CSC 261/461 – Database Systems
Lecture 21

Spring 2018

Announcement

- MongoDb on Bluehive:
- http://www.cs.rochester.edu/courses/261/spring2018/projects/proj3/mongodb-tutorial.html
- https://info.circ.rochester.edu/BlueHive/Software/Data_Analysis/mongodb.html

- Use the same password you use for blackboard
- Duo-authentication:
  - https://tech.rochester.edu/services/two-factor-authentication/

4. HASH JOIN (HJ)

Recall: Hashing

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

- A hash **collision** is when
- x != y but $h_B(x) = h_B(y)$

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

Hash Join: High-level procedure

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

Note again that we are only considering equality constraints here

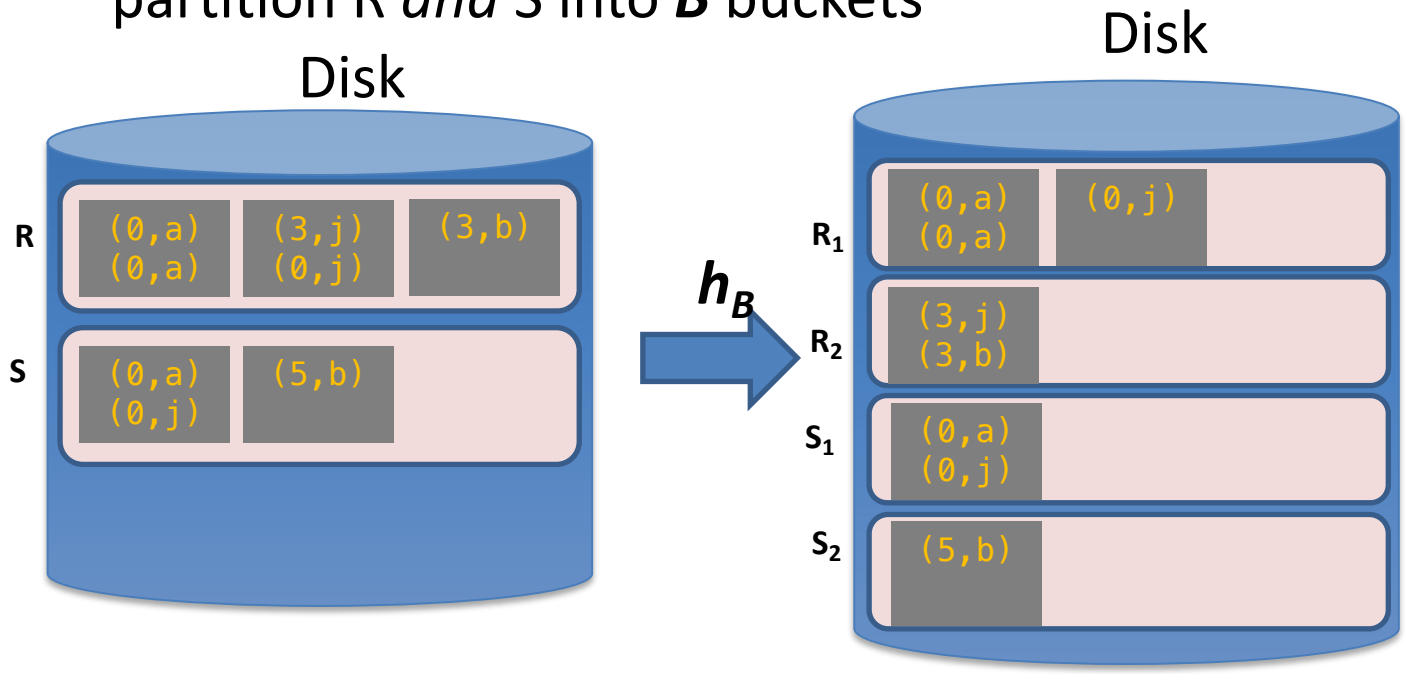
1. **Partition Phase:** Using one (shared) hash function $\boldsymbol{h_B}$, partition R *and* S into ***B*** buckets

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

Idea: 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



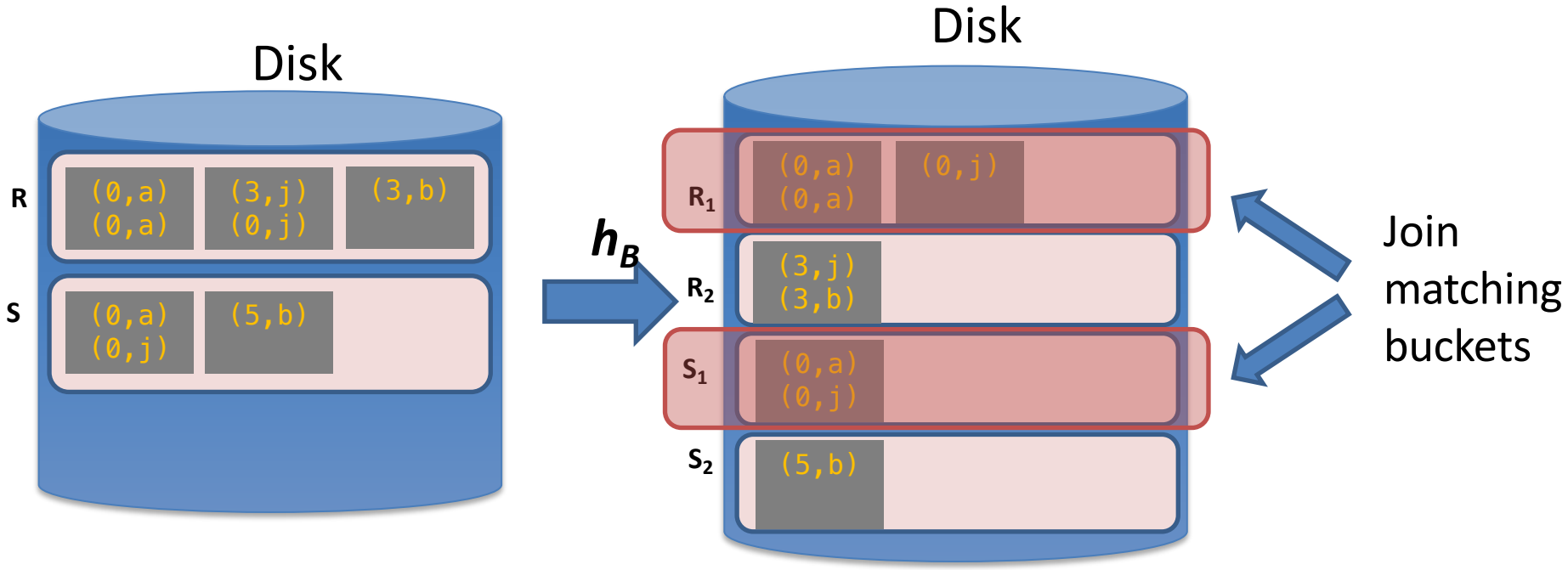
Note our new convention: pages each have two tuples (one per row)

More detail in a second…

Two buckets for each file: Even and Odd

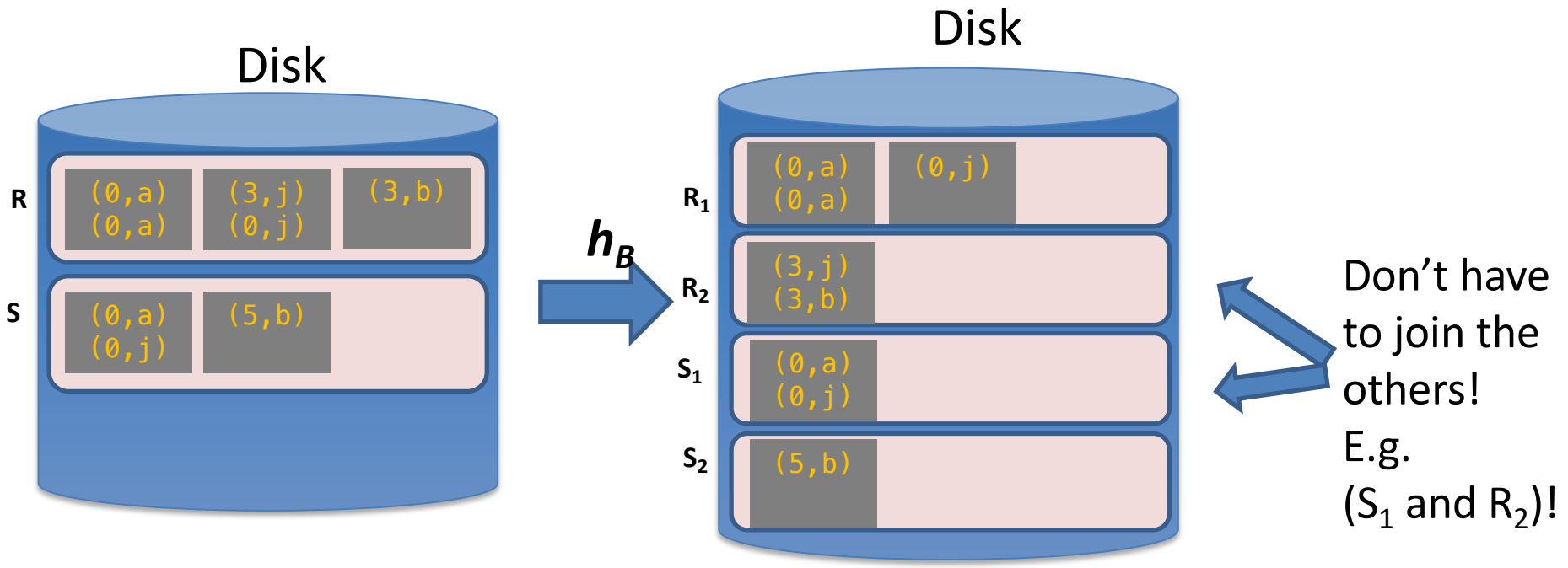
Hash Join: High-level procedure

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



Hash Join: High-level procedure

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



Hash Join Phase 1: Partitioning

Goal: For each relation, partition relation into buckets such that if $h_B(t_i.A) = h_B(t_j.A)$ they are in the same bucket

For a Relation, Two tuples/records with the same A attribute must be 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:

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

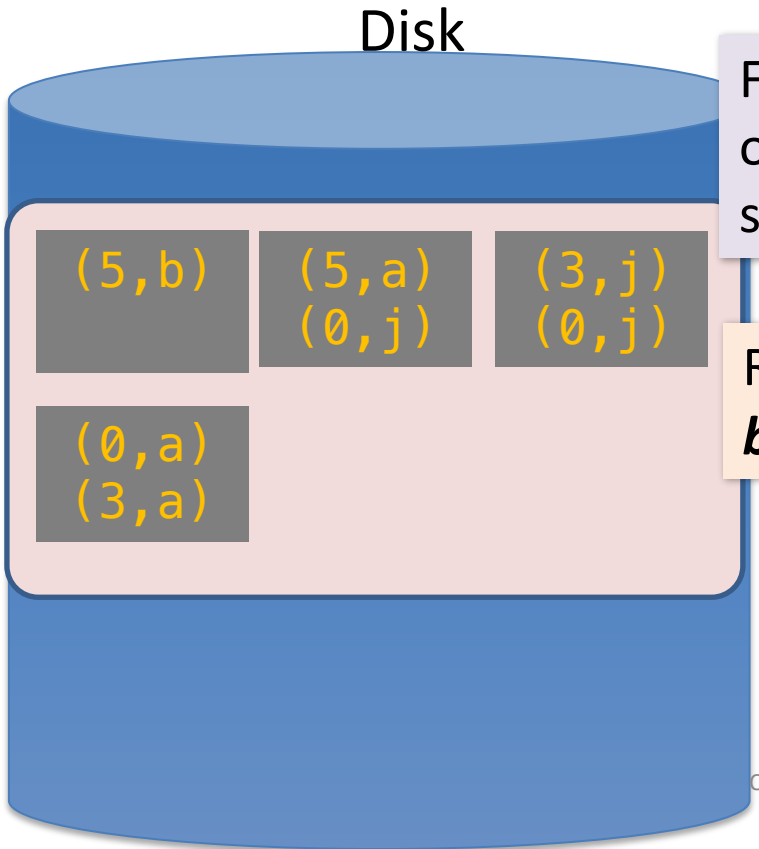
- We can keep partitioning buckets that are > B-1 pages, until they are $\leq \boldsymbol{B-1}$ **pages**
  - Using a new hash key which will split them…

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)



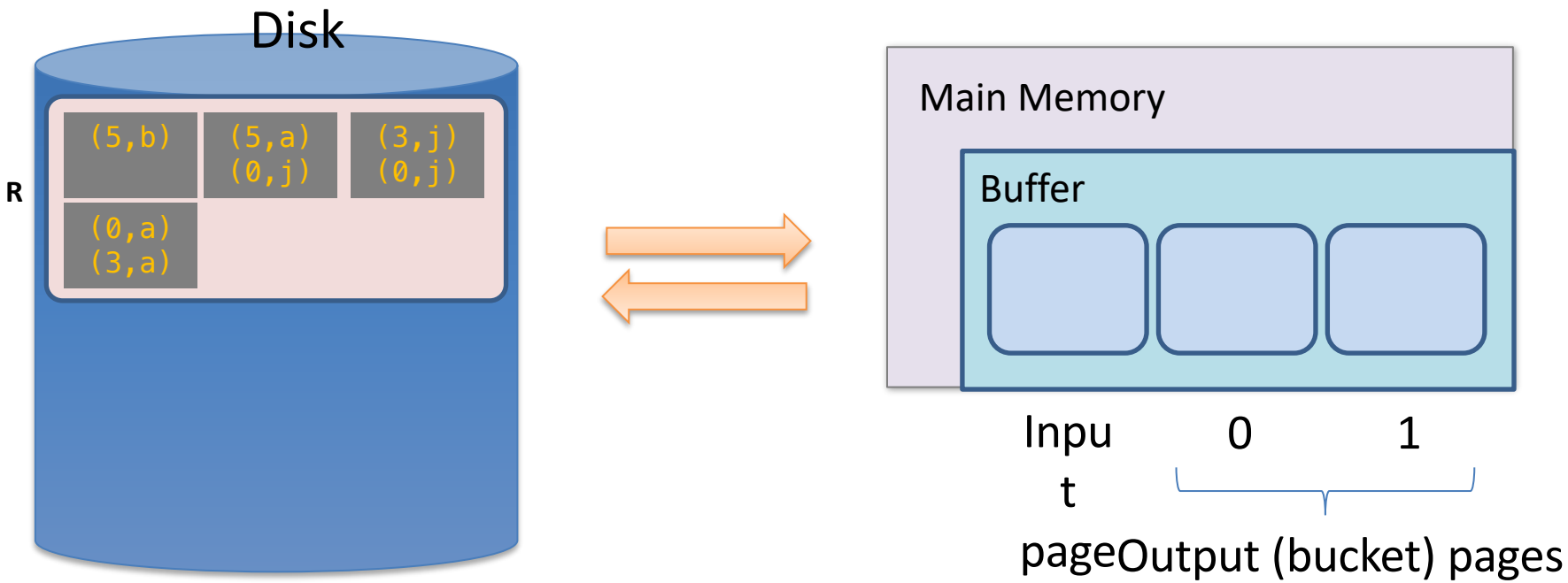
For simplicity, we'll look at partitioning one of the two relations- we just do the same for the other relation!

