CSC 261/461 – Database Systems
Lecture 20

Fall 2017

Announcements

- Project 3 (MongoDB) is out
  - Due on Dec 01
- Term paper is due on:
  - Dec 08, 2017
  - (You need to finish your poster before that to have ample time for getting it printed)
  - Details will follow…

Topics for Today

- MongoDb
- Query Processing (Chapter 18)
- Query Optimization (Chapter 19)

MONGODB

What is MongoDB

- Scalable High-Performance Open-source, Document-orientated database.
- Built for Speed
- Rich Document based queries for Easy readability.
- Full Index Support for High Performance.
- Map / Reduce for Aggregation.

Why use MongoDB?

- SQL was invented in the 70's to store data.
- MongoDB stores documents (or) objects
- Embedded documents and arrays reduce need for joins

Why will we use Mongodb?

- Semi-Structured Content Management

XML -> Tables

- Items -> User, Item, Category, Bid

Object-relational impedance mismatch

- A set of conceptual and technical difficulties that are often encountered:
  - when a relational database management system (RDBMS) is being served by an application program (or multiple application programs) written in an object-oriented programming language
- Objects or class definitions must be mapped to database tables defined by relational schema.

MongoDB: No Impedance Mismatch

```
// your application code
class Foo { int x; string [] tags;}

// mongo document for Foo
{ x: 1, tags: ['abc','xyz'] }
```

When I say

Database

Think

Database

- Made up of Multiple Collections.
- Created on-the-fly when referenced for the first time.

When I say **Think**

Collection **Table**

- Schema-less, and contains Documents.
- Indexable by one/more keys.
- Created on-the-fly when referenced for the first time.
- Capped Collections: Fixed size, older records get dropped after reaching the limit.

When I say **Think**

Document **Record/Row**

- Stored in a Collection.
- Have _id key – works like Primary keys in MySQL.
- Supported Relationships – Embedded (or) References.
- Document storage in BSON (Binary form of JSON).

The Document Model

```
var post = {
        '_id': ObjectId('3432'),
        'author': ObjectId('2311'),
        'title': 'Introduction to MongoDB',
        'body': 'MongoDB is an open sources.. ',
        'timestamp': Date('01-04-12'),
        'tags': ['MongoDB', 'NoSQL'],
        'comments': [{'author': ObjectId('5331'),
                      'date': Date('02-04-12'),
                      'text': 'Did you see.. ',
                      'upvotes': 7} ]
}

> db.posts.insert(post);
```

Find

```
// find posts which has 'MongoDB' tag.
> db.posts.find({tags: 'MongoDB'});

// find posts by author's comments.
> db.posts.find({'comments.author': 'Johnson'}).count();

// find posts written after 31st March.
> db.posts.find({'timestamp': {'$gte': Date('31-03-12')}});
```

$gt, $lt, $gte, $lte, $ne, $all, $in, $nin…

Find

Which fields?

```
db.foo.find(query, projection)
```

Which documents?

Find: Projection

```
> db.posts.find({}, {title:1})

  { "_id" : ObjectId("5654381f37f63ffc4ebf1964"),
          "title" : "NodeJS server" }
  { "_id" : ObjectId("5654385c37f63ffc4ebf1965"),
          "title" : "Introduction to MongoDB" }
```

Like

select **title** from **posts**

Empty projection like

select * from **posts**

Find

Find
- Query criteria
  - Single value field
  - Array field
  - Sub-document / dot notation

Projection
- Field inclusion and exclusion

Cursor
- Sort
- Limit
- Skip

Update

```
> db.posts.update(
	{"_id" : ObjectId("5654381f37f63ffc4ebf1964")},
	{
		title:"NodeJS server"
	});
```

This will **replace** the document by {title:"NodeJS server"}

Update: Change part of the document

```
> db.posts.update(
        {"_id" : ObjectId("5654381f37f63ffc4ebf1964")},
        {
                $addToSet: {tags:"JS"},
                $set: {title:"NodeJS server"},
                $unset: { comments: 1}
        });
```

$set, $unset

$push, $pull, $pop, $addToSet

$inc, $decr, many more…

Update

Which Document?

```
db.foo.update(query,update,options);
```

Collection Name

What Change?

One? Many? Upsert?

Options:

{multi: true} – will change all found documents; by default only first found will be updated

{upsert: true} – will insert document if it was not found

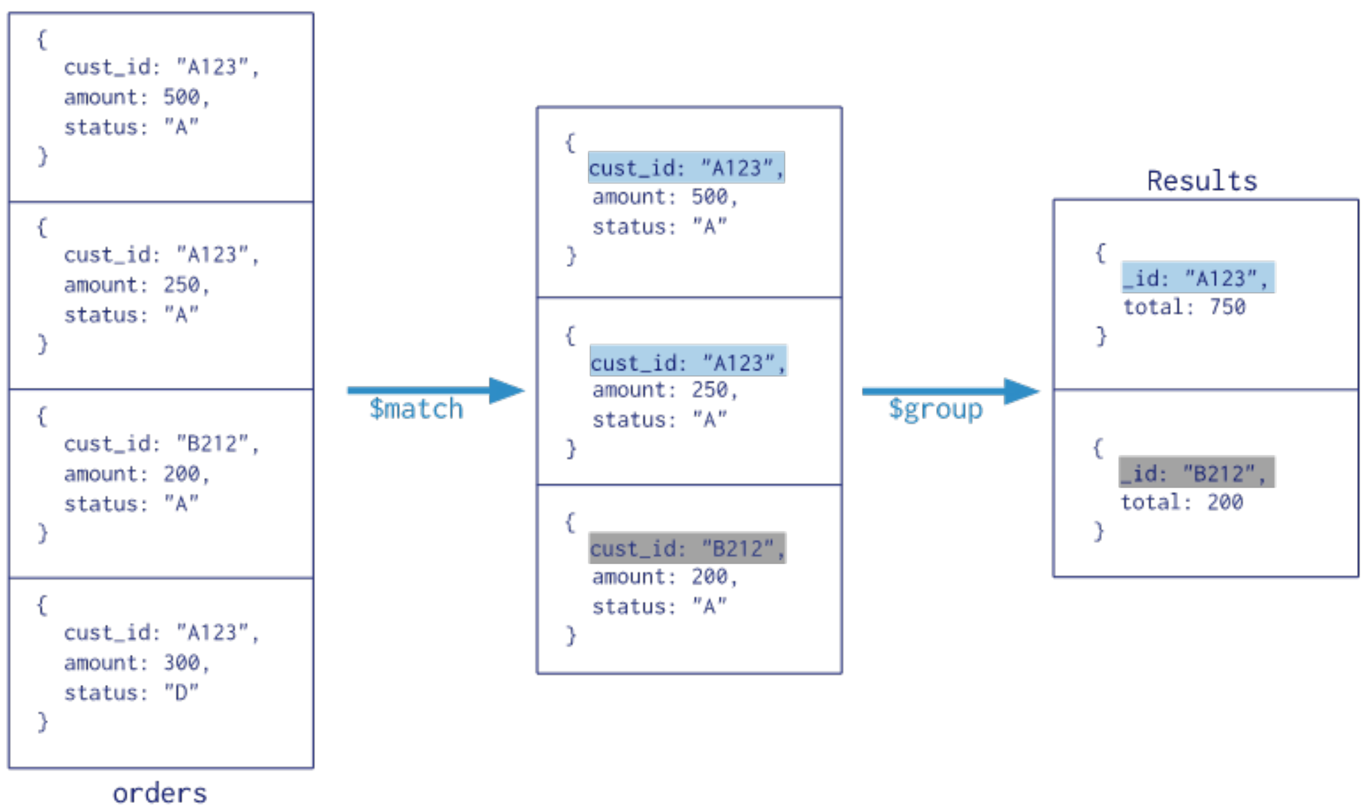
Remove

- db.collection.remove(<query>, <justOne>)

- db.items.remove({Currently: { $gt: 20 } })

Aggregation

```
Collection
db.orders.aggregate( [
    $match stage ───►  { $match: { status: "A" } },
    $group stage ───►  { $group: { _id: "$cust_id",total: { $sum: "$amount" } } }
                   ] )
```

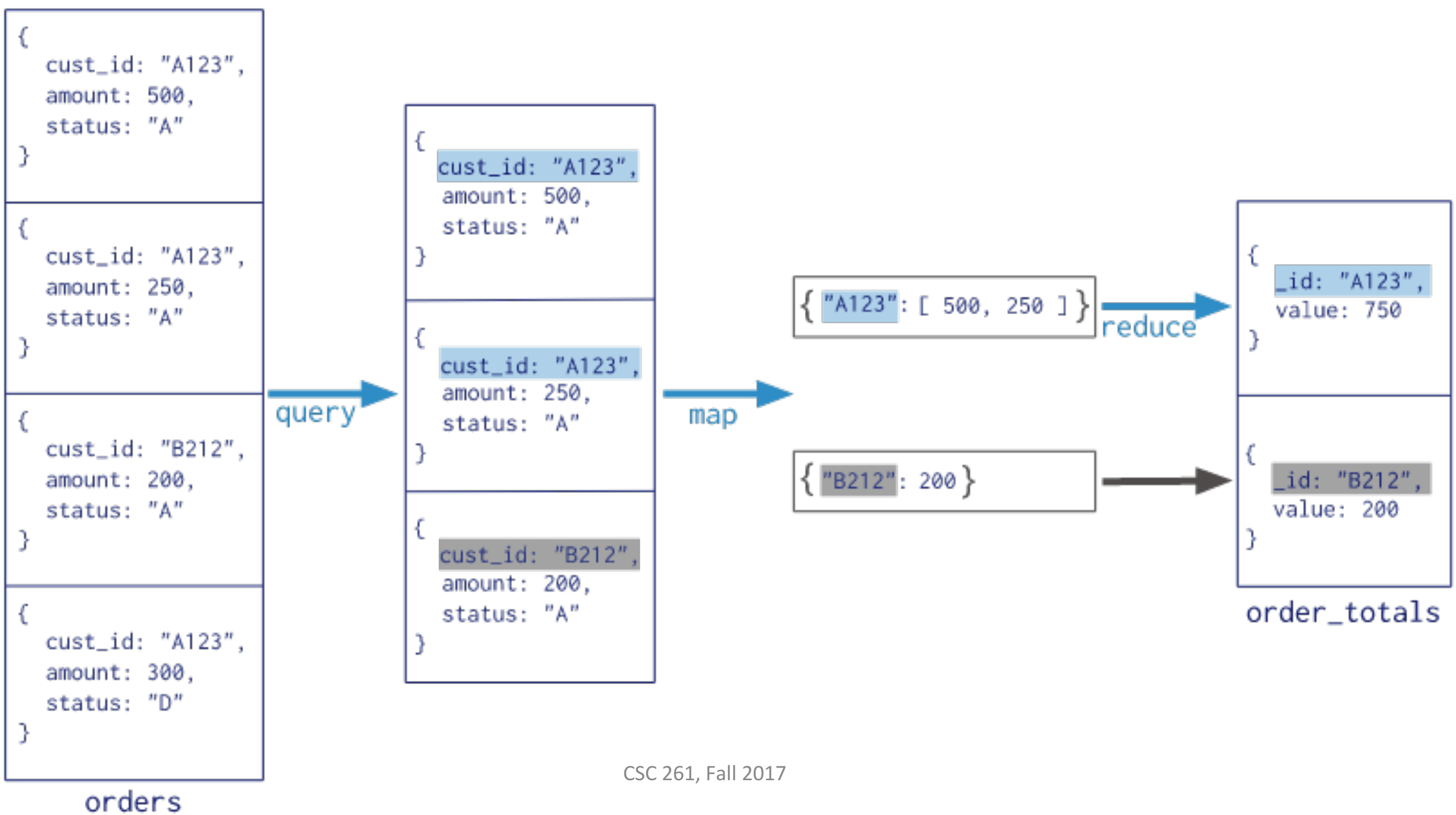


Aggregation

- https://docs.mongodb.com/v3.0/applications/aggregation/
- https://www.safaribooksonline.com/blog/2013/06/21/aggregation-in-mongodb/

MapReduce

```
Collection
    ↓
db.orders.mapReduce(
        map     →  function() { emit( this.cust_id, this.amount ); },
        reduce  →  function(key, values) { return Array.sum( values ) },
                   {
        query   →    query: { status: "A" },
        output  →    out: "order_totals"
                   }
                )
```



Acknowledgement

- Many of these slides are produced by Luxoft.com

QUERY PROCESSING

Steps in Query Processing

- Scanning
- Parsing
- Validation
- Query Tree Creation
- Query Optimization (Query planning)
- Code generation (to execute the plan)
- Running the query code

Nested Loop Joins

Notes

- We are again considering "IO aware" algorithms: ***care about disk IO***
- Given a relation R, let:
  - $T(R)$ = # of tuples in R
  - $P(R)$ = # of pages in R
- Note also that we omit ceilings in calculations… good exercise to put back in!

Recall that we read / write entire pages with disk IO

Nested Loop Join (NLJ)

```
Compute R ⋈ 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 ⋈ 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 ⋈ 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**

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

Nested Loop Join (NLJ)

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

<u>Cost:</u>

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

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

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

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

Block Nested Loop Join (BNLJ)

Block Nested Loop Join (BNLJ)

Given ***3*** pages of memory

Cost:

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

```
for each page pr of R:
  for page ps of S:
    for each tuple r in pr:
      for each tuple s in ps:
        if r[A] == s[A]:
          yield (r,s)
```

$\mathrm{P}(R)$

1. **Load in 1 page of R at a time (leaving 1 page each free for S & output)**

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

Block Nested Loop Join (BNLJ)

Given ***3*** pages of memory

<u>Cost:</u>

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

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

```
for each page 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 1 page of R at a time (leaving 1 page each free for S & output)
2. **For each page segment of R, load each page of S**

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

Block Nested Loop Join (BNLJ)

Given ***3*** pages of memory

<u>Cost:</u>

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

```
Compute R ⋈ S on A:
  for each page 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 1 page of R at a time (leaving 1 page each free for S & output)
2. For each 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 3 pages of memory

Cost:

$$P(R) + P(R).P(S)$$

```
Compute R ⋈ S on A:
  for each page 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 1 page of R at a time (leaving 1 page each free for S & output)
2. For each page segment of R, load each page of S
3. Check against the join conditions
4. **Write out**

Block Nested Loop Join (BNLJ) (B+1 pages of Memory)

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

Cost:

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

$P(R)$

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 ⋈ 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:

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

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

2. **For each (B-1)-page segment of R, load each page of S**

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

Block Nested Loop Join (BNLJ)

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

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

Cost:

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

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

BNLJ can also handle non-equality constraints

Block Nested Loop Join (BNLJ)

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

Cost:

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

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

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)*P(S) + OUT

BNLJ

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

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

BNLJ vs. NLJ: Benefits of IO Aware

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

Ignoring OUT here…

- NLJ: Cost = 500 + **50,000*1000** = **50 Million IOs** ~= **<u>140 hours</u>**

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

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

Smarter than Cross-Products

Smarter than Cross-Products: From Quadratic to Nearly Linear

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

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

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

- Now we'll see some (nearly) linear joins:
  - ~ O(P(R) + P(S) + ***OUT***), where again ***OUT*** could be quadratic but is usually better

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

Index Nested Loop Join (INLJ)

```
Compute R ⋈ S on A:
  Given index idx on S.A:
    for r in R:
      s in idx(r[A]):
        yield r,s
```

Cost:

P(R) + T(R)**L** + OUT

where ***L*** is the IO cost to access all the distinct values in the index; assuming these fit on one page, $\mathrm{L} \sim 3$ is good est.

