Lecture 17: Optimization!

Announcements

Systems design & scale: E.g, Related products [solution]

Systems Design Exercise

Problem Overview

Pre-design: Schemas, Disk, and RAM
- What's size of IDs and tables?
- Managing RAM and disk
- What's the problem?
- What usual techniques can you use to improve?

Design with 1 machine

Design with 1 machine, UserSession assumption

Design with M machines

Design Optimizations

Pre-design: Schemas, Disk, and RAM

1. What's size of IDs and tables?

	Size	Why?
ProductId		
UserID		
LogOfViewsID		
Product		
Users		
LogOfViews		
CoOccur		

Design with M machines

Design #2P: Another engineer proposes following design, with M machines. Analyze with UserSession assumption and assume network is infinitely fast to copy files, for simplicity.

Design 2P
1. Scan LogOfviews. For each user, append <p_i, p_j> to log TempCoOccurLog.$q, where [q = hash(p_i, p_j) % M], if the user has viewed product p_i and p_j. (i.e., break into M log **partitions**, one per machine, by hashing so each unique product pair is in one log)
2. For each TempCoccurLog partition on a machine, in parallel
   a. Externally sort log on disk on each machine
   b. Scan sorted log, and count co-occur pairs in a single pass. Drop co-occur pairs with < 1 million.
3. At this point, each machine has a CoOccur partition. Push CoOccur partitions to central machine.

Answer:

Steps	Cost (time)	Why?
Scan LogOfViews Append <p_i, p_j> to TempCoOccurLog.$q		
Externally sort each TempCoOccurLog.$q on disk (Assume sort cost is ~2N, where N is number of		

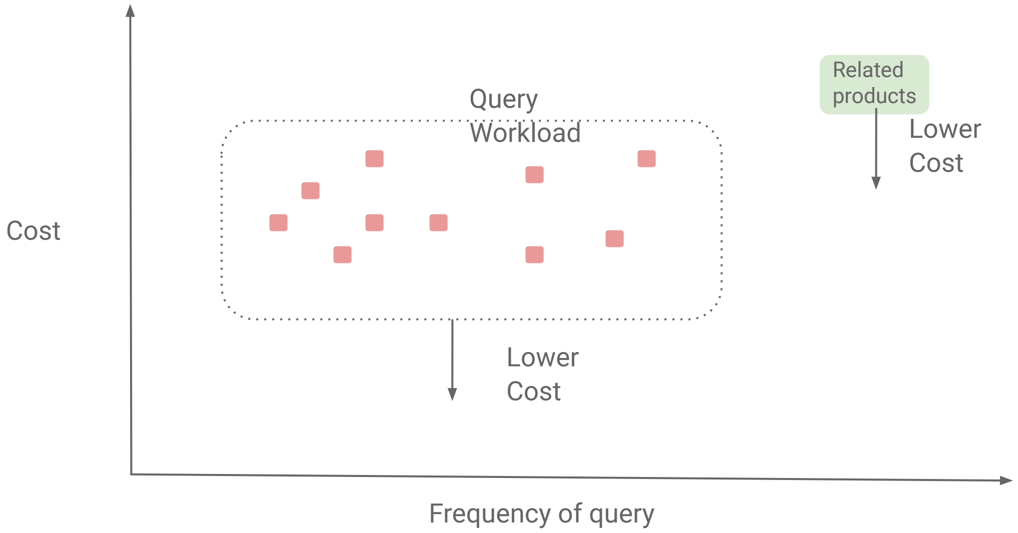
Why?

Real systems - start with a design doc

Interview questions

Optimizing

Queries and workloads



Workload = <Query, Frequency of query>

Example:

Basic SFW queries

Workload description

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

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

Lower cost (query and update cost)

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

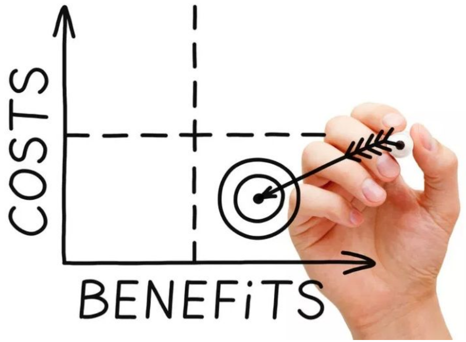
Intuition

Manufacturers likely most **Selective.**

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

Optimization Roadmap

Build Query Plans



Analyze Plans

1. For SFW, Joins queries
   a. Sort? Hash? Count? Brute-force?
   b. Pre-build an index? B+ tree, Hash?

2. What statistics can I keep to optimize?
   a. E.g. Selectivity of columns, values

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

1. Nested Loop Joins

What you will learn about in this section

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

RECAP: Joins

Joins: Example

$R \bowtie S$

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

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

R

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

S

A	D
3	7
2	2
2	3

A	B	C	D
2	3	4	2

Joins: Example

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

```
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

Joins: 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 $\mathrm{r} \in R, s \in S$ *such that* $r.A = s.A$

R

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

S

A	D
3	7
2	2
2	3

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

Joins: Example

$R \bowtie S$

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

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

R

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

S

A	D
3	7
2	2
2	3

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

Joins: Example

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

```
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 in this way?

Nested Loop Joins

Notes

We consider “IO aware” algorithms: ***care about disk IO***

Given a relation R, let:

- T(R) = # of tuples in R
- P(R) = # of pages in R

Recall that we read / write entire pages with disk IO

We’ll see lots of formulae from now

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

Nested Loop Join (NLJ)

Compute $R \bowtie S \ on \ A$:

```
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 $R \bowtie S$ 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)

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

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

Nested Loop Join (NLJ)

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

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

<u>Cost:</u>

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)
```

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

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

Cost:

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

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

Nested Loop Join (NLJ)

```
Compute R ⋈ 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!

IO-Aware Approach

Block Nested Loop Join (BNLJ)

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

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

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

Cost:

$P(R)$

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)

```
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
3. **Check against the join conditions**

BNLJ can also handle non-equality constraints

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)
```

Again, ***OUT*** could be bigger than P(R)*P(S)… but usually not that bad

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

Cost:

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

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

$\mathrm{P}(R) + \frac{P(R)}{B-1} P(S)$ + 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 computing the ***full cross-product*** have a **quadratic** term

- For example we saw:

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

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

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

- ~ O(P(R) + P(S) + ***OUT***)

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
```

<u>Cost:</u>

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

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

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

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

Optimizing Joins

(the good stuff, multi table joins)

Message: It’s all about the IO and memory!

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

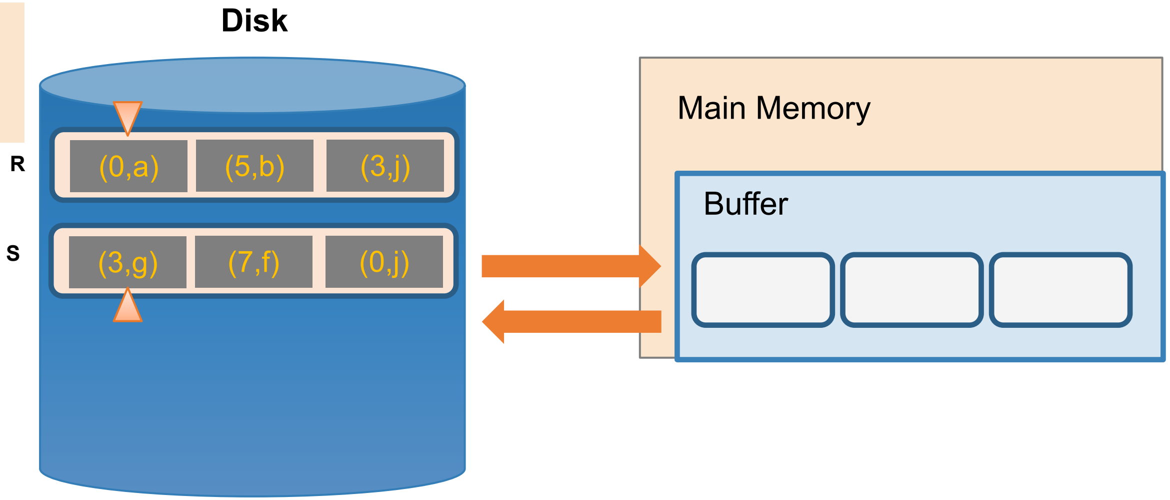
1. Sort R, S on A using ***external merge sort***
2. ***Scan*** sorted files and "merge"
3. *[May need to "backup"- see next subsection]*

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

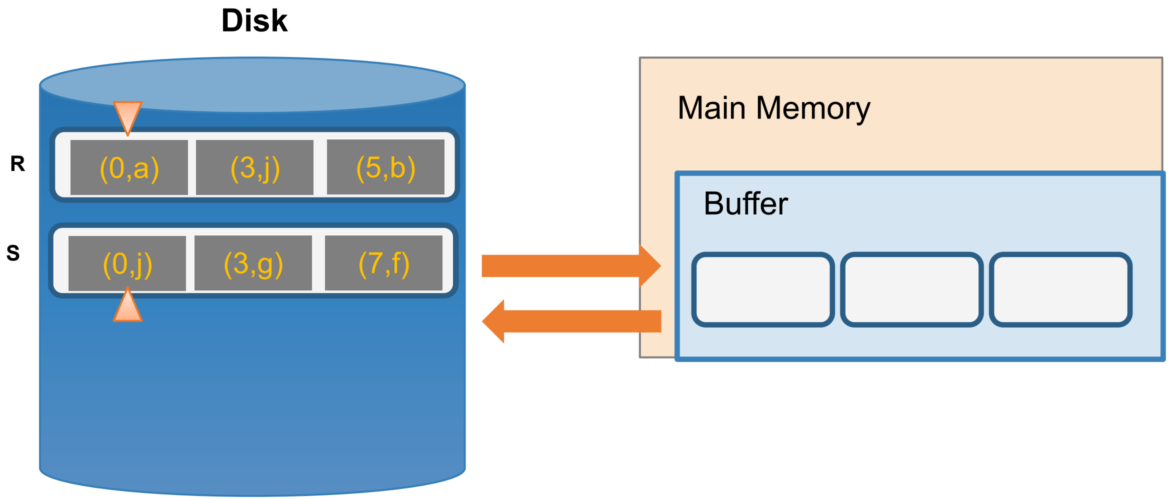
SMJ Example: R⋈S *on A* with 3 page buffer

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

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

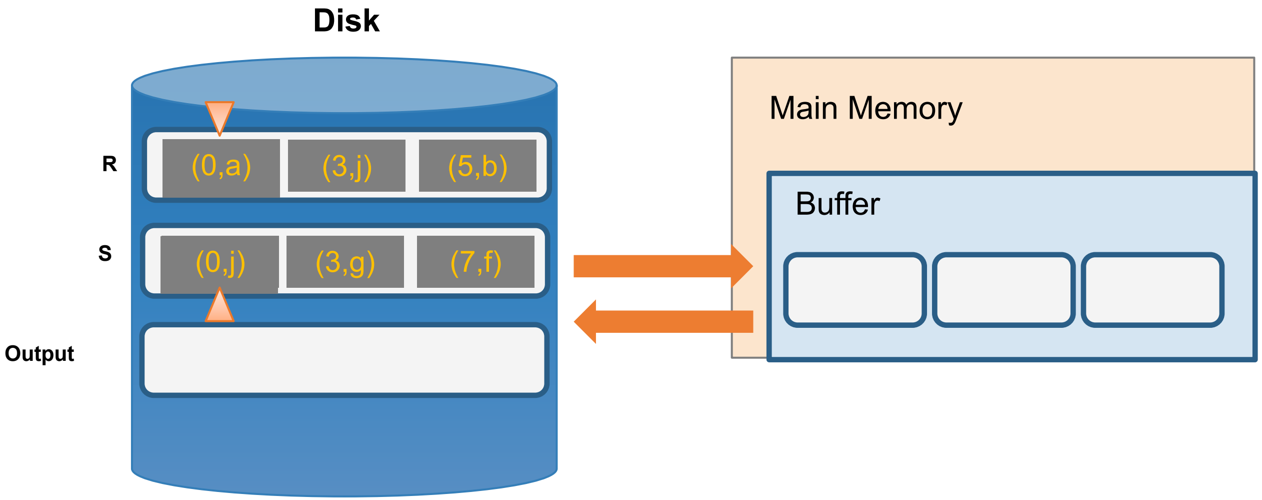


SMJ Example: R⋈S on A with 3 page buffer



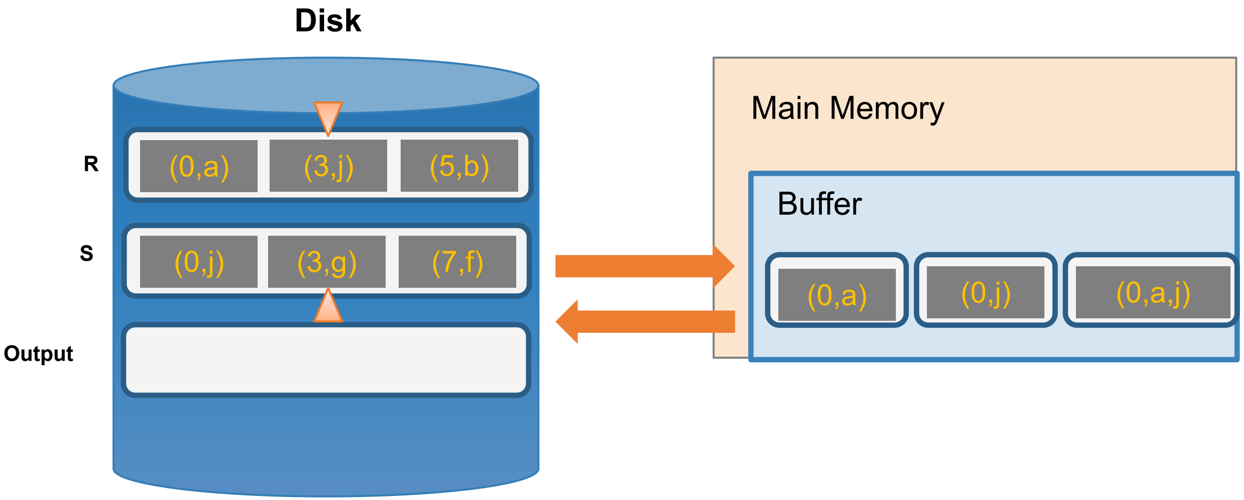
SMJ Example: R⋈S *on* A with 3 page buffer

