

Disclaimer:
These slides are from a
University of Wisconsin
Database Course available
online

What you will learn about in this section

1. Sort-Merge Join (SMJ)

2. Hash Join (HJ)

3. SMJ vs. HJ

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

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]

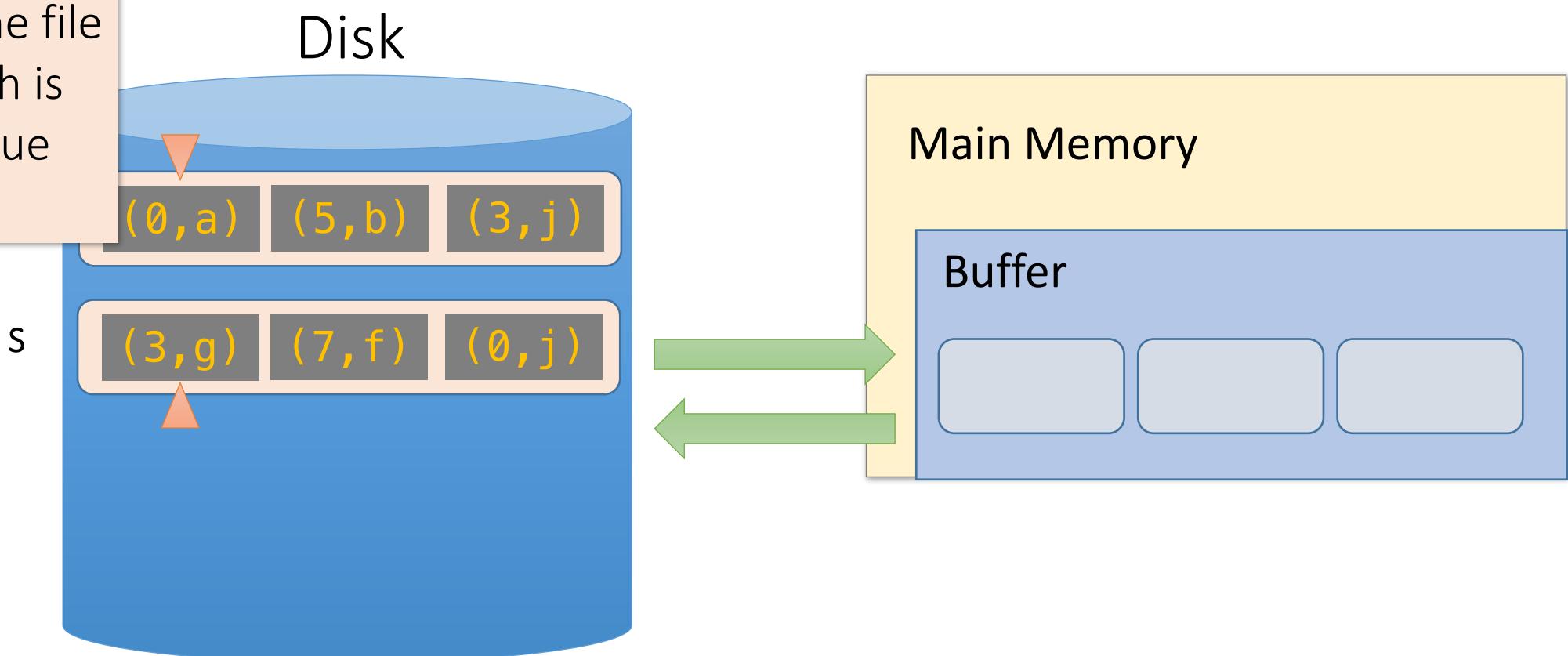
Note that we are only considering equality join conditions here

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

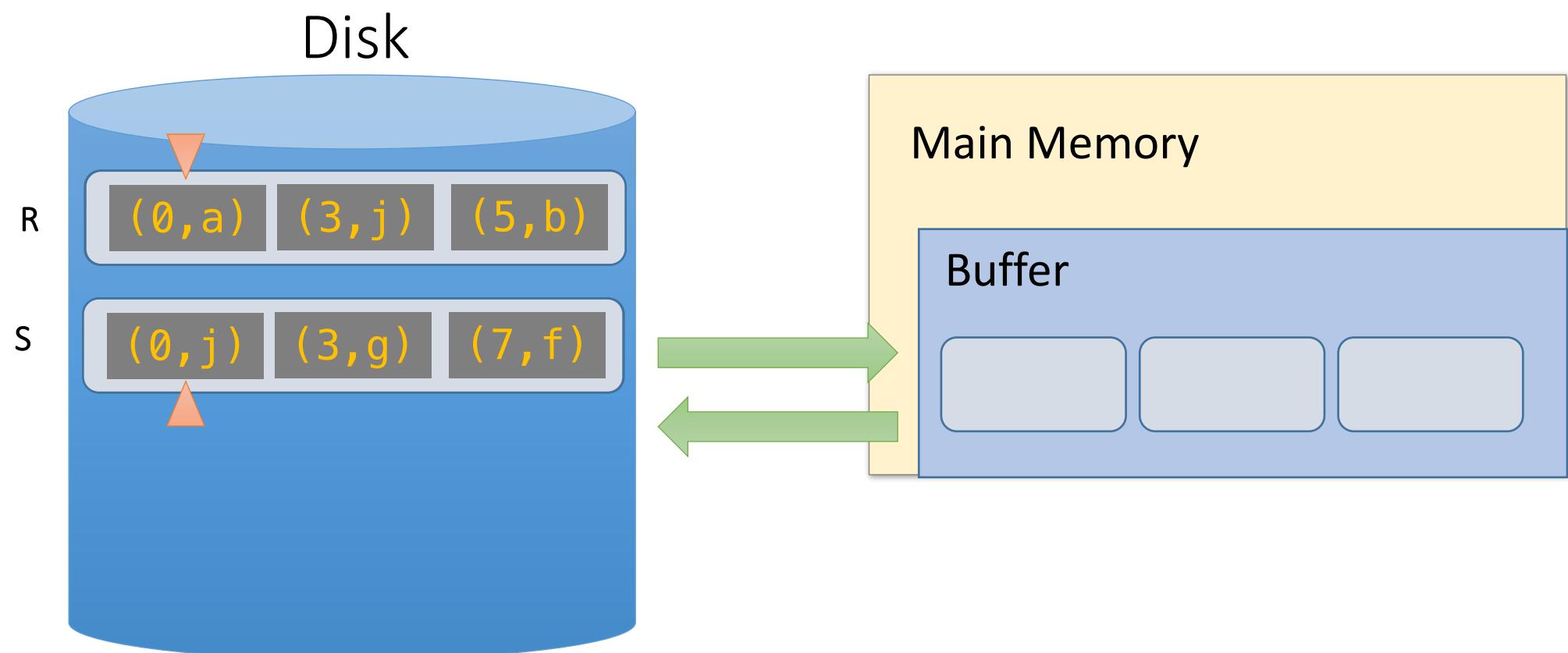
- For simplicity: Let each page be **one tuple**, and let the first value be A

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



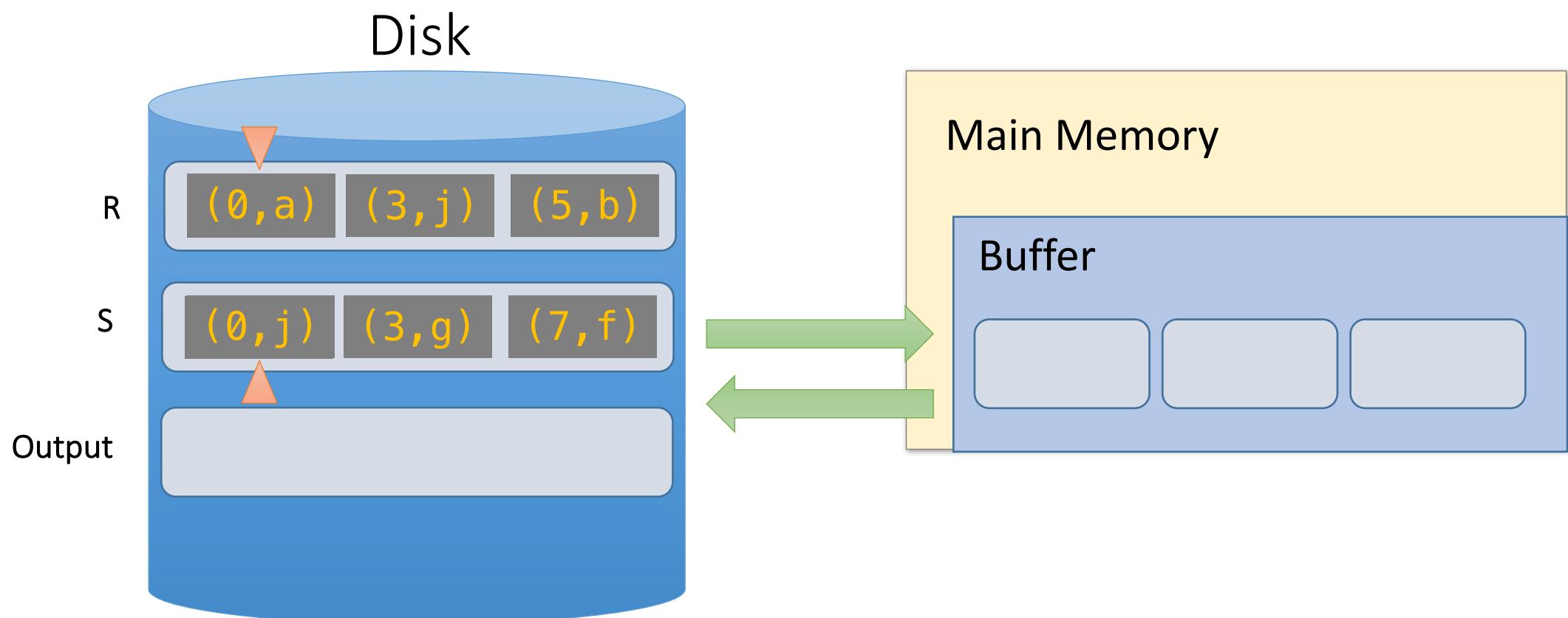
SMJ Example: $R \bowtie S$ on A with 3 page buffer

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



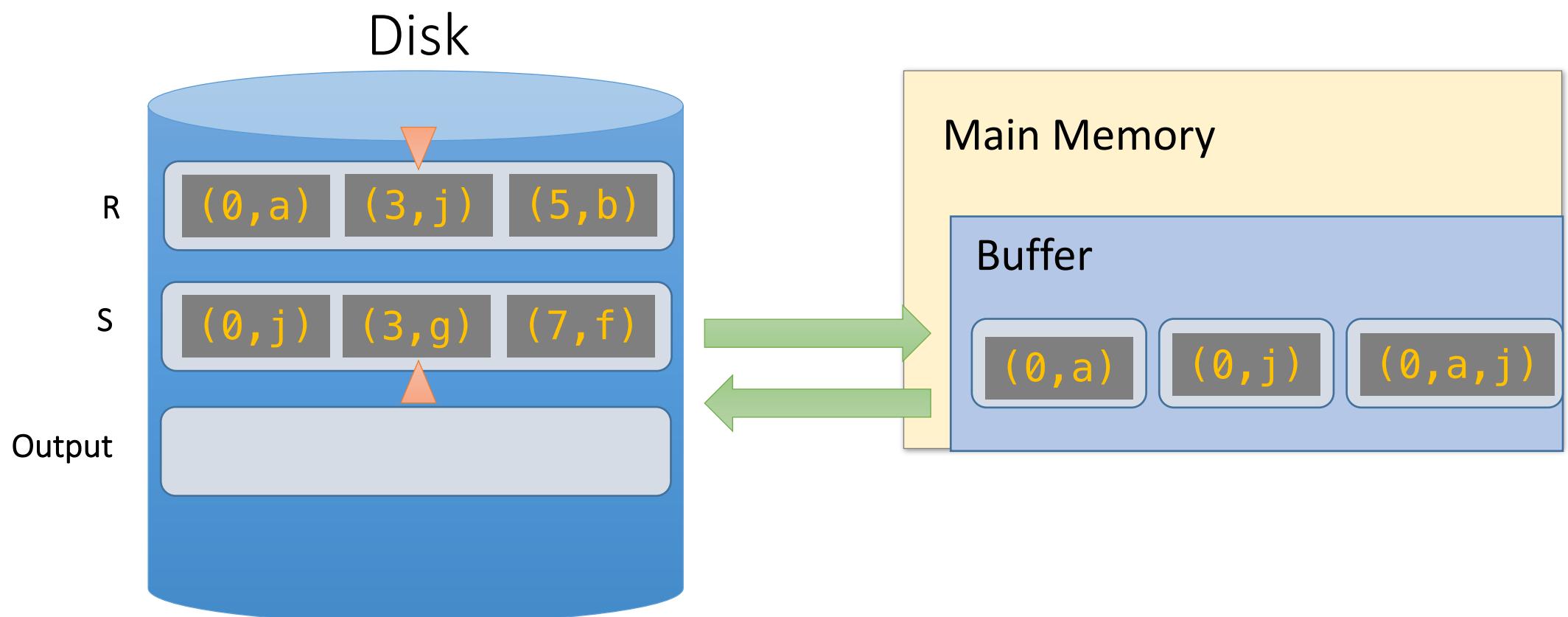
SMJ Example: $R \bowtie S$ on A with 3 page buffer

2. Scan and “merge” on join key!



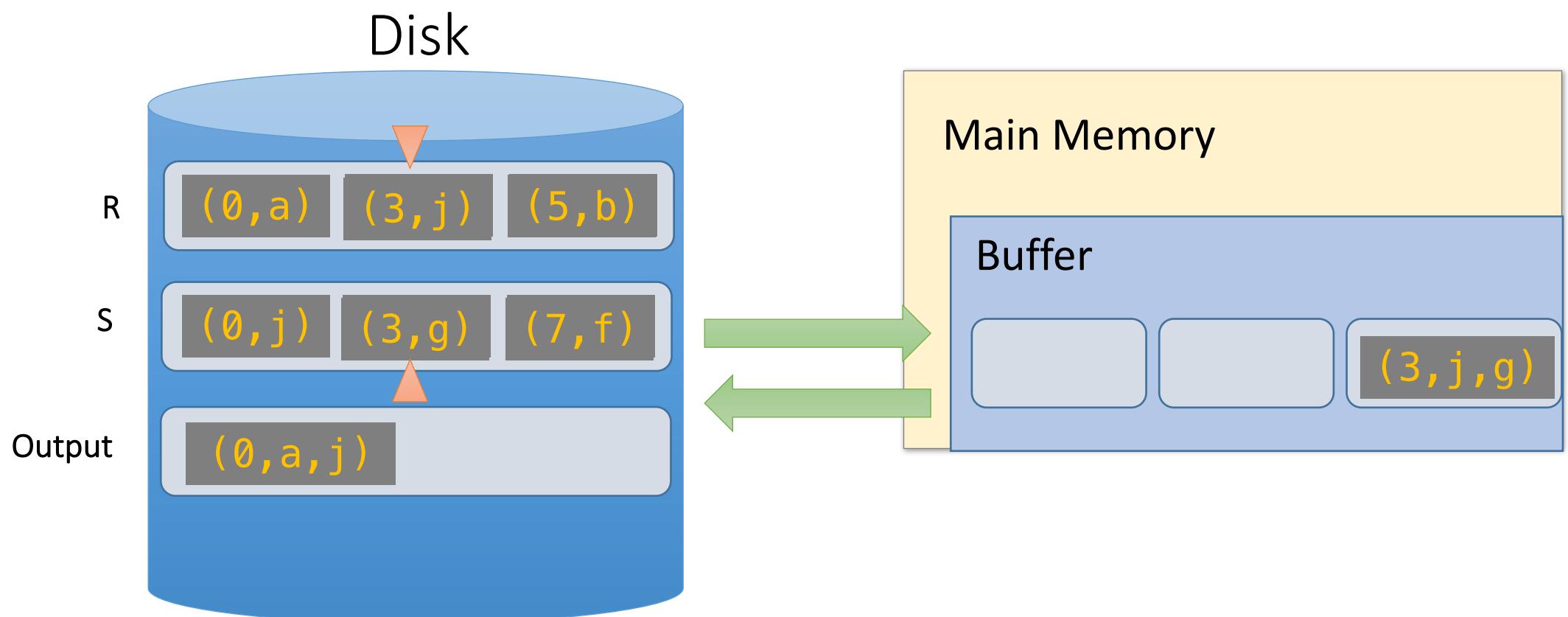
SMJ Example: $R \bowtie S$ on A with 3 page buffer

2. Scan and “merge” on join key!



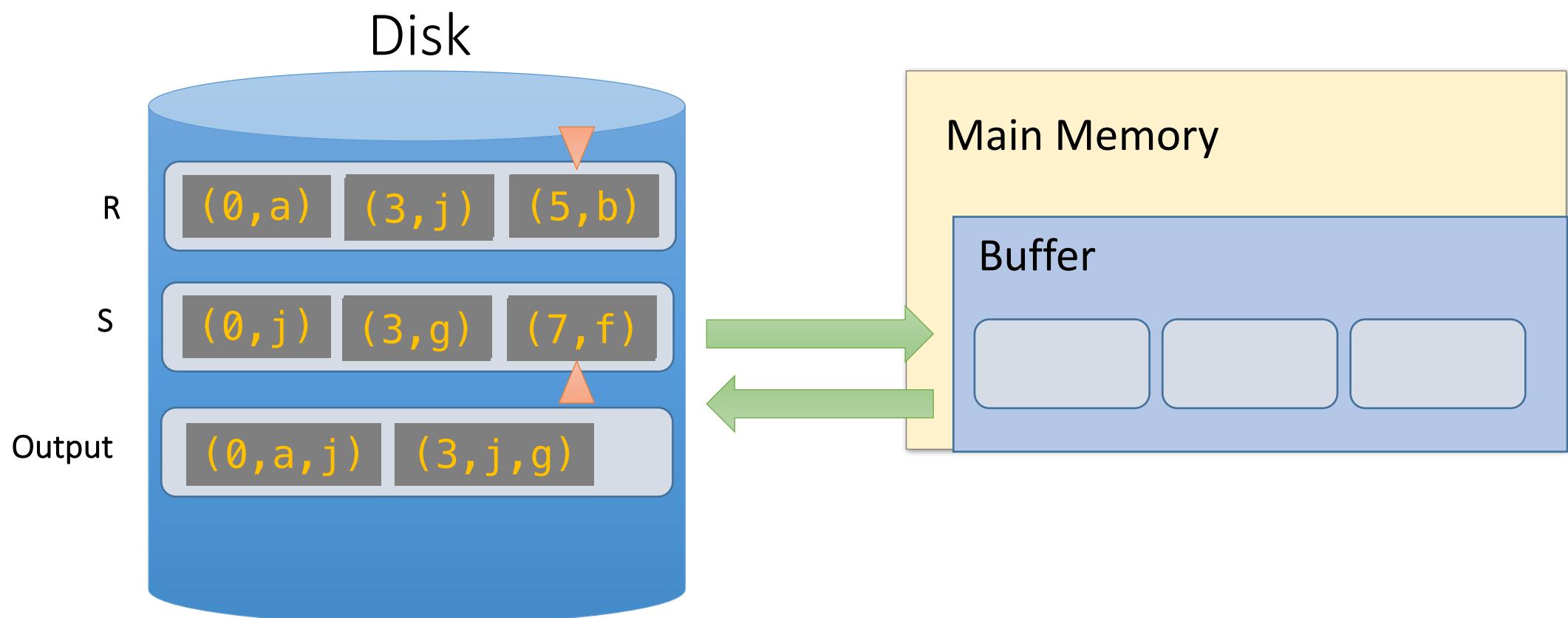
SMJ Example: $R \bowtie S$ on A with 3 page buffer

2. Scan and “merge” on join key!



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

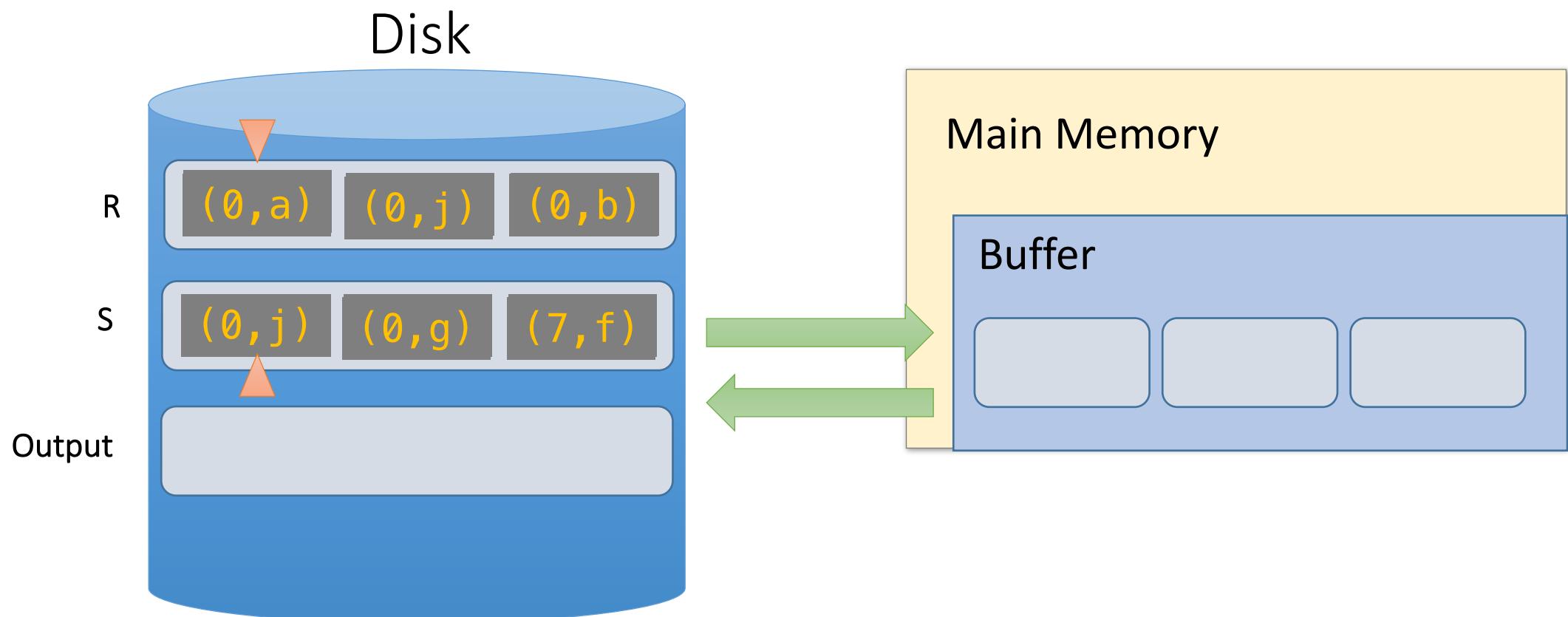
2. Done!



What happens with duplicate join keys?

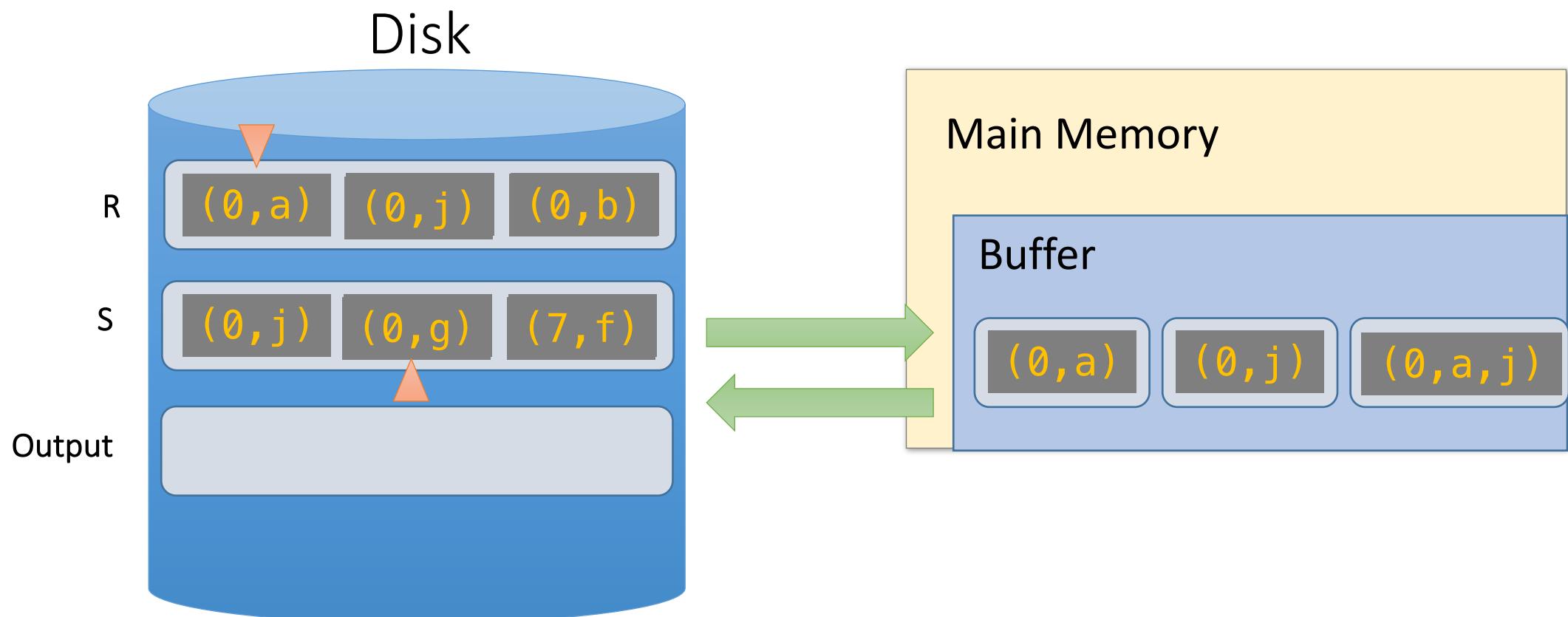
Multiple tuples with Same Join Key: “Backup”

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



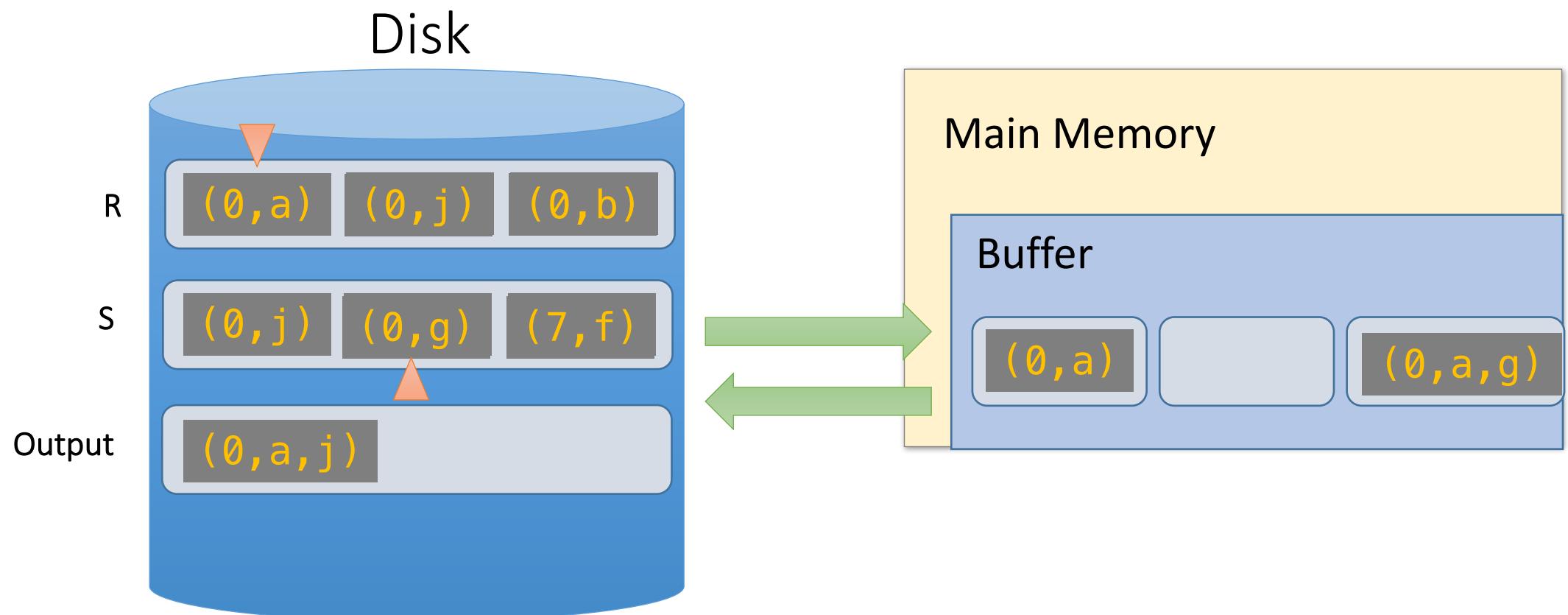
Multiple tuples with Same Join Key: “Backup”

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



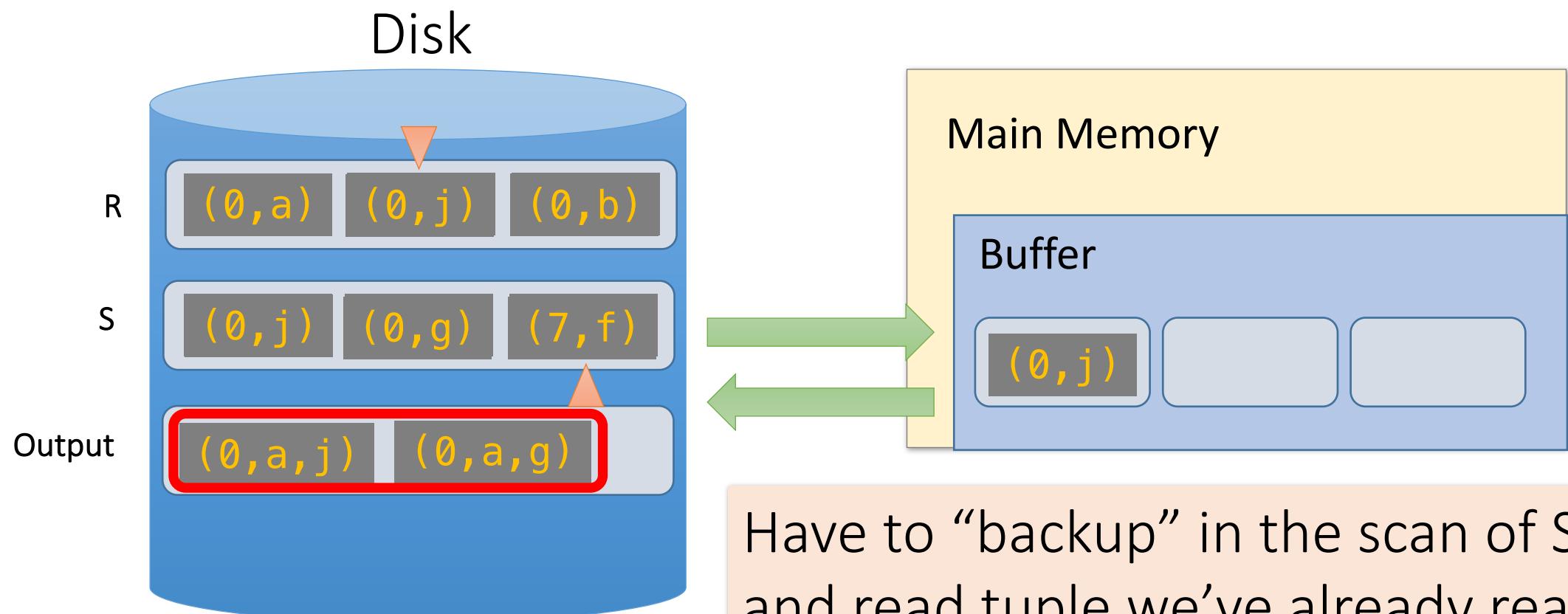
Multiple tuples with Same Join Key: “Backup”

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...



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, plus we can:
 - Leave more data in buffer (for larger buffers)
 - Can “zig-zag” (see animation)

SMJ: Total cost

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

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

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

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

SMJ vs. BNLJ: Steel Cage Match

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

What is BNLJ?

- $500 + 1000 * \left\lceil \frac{500}{98} \right\rceil = \mathbf{\underline{6,500 IOs + OUT}}$
- But, if we have 35 buffer pages?
 - Sort Merge has same behavior (still 2 passes)
 - BNLJ? 15,500 IOs + OUT!

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

A Simple Optimization: Merges Merged!

Given $B+1$ buffer pages

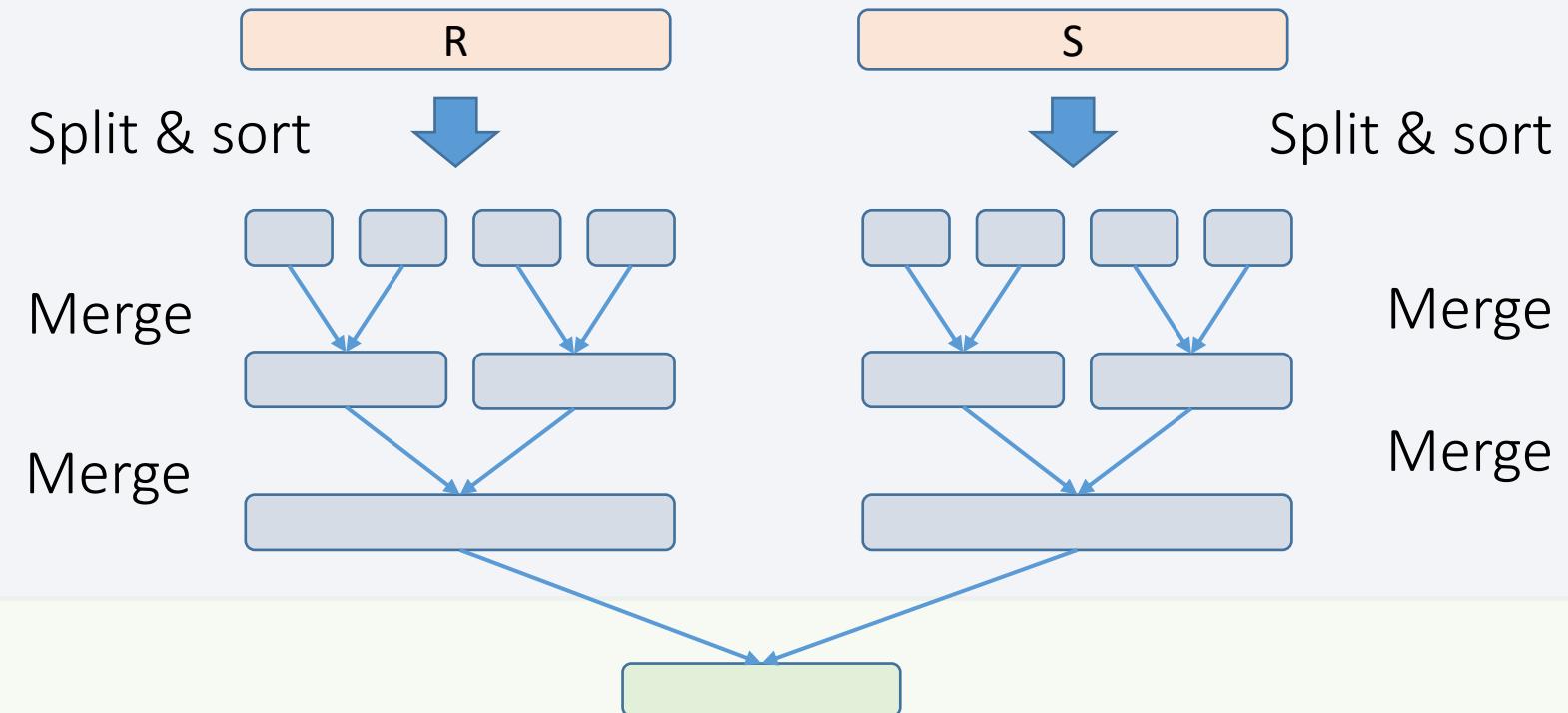
- SMJ is composed of a ***sort phase*** and a ***merge phase***
- During the ***sort phase***, run passes of external merge sort on R and S
 - Suppose at some point, R and S have $\leq B$ (sorted) runs in total
 - We could do two merges (for each of R & S) at this point, complete the sort phase, and start the merge phase...
 - OR, we could combine them: do **one** B-way merge and complete the join!

Un-Optimized SMJ

Given $B+1$ buffer pages

Sort Phase (Ext. Merge Sort)

Unsorted input relations

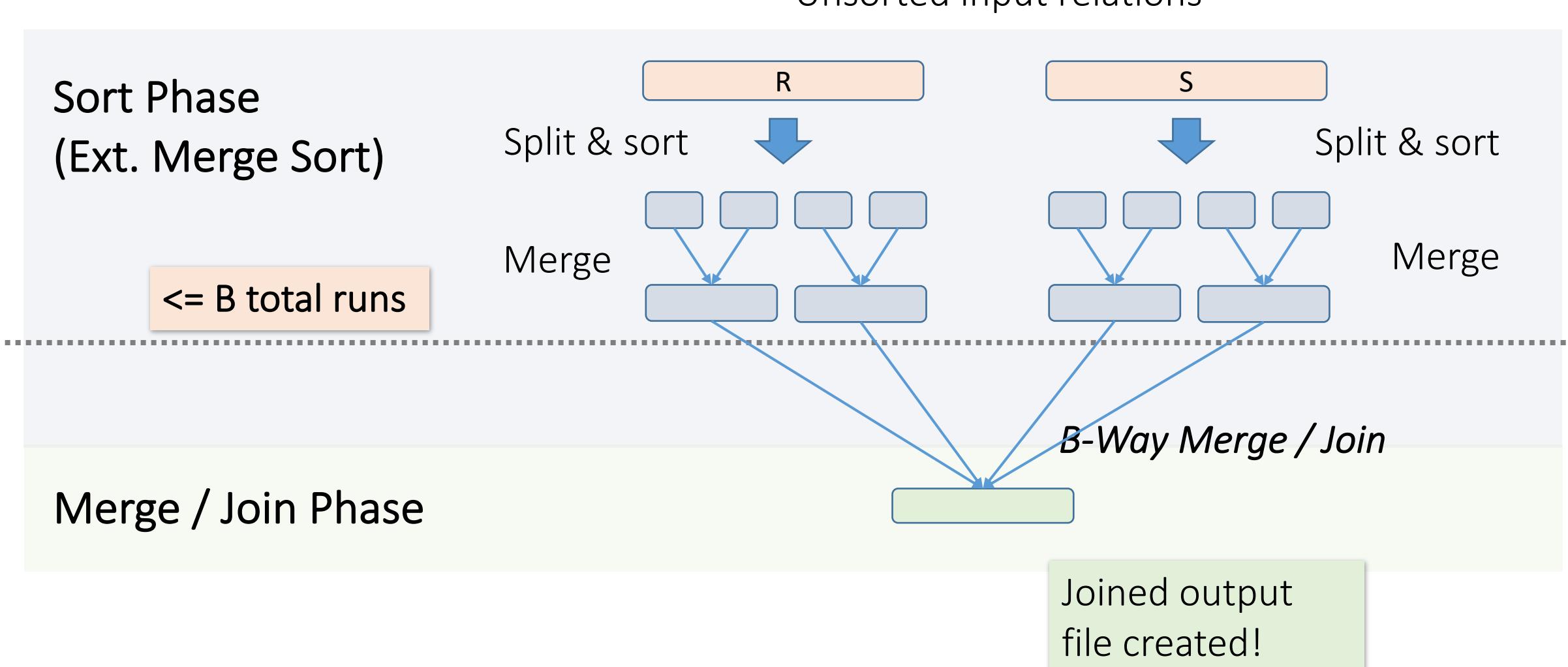


Merge / Join Phase

Joined output file created!