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

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

Note that we are only considering equality join conditions here

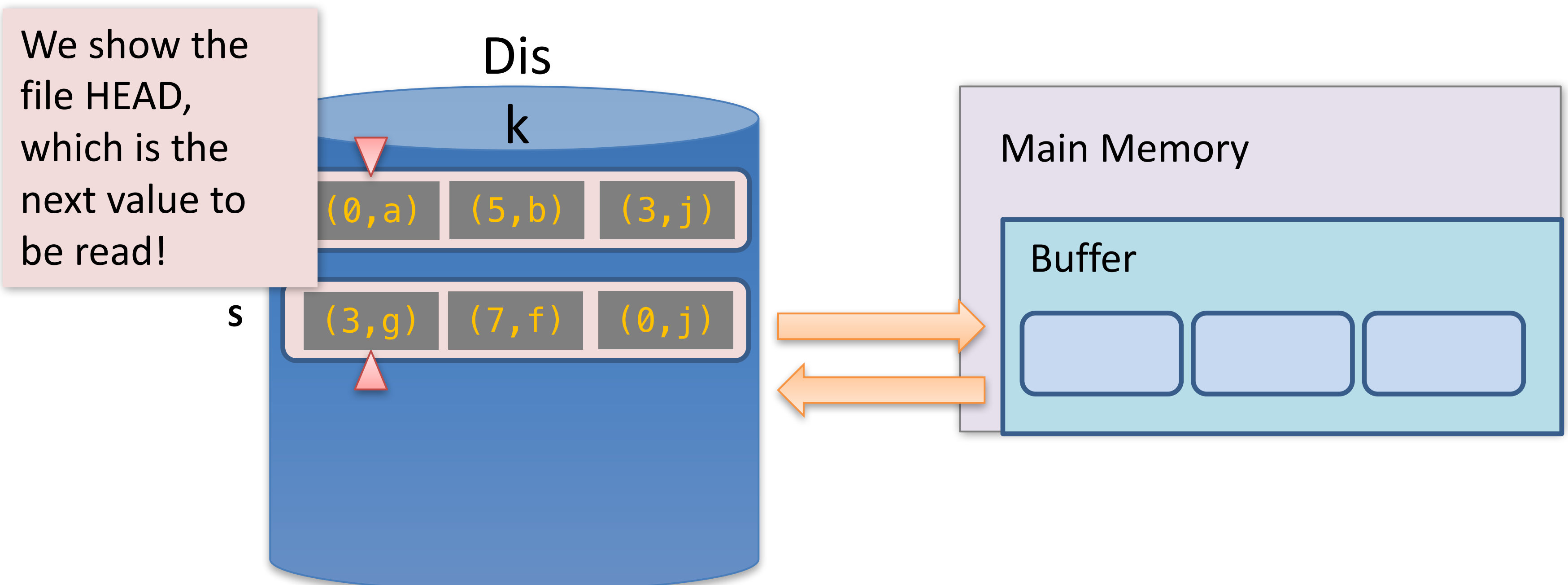
To compute $R \bowtie S$ *on A*:

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

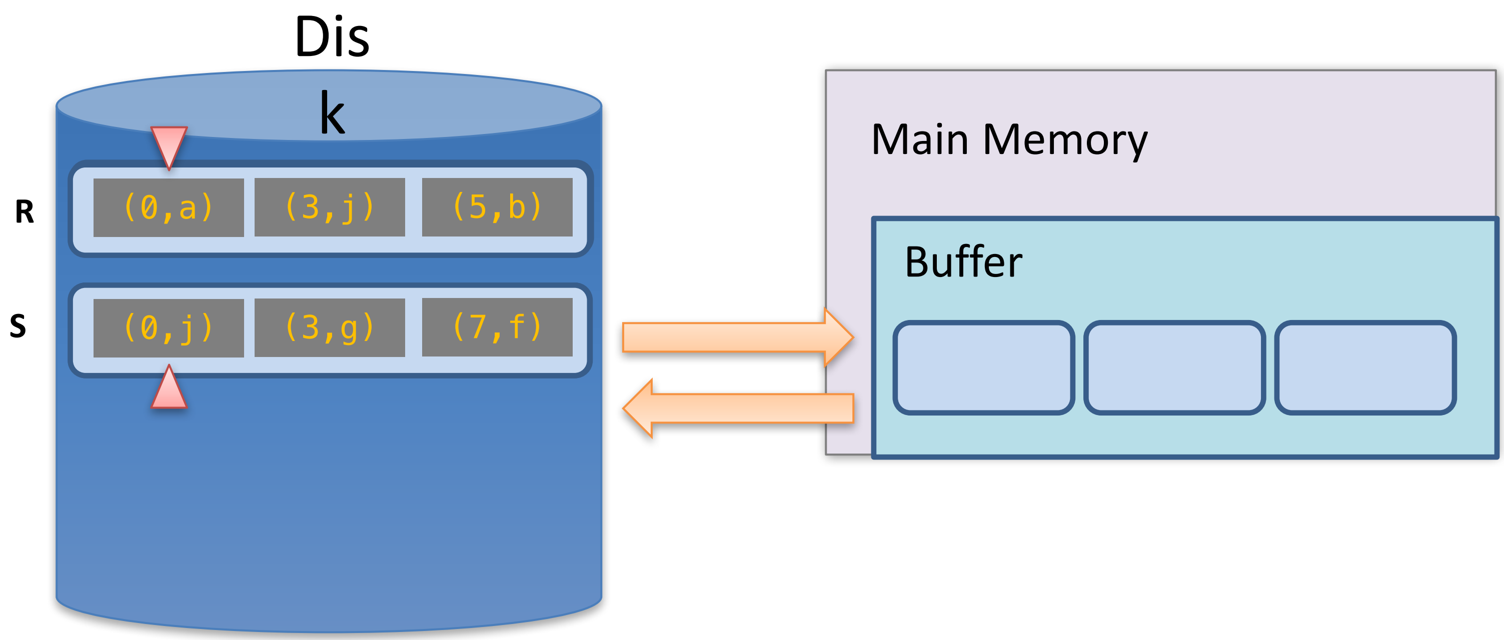
SMJ Example: $R \bowtie S$ *on* A with 3 page buffer

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



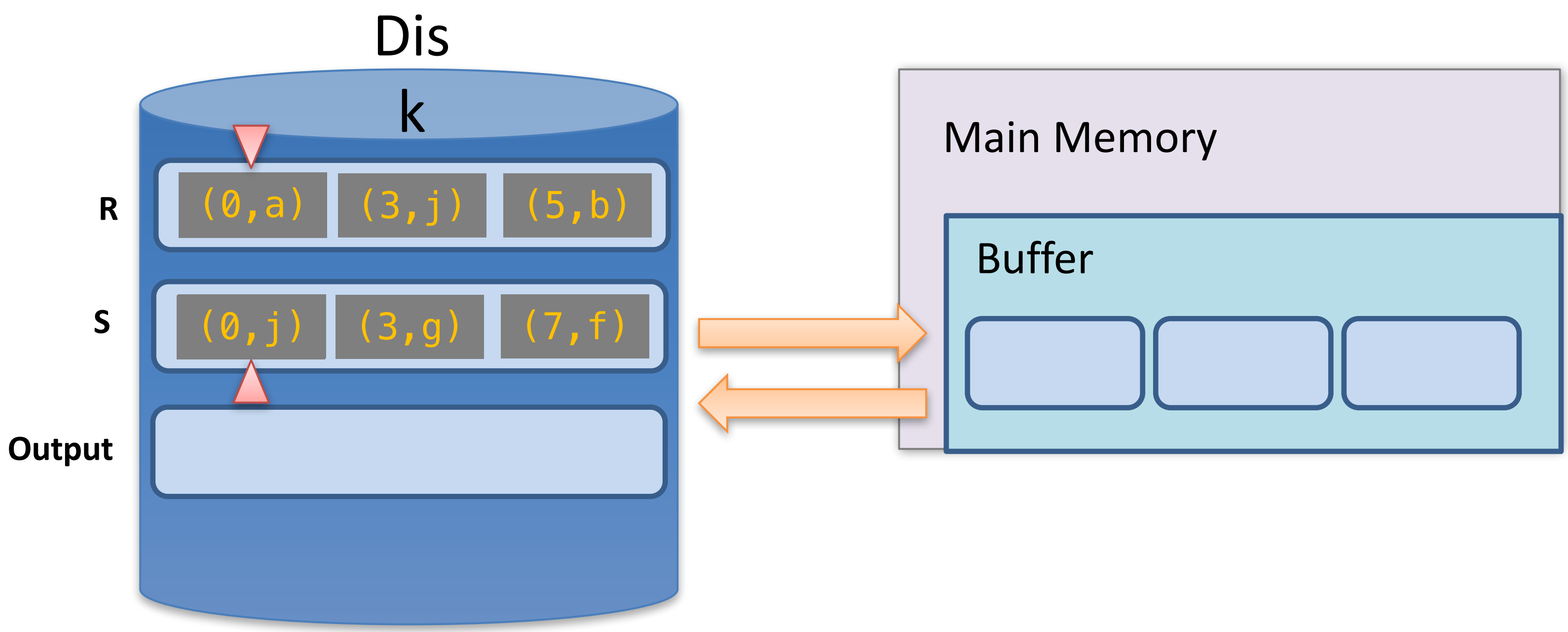
SMJ Example: $R \bowtie S$ *on* A with 3 page buffer

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



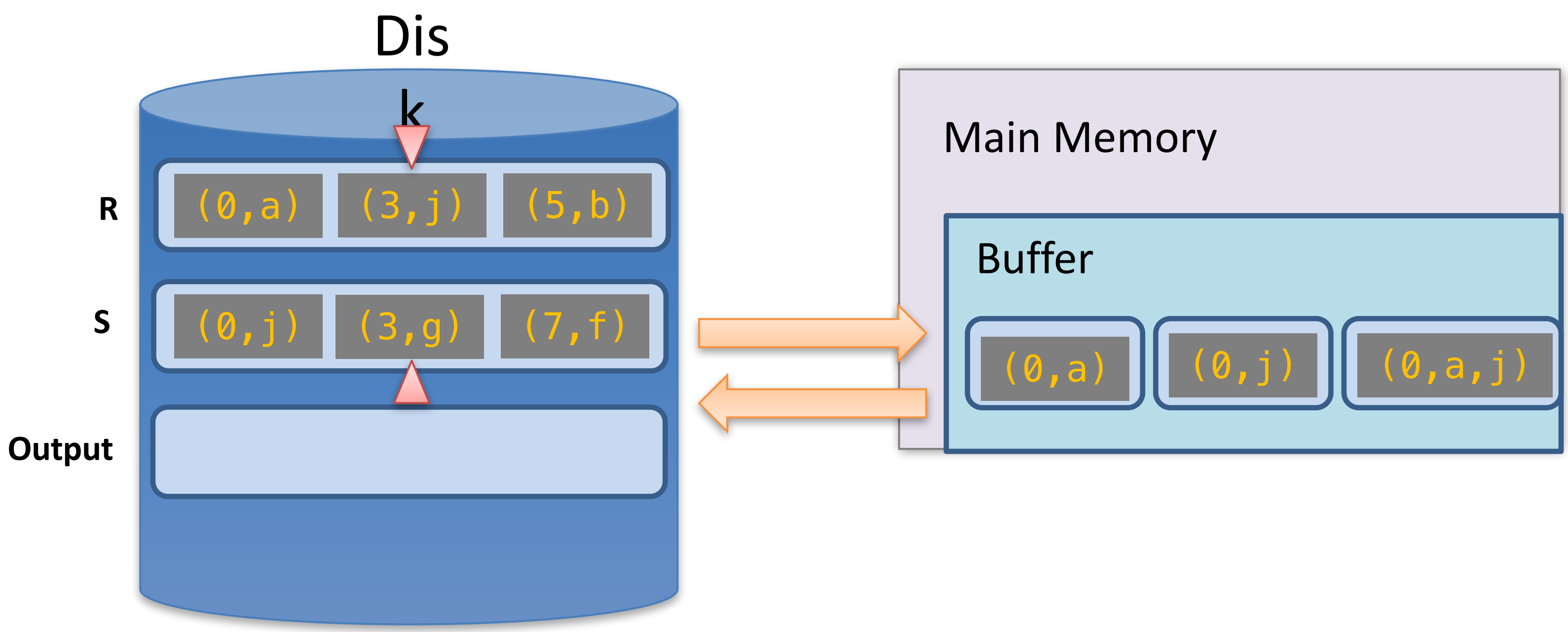
SMJ Example: $R \bowtie S$ *on A* with 3 page buffer

2. Scan and “merge” on join key!



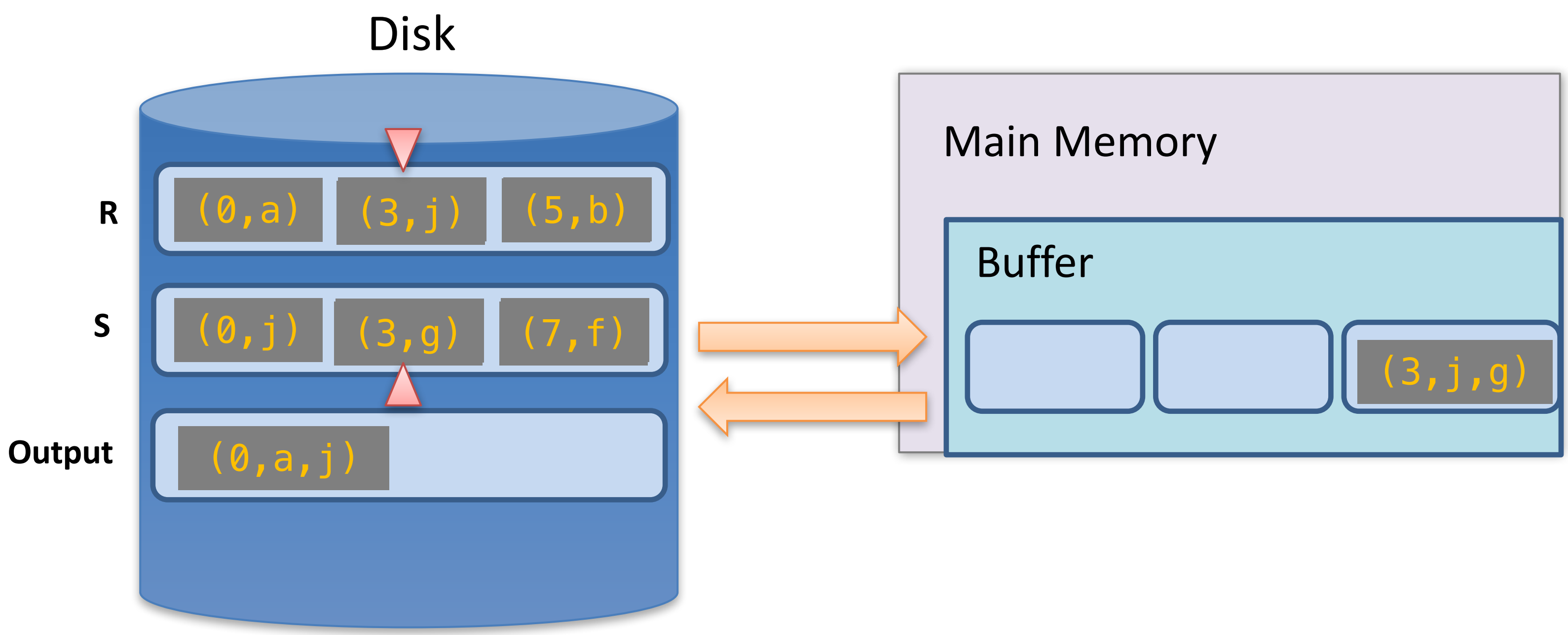
SMJ Example: $R \bowtie S$ *on A* with 3 page buffer

