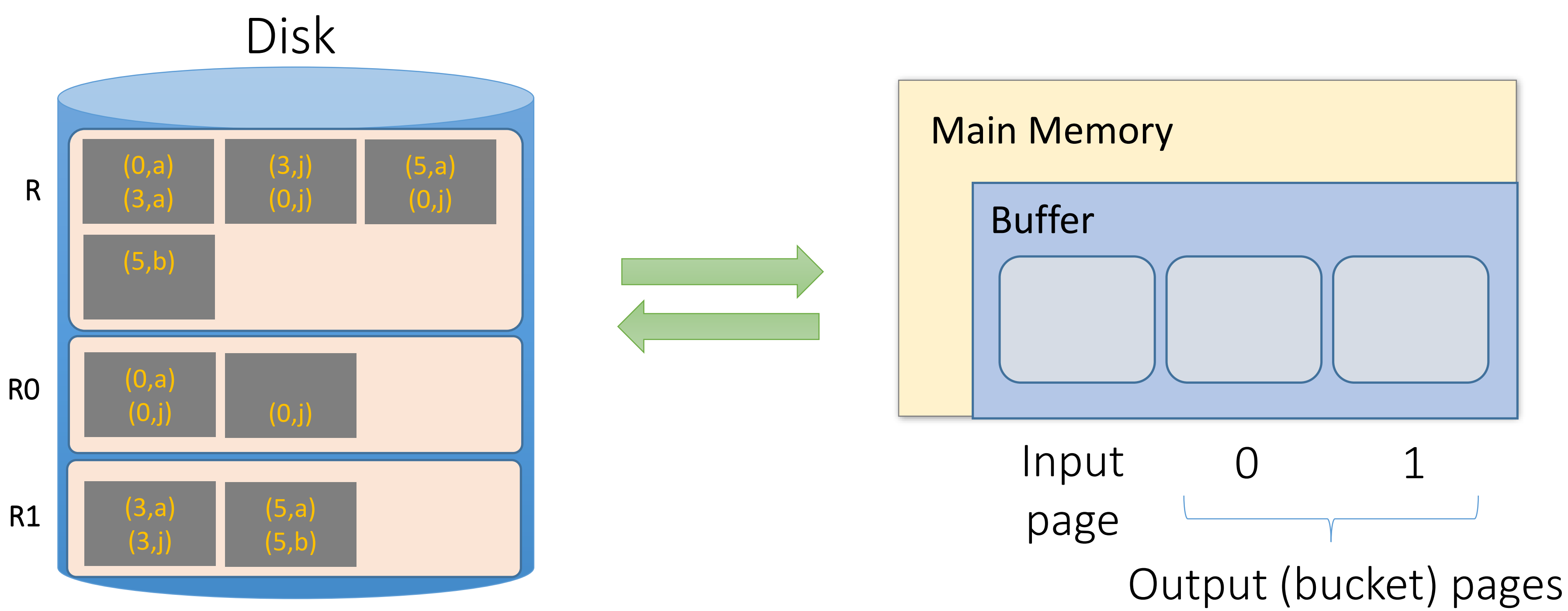
Note that collisions can occur!



Hash Join Phase 1: Example

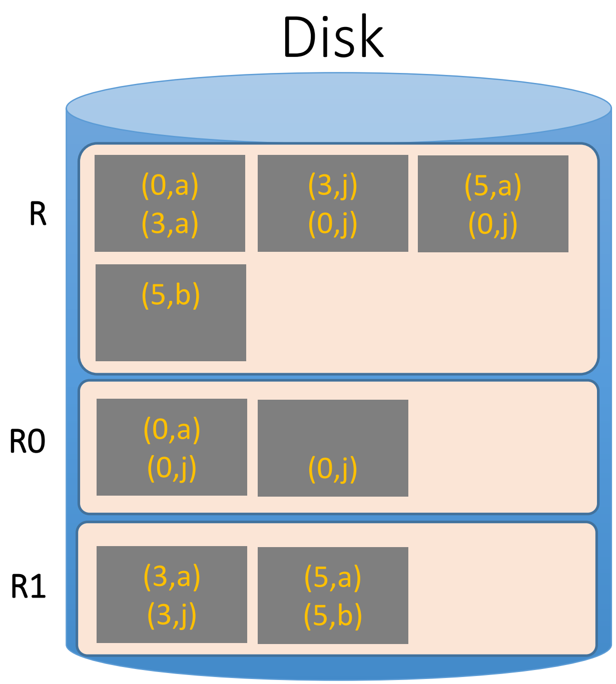
Given $M = 3$ buffer pages

Finished phase I for R



Hash Join Phase 1: complete

Given *M = 3* buffer pages



We wanted buckets of size ***M-1 = 2…***
Some of them could be larger due to:

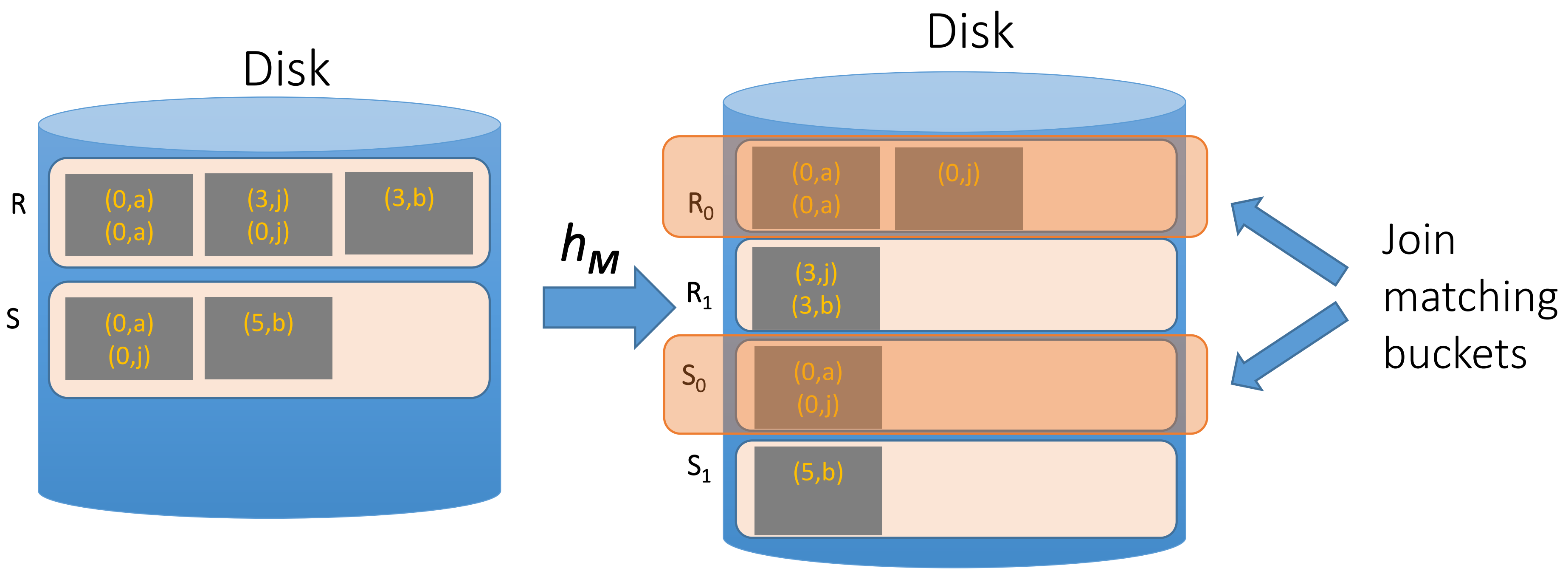
(1) Duplicate join keys

(2) Hash collisions

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!