Simple SMJ Optimization

Given $B+1$ buffer pages



Simple SMJ Optimization

Given $B+1$ buffer pages

- Now, on this last pass, we only do $P(R) + P(S)$ IOs to complete the join!
- If we can initially split R and S into **B total runs each of length approx. $\leq 2(B+1)$** , *assuming repacking lets us create initial runs of $\sim 2(B+1)$* - then we only need **$3(P(R) + P(S)) + OUT$** for SMJ!
 - 2 R/W per page to sort runs in memory, 1 R per page to B-way merge / join!
- How much memory for this to happen?
 - $\frac{P(R)+P(S)}{B} \leq 2(B + 1) \Rightarrow \sim P(R) + P(S) \leq 2B^2$
 - **Thus, $\max\{P(R), P(S)\} \leq B^2$ is an approximate sufficient condition**

If the larger of R,S has $\leq B^2$ pages, then SMJ costs
 $3(P(R)+P(S)) + OUT!$

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

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 $x \neq y$ but $h_B(x) = h_B(y)$
 - Note however that it will never occur that $x = y$ but $h_B(x) \neq h_B(y)$
- We hash on an attribute A , so our hash function is $h_B(t)$ has the form $h_B(t.A)$.
 - **Collisions** may be more frequent.

Hash Join: High-level procedure

To compute $R \bowtie S$ on A :

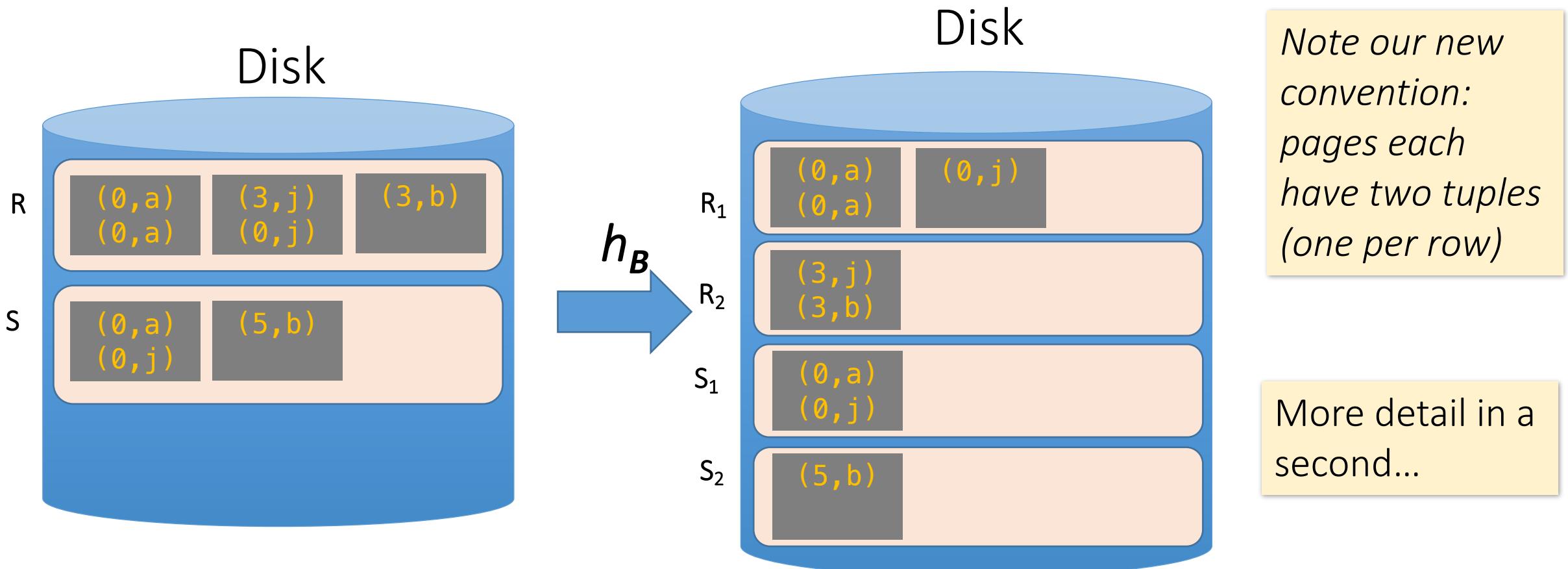
Note again that we are only considering equality constraints here

1. **Partition Phase:** Using one (shared) hash function 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!

We *decompose* the problem using h_B , then complete the join

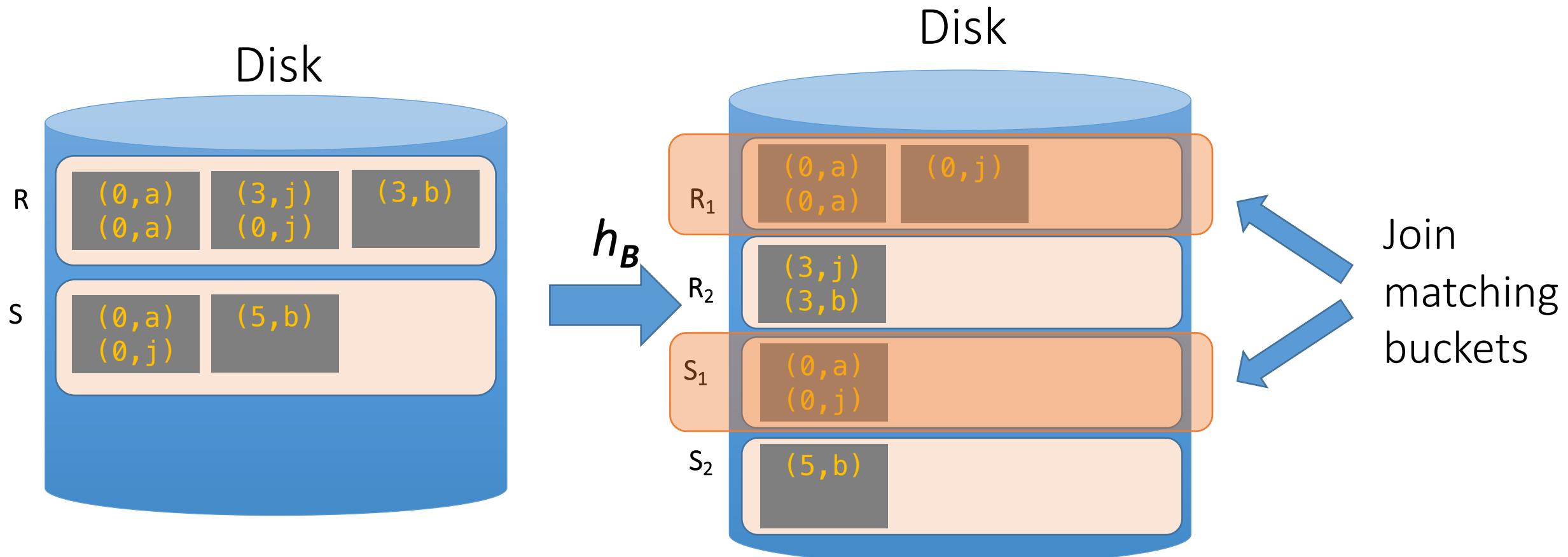
Hash Join: High-level procedure

1. Partition Phase: Using one (shared) hash function h_B , partition R and S into B buckets



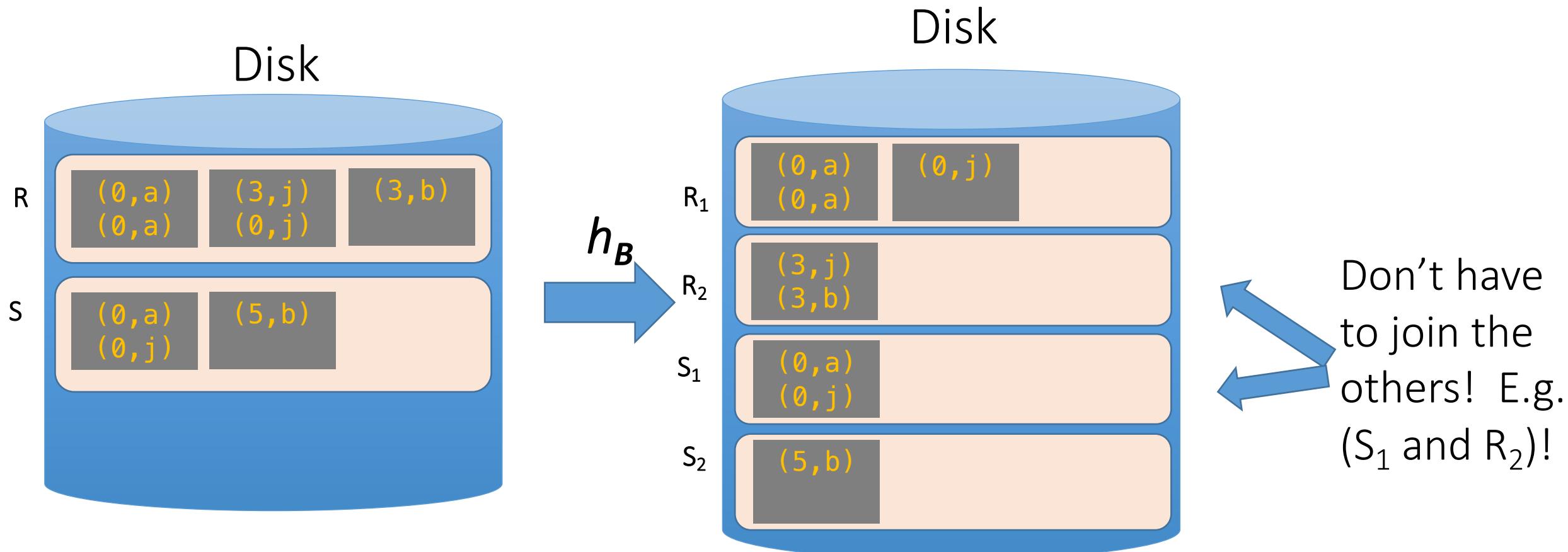
Hash Join: High-level procedure

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 $h_B(t.A) = h_B(t'.A)$ 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 $\sim N/B$ 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

How big do we want the resulting buckets?

- Ideally, our buckets would be of size $\leq B - 1$ pages
 - 1 for input page, 1 for output page, $B-1$ for each bucket
- 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)*, $P(R) \leq B - 1$!
 - And more generally, being able to fit bucket in memory is advantageous
- We can keep partitioning buckets that are $> B-1$ pages, until they are $\leq B - 1$ pages
 - Using a new hash key which will split them...

Given $B+1$ buffer pages

Recall for BNLJ:

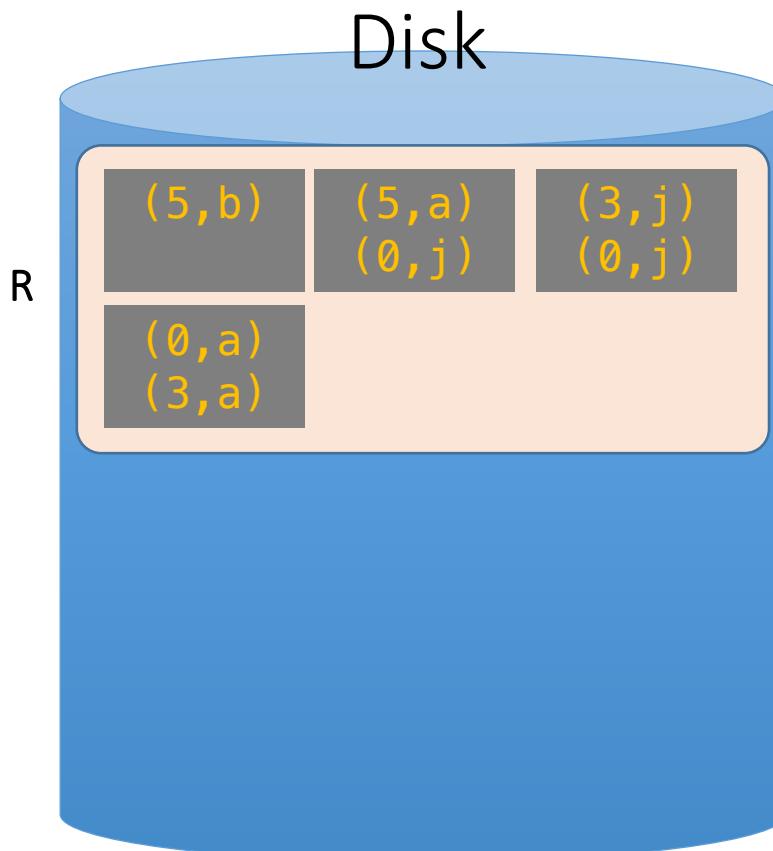
$$P(R) + \frac{P(R)P(S)}{B - 1}$$

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)



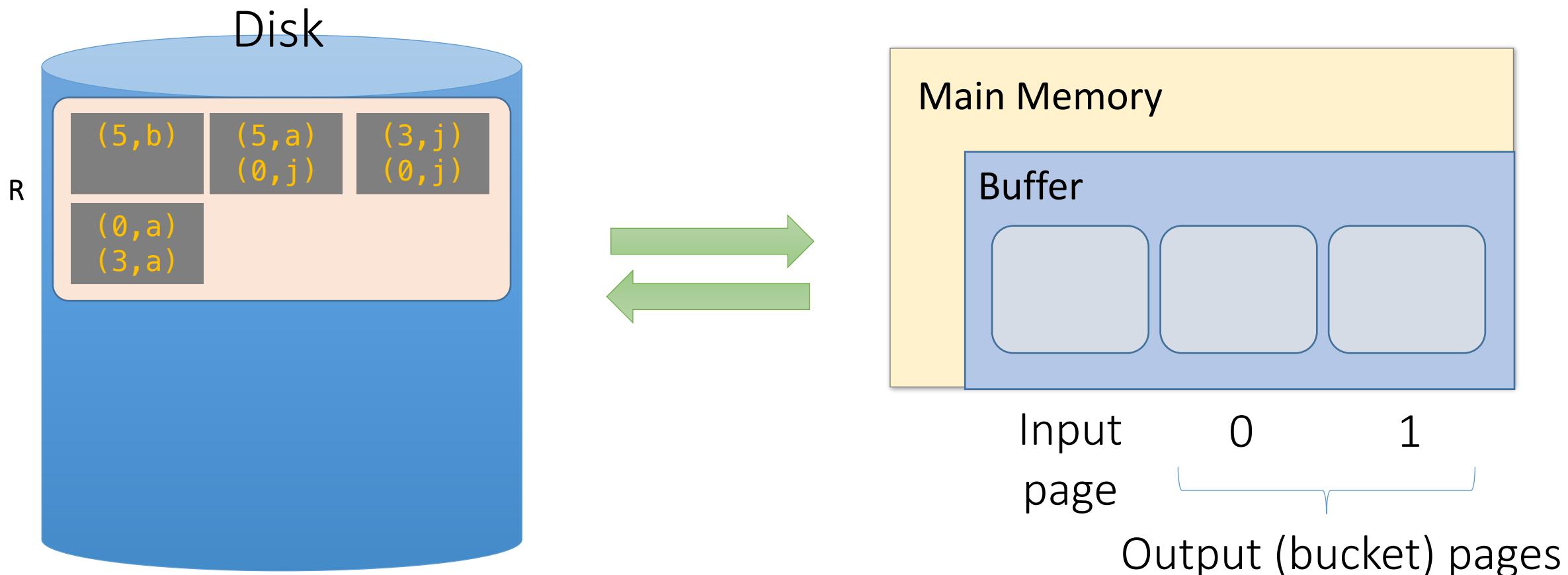
For simplicity, we'll look at partitioning one of the two relations- we just do the same for the other relation!

Recall: our goal will be to get $B = 2$ buckets of size $\leq B-1 \rightarrow 1$ page each

Hash Join Phase 1: Partitioning

Given $B+1 = 3$ buffer pages

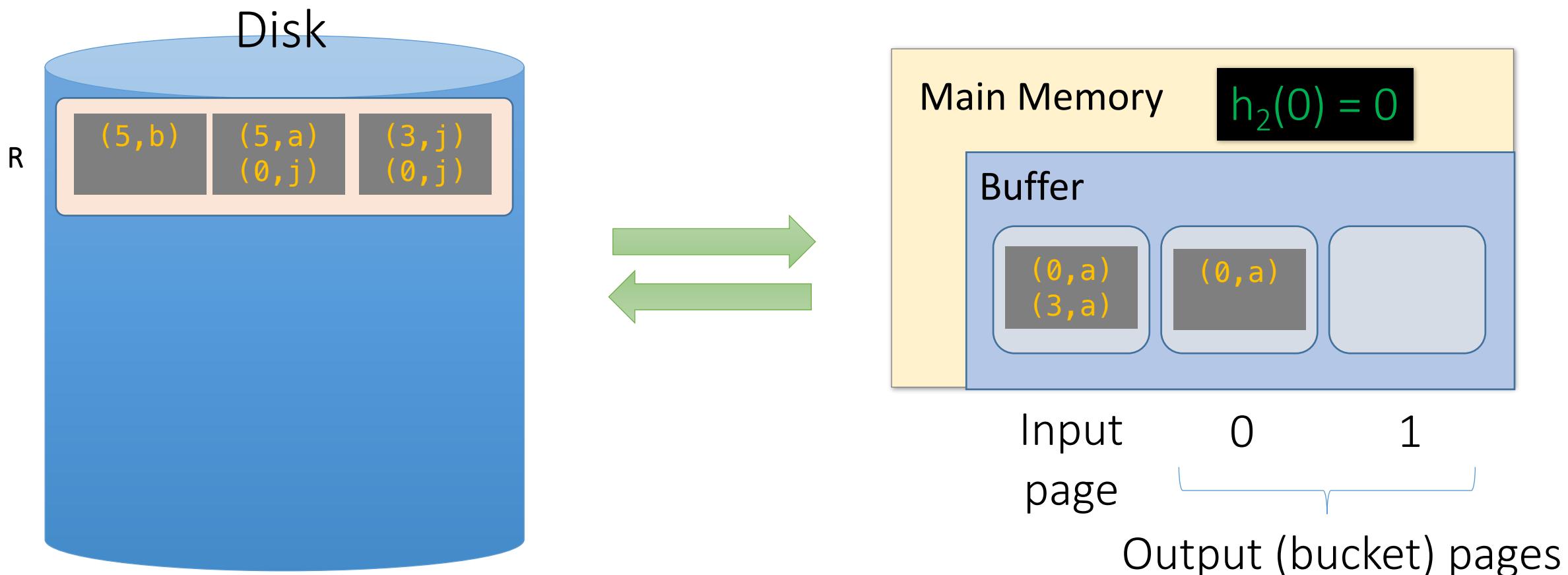
1. We read pages from R into the “input” page of the buffer...



Hash Join Phase 1: Partitioning

Given $B+1 = 3$ buffer pages

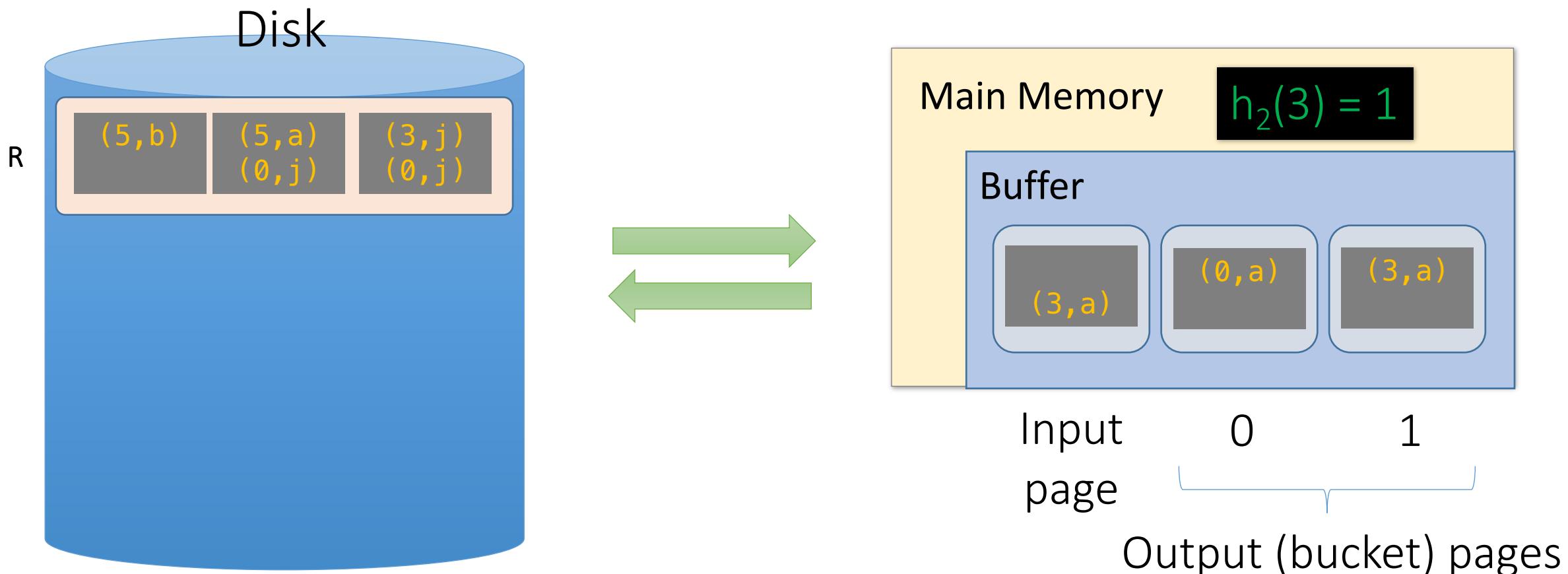
2. Then we use **hash function h_2** to sort into the buckets, which each have one page in the buffer



Hash Join Phase 1: Partitioning

Given $B+1 = 3$ buffer pages

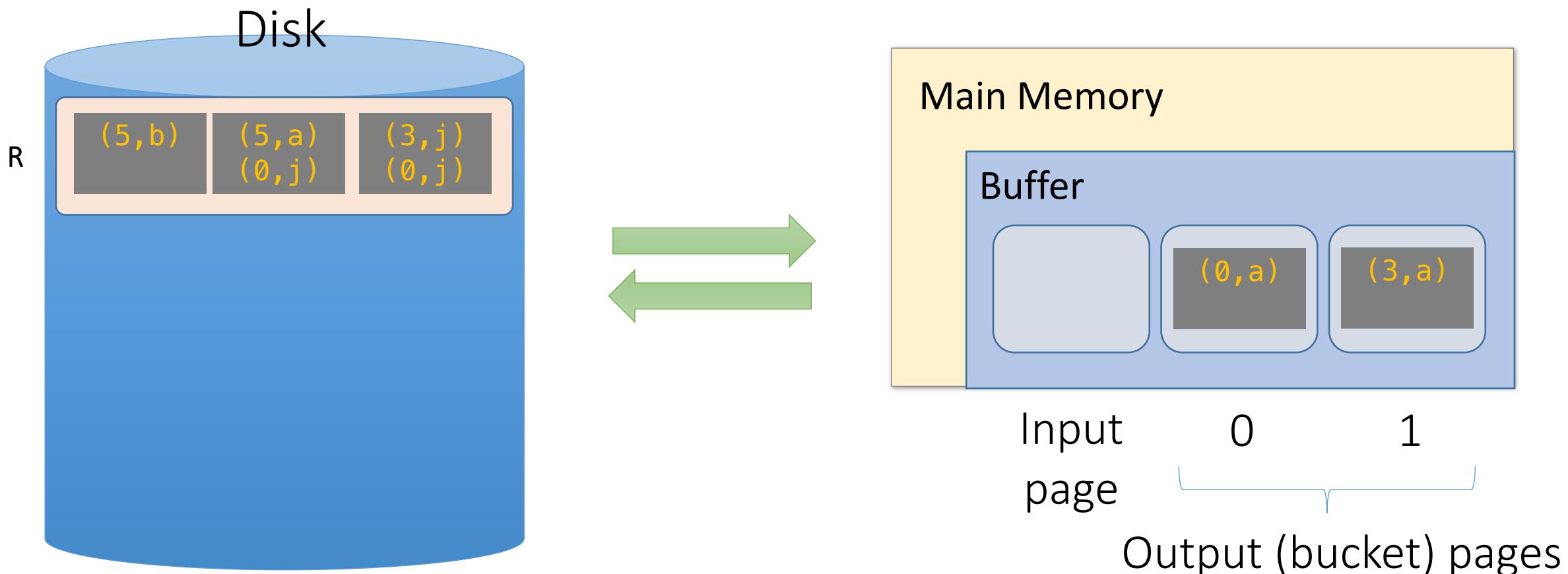
2. Then we use **hash function h_2** to sort into the buckets, which each have one page in the buffer



Hash Join Phase 1: Partitioning

Given $B+1 = 3$ buffer pages

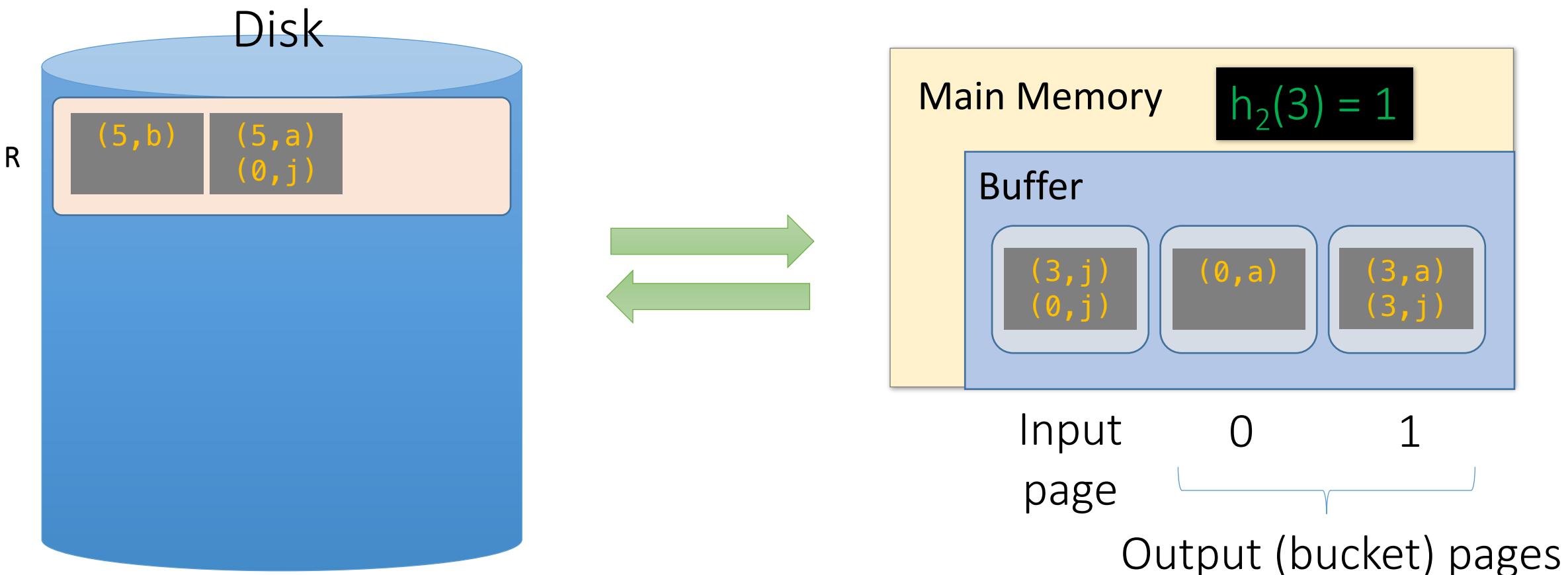
3. We repeat until the buffer bucket pages are full...



Hash Join Phase 1: Partitioning

Given $B+1 = 3$ buffer pages

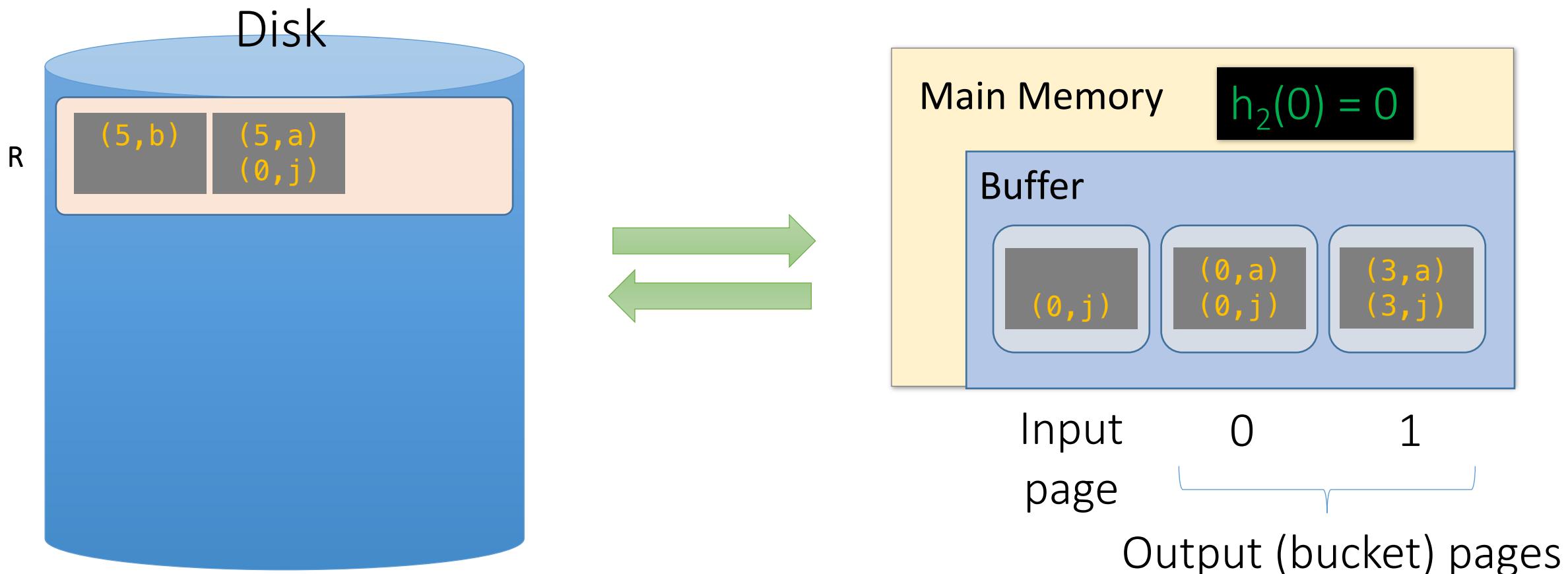
3. We repeat until the buffer bucket pages are full...



Hash Join Phase 1: Partitioning

Given $B+1 = 3$ buffer pages

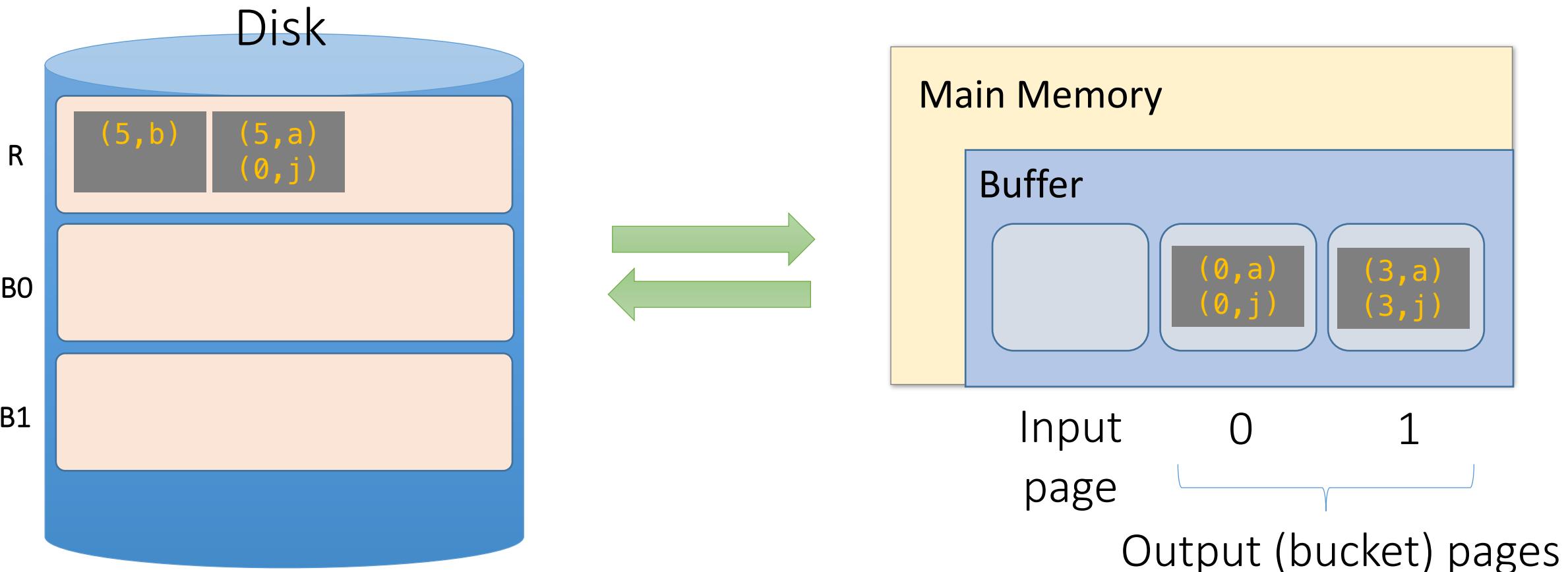
3. We repeat until the buffer bucket pages are full...