2. Scan and "merge" on join key!



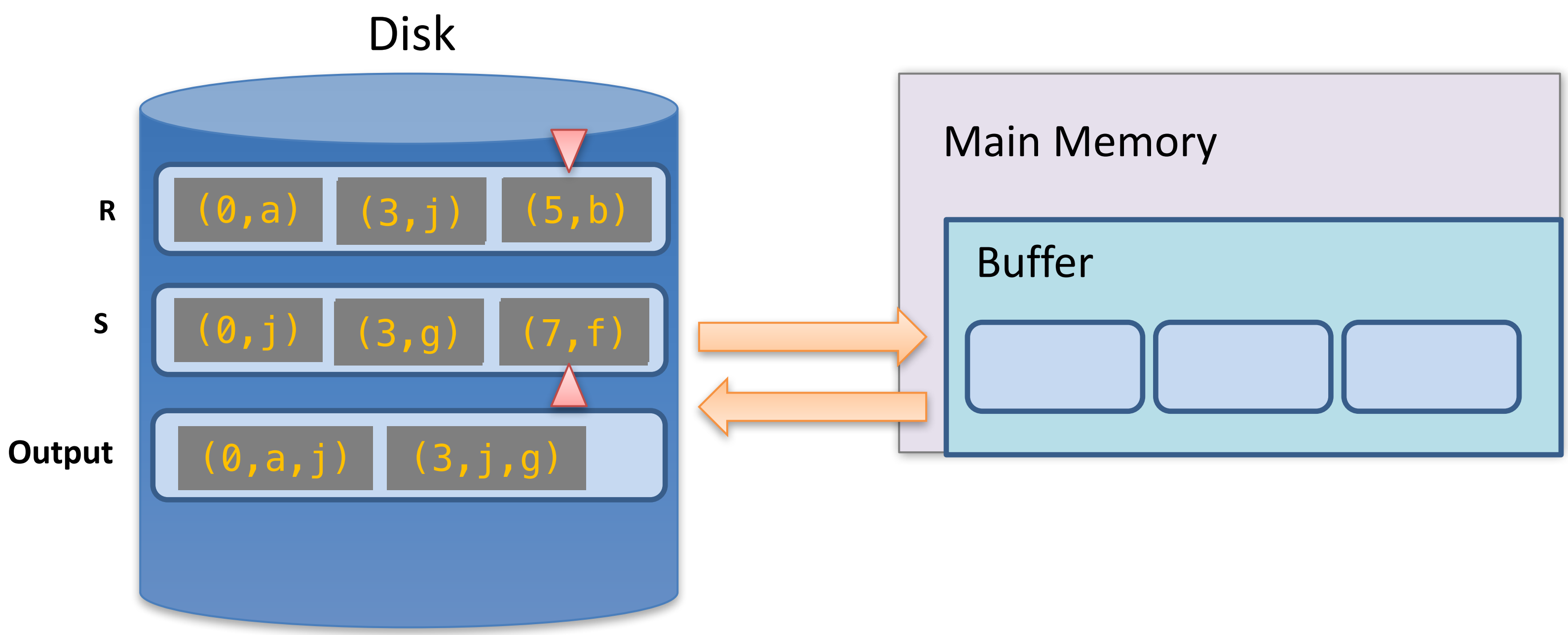
SMJ Example: $R \bowtie S$ *on A* with 3 page buffer

2. Scan and "merge" on join key!



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

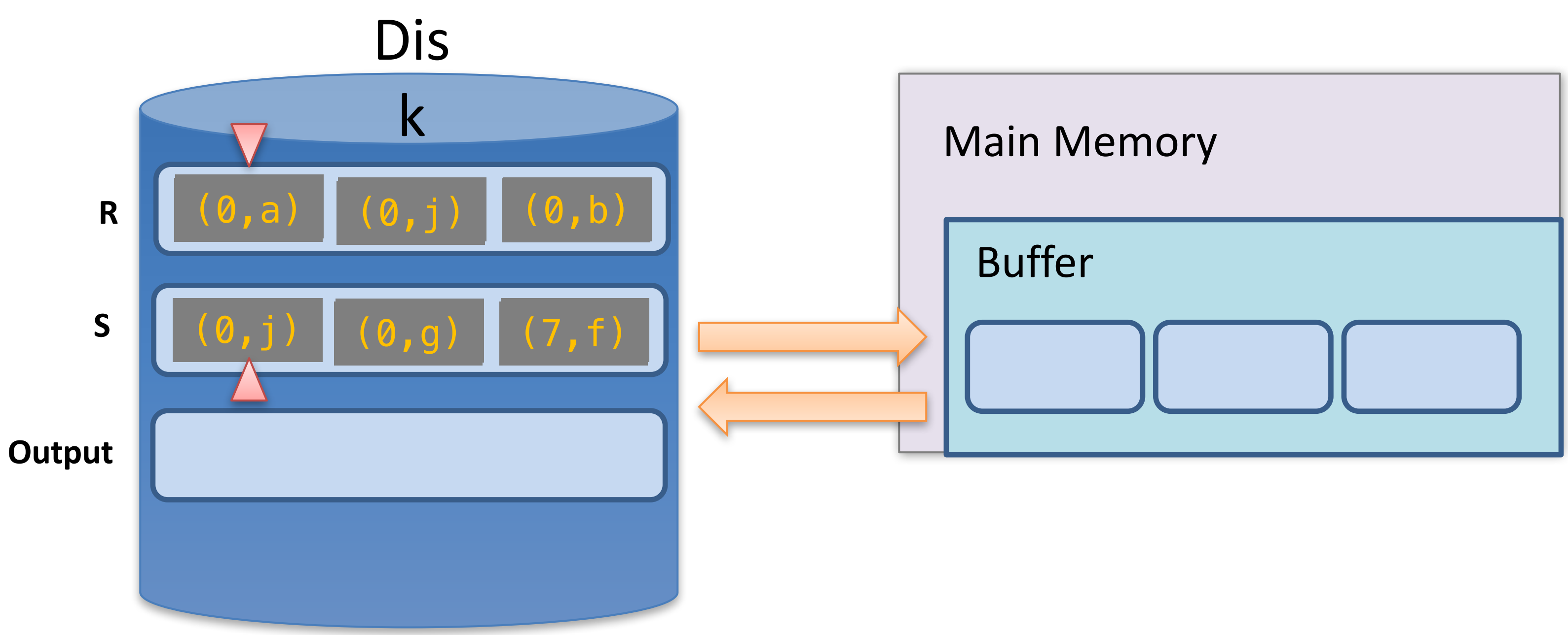
2. Done!



What happens with duplicate join keys?

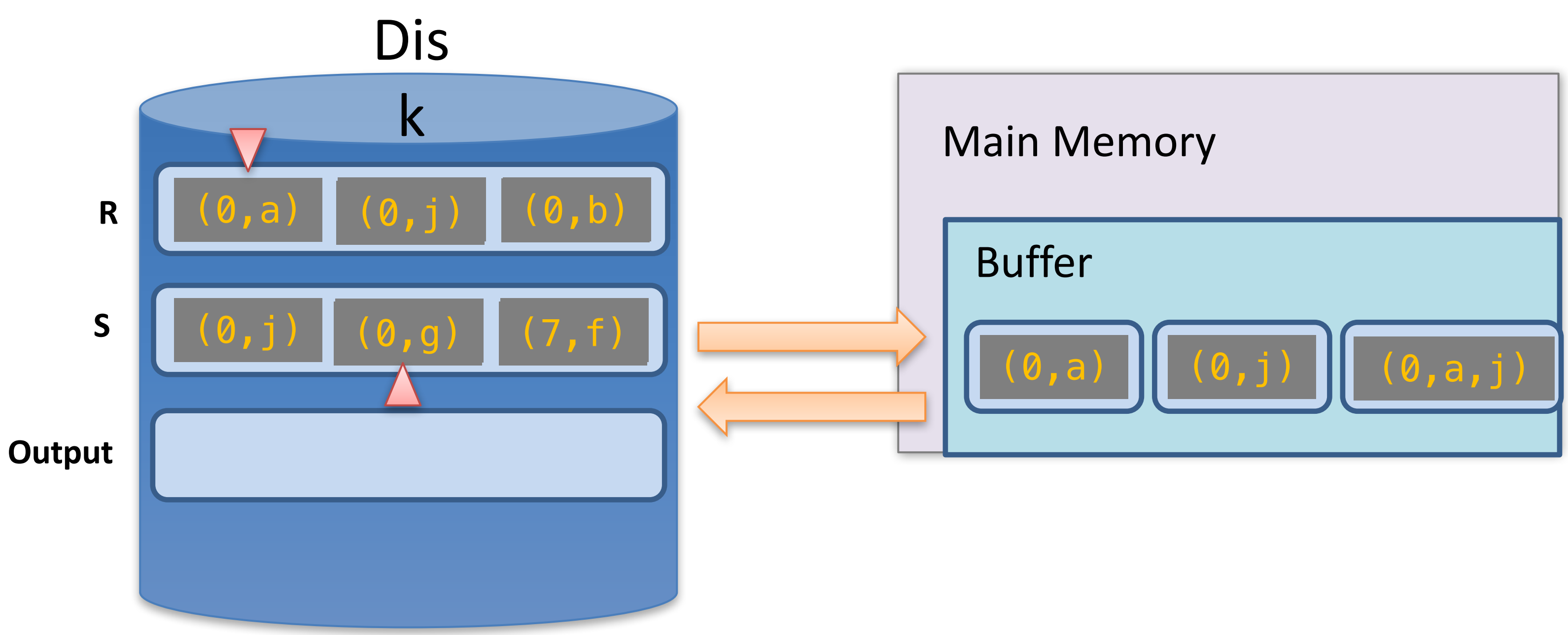
Multiple tuples with Same Join Key: “Backup”

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



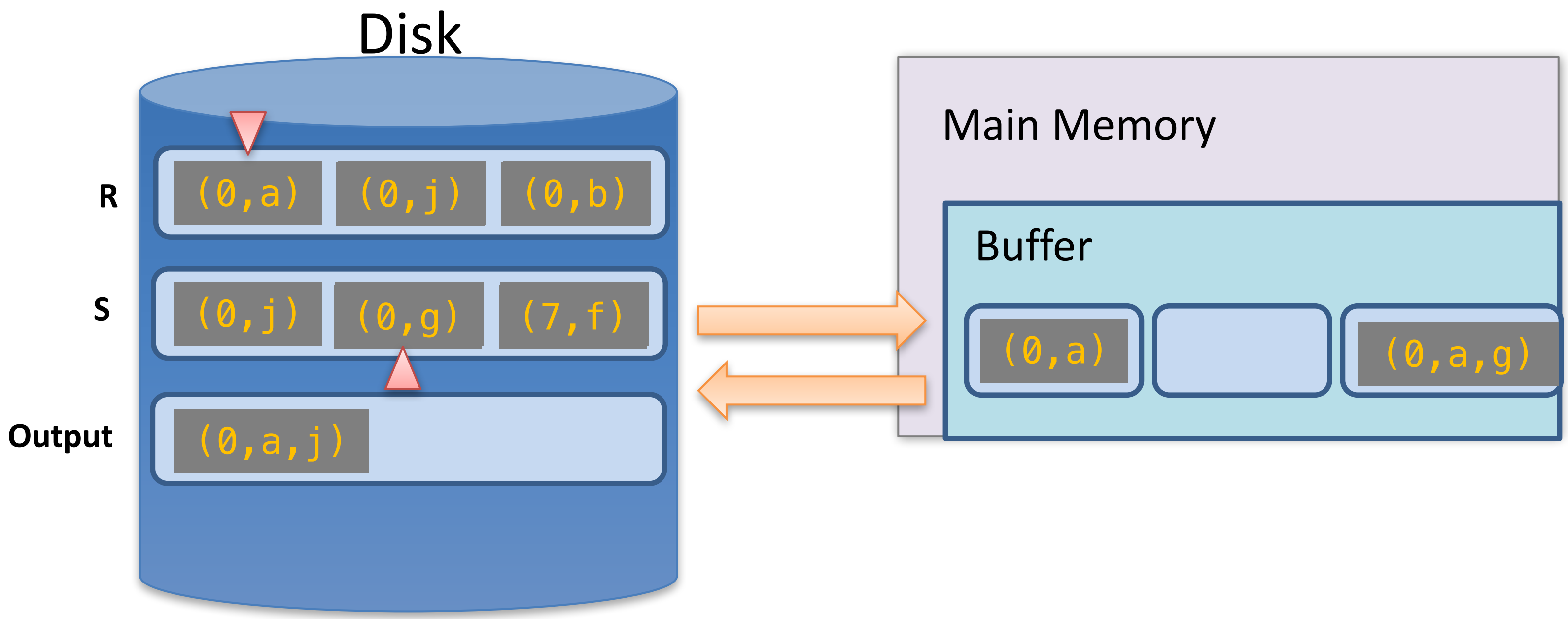
Multiple tuples with Same Join Key: "Backup"

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



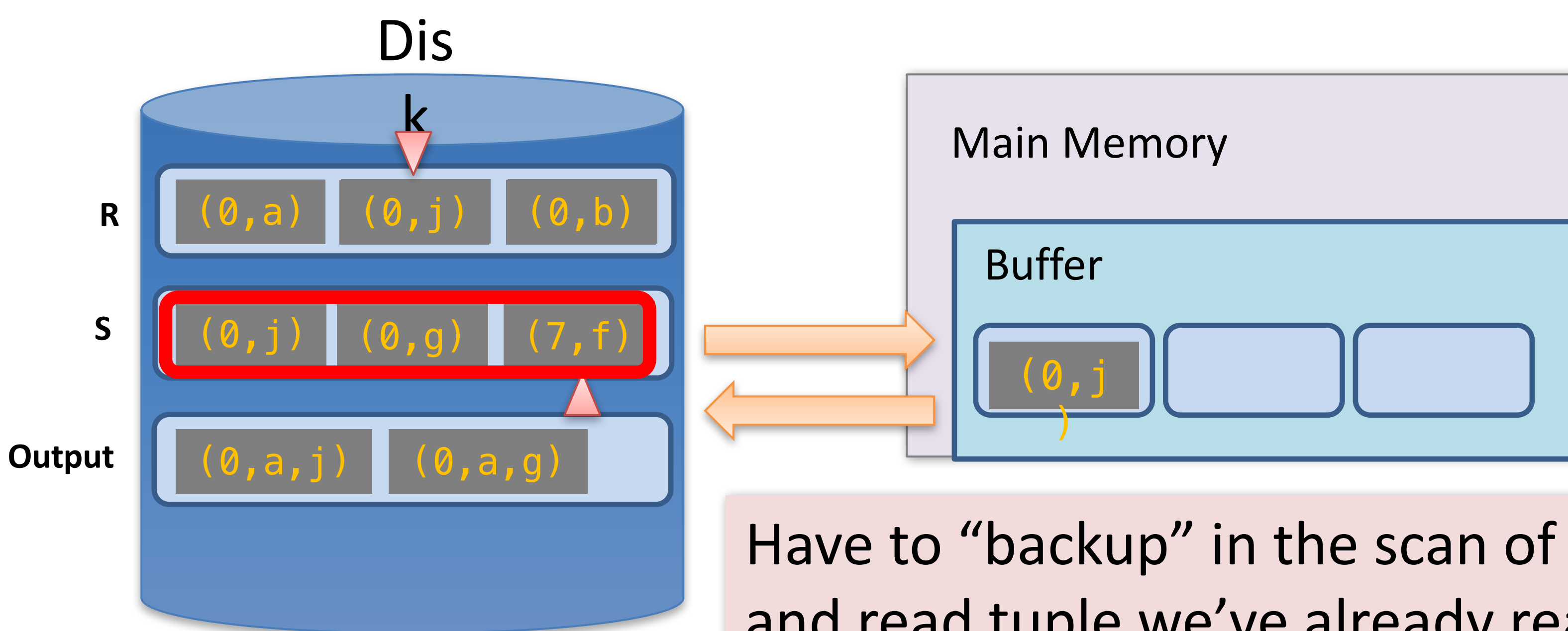
Multiple tuples with Same Join Key: "Backup"