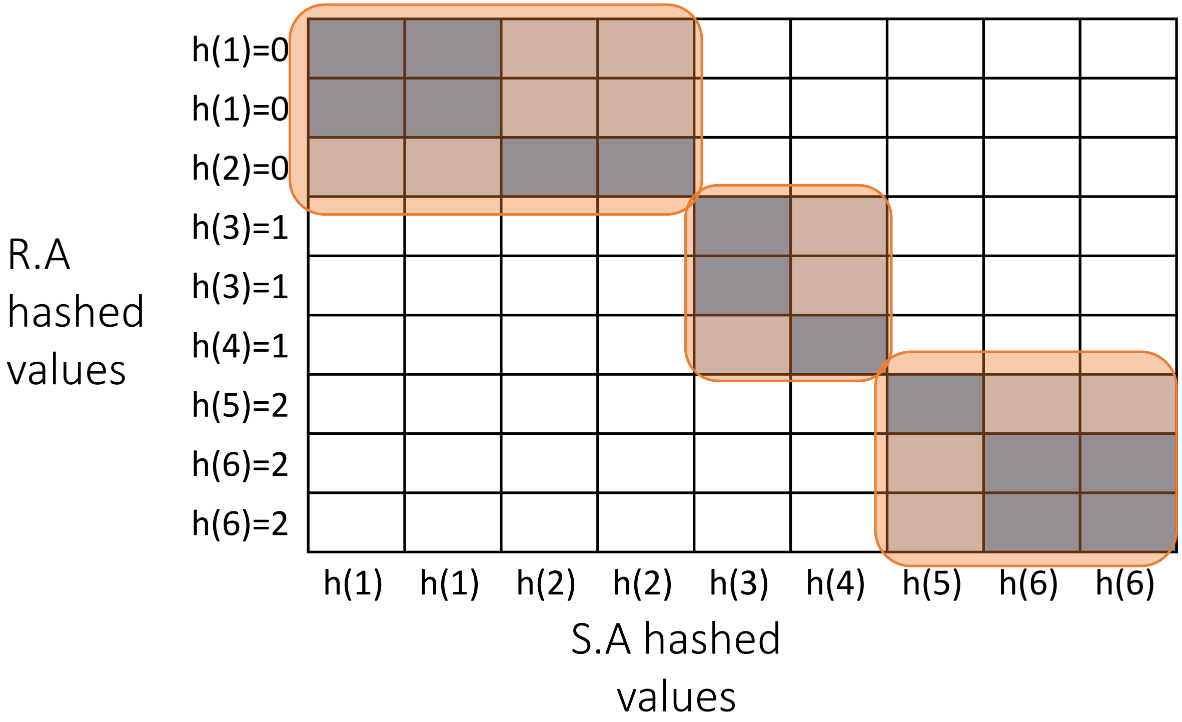
Hash Join Phase 2: Matching

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

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

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

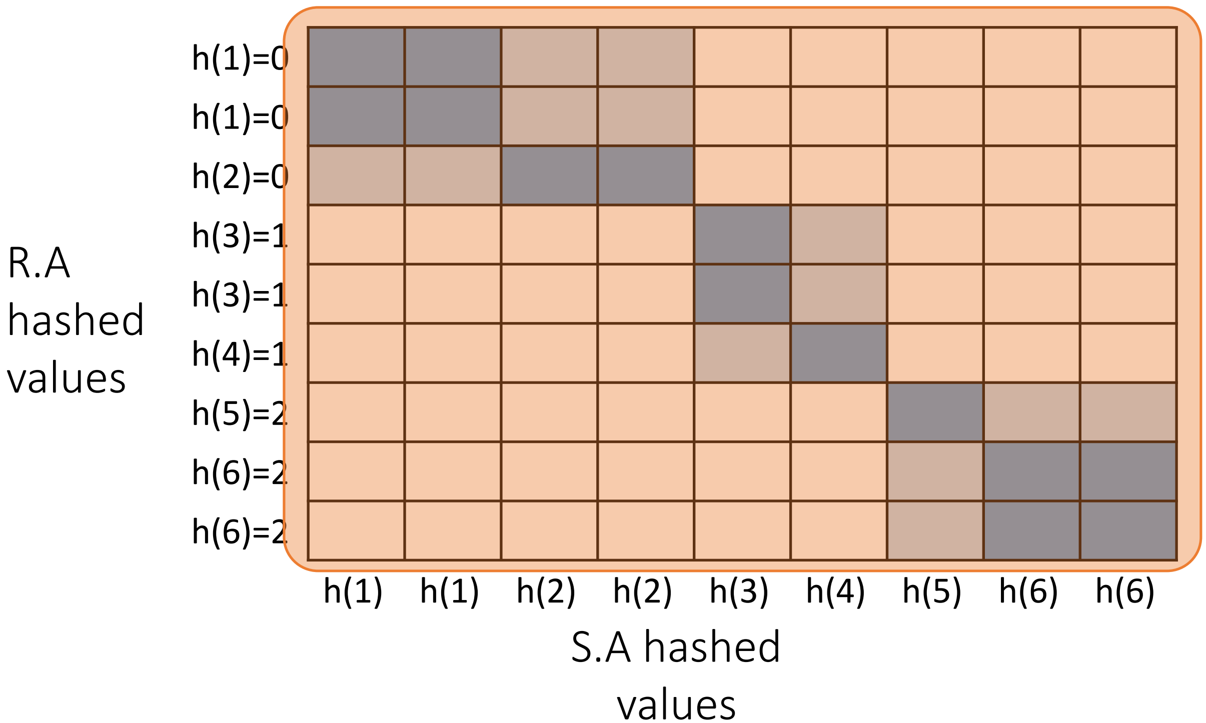
Hash Join Phase 2: Matching



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

If condition is an equality – we explore only matching buckets – diagonal

Hash Join Phase 2: Matching



$R \bowtie S \ on \ A$

If it is not an equijoin, we explore this ***whole grid!***

Hash Join: memory requirements

- Given M buffer pages

Assume B(R) <= B(S)

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

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

Quadratic relationship between ***smaller relation's*** size & memory!

Hash Join: cost

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

HJ takes ~3(B(R)+B(S)) !

Sort-Merge vs. Hash Join

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