Hash Join Phase 1: Partitioning

Given $B+1 = 3$ buffer pages

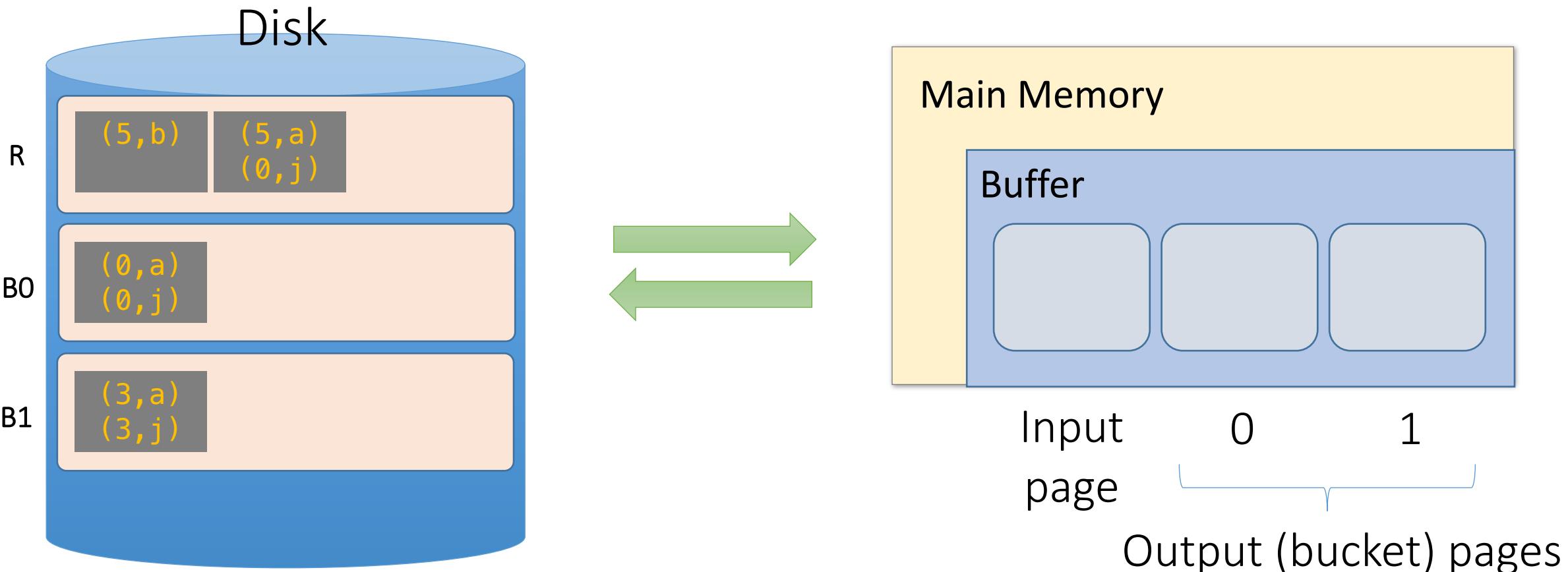
3. We repeat until the buffer bucket pages are full... then flush to disk



Hash Join Phase 1: Partitioning

Given $B+1 = 3$ buffer pages

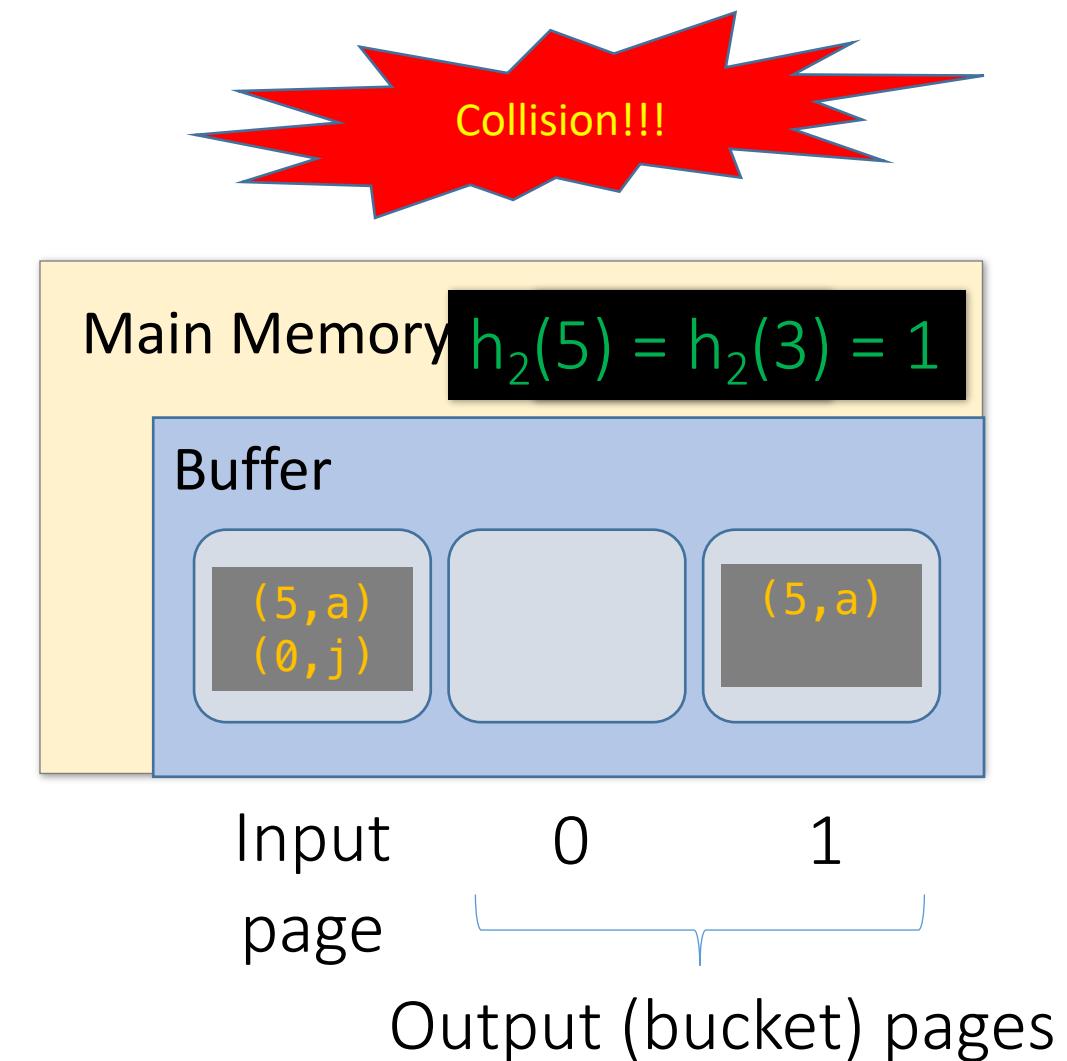
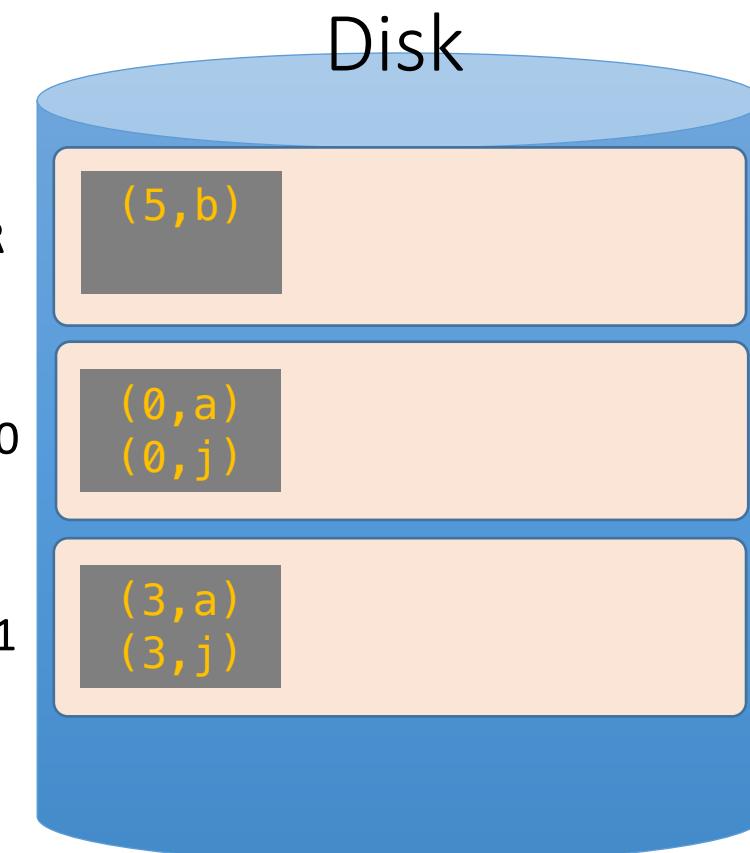
3. We repeat until the buffer bucket pages are full... then flush to disk



Hash Join Phase 1: Partitioning

Given $B+1 = 3$ buffer pages

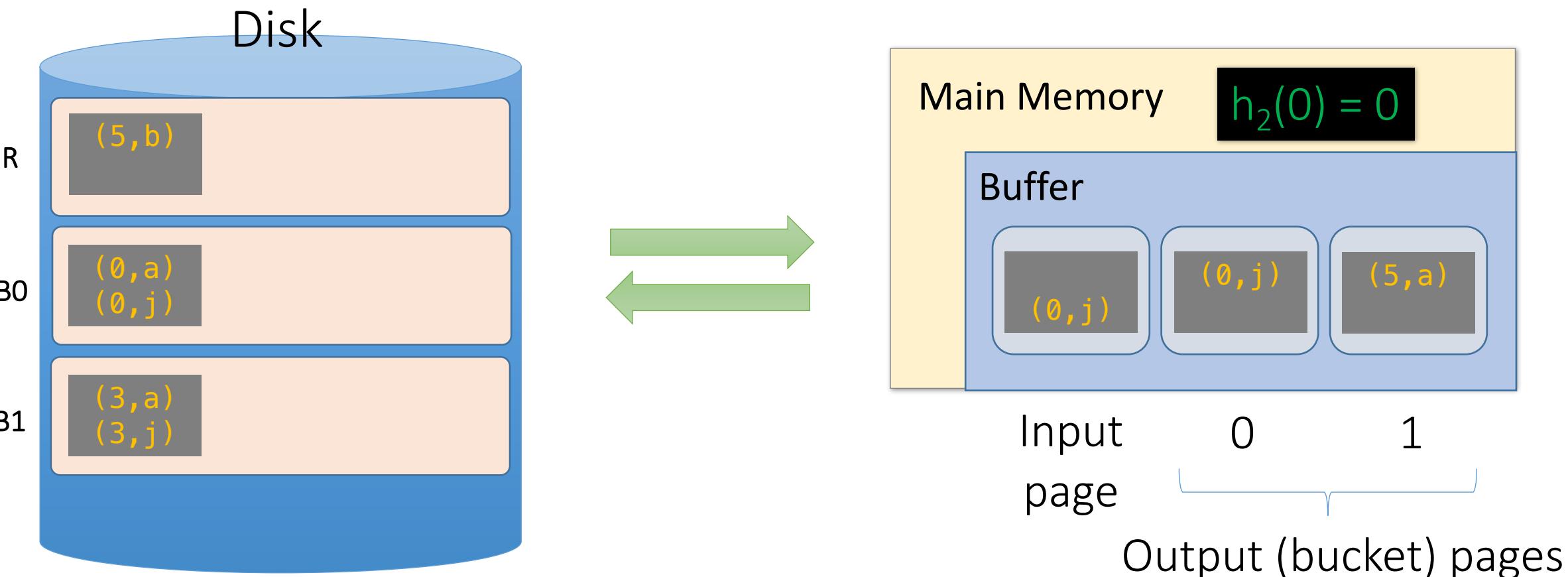
Note that collisions can occur!



Hash Join Phase 1: Partitioning

Given $B+1 = 3$ buffer pages

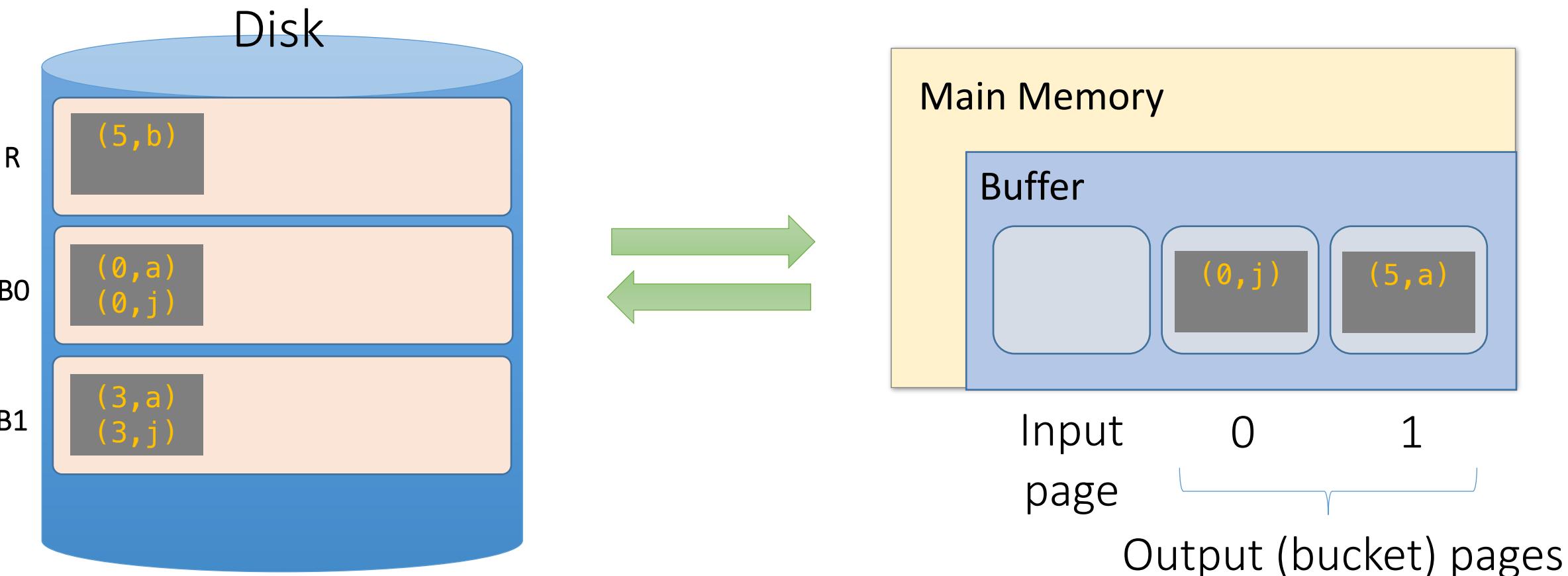
Finish this pass...



Hash Join Phase 1: Partitioning

Given $B+1 = 3$ buffer pages

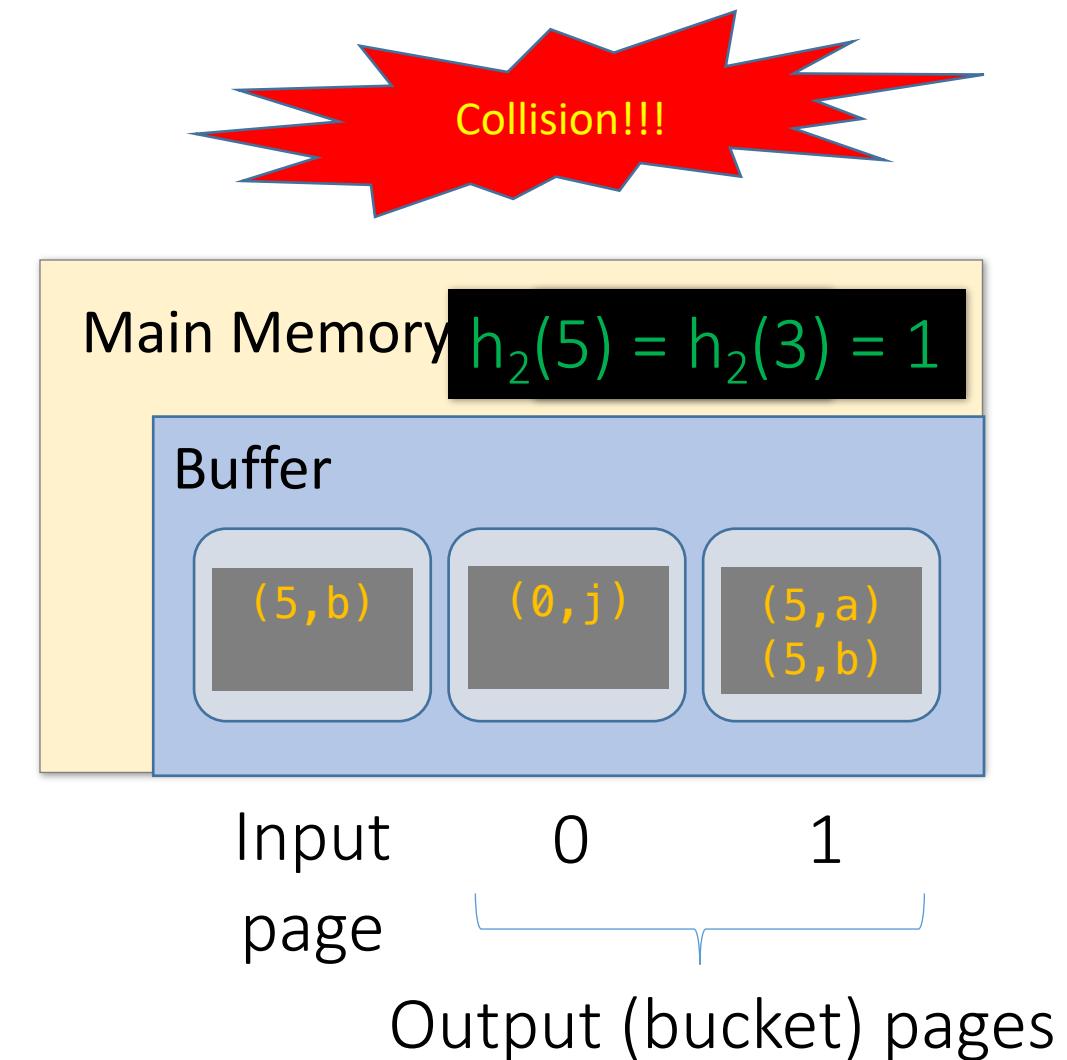
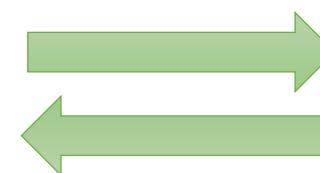
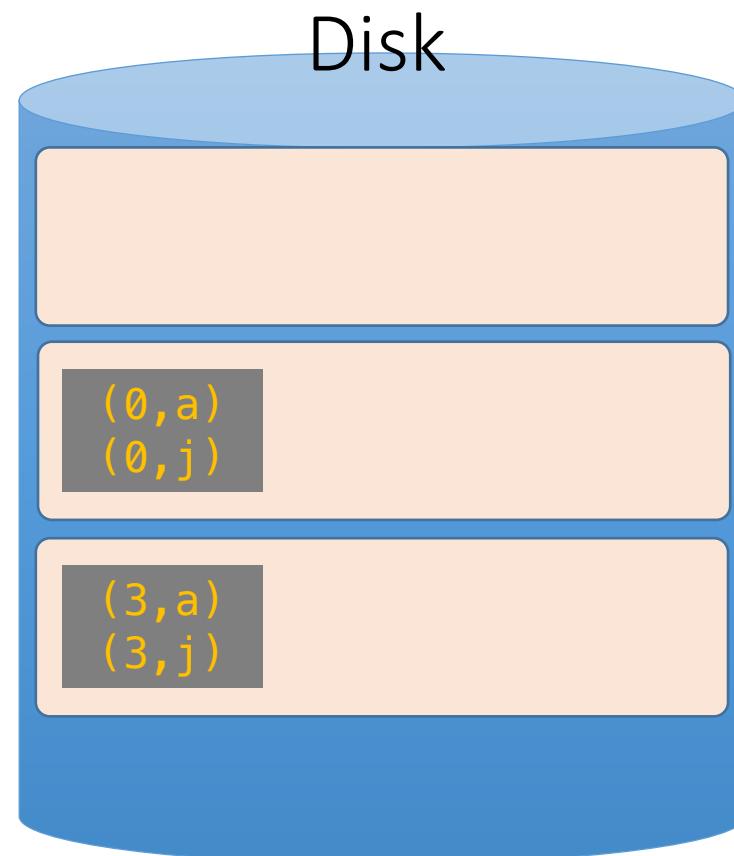
Finish this pass...



Hash Join Phase 1: Partitioning

Given $B+1 = 3$ buffer pages

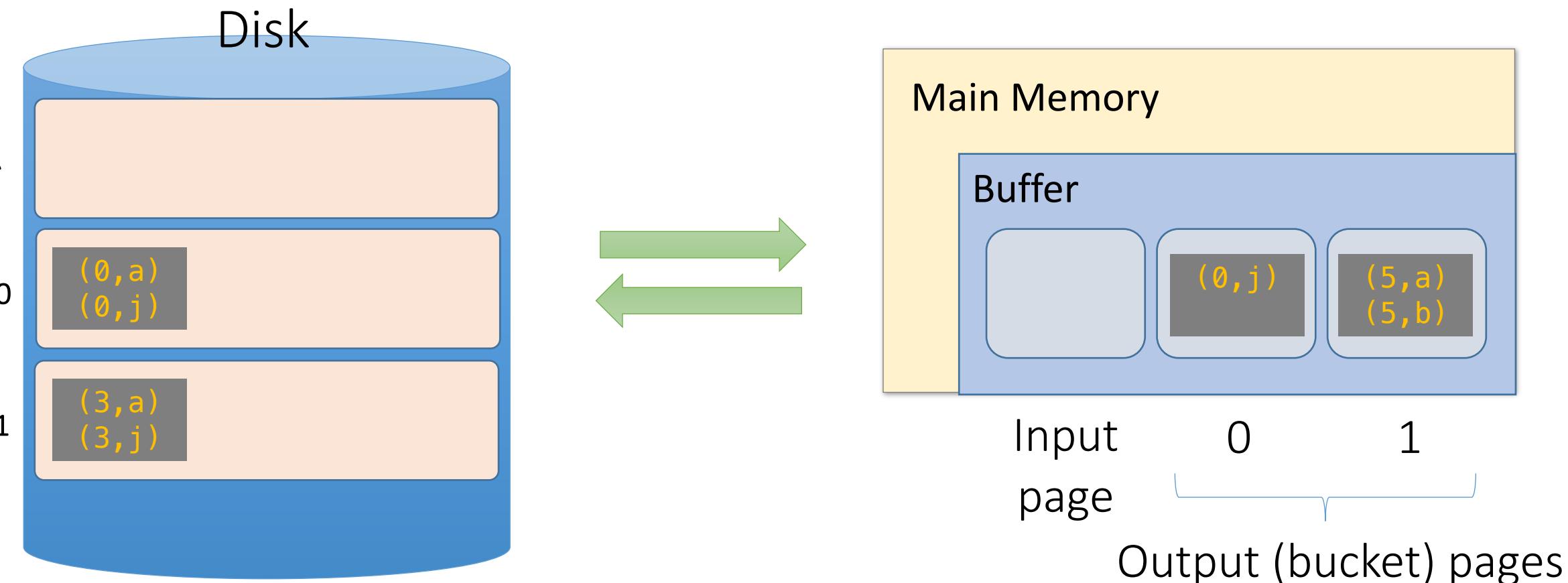
Finish this pass...



Hash Join Phase 1: Partitioning

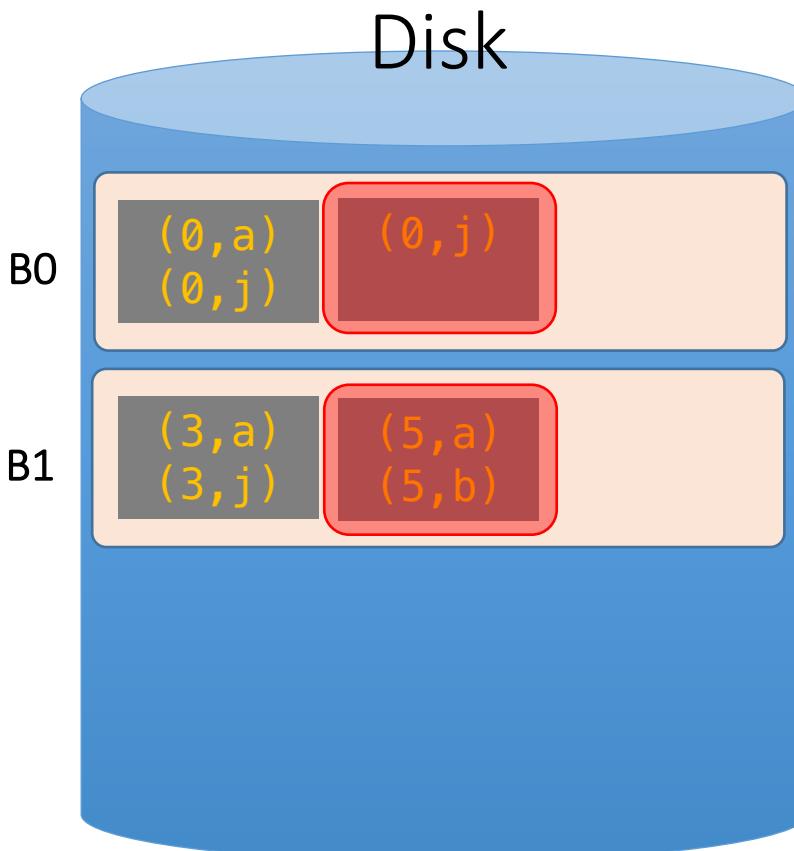
Given $B+1 = 3$ buffer pages

Finish this pass...



Hash Join Phase 1: Partitioning

Given $B+1 = 3$ buffer pages



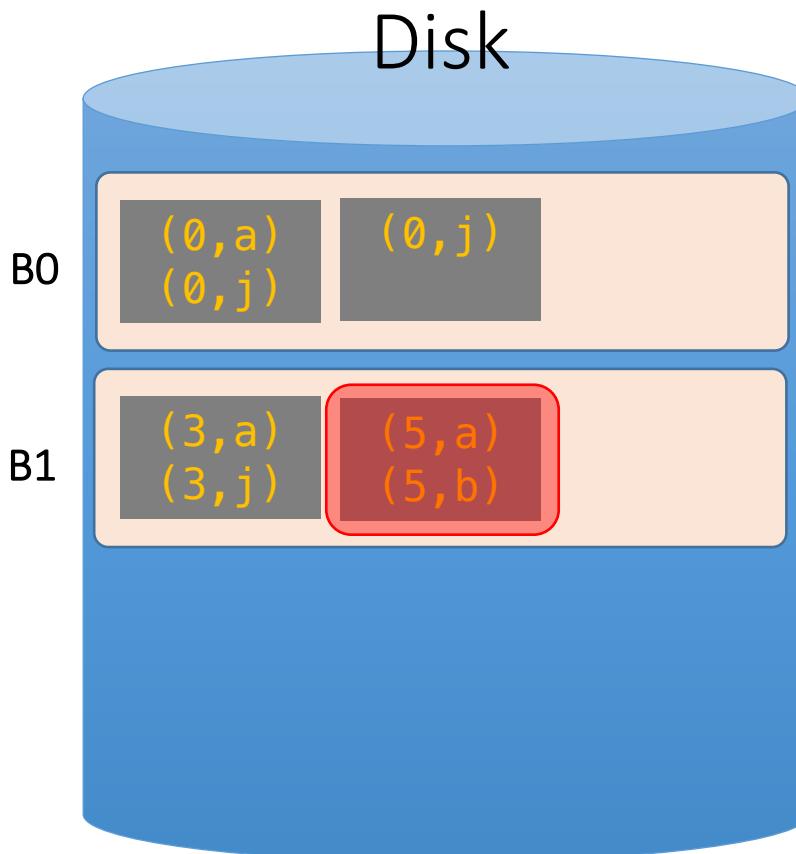
We wanted buckets of size $B-1 = 1$...
however we got larger ones due to:

(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!

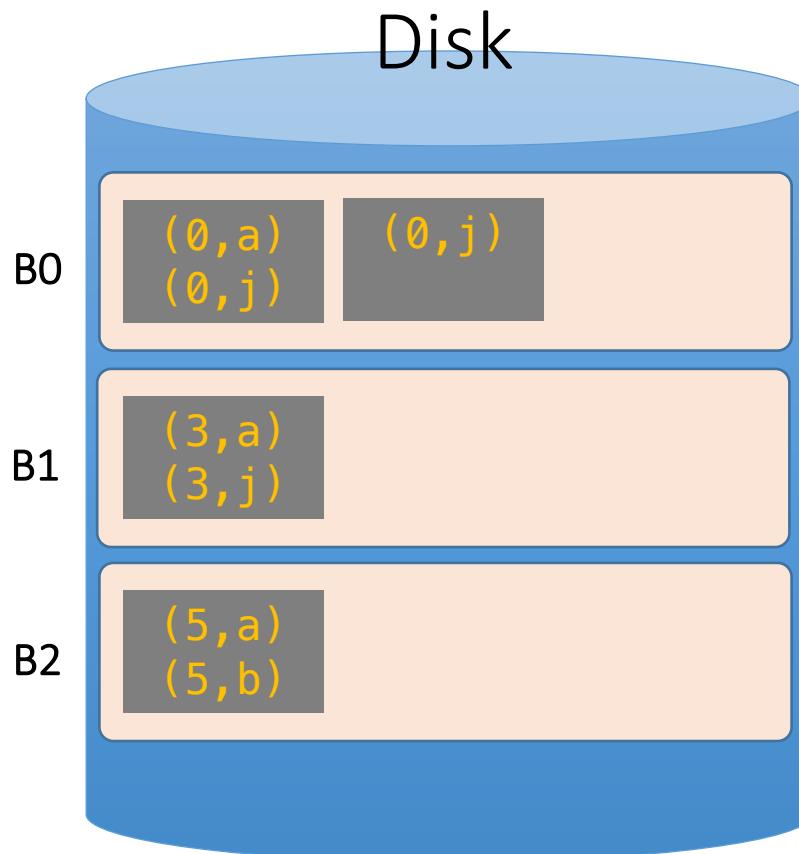
What hash function should we use?

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

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

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!

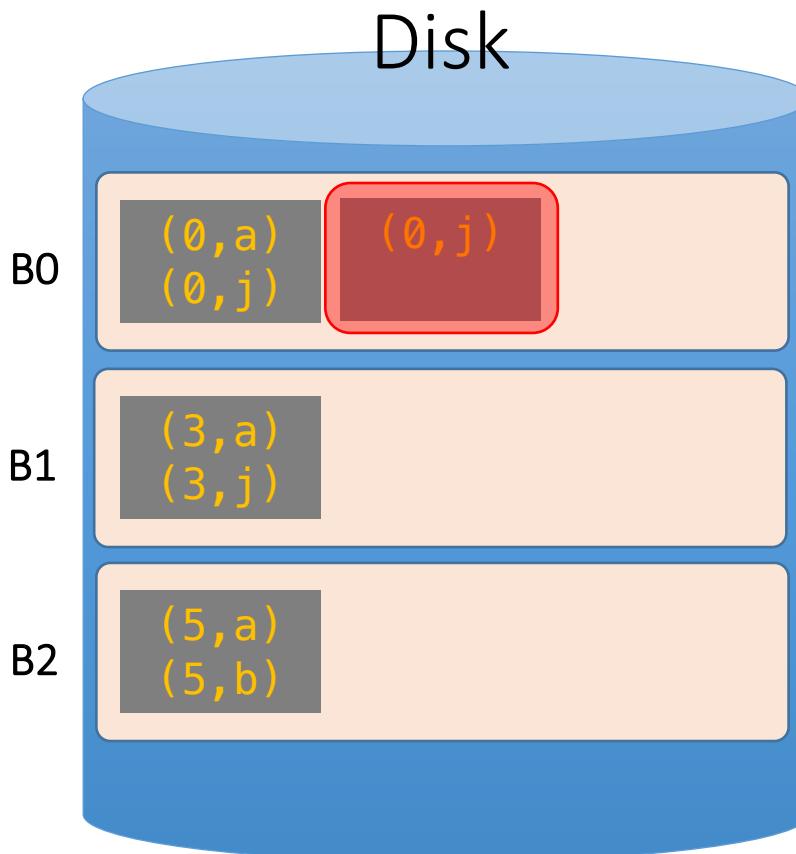
What hash function should we use?

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



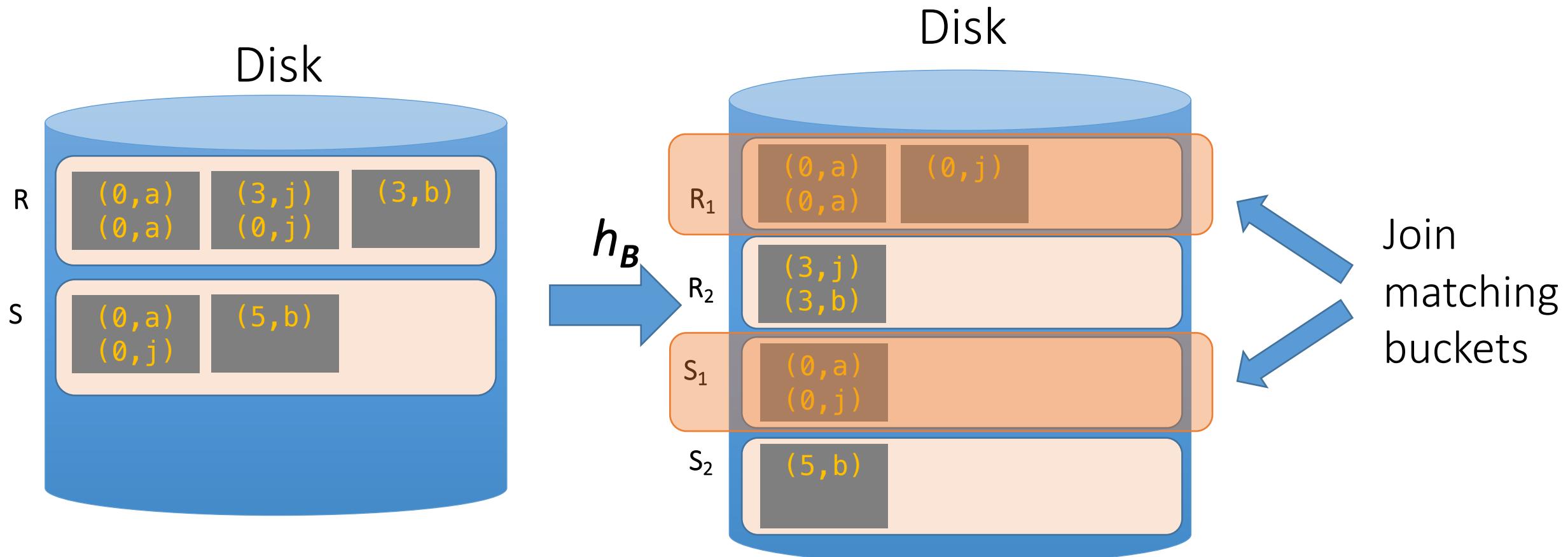
What about duplicate join keys?
Unfortunately this is a problem... but
usually not a huge one.

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!



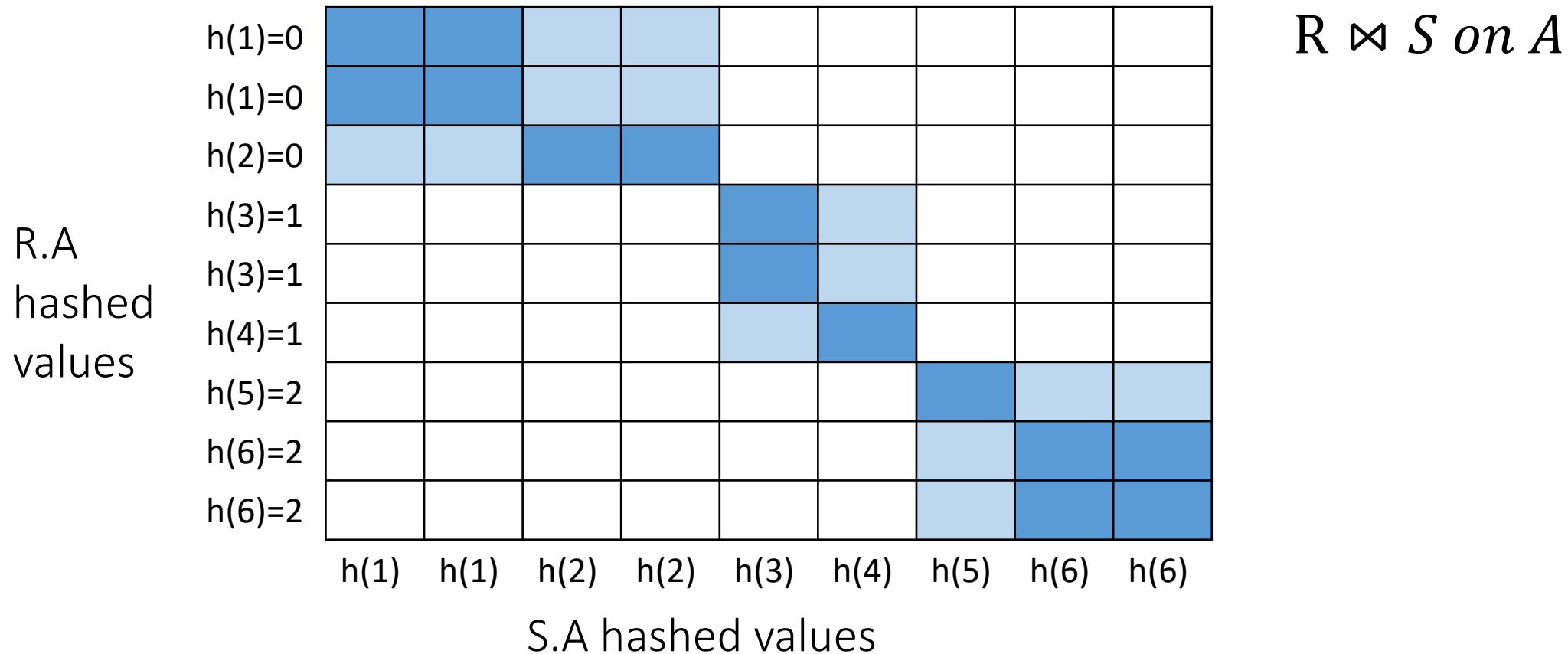
Hash Join Phase 2: Matching

- Note that since $x = y \rightarrow h(x) = h(y)$, 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 $\sim B - 1$ pages, can join each such pair using BNLJ ***in linear time***; recall (with $P(R) = B-1$):

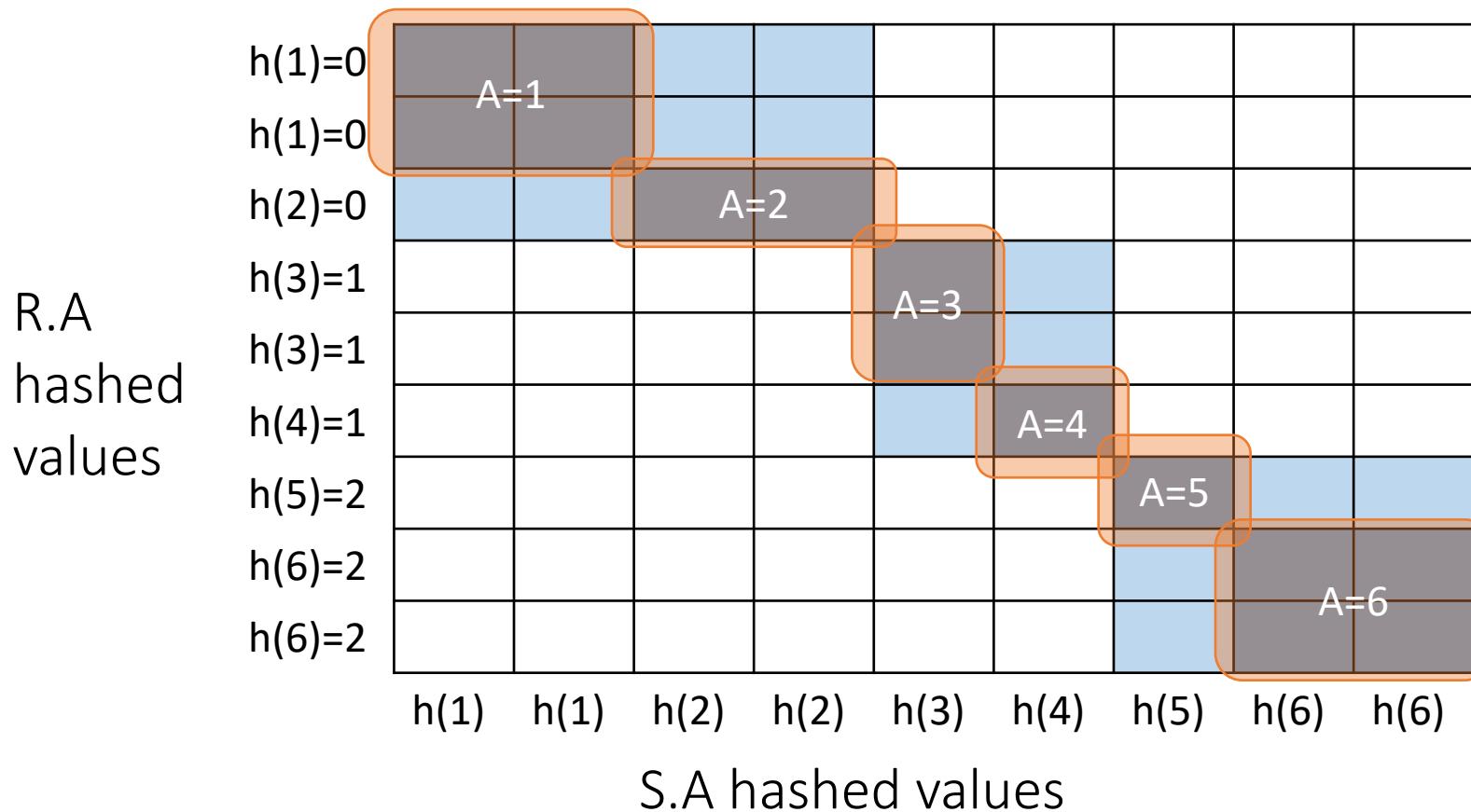
$$\text{BNLJ Cost: } P(R) + \frac{P(R)P(S)}{B-1} = P(R) + \frac{(B-1)P(S)}{B-1} = P(R) + P(S)$$