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

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

SMJ vs. BNLJ

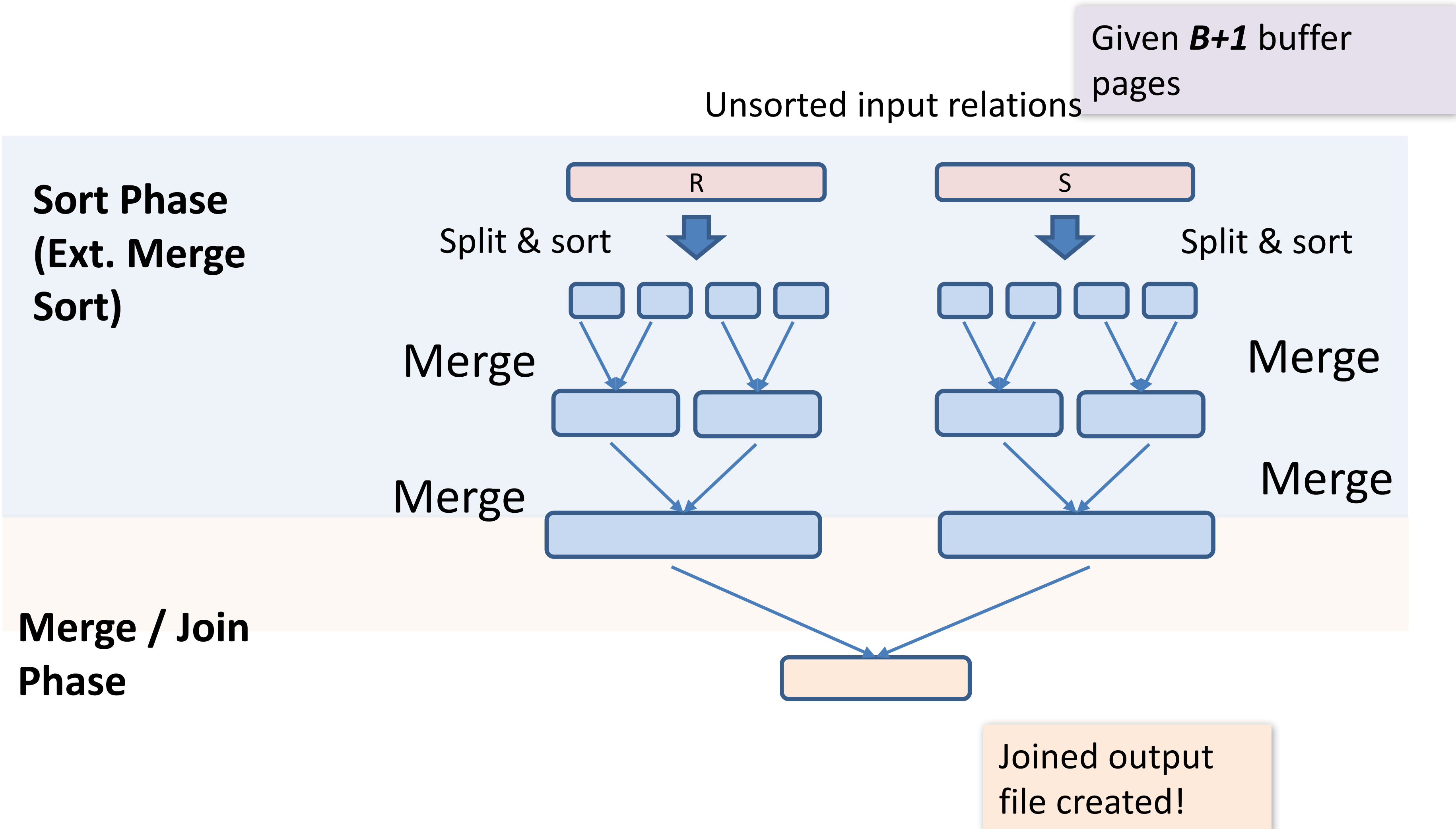
- If we have 100 buffer pages, P(R) = 1000 pages and P(S) = 500 pages:
  - Sort both in two passes: 2 * 2 * 1000 + 2 * 2 * 500 = **6,000 IOs**
  - Merge phase 1000 + 500 = 1,500 IOs
  - = **7,500 IOs + OUT**

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!***

Basic SMJ


Given ***B+1*** buffer pages
Unsorted input relations
Sort Phase (Ext. Merge Sort)
R
S
Split & sort
Split & sort
Merge
Merge
Merge
Merge
Merge / Join Phase
Joined output file created!

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.

4. 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 $h_B(x)$!= $h_B(y)$

Hash Join: High-level procedure

Note again that we are only considering equality constraints here

To compute $R \bowtie S$ *on* A:

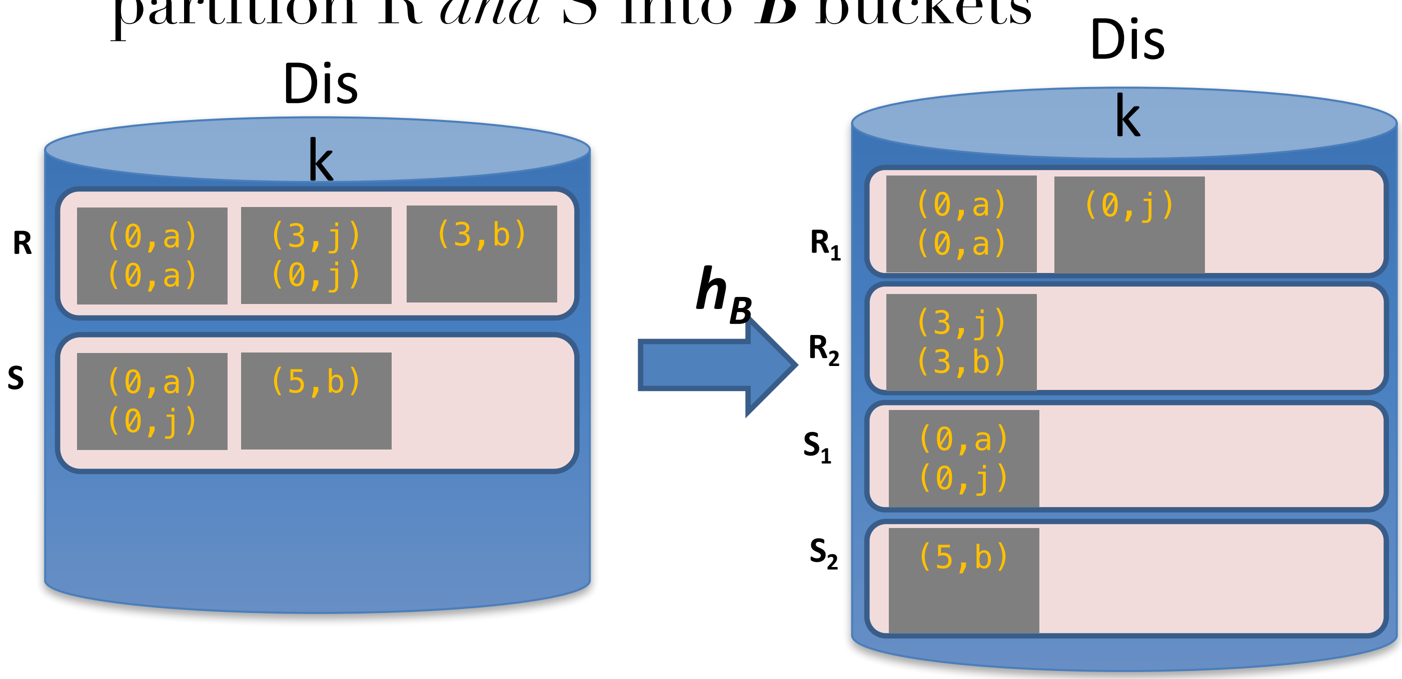
1. **Partition Phase**: Using one (shared) hash function $\boldsymbol{h_B}$, partition R *and* S into $\boldsymbol{B}$ buckets

2. **Matching Phase**: Take pairs of buckets whose tuples have the same values for $\boldsymbol{h}$, 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

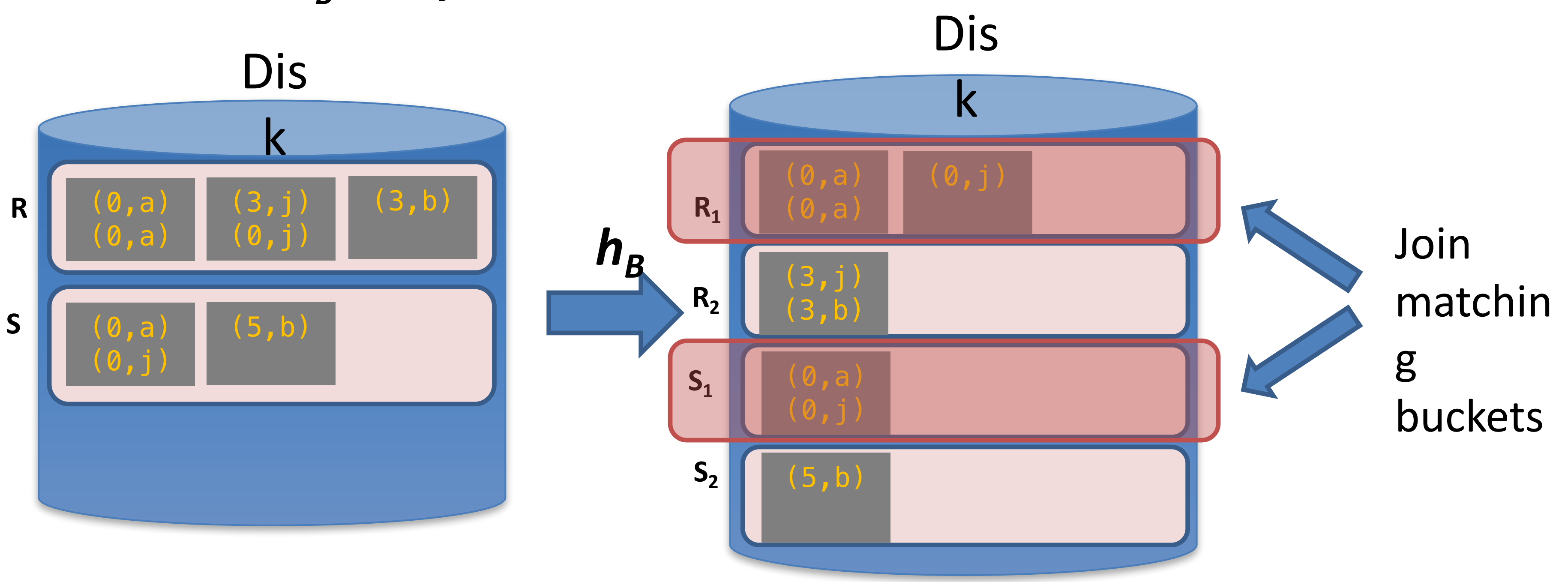


Note our new convention: pages each have two tuples (one per row)

More detail in a second…

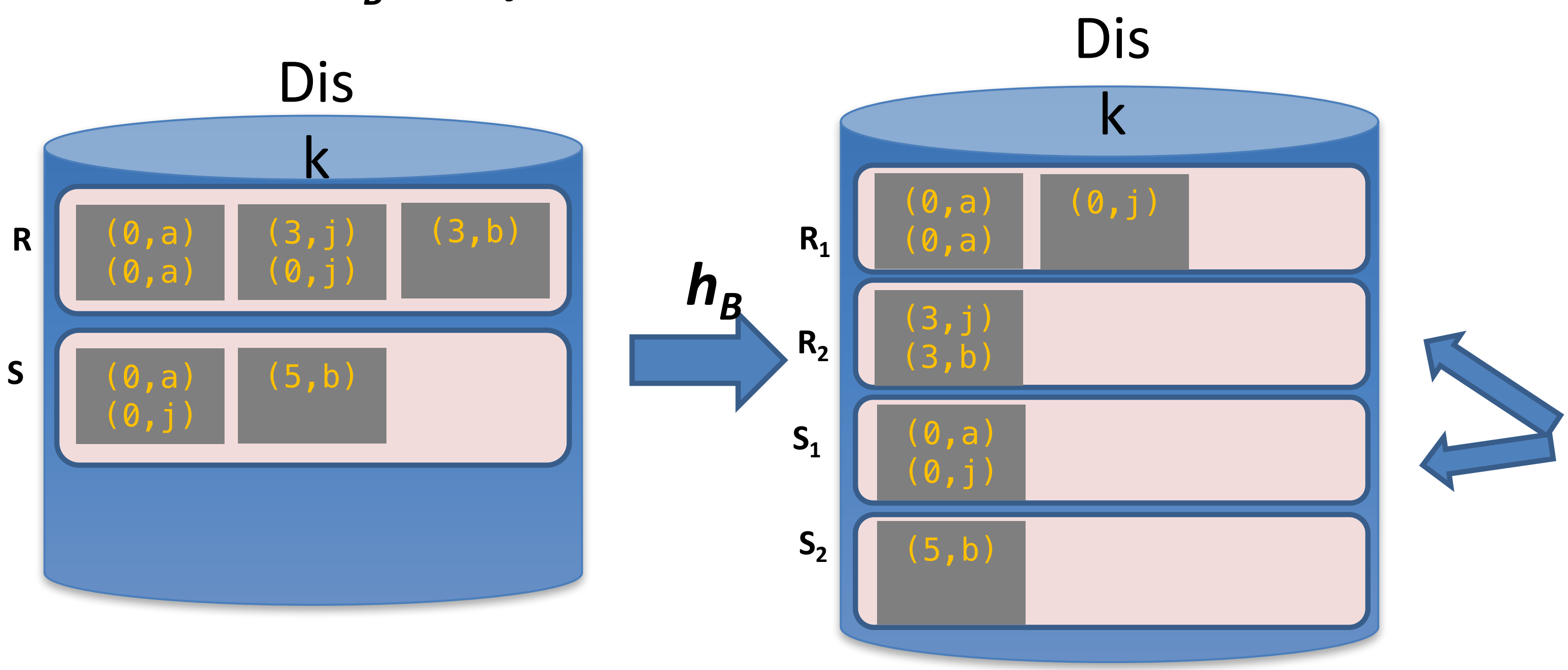
Hash Join: High-level procedure

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 $\boldsymbol{h_B}$, and join these



Don't have to join the others!

E.g. (S_1 and R_2)!

Hash Join Phase 1: Partitioning

Goal: For each relation, partition relation into buckets such that if $h_B(t_i.A) = h_B(t_j.A)$ 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, 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**

How big *do we want* the resulting buckets?