2. Scan and “merge” on join key!



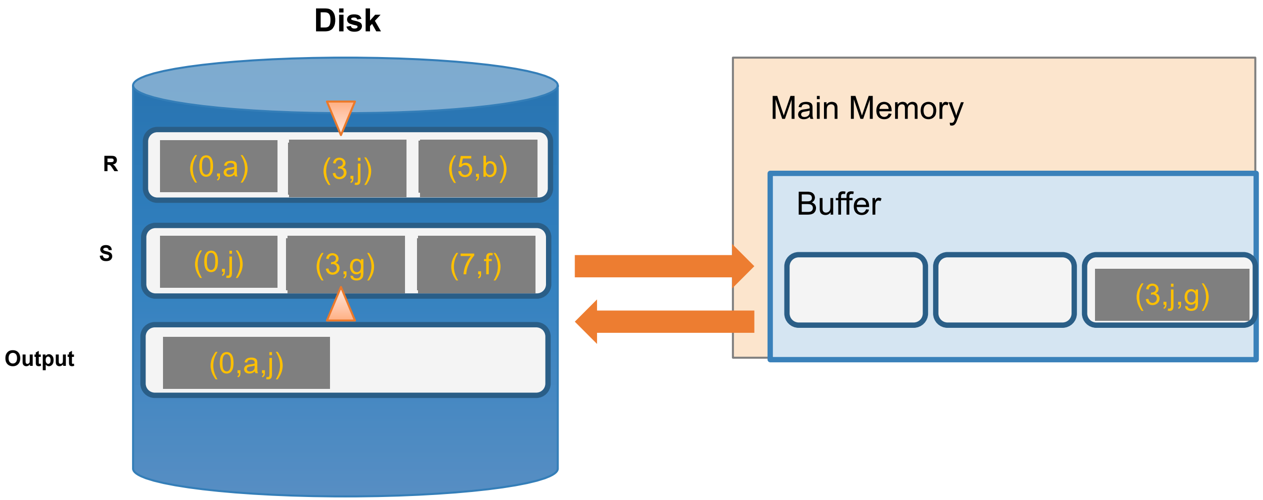
SMJ Example: $R \bowtie S$ *on A* with 3 page buffer

2. Scan and “merge” on join key!



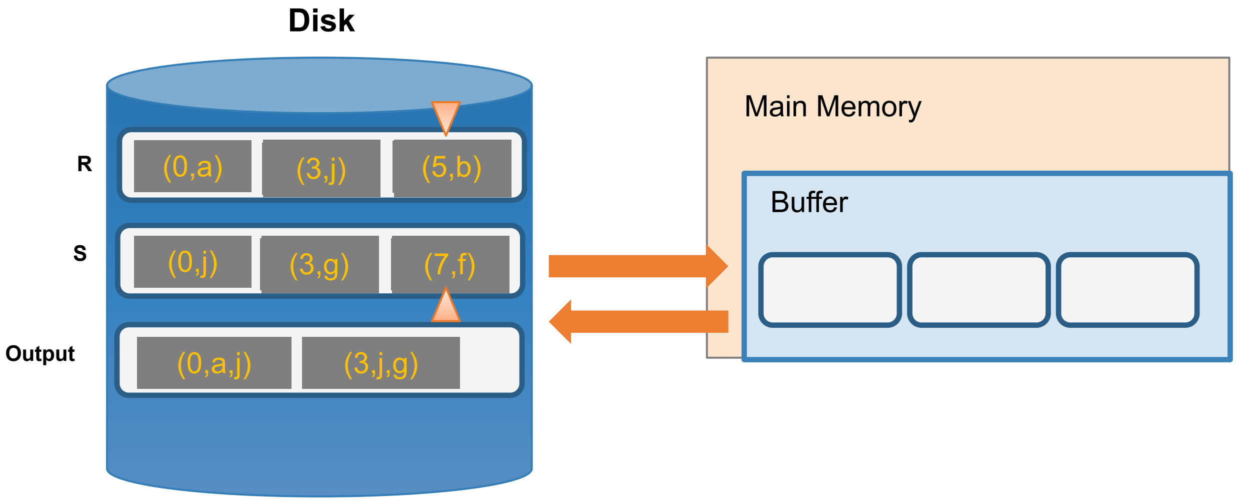
SMJ Example: R⋈S on A with 3 page buffer

2. Scan and "merge" on join key!



SMJ Example: R⋈S on A with 3 page buffer

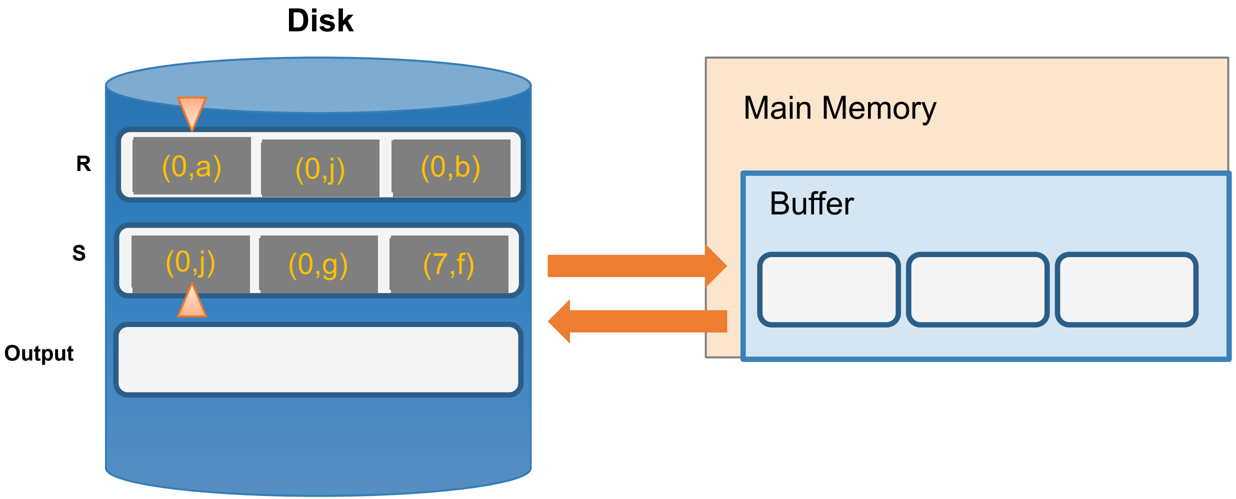
2. Done!



What happens with duplicate join keys?

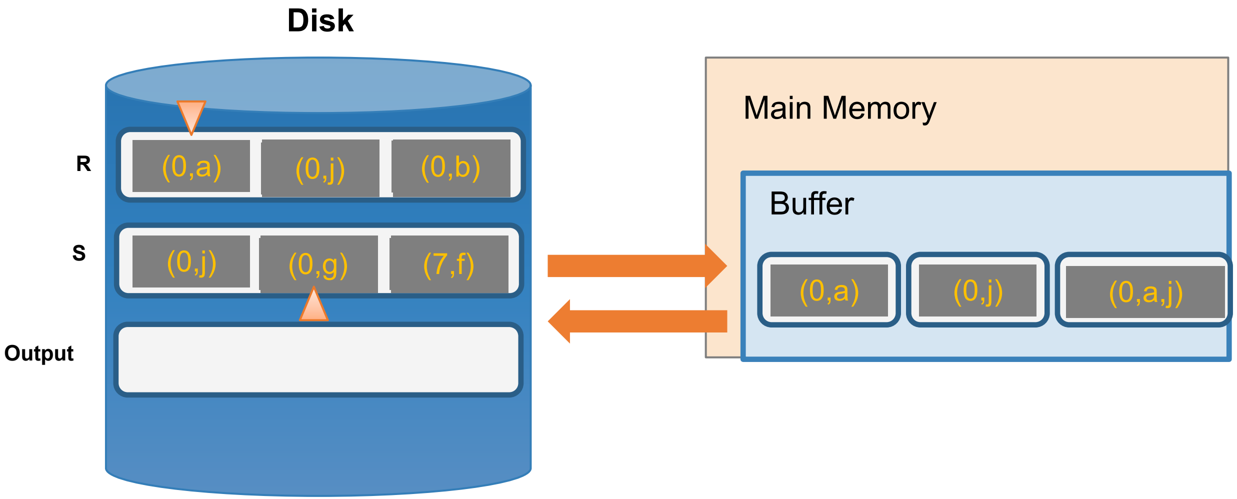
Multiple tuples with Same Join Key: “Backup”

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



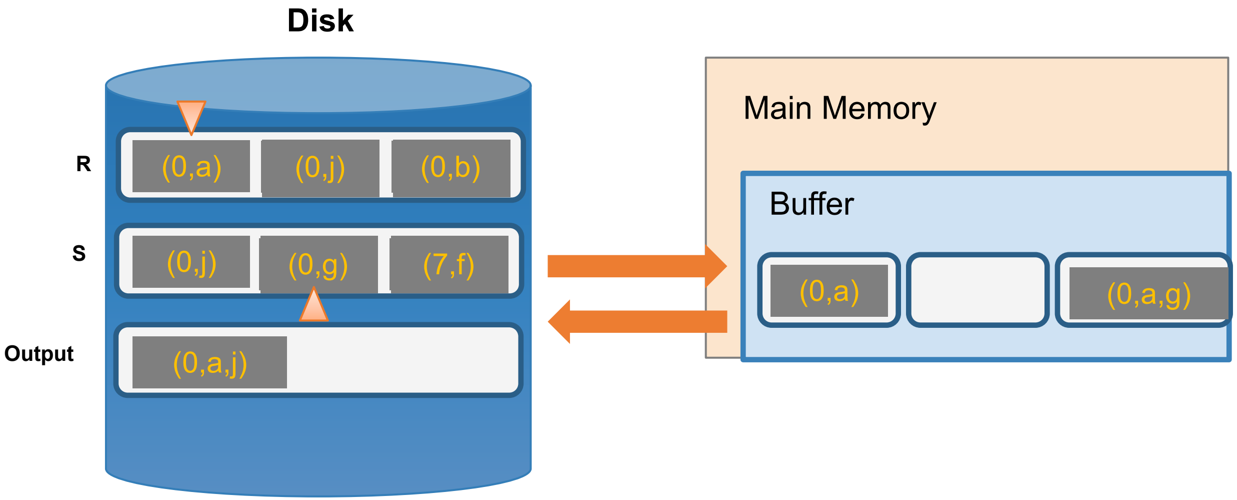
Multiple tuples with Same Join Key: "Backup"

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



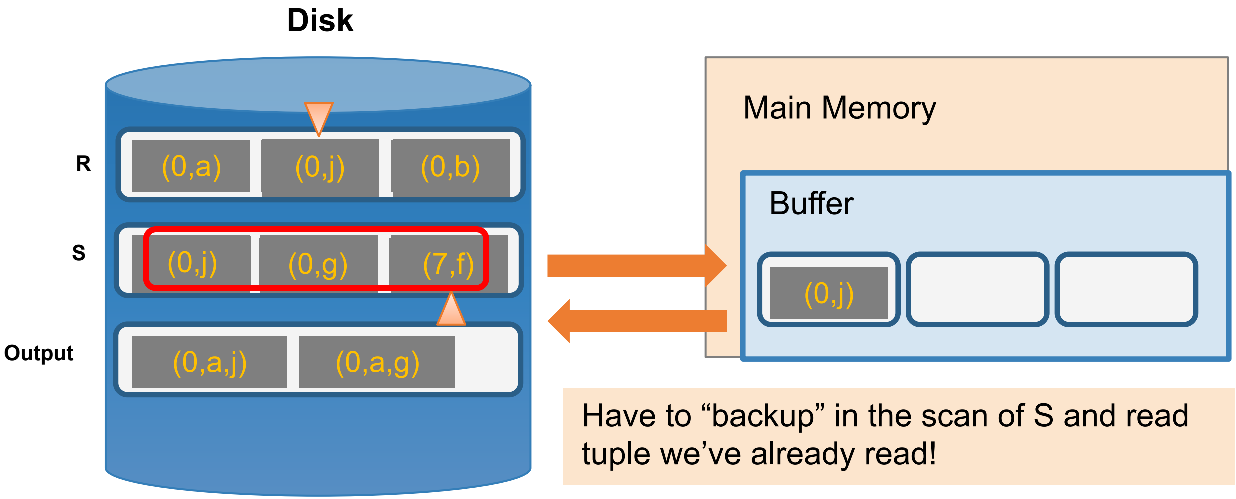
Multiple tuples with Same Join Key: “Backup”