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

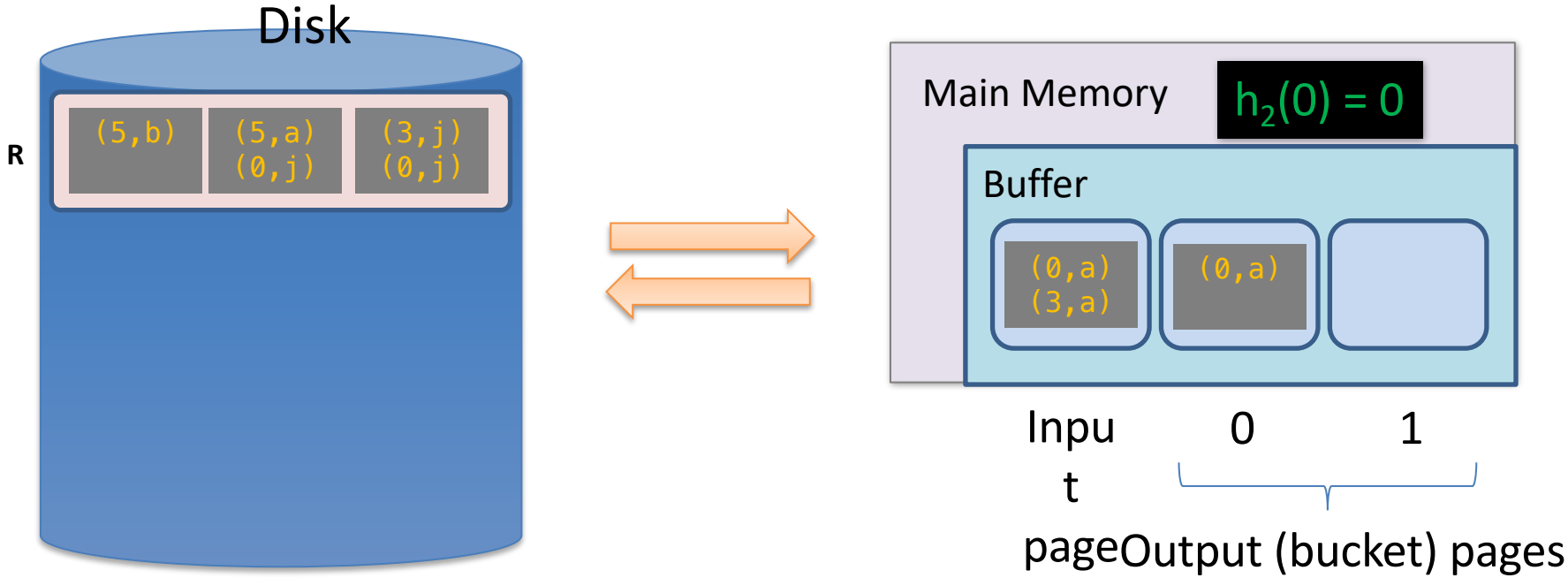
1. We read pages from R into the "input" page of the buffer…



Hash Join Phase 1: Partitioning

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

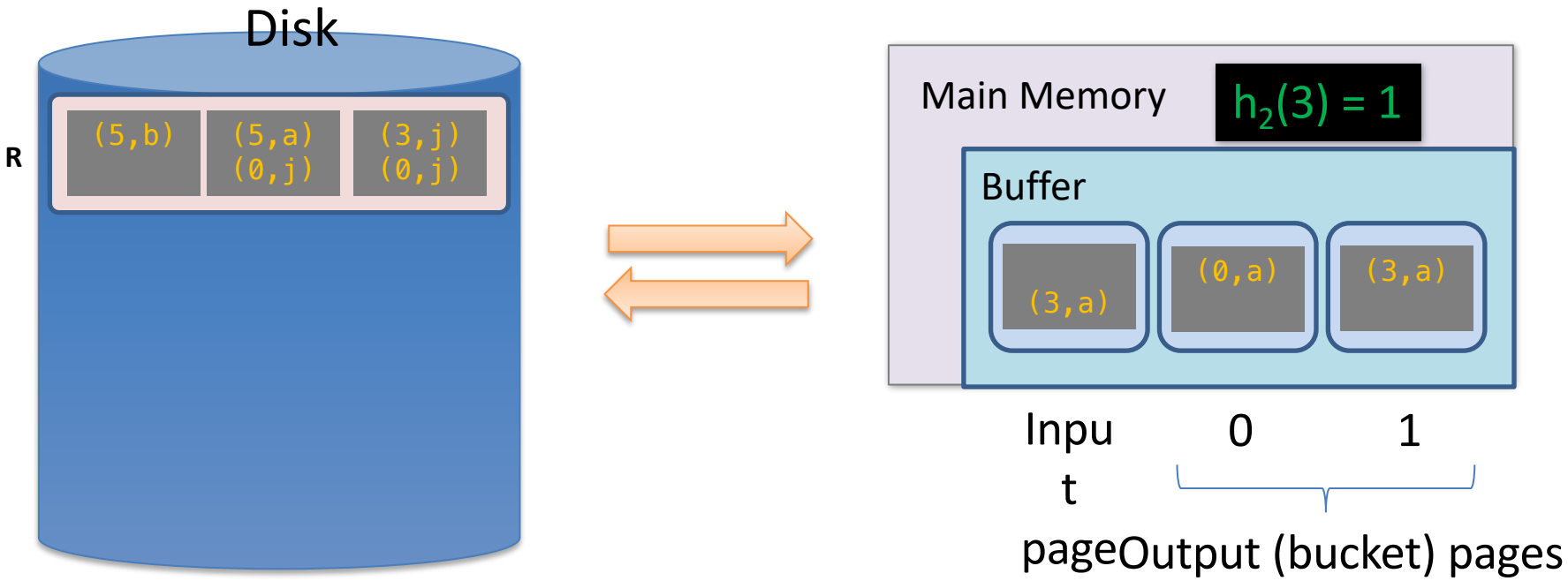
2. Then we use **hash function h_2** to sort into the buckets, which each have one page in the buffer



Hash Join Phase 1: Partitioning

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

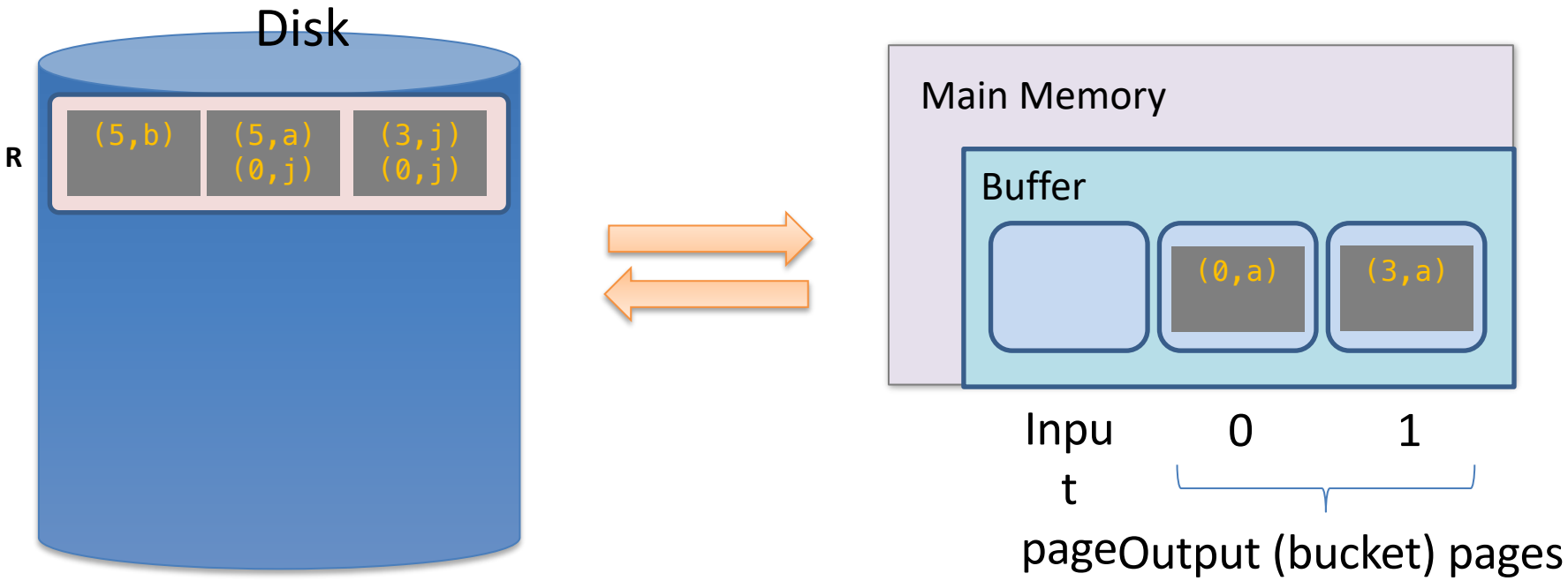
2. Then we use **hash function h_2** to sort into the buckets, which each have one page in the buffer



Hash Join Phase 1: Partitioning

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

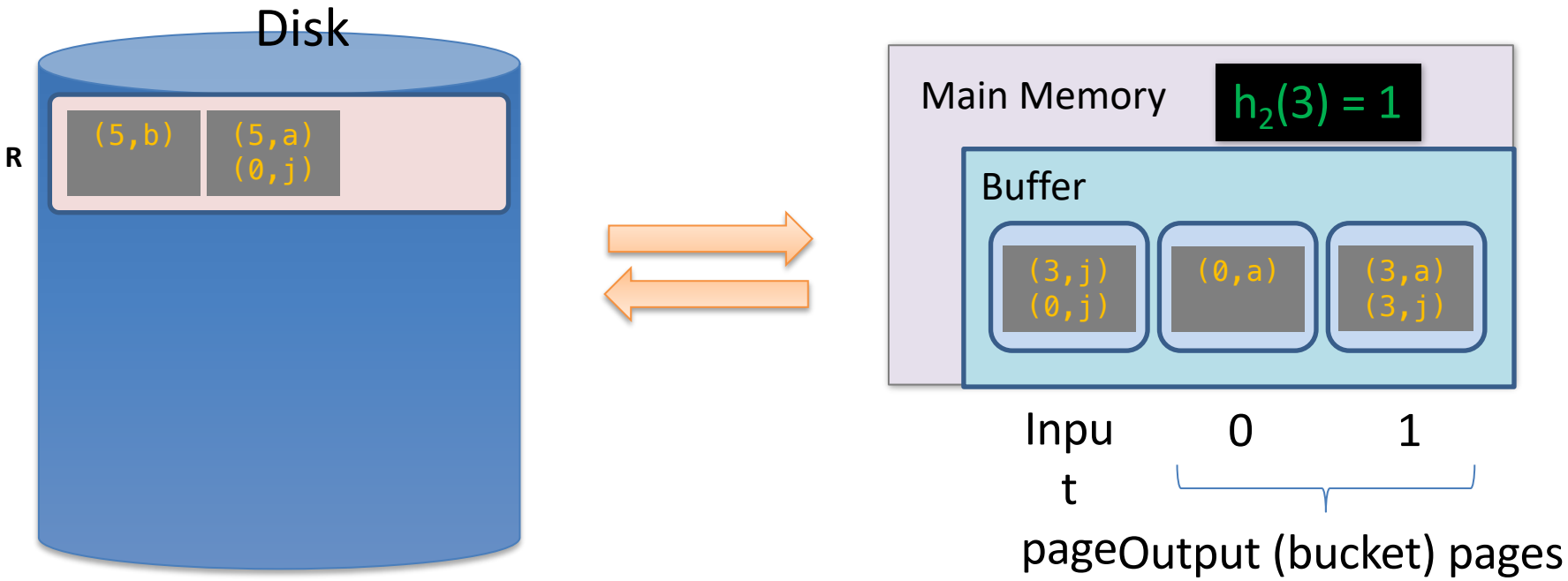
3. We repeat until the buffer bucket pages are full…



Hash Join Phase 1: Partitioning

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

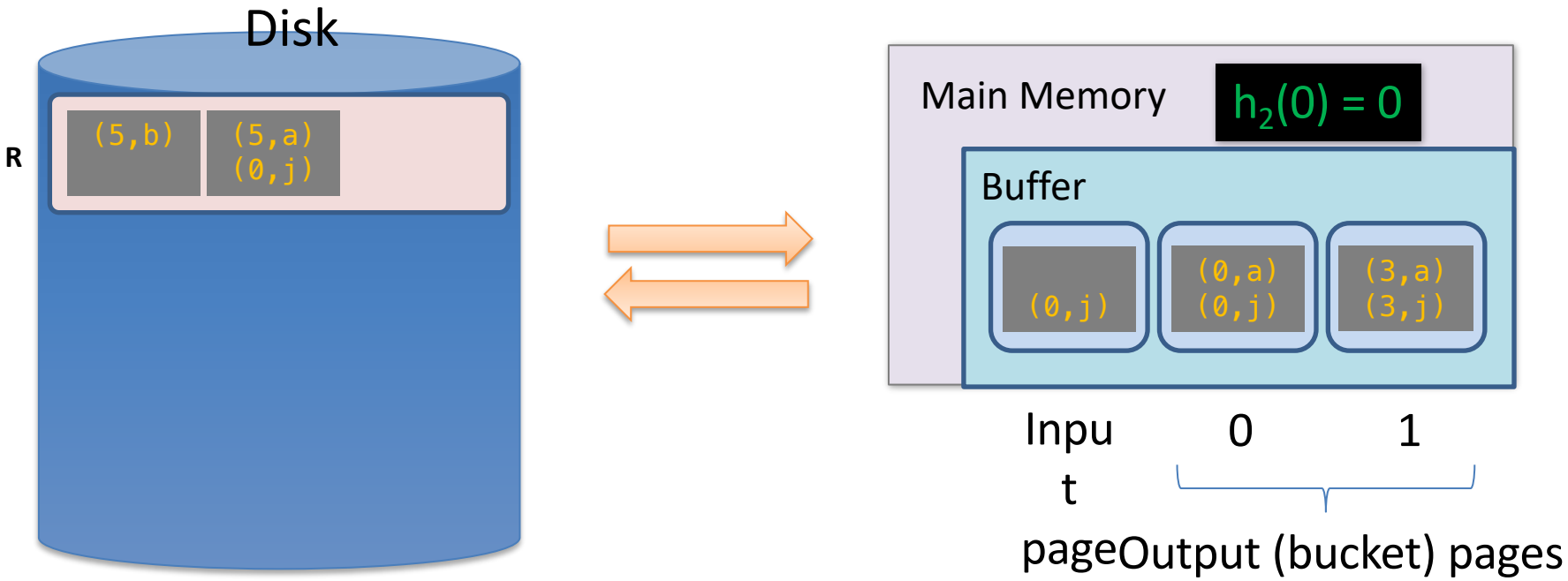
3. We repeat until the buffer bucket pages are full…