- 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

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

- Recall: If we want to join a bucket from R and one from S, we can do BNLJ **in linear time** if for *one of them (wlog say R),* $\boldsymbol{P(R) \leq B-1}$!
  - And more generally, being able to fit bucket in memory is advantageous

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

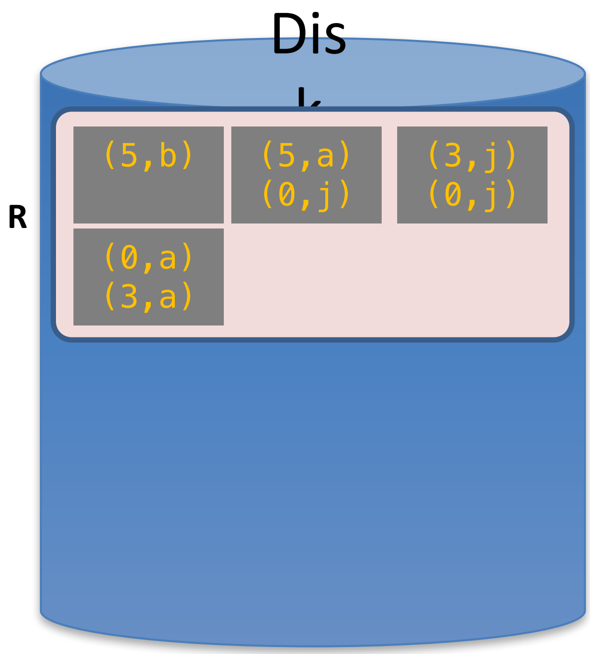
- We can keep partitioning buckets that are > B-1 pages, until they are $\leq \boldsymbol{B-1}$ **pages**
  - Using a new hash key which will split them…

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)



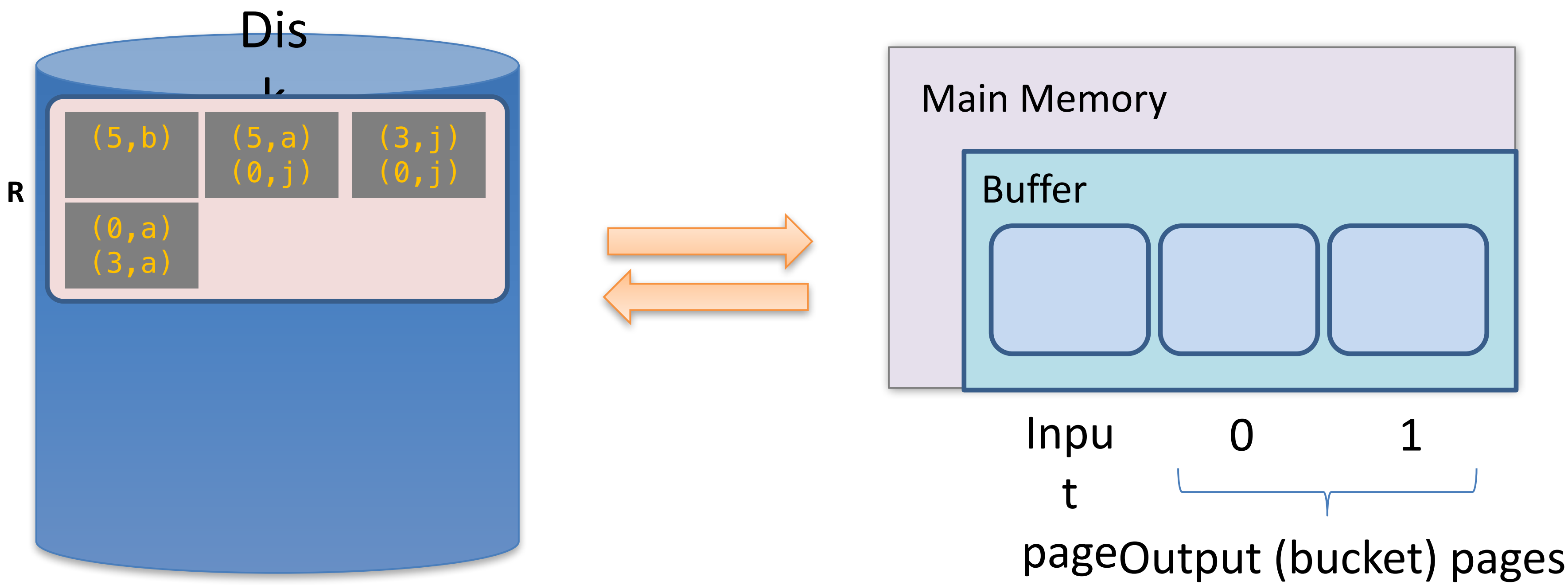
For simplicity, we'll look at partitioning one of the two relations- we just do the same for the other relation!

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

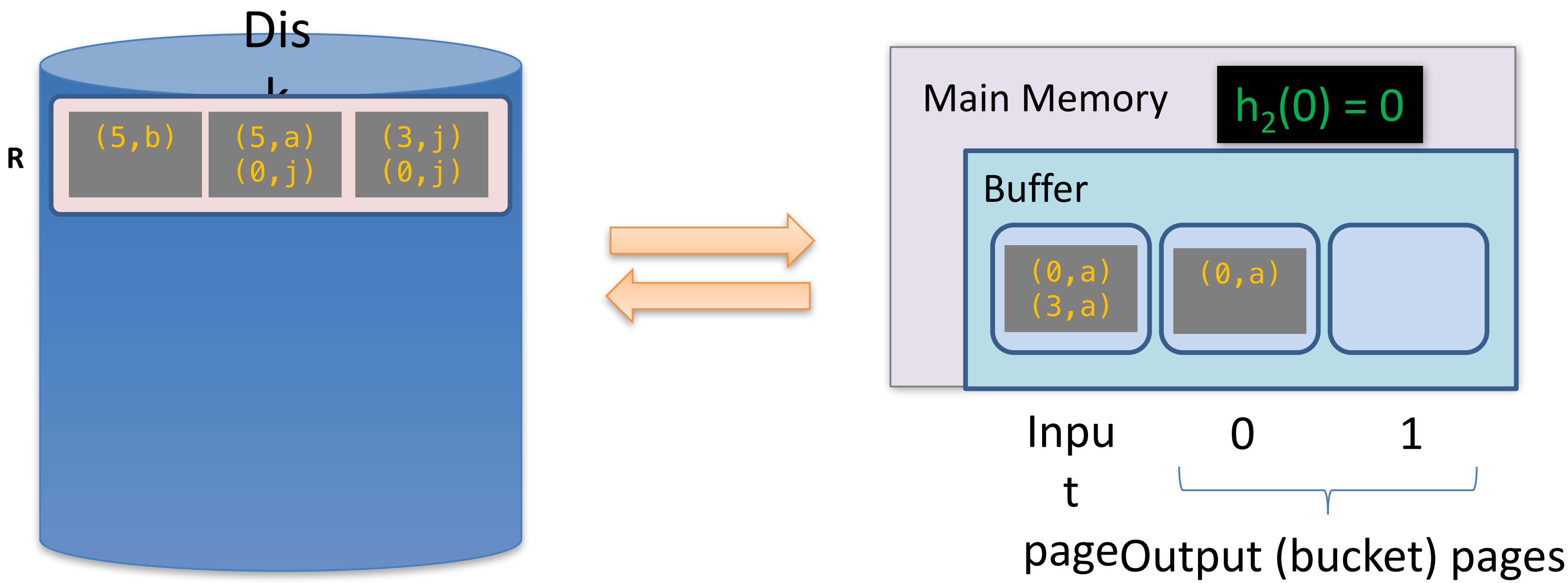
1. We read pages from R into the "input" page of the buffer…



Hash Join Phase 1: Partitioning

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

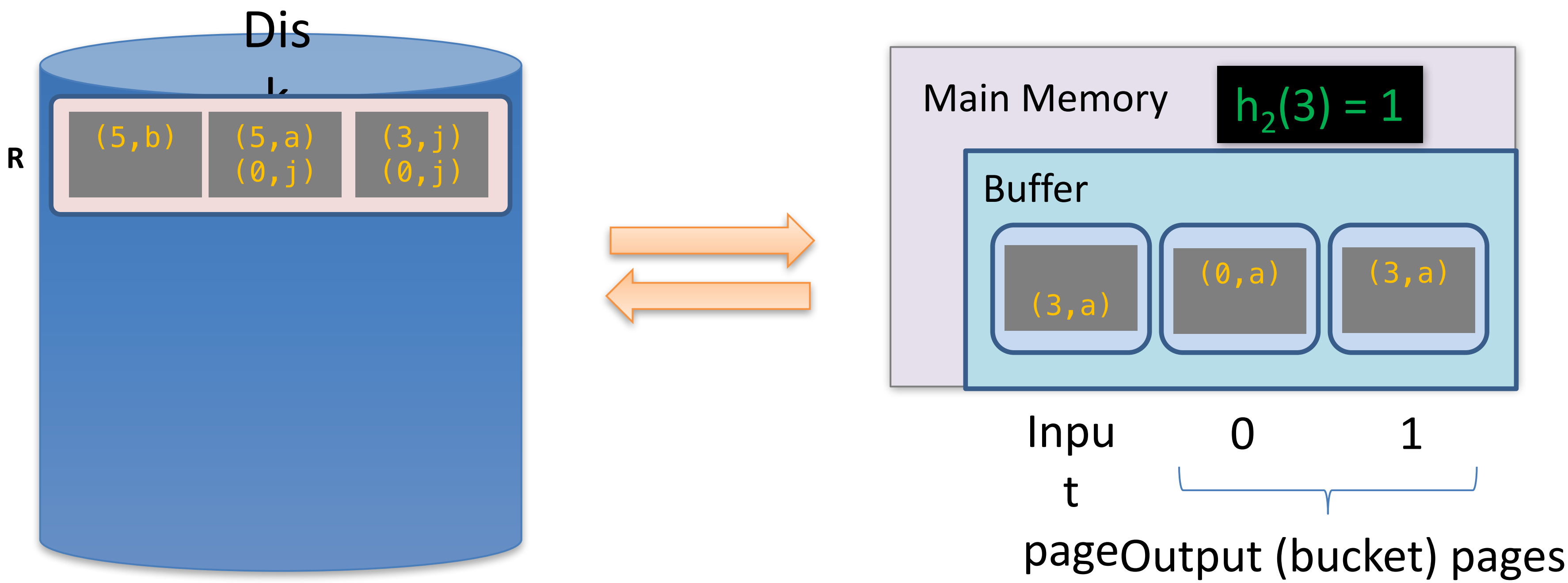
2. Then we use **hash function h_2** to sort into the buckets, which each have one page in the buffer



Hash Join Phase 1: Partitioning

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

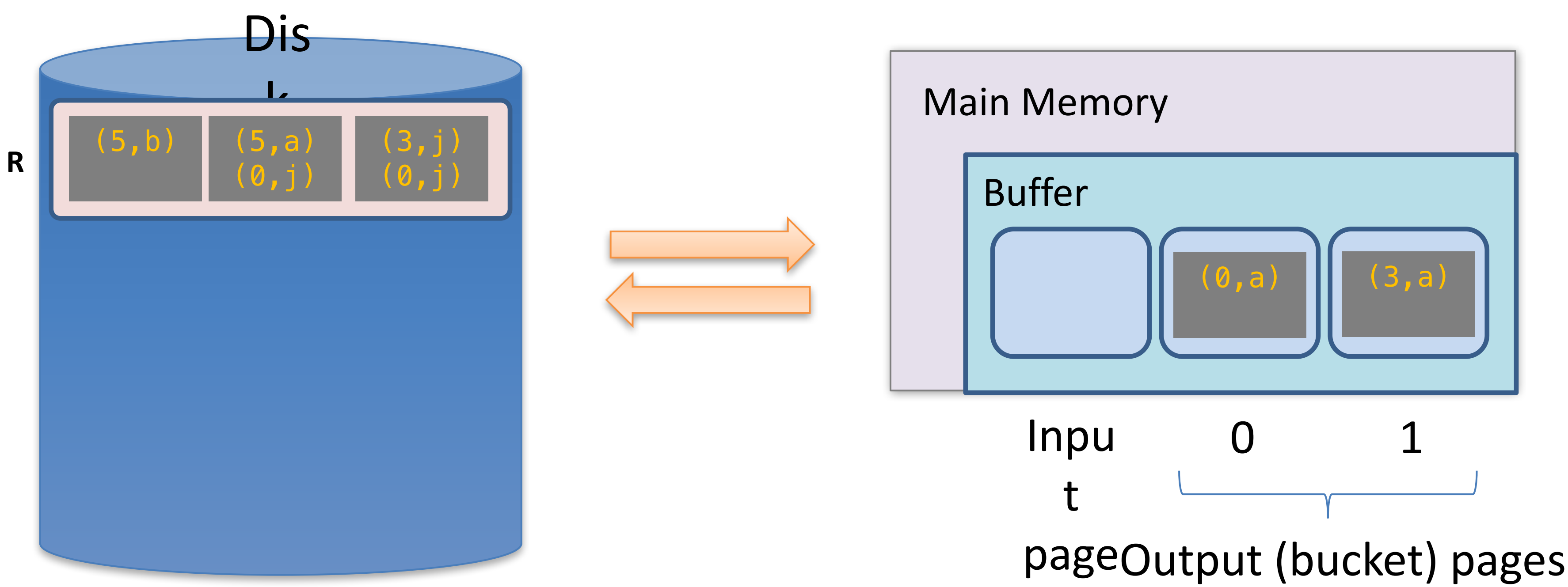
2. Then we use **hash function h_2** to sort into the buckets, which each have one page in the buffer



Hash Join Phase 1: Partitioning

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

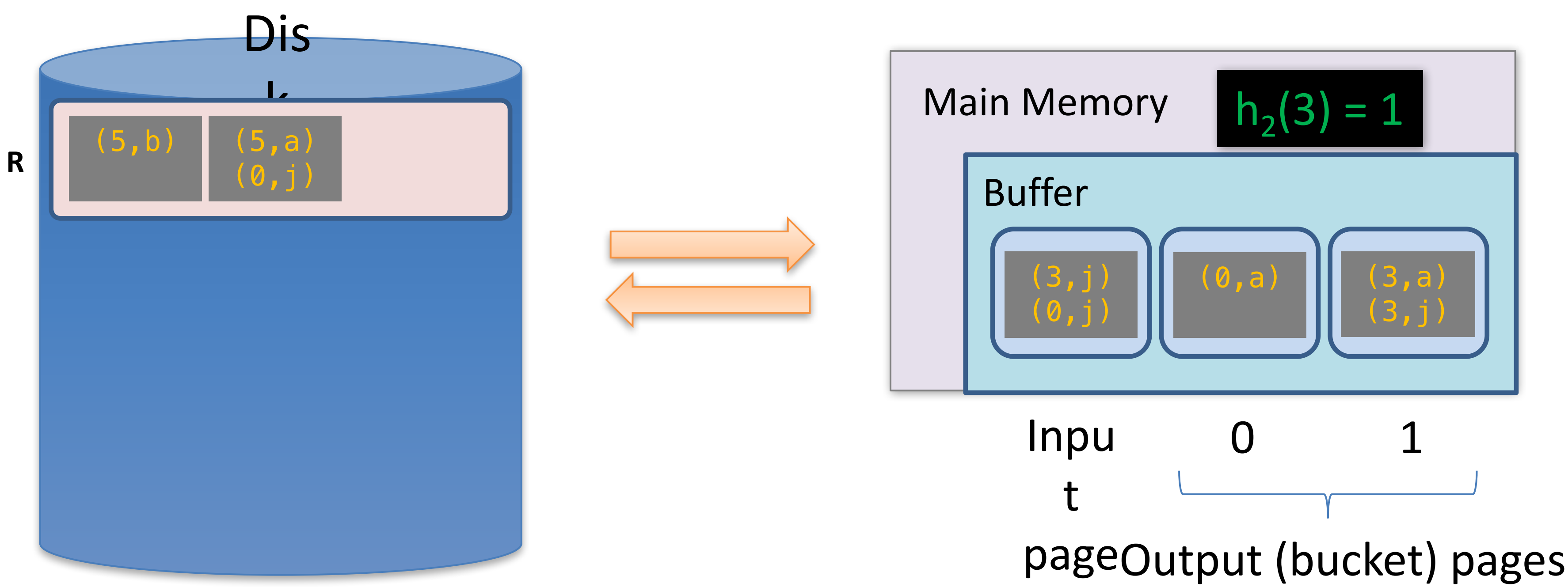
3. We repeat until the buffer bucket pages are full…