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 *back up* and read each page of S…

- Often not that bad however, plus we can:
  - Leave more data in buffer (for larger buffers)
  - Can "zig-zag"

SMJ: Total cost

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

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

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

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

SMJ vs. BNLJ: Steel Cage Match

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

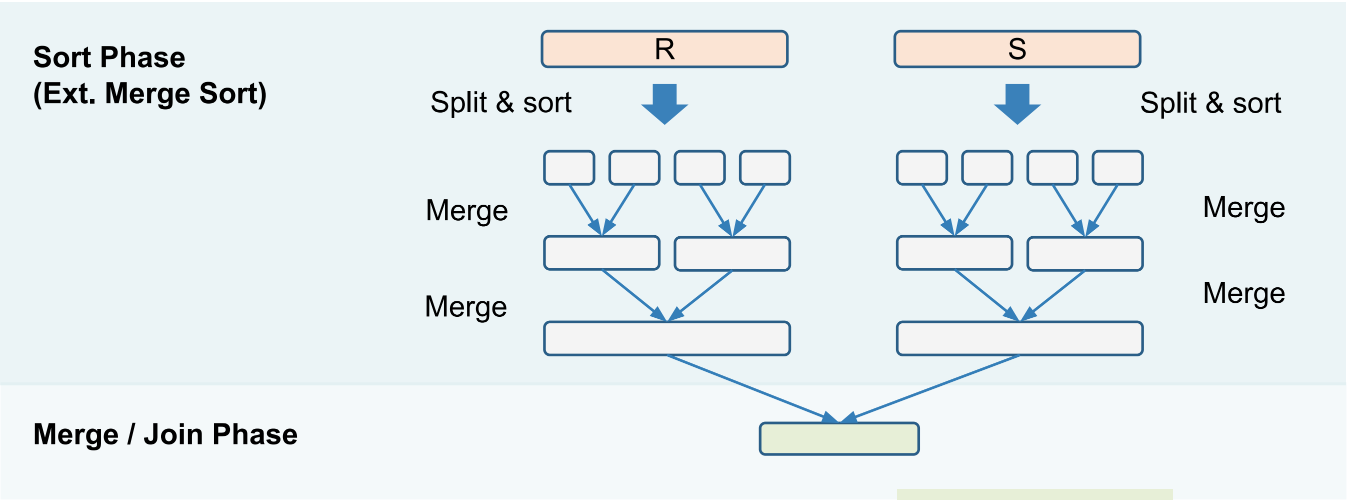
	Buffer = 100	Buffer = 20
SMJ	Sort R and S in ‘k=2’ passes: 2* (k* 1000 + k* 500) Merge: 1000 + 500 = 1500 IOs ⇒ 7500 IOs + OUT	Sort R and S in ‘k=3’ passes: 2* (k* 1000 + k* 500) Merge: 1000 + 500: 1500 IOs ⇒ 10,500 IOs + OUT
BNLJ	500 + 1000*500/(100-2) ⇒ ~5.6K+OUT	500 + 1000*500/(20-2) ⇒ 28.2K IOs +OUT

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

Un-Optimized SMJ

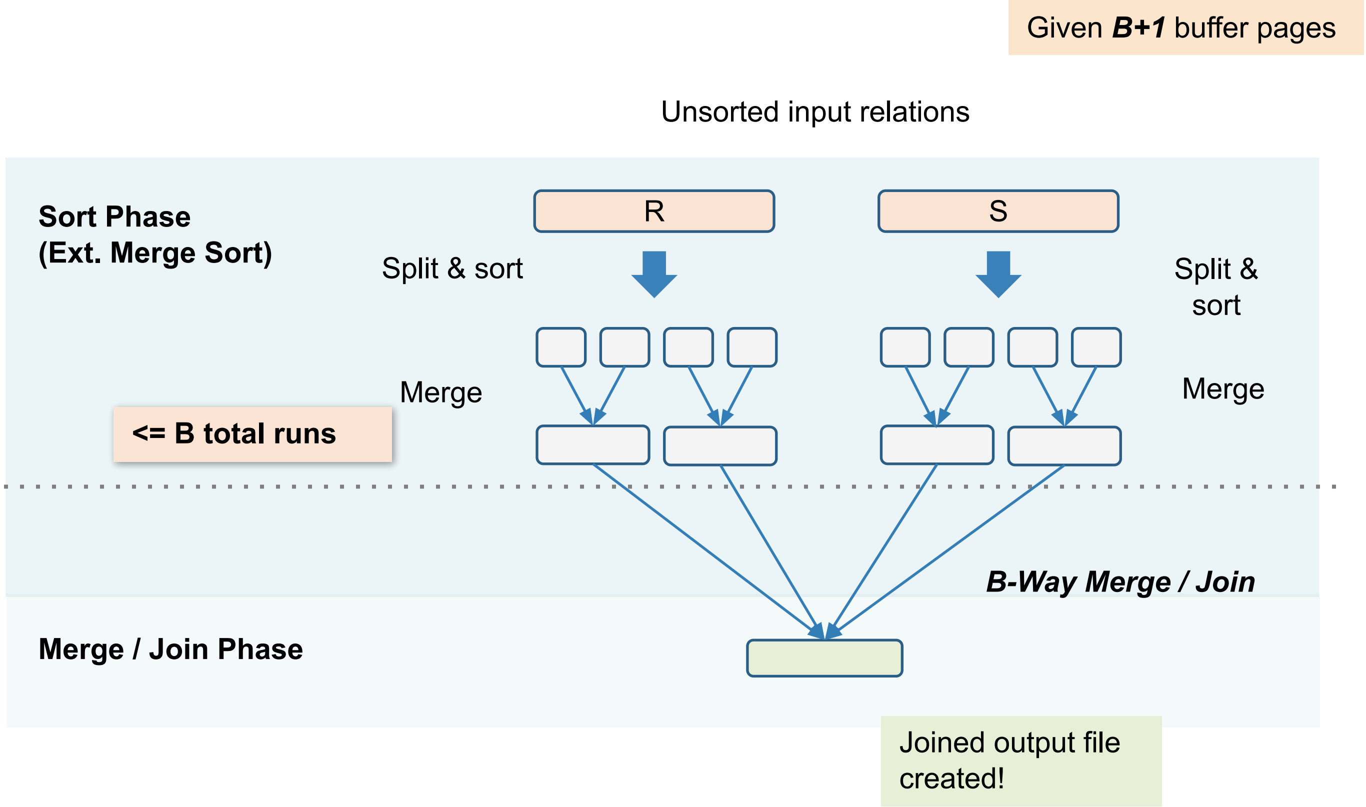
Given ***B+1*** buffer pages

Unsorted input relations



Joined output file created!

Simple SMJ Optimization



Takeaway points from SMJ

If input already sorted on join key, skip the sorts.

- SMJ is basically linear.
- Nasty but unlikely case: Many duplicate join keys.

SMJ needs to sort **both** relations

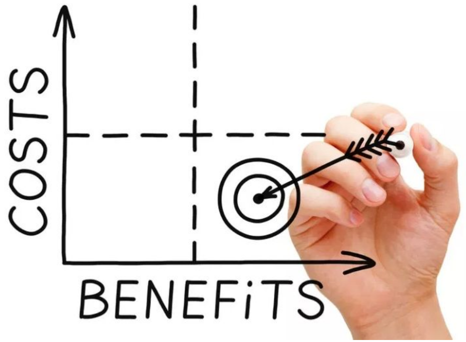
- If max { P(R), P(S) } < B^2 then cost is 3(P(R)+P(S)) + OUT

Lecture 18: Optimization!

Optimization

Roadmap

Build Query Plans



Analyze Plans

1. For SFW, Joins queries
   a. Brute-force? Sort? Hash? Count?
   b. Pre-build an index? B+ tree, Hash?

2. What statistics can I keep to optimize?
   a. E.g. Selectivity of columns, values

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

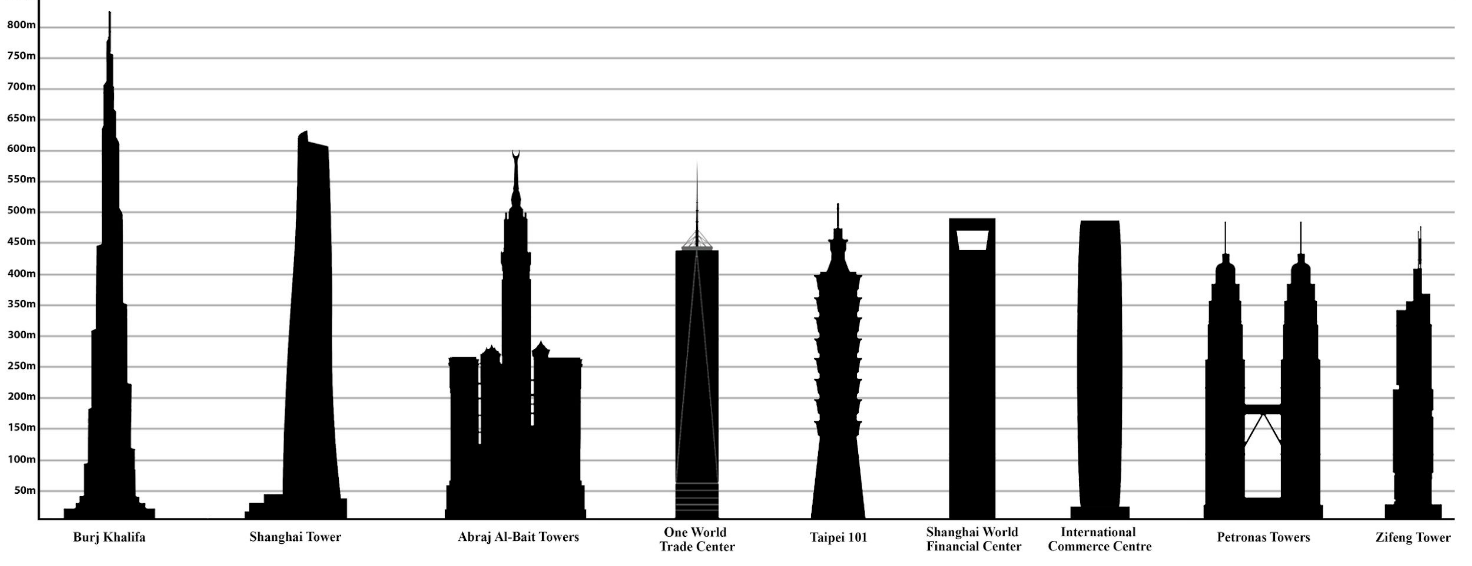
Example

Elevators

Hashing and sorting ⇐ floor numbers :-)

Example

Elevators



<u>Design choices?</u>

- Split problem into small, independent buckets (<u>partition</u>)
- What are good buckets? (floor ranges)
  - 1 elevator per floor? 1 elevator per floor group?
  - By height? By #floors? By volume of people?

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

What you will learn about in this section

1. Hash Partition Join

2. Memory requirements

Recall: Hashing

Magic of hashing

- A hash function $h_B(t.A)$ maps t.A (attribute A) into [0,B-1]
- And maps nearly uniformly

A hash **collision** is when a1 != a2 but $h_B(a1) = h_B(a2)$

- Note however that it will **never** occur that a1 = a2 but $h_B(a1) \neq h_B(a2)$

Hash Partition Join: High-level

Note again that we are only considering equality constraints here

To compute R ⋈ S on A:

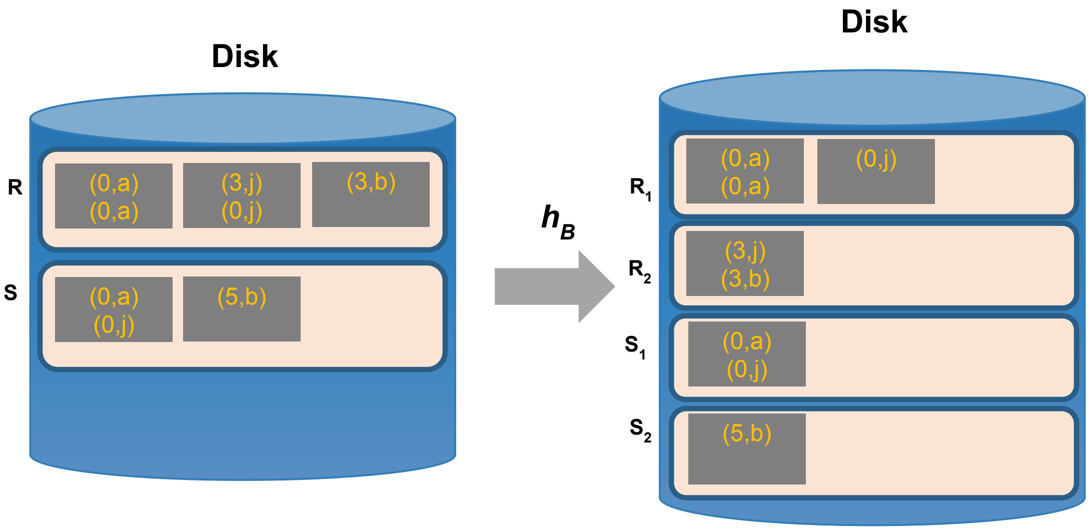
1. Hash Partition: Split R, S into B buckets, using h_B on A