Hash Join Phase 1: Partitioning

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

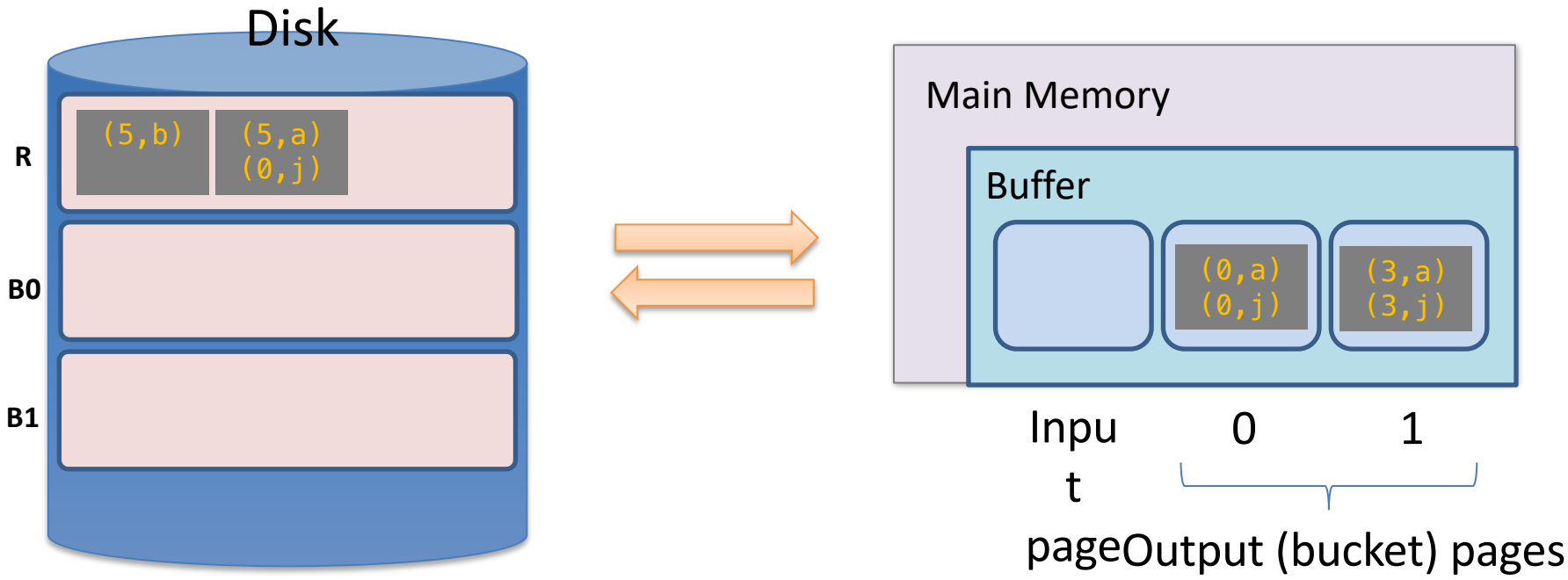
3. We repeat until the buffer bucket pages are full…



Hash Join Phase 1: Partitioning

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

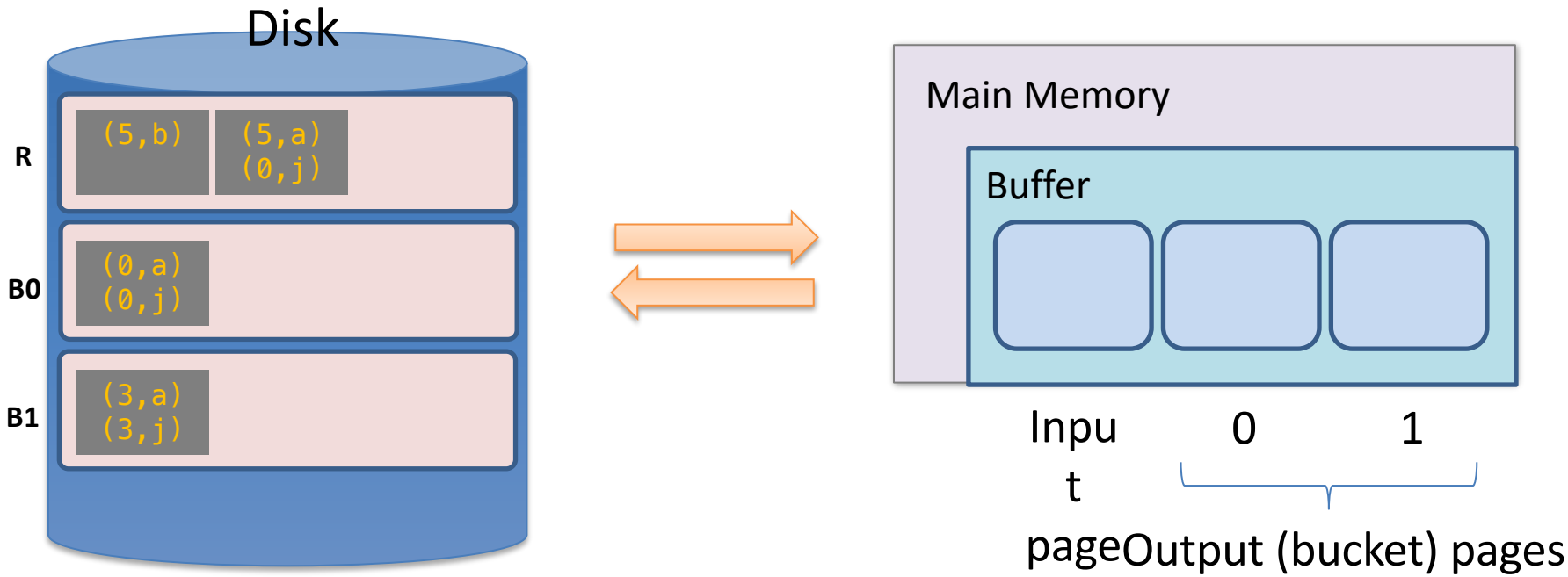
3. We repeat until the buffer bucket pages are full… then flush to disk



Hash Join Phase 1: Partitioning

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

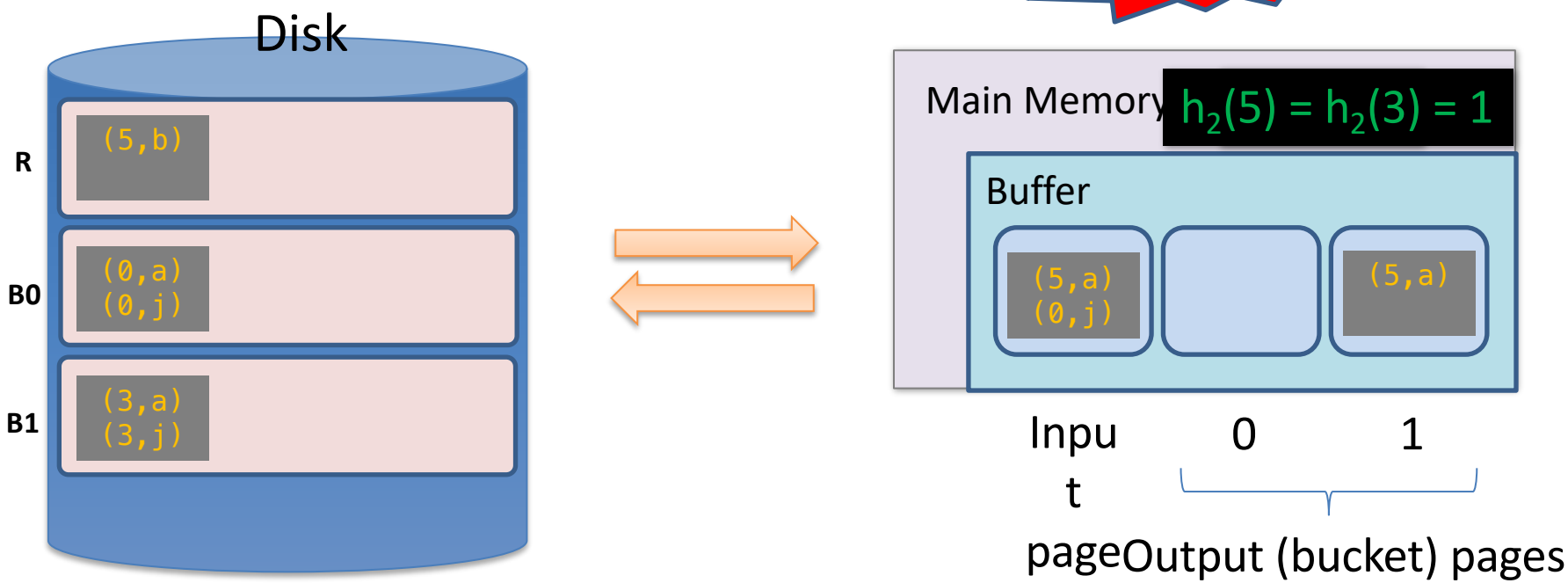
3. We repeat until the buffer bucket pages are full… then flush to disk



Hash Join Phase 1: Partitioning

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

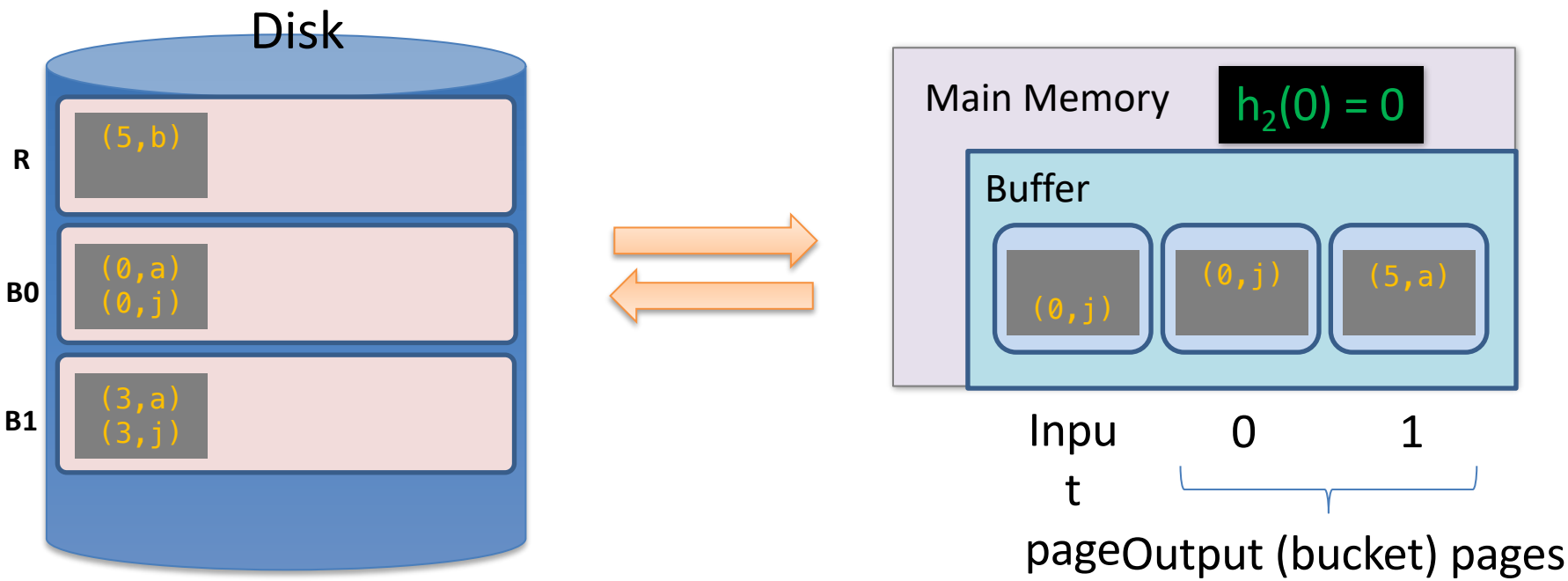
Note that collisions can occur!