Hash Join Phase 1: Partitioning

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

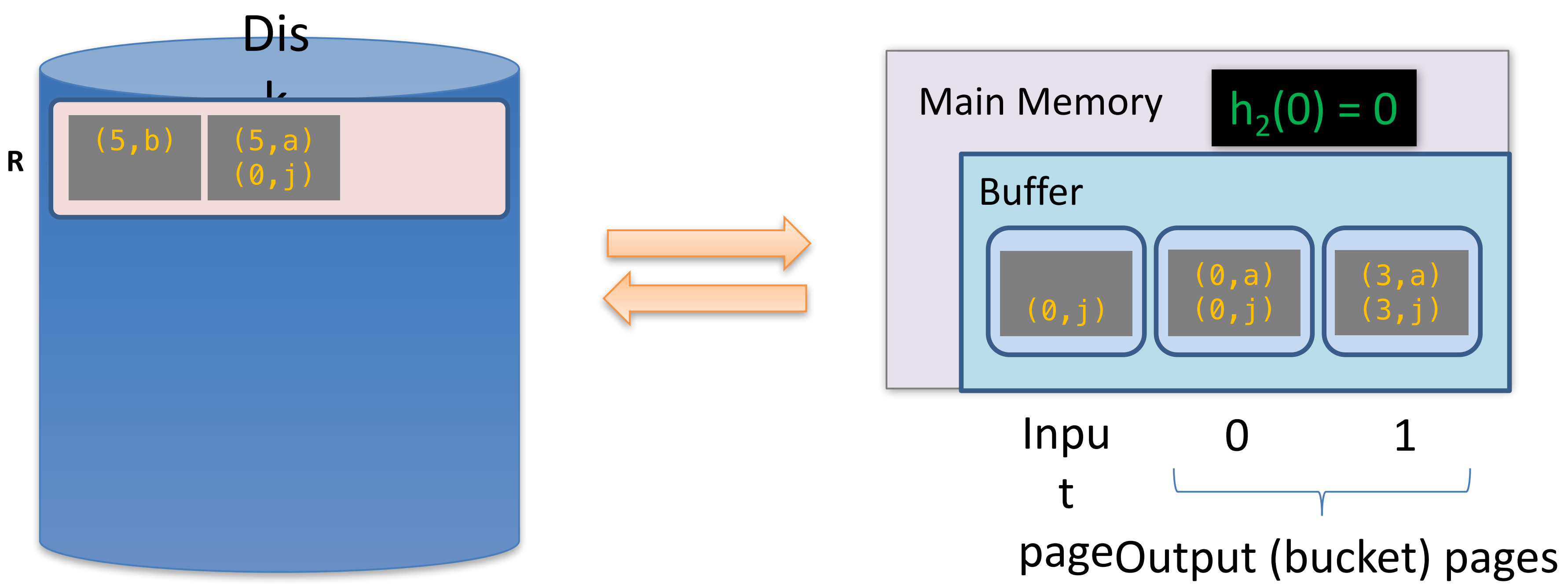
3. We repeat until the buffer bucket pages are full…



Hash Join Phase 1: Partitioning

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

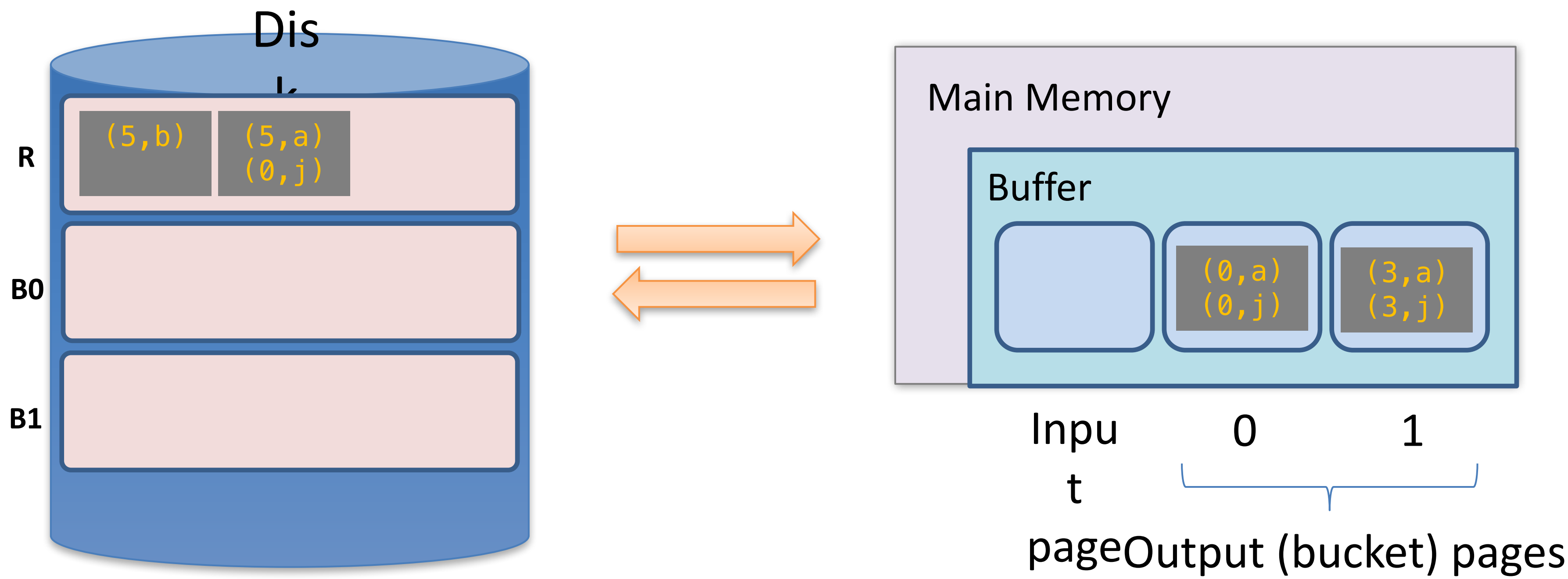
3. We repeat until the buffer bucket pages are full…



Hash Join Phase 1: Partitioning

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

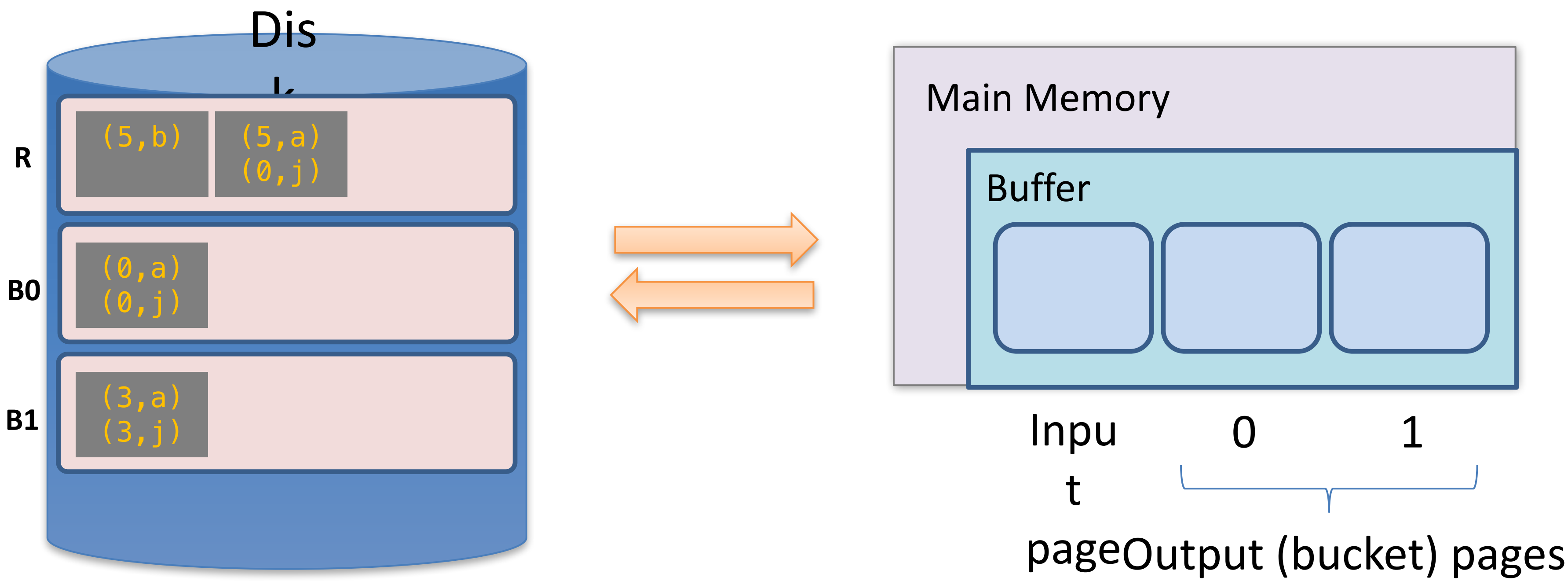
3. We repeat until the buffer bucket pages are full… then flush to disk



Hash Join Phase 1: Partitioning

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

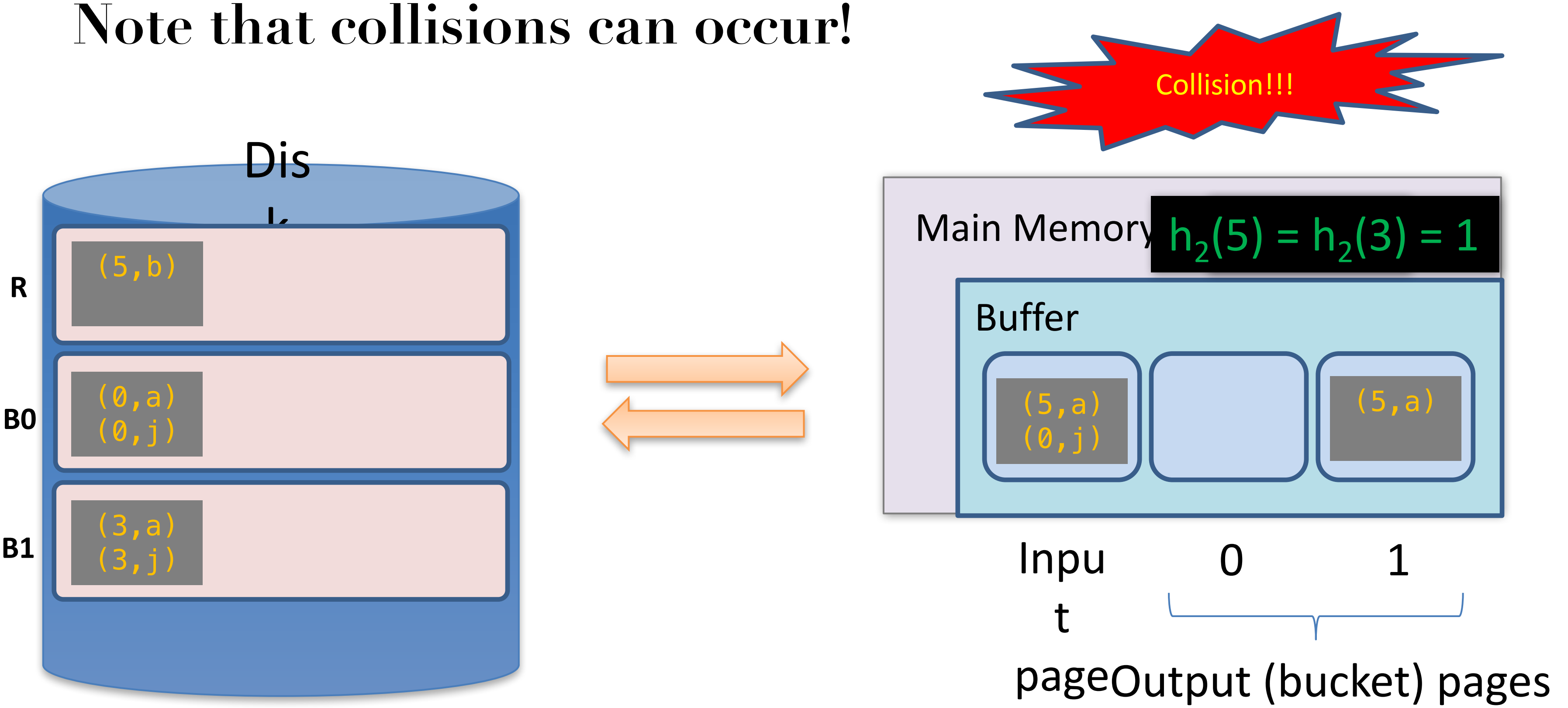
3. We repeat until the buffer bucket pages are full… then flush to disk



Hash Join Phase 1: Partitioning

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

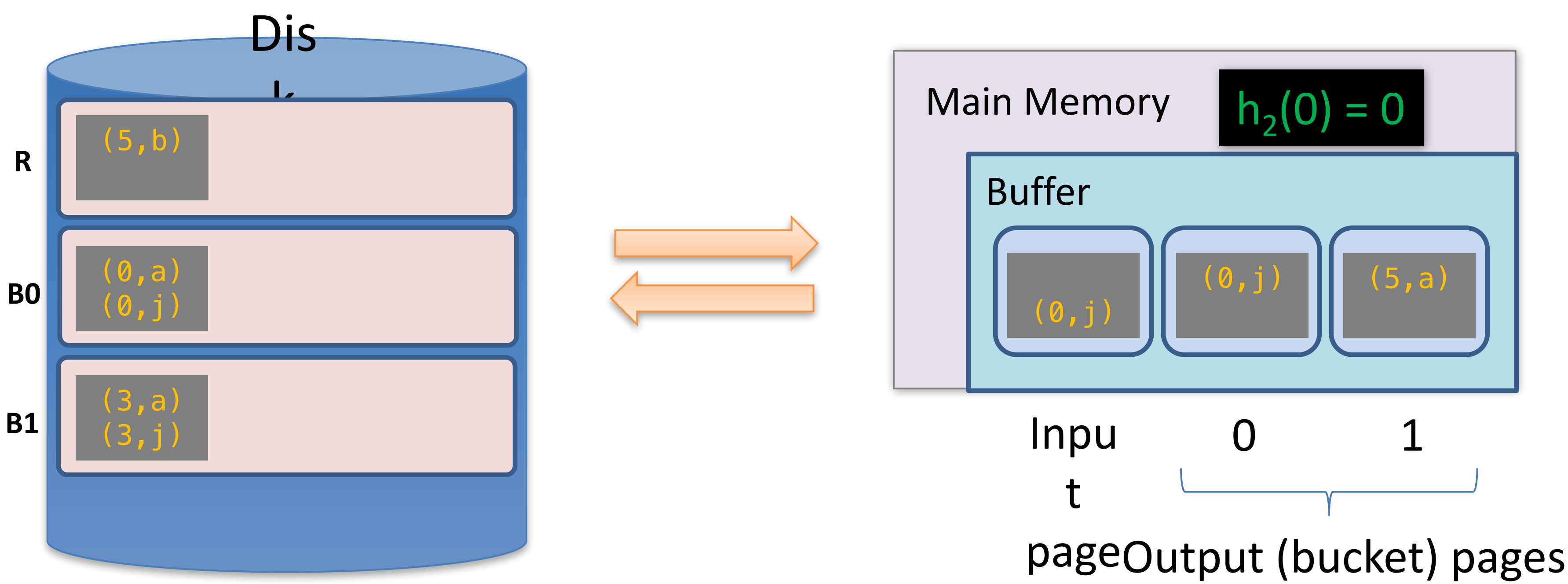
Note that collisions can occur!