2. Per-Partition Join: JOIN tuples in same partition (i.e, same hash value)

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

HPJ: High-level procedure

1. Hash Partition: Split R, S into B buckets, using h_B on A

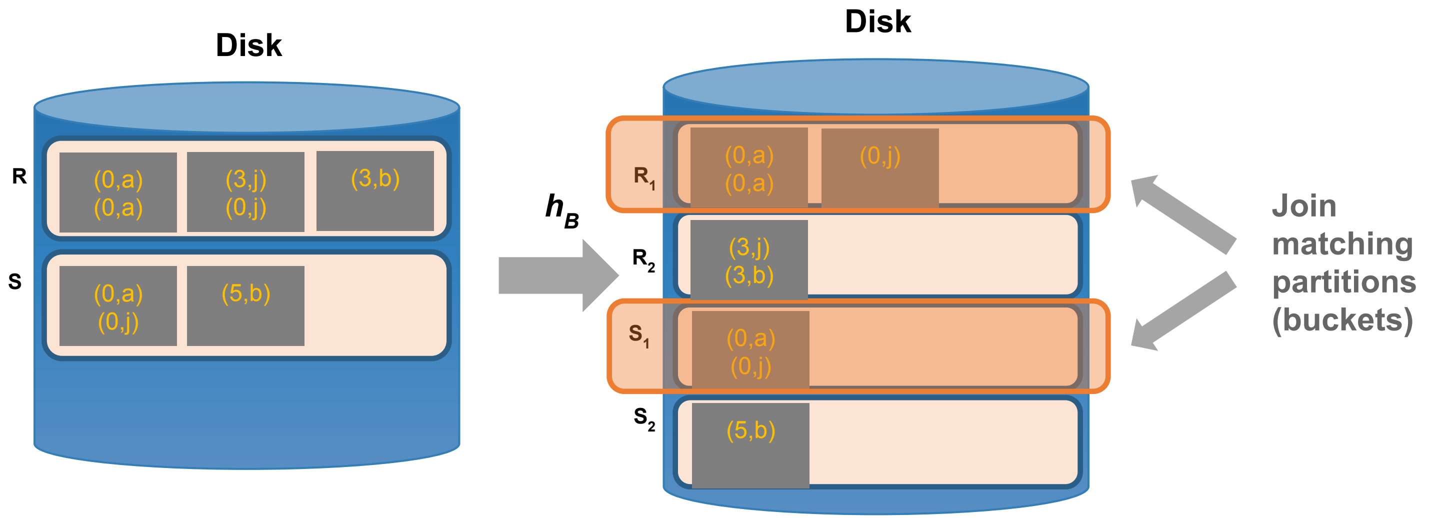


Note our new convention: pages each have two tuples (one per row)

More detail in a second…

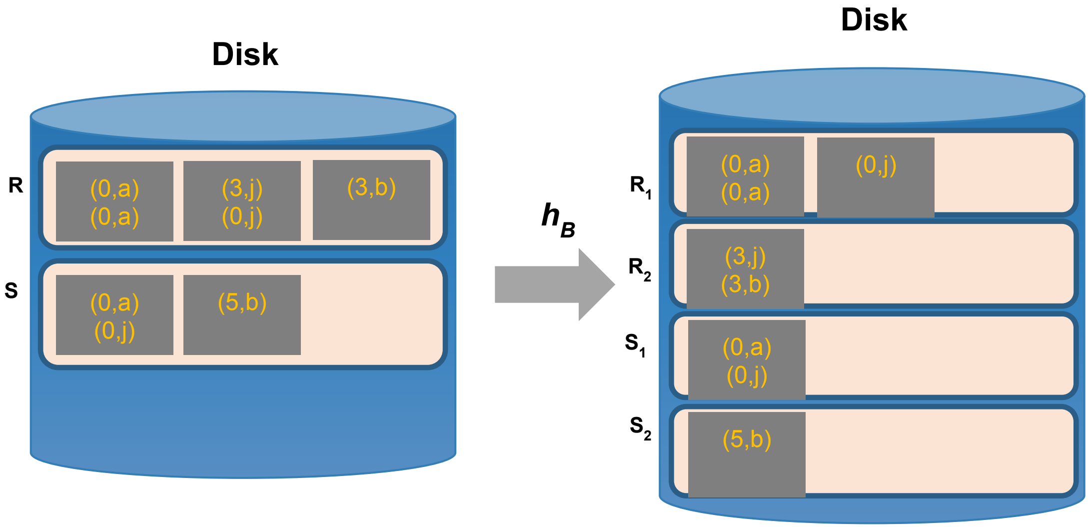
HPJ: High-level procedure

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



HPJ: High-level procedure

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



Don't have to join the others! E.g. (S_1 and R_2)!

HPJ Phase 1: Hash Partitioning

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

Given B+1 buffer pages, we partition into B buckets:
- We use B buffer pages for output (one for each bucket), and 1 for input
  - The "dual" of sorting.
  - For each tuple t in input, copy to buffer page for $h_B(t.A)$
  - When page fills up, flush to disk.

How big are the resulting buckets?

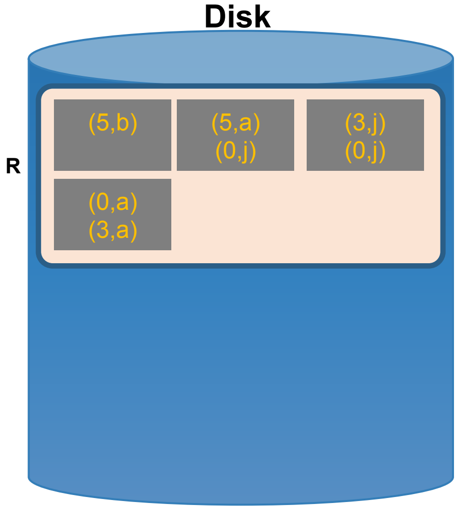
Given ***B+1*** buffer pages

- Given **N input pages, we partition into B buckets**:
  - → Ideally our buckets are each of size **~ N/B pages**
- What happens if there are **hash collisions**?
  - Buckets could be > N/B
  - **We'll do several passes…**
- What happens if there are **duplicate join keys?**
  - Nothing we can do here… could have some **skew** in size of the buckets

HPJ Phase 1: Partitioning

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

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



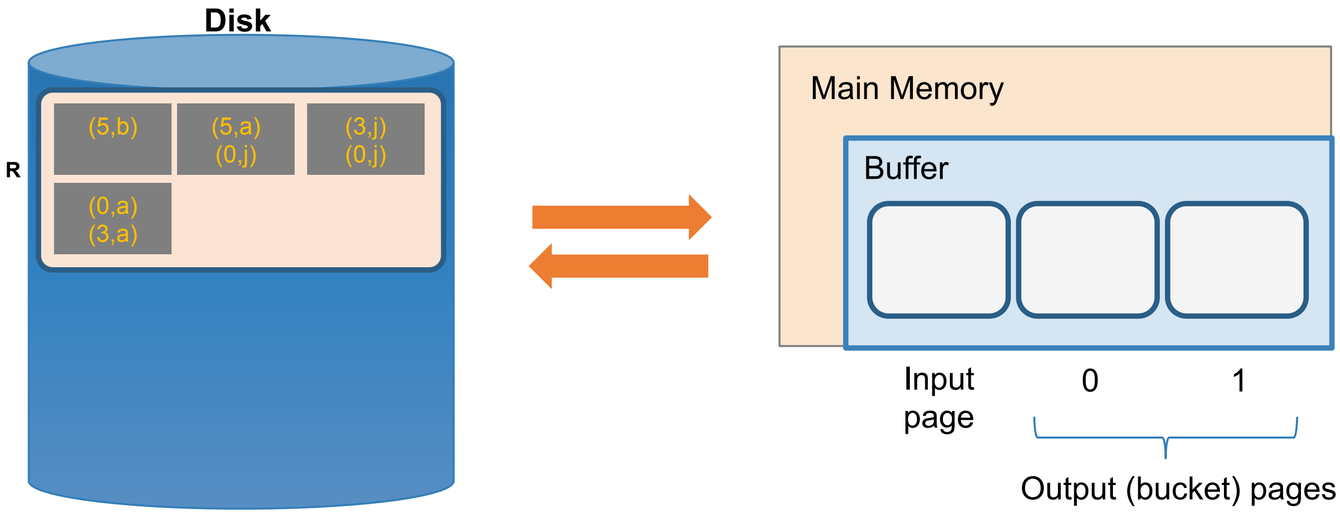
For simplicity, we'll look at partitioning one of the two relations- we just do the same for the other relation!