Joining the pairs of buckets is linear!
(As long as smaller bucket $\leq B-1$ pages)

Hash Join Phase 2: Matching



Hash Join Phase 2: Matching

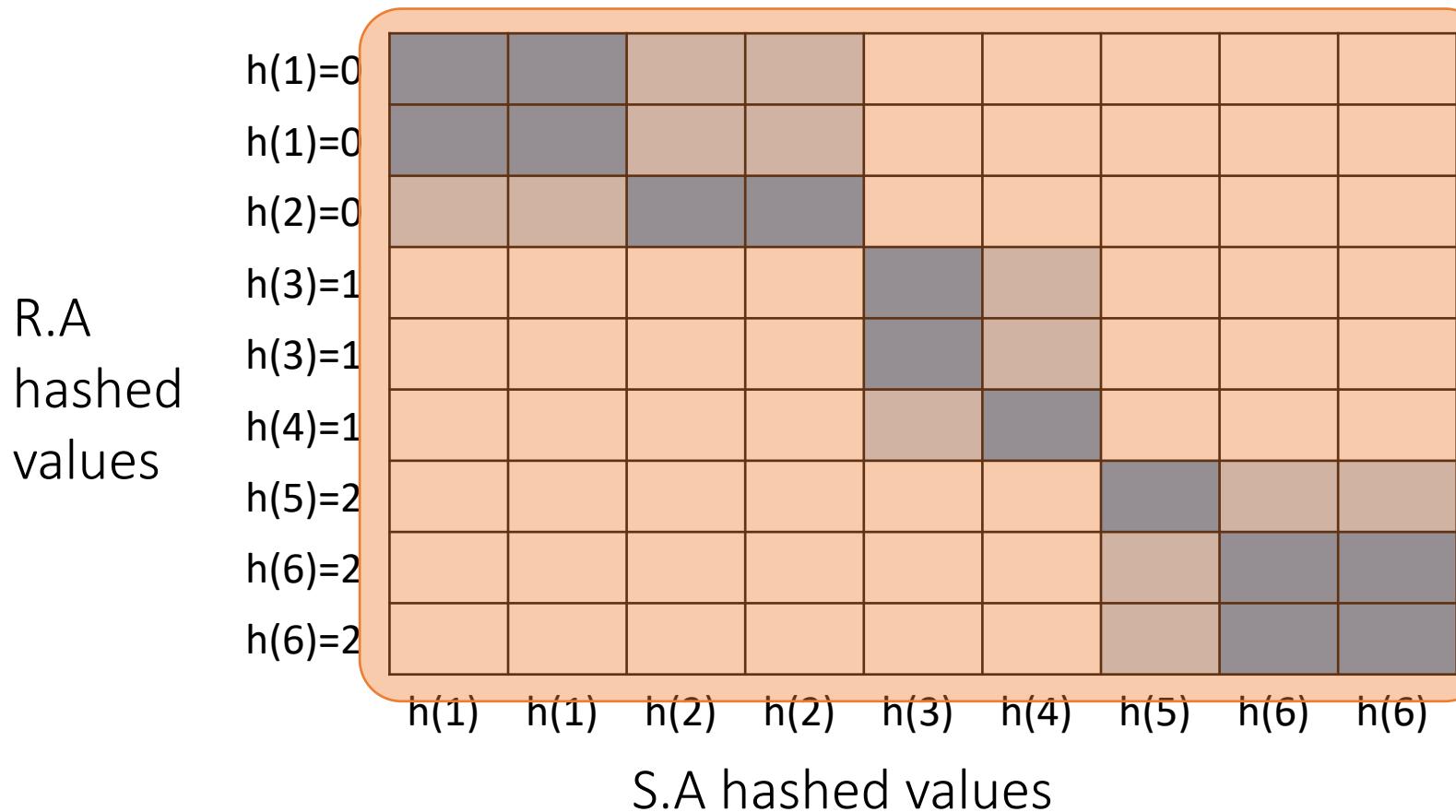


$R \bowtie S \text{ on } A$

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

= the tuples with same values of the join key A

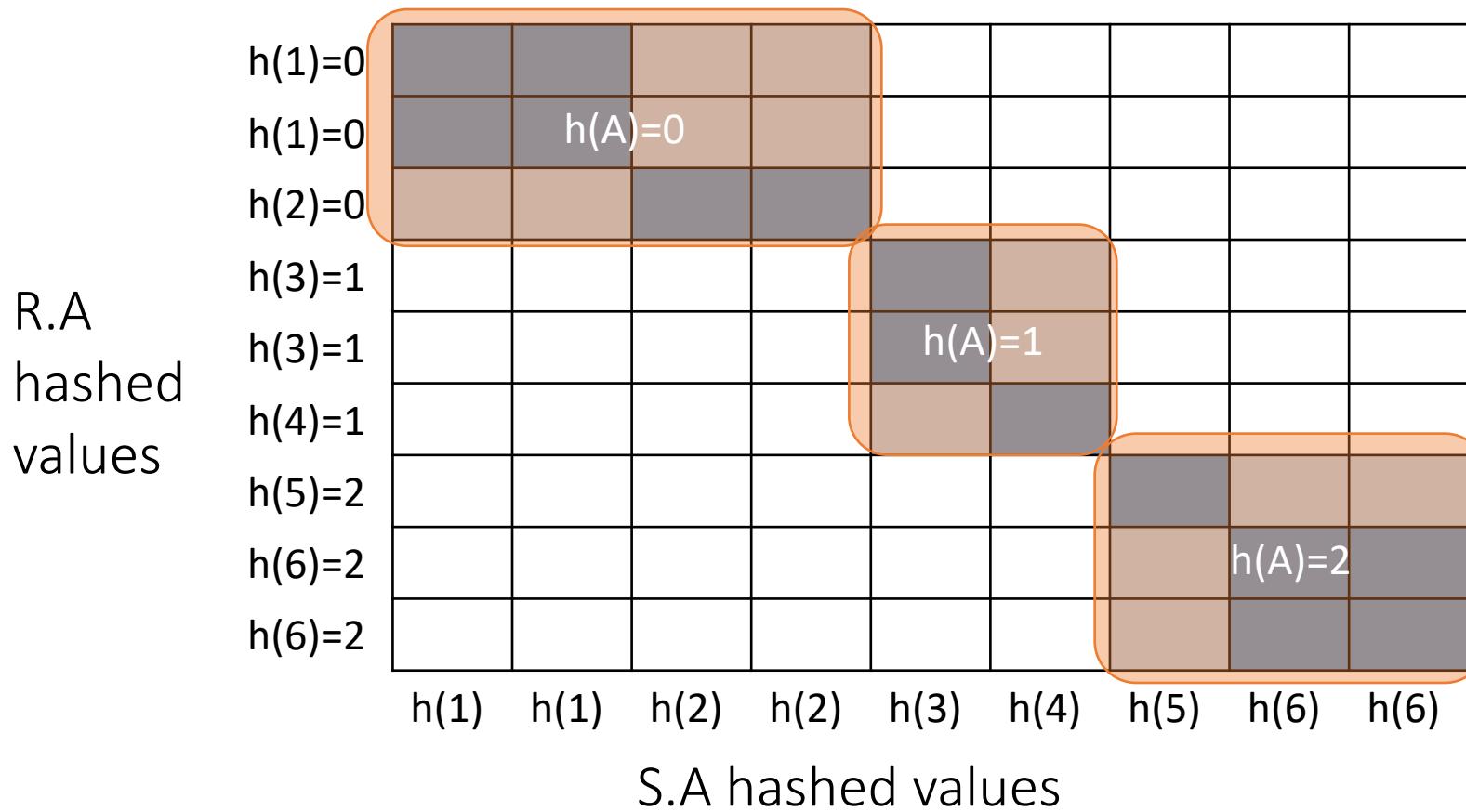
Hash Join Phase 2: Matching



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

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



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

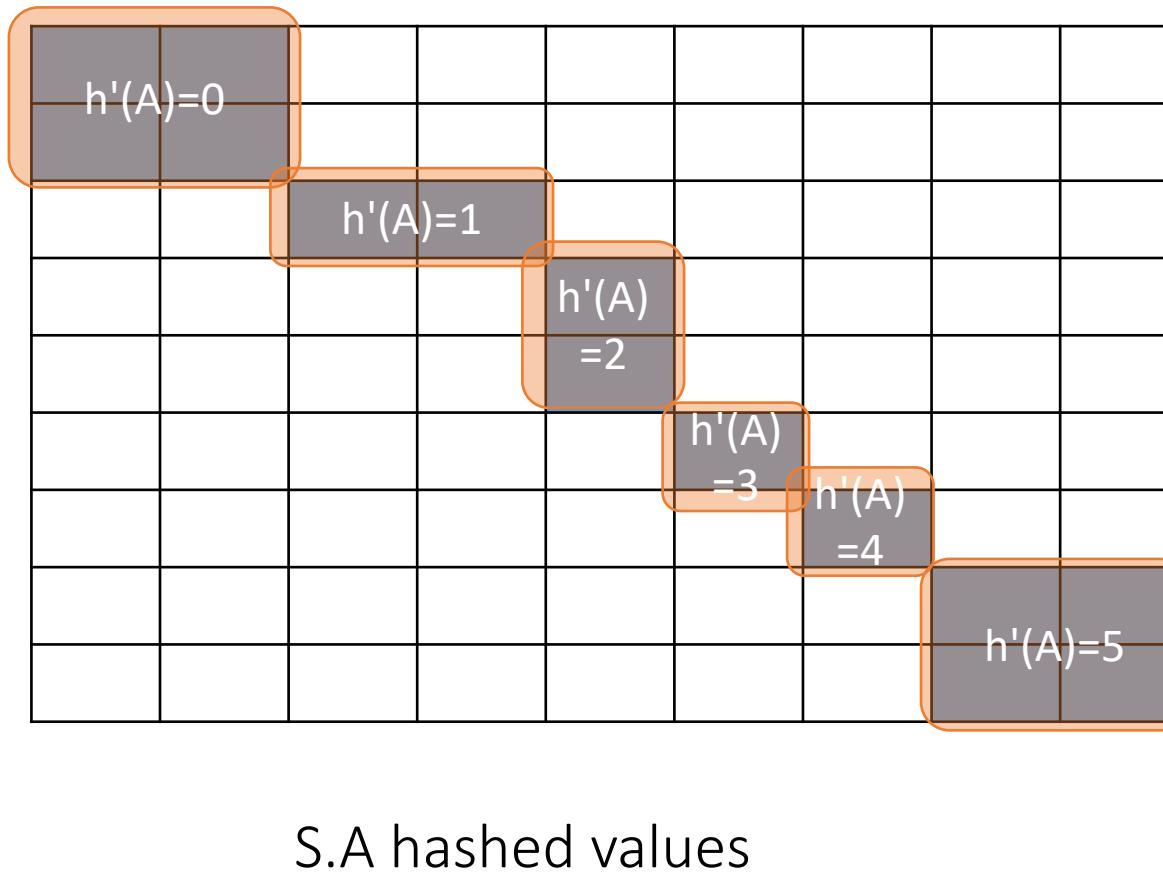
With HJ, we only explore the *blue* regions

= 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



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

An alternative to applying BNLJ:

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

How much memory do we need for HJ?

- Given $B+1$ buffer pages + WLOG: Assume $P(R) \leq P(S)$
- Suppose (reasonably) that we can partition into B buckets in 2 passes:
 - For R , we get B buckets of size $\sim P(R)/B$
 - To join these buckets in linear time, we need these buckets to fit in $B-1$ pages, so we have:

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

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

Hash Join Summary

- *Given enough buffer pages as on previous slide...*
 - **Partitioning** requires reading + writing each page of R,S
 - $\rightarrow 2(P(R)+P(S))$ IOs
 - **Matching** (with BNLJ) requires reading each page of R,S
 - $\rightarrow P(R) + P(S)$ IOs
 - **Writing out results** could be as bad as $P(R)*P(S)$... but probably closer to $P(R)+P(S)$

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

SMJ vs. HJ

Sort-Merge v. Hash Join

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

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

- ***“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

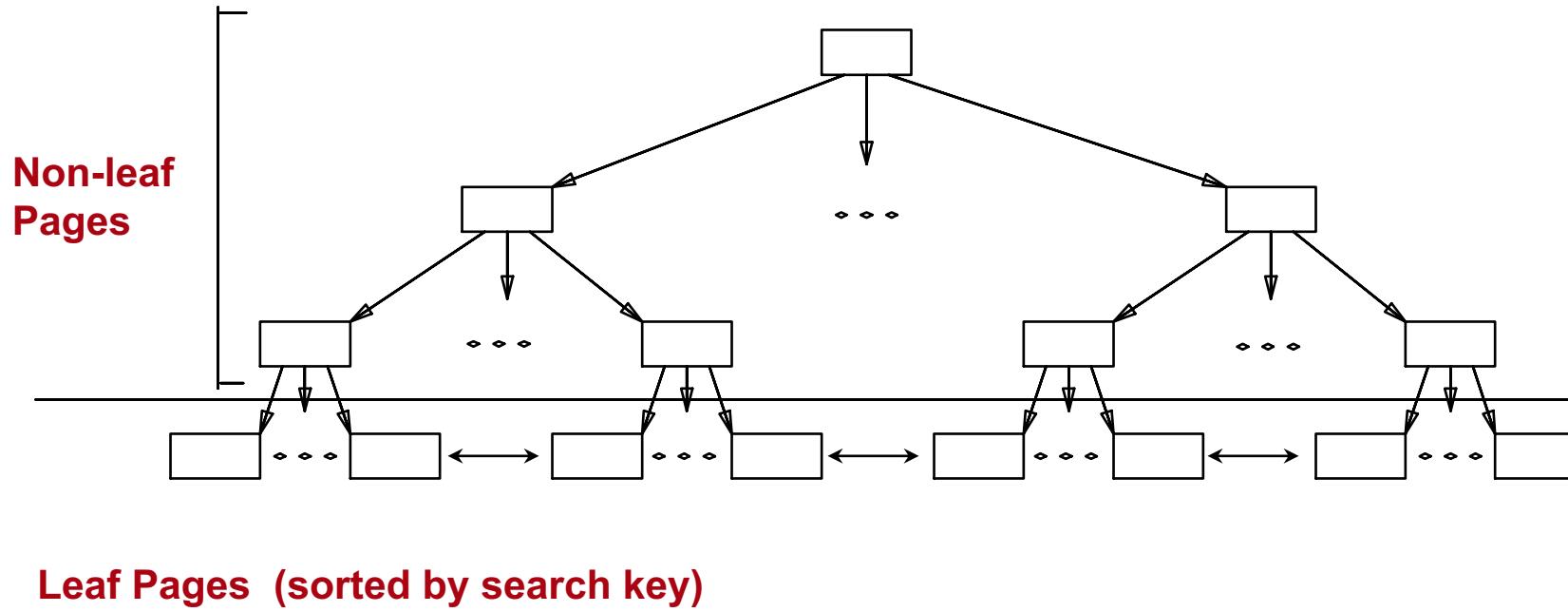
- Saw IO-aware join algorithms
 - Massive difference
- Memory sizes key in hash versus sort join
 - Hash Join = Little dog (depends on smaller relation)
- Skew is also a major factor

B+ Tree

B+ Trees

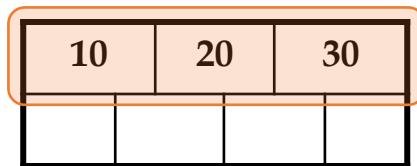
- Search trees
 - B does not mean binary!
- Idea in B Trees:
 - make 1 node = 1 physical page
 - Balanced, height adjusted tree (not the B either)
- Idea in B+ Trees:
 - Make leaves into a linked list (for range queries)

B+ Tree Index



- Leaf pages contain data entries, and are chained (prev & next)
- Non-leaf pages have data entries

B+ Tree Basics

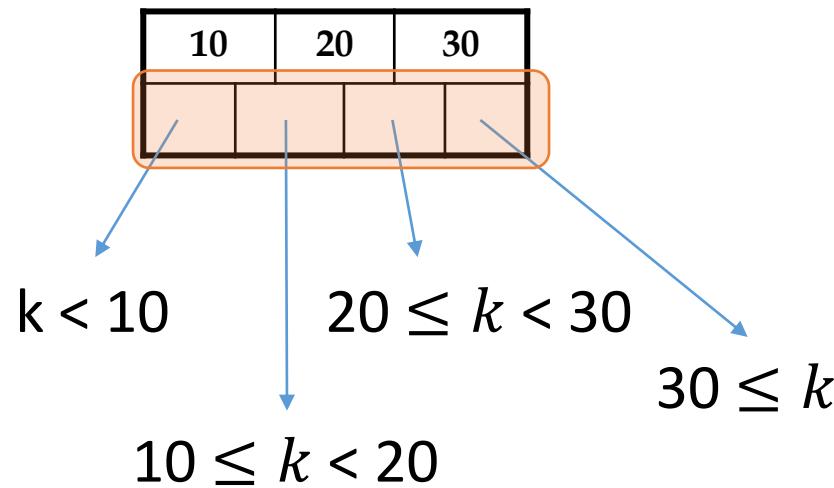


Parameter d = the order

Each *non-leaf (“interior”)* *node*
has $d \leq m \leq 2d$ *entries*
• *Minimum 50% occupancy*

Root *node* has $1 \leq m \leq 2d$
entries

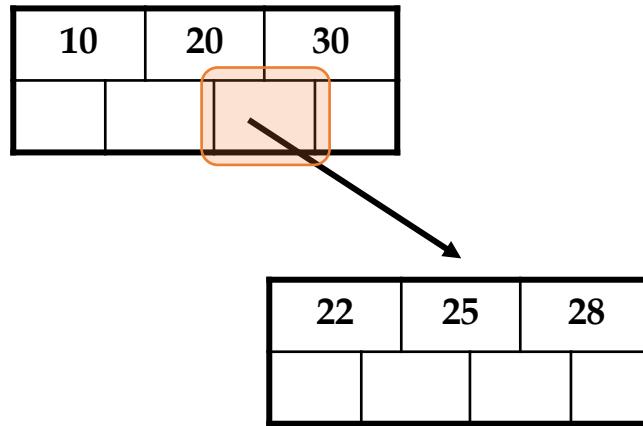
B+ Tree Basics



The n entries in a node define $n+1$ ranges

B+ Tree Basics

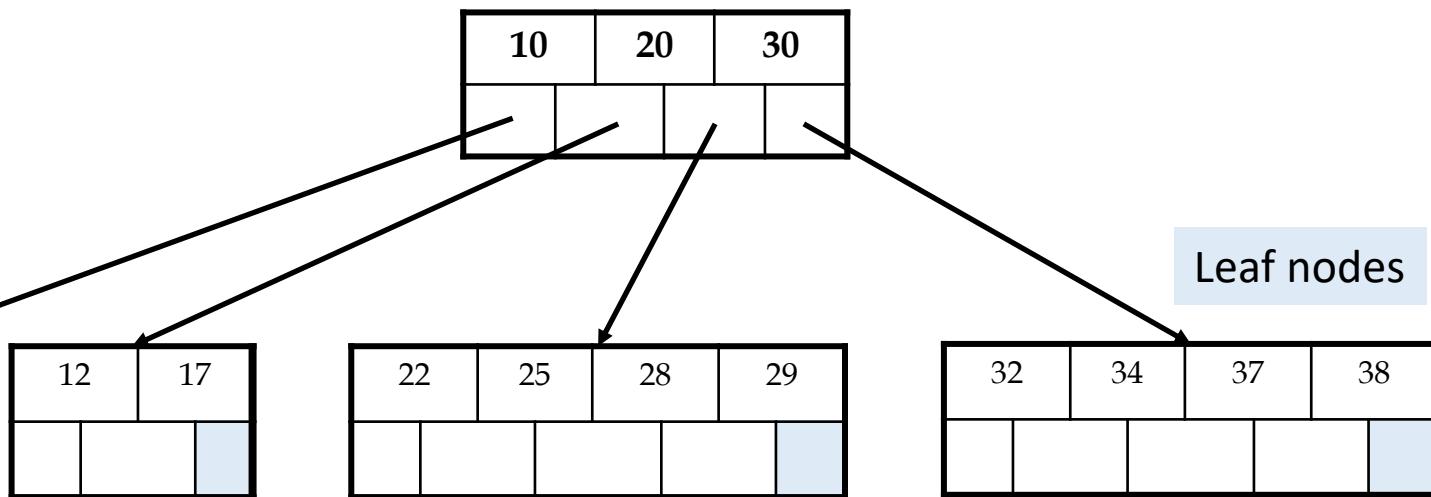
Non-leaf or *internal* node



For each range, in a *non-leaf* node, there is a **pointer** to another node with entries in that range

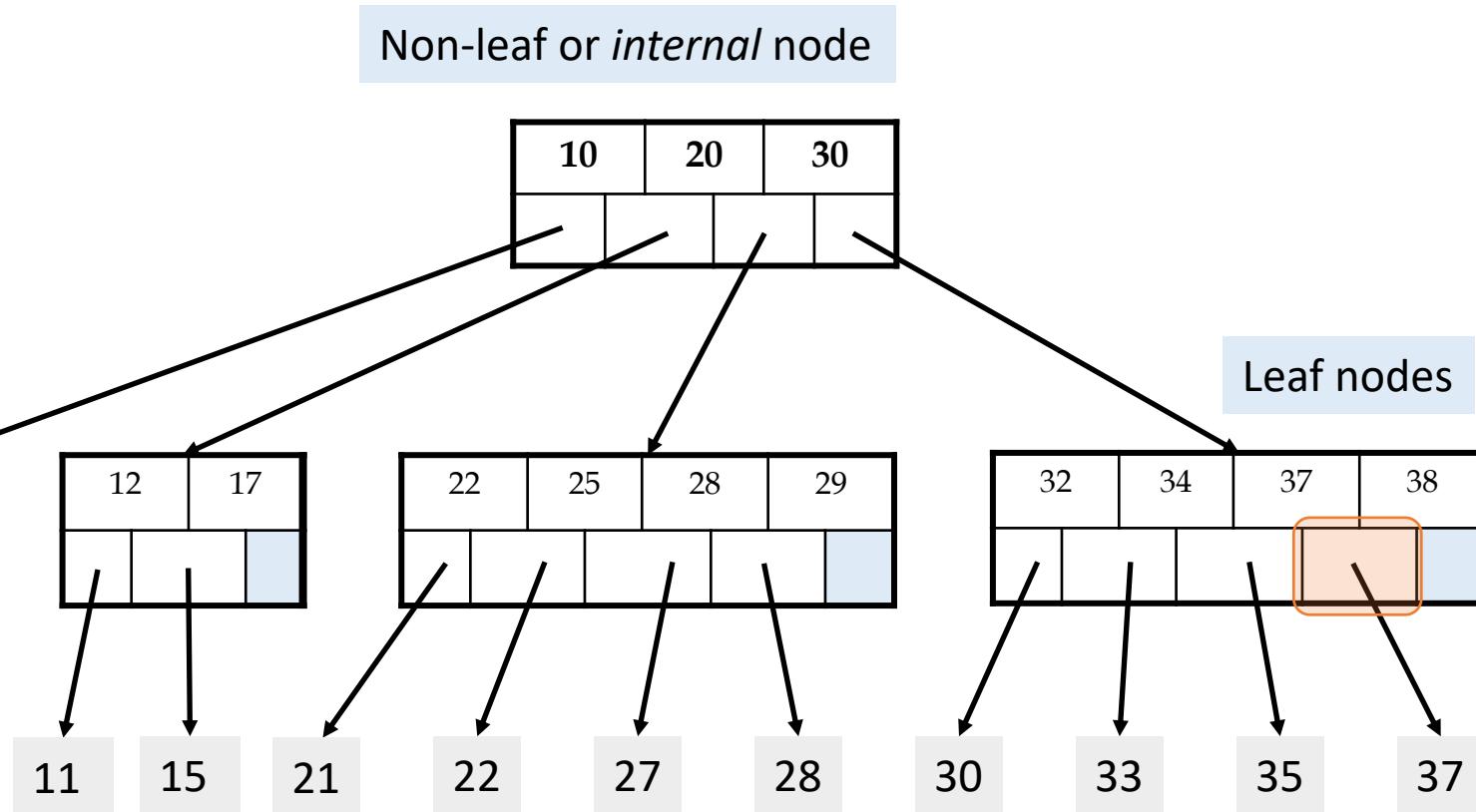
B+ Tree Basics

Non-leaf or *internal* node



Leaf nodes also have between d and $2d$ entries, and are different in that:

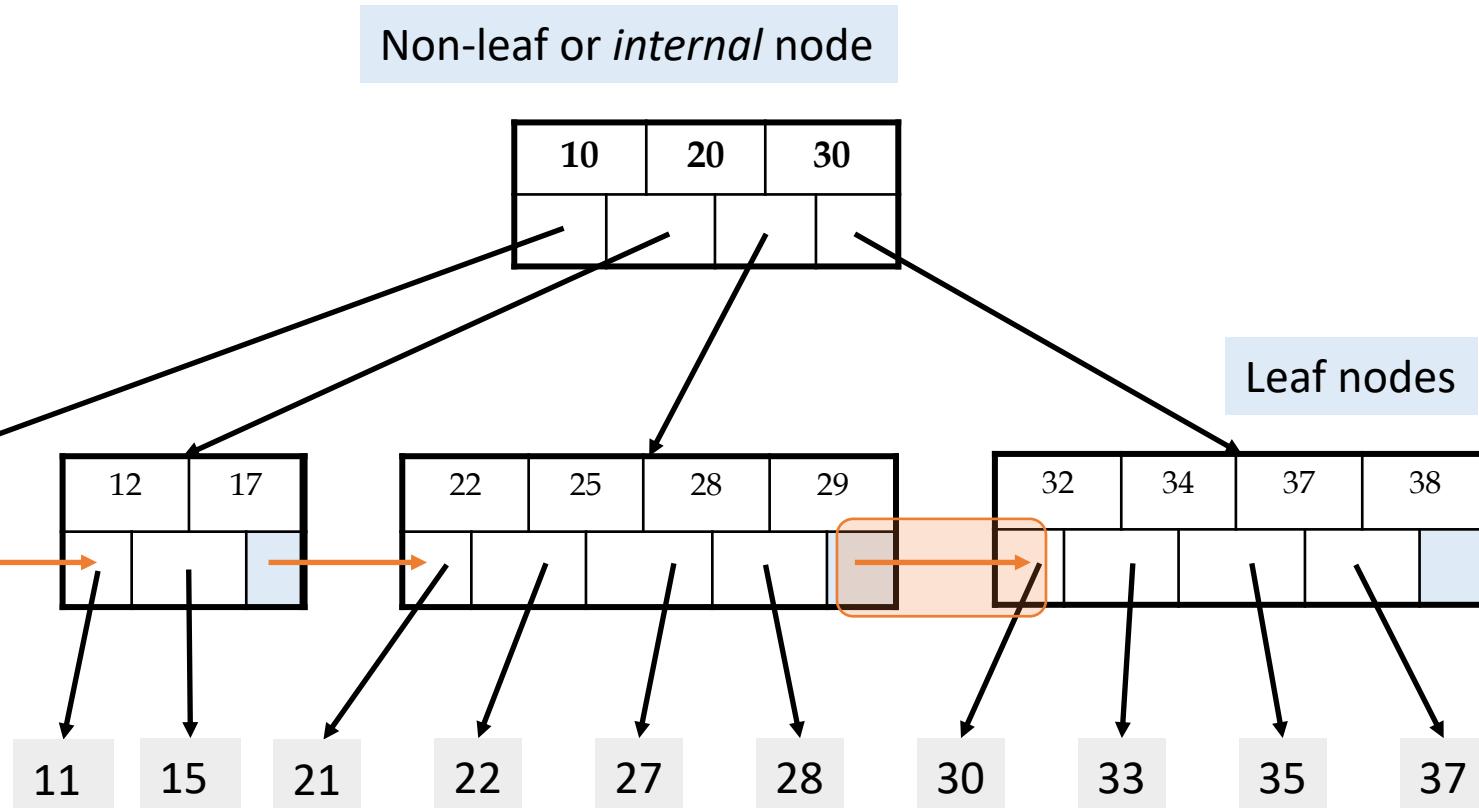
B+ Tree Basics



Leaf nodes also have between d and $2d$ entries, and are different in that:

Their entry slots contain pointers to data records

B+ Tree Basics

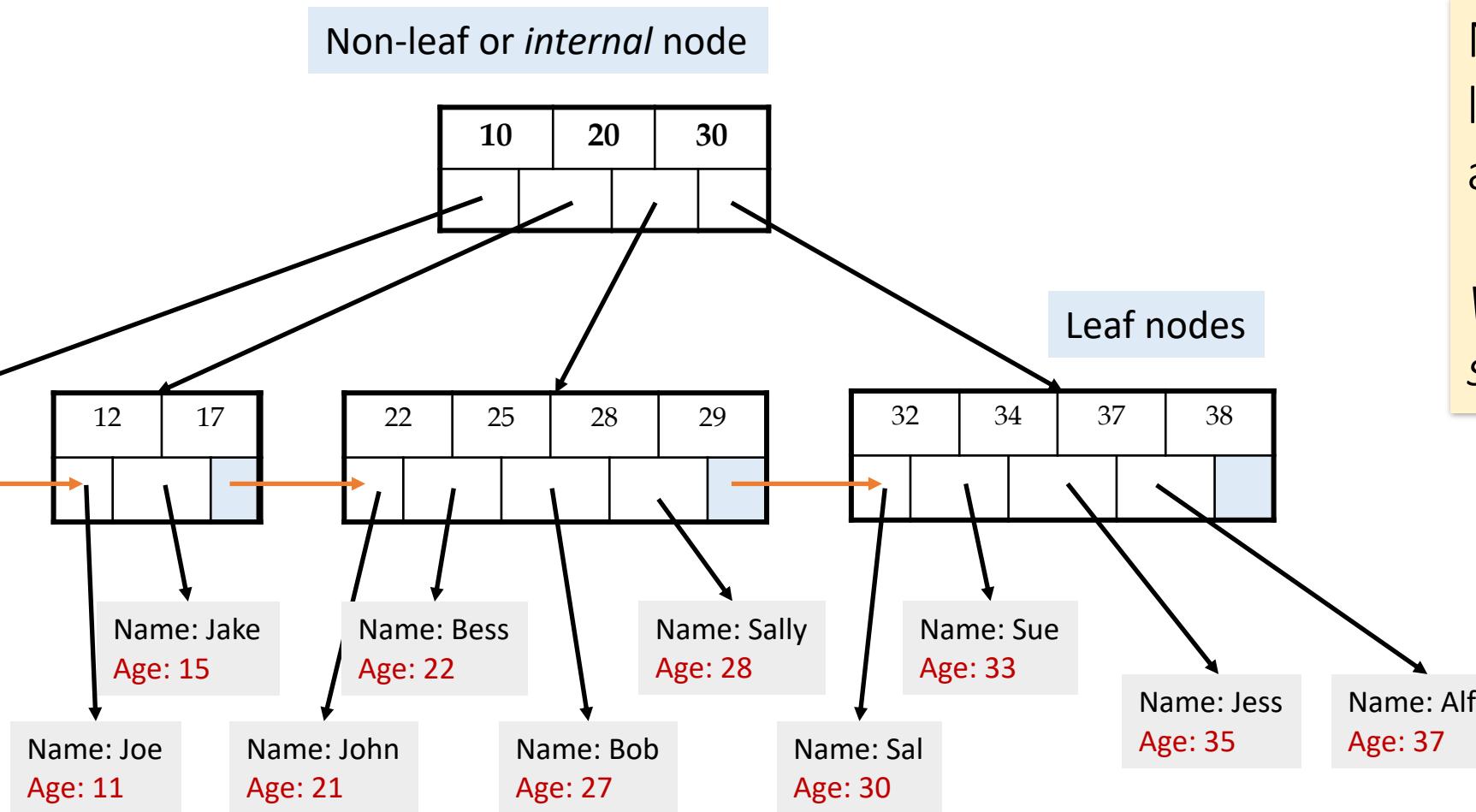


Leaf nodes also have between d and $2d$ entries, and are different in that:

Their entry slots contain pointers to data records

They contain a pointer to the next leaf node as well, *for faster sequential traversal*

B+ Tree Basics



Note that the pointers at the leaf level will be to the actual data records (rows).

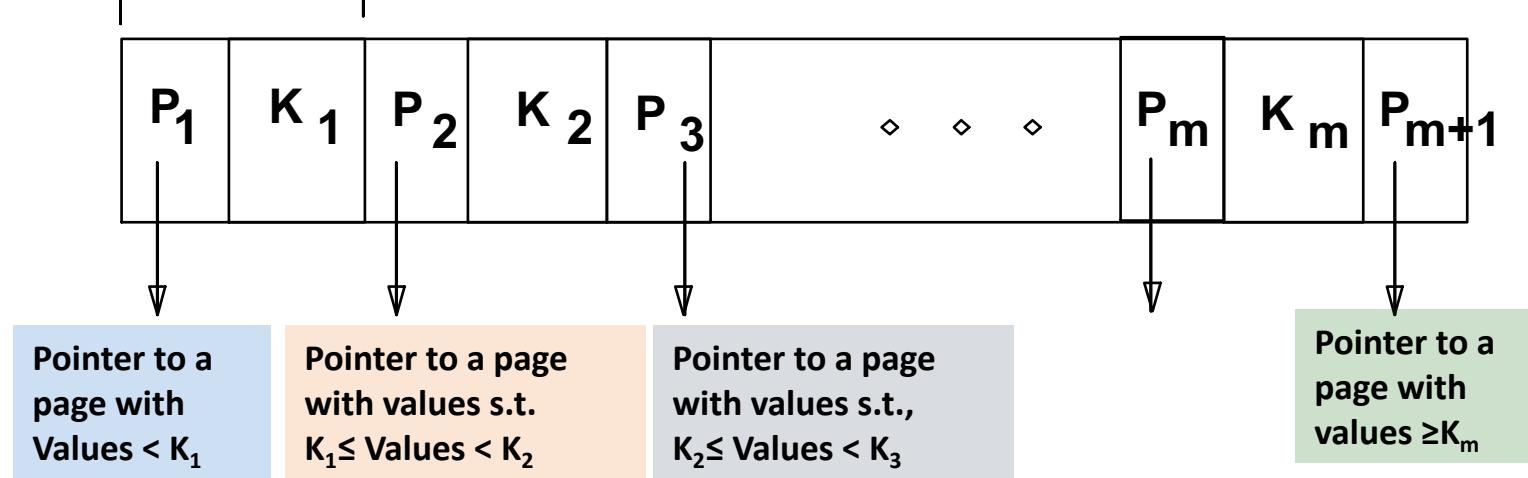
We might truncate these for simpler display (as before)...

Height = 1

B+ Tree Page Format

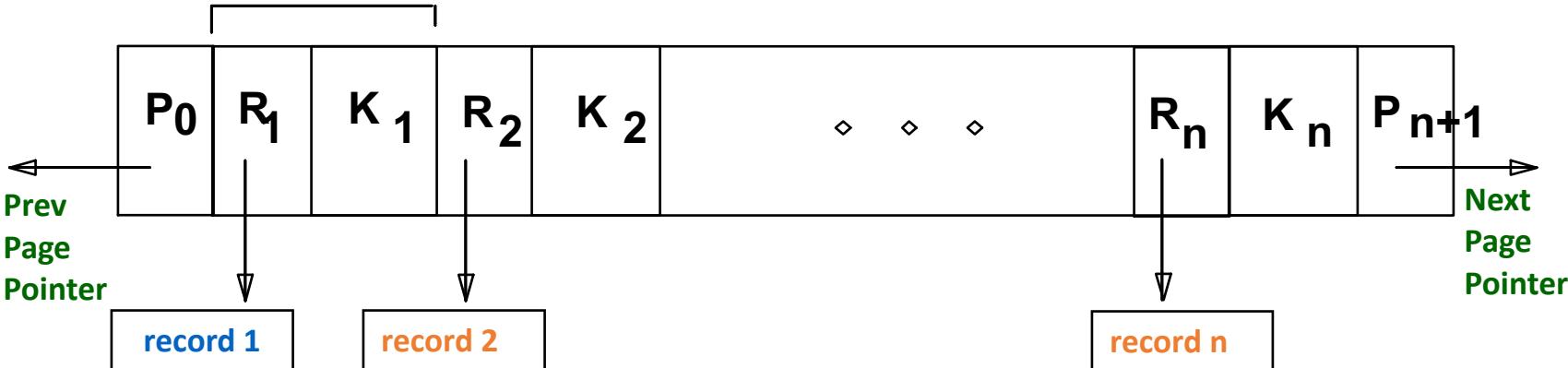
Non-leaf
Page

index entries



Leaf Page

data entries



B+ Tree operations

A B+ tree supports the following operations:

- equality search
- range search
- insert
- delete
- bulk loading

Searching a B+ Tree

- For exact key values:
 - Start at the root
 - Proceed down, to the leaf
- For range queries:
 - As above
 - *Then sequential traversal*

```
SELECT name  
FROM people  
WHERE age = 25
```

```
SELECT name  
FROM people  
WHERE 20 <= age  
      AND age <= 30
```

B+ Tree: Search

- start from root
- examine index entries in non-leaf nodes to find the correct child
- traverse down the tree until a leaf node is reached
- non-leaf nodes can be searched using a binary or a linear search

B+ Tree Exact Search Animation

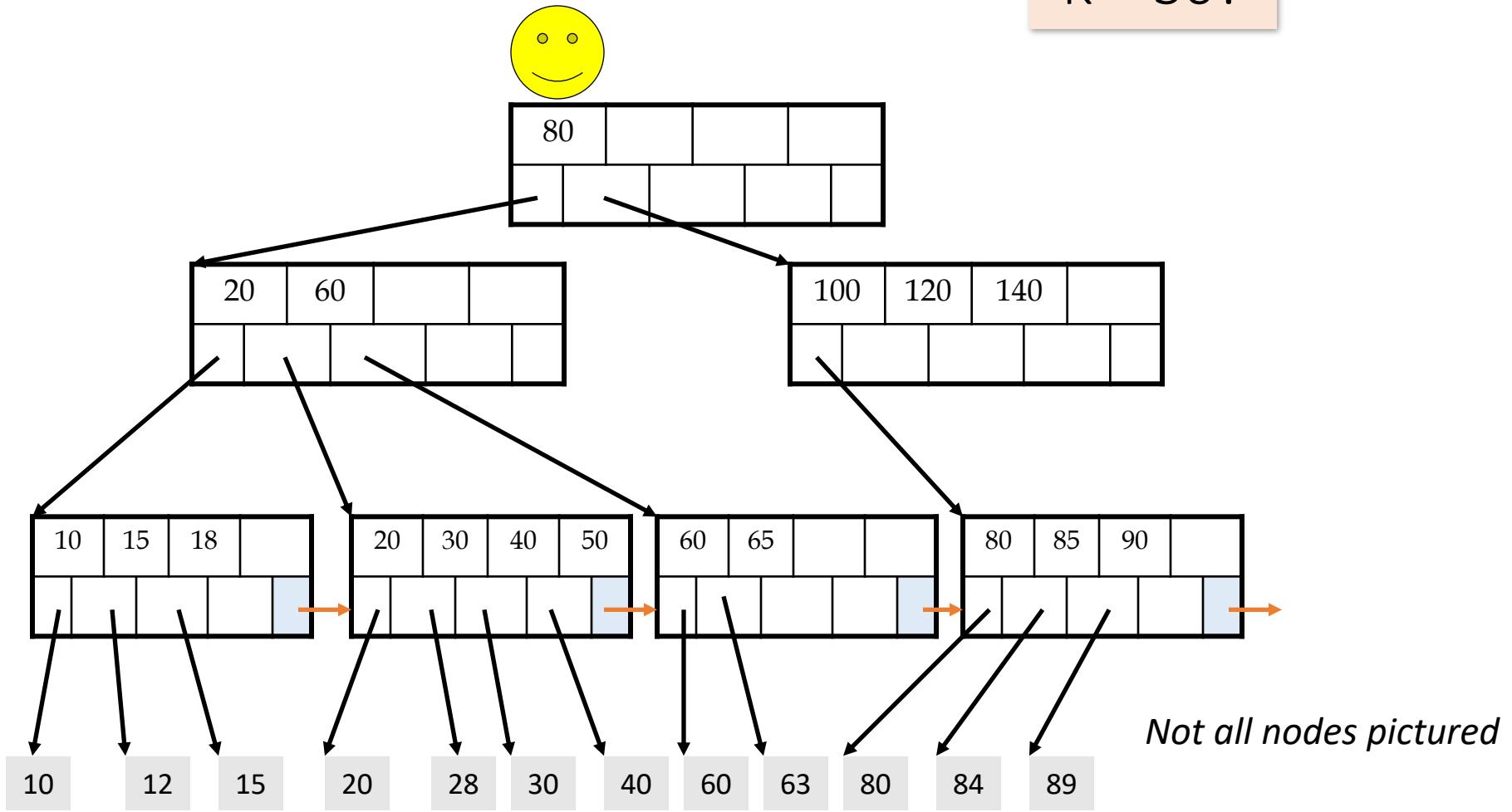
K = 30?