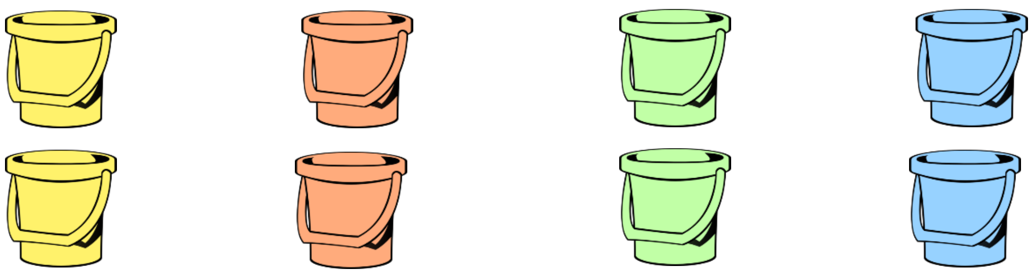
$$\sim 3(B(R)+B(S))$$

- ***"Enough" memory =***
  - SMJ: $M^2 > B(R) + B(S)$
  - HJ: $M^2 > \min\{B(R), B(S)\}$

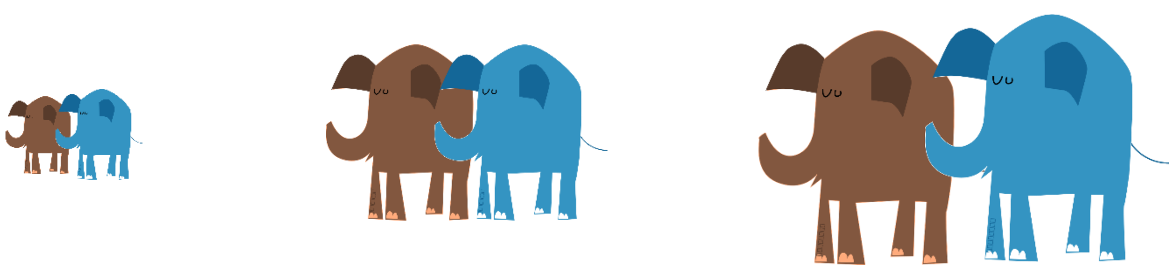
Hash Join superior if relation sizes ***differ greatly***. Why?

Further Comparison of Hash vs. 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 are the key in hash versus sort join
  - Hash Join = Little dog (depends on smaller relation)
- Skew is also a major factor

Impact of Buffering

- If several operations are executing concurrently, estimating the number of available buffer pages is guesswork
- Repeated access patterns interact with buffer replacement policy
  - e.g., Inner relation is scanned repeatedly in Simple Nested Loop Join. With enough buffer pages to hold inner, replacement policy does not matter. Otherwise, MRU is best, LRU is worst (*sequential flooding*).
  - Does replacement policy matter for Block Nested Loops?
  - What about Index Nested Loops? Sort-Merge Join?