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

HPJ Phase 1: Partitioning

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

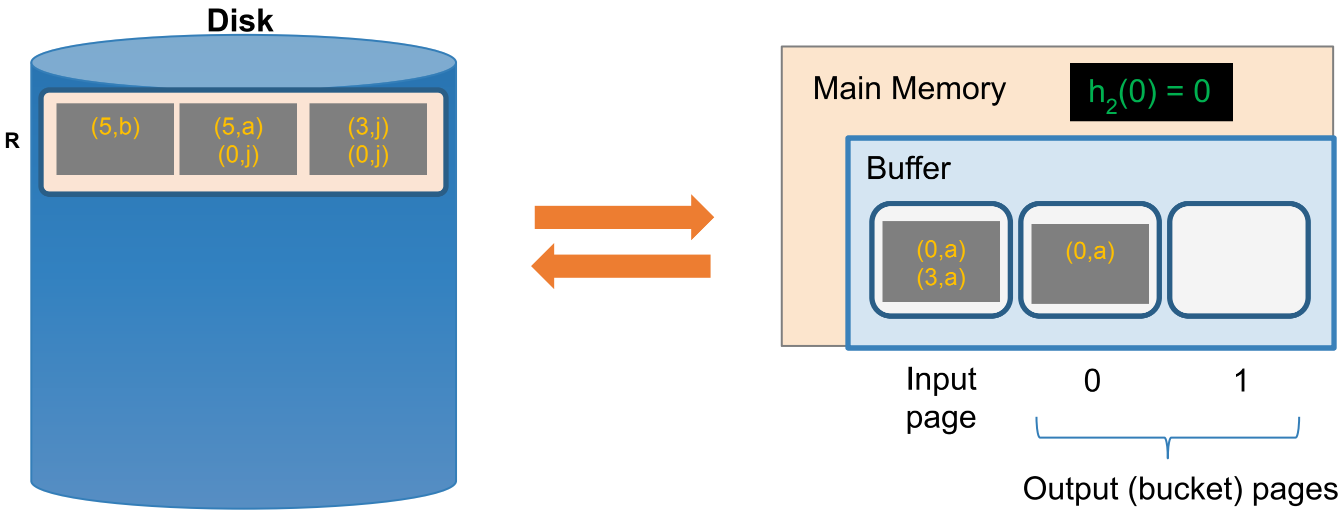
Given ***B+1 = 3*** buffer pages



HPJ Phase 1: Partitioning

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

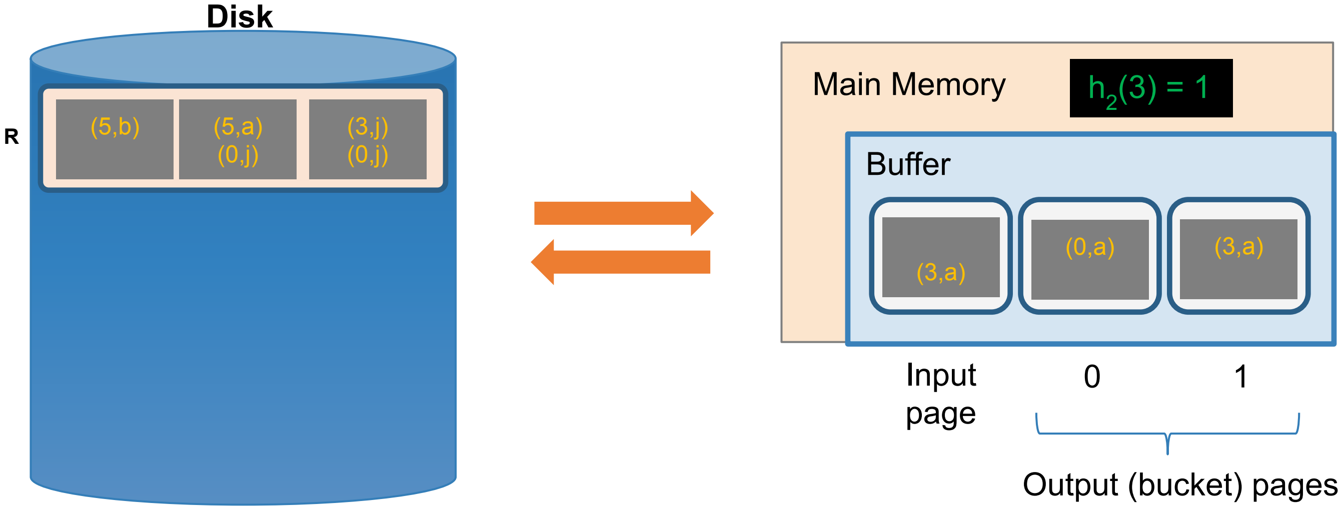
Given ***B+1 = 3*** buffer pages



HPJ Phase 1: Partitioning

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

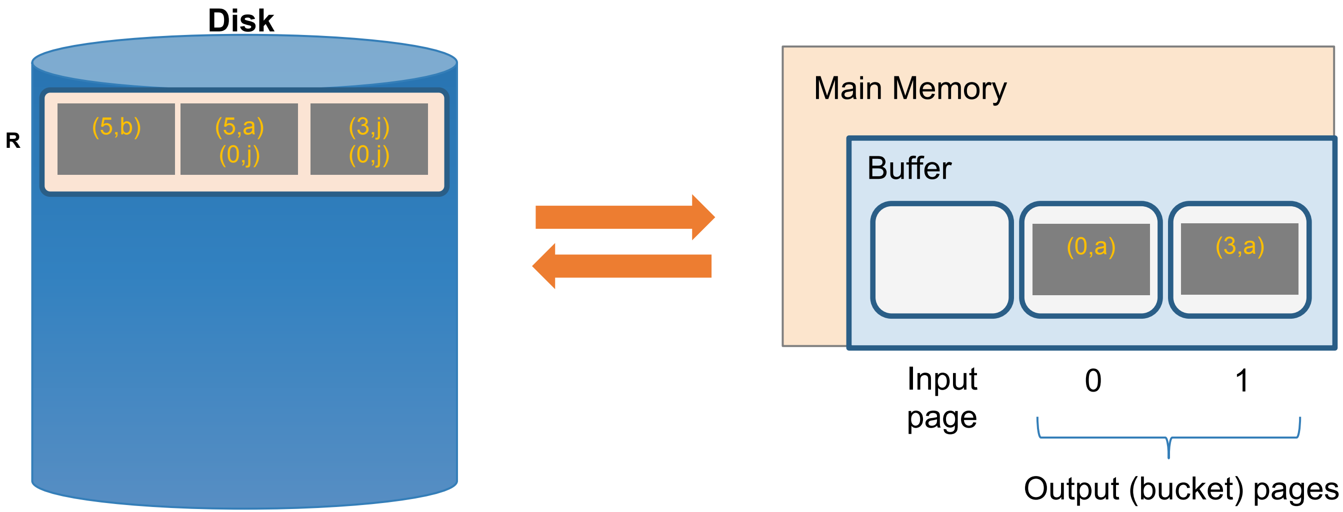
Given ***B+1* = *3*** buffer pages



HPJ Phase 1: Partitioning

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

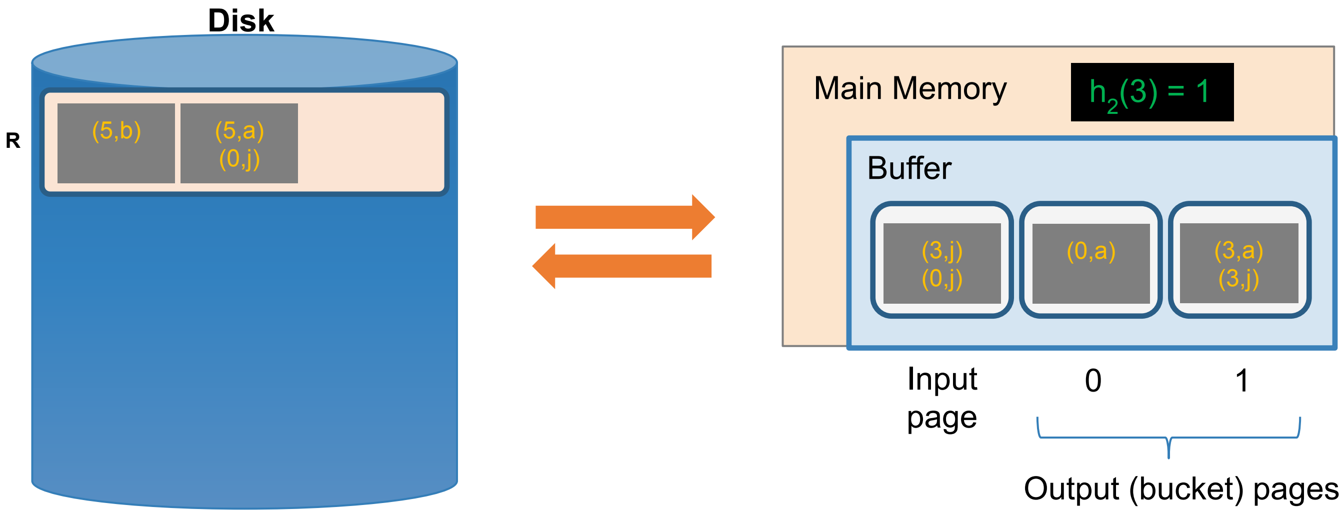
Given ***B+1 = 3*** buffer pages



HPJ Phase 1: Partitioning

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

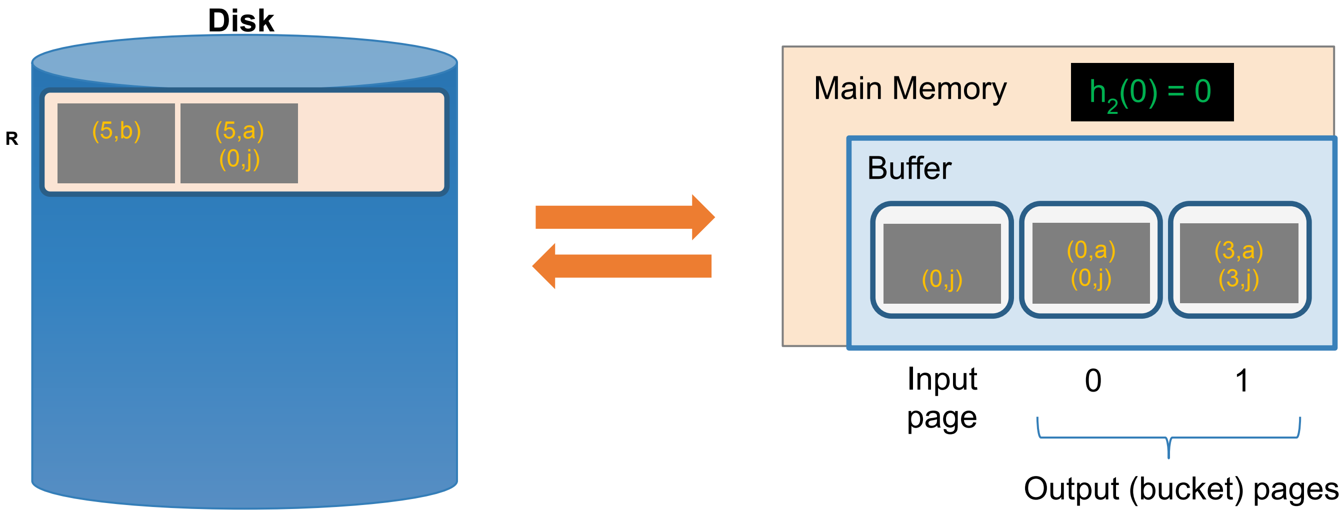
Given ***B+1 = 3*** buffer pages



HPJ Phase 1: Partitioning

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

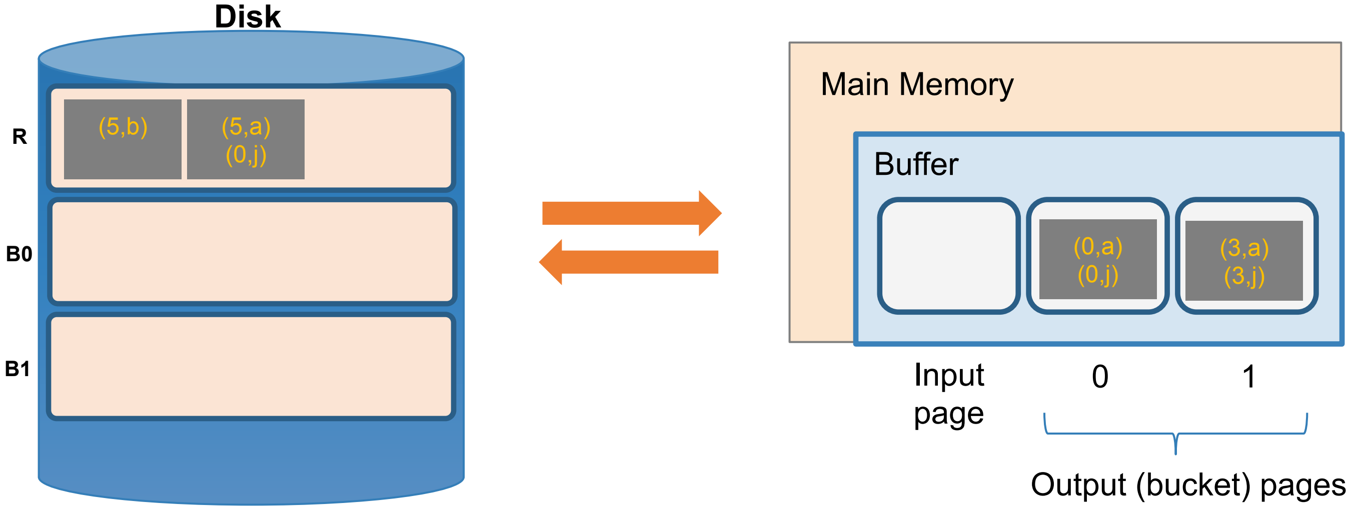
Given ***B+1 = 3*** buffer pages



HPJ Phase 1: Partitioning

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

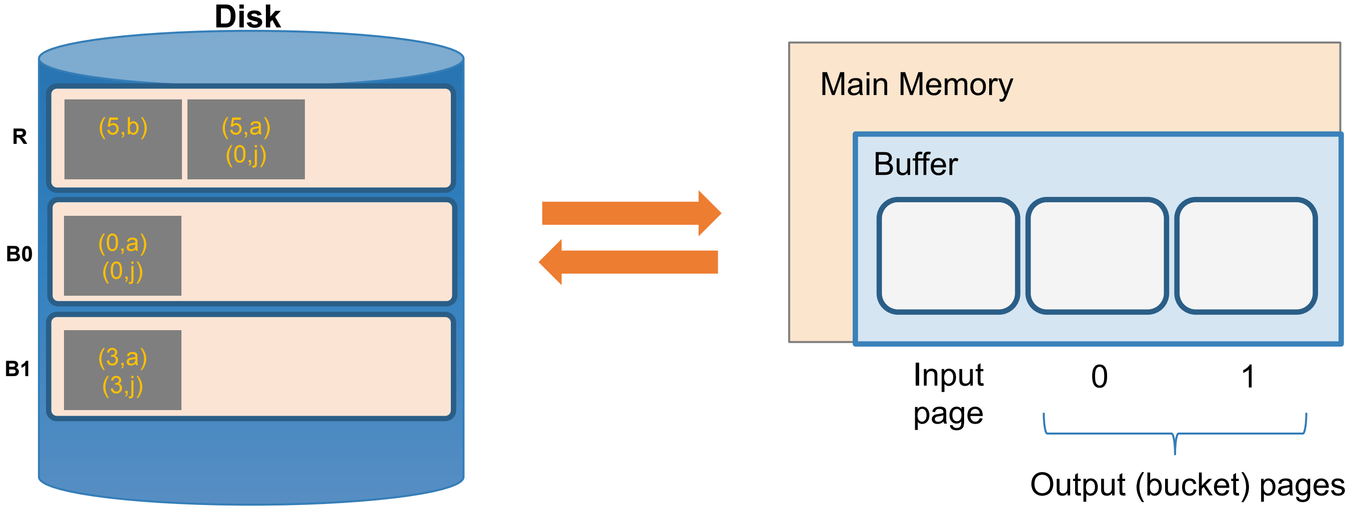
Given ***B+1 = 3*** buffer pages



HPJ Phase 1: Partitioning

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

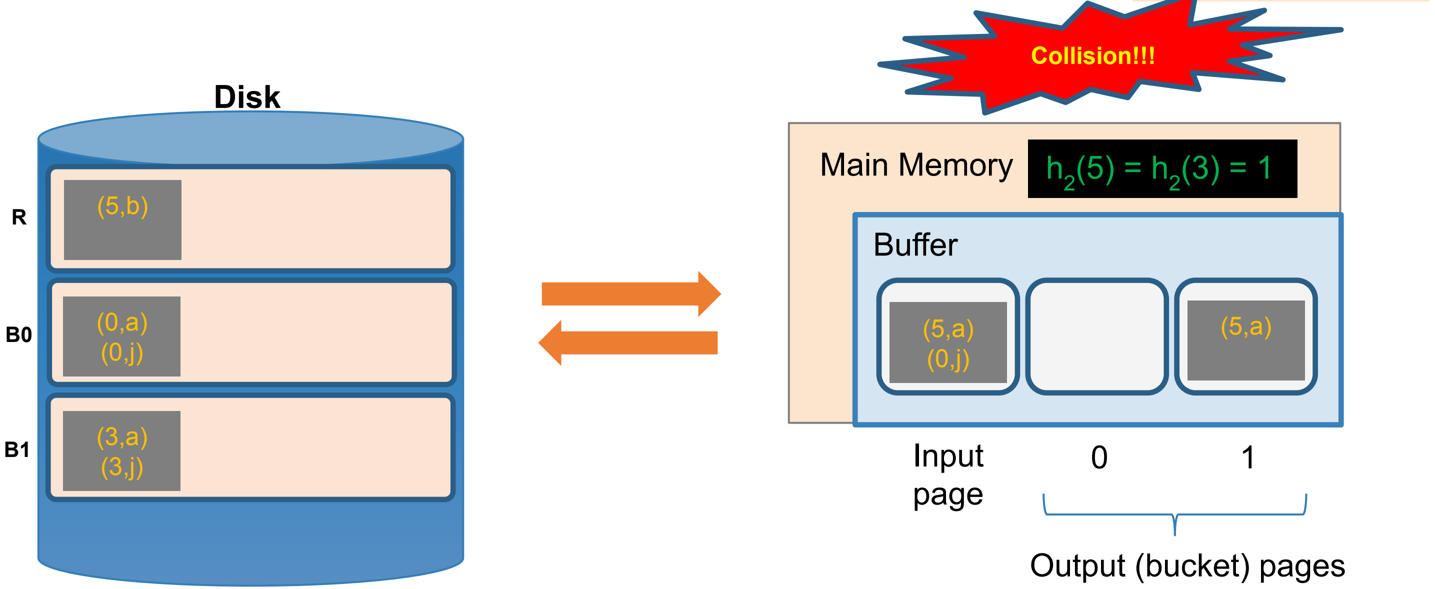
Given ***B+1 = 3*** buffer pages



HPJ Phase 1: Partitioning

Note that collisions can occur!