Hash Join Phase 1: Partitioning

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

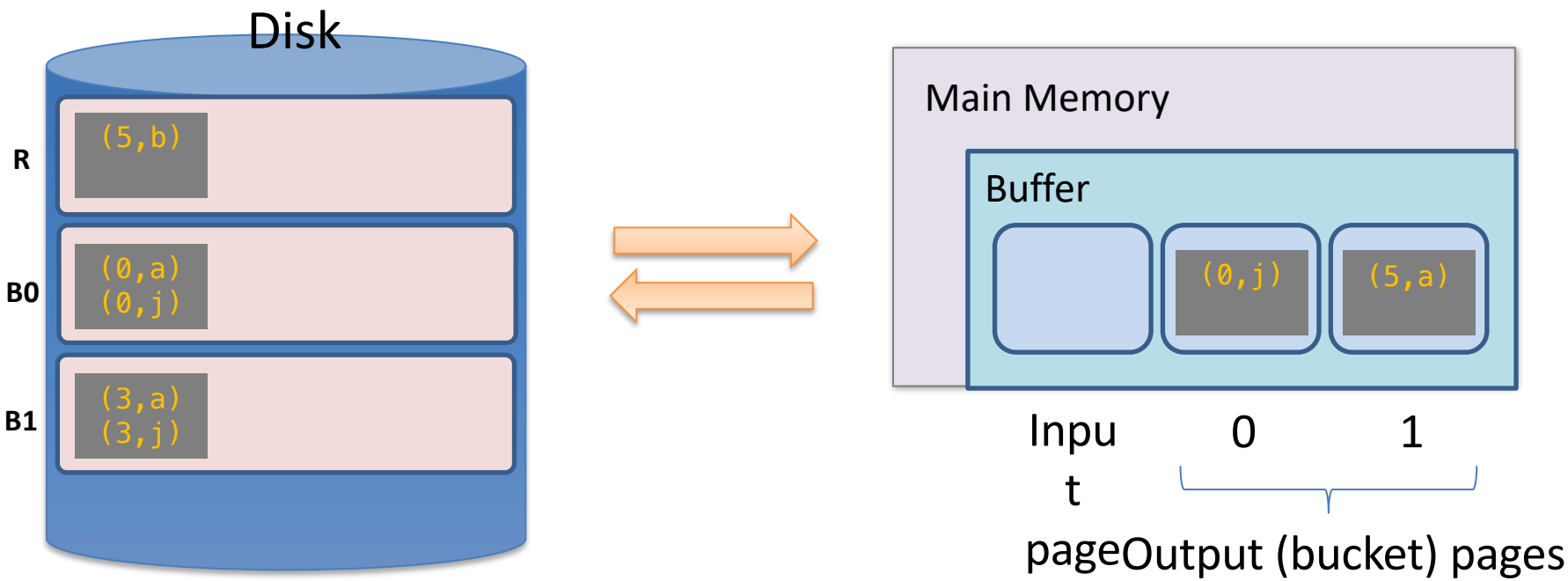
Finish this pass…



Hash Join Phase 1: Partitioning

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

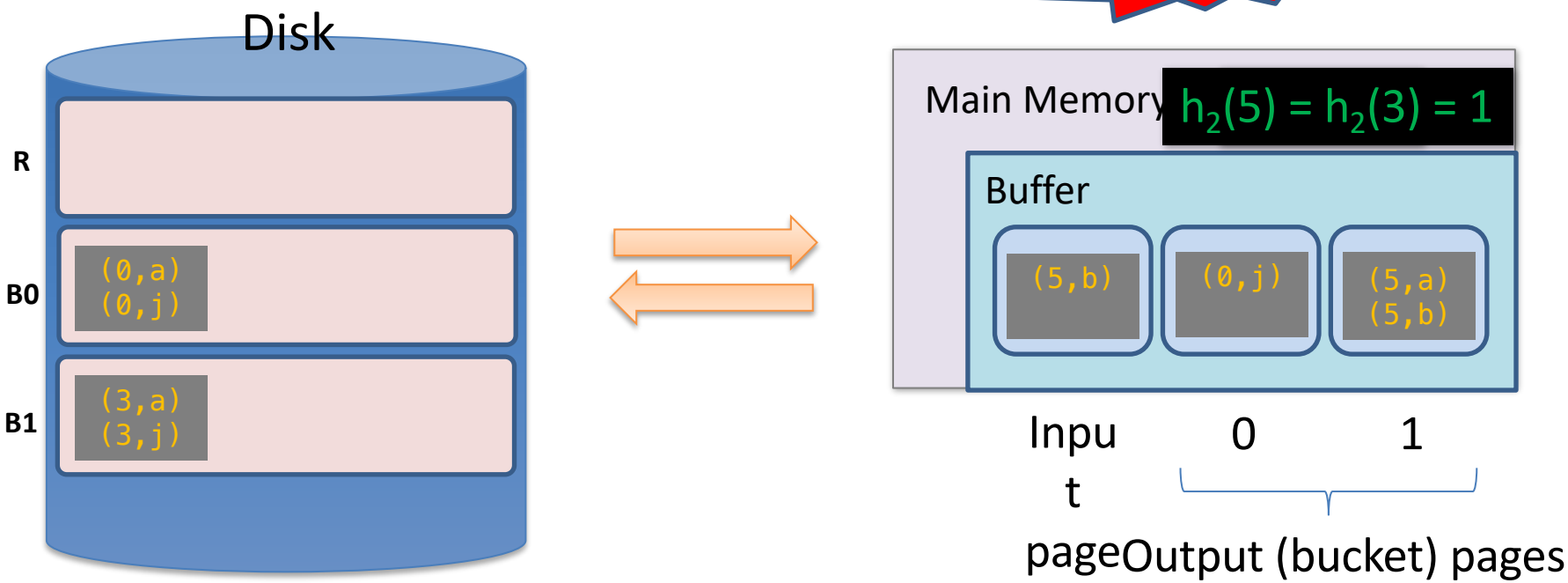
Finish this pass…



Hash Join Phase 1: Partitioning

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

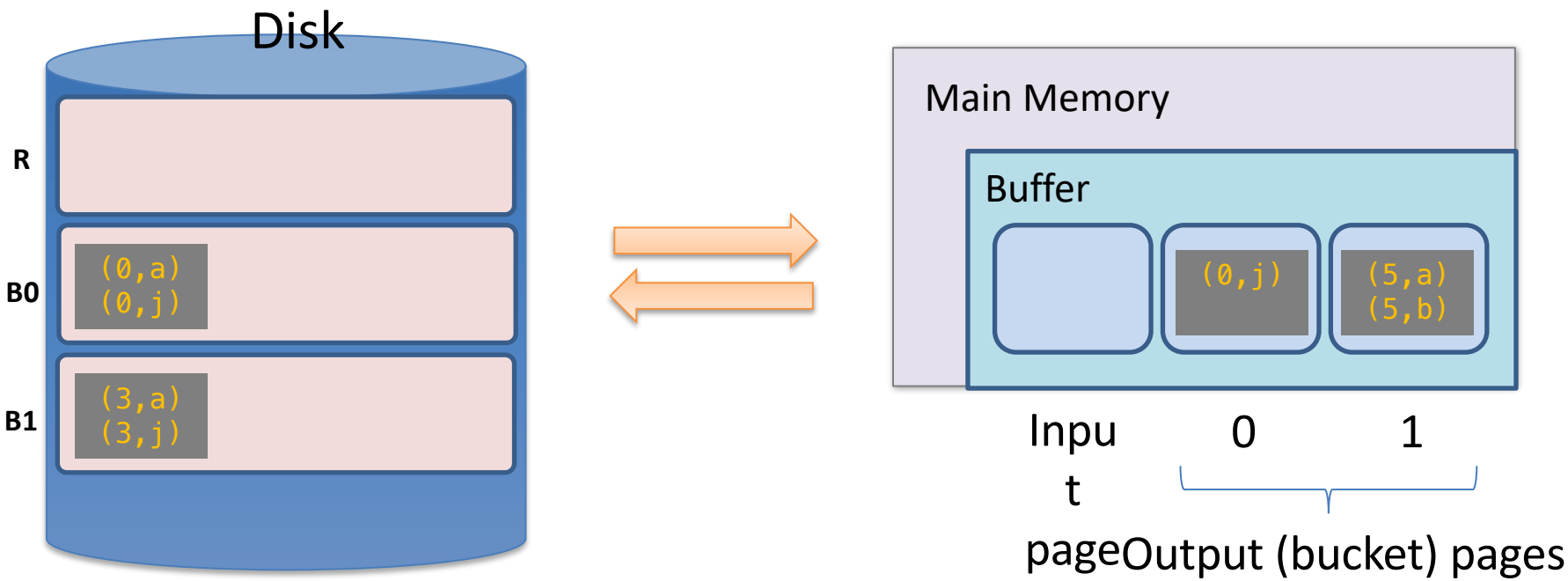
Finish this pass…



Hash Join Phase 1: Partitioning

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

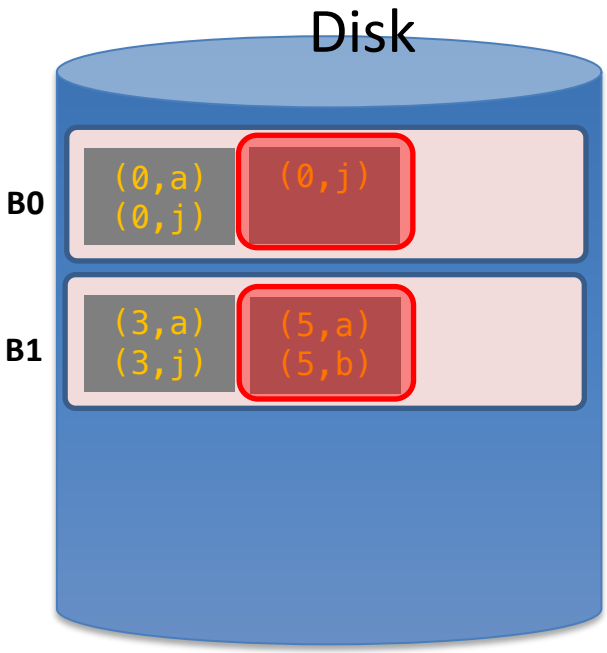
Finish this pass…



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:

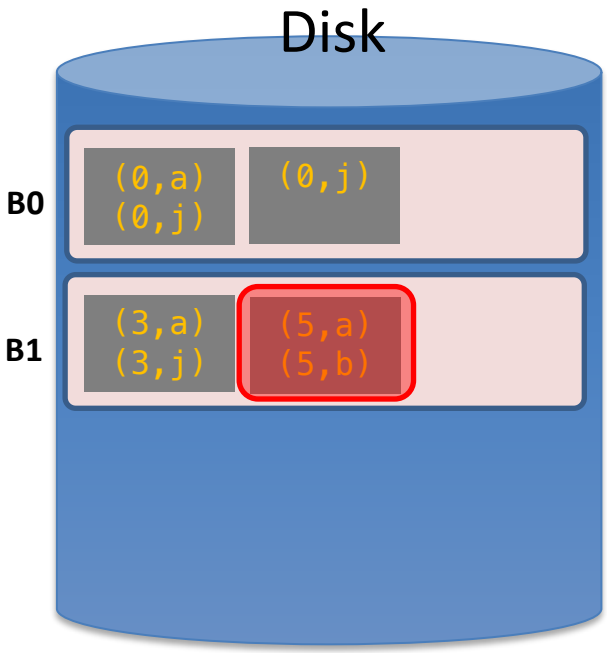


(1) Duplicate join keys

(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!

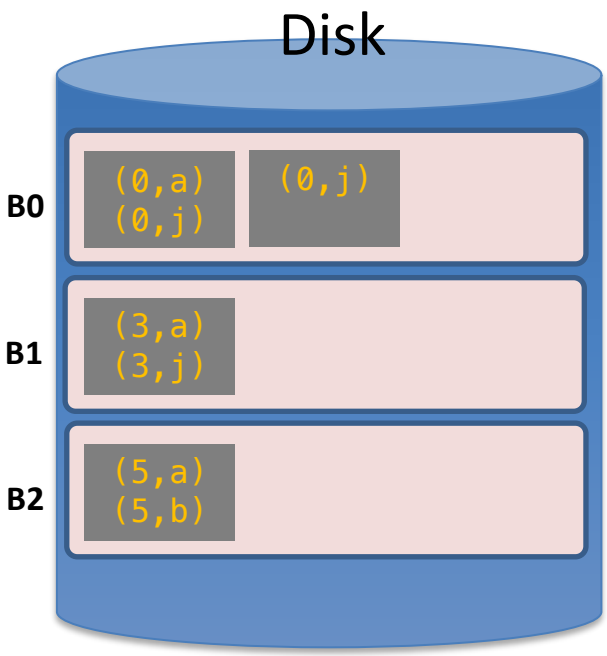
What hash function should we use?

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

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

Hash Join Phase 1: Partitioning

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



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

What hash function should we use?

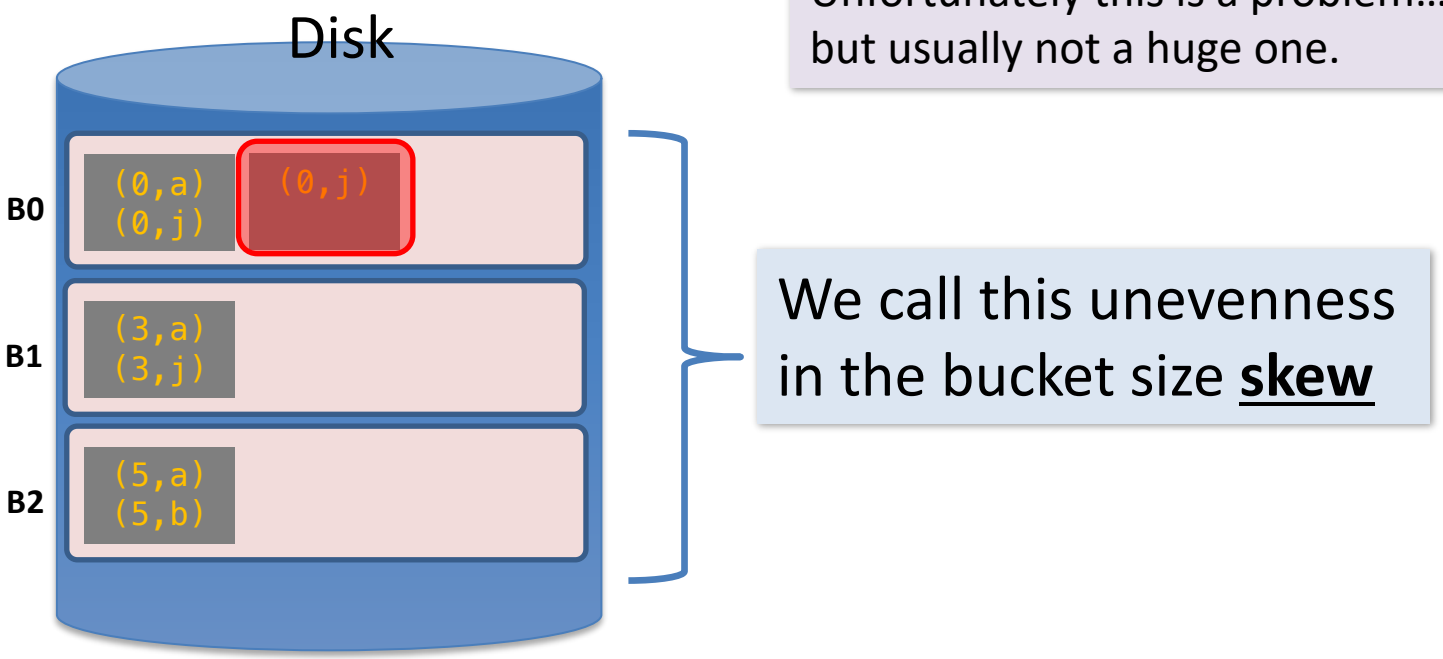
Do another pass with a different hash function, h'_2, ideally such that:

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

Hash Join Phase 1: Partitioning

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

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

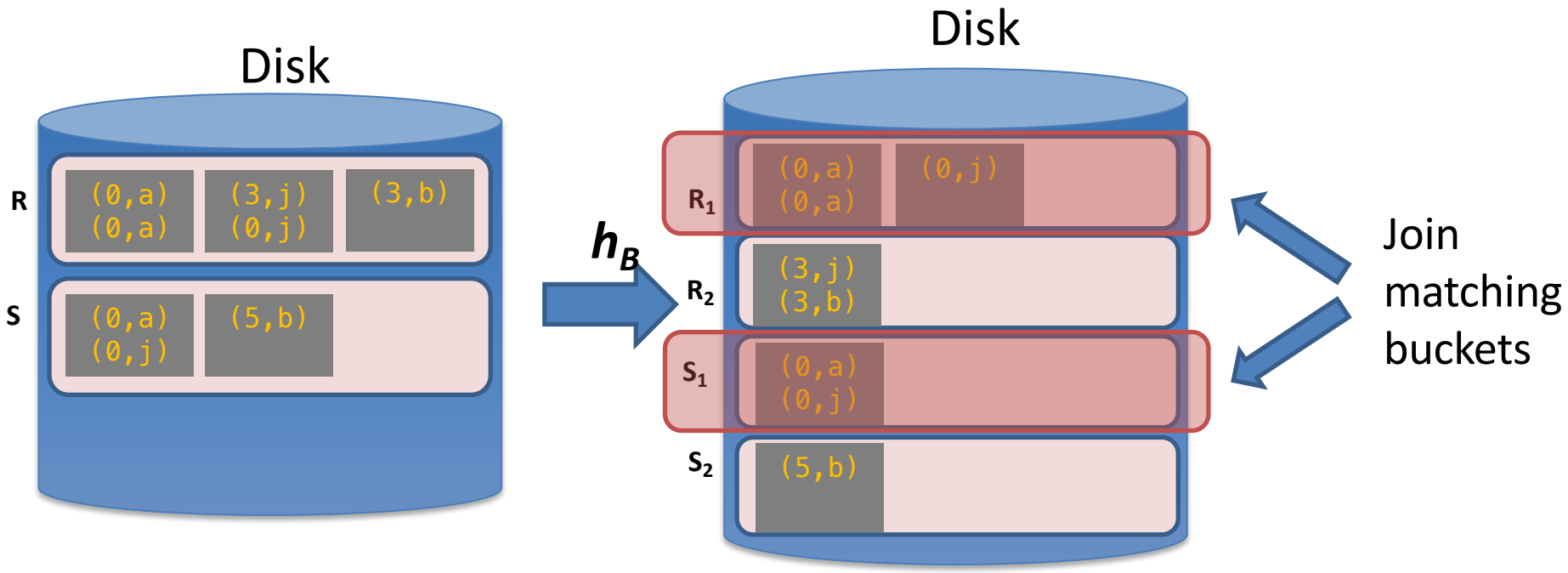


We call this unevenness in the bucket size **skew**

Now that we have partitioned R and S…

Hash Join Phase 2: Matching

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



Hash Join Phase 2: Matching

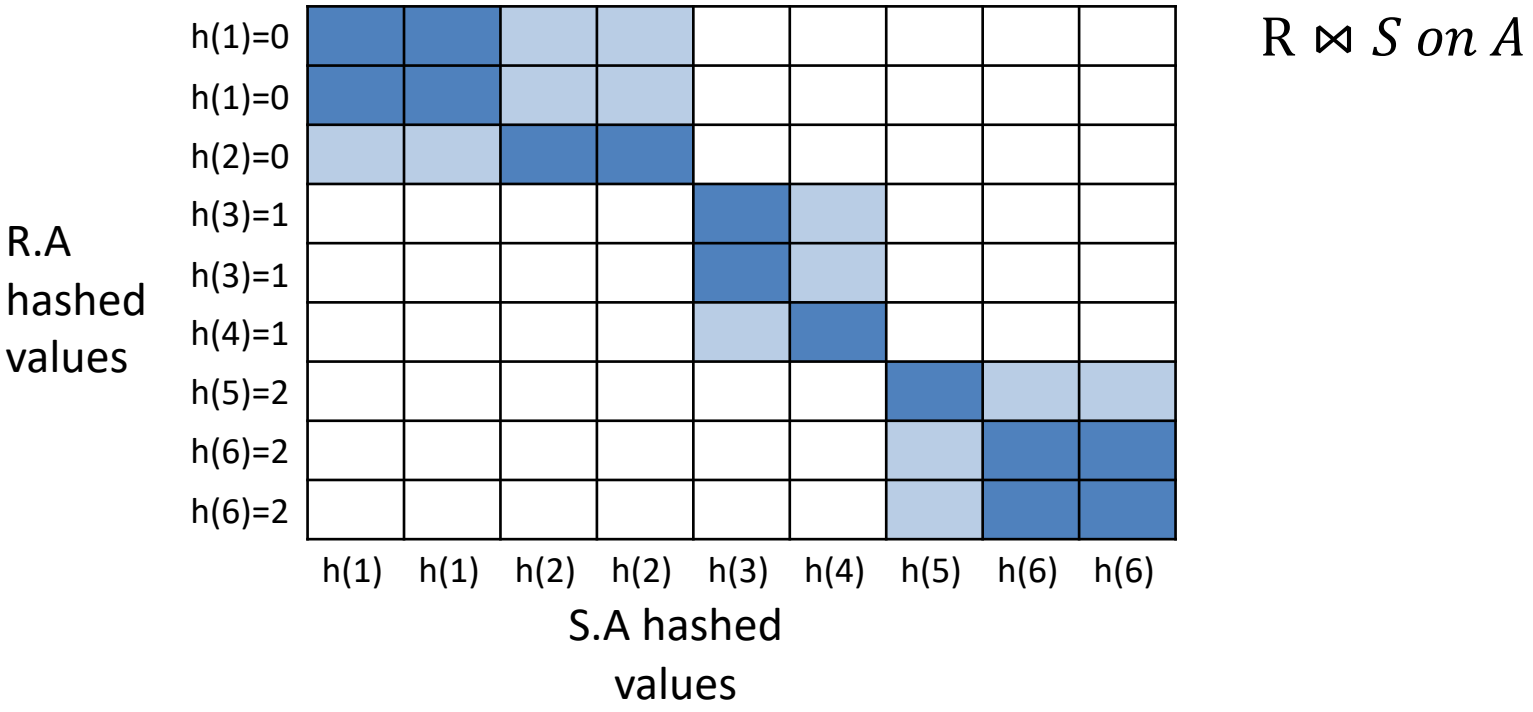
- Note that since x = y → h(x) = h(y), we only need to consider pairs of buckets (one from R, one from S) that have the same hash function value

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

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

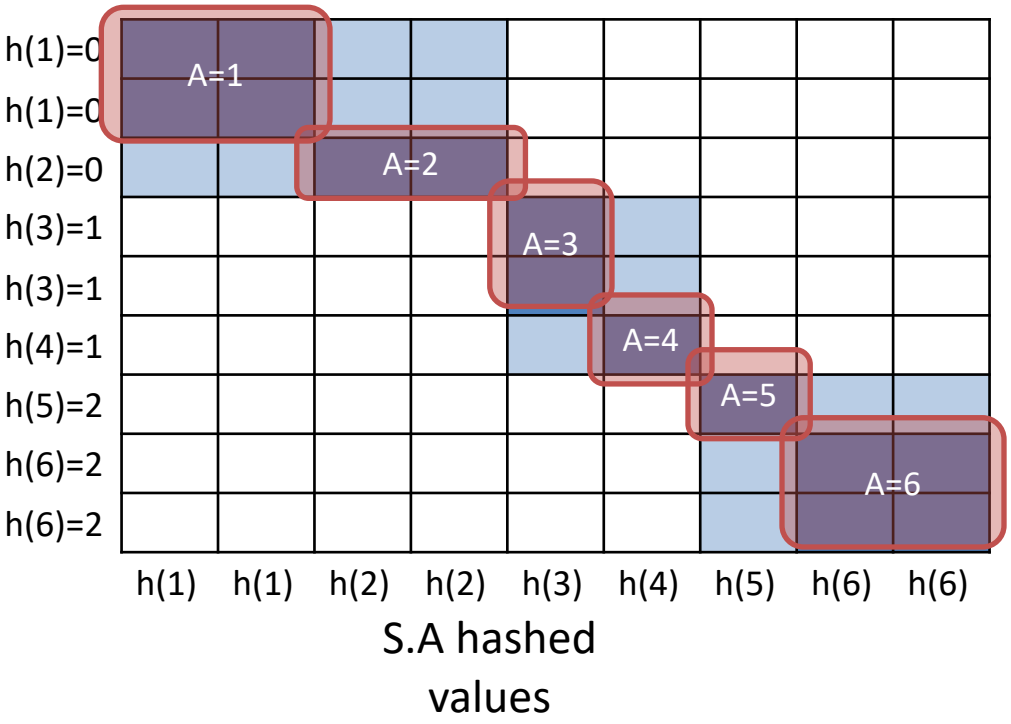
Joining the pairs of buckets is linear!
(As long as smaller bucket <= B-1 pages)

Hash Join Phase 2: Matching



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

Hash Join Phase 2: Matching

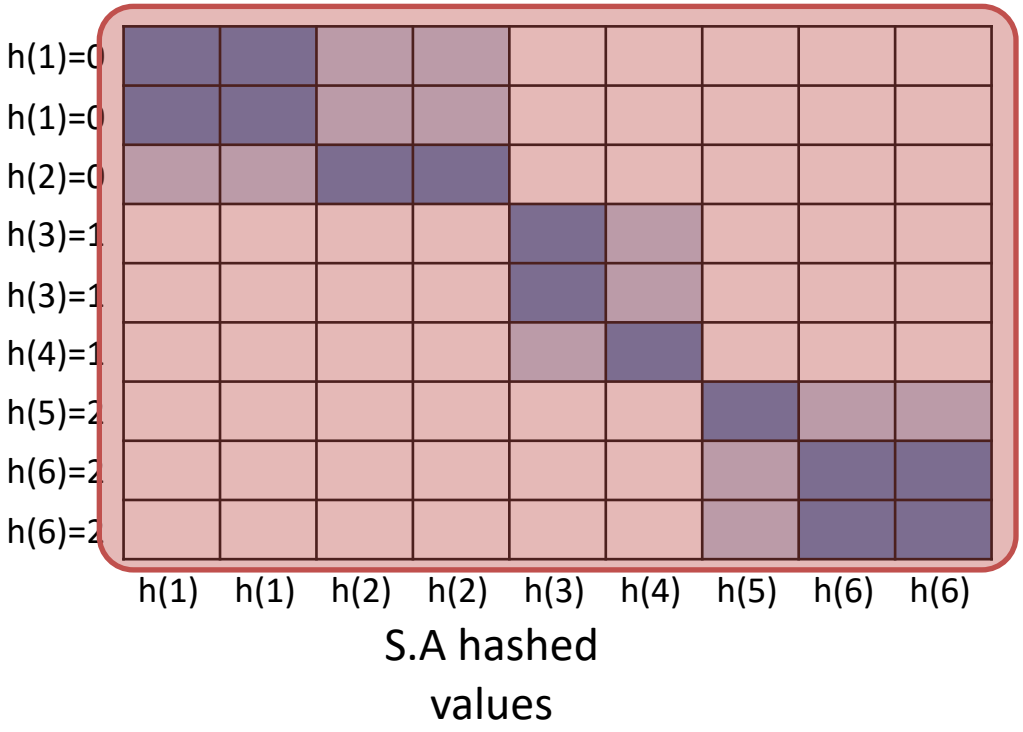


$R \bowtie S \; on \; A$

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

= the tuples with same values of the join key A

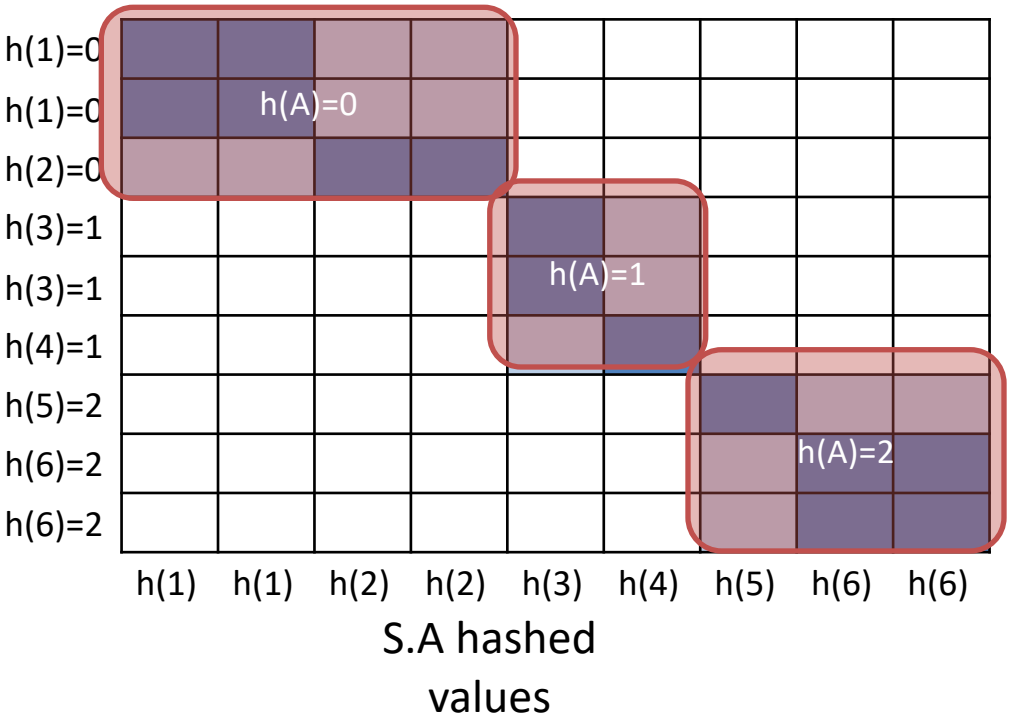
Hash Join Phase 2: Matching



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

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

Hash Join Phase 2: Matching



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

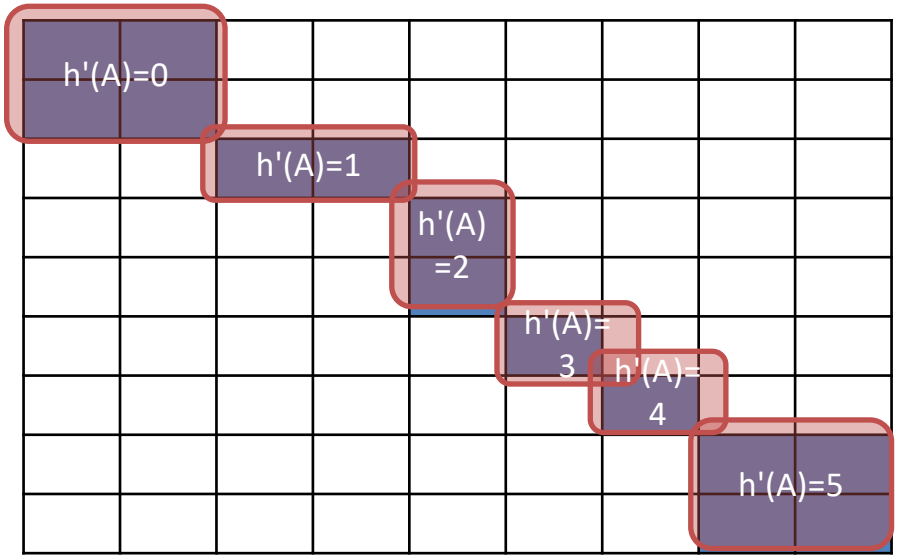
With HJ, we only explore the ***blue*** regions

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

We can apply BNLJ to each of these regions

Hash Join Phase 2: Matching

R.A hashed values



S.A hashed values

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

An alternative to applying BNLJ:

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

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*

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.