Hash Join Phase 1: Partitioning

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

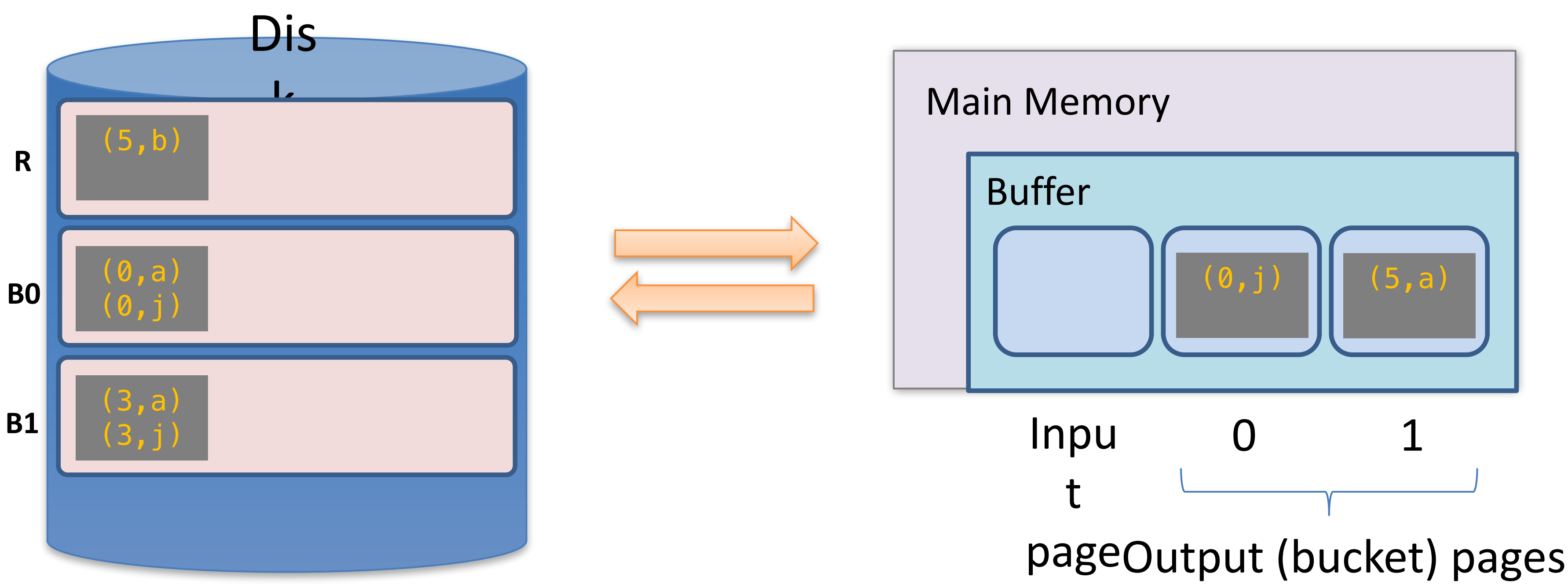
Finish this pass…



Hash Join Phase 1: Partitioning

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

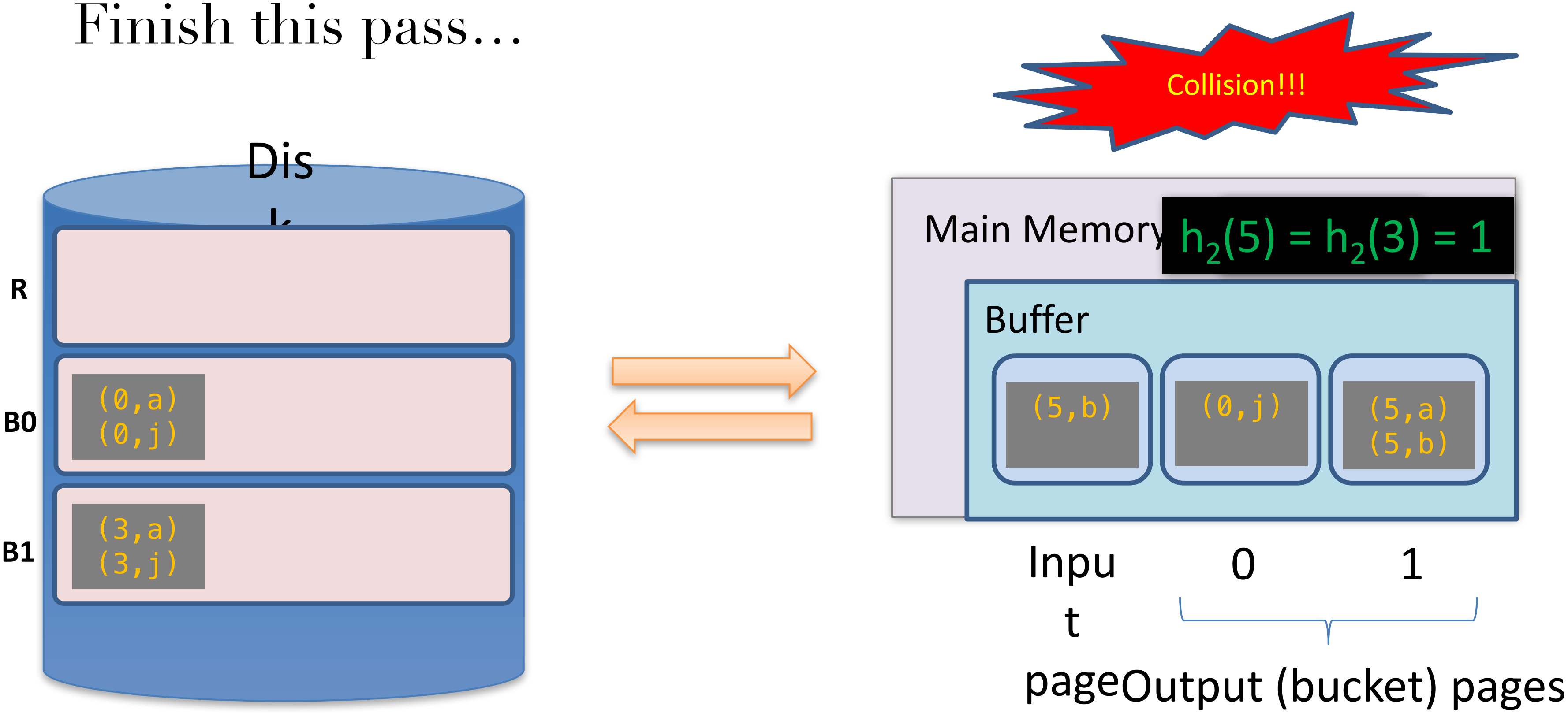
Finish this pass…



Hash Join Phase 1: Partitioning

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

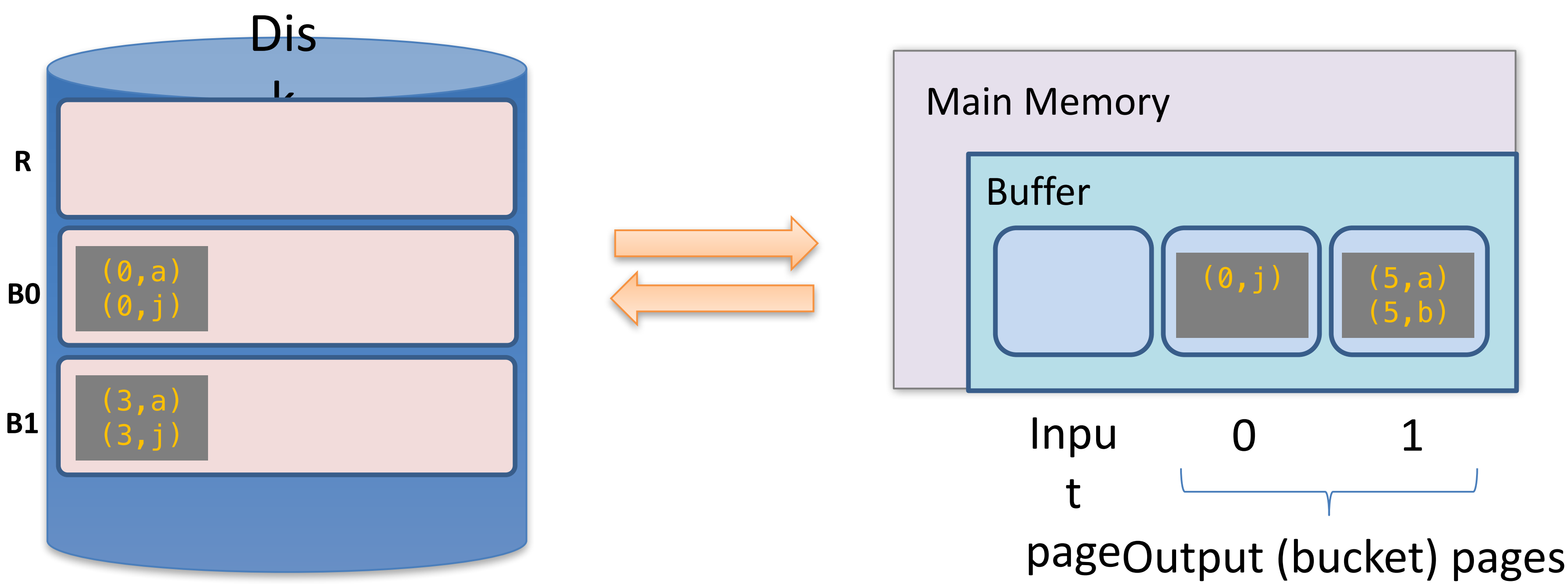
Finish this pass…



Hash Join Phase 1: Partitioning

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

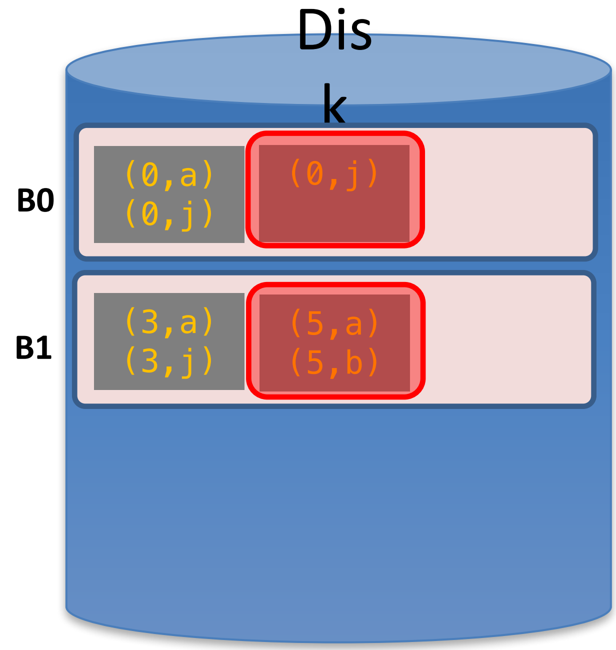
Finish this pass…



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:***

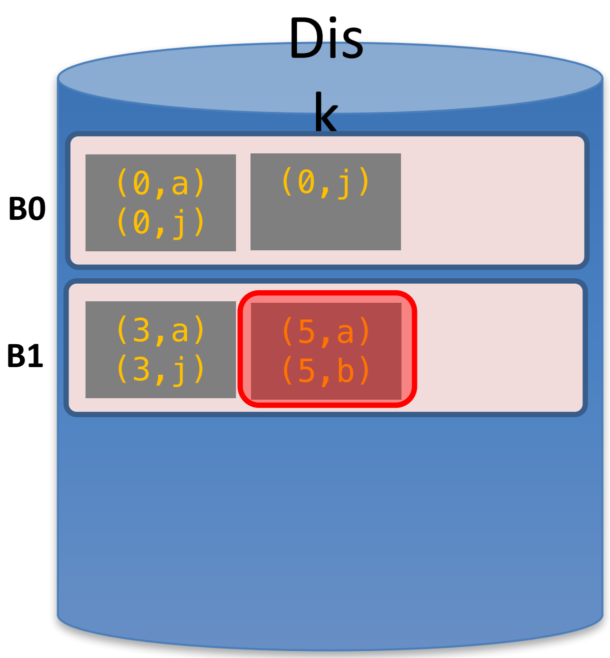


(1) Duplicate join keys

(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!

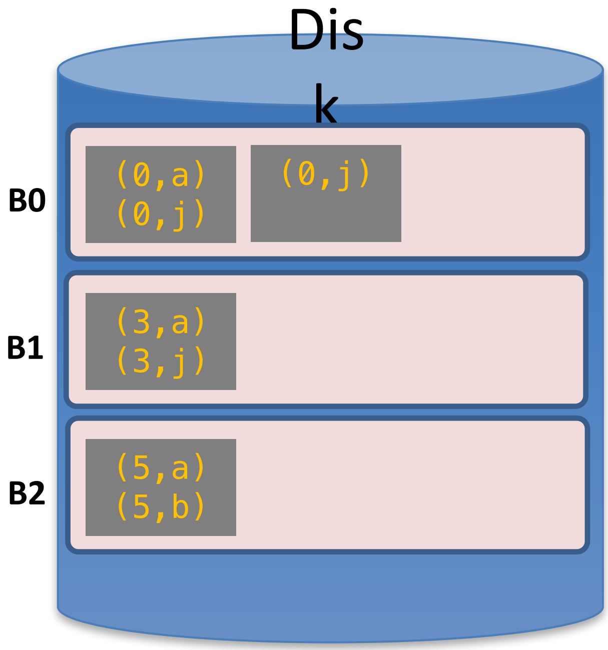
What hash function should we use?

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

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

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?

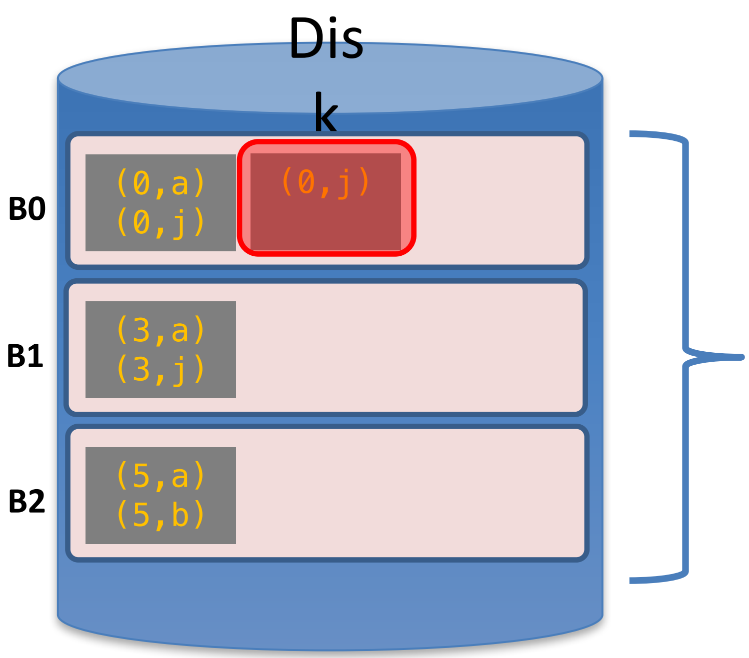
Do another pass with a different hash function, h'_2, ideally such that:

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

Hash Join Phase 1: Partitioning

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

What about duplicate join keys? Unfortunately this is a problem… but usually not a huge one.