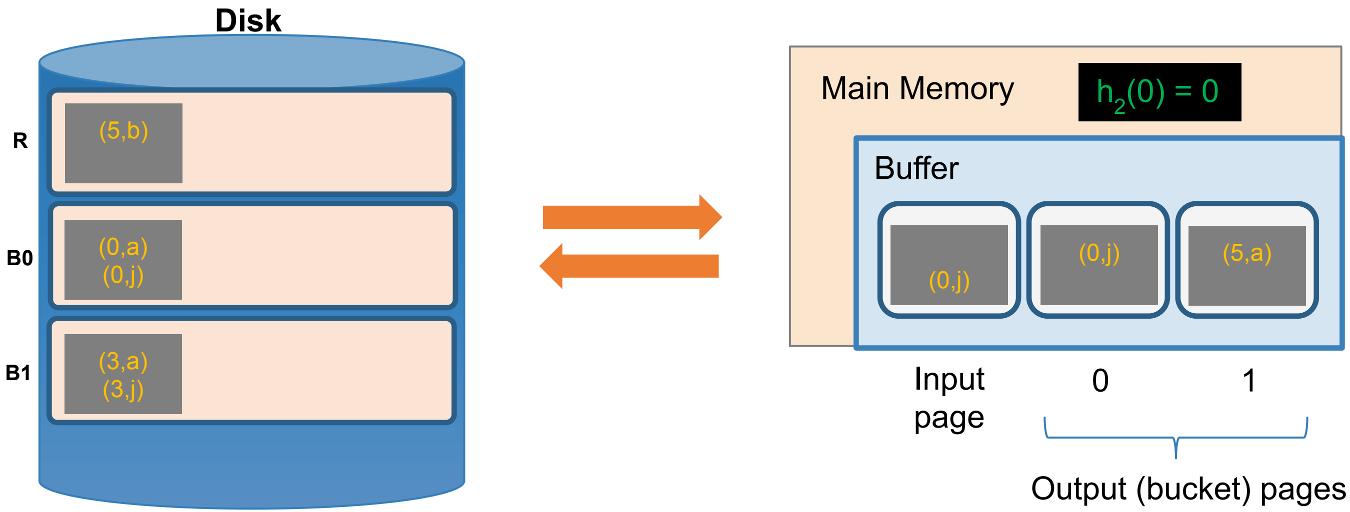
Given ***B+1 = 3*** buffer pages



HPJ Phase 1: Partitioning

Finish this pass…

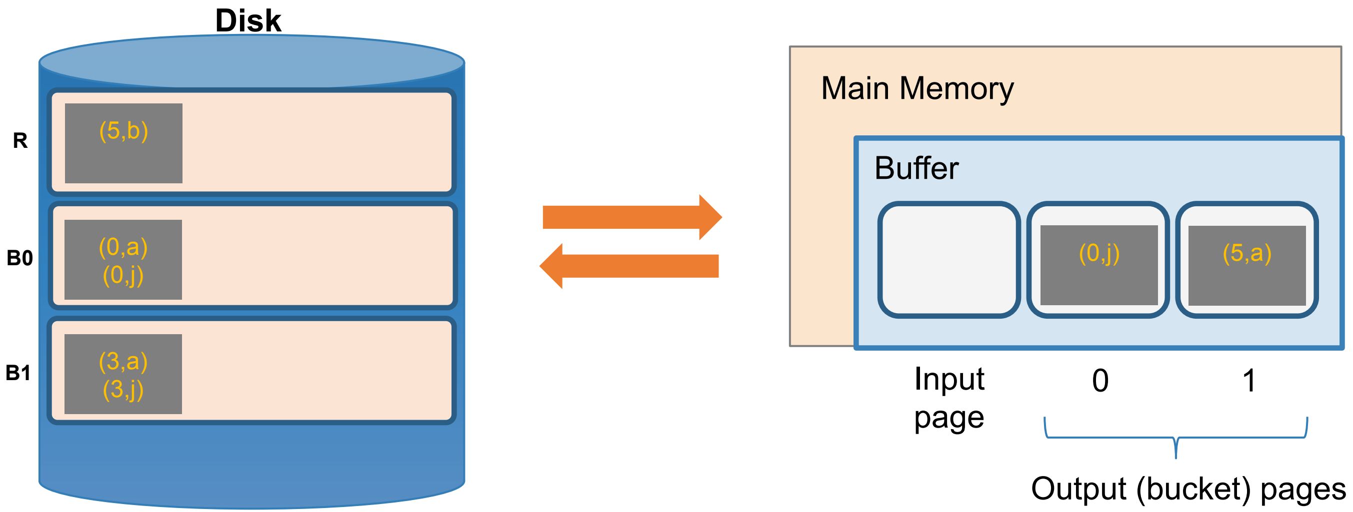
Given ***B+1 = 3*** buffer pages



HPJ Phase 1: Partitioning

Finish this pass…

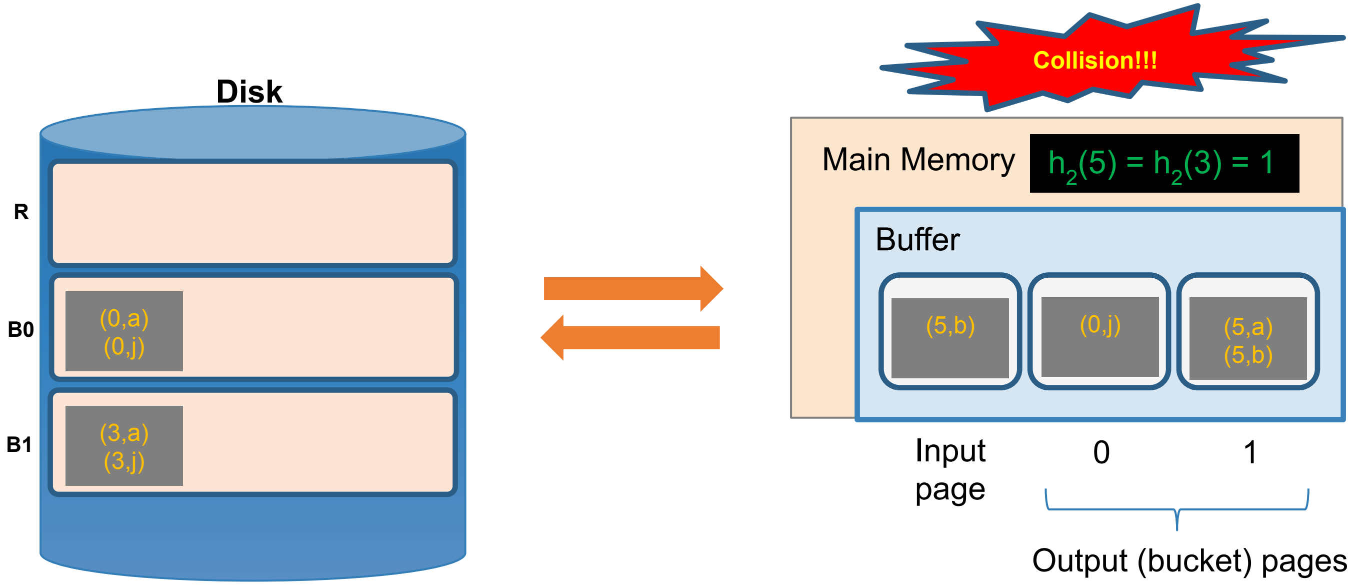
Given ***B+1 = 3*** buffer pages



HPJ Phase 1: Partitioning

Finish this pass…

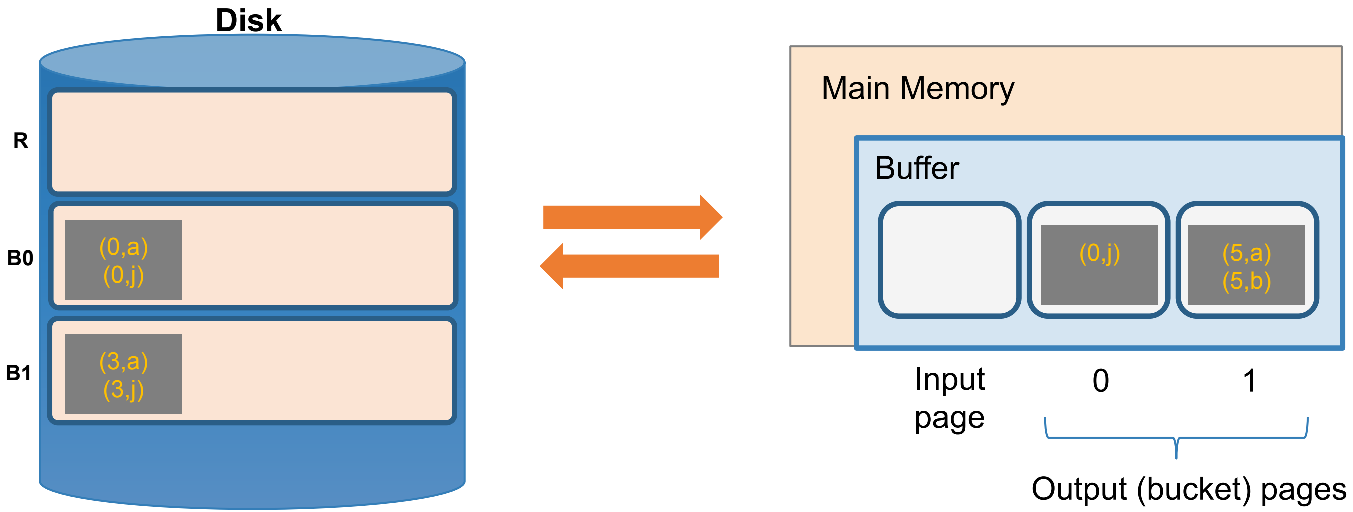
Given ***B+1 = 3*** buffer pages



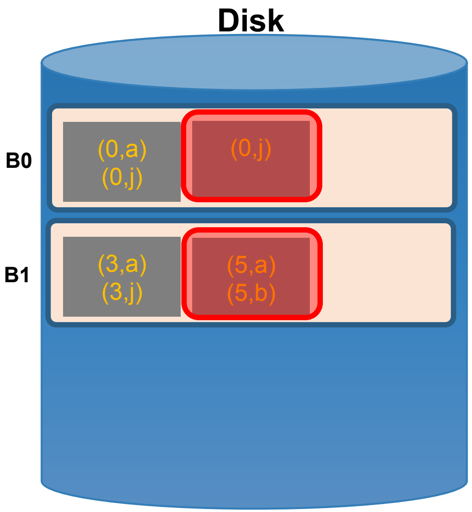
HPJ Phase 1: Partitioning

Finish this pass…

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



HPJ Phase 1: Partitioning



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

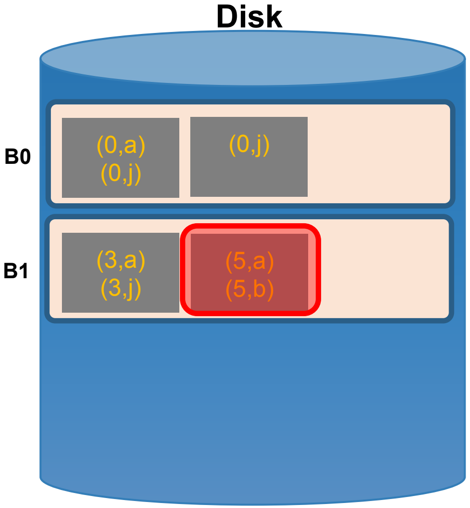
(1) Duplicate join keys

(2) Hash collisions

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

HPJ Phase 1: Partitioning

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



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

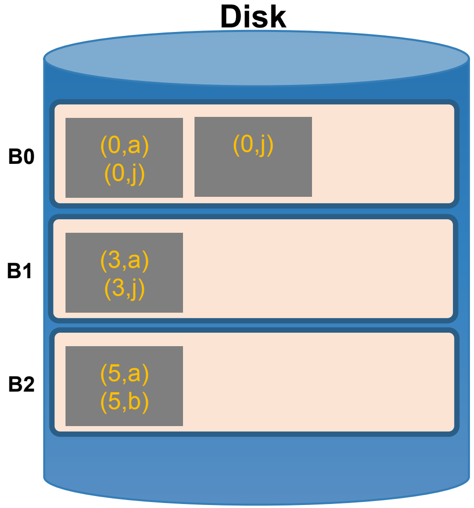
What hash function should we use?