Topics for Today

- Query Optimization (Chapter 19)

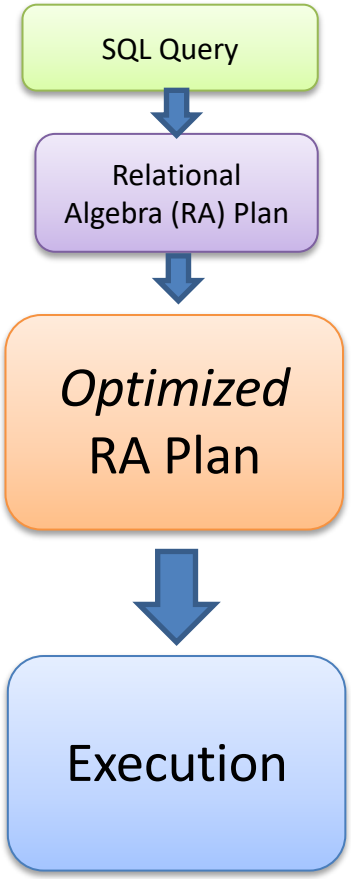
We will cover

1. Logical Optimization

2. Physical Optimization

Logical vs. Physical Optimization

- **Logical optimization:**
  - Find equivalent plans that are more efficient
  - Intuition: Minimize # of tuples at each step by changing the order of RA operators
- Physical optimization:
  - Find algorithm with lowest IO cost to execute our plan
  - Intuition: Calculate based on physical parameters (buffer size, etc.) and estimates of data size (histograms)



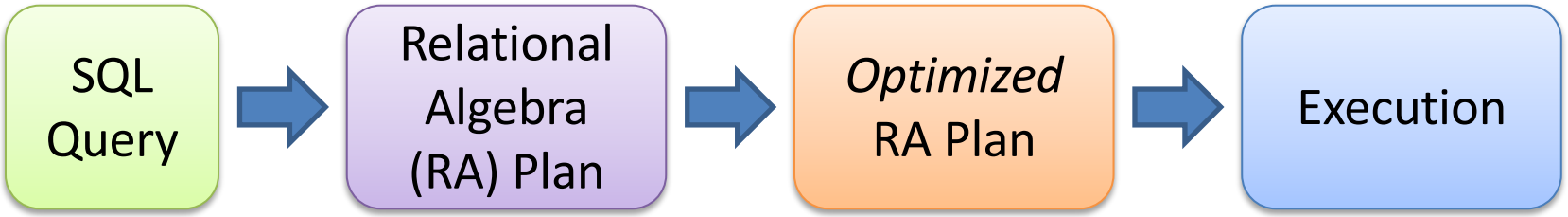
1. LOGICAL OPTIMIZATION

What you will learn about in this section

1. Optimization of RA Plans

RDBMS Architecture

How does a SQL engine work ?



Declarative query (from user)

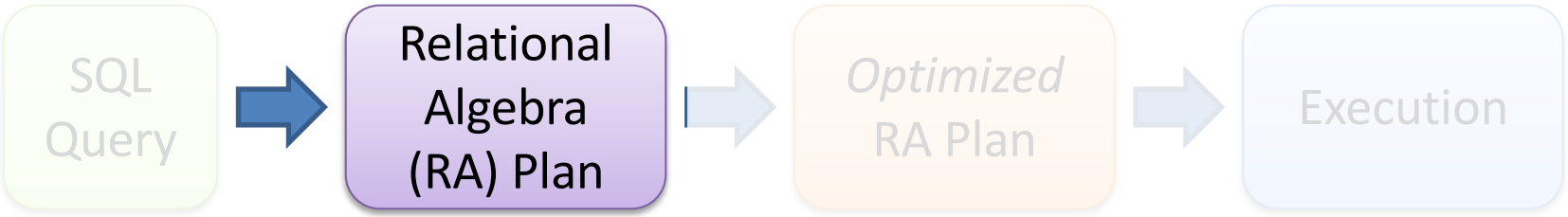
Translate to relational algebra expresson

Find logically equivalent- but more efficient- RA expression

Execute each operator of the optimized plan!

RDBMS Architecture

How does a SQL engine work ?



Relational Algebra allows us to translate declarative (SQL) queries into precise and optimizable expressions!

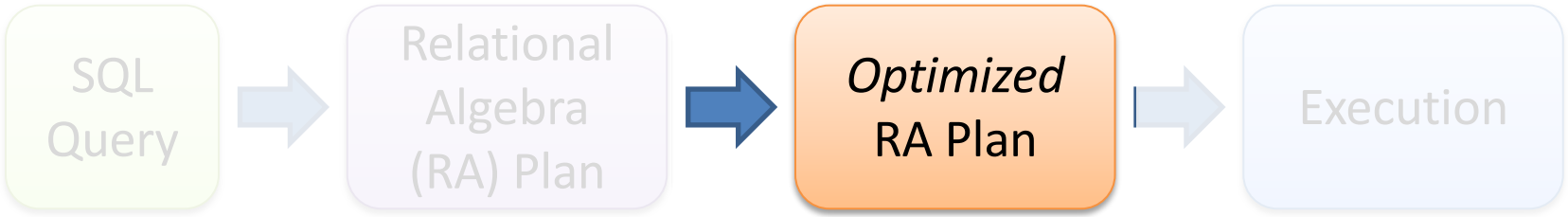
Recall: Logical Equivalence of RA Plans

- Given relations R(A,B) and S(B,C):
  - Here, projection & selection commute:
    - $\sigma_{A=5}(\Pi_A(R)) = \Pi_A(\sigma_{A=5}(R))$
  - What about here?
    - $\sigma_{A=5}(\Pi_B(R)) \stackrel{?}{=} \Pi_B(\sigma_{A=5}(R))$

We'll look at this in more depth later in the lecture…

RDBMS Architecture

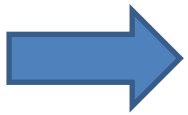
How does a SQL engine work ?

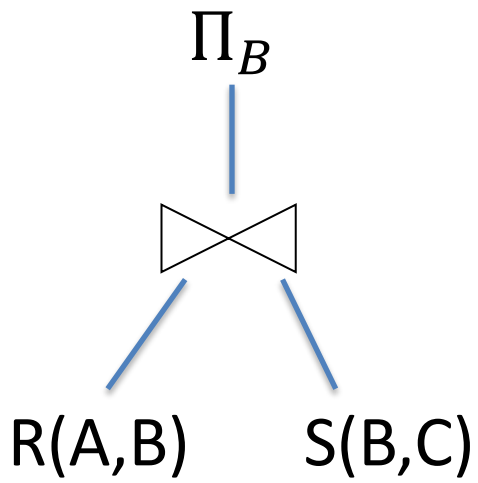


We'll look at how to then optimize these plans now

Note: We can visualize the plan as a tree

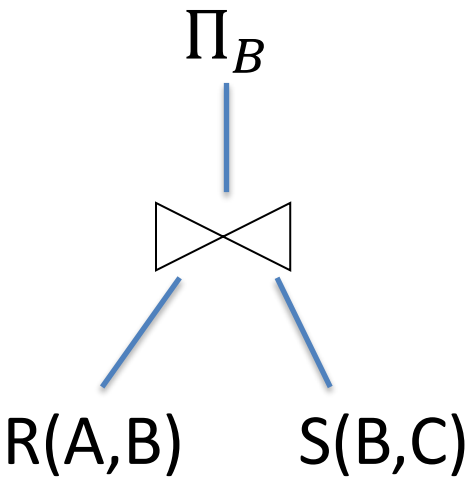
$$\Pi_B(R(A,B) \bowtie S(B,C))$$





Bottom-up tree traversal = order of operation execution!

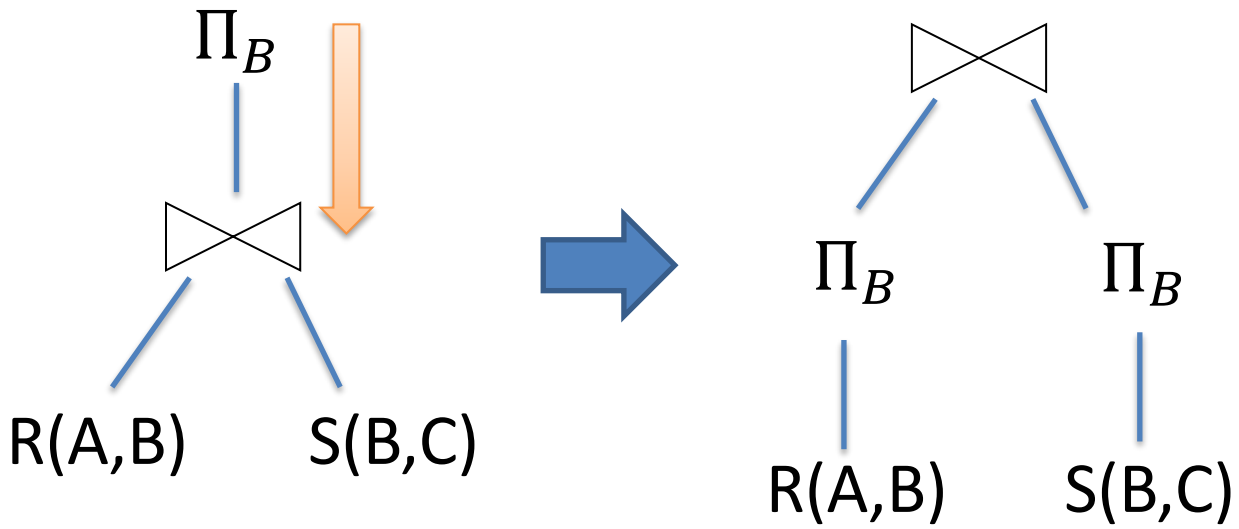
A simple plan



What SQL query does this correspond to?

Are there any logically equivalent RA expressions?

"Pushing down" projection



Why might we prefer this plan?

Takeaways

- This process is called logical optimization
- Many equivalent plans used to search for “good plans”
- Relational algebra is an important abstraction.

Optimizing the SFW RA Plan

RA commutators

- The basic commutators:
  - Push projection through **(1) selection**, **(2) join**
  - Push selection through **(3) selection, (4) projection, (5) join**
  - *Also:* Joins can be re-ordered!

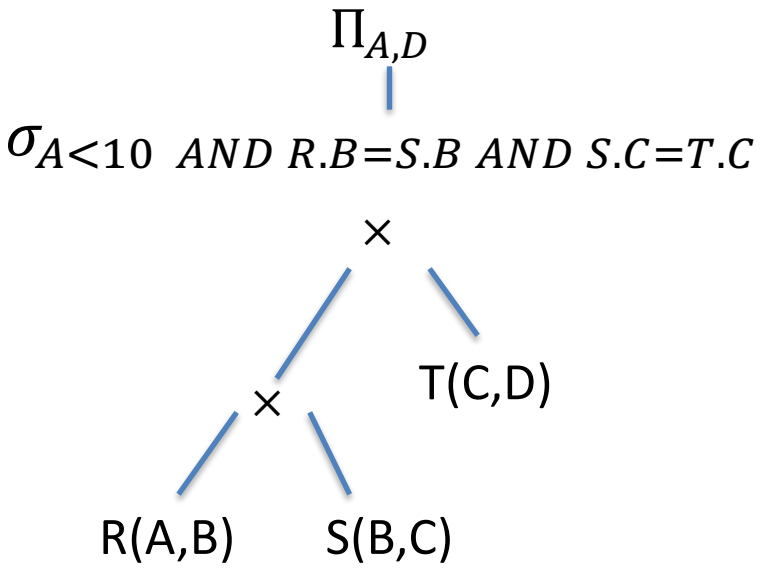
- Note that this is not an exhaustive set of operations

This simple set of tools allows us to greatly improve the execution time of queries by optimizing RA plans!