30 < 80

30 in [20,60)

30 in [30,40)

To the data!



B+ Tree Range Search Animation

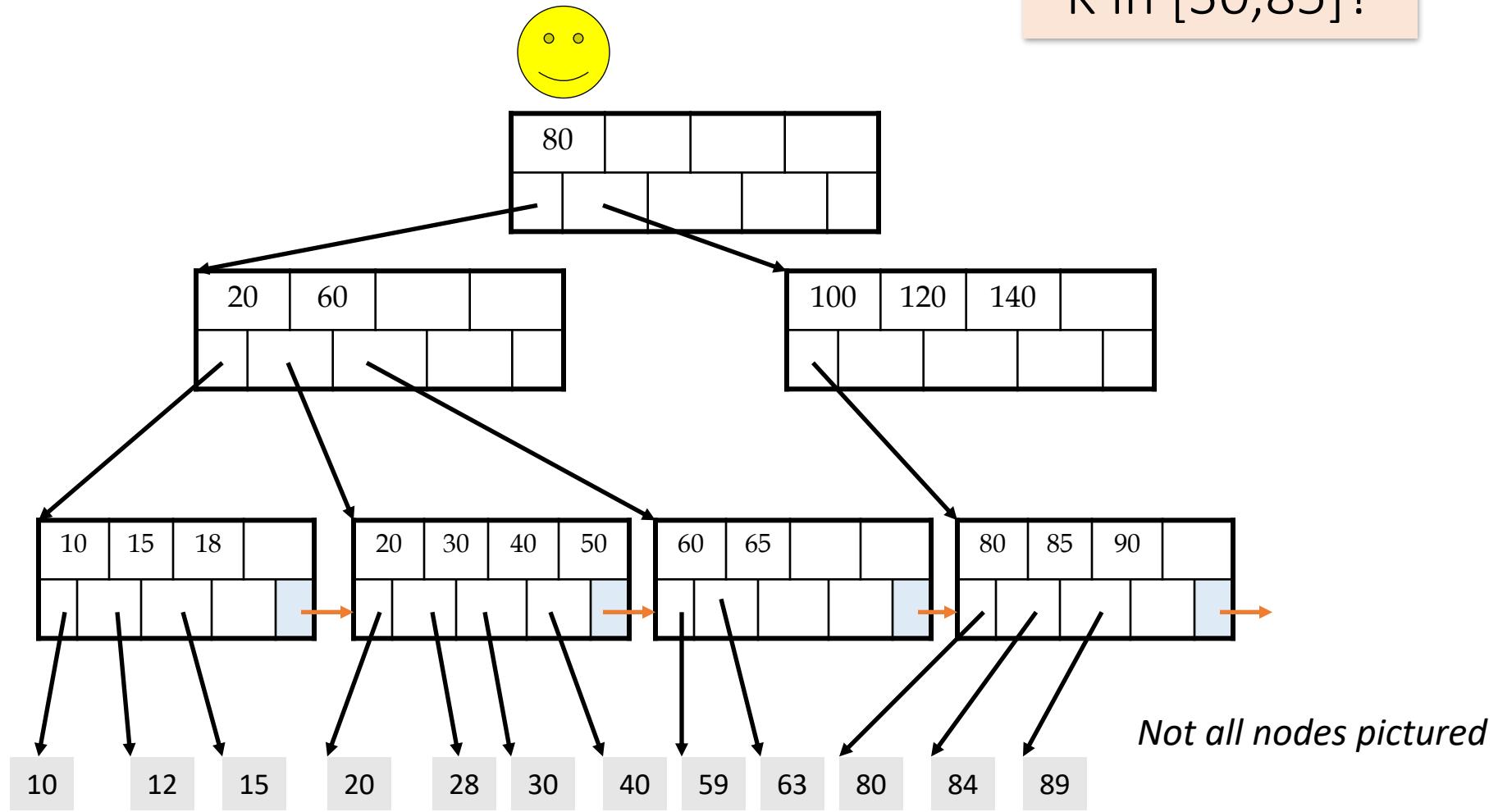
K in [30,85]?

30 < 80

30 in [20,60)

30 in [30,40)

To the data!



B+ Tree Design

- How large is d ?
- Example:
 - Key size = 4 bytes
 - Pointer size = 8 bytes
 - Block size = 4096 bytes
- We want each *node* to fit on a single *block/page*
 - $2d \times 4 + (2d+1) \times 8 \leq 4096 \rightarrow d \leq 170$

NB: Oracle allows 64K =
 2^{16} byte blocks
 $\rightarrow d \leq 2730$

B+ Tree: High Fanout = Smaller & Lower IO

- As compared to e.g. binary search trees, B+ Trees have **high fanout (between $d+1$ and $2d+1$)**
- This means that the **depth of the tree is small** → getting to any element requires very few IO operations!
 - Also can often store most or all of the B+ Tree in main memory!
- A TiB = 2^{40} Bytes. What is the height of a B+ Tree (with fill-factor = 1) that indexes it (with 64K pages)?
 - $(2 * 2^{730} + 1)^h = 2^{40} \rightarrow h = 4$

The fanout is defined as the number of pointers to child nodes coming out of a node

Note that fanout is dynamic - we'll often assume it's constant just to come up with approximate eqns!

The known universe contains $\sim 10^{80}$ particles... what is the height of a B+ Tree that indexes these?

B+ Trees in Practice

- Typical order: $d=100$. Typical fill-factor: 67%.
 - average fanout = 133
- Typical capacities:
 - Height 4: $133^4 = 312,900,700$ records
 - Height 3: $133^3 = 2,352,637$ records
- Top levels of tree sit *in the buffer pool*:
 - Level 1 = 1 page = 8 Kbytes
 - Level 2 = 133 pages = 1 Mbyte
 - Level 3 = 17,689 pages = 133 MBytes

Fill-factor is the percent of available slots in the B+ Tree that are filled; is usually < 1 to leave slack for (quicker) insertions

Typically, only pay for one IO!

Simple Cost Model for Search

- Let:
 - f = fanout, which is in $[d+1, 2d+1]$ (*we'll assume it's constant for our cost model...*)
 - N = the total number of *pages* we need to index
 - F = fill-factor (usually $\approx 2/3$)
- Our B+ Tree needs to have room to index N/F pages!
 - We have the fill factor in order to leave some open slots for faster insertions
- What height (h) does our B+ Tree need to be?
 - $h=1 \rightarrow$ Just the root node- room to index f pages
 - $h=2 \rightarrow f$ leaf nodes- room to index f^2 pages
 - $h=3 \rightarrow f^2$ leaf nodes- room to index f^3 pages
 - ...
 - $h \rightarrow f^{h-1}$ leaf nodes- room to index f^h pages!

→ We need a B+ Tree
of height $h = \lceil \log_f \frac{N}{F} \rceil$!

Simple Cost Model for Search

- Note that if we have B available buffer pages, by the same logic:
 - We can store L_B levels of the B+ Tree in memory
 - where L_B is the number of levels such that the sum of all the levels' nodes fit in the buffer:
 - $B \geq 1 + f + \dots + f^{L_B-1} = \sum_{l=0}^{L_B-1} f^l$
- In summary: to do exact search:
 - We read in one page per level of the tree
 - However, levels that we can fit in buffer are free!
 - Finally we read in the actual record

IO Cost: $\left\lceil \log_f \frac{N}{F} \right\rceil - L_B + 1$

where $B \geq \sum_{l=0}^{L_B-1} f^l$

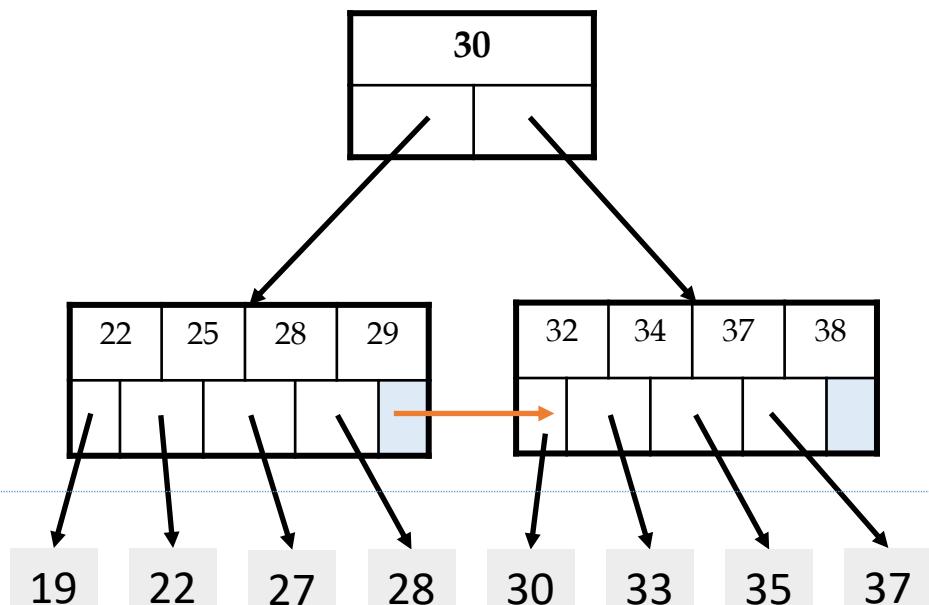
Simple Cost Model for Search

- To do range search, we just follow the horizontal pointers
- The IO cost is that of loading additional leaf nodes we need to access + the IO cost of loading each *page* of the results- we phrase this as “Cost(OUT)”

$$\text{IO Cost: } \left\lceil \log_f \frac{N}{F} \right\rceil - L_B + \text{Cost(OUT)}$$

where $B \geq \sum_{l=0}^{L_B-1} f^l$

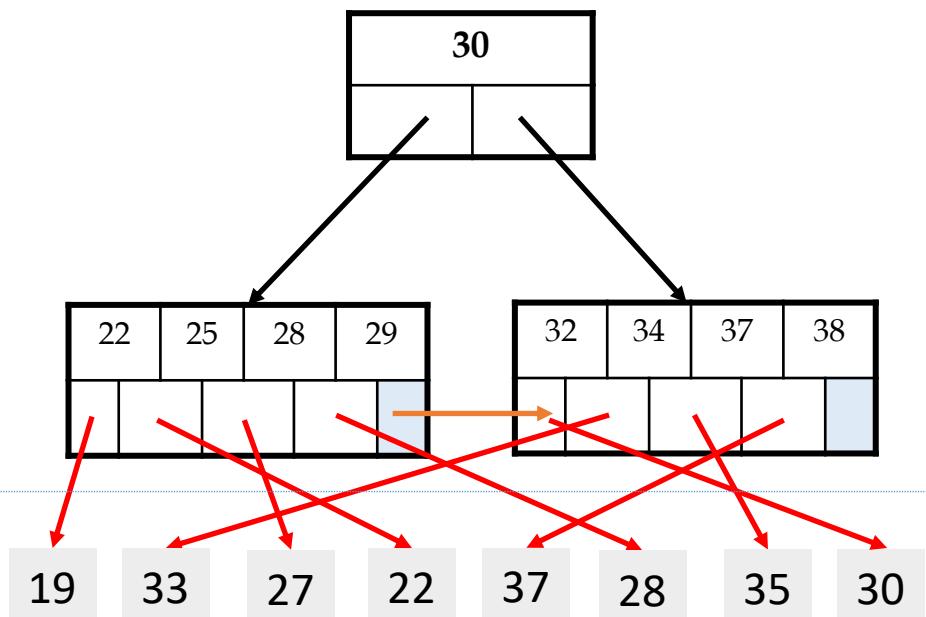
Clustered vs. Unclustered Index



Index Entries

Data Records

Clustered



Unclustered

An index is **clustered** if the underlying data is ordered in the same way as the index's data entries.

Clustered vs. Unclustered Index

- Recall that for a disk with block access, **sequential IO is much faster than random IO**
- For exact search, no difference between clustered / unclustered
- For range search over R values: difference between **1 random IO + R sequential IO, and R random IO:**
 - A random IO costs ~ 10ms (sequential much much faster)
 - For R = 100,000 records- **difference between ~10ms and ~17min!**

Hash Indexes

Hash Index

- A **hash index** is a collection of buckets
 - bucket = primary page plus overflow pages
 - buckets contain one or more data entries
- uses a hash function h
 - $h(r)$ = bucket in which (data entry for) record r belongs

Hash Index

- A **hash index** is:
 - good for equality search
 - not so good for range search (use **tree indexes** instead)
- Types of hash indexes:
 - **Static** hashing
 - **Extendible** hashing (dynamic)
 - Linear hashing (dynamic) – not covered in the course

Operations on Hash Indexes

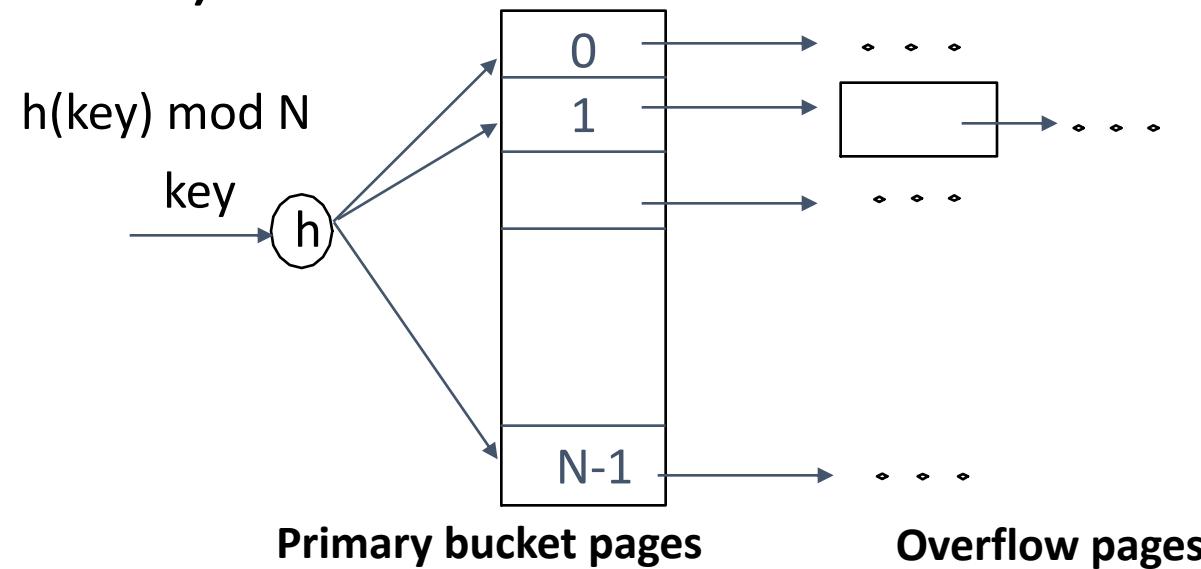
- **Equality search**
 - apply the hash function on the search key to locate the appropriate bucket
 - search through the primary page (plus overflow pages) to find the record(s)
- **Deletion**
 - find the appropriate bucket, delete the record
- **Insertion**
 - find the appropriate bucket, insert the record
 - if there is no space, create a new overflow page

Hash Functions

- An *ideal* hash function must be **uniform**: each bucket is assigned the same number of key values
- A *bad* hash function maps all search key values to the same bucket
- Examples of good hash functions:
 - $h(k) = a * k + b$, where a and b are constants
 - a random function

Static Hashing

- # primary bucket pages fixed, allocated sequentially, never de-allocated; overflow pages if needed.
- **$h(k) \bmod N$** = bucket to which data entry with key k belongs.
(N = # of buckets)



Static Hashing: Example

Person(name, zipcode, phone)

- *search key*: zipcode
- *hash function h*: last 2 digits

- 4 buckets
- each bucket has 2 data entries (full record)

primary pages

bucket 0

(John, 53400, 23218564)
(Alice, 54768, 60743111)

overflow pages

(Anna, 53632, 23209964)

bucket 1

(Theo, 53409, 23200564)

bucket 2

bucket 3

(Bob, 34411, 29010533)

Hash Functions

- An *ideal* hash function must be **uniform**: each bucket is assigned the same number of key values
- A *bad* hash function maps all search key values to the same bucket
- Examples of good hash functions:
 - $h(k) = a * k + b$, where a and b are constants
 - a random function

Bucket Overflow

- Bucket *overflow* can occur because of
 - insufficient number of buckets
 - *skew* in distribution of records
 - many records have the same search-key value
 - the hash function results in a non-uniform distribution of key values
- Bucket overflow is handled using *overflow buckets*

Problems of Static Hashing

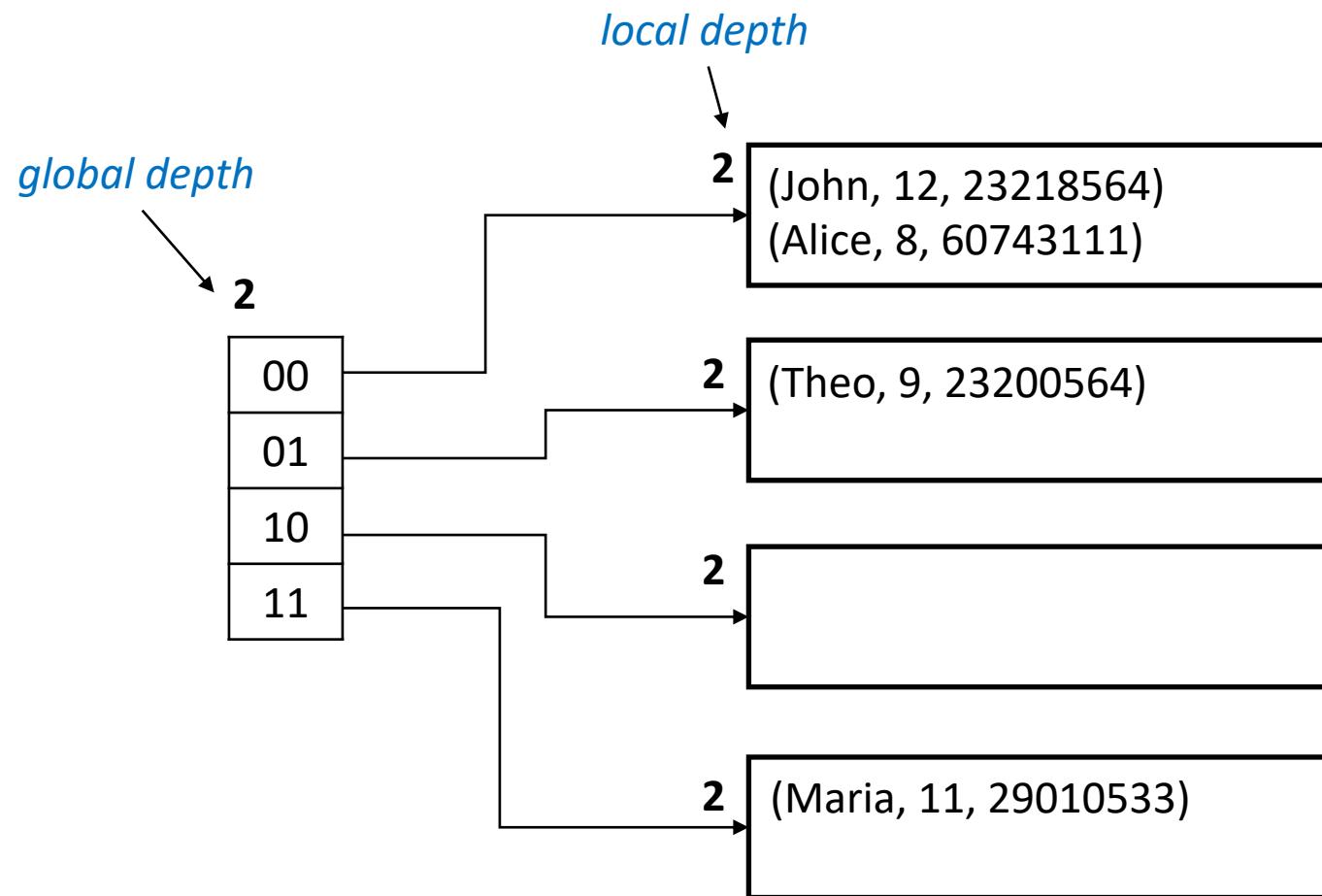
- In static hashing, there is a **fixed** number of buckets in the index
- Issues with this:
 - if the database grows, the number of buckets will be too small: long overflow chains degrade performance
 - if the database shrinks, space is wasted
 - reorganizing the index is expensive and can block query execution

Extendible Hashing

- **Extendible hashing** is a type of *dynamic* hashing
- It keeps a directory of pointers to buckets
- On overflow, it reorganizes the index by **doubling the directory** (and not the number of buckets)

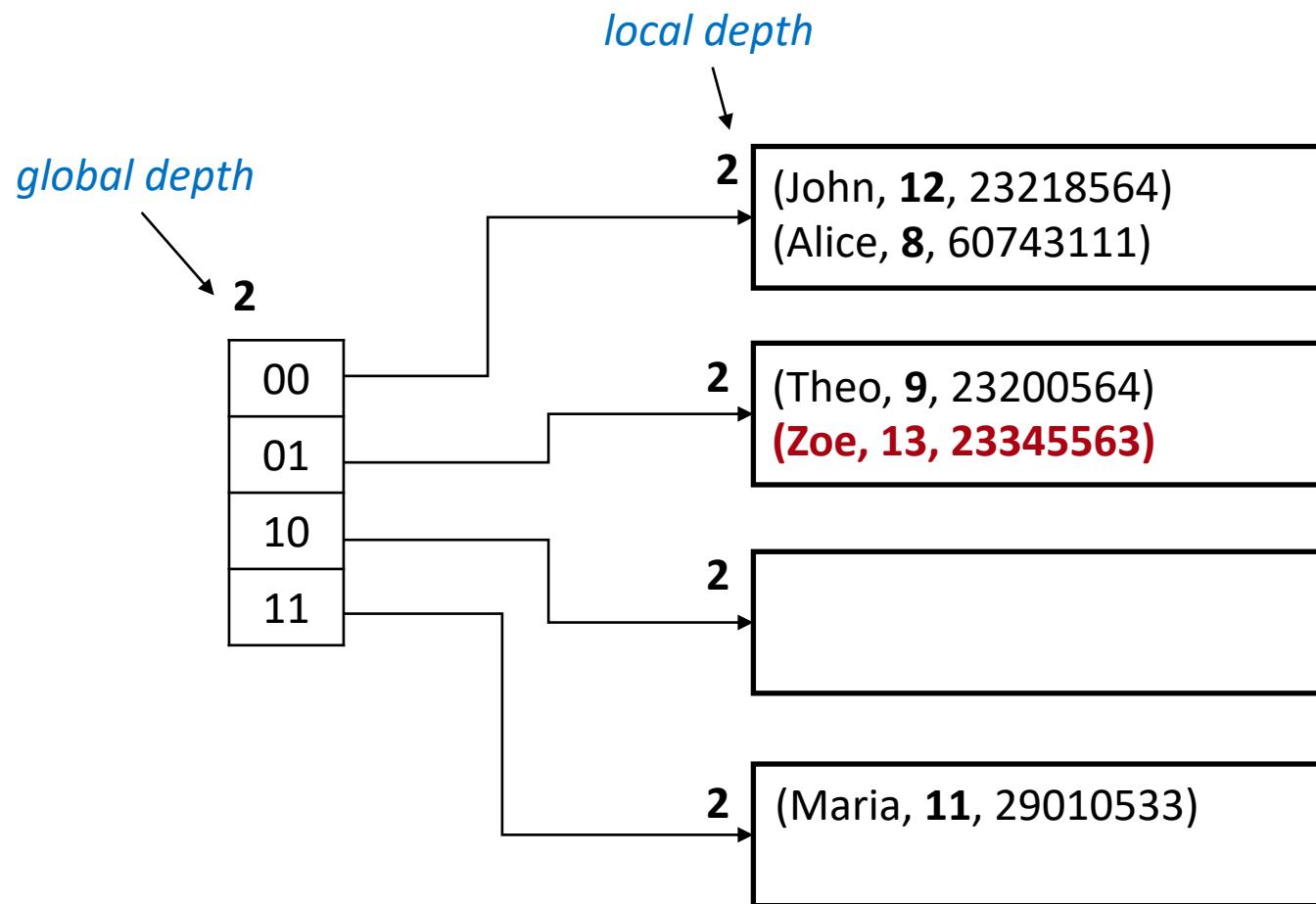
Extendible Hashing

To search, use the last **2** digits of the **binary** form of the search key value



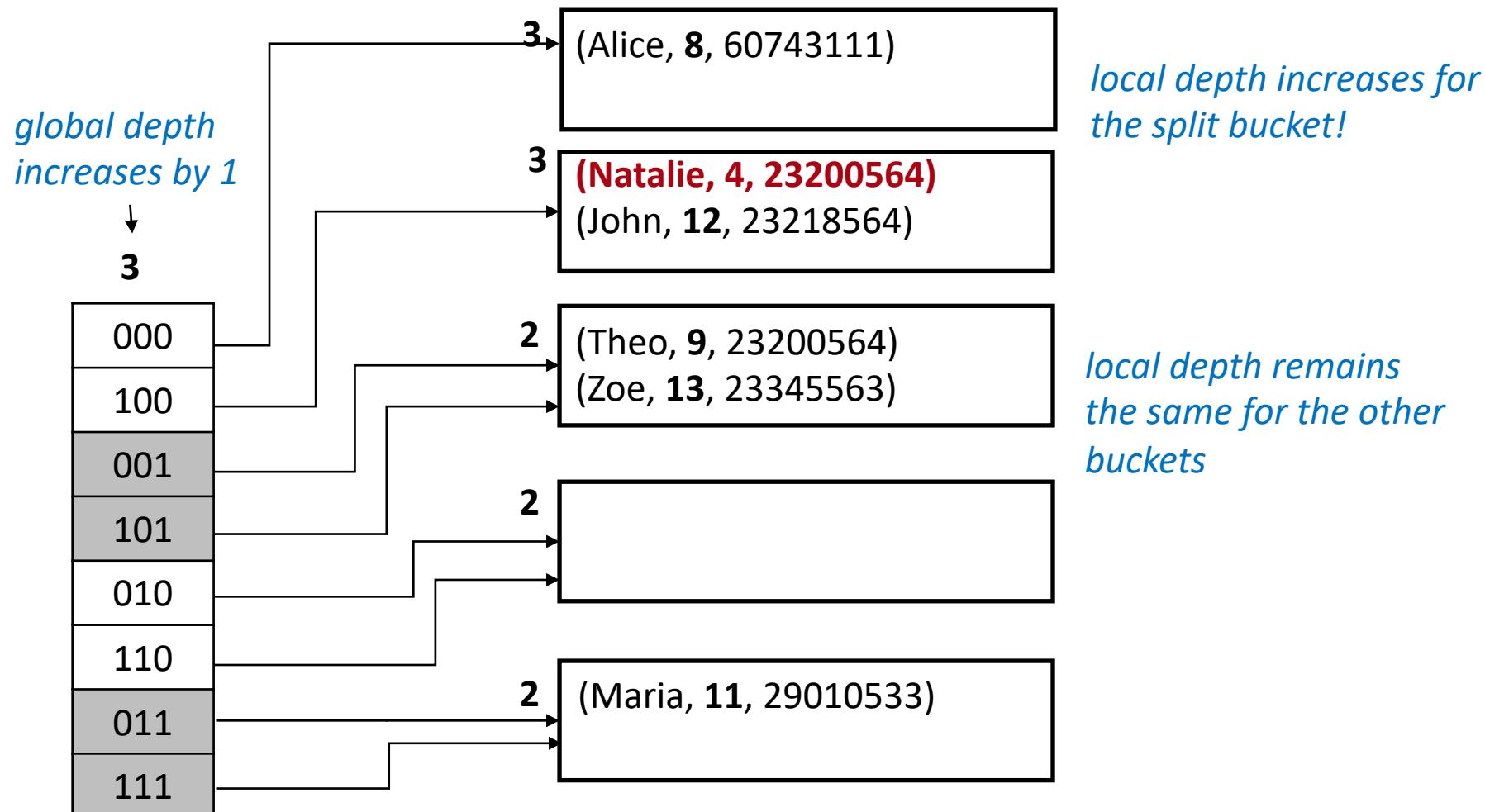
Extendible Hashing: Insert

If there is space in the bucket, simply add the record



Extendible Hashing: Insert

If the bucket is full, split the bucket and redistribute the entries



Extendible Hashing: Delete

- Locate the bucket of the record and remove it
- If the bucket becomes empty, it can be removed (and update the directory)
- Two buckets can also be coalesced together if the sum of the entries fit in a single bucket
- Decreasing the size of the directory can also be done, but it is expensive

More on Extendible Hashing

- How many disk accesses for equality search?
 - One if directory fits in memory, else two
- Directory grows in spurts, and, if the distribution of hash values is skewed, the directory can grow very large
- We may need overflow pages when multiple entries have the same hash

Bitmap indexes

Motivation

Consider the following table:

```
CREATE TABLE Tweets (
    uniqueMsgID INTEGER, -- unique message id
    tstamp      TIMESTAMP, -- when was the tweet posted
    uid         INTEGER, -- unique id of the user
    msg         VARCHAR (140), -- the actual message
    zip         INTEGER, -- zipcode when posted
    retweet     BOOLEAN -- retweeted?
);
```

Consider the following query, Q1:

```
SELECT * FROM Tweets
WHERE uid = 145;
```

And, the following query, Q2:

```
SELECT * FROM Tweets
WHERE zip BETWEEN 53000 AND 54999
```

Speed-up queries using a B+-tree for the uid and the zip values.

Motivation

Consider the following table:

```
CREATE TABLE Tweets (
    uniqueMsgID INTEGER, -- unique message id
    tstamp      TIMESTAMP, -- when was the tweet posted
    uid         INTEGER, -- unique id of the user
    msg         VARCHAR (140), -- the actual message
    zip         INTEGER, -- zipcode when posted
    retweet     BOOLEAN -- retweeted?
);
```

In a B+-tree, how many bytes do we use for each record?

At least key + rid,
so key-size+rid-size

Can we do better, i.e. an index with lower storage overhead? Especially for attributes with small domain cardinalities?

Bit-based indices: Two flavors
a) *Bitmap indices and*
b) *Bitslice indices*

Bitmap Indices

- Consider building an index to answer equality queries on the **retweet** attribute
- Issues with building a B-tree:
 - Three distinct values: True, False, NULL
 - Lots of duplicates for each distinct value
 - Sort of an odd B-tree with three long rid lists
- Bitmap Index: Build three bitmap arrays (stored on disk), one for each value.
 - The i^{th} bit in each bitmap correspond to the i^{th} tuple (need to map i^{th} position to a rid)

Bitmap Example

Table (stored in a heapfile)

uniqueMsgID	...	zip	retweet
1		11324	Y
2		53705	Y
3		53706	N
4		53705	NULL
5		90210	N
...
1,0000,000,000		53705	Y

Bitmap index on “retweet”

R-Yes	R-No	R-Null
1	0	0
1	0	0
0	1	0
0	0	1
0	1	0
...
1	0	0

```
SELECT * FROM Tweets WHERE retweet = 'N'
```

1. Scan the R-No Bitmap file
2. For each bit set to 1, compute the tuple #
3. Fetch the tuple # (s)

Critical Issue

- Need an efficient way to compute a bit position
 - Layout the bitmap in page id order.
 - Need an efficient way to map a bit position to a record.
- How?
1. If you fix the # records per page in the heapfile
 2. And lay the pages out so that page #s are sequential and increasing
 3. Then can construct **rid (page-id, slot#)**
 - **page-id** = Bit-position / #records-per-page
 - **slot#** = Bit-position % #records-per-page

Implications
of #1?

With variable length records, have to set the limit based on the size of the largest record, which may result in under-filled pages.

Other Queries

Table (stored in a heapfile)

uniqueMsgID	...	zip	retweet
1		11324	Y
2		53705	Y
3		53706	N
4		53705	NULL
5		90210	N
...
1,0000,000,000		53705	Y

Bitmap index on “retweet”

R-Yes	R-No	R-Null
1	0	0
1	0	0
0	1	0
0	0	1
0	1	0
...
1	0	0

```
SELECT COUNT(*) FROM Tweets WHERE retweet = 'N'
```

```
SELECT * FROM Tweets WHERE retweet IS NOT NULL
```

2. Storing a bitmap index

Storing the Bitmap index

- One bitmap for each value, and one for Nulls
- Need to store each bitmap
- Simple method: 1 file for each bitmap
- Can compress the bitmap!

Index size? $\# \text{tuples} * (\text{cardinality of the domain} + 1)$ bits

When is a bitmap index more space efficient than a B+-tree?

$\#\text{distinct values} < \text{data entry size in the B+-tree}$

3. Bit-sliced Index

Bit-sliced Index: Motivation

(Re)consider the following table:

```
CREATE TABLE Tweets (
    uniqueMsgID INTEGER,          -- unique message id
    tstamp      TIMESTAMP,        -- when was the tweet posted
    uid         INTEGER,          -- unique id of the user
    msg         VARCHAR (140),   -- the actual message
    zip         INTEGER,          -- zipcode when posted
    retweet     BOOLEAN           -- retweeted?
);
```

```
SELECT * FROM Tweets WHERE zip = 53706
```

Would we build a bitmap index on zipcode?

Bit-sliced index

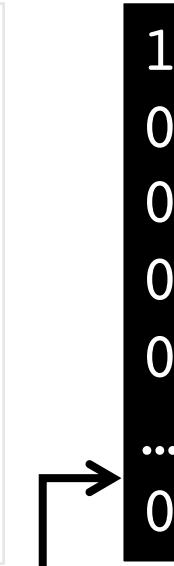
Why do we have 17 bits for zipcode?

Table

uniqueMsgID	...	zip	retweet
1		11324	Y
2		53705	Y
3		53706	N
4		53705	NULL
5		90210	N
...
1,000,000,000		53705	Y

Slice 16	Bit-sliced index (1 slice per bit)	Slice 1	Slice 0
	00010110000111100		1
	01101000111001001	0	0
	01101000111001010	0	0
	01101000111001001	0	0
	10110000001100010	0	0

	01101000111001001	0	0



Query evaluation: Walk through each slice constructing a **result bitmap**

e.g. zip \leq 11324, skip entries that have 1 in the first three slices (16, 15, 14)

(Null bitmap is not shown)

Bitslice Indices

- Can also do aggregates with Bitslice indices
 - E.g. SUM(attr): Add bit-slice by bit-slice.

First, count the number of 1s in the **slice17**, and multiply the count by 2^{17}
Then, count the number of 1s in the **slice16**, and multiply the count by ...
- Store each slice using methods like what you have for a bitmap.
 - Note once again can use compression

Bitmap v/s Bitslice

- Bitmaps better for low cardinality domains
- Bitslice better for high cardinality domains
- Generally easier to “do the math” with bitmap indices

ECE:5995 Modern Databases

Intro to RDBMS, Postgres, and CRUD

Relational databases

- Relational database management systems (RDBMS) store data in tables or relations
- A table is a collection of related data held in a structured format within a database.
- Tables are uniquely identified by their names
- Tables (relations) are comprised of columns (attributes) and rows (tuples).
 - Columns contain the column name, data type, and any other attributes.
 - Rows contain the records or data for the columns.