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

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

HPJ Phase 1: Partitioning

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



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

What hash function should we use?

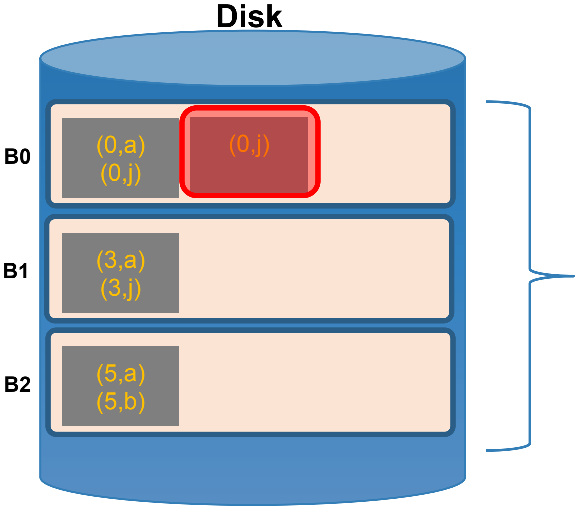
Do another pass with a different hash function, h'_2, ideally such that:

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

HPJ Phase 1: Partitioning

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

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

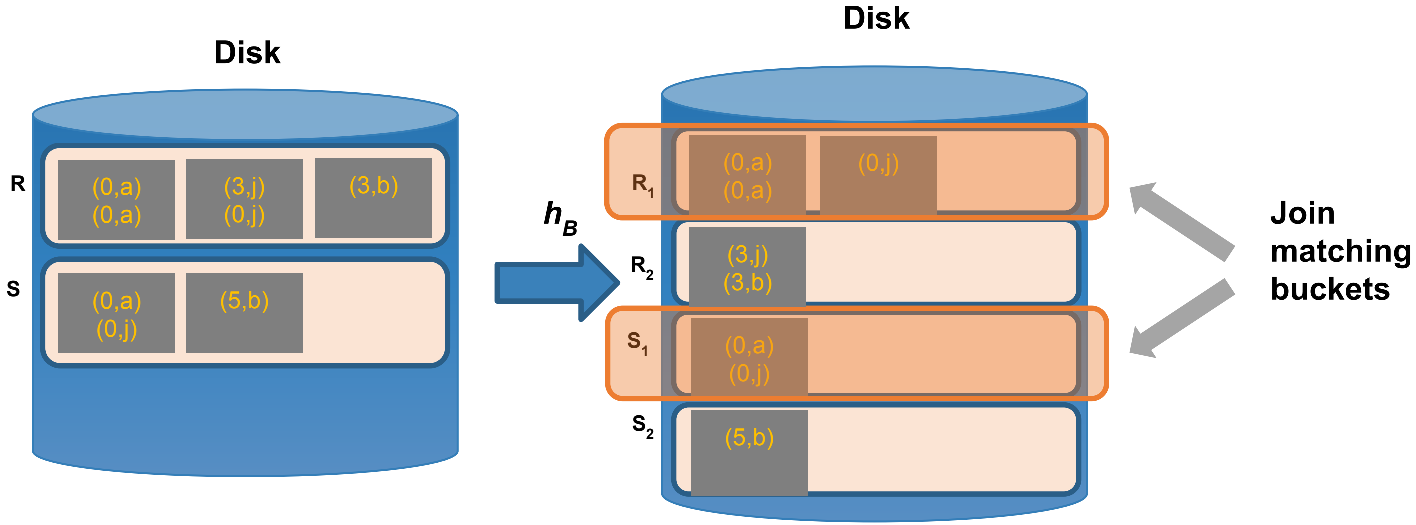


We call this unevenness in the bucket size ***skew***

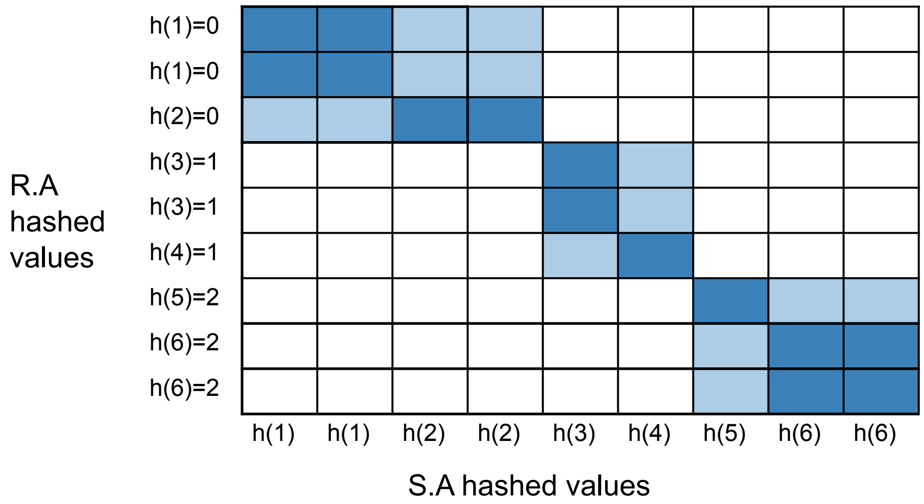
Now that we have partitioned R and S…

HPJ Phase 2: Partition Join

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

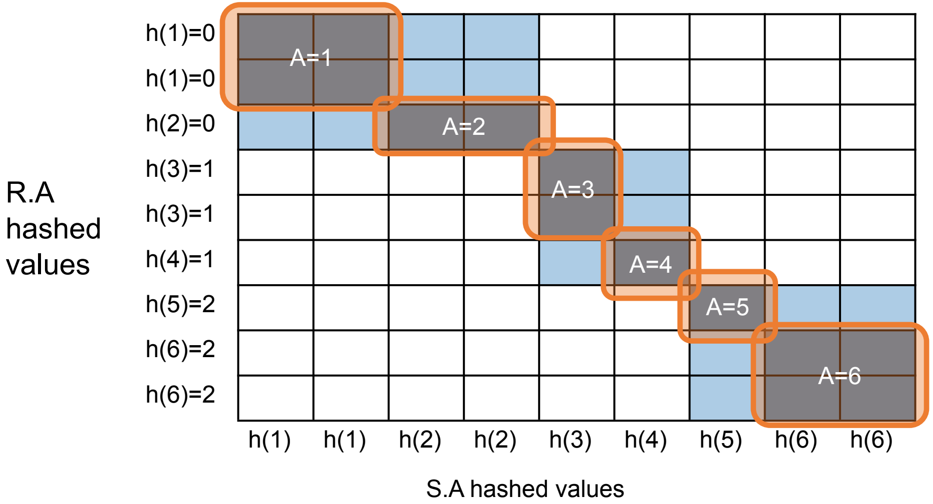


HPJ Phase 2: Partition Join



$R \bowtie S \ on \ A$

HPJ Phase 2: Partition Join

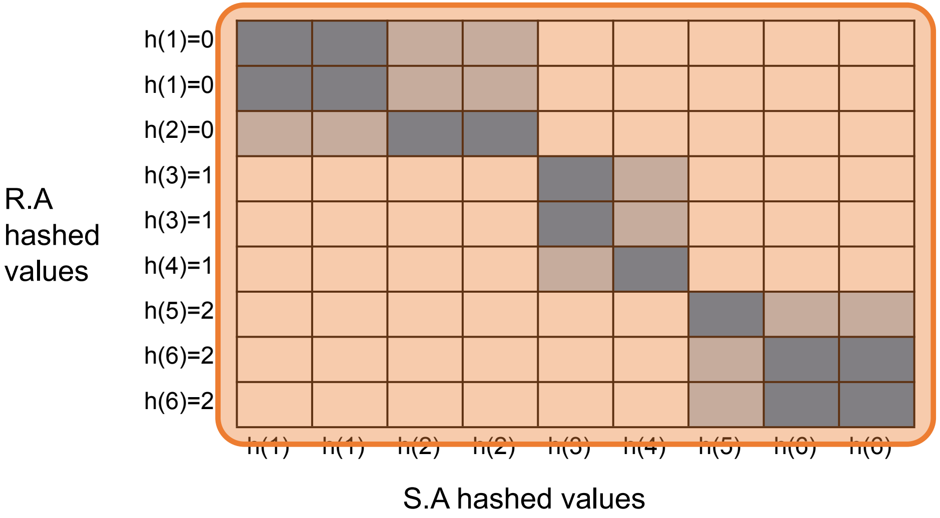


$R \bowtie S \text{ on } A$

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

= the tuples with same values of the join key A

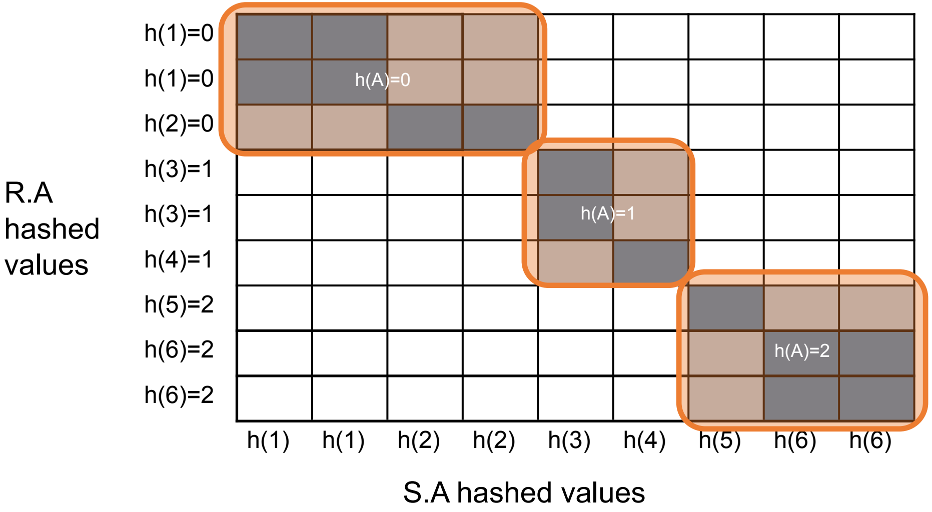
HPJ Phase 2: Partition Join



$R \bowtie S$ *on A*

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

HPJ Phase 2: Partition Join



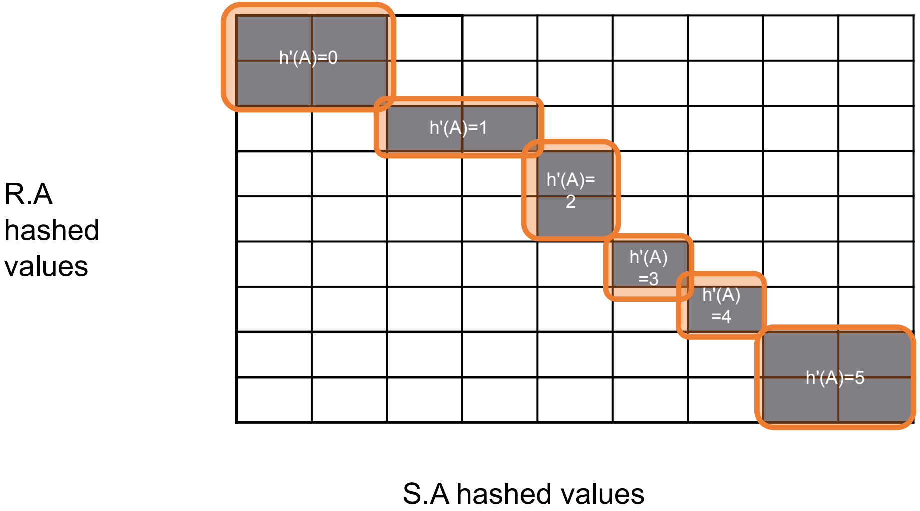
$R \bowtie S$ *on A*

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

HPJ Phase 2: Partition Join



$R \bowtie S$ on A

An alternative to applying BNLJ:

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

HPJ Summary

Given enough buffer pages…

- **Hash Partition** requires reading + writing each page of R,S
  - → 2(P(R)+P(S)) IOs
- **Partition Join** (with BNLJ) requires reading each page of R,S
  - → P(R) + P(S) IOs
- **Writing out results** could be as bad as P(R)*P(S)… but probably closer to P(R)+P(S)