Translating to RA

```
R(A,B)  S(B,C)
T(C,D)
```

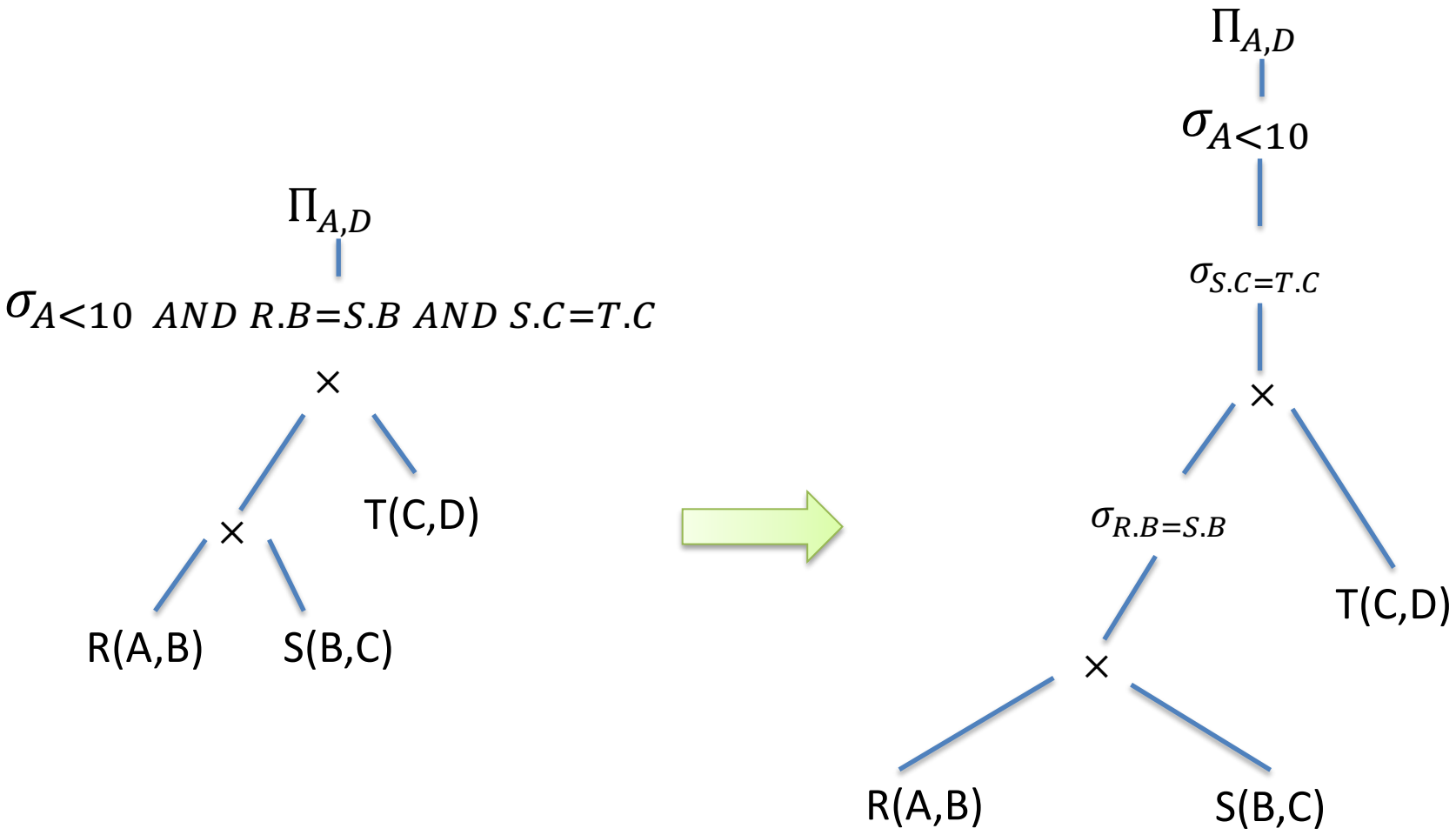
```
SELECT R.A,S.D
FROM R,S,T
WHERE R.B = S.B
  AND S.C = T.C
  AND R.A < 10;
```



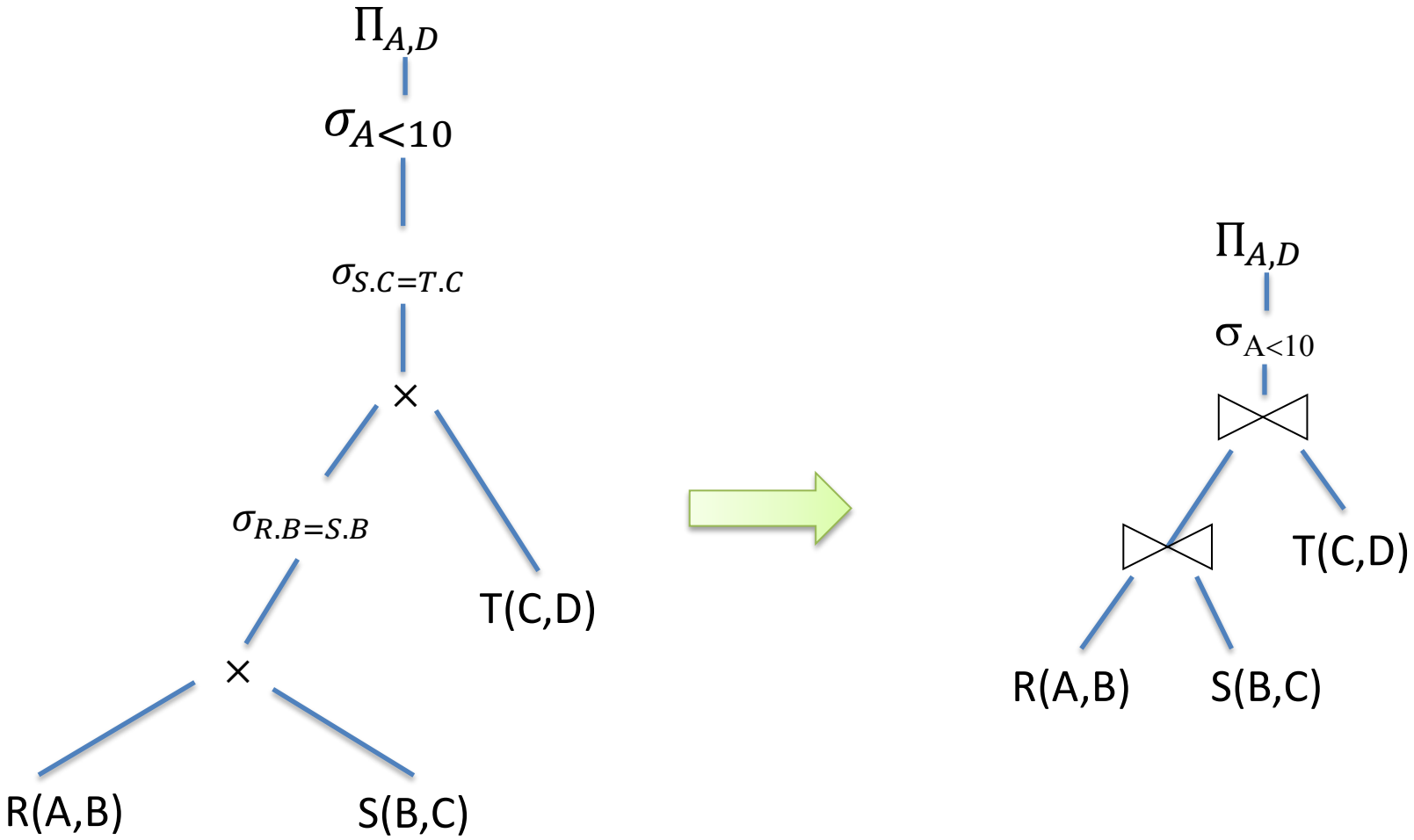
$$\Pi_{A,D}(\sigma_{A<10\ AND\ R.B=S.B\ AND\ S.C=T.C}(R \times S \times T))$$

Note: For simplicity we are not using rename operator. We will allow this format for Exams and quizzes

Translating to RA



Translating to RA



Logical Optimization

- Heuristically, we want selections and projections to occur as early as possible in the plan
  - Terminology:
    - "push down **selections**" and "push down **projections**."

- **Intuition:** We will have fewer tuples in a plan.

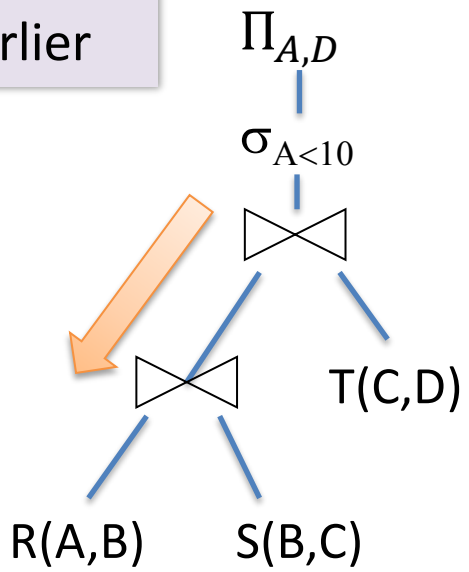
Optimizing RA Plan

```
R(A,B)  S(B,C)  T(C,D)
```

```
SELECT R.A,S.D
FROM R,S,T
WHERE R.B = S.B
  AND S.C = T.C
  AND R.A < 10;
```

Push down selection on A so it occurs earlier

$$\Pi_{A,D}(\sigma_{A<10}(T \bowtie (R \bowtie S)))$$



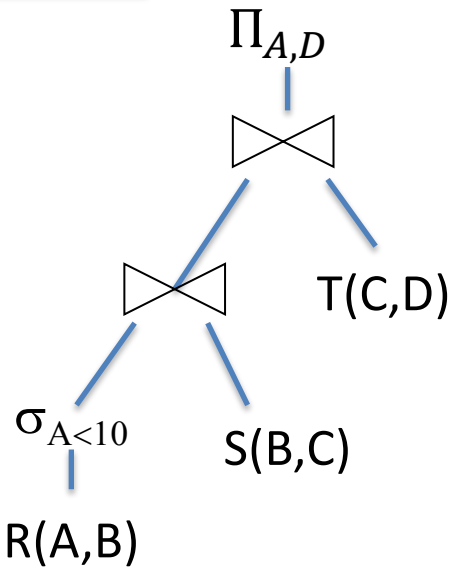
Optimizing RA Plan

```
R(A,B)  S(B,C)
T(C,D)
```

```
SELECT R.A,S.D
FROM R,S,T
WHERE R.B = S.B
  AND S.C = T.C
  AND R.A < 10;
```

Push down selection on A so it occurs earlier

$$\Pi_{A,D}\left(T \bowtie \left(\sigma_{A<10}(R) \bowtie S\right)\right)$$



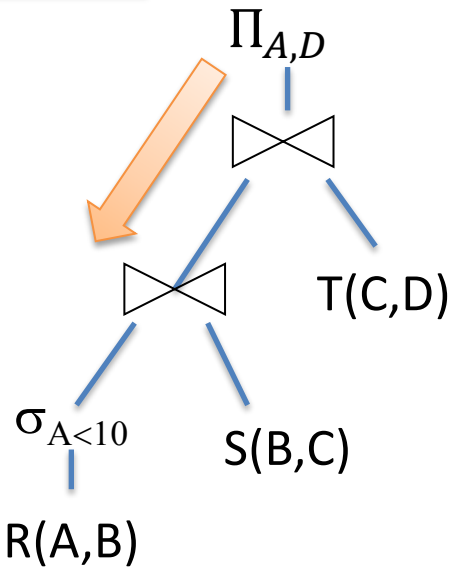
Optimizing RA Plan

```
R(A,B)  S(B,C)
T(C,D)
```

```
SELECT R.A,S.D
FROM R,S,T
WHERE R.B = S.B
  AND S.C = T.C
  AND R.A < 10;
```

Push down projection so it occurs earlier

$$\Pi_{A,D}\left(T \bowtie \left(\sigma_{A<10}(R) \bowtie S\right)\right)$$



Optimizing RA Plan

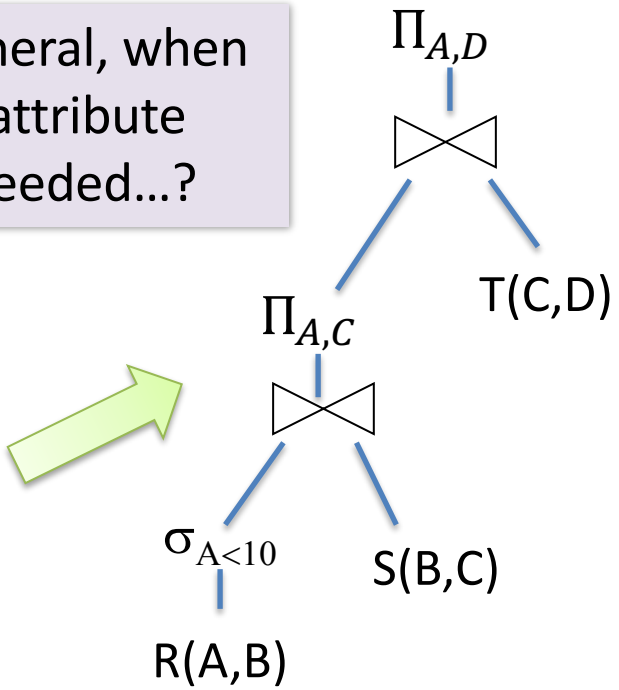
```
R(A,B)  S(B,C)
T(C,D)
```

```
SELECT R.A,S.D
FROM R,S,T
WHERE R.B = S.B
  AND S.C = T.C
  AND R.A < 10;
```

We eliminate B earlier!

In general, when is an attribute not needed…?

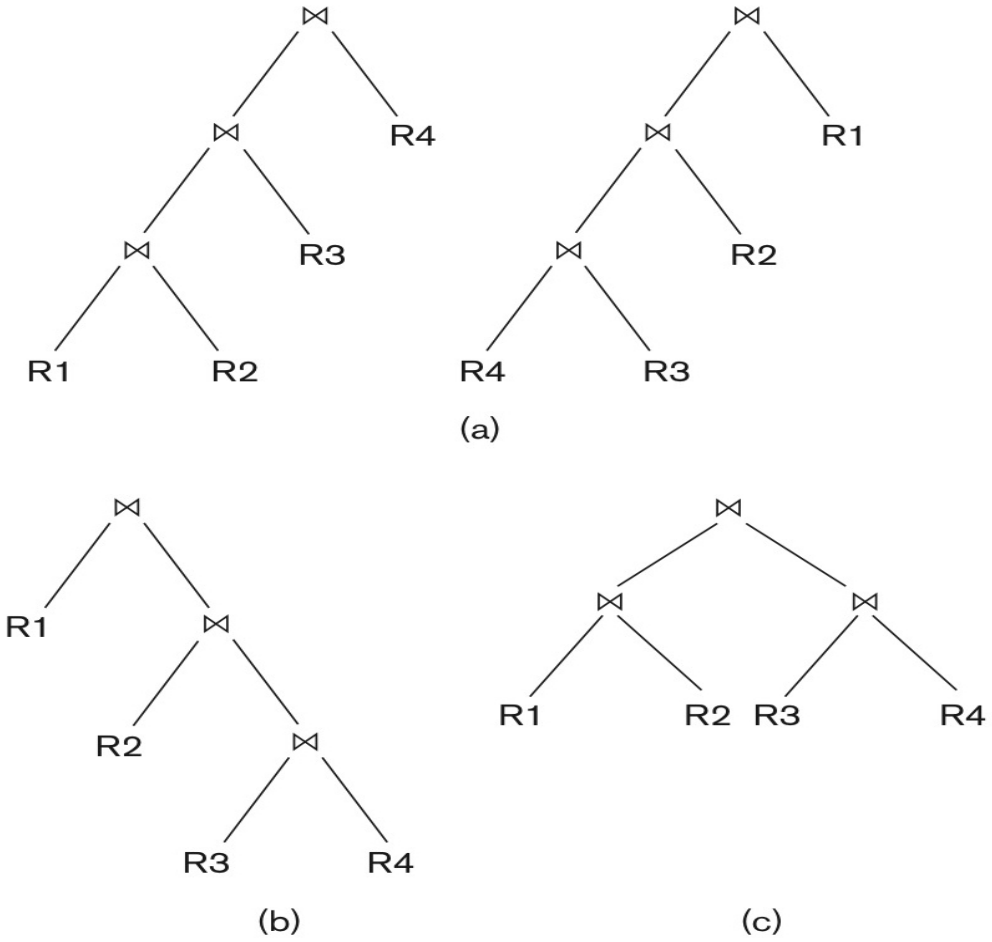
$$\Pi_{A,D}\left(T \bowtie \Pi_{A,C}(\sigma_{A<10}(R) \bowtie S)\right)$$



Logical optimization

- Selections and Cross Product can be combined into joins
- Selections and projections can be pushed down (below joins)
- Joins can be extensively reordered.