Last Name	First Name	Address	City
Jones	Sarah	123 Main Street	Orlando
Smith	Penny	567 First Street	Tampa
Reed	Gary	890 Third Street	Jacksonville

Structured Query Language (SQL)

- Standard for RDBMS
- Originally proposed by E.F.Codd in 1970
- Oracle adopted in late 1970s
- IBM's SQL/DS in 1981, and DB2 in 1983
- ANSI and ISO published SQL standards in 1986, called SQL-1, later revised SQL-89, SQL-2 (1999), SQL-3 (2003, 2008, 2011, and 2016)

Used today in PostgreSQL, Microsoft SQL Server, MySQL, Informix, Sybase, dBase, Paradox, r:Base, FoxPro, and others

Most vendors support standard, but have slight variations of their own

Components of SQL

- Data definition language – DDL

CREATE TABLE

ALTER TABLE

DROP TABLE

CREATE VIEW

CREATE INDEX

RENAME TABLE

DROP INDEX

- Data manipulation language – DML

- Create, Read, Update, Delete (CRUD)

INSERT

SELECT

UPDATE

DELETE

- Authorization language – grant privileges to users

Relationships between tables

- Larger tables can be broken into smaller tables and still maintain a relationship
- We can create relationships between two tables, through matching the data from one column in a table with the data in a column in the second table
- Two SQL relational concepts that allows to do this:
 - **Primary Key.** Primary Key is a column or a combination of columns that uniquely identifies each row in a table.
 - **Foreign Key.** Foreign Key is a column or a combination of columns whose values match a Primary Key in a different table.
- Three types of relations between tables:
 1. **One-To-Many** (Most Common) - a row in one of the tables can have many matching rows in the second table, but a row in the second table can match only one row in the first table
 2. **Many-To-Many** - many rows from the first table can match many rows in the second and the other way around
 3. **One-To-One** - each row in the first table may match only one row in the second and the other way around

Data Integrity

- Data integrity refers to the overall completeness, accuracy and consistency of data.
- Data integrity is usually imposed during the database design phase through the use of standard procedures and rules as part of normalization.
- Three integrity constraints to achieve data integrity:
 1. **Entity Integrity:** Every table must have its own primary key and that has to be unique and not null
 2. **Referential Integrity:** This is the concept of foreign keys. The rule states that the foreign key value can be in two states. The first state is that the foreign key value would refer to a primary key value of another table, or it can be null. Being null could simply mean that there are no relationships, or that the relationship is unknown.
 3. **Domain Integrity:** This states that every column in a relational database is in a defined domain.
- The concept of **data integrity ensures that all data in a database can be traced and connected to other data.**
- **Normalization** ensures data integrity.

PostgreSQL

- PostgreSQL is an enterprise-class, open-source RDBMS
- More than 20 years of development by the open-source community.
- PostgreSQL project started in 1986 at Berkeley, named POSTGRES, in reference to the older Ingres database which also developed at Berkeley.
- PostgreSQL support most popular programming languages:
 - Python, Java, C#, C/C+, Ruby, JavaScript (Node.js), Perl, Go, and others
- PostgreSQL offer many advanced features:
 - User-defined types, table inheritance
 - Foreign key referential integrity, views, rules, subquery
 - Sophisticated locking mechanism, nested transactions (savepoints)
 - Multi-version concurrency control (MVCC), asynchronous replication
- Many companies have built products and solutions based on PostgreSQL. Some featured companies are Apple, Fujitsu, Red Hat, Cisco, Juniper Network, Instagram, etc.

Let's give it a try

Webserver:

<https://s-l112.engr.uiowa.edu/phpPgAdmin/>

NOTE: The Postgres databases are hosted on a campus subnet and will only be accessible through eduroam or VPN.

Check ICON for your group number under the postgres groups, i.e. if you are in group 1, your username will be student01 and password engr-2020-01)

username: studentXX

password: engr-2020-XX

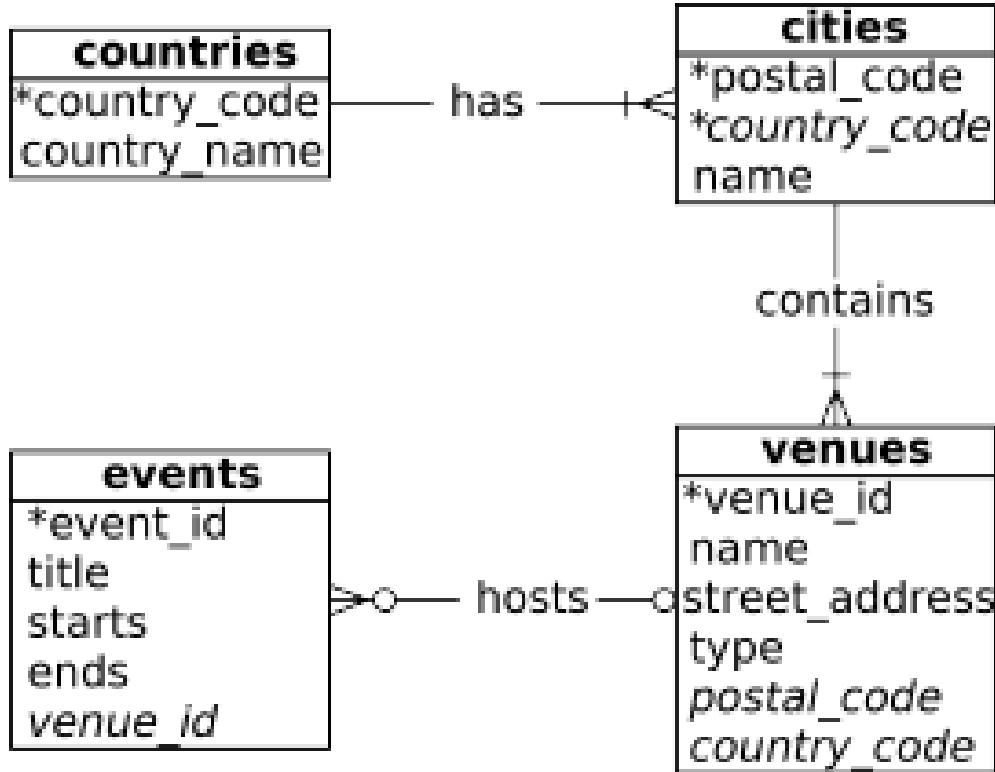
We will be using phpPgAdmin to interact with the database.

If you want to setup psql or pgAdmin in your own computer:

<https://www.enterprisedb.com/postgres-tutorials/connecting-postgresql-using-psql-and-pgadmin>

hostname: s-l112.engr.uiowa.edu

Today's example



Create a new table named **events**. Your events table should have these columns: a SERIAL integer event_id (primary key), a title, starts and ends (of type timestamp), and a venue_id (foreign key that references **venues**).

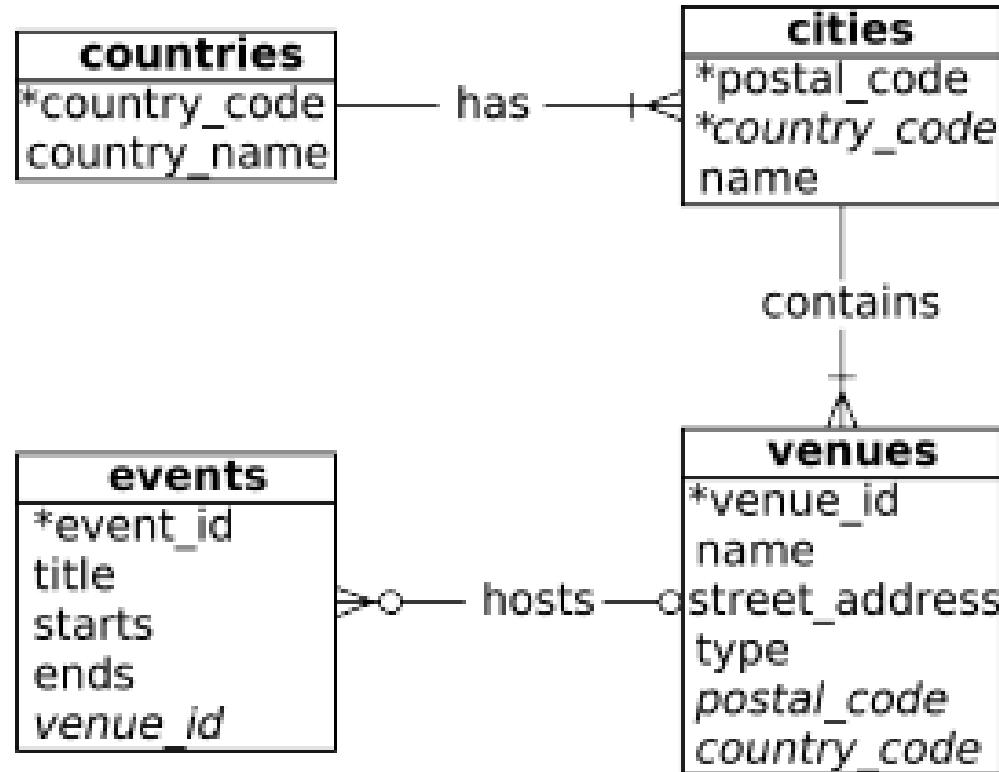
After creating the **events** table, INSERT the following values (timestamps are inserted as a string like *2018-02-15 17:30*) :

title	starts	ends	venue_id	event_id
Fight Club	2018-02-15 17:30:00	2018-02-15 19:30:00	2	1
April Fools Day	2018-04-01 00:00:00	2018-04-01 23:59:00		2
Christmas Day	2018-02-15 19:30:00	2018-12-25 23:59:00		3

Today's terms

Term	Definition
Column	A domain of values of a certain type, sometimes called an <i>attribute</i>
Row	An object comprised of a set of column values, sometimes called a <i>tuple</i>
Table	A set of rows with the same columns, sometimes called a <i>relation</i>
Primary key	The unique value that pinpoints a specific row
Foreign key	A data constraint that ensures that each entry in a column in one table uniquely corresponds to a row in another table (or even the same table)
CRUD	Create, Read, Update, Delete
SQL	Structured Query Language, the <i>standard</i> language of a relational database
Join	Combining two tables into one by some matching columns

Our working example



Let's add more data to countries and cities.

Indexes: B-TREES and Hashing

Files, pages, and records

The raw disk space is organized into **files**.

Files are made up of **pages**, and pages contain **records**

Data is allocated and deallocated in increments of pages.

- **File**: A collection of pages
Page: a collection of records.
- File operations:
 - insert/delete/modify record
 - read a particular record (specified using the *record id*)
 - scan all records (possibly with some conditions on the records to be retrieved)

Unordered (Heap) Files

- Simplest file structure contains records in no particular order.
- As file grows and shrinks, disk pages are allocated and de-allocated.
- To support record level operations, the system must:
 - keep track of the *pages* in a file: **page id (pid)**
 - keep track of *free space* on pages
 - keep track of the *records* on a page: **record id (rid)**
 - Many alternatives for keeping track of this information
- Operations: create/destroy file, insert/delete record, fetch a record with a specified **rid**, scan all records

Indexes

- A Heap file allows us to retrieve records:
 - by specifying the *rid*, or
 - by scanning all records sequentially
- Sometimes, we want to retrieve records by specifying the *values in one or more fields*, e.g.,
 - Find all students in the “ECE” department
 - Find all students with a gpa > 3
- Indexes are file structures that enable us to answer such *value-based queries* efficiently.

Motivation

Consider the following table:

```
CREATE TABLE Tweets (
    uniqueMsgID INTEGER,          -- unique message id
    timestamp TIMESTAMP,           -- when was the tweet posted
    uid INTEGER,                  -- unique id of the user
    msg VARCHAR (140),            -- the actual message
    zip INTEGER                   -- zipcode when posted
);
```

Consider the following query, Q1: `SELECT * FROM Tweets
WHERE uid = 145;`

And, the following query, Q2: `SELECT * FROM Tweets
WHERE zip BETWEEN 53000 AND 54999`

Ways to evaluate the queries, efficiently?

1. Store the table as a heapfile, scan the file. I/O Cost?
2. Store the table as a **sorted file**, binary search the file. I/O Cost?
3. Store the table as a heapfile, build an **index**, and search using the index.
4. Store the table in an **index** file. The entire tuple is stored in the index!

Index

- Two main types of indices
 - **Hash** index: good for equality search (e.g. Q1)
 - **B-tree** index: good for both range search (e.g. Q2) and equality search (e.g. Q1)
 - Generally a hash index is faster than a B-tree index for equality search
- Hash indices aim to get $O(1)$ I/O and CPU performance for search and insert
- B-Trees have $O(\log_F N)$ I/O and CPU cost for search, insert and delete.

What is in the index

- Two things: **index key** and **some value**
 - Insert(indexKey, value)
 - Search (indexKey) → value (s)
- What is the index key for Q1 and Q2?
- Consider Q3:

```
SELECT * FROM Tweets  
WHERE uid = 145 AND  
zip BETWEEN 53000 AND 54999
```

- Value:
 - Record id
 - List of record id
 - The entire tuple!

Hash indexes

- *Hash-based indexes are best for equality selections*
 - Cannot support range searches, except by generating all values
 - Static and dynamic hashing techniques exist
- *Hash indexes not as widespread as B+-Trees*
 - Some DBMS do not provide hash indexes (Postgres does)
 - But hashing still useful in query optimizers (DB Internals)
 - E.g., in case of equality joins

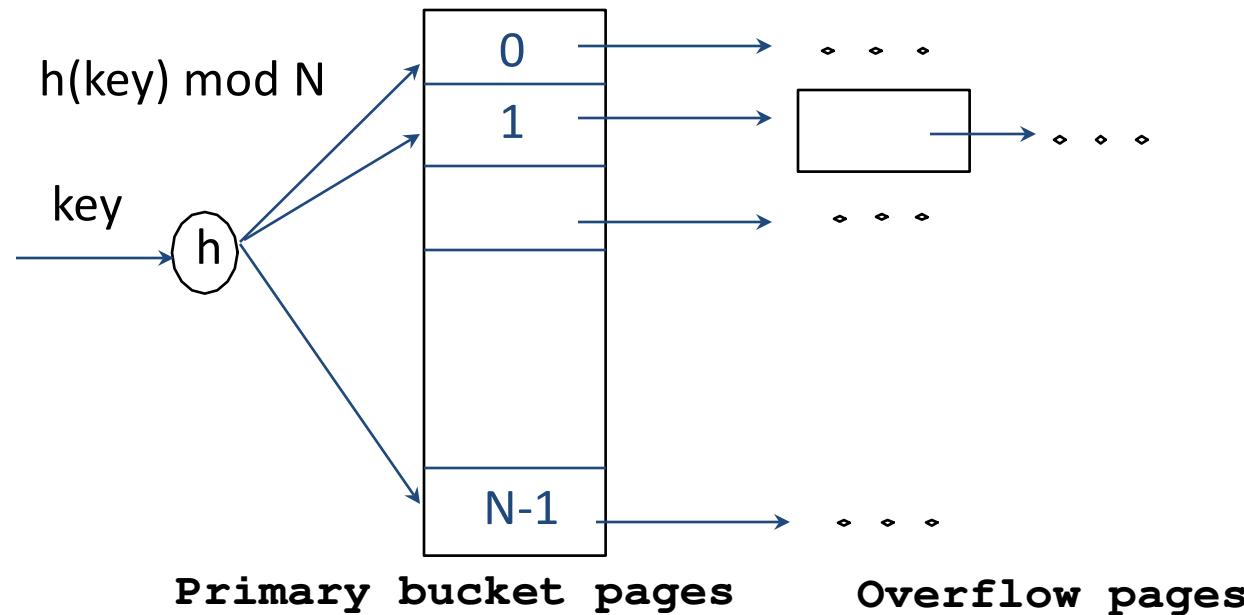
Static Hashing

Number of primary pages N fixed, allocated sequentially

overflow pages may be needed when file grows

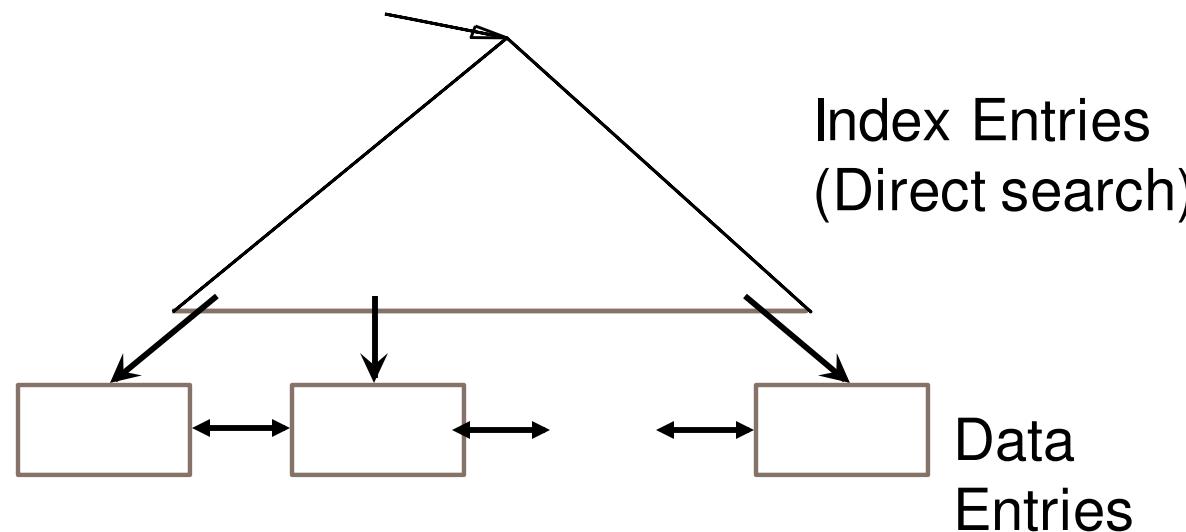
Buckets contain data entries

$h(k) \bmod N = \text{bucket for data entry with key } k$



(Ubiquitous) B+ Tree

- Height-balanced (dynamic) tree structure
- Insert/delete at $\log_F N$ cost (F = fanout, N = # leaf pages)
- Minimum 50% occupancy (except for root).
Each node contains $d \leq m \leq 2d$ entries. The parameter d is called the **order** of the tree.
- Supports equality and range-searches efficiently.



Index Entries

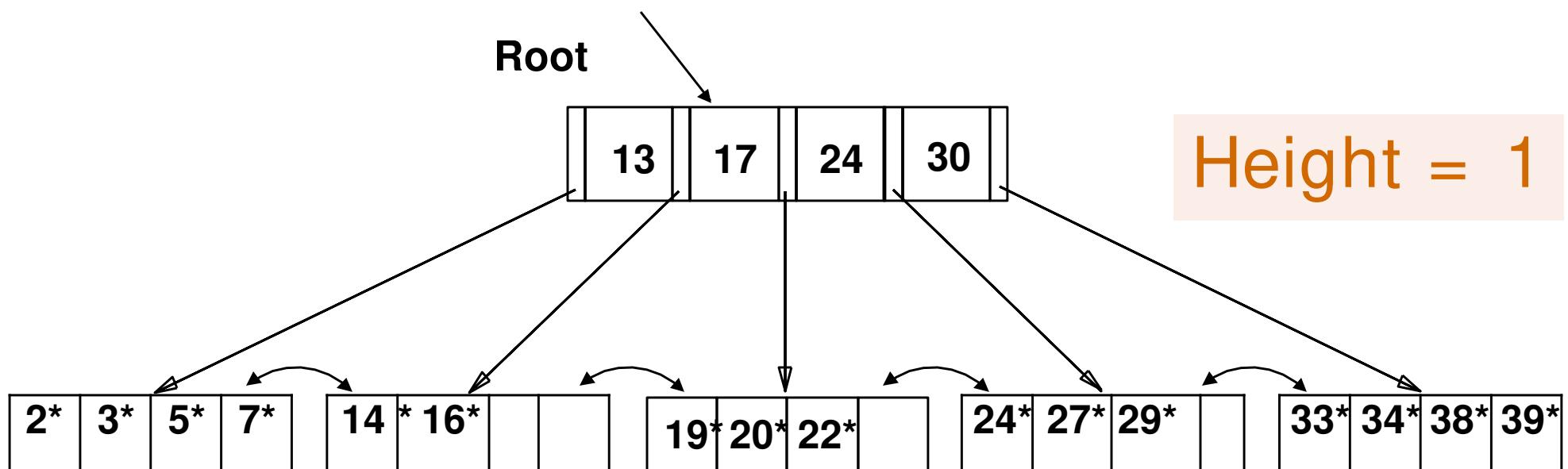
Entries in the index
(i.e. non-leaf) pages:
(search key value, pageid)

Data Entries

Entries in the leaf pages:
(search key value, recordid)

Example B+ Tree

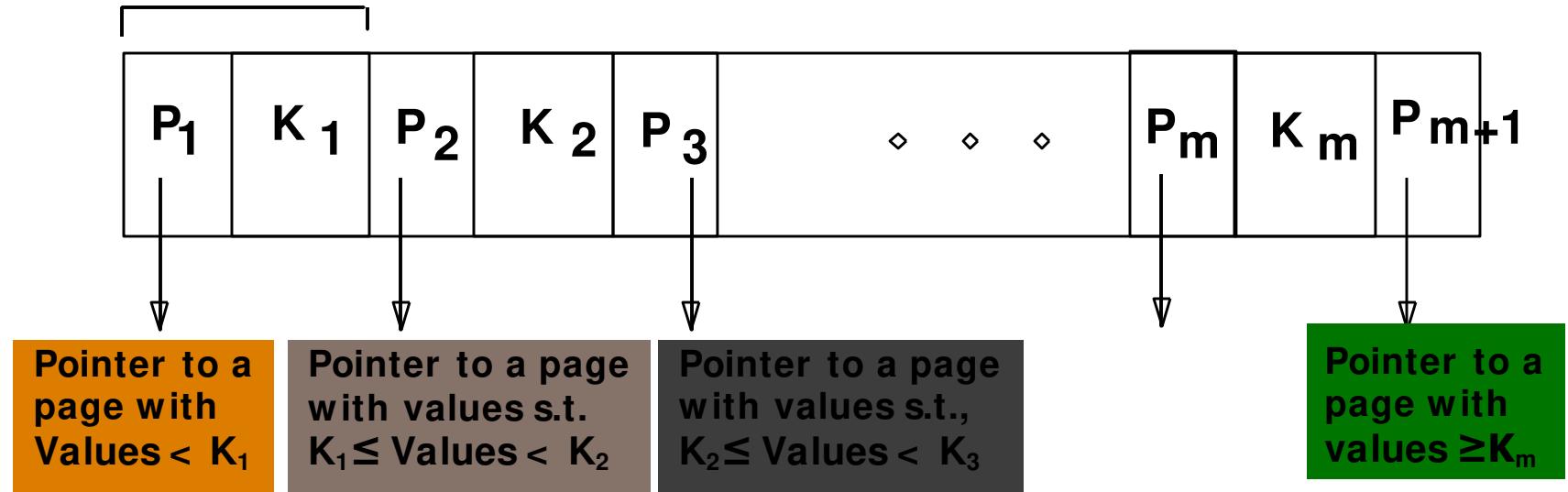
- Search: Starting from root, examine index entries in non-leaf nodes, and traverse down the tree until a leaf node is reached
 - Non-leaf nodes can be searched using a binary or a linear search.
- Search for 5*, 15*, all data entries $\geq 24^*$



B+-tree Page Format

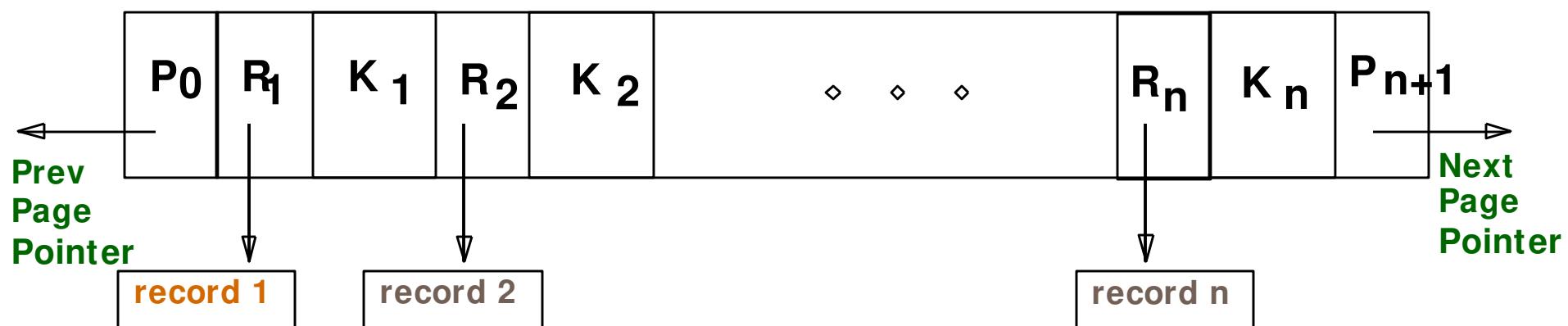
Non-leaf Page

index entries



Leaf Page

data entries



B+ Trees in Practice

- Typical order: 100. Typical fill-factor: 67%.
 - average fanout = 133
- Typical capacities:
 - Height 4: $133^4 = 312,900,700$ records
 - Height 3: $133^3 = 2,352,637$ records
- Can often hold top levels in buffer pool:
 - Level 1 = 1 page = 8 Kbytes
 - Level 2 = 133 pages = 1 Mbyte
 - Level 3 = 17,689 pages = 133 MBytes

System Catalogs/Data dictionary

- ❖ For each index:
 - structure (e.g., B+ tree) and search key fields
- ❖ For each relation:
 - name, file name, file structure (e.g., Heap file)
 - attribute name and type, for each attribute
 - index name, for each index
 - integrity constraints
- ❖ For each view:
 - view name and definition
- ❖ Plus statistics, authorization, buffer pool size, etc.
 - ☛ *Catalogs are themselves stored as relations!*

Alternatives for Data Entry k^* in Index

- ❖ Three alternatives:
 - ① Data record with key value k
 - ② $\langle k, \text{rid of data record with search key value } k \rangle$
 - ③ $\langle k, \text{list of rids of data records with search key } k \rangle$
- ❖ Choice of alternative for data entries is orthogonal to the indexing technique used to locate data entries with a given key value k .
 - Examples of indexing techniques: B+ trees, hash-based structures
 - Typically, index contains auxiliary information that directs searches to the desired data entries

Alternatives for Data Entries (Contd.)

❖ Alternative 1:

- If this is used, index structure is a file organization for data records (like Heap files or sorted files).
- At most one index on a given collection of data records can use Alternative 1. (Otherwise, data records duplicated, leading to redundant storage and potential inconsistency.)
- If data records very large, # of pages containing data entries is high. Implies size of auxiliary information in the index is also large, typically.

Alternatives for Data Entries (Contd.)

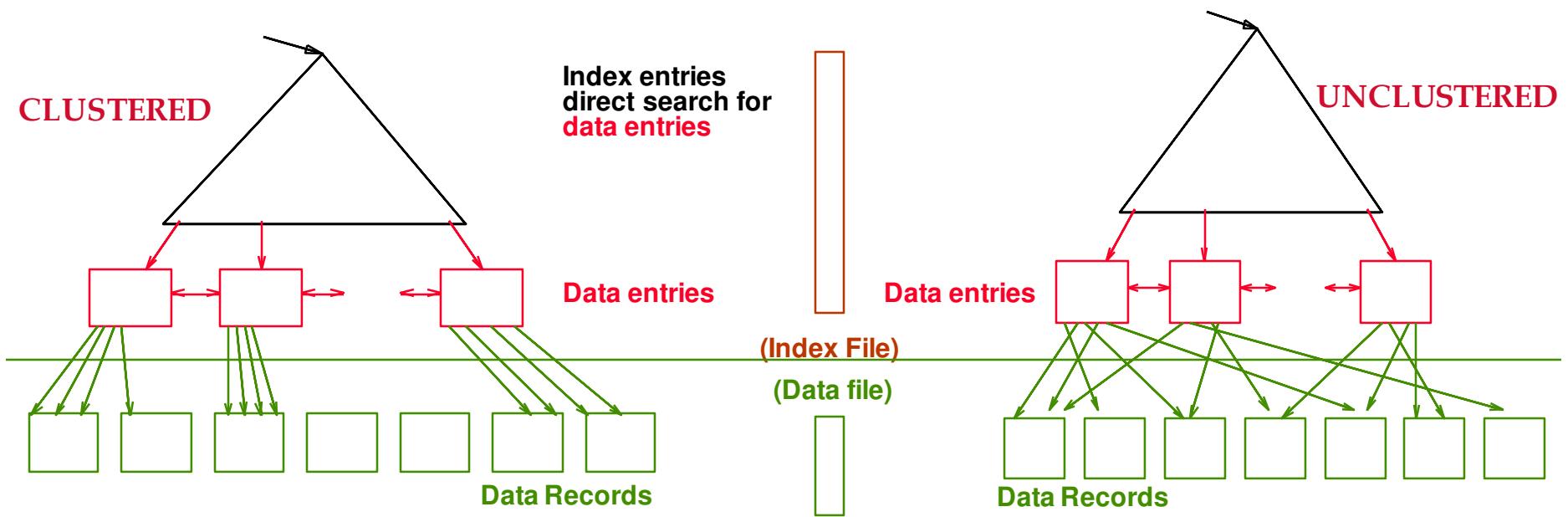
- ❖ Alternatives 2 and 3:
 - Data entries typically much smaller than data records. So, better than Alternative 1 with large data records, especially if search keys are small. (Portion of index structure used to direct search is much smaller than with Alternative 1.)
 - If more than one index is required on a given file, at most one index can use Alternative 1; rest must use Alternatives 2 or 3.
 - Alternative 3 more compact than Alternative 2, but leads to variable sized data entries even if search keys are of fixed length.

Index Classification

- ❖ *Primary vs. secondary*: If search key contains primary key, then called primary index.
 - *Unique* index: Search key contains a candidate key.
- ❖ *Clustered vs. unclustered*: If order of data records is the same as, or ‘close to’, order of data entries, then called clustered index.
 - Alternative 1 implies clustered, but not vice-versa.
 - A file can be clustered on at most one search key.
 - Cost of retrieving data records through index varies *greatly* based on whether index is clustered or not!

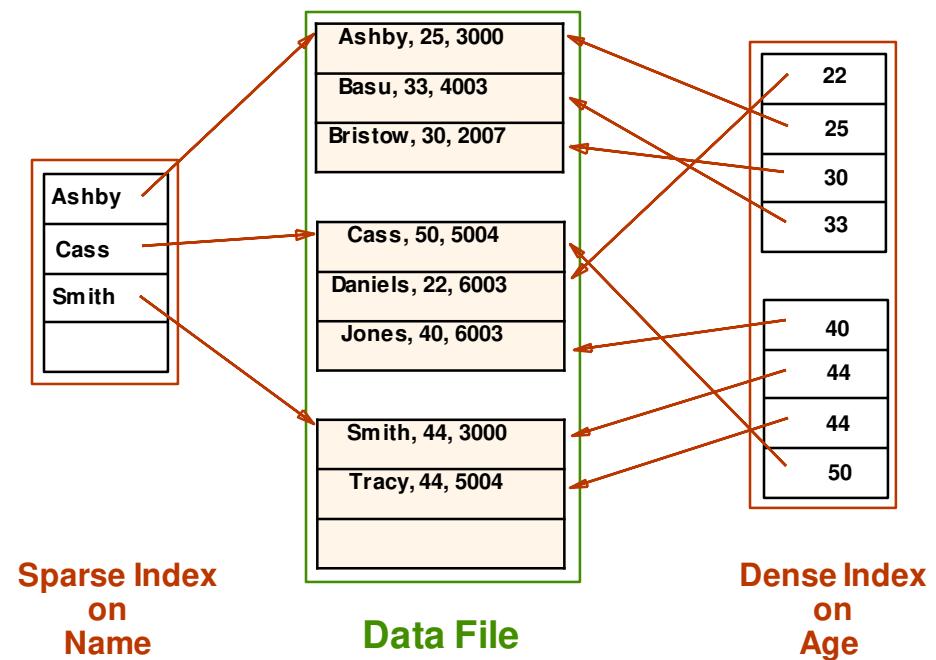
Clustered vs. Unclustered Index

- ❖ Suppose that Alternative (2) is used for data entries, and that the data records are stored in a Heap file.
 - To build clustered index, first sort the Heap file (with some free space on each page for future inserts).
 - Overflow pages may be needed for inserts. (Thus, order of data recs is ‘close to’, but not identical to, the sort order.)



Index Classification (Contd.)

- ❖ *Dense vs. Sparse:* If there is at least one data entry per search key value (in some data record), then dense.
 - Alternative 1 always leads to dense index.
 - Every sparse index is clustered!
 - Sparse indexes are smaller; however, some useful optimizations are based on dense indexes.



Index Classification (Contd.)

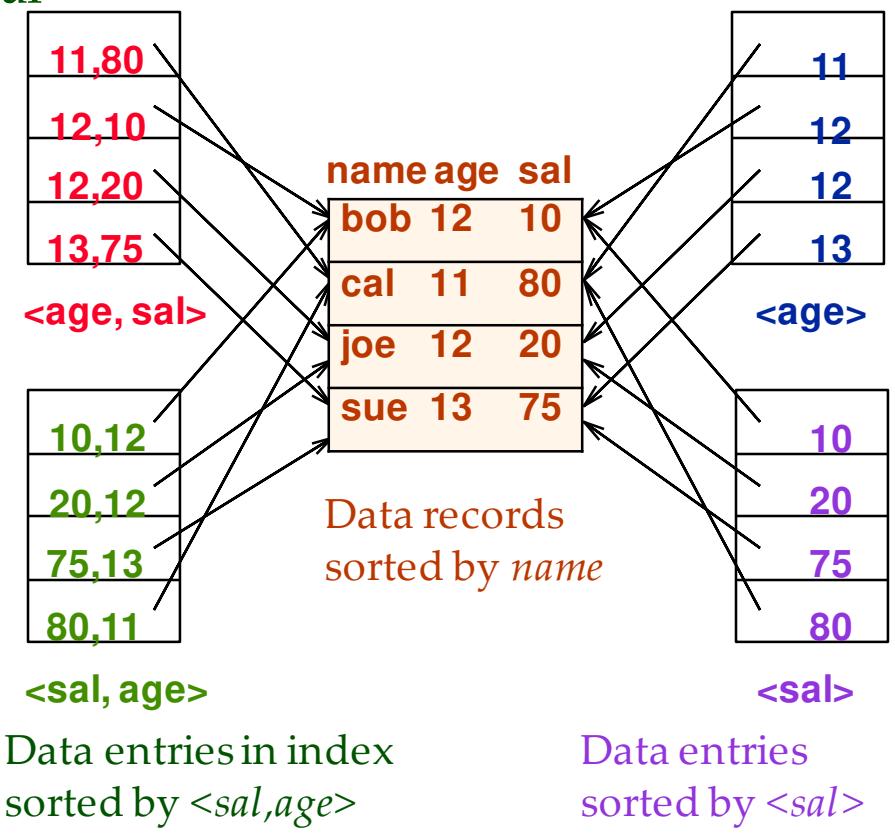
❖ *Composite Search Keys: Search on a combination of fields.*

- Equality query: Every field value is equal to a constant value. E.g. wrt $\langle \text{sal}, \text{age} \rangle$ index:
 - ◆ age=20 and sal =75
- Range query: Some field value is not a constant. E.g.:
 - ◆ age =20; or age=20 and sal > 10

❖ *Data entries in index sorted by search key to support range queries.*

- Lexicographic order, or
- Spatial order.

Examples of composite key indexes using lexicographic order.



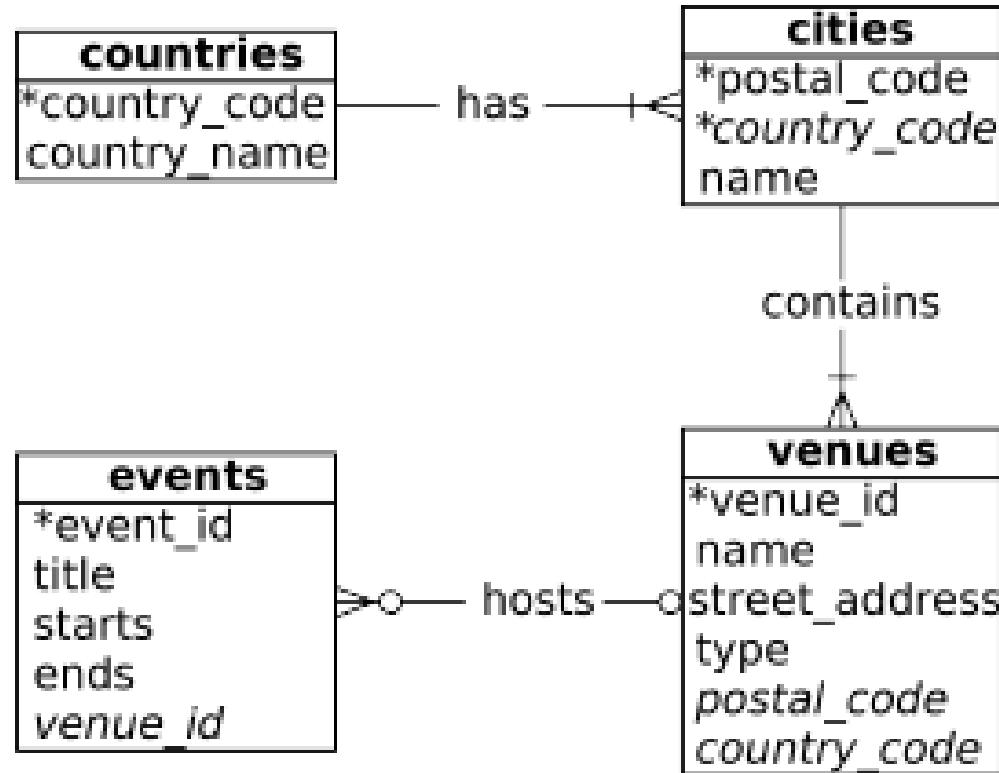
For today...

1. Write a query that finds the country name of the Fight Club event. Store it as a view.
2. Alter the venues table such that it contains a Boolean column called active with a default value of TRUE.
3. Add a b-tree on table cities over country_code

QUERIES and AGGREGATES

See <http://sqlzoo.net/>

Our working example



Let's add more venues and events.

Basic SQL Query

```
SELECT [DISTINCT] target-list
FROM   relation-list
WHERE  qualification
```

Optional

Attributes from input relations

List of relations

Attr1 **op** Attr2
OPS: < , > , = , <= , >= , <>
Combine using AND, OR, NOT

- Semantics/Conceptual evaluation strategy:
 - Compute the cross-product of *relation-list*.
 - Discard resulting tuples if they fail *qualifications*.
 - Delete attributes that are not in *target-list*.
 - If DISTINCT is specified, eliminate duplicate rows.
- Not an efficient evaluation plan! (Optimizer picks efficient plans)

Find venues with events on '19-Oct-2019'

```
SELECT v.name as Venue  
      FROM events e, venues v  
     WHERE e.venue_id = v.venue_id  
       AND (DATE(e.starts) = '2019-10-19')
```

- Add DISTINCT to this query. Effect?
- Equivalent SQL using JOIN?