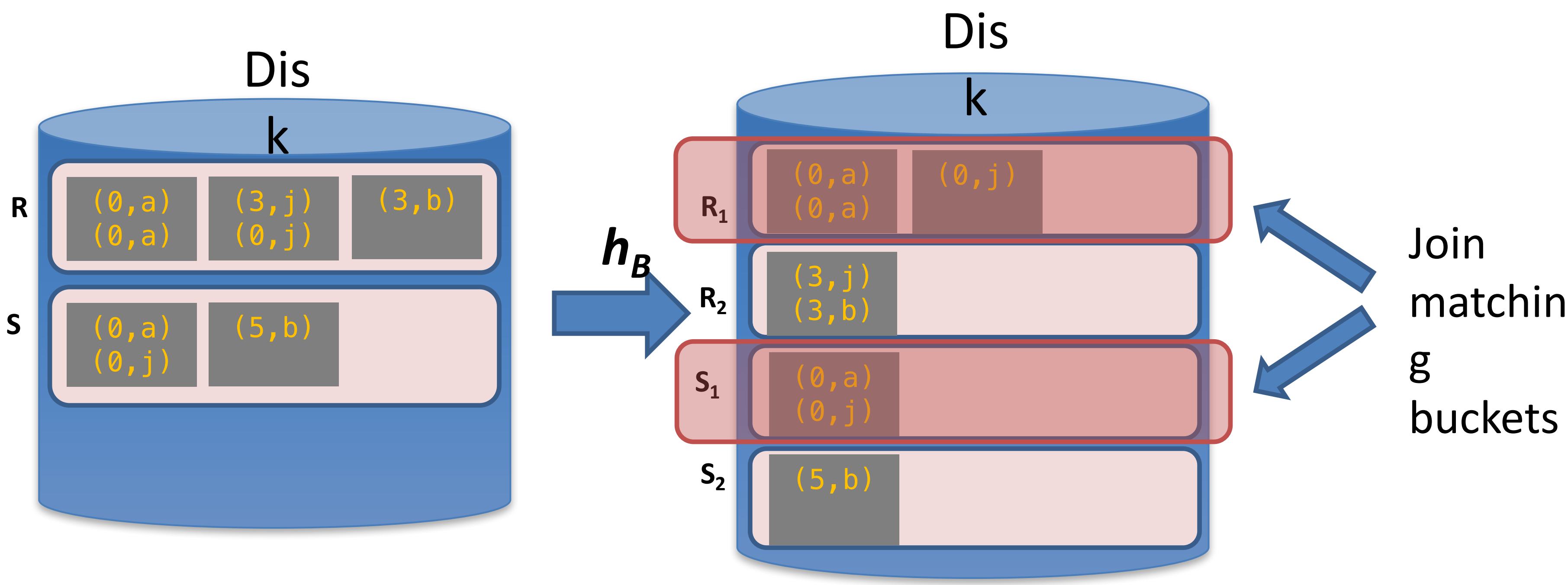


We call this unevenness in the bucket size __**skew**__

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

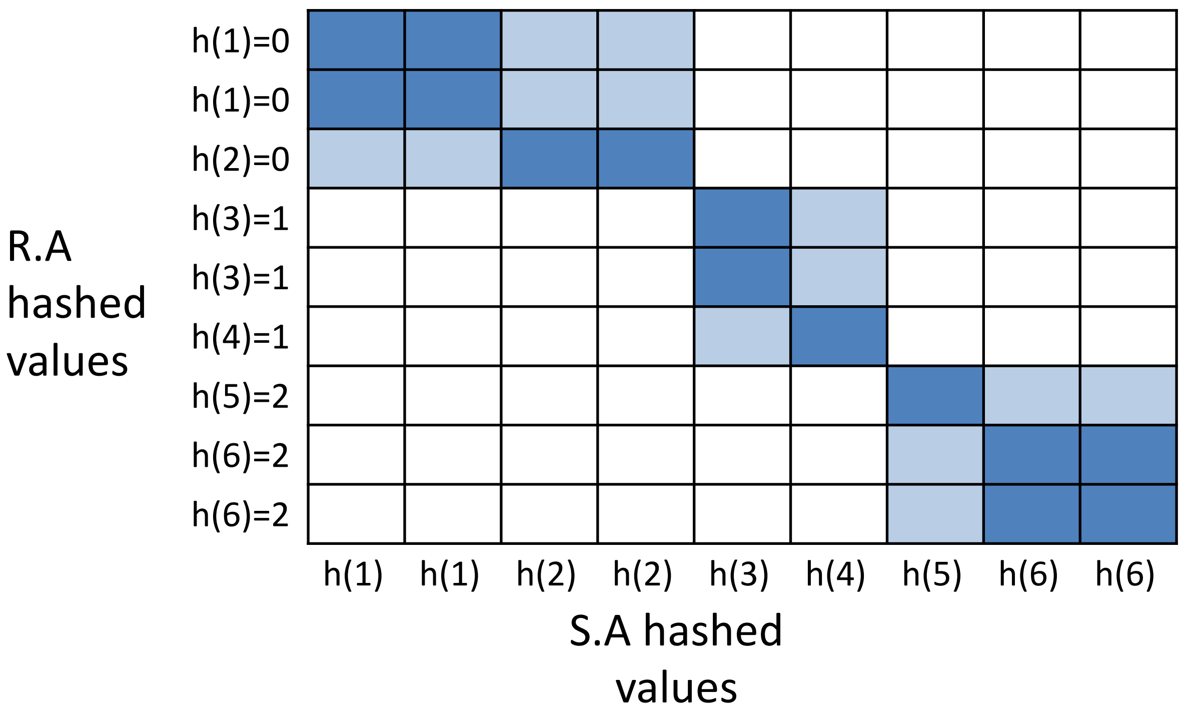
- Note that since x = y → h(x) = h(y), we only need to consider pairs of buckets (one from R, one from S) that have the same hash function value

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

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

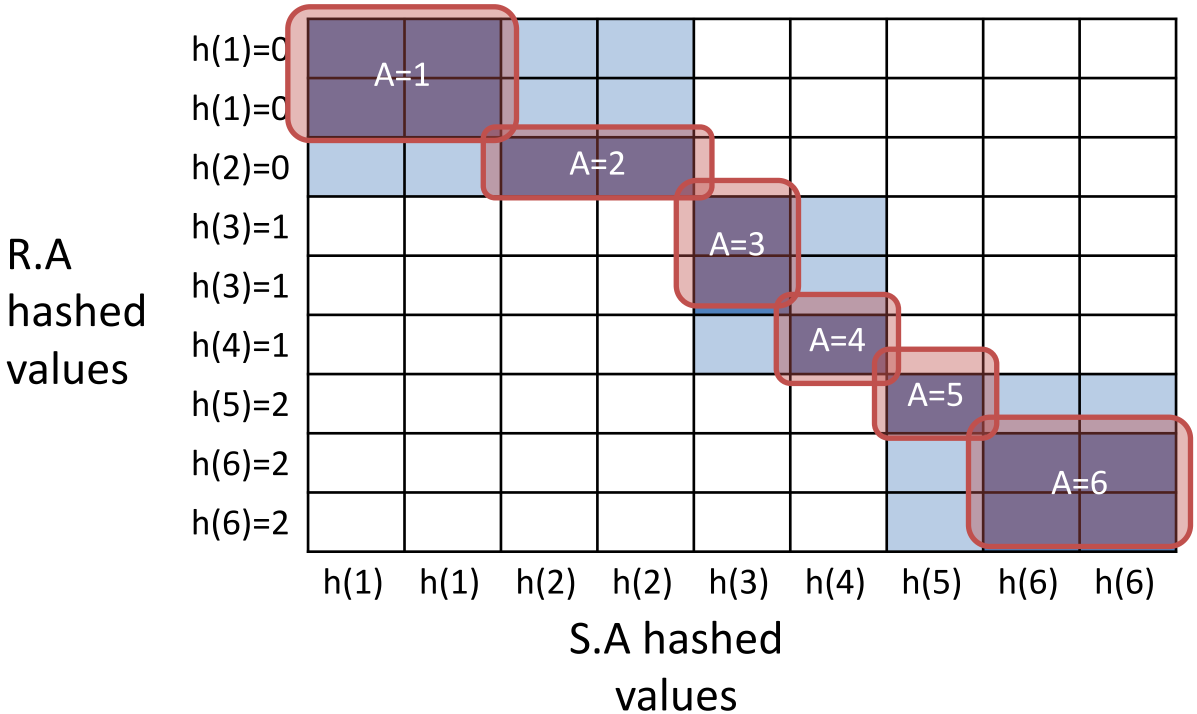
Joining the pairs of buckets is linear!
(As long as smaller bucket <= B-1 pages)

Hash Join Phase 2: Matching



$R \bowtie S\ on\ A$

Hash Join Phase 2: Matching

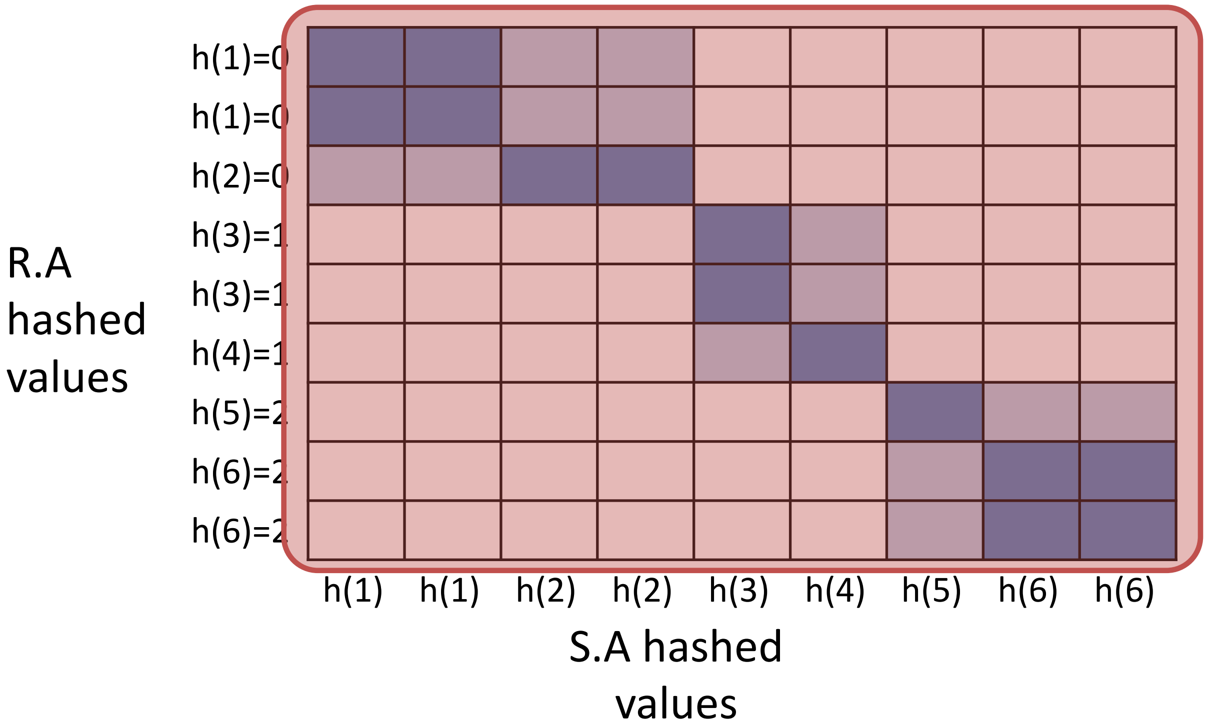


$R \bowtie S \text{ on } A$

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

= the tuples with same values of the join key A

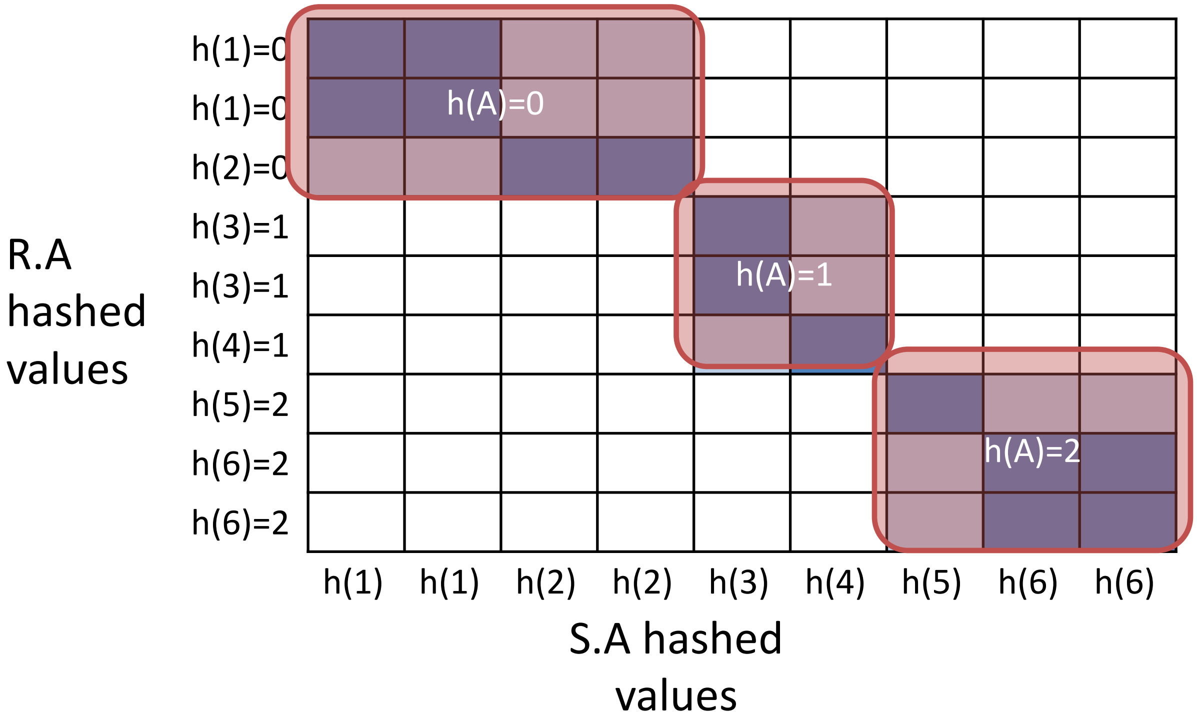
Hash Join Phase 2: Matching



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

With a join algorithm like BNLJ that doesn't take advantage of equijoin structure, we'd have to explore this ***whole grid!***

Hash Join Phase 2: Matching



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

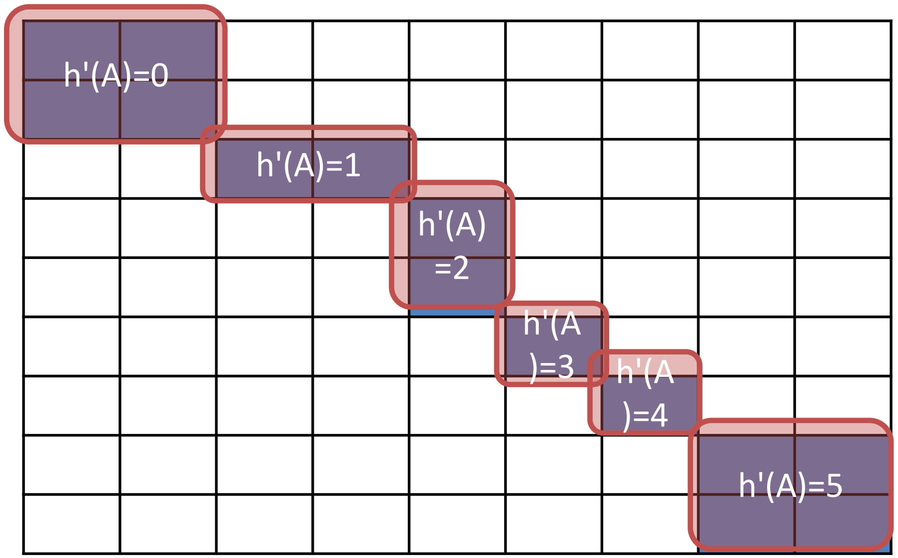
With HJ, we only explore the ***blue*** regions

*= 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!

Hash Join Summary

– **Partitioning** requires reading + writing each page of R,S
  - → 2(P(R)+P(S)) IOs

– **Matching** (with BNLJ) requires reading each page of R,S
  - → P(R) + P(S) IOs

– **Writing out results** could be as bad as P(R)*P(S)… but probably closer to P(R)+P(S)

HJ takes **~3(P(R)+P(S)) + *OUT*** IOs!

Sort-Merge vs. Hash Join

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

~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 differences in performance.

Acknowledgement

- Some of the slides in this presentation are taken from the slides provided by the authors.
- Many of these slides are taken from cs145 course offered by Stanford University.