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

SMJ vs HPJ Joins Summary

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

$$\sim 3(P(R)+P(S)) + \mathit{OUT}$$

- Hash Joins are highly parallelizable

- Sort-Merge less sensitive to data skew and result is sorted

⇒ <u>Big takeaway</u>: IO-aware join algorithms

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

Histograms & IO Cost Estimation

Optimization

Roadmap

Build Query Plans

Analyze Plans

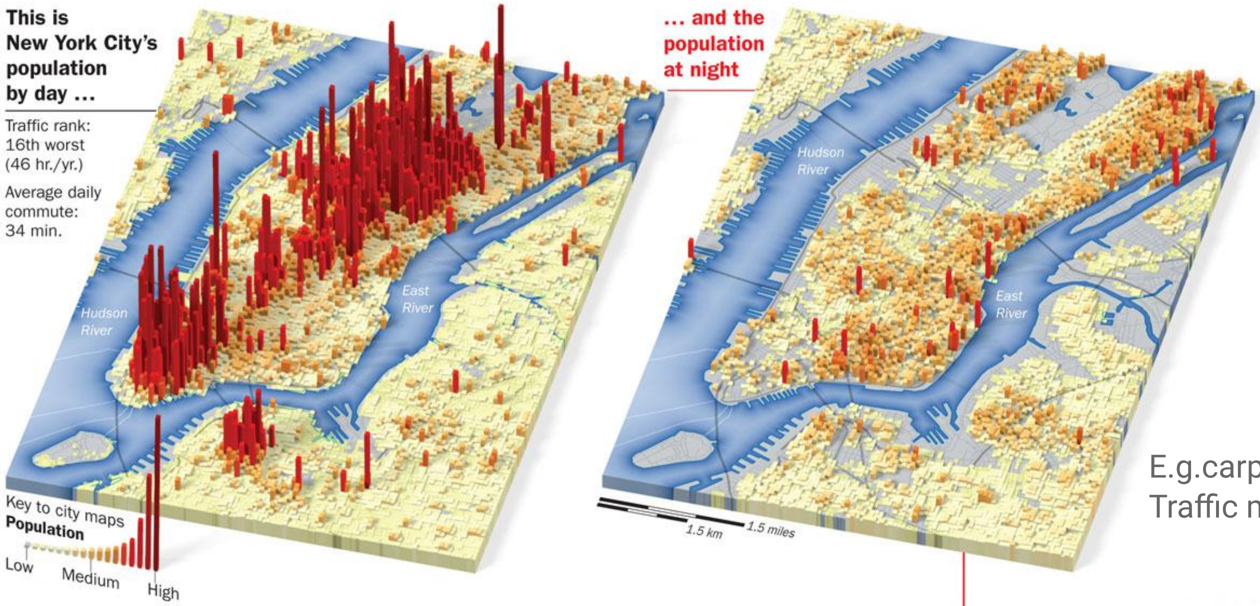
1. For SFW, Joins queries
   a. Brute-force? Sort? Hash? Count?
   b. Pre-build an index? B+ tree, Hash?

2. What statistics can I keep to optimize?
   a. E.g. Selectivity of columns, values

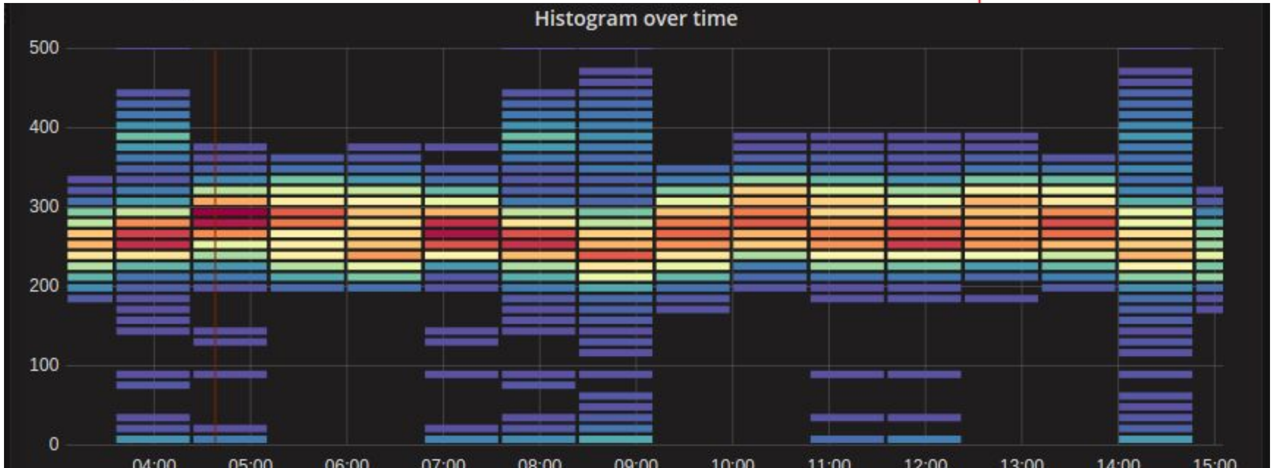
Cost in I/O, resources?
To query, maintain?

Example

Stats for spatial and temporal data



E.g.carpool or Traffic model



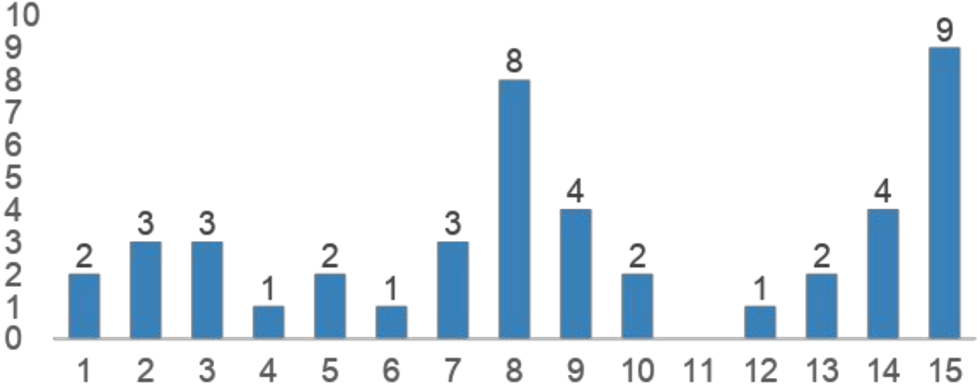
E.g. user traffic

Histograms

- A histogram is a set of value ranges ("buckets") and the frequencies of values in those buckets
- How to choose the buckets?
  - Equi-width & Equi-depth
- High-frequency values are **very** important(e.g, related products)

Example

Frequency

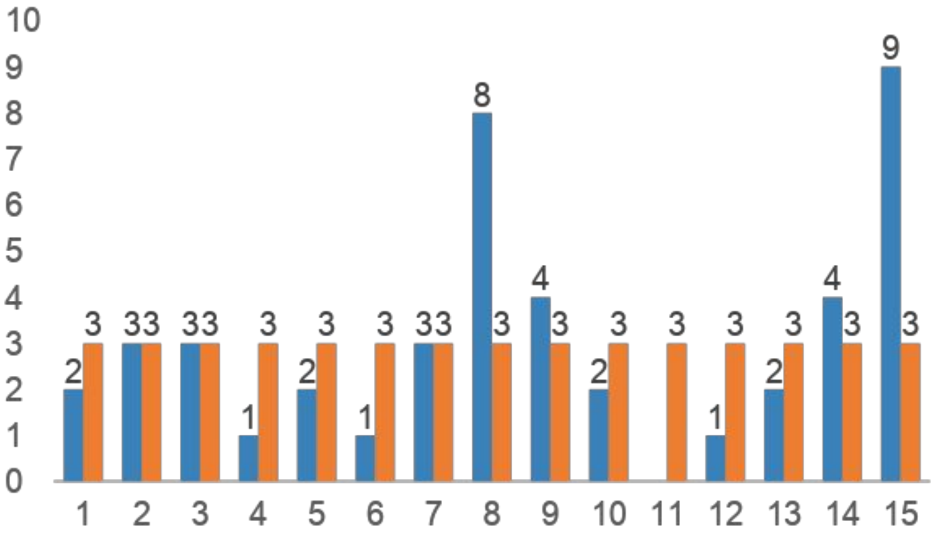


Values

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

Problem: counts take up too much space!

Full vs. Uniform Counts



How much space do the full counts (bucket_size=1) take?

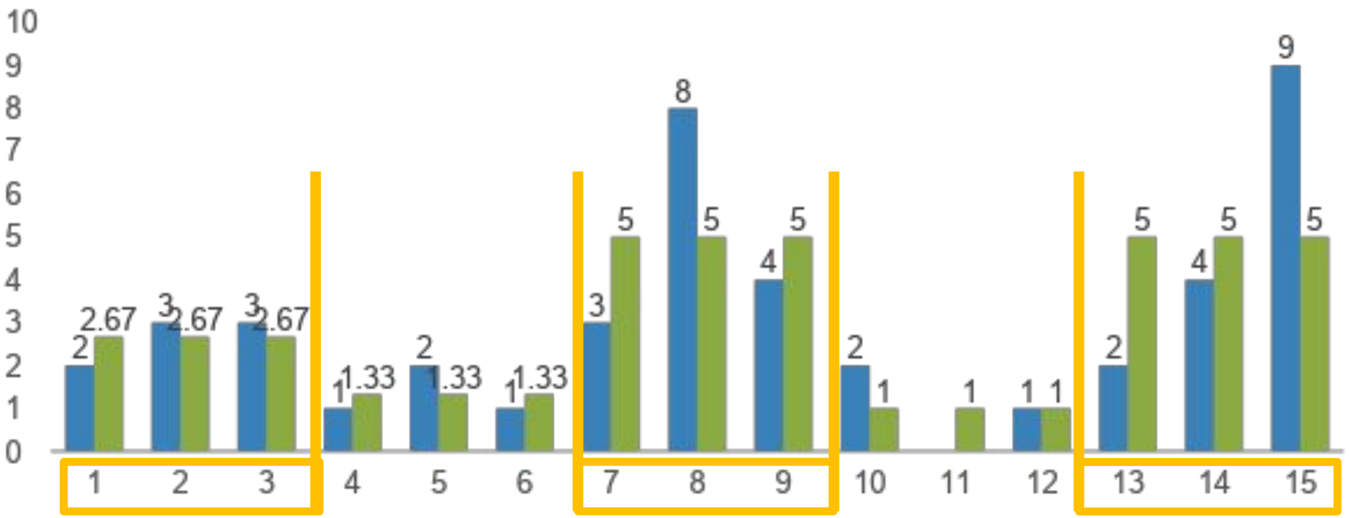
How much space do the uniform counts (bucket_size=ALL) take?

Fundamental Tradeoffs

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

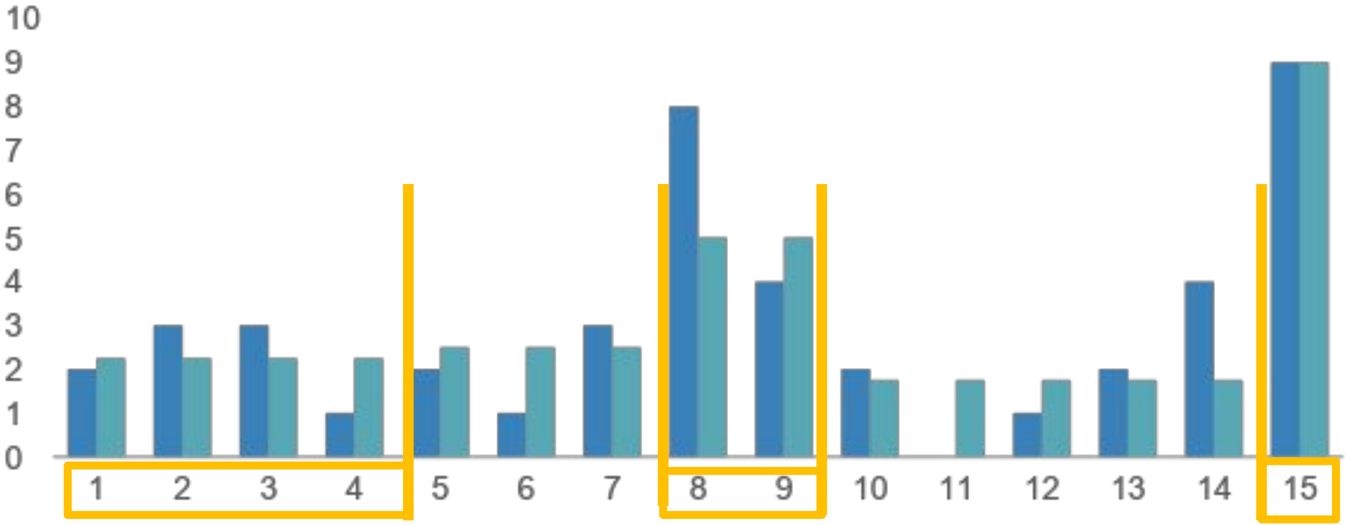
So how do we compute the “bucket” sizes?

Equi-width



Partition buckets into roughly same width (value range)

Equi-depth



Partition buckets for roughly 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!

- Research on self-tuning histograms and the use of query feedback

Compressed Histograms

One popular approach

1. Store the most frequent values and their counts explicitly
2. Keep an equiwidth or equidepth one for the rest of the values

People continue to try all manner of fanciness here
wavelets, graphical models, entropy models,…

THANK YOU!