Equivalent Trees



(a) Two left-deep join query trees. (b) A right-deep join query tree. (c) A bushy query tree.

How to Handle Many Joins

Table19.1 Number of Permutations of Left-Deep and Bushy Join Trees of *n* Relations

No. of Relations *N*	No. of Left-Deep Trees *N*!	No. of Bushy Shapes *S*(*N*)	No. of Bushy Trees (2*N* – 2)!/(*N* – 1)!
2	2	1	2
3	6	2	12
4	24	5	120
5	120	14	1,680
6	720	42	30,240
7	5,040	132	665,280

Reasons for Left-deep plans

- As the number of joins increases, the number of alternative plans increases rapidly. It becomes necessary to prune the space of alternative plans

- Left-deep trees allow us to fully pipelined plans.

2. PHYSICAL OPTIMIZATION

Cost functions for SELECT operation

- Teminology:
  - r: Number of records
  - b: Number of blocks
  - bfr: blocking factor
  - sl: selectivity (fraction of record satisfying the condition)
  - s: selection cardinality = sl * r = number of records satisfying the condition
  - x: Number of levels (you can treat as depth)
  - l : number of leaves (# number of first level blocks)

Cost functions for Select (Section: 19.4)

Algorithm	Cost	Special Cases
Linear Search	b/2	b (if not found)
Binary Search	$\log_2 b + \left\lceil \left(\frac{s}{bfr} \right) \right\rceil - 1$	$\log_2 b$ (if on a unique key)
Primary Index	$x + 1$	
Hash Key	1	
Ordering index (>, < , >=, <=)	$x + (b/2)$	
Clustering Index	$x + \left\lceil \left(\frac{s}{bfr} \right) \right\rceil$	
B+ tree index	$x + 1 + s$	$x + l/2 + r/2$

What you will learn about in this section

1. Index Selection

2. Histograms

Index Selection

Input:

- Schema of the database
- **Workload description:** set of (query template, frequency) pairs

Goal: Select a set of indexes that minimize execution time of the workload.

- Cost / benefit balance: Each additional index may help with some queries, but requires updating

This is an optimization problem!

Example

Workload description:

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

Frequency
10,000,000

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

Frequency
10,000,000

Which indexes might we choose?

Example

Workload description:

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

Frequency 10,000,000

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

Frequency 100

Now which indexes might we choose? Worth keeping an index with manufacturer in its search key around?

Estimating index cost?

- Note that to frame as optimization problem, we first need an estimate of the ***cost*** of an index lookup

- Need to be able to estimate the costs of different indexes / index types…

We will see this mainly depends on getting estimates of result set size!

Ex: Clustered vs. Unclustered

Cost to do a range query for M entries over N-page file (P per page):

- Clustered:
  - To traverse: $\text{Log}_f(1.5N)$
  - To scan: 1 random IO + $\left\lceil \frac{M-1}{P} \right\rceil$ sequential IO
- Unclustered:
  - To traverse: $\text{Log}_f(1.5N)$
  - To scan: ~ M random IO

Suppose we are using a B+ Tree index with:
- Fanout f
- Fill factor 2/3

Plugging in some numbers

To simplify:
- Random IO = ~10ms
- Sequential IO = free

- Clustered:
  - To traverse: $\text{Log}_F(1.5N)$
  - **To scan: 1 random IO +** $\left\lceil \frac{M-1}{P} \right\rceil$ **sequential IO**

~ 1 random IO = 10ms

- Unclustered:
  - To traverse: $\text{Log}_F(1.5N)$
  - **To scan: ~ M random IO**

~ ***M*** random IO = M*10ms

- If M = 1, then there is no difference!
- If M = 100,000 records, then difference is ~10min. Vs. 10ms!

If only we had good estimates of M…

HISTOGRAMS & IO COST ESTIMATION

IO Cost Estimation via Histograms

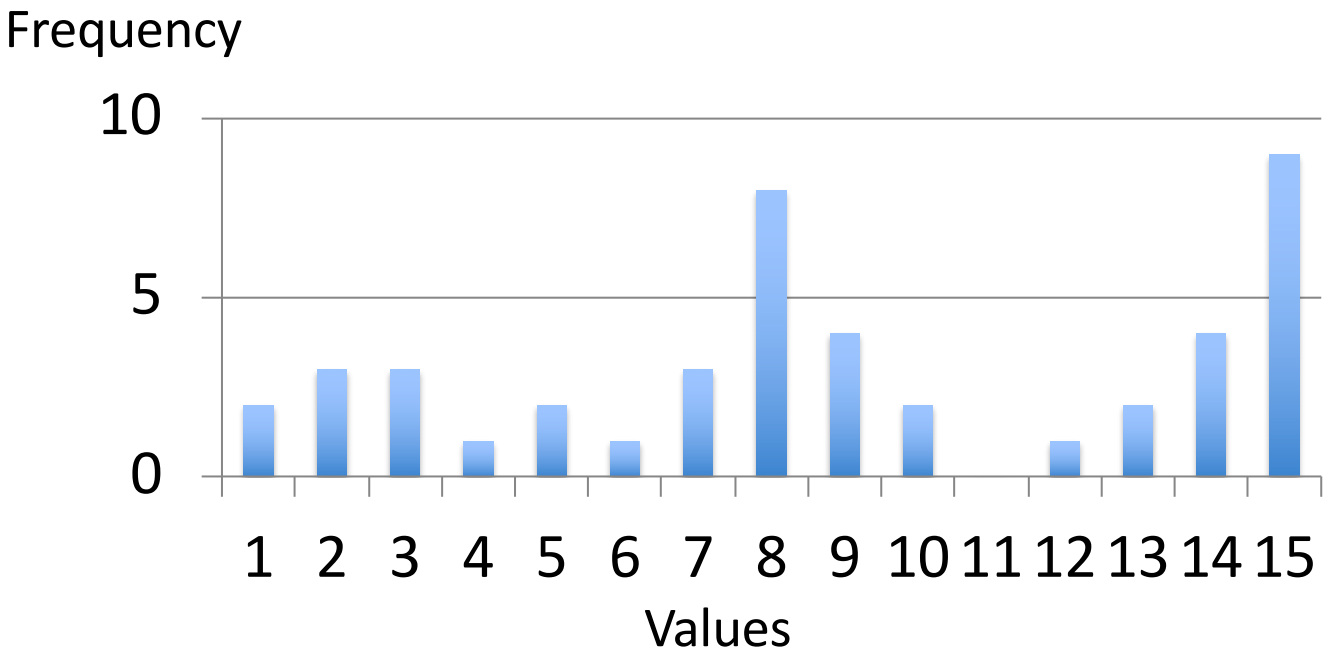
- For **index selection**:
  - What is the cost of an index lookup?
- Also for **deciding which algorithm to use**:
  - Ex: To execute $\mathrm{R} \bowtie S$, which join algorithm should DBMS use?
  - **What if we want to compute $\boldsymbol{\sigma_{A>10}(\mathrm{R}) \bowtie \sigma_{B=1}(S)}$?**
- In general, we will need some way to ***estimate intermediate result set sizes***

Histograms provide a way to efficiently store estimates of these quantities

Histograms

- A histogram is a set of value ranges ("buckets") and the frequencies of values in those buckets occurring
- How to choose the buckets?
  - Equiwidth & Equidepth
- Turns out high-frequency values are **very** important

Example



How do we compute how many values between 8 and 10?
(Yes, it's obvious)

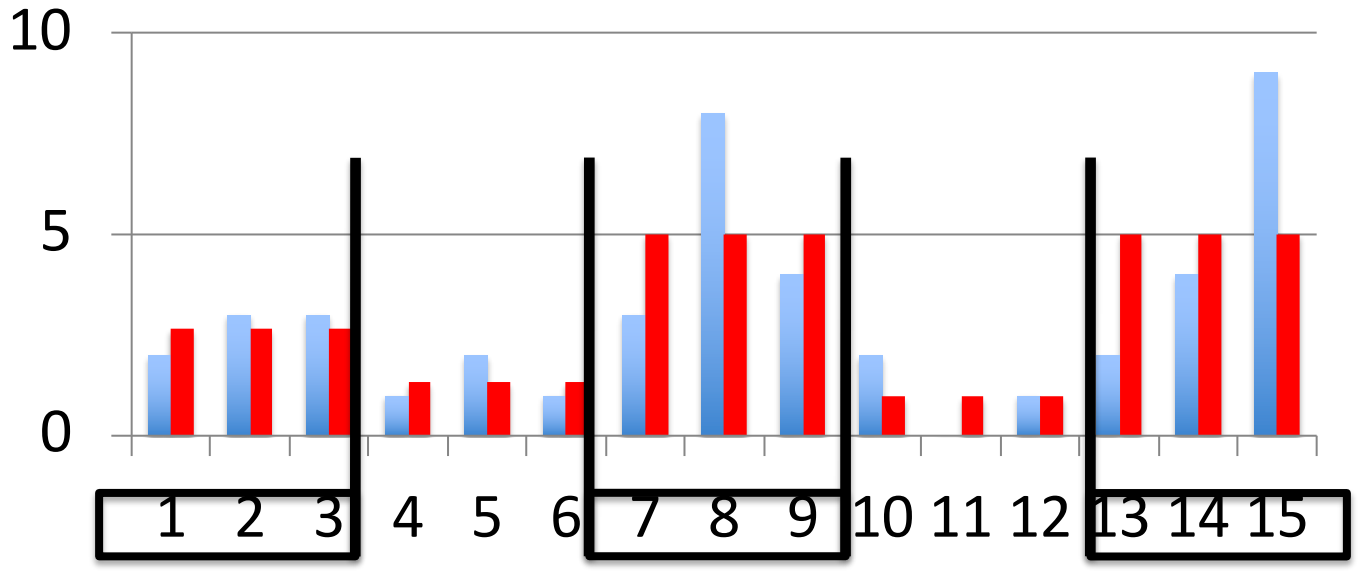
Problem: counts take up too much space!

Fundamental Tradeoffs

- Want high resolution (like the full counts)
- Want low space (like uniform)
- Histograms are a compromise!

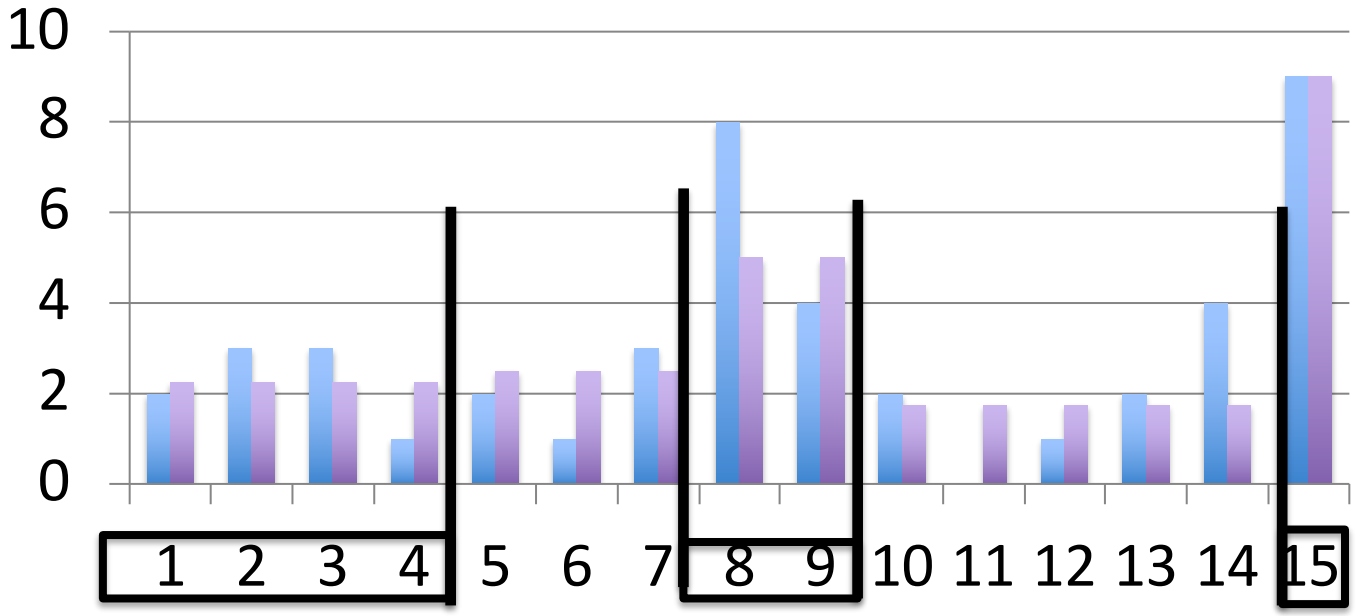
So how do we compute the “bucket” sizes?

Equi-width



All buckets roughly the same width

Equidepth



All buckets contain roughly the same number of items (total frequency)

Histograms

- Simple, intuitive and popular
- Parameters: # of buckets and type
- Can extend to many attributes (multidimensional)

Maintaining Histograms

- Histograms require that we update them!
  - Typically, you must run/schedule a command to update statistics on the database
  - Out of date histograms can be terrible!

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.

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.