Expressions and Strings

```
SELECT e.title,  
       to_char(e.starts, 'FMDay, Mon FMDD YYYY HH12:MI AM') as starts  
  FROM events e  
 WHERE e.title LIKE '%visit%'  
 ORDER BY e.title;
```

- Illustrates date formatting and string pattern matching
- AS is a way to name fields in result
- LIKE is used for string matching. `_` stands for any one character and `%' stands for 0 or more arbitrary characters.

Find venues with events either in '22-Apr-2020' or '23-Apr-2020'

- UNION: Compute the union of two *union-compatible* sets of tuples
 - Same number/types of fields.
- Also available: INTERSECT and EXCEPT (What do we get if we replace UNION by EXCEPT?)
- SQL oddities: duplicates with union, except, intersect
 - Default: eliminate duplicates!
 - Use ALL to keep duplicates

```
SELECT v.name as Venue, v.country_code
FROM events e, venues v
WHERE e.venue_id = v.venue_id AND
(DATE(e.starts) = '2020-04-22' OR
DATE(e.starts) = '2020-04-23')
```

```
SELECT v.name as Venue, v.country_code
FROM events e, venues v
WHERE e.venue_id = v.venue_id
AND DATE(e.starts) = '2020-04-22'
```

UNION

```
SELECT v.name as Venue, v.country_code
FROM events e, venues v
WHERE e.venue_id = v.venue_id
AND DATE(e.starts) = '2020-04-23'
```

Find venues with events either in '22-Apr-2020' AND '23-Apr-2020'

- **INTERSECT:** Compute the intersection of any two *union-compatible* sets of tuples.
- In the SQL/92 standard, but some systems don't support it.

```
SELECT v.name as Venue, v.country_code  
FROM events e1, events e2, venues v  
WHERE e1.venue_id = v.venue_id  
AND e2.venue_id = v.venue_id  
AND DATE(e1.starts) = '2020-04-22'  
AND DATE(e2.starts) = '2020-04-23';
```

```
SELECT v.name as Venue, v.country_code  
FROM events e, venues v  
WHERE e.venue_id = v.venue_id AND  
DATE(e.starts) = '2020-04-22'
```

INTERSECT

```
SELECT v.name as Venue, v.country_code  
FROM events e, venues v  
WHERE e.venue_id = v.venue_id AND  
DATE(e.starts) = '2020-04-23';
```

Aggregate Operators

```
SELECT count(title) FROM events
```

```
SELECT COUNT (DISTINCT title)  
FROM events
```

```
SELECT min(starts), max(ends)  
      FROM events INNER JOIN venues  
        ON events.venue_id = venues.venue_id  
      WHERE venues.name = 'Crystal Ballroom';
```

```
SELECT e.title FROM events e  
  WHERE e.venue_id IN (SELECT v.venue_id FROM venues v  
    WHERE v.name = 'University of Iowa') AND  
  (e.ends-e.starts) = (SELECT max(e2.ends-e2.starts)  
    FROM events e2 WHERE e2.venue_id IS NOT NULL);
```

COUNT (*)
COUNT ([DISTINCT] A)
SUM ([DISTINCT] A)
AVG ([DISTINCT] A)
MAX (A) *Can use Distinct*
MIN (A) *Can use Distinct*

single column

Find date & title of the first event

- The first query is illegal! (wait for GROUP BY.)

```
SELECT e.title, min(e.starts)  
FROM events e
```

How many tuples
in the result?

```
SELECT e.title, e.starts FROM events e  
  
WHERE e.starts = (SELECT  
min(e2.starts) FROM events e2);
```

GROUP BY and HAVING

- Apply aggregate to each of several *groups* of tuples
- Find the number of events registered *for each venue*

For *each venue i*:

```
SELECT COUNT (*)
FROM events
WHERE venue_id = /
```

```
SELECT venue_id, count(* )
FROM events
GROUP BY venue_id;
```

Queries With GROUP BY and HAVING

```
SELECT      [DISTINCT] target-list
FROM        relation-list
WHERE       qualification
GROUP BY   grouping-list
HAVING     group-qualification
```

How many tuples
in the result?

- The **target-list** contains
 - Attribute names: must be a subset of *grouping-list*.
 - Terms with aggregate operations (e.g., COUNT (*)).
- The **group-qualification**
 - Must have a single value per group

Conceptual Evaluation

- Cross-product -> discard tuples -> apply projection
 - > partition into groups using the *grouping-list* attribute values
 - > eliminate groups that don't satisfy the *group-qualification*
- Expressions in *group-qualification* have a single value per group!
 - In effect, an attribute in *group-qualification* that is not an argument of an aggregate op also appears in *grouping-list*. (SQL does not exploit primary key semantics here!)
- One answer tuple is generated per qualifying group.

Find the venues with at least 2 events in 2019

```
SELECT venue_id, count(*)  
      FROM events  
     WHERE venue_id IS NOT NULL AND  
           EXTRACT(YEAR FROM starts)=2019  
     GROUP BY venue_id  
    HAVING count(*) >= 2;
```

Null Values

- Represent
 - *unknown* (e.g., rating not assigned) or
 - *inapplicable* (e.g., no spouse's name)
- Complications with nulls:
 - Operators to check if value is/is not *null*.
 - Is *rating > 8* true or false when *rating* is null?
 - Answer: Evaluate to unknown
 - What about **AND**, **OR** and **NOT** connectives?
 - Need 3-valued logic (true, false and *unknown*)
 - Not unknown = unknown
 - WHERE clause eliminates rows that **don't evaluate to true**
 - New operators (in particular, *outer joins*) possible/needed.

p	q	p AND q	p OR q
T	T	T	T
T	F	F	T
T	U	U	T
F	T	F	T
F	F	F	F
F	U	F	U
U	T	U	T
U	F	F	U
U	U	U	U



Window functions – Partition by

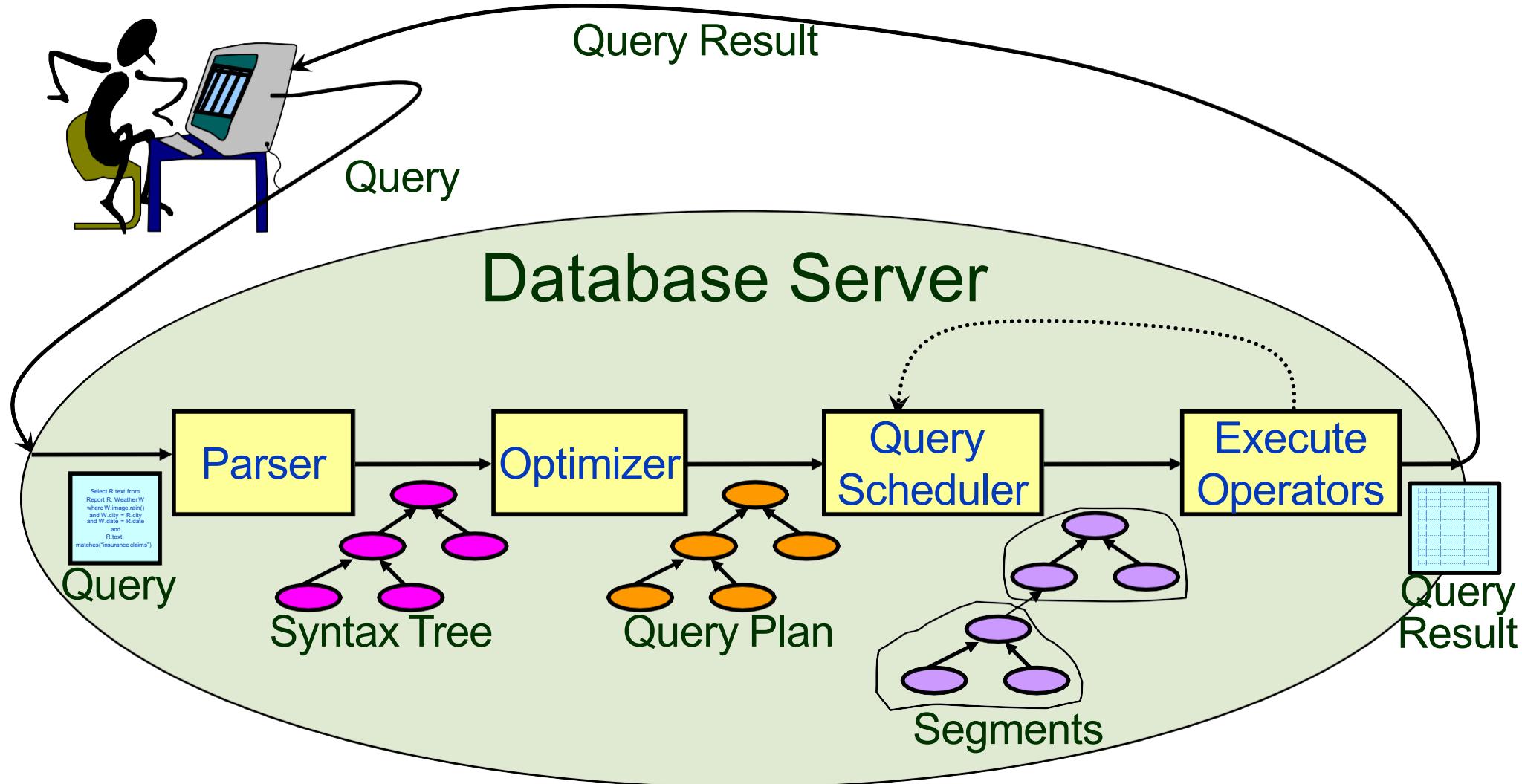
- Not all RDBMS support it
- Similar to GROUP BY
 - Calculates aggregates on the OVER a PARTITION of the result set.
 - Rather than grouping the results outside of the SELECT attribute list, it returns grouped values as any other field

```
SELECT venue_id, count(*)  
      OVER (PARTITION BY venue_id)  
    FROM events  
  ORDER BY venue_id;
```

```
SELECT v.venue_id, v.name, e.title, count(*)  
      OVER (PARTITION BY v.venue_id)  
    FROM events e INNER JOIN venues v  
  ON e.venue_id = v.venue_id;
```

JOINS and QUERY PROCESSING

Life Cycle of a Query



Problem Statement

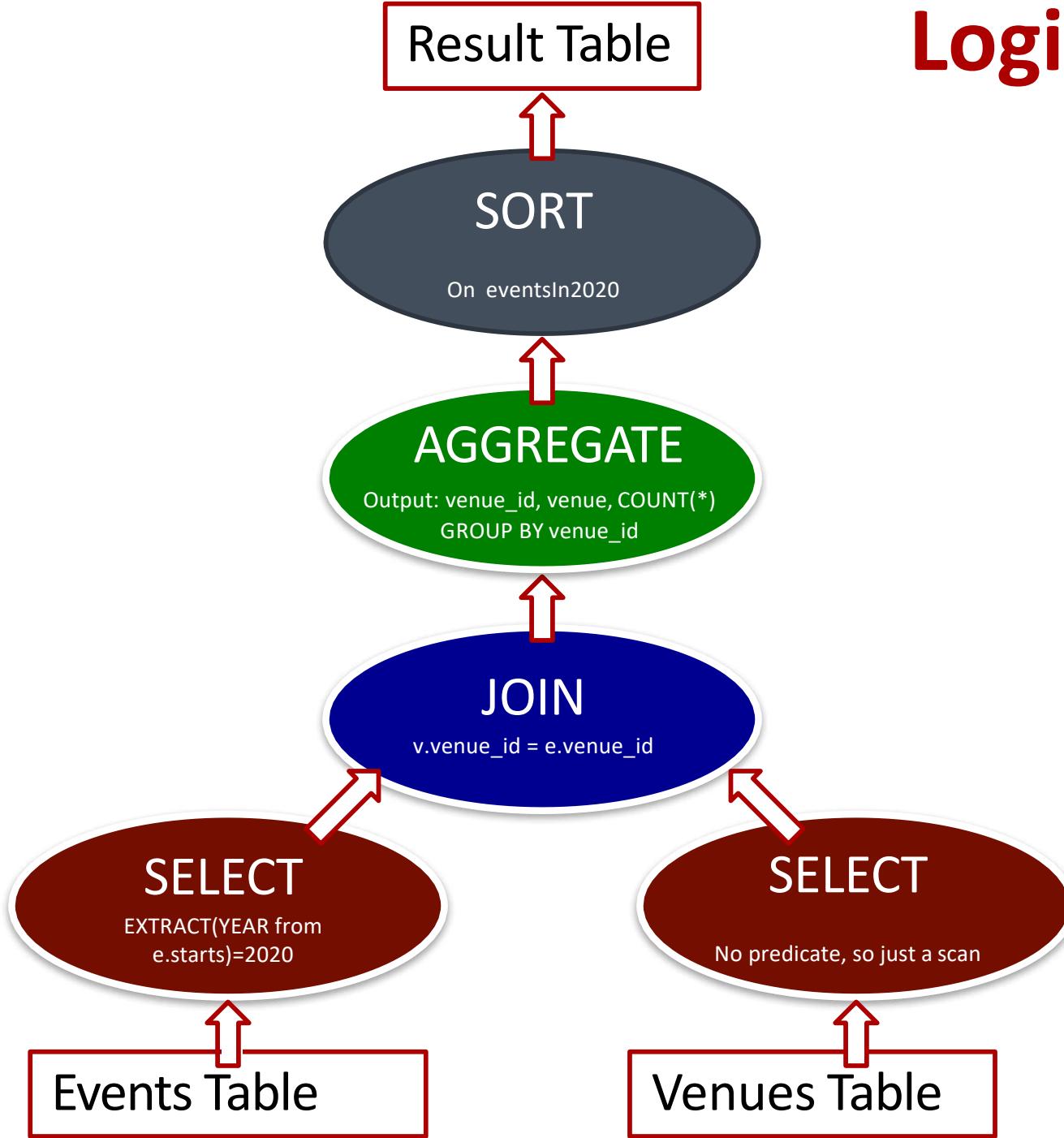
- Run the following query:

```
SELECT v.venue_id, v.name as venue, COUNT(*) as eventsIn2020
FROM venues v, events e
WHERE v.venue_id = e.venue_id
AND EXTRACT(YEAR from e.starts)=2020 -- select this year events
GROUP BY v.venue_id, v.name           -- group by venue
ORDER BY eventsIn2020 DESC          -- order by descending event count
```

Sample output table

venue_id	venue	eventsIn2020
72	Universidade Metodista de Piracicaba	8
56	University of Iowa	5
61	Timirjazev Moscow Academy of Agriculture	4
...		...

Logical Query Plan

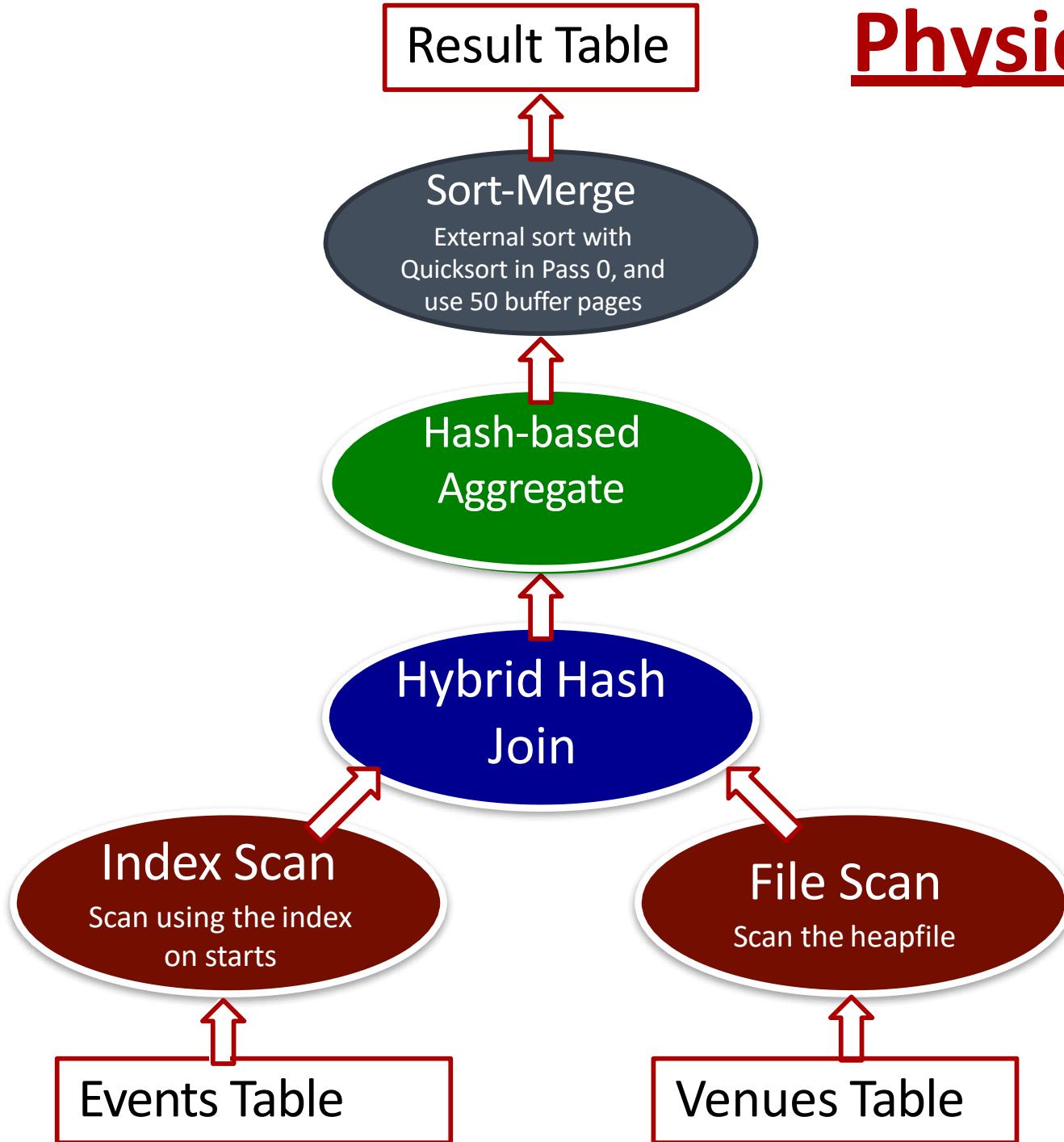


Here the ovals are logical operators. There are many different algorithms for each of these operators.

We go over some of these algorithms next.

You probably already know several sort algorithms: quicksort and merge sort.

Physical Query Plan



Here the ovals are **physical operators**.

Each physical operator specifies the exact algorithm/code that should be run, and parameters (if any) for that algorithm.

Select Operation

- Algorithms: File Scan or Index Scan
- **File Scan:** Disk I/O cost:
- **Index Scan:** (on some predicate). Disk I/O cost:
 - Hash: $O()$ can only use with equality predicates
 - B+-tree: $O() + X$
 - $X = \text{number of selected tuples}/\text{number of tuples per page}$
 - $X = 1$ per selected tuple with an unclustered index. To reduce the value of X , we could sort the rids and then fetch the tuples.

When to use a B+tree index

- Consider
 - A relation with 1M tuples
 - 100 tuples on a page
 - 500 (key, rid) pairs on a page

$$\begin{aligned}\# \text{ data pages} &= 1\text{M}/100 = 10\text{K pages} \\ \# \text{ leaf idx pgs} &= 1\text{M} / (500 * 0.67) \\ &\sim 3\text{K pages}\end{aligned}$$

	1% Selection	10% Selection
Clustered	$30 + 100$	$300 + 1000$
Non-Clustered	$30 + 10,000$	$300 + 100,000$
NC + Sort Rids	$30 + (\sim 10,000)$	$300 + (\sim 10,000)$

- ⇒ Choice of Index access plan, consider:
- 1. Index Selectivity 2. Clustering**
- ⇒ Similar consideration for hash based indices

When can we use an index

- Notion of “index matches a predicate”
- Basically mean when can an index be used to evaluate predicates in the query

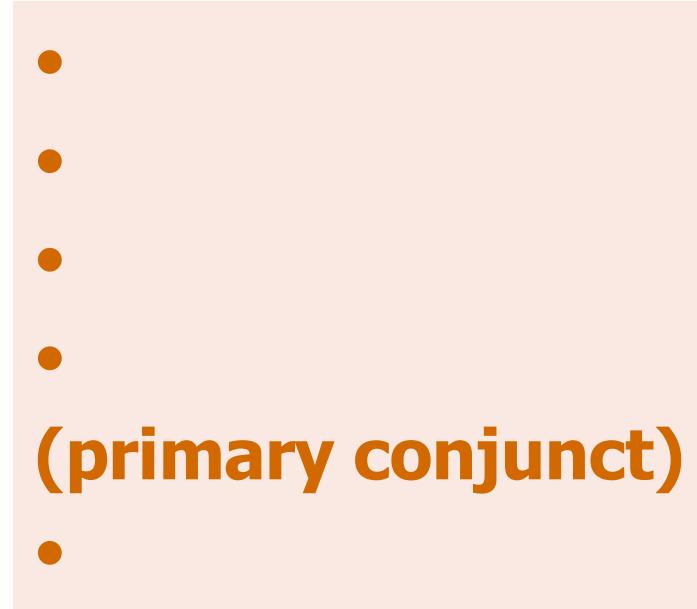
General Selection Conditions

- Index on (R.a, R.b)
 - Hash or tree-based
- Predicate:
 - $R.a > 10$
 - $R.b < 30$
 - $R.a = 10$ and $R.b = 30$
 - $R.a = 10$ or $R.b = 30$
- Predicate: $(p1 \text{ and } p2) \text{ or } p3$
- Convert to Conjunctive Normal Form(CNF)
 $(p1 \text{ or } p3) \text{ and } (p2 \text{ or } p3)$
- An index ***matches*** a predicate
 - Index can be used to evaluate the predicate

Index Matching

- B+-tree index on $\langle a, b, c \rangle$

- $a=5$ and $b=3$?
- $a > 5$ and $b < 3$
- $b=3$
- $a=7$ and $b=5$ and $c=4$
and $d>4$
- $a=7$ and $c=5$



- Index matches (part of) a predicate
 1. Conjunction of terms involving only attributes (**no disjunctions**)
 2. Hash: only equality operation, predicate has all index attributes.
 3. Tree: Attributes are a prefix of the search key, any ops.

Hash Idx

-
-
-
-
-

Index Matching

- A predicate could match more than 1 index
- Hash index on $\langle a, b \rangle$ and B+tree index on $\langle a, c \rangle$
 - $a=7$ and $b=5$ and $c=4$ Which index?
 - Option1: Use either (or a file scan!)
 - Check selectivity of the primary conjunct
 - Option2: Use both! Algorithm: Intersect rid sets.
 - Sort rids, retrieve rids in both sets.
 - Side-effect: tuples retrieved in the order on disk!

Selection

- Hash index on $\langle a \rangle$ and Hash index on $\langle b \rangle$
 - $a=7$ **or** $b>5$ Which index?
 - Neither! File scan required for $b>5$
- Hash index on $\langle a \rangle$ and B+-tree on $\langle b \rangle$
 - $a=7$ **or** $b>5$ Which index?
 - Option 1: Neither
 - Option 2: Use both! Fetch rids and union
 - Look at selectivities closely. Optimizer!
- Hash index on $\langle a \rangle$ and B+-tree on $\langle b \rangle$
 - $(a=7$ **or** $c>5)$ and $b > 5$ Which index?
 - Could use B+-tree (check selectivity)

Projection Algorithm

- Used to project the selected attributes.

Simple case: Example SELECT R.a, R.d.

- Algorithm: for each tuple, only output R.a, R.d

Harder case: DISTINCT clause

- Example: SELECT DISTINCT R.a, R.d
 - Remove attributes and eliminate duplicates
- Algorithms
 - Sorting: Sort on all the projection attributes
 - Pass 0: eliminate unwanted fields. Tuples in the sorted-runs may be smaller
 - Eliminate duplicates in the merge pass & in-memory sort
 - Hashing: Two phases
 - Partitioning
 - Duplicate elimination

Projection ...

- Sort-based approach
 - better handling of skew
 - result is sorted
 - I/O costs are comparable if Buffers used² > |R'|
- Index-only scan
 - Projection attributes subset of index attributes
 - Apply projection techniques to data entries (much smaller!)
- If an ordered (i.e., tree) index contains all projection attributes as *prefix* of search key:
 1. Retrieve index data entries in order
 2. Discard unwanted fields
 3. Compare adjacent entries to eliminate duplicates (if required)

Joins

- The focus here is on “equijoins”
- These are very common, given how we design the database schemas using primary and foreign keys
- Equijoins are used to bring the tuples back together
- Example:

```
SELECT v.venue_id, v.name as venue, COUNT(*) as eventsIn2020
FROM venues v, events e
WHERE v.venue_id = e.venue_id
AND EXTRACT(YEAR from e.starts)=2020
GROUP BY v.venue_id, v.name
ORDER BY eventsIn2020 DESC
```

We look at equijoin algorithms next

Page Nested Loops Join: PNL

1. For each page in the Venues table, p_v
2. For each page of Event, p_e
3. Join the tuples on page p_v with tuple in p_e
4. Output matching tuples (after applying any projection)

Let $|V|$ denote the # pages in the Venues table and
 $|E|$ denote the # pages in the Events table,

Then, the IO cost of the PNL Algorithm is:

$$|V| * |E|$$

Block Nested Loops Join: BNL

1. Scan the Venues table B-2 pages at a time
2. Insert the Venues tuples into an in-memory hash table on the join attribute
3. For each page of Events, p_e
 4. Probe the hash table with each tuple on the page p_e
 5. Output matching tuples (after applying any projection)

Let $|V|$ denote the # pages in the Venues table and
 $|E|$ denote the # pages in the Events table,

Then, the IO cost of the BNL Algorithm is: $O(|V| + |V| / (B-2) * |E|)$

B is number of memory buffers available

Index Nested Loops Join: INL

Can be used when there is an index on the join attribute on one of the tables

BNLJ vs. NLJ: Benefits of IO Aware

Example:

R: 500 pages

S: 1000 pages

100 tuples / page

We have 12 pages of memory ($B = 12$)

Ignoring OUT here...

NLJ: Cost = $500 + 500 * 1000 = 500 \text{ Thousand IOs} \approx \underline{1.4 \text{ hours}}$

BNLJ: Cost = $500 + \frac{500 * 1000}{10} = 50 \text{ Thousand IOs} \approx \underline{0.14 \text{ hours}}$

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

Sort-Merge Join: SMJ

1. Generate sorted runs for V (Pass 0)
2. Generate sorted runs for E (Pass 0)
3. Merge the sorted runs for V and E
4. While merging check for the join condition
5. Output matching tuples

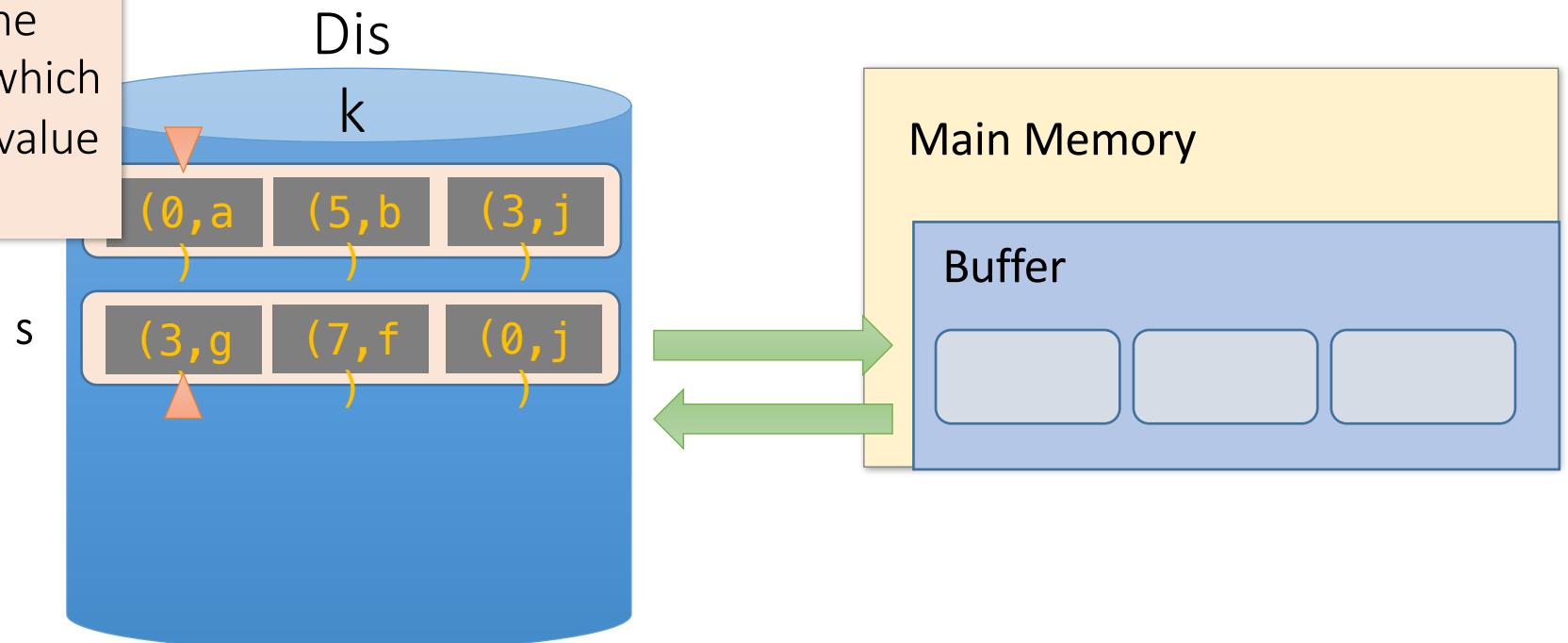
We have the following pros:

- There is no need to hold either table in memory (unlike the hash joins)
- Performance can be comparable with hash joins if the records are pre-sorted (which is made possible by the underlying storage engine with an index)

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

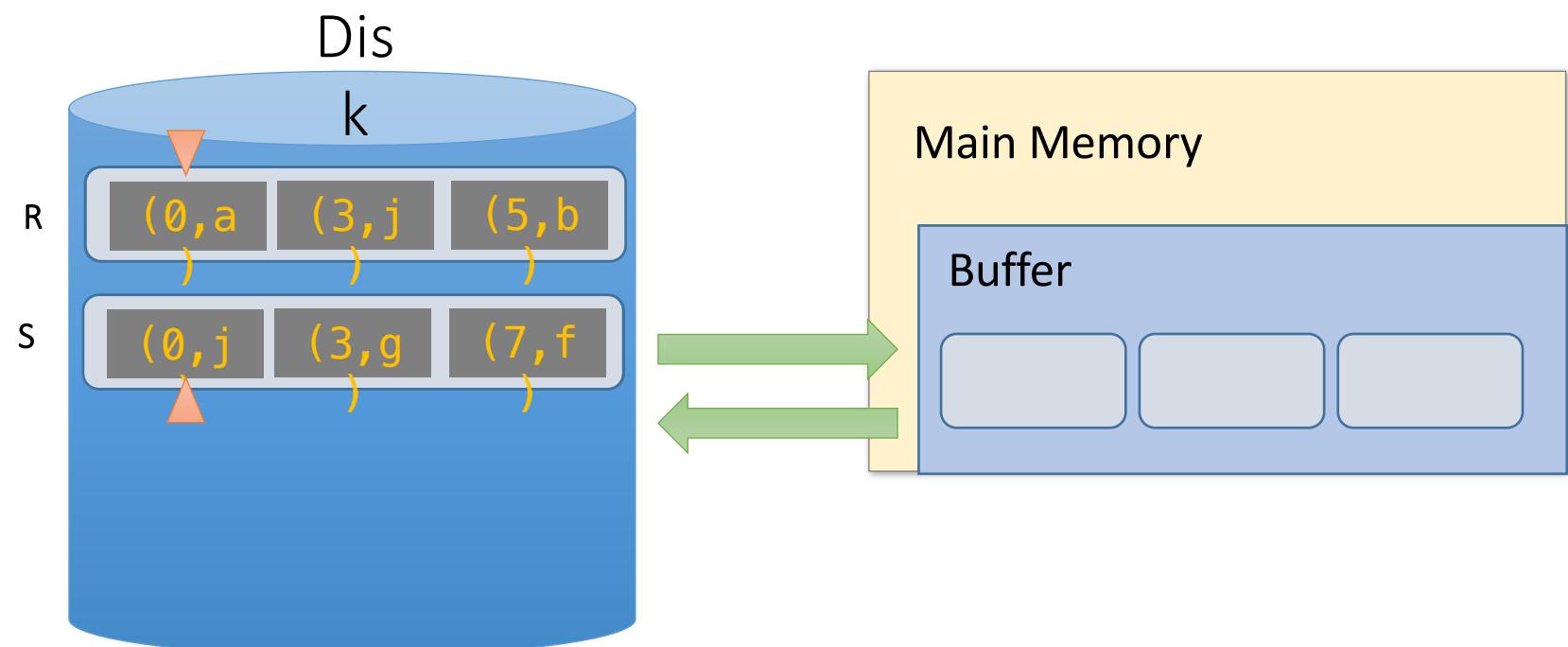
- For simplicity: Let each page be ***one tuple***, and let the first value be A

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



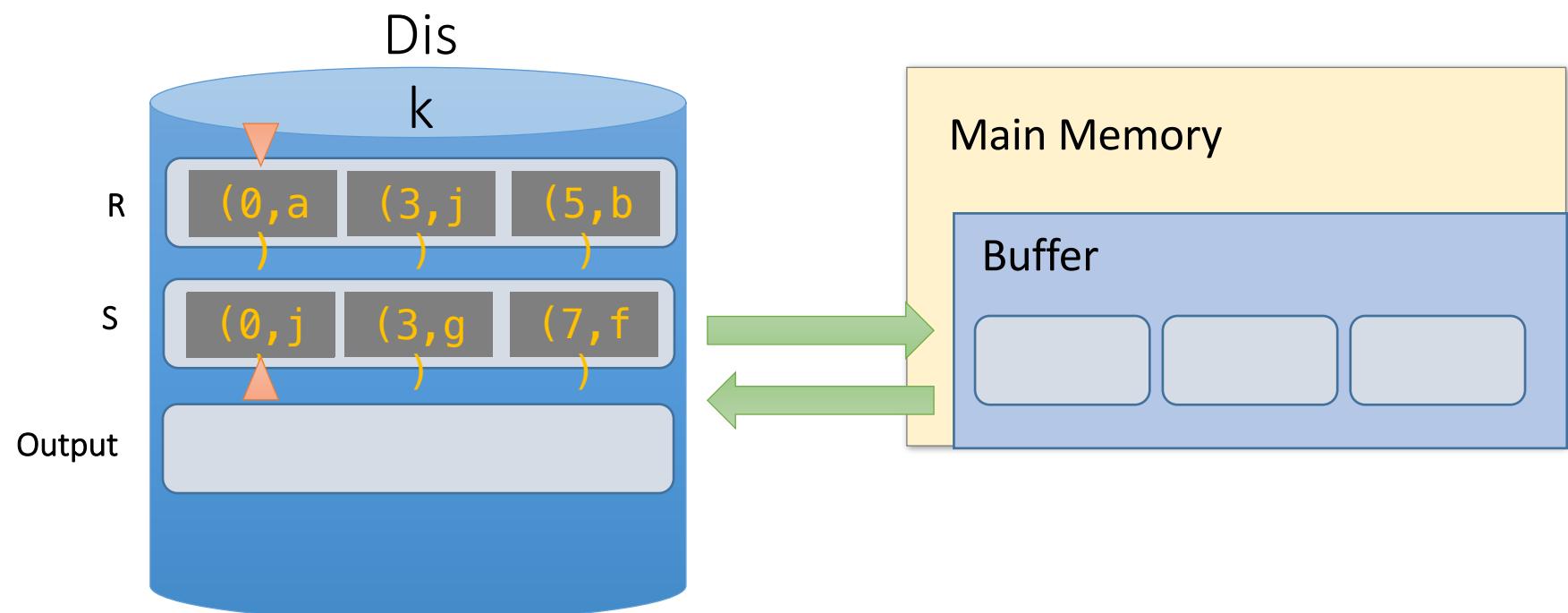
SMJ Example: $R \bowtie S$ on A with 3 page buffer

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



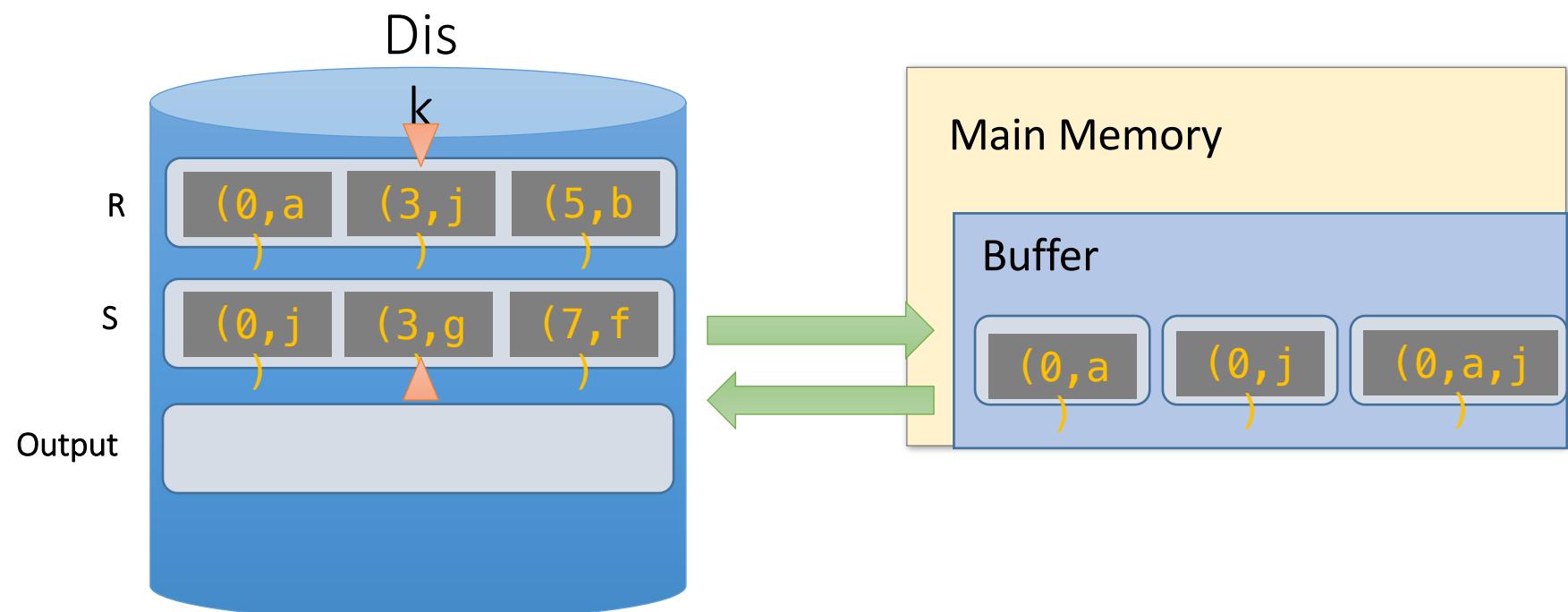
SMJ Example: $R \bowtie S$ on A with 3 page buffer

2. Scan and “merge” on join key!



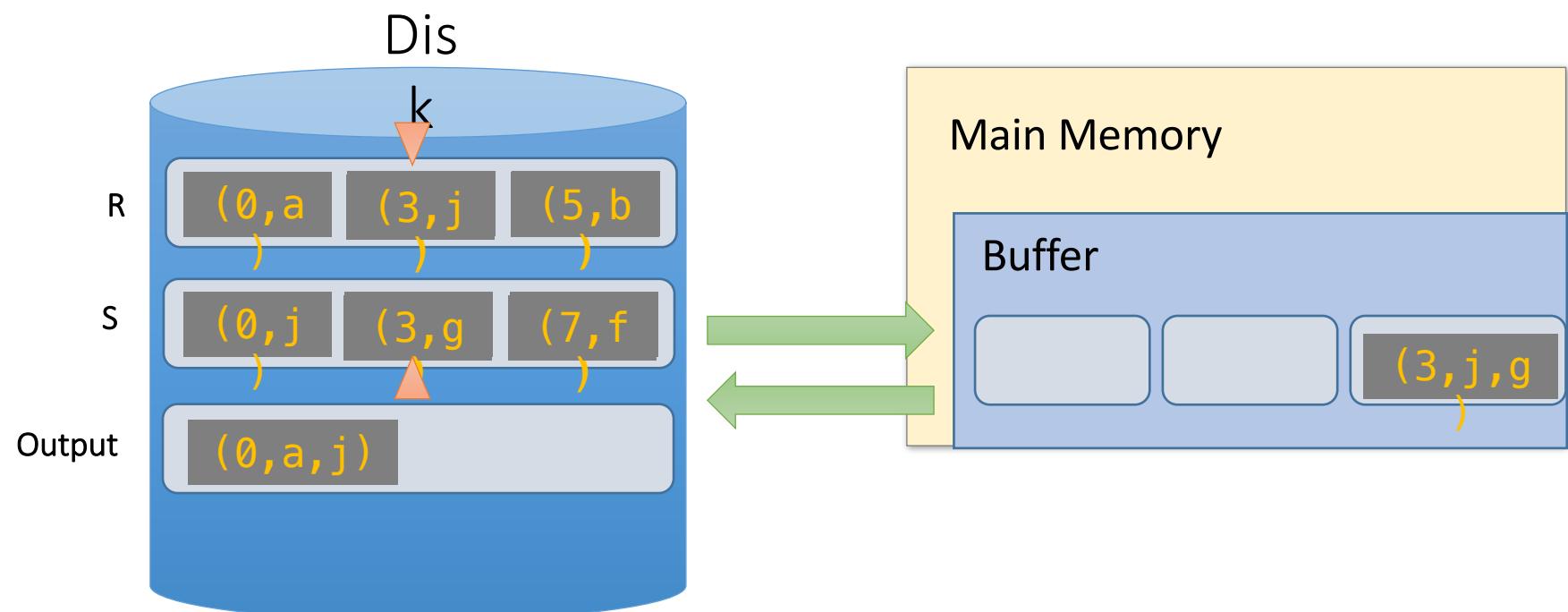
SMJ Example: $R \bowtie S$ on A with 3 page buffer

2. Scan and “merge” on join key!



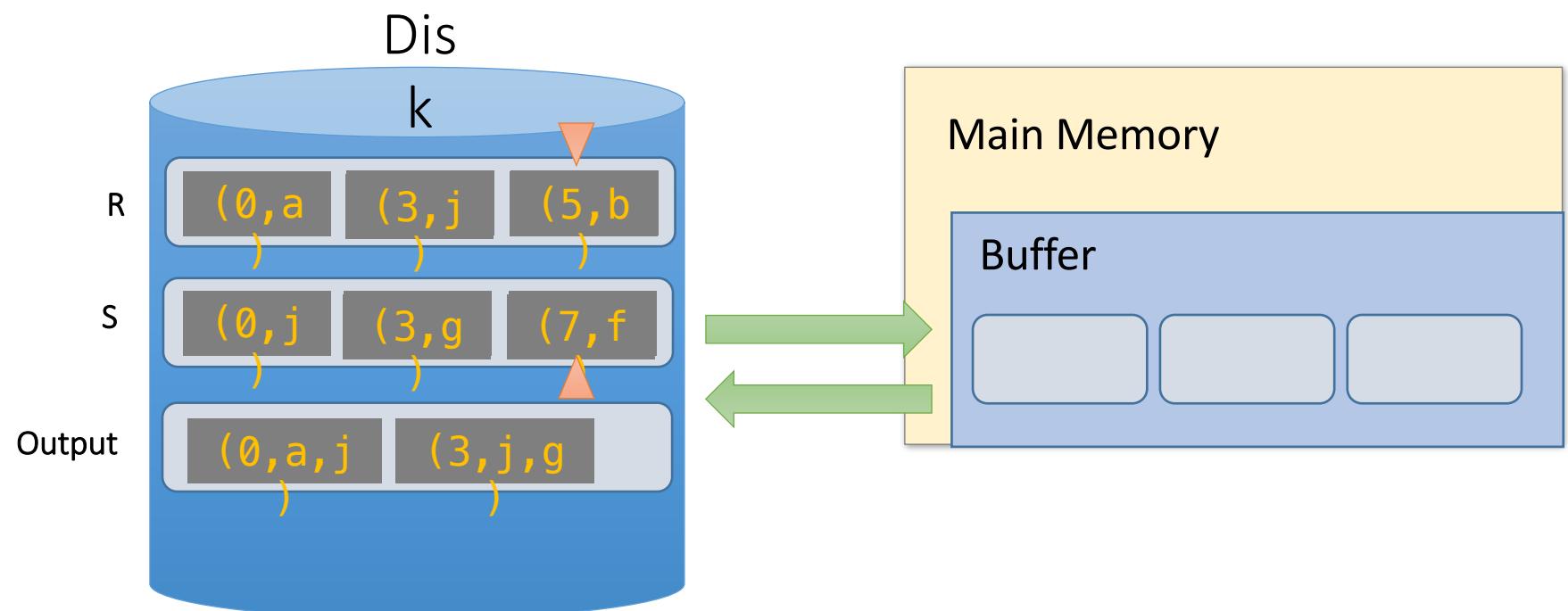
SMJ Example: $R \bowtie S$ on A with 3 page buffer

2. Scan and “merge” on join key!



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

2. Done!



SMJ: Total cost

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

(Simple) Hash Join Algorithm: HJ

1. Partition V into P partitions, using a hash function h_1 on the join key
 2. Partition E into P partitions, using a hash function h_1 on the join key
 3. Join each partition of V with the corresponding partition of E
 4. (using hashing as in BNL, so build a hash table on the V partition)
- // Note the hash function in the second part must be different from h_1

With B buffer pages, # partitions is $\sim B$ (for each V and E)

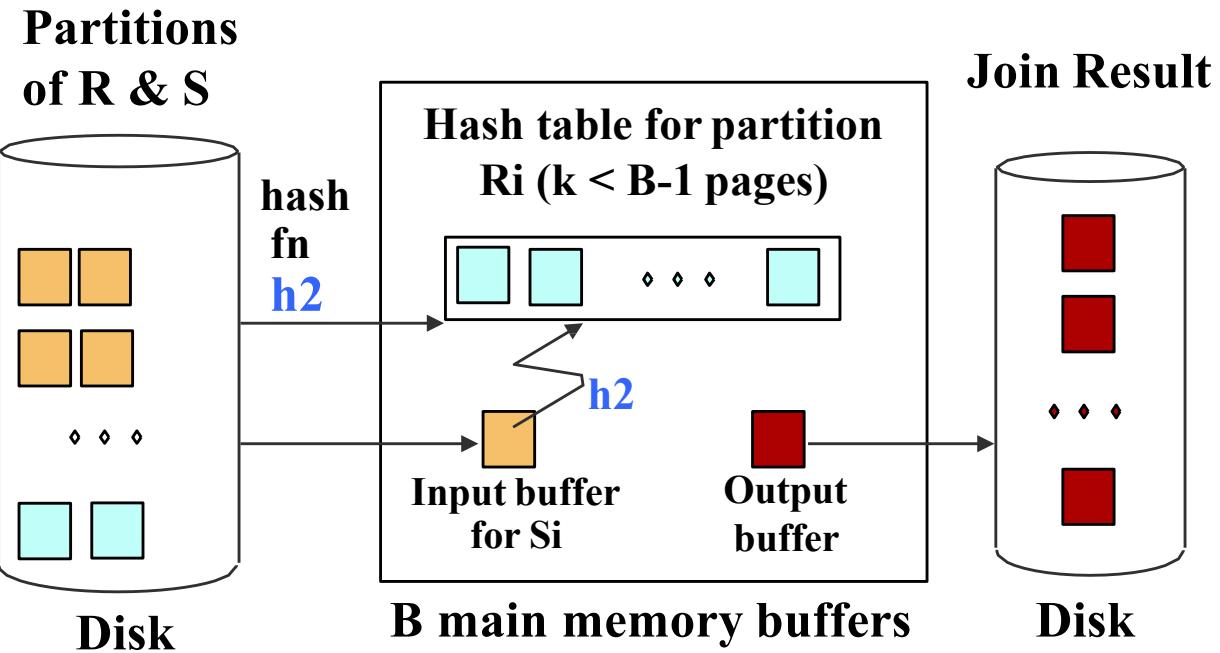
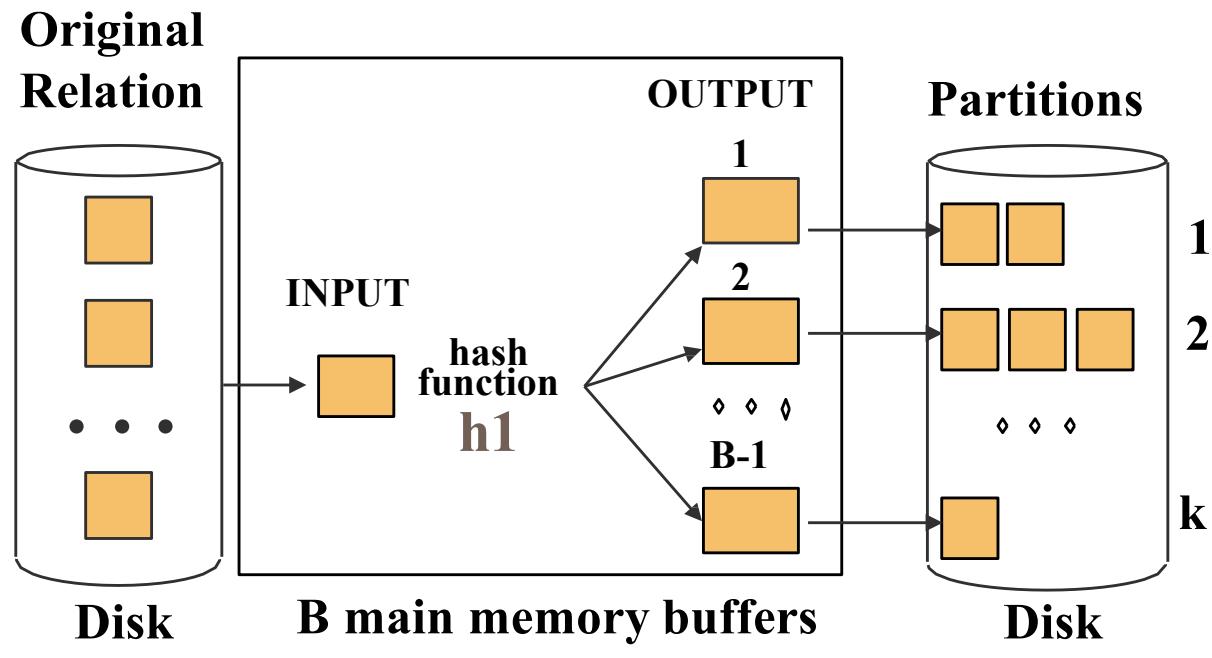
Each partition of V must fit in memory (with its hash table).

Assume that the hash table increases the space required by a factor of F.

Thus, the largest V that can be joined in two passes is constrained by:

$$B - 2 \geq \frac{|V|}{B} \Rightarrow \sim B^2 \geq |V|$$

Hash-Join



Hash Join versus Sort-Merge Join

- Need to join V with E, where $|E| > |V|$, using B buffer pages
- To do a two-pass join, SMJ needs
 - In this case the IO cost is: $3 * (|V| + |E|)$
- To do a two-pass join, HJ needs
 - In this case the IO cost is: $3 * (|V| + |E|)$

So HJ can sort two relations with fewer buffer pages!

$$\text{SMJ: } B^2 > \max\{|R|, |S|\}$$

$$\text{HJ: } B^2 > \min\{|R|, |S|\}$$

General Join Conditions

- Equalities over several attributes
 - g. , $R.sid=S.sid$ AND $R.rname=S.sname$:
 - Index NL
 - index on $\langle sid, sname \rangle$
 - index on sid or $sname$.
 - SM and Hash, sort/hash on combination of join attrs
- Inequality conditions (e.g., $R.rname < S.sname$):
 - For Index NL, need (clustered!) B+ tree index.
 - Large # index matches
 - SM and Hash not applicable
 - Block NL likely to be the winner

Set Operations

- \cap and Δ special cases of join
- \cup and $-$ similar; we'll do \cup .
 - Duplicate elimination
- Sorting:
 - Sort both relations (on all attributes).
 - Merge sorted relations eliminating duplicates.
 - *Alternative:* Merge sorted runs from *both* relations.
- Hashing:
 - Partition R and S
 - Build hash table for R_i .
 - Probe with tuples in S_i , add to table if not a duplicate

Aggregates

- Sorting
 - Sort on group by attributes (if any)
 - Scan sorted tuples, computing running aggregate
 - Max: Max
 - Average: Sum, Count
 - If the group by attribute changes, output aggregate result
- Cost: sorting cost

Aggregates

- Hashing
 - Hash on group by attributes (if any)
 - Hash entry: group attributes + running aggregate
 - Scan tuples, probe hash table, update hash entry
 - Scan hash table, and output each hash entry
- Cost: Scan relation!
- What if we have a large # groups?

Aggregates

- Index
 - Without Grouping
 - Can use B+tree on aggregate attribute(s)
 - Where clause?
 - With grouping
 - B+tree on all attributes in SELECT, WHERE and GROUP BY clauses
 - Index-only scan
 - If group-by attributes prefix of search key
=> data entries/tuples retrieved in group-by order
 - Else => get data entries and then use a sort or hash aggregate algorithm

For today

- Write a query to display city_name and country_code for Iowa City.
- Obtain the query plan for this query (and copy it in a text file).
- Create a hash index on city_name
- Get a new query plan for the same query and compare it with the previous one (store it in the same file).
- Create a btree index on city_name
- Compare the new query plan with the previous two (store it in the file).
- Drop the unused index(es) on city_name
- Submit the query plans to ICON.

Functions, Triggers, Rules

Procedures and Functions

- A procedure (or function) is a module performing one or more actions
- <https://www.postgresql.org/docs/10/plpgsql-structure.html>
- Pros:
 - Server performs heavy-lifting
 - Minimize traffic between client-server (no need to send data to the client)
 - Extend the functionality of the database
- Cons:
 - Software Architecture/vendor dependency
 - Code is in the database

Functions

- The syntax for creating a function:

```
CREATE [OR REPLACE] FUNCTION function_name  
    (parameter list)  
    RETURN datatype  
AS  
BEGIN  
    <body>  
    RETURN (return_value);  
END;
```

- To call it:

```
SELECT function_name(parameters,...)
```

Example

```
CREATE OR REPLACE FUNCTION add_event(
    title text, starts timestamp, ends timestamp,
    venue text, postal varchar(9), country char(2))
RETURNS boolean AS $$

DECLARE
    did_insert boolean := false;
    found_count integer;
    the_venue_id integer;

BEGIN
    SELECT venue_id INTO the_venue_id
    FROM venues v
    WHERE v.postal_code=postal AND
    v.country_code=country AND v.name ILIKE venue
    LIMIT 1;
    IF the_venue_id IS NULL THEN
        INSERT INTO venues (name, postal_code,
        country_code)
        VALUES (venue, postal, country) RETURNING
        venue_id INTO the_venue_id;
    END IF;
    RETURN did_insert;
END;

$$ LANGUAGE plpgsql;
```

```
did_insert := true;
END IF;
-- Note: this is a notice, not an error as in
some programming languages
RAISE NOTICE 'Venue found %', the_venue_id;
INSERT INTO events (title, starts, ends,
venue_id)
VALUES (title, starts, ends, the_venue_id);
RETURN did_insert;
END;
$$ LANGUAGE plpgsql;
```

- To execute it:

```
SELECT add_event('Modern Marvels', '2020-12-10 14:00',
'2020-12-10 17:00', 'Seamans Center', '52242', 'us');
```

Triggers

- Programs executed (fired) automatically when a given SQL operation (like INSERT, UPDATE or DELETE) affects the table associated with the trigger.
- Unlike a procedure, or a function, which must be invoked explicitly, database triggers are invoked implicitly.
- Database triggers can be used to perform any of the following:
 - Audit data modification
 - Log events transparently
 - Enforce complex business rules
 - Derive column values automatically
 - Implement complex security authorizations
 - Maintain replicate tables

Triggers

- To use a trigger, we need to first define a trigger procedure, then create the trigger which will execute the trigger procedure
- A trigger procedure is created with the CREATE FUNCTION command, declaring it as a function with no arguments and a return type of trigger.
- The function must be declared with no arguments even if it expects to receive arguments specified in CREATE TRIGGER — trigger arguments are passed via TG_ARGV

Triggers

- Let's create a logs event table (to make sure no one changes an event and tries to deny it later)

```
CREATE TABLE logs (
    event_id integer,
    old_title varchar(255),
    old_starts timestamp,
    old_ends timestamp,
    logged_at timestamp DEFAULT current_timestamp
);
```

Trigger example

```
CREATE OR REPLACE FUNCTION log_event() RETURNS trigger AS $$  
DECLARE  
BEGIN  
    INSERT INTO logs (event_id, old_title, old_starts, old_ends)  
        VALUES (OLD.event_id, OLD.title, OLD.starts, OLD.ends);  
    RAISE NOTICE 'Someone just changed event #%', OLD.event_id;  
    RETURN NEW;  
END;  
$$ LANGUAGE plpgsql;
```

Trigger example (cont)

```
CREATE TRIGGER log_events  
    AFTER UPDATE ON events  
    FOR EACH ROW EXECUTE PROCEDURE log_event();
```

- Now let's update the Modern Marvels event

```
UPDATE events  
SET ends='2020-12-10 18:00:00'  
WHERE title='Modern Marvels';
```

- And make sure the old event is logged:

```
SELECT event_id, old_title, old_ends, logged_at  
FROM logs;
```

Views/Materialized views

```
CREATE VIEW holidays AS
```

```
    SELECT event_id AS holiday_id, title AS name, starts AS date  
    FROM events  
    WHERE title LIKE '%Day%' AND venue_id IS NULL;
```

Can be queried as any other table:

```
SELECT name, to_char(date, 'Month DD, YYYY') AS date  
FROM holidays  
WHERE date <= '2018-04-01';
```

- Materialized view
 - CREATE MATERIALIZED VIEW mymatview;
- To refresh it
 - REFRESH MATERIALIZED VIEW mymatview;
- It is a table that you can indexed

Updatable views

- Simple views are automatically updatable: the system will allow INSERT, UPDATE and DELETE statements to be used on the view in the same way as on a regular table.
- A view is automatically updatable if it satisfies all of the following conditions:
 - The view must have exactly one entry in its FROM list, which must be a table or another updatable view.
 - The view definition must not contain WITH, DISTINCT, GROUP BY, HAVING, LIMIT, or OFFSET clauses at the top level.
 - The view definition must not contain set operations (UNION, INTERSECT or EXCEPT) at the top level.
 - The view's select list must not contain any aggregates, window functions or set-returning functions.

Update events through the holidays view?

Alter the events table to have an array of associated colors

```
ALTER TABLE events
ADD colors text ARRAY;
```

Update the VIEW query to contain the colors array.

```
CREATE OR REPLACE VIEW holidays AS
SELECT event_id AS holiday_id, title AS name,
starts AS date, colors
FROM events
WHERE title LIKE '%Day%' AND venue_id IS NULL;
```

Now let's try to update the colors for Christmas

```
UPDATE holidays SET colors = '{"red", "green"}'
where name = 'Christmas Day';
```

Update Rules

Rules defined on INSERT, UPDATE, and DELETE

```
CREATE [ OR REPLACE ] RULE name AS ON event TO table
[ WHERE condition ] DO [ ALSO | INSTEAD ] { NOTHING |
command | ( command ; command ... ) }
```

CREATE RULE command allows:

- To have no action.
- Can have multiple actions.
- Can be INSTEAD or ALSO (the default).
- The pseudorelations NEW and OLD become useful.
 - NEW contains the values we're setting
 - OLD contains the values we query by

Rule Example (cont)

```
CREATE RULE update_holidays AS ON UPDATE TO holidays DO INSTEAD  
UPDATE events  
SET title = NEW.name,  
starts = NEW.date,  
colors = NEW.colors  
WHERE title = OLD.name;
```

Now try inserting 'New Years Day' on 2020-12-31

```
CREATE RULE insert_holidays AS ON INSERT TO holidays DO INSTEAD  
INSERT INTO ...
```

For today

- Create a rule that captures DELETEs on venues and instead sets the active flag (you added in a previous class assignment) to FALSE.
- Try:
- Delete from venues
where name='University of South Carolina';

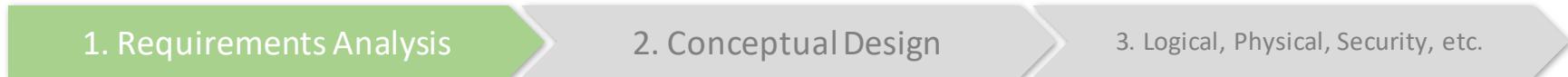
ER DIAGRAMS AND NORMAL FORMS

Some figures taken from the book “Concepts of Database Management”

Database Design

- **Database design: Why do we need it?**
 - Agree on structure of the database before deciding on a particular implementation
- **Consider issues such as:**
 - What entities to model
 - How entities are related
 - What constraints exist in the domain
 - How to achieve good designs
- **Several formalisms exist**
 - We discuss one flavor of E/R diagrams

Database Design Process



1. Requirements analysis

- What is going to be stored?
- How is it going to be used?
- What are we going to do with the data?
- Who should access the data?

Technical and non-technical people are involved

Database Design Process

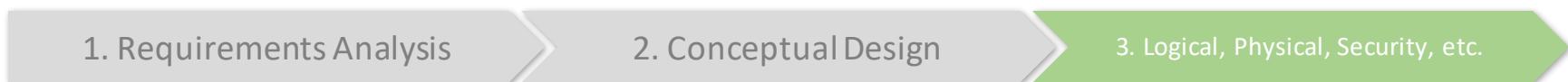


2. Conceptual Design

- A high-level description of the database
- Sufficiently precise that technical people can understand it
- But, not so precise that non-technical people can't participate

This is where E/R fits in.

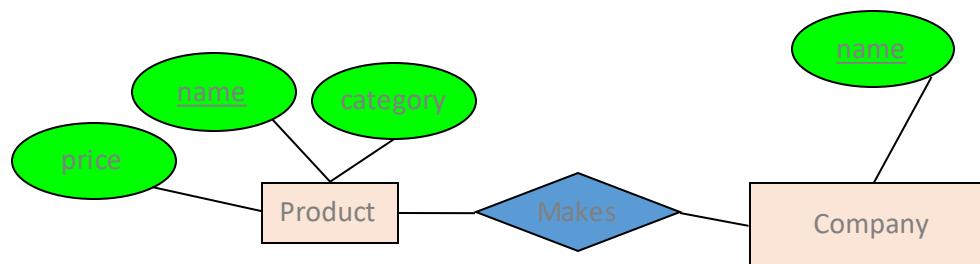
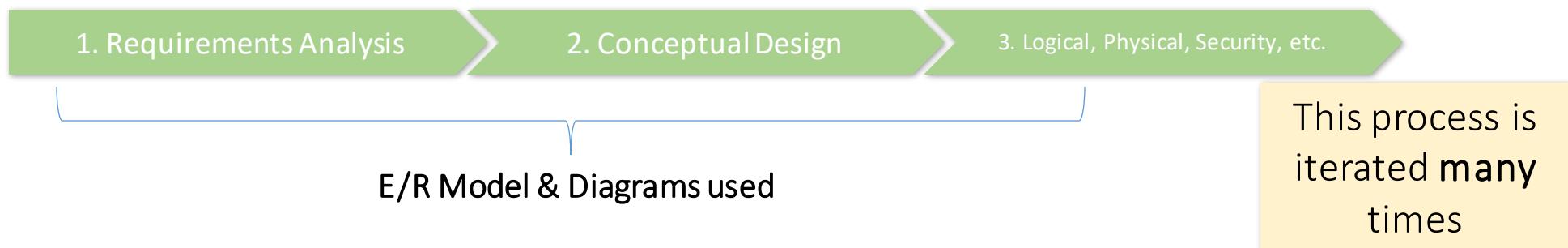
Database Design Process



3. More:

- Logical Database Design
- Physical Database Design
- Security Design

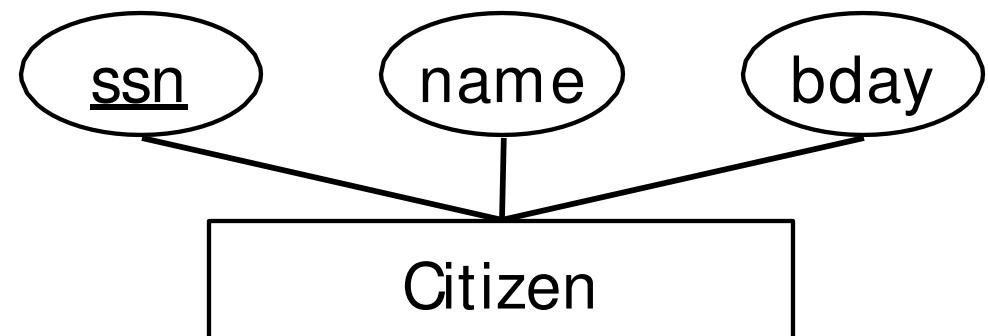
Database Design Process



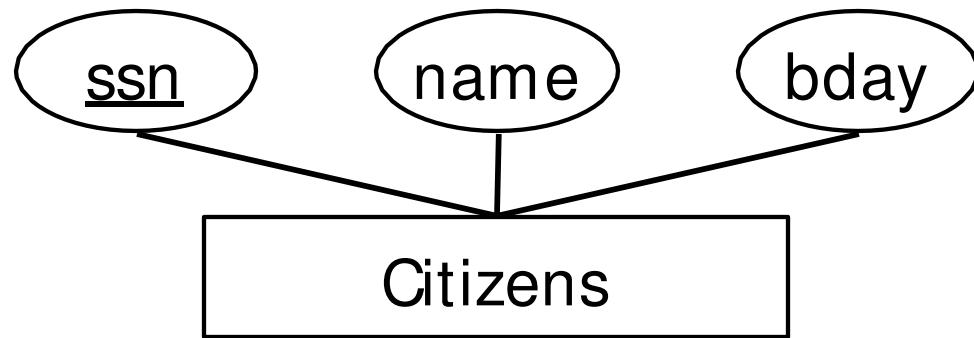
E/R is a *visual syntax* for DB design which is *precise enough* for technical points, but *abstracted enough* for non-technical people

ER Model Basics

- *Entity*: Distinguishable real-world object
 - Described by a set of *attributes*, Each attribute has a *domain*
- *Entity Set*: A collection of similar entities. E.g., all citizens.
 - All entities in an entity set have the same set of attributes.
Key : minimal set of attributes whose values uniquely identify an entity in an entity set
 - Primary key
 - Candidate key
- Pictorially ...
- *Relationship* : Association among two or more **entities**

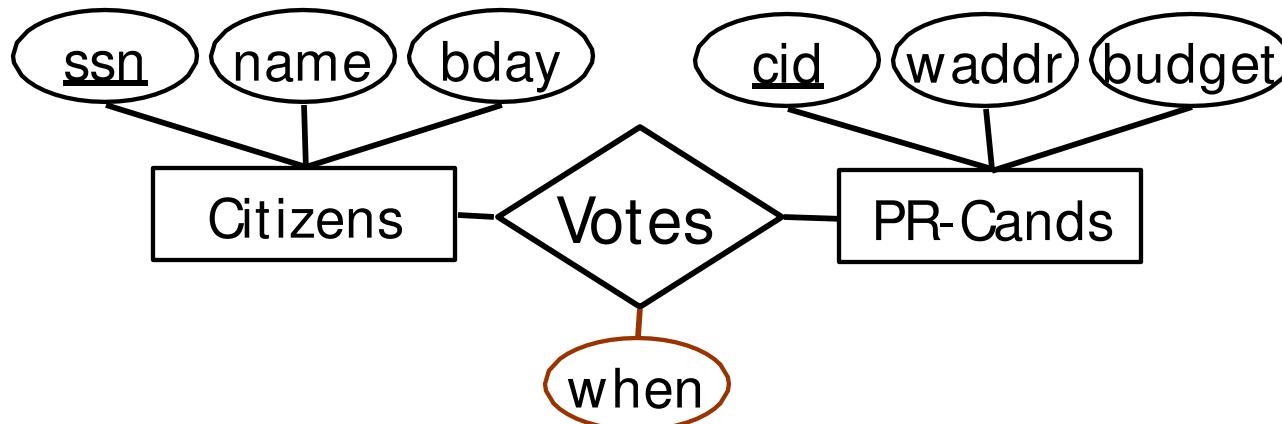


Logical DB Design: ER to Relational



```
CREATE TABLE Citizens  
  (ssn  CHAR(11),  
   name CHAR(20),  
   bday DATE,  
   PRIMARY KEY (ssn))
```

Relationship Sets to Tables



```
CREATE TABLE Votes(  
    ssn    CHAR(11),  
    cid    INTEGER,  
    when   DATE,  
PRIMARY KEY (ssn, cid),  
FOREIGN KEY (ssn) REFERENCES Citizens,  
FOREIGN KEY (cid) REFERENCES PR-Cands)
```

Relationship set -> Table

Attributes:

- Participating entity set primary keys
 - Foreign key
 - Superkey
- Descriptive attributes

Can ssn have
a null value?

Can generalize to
n-ary relationships

Another E/R diagram format

Department (DepartmentNum, Name, Location)

Employee (EmployeeNum, LastName, FirstName, Street, City,
State, PostalCode, WageRate, SocSecNum, DepartmentNum)

AK SocSecNum

SK LastName, FirstName

FK DepartmentNum → Department

The E-R diagram for the preceding database design appears in Figure 6-2.

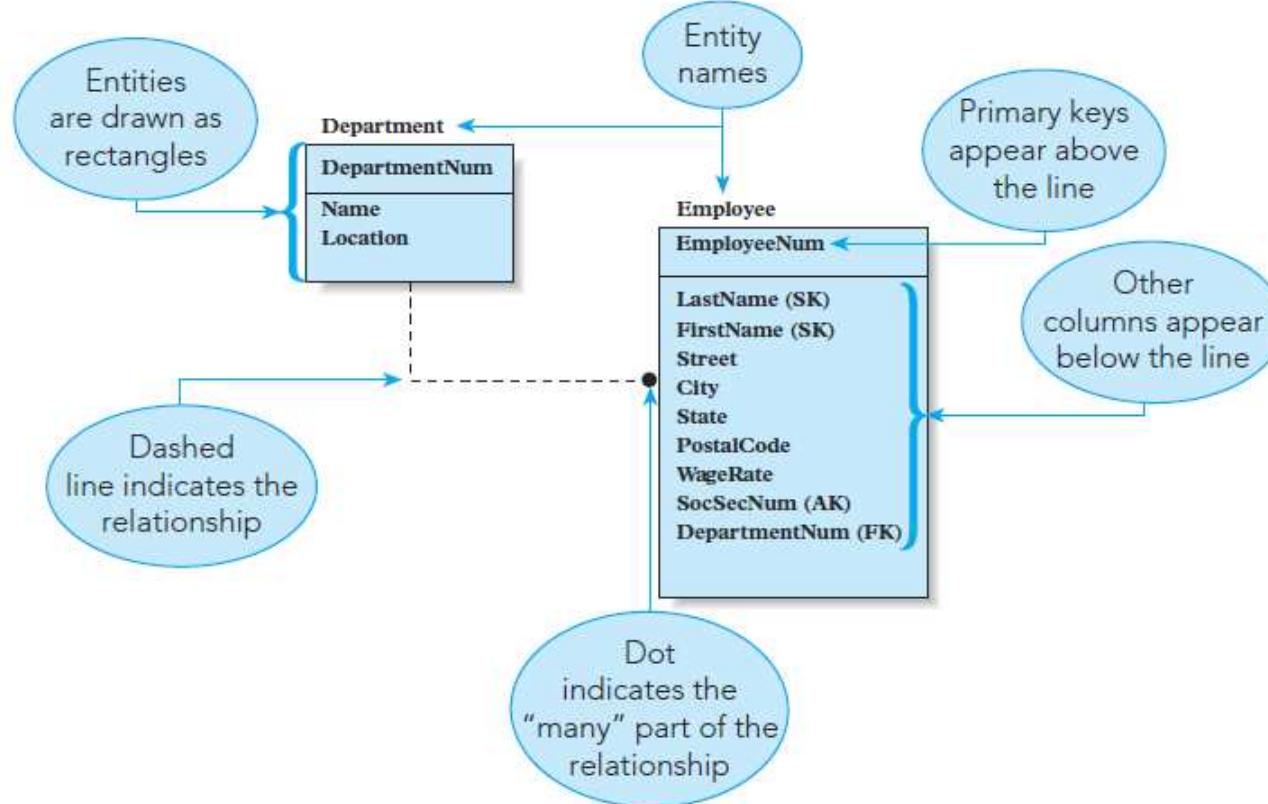


FIGURE 6-2 E-R diagram

Design theory

- Design theory is about how to represent your data to avoid ***anomalies***.

What is a good relational schema? How can we improve it?

- e.g.: Suppliers (name, item, desc, addr, price)

Redundancy Problems:

1. A supplier supplies two items: **Redundant Storage**
2. Change address of a supplier: **Update Anomaly**
3. Insert a supplier: **Insertion Anomaly**
 - What if the supplier does not supply any items (nulls?)
 - Record desc for an item that is not supplied by any supplier
4. Delete the only supplier tuple: **Delete Anomaly**
 - Use nulls?
 - Delete the last item tuple. Can't make name null. Why?

Dealing with Redundancy

- Identify “bad” schemas
 - functional dependencies
- Main refinement technique: decomposition
 - replacing larger relation with smaller ones
- Decomposition should be used judiciously:
 - Is there a reason to decompose a relation?
 - Normal forms: guarantees against (some) redundancy
 - Does decomposition cause any problems?
 - Lossless join
 - Dependency preservation
 - Performance (must join decomposed relations)

Functional Dependency (FD)

Def: Let A,B be sets of attributes

We write $A \rightarrow B$ or say A ***functionally determines*** B if, for any tuples t_1 and t_2 :

$$t_1[A] = t_2[A] \text{ implies } t_1[B] = t_2[B]$$

and we call $A \rightarrow B$ a **functional dependency**

*A->B means that
“whenever two tuples agree on A then they agree on B.”*

Consider, $(X,Y) \rightarrow Z$

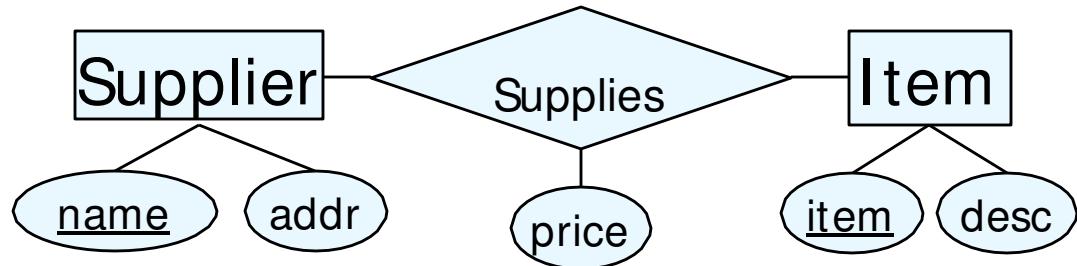
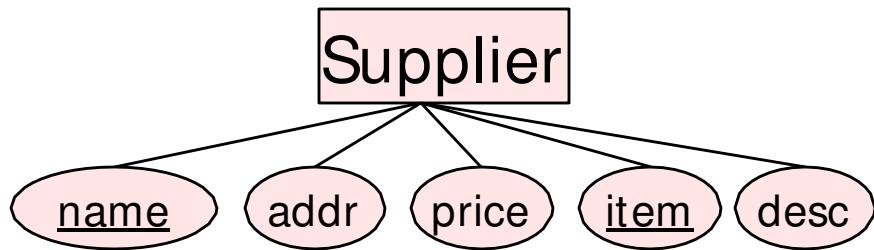
X	Y	Z	K
1	1	11	A
1	2	12	A
2	2	22	A
2	2	22	B

- An FD is a statement about all allowable relations.
 - Based only on application semantics, can't deduce from instances
 - Can simply check if an instance violates FD (and other ICs)

FDs for Relational Schema Design

- High-level idea: **why do we care about FDs?**
 1. Start with some relational *schema*
 2. Model its *functional dependencies (FDs)*
 3. Use these to *design a better schema*
 1. One which minimizes the possibility of anomalies

Example: Constraints on Entity Set



- $S(\underline{\text{name}}, \underline{\text{item}}, \text{desc}, \text{addr}, \text{price})$
- FD: $\{\text{n}, \text{i}\} \rightarrow \{\text{n}, \text{i}, \text{d}, \text{a}, \text{p}\}$
- FD: $\{\text{n}\} \rightarrow \{\text{a}\}$
- FD: $\{\text{i}\} \rightarrow \{\text{d}\}$
- Decompose to: NA, ID, INP

- $\text{Spl}(\underline{\text{name}}, \underline{\text{item}}, \text{price})$
 - FD: $\{\text{n}, \text{i}\} \rightarrow \{\text{n}, \text{i}, \text{p}\}$
- $\text{Sup}(\text{name}, \text{addr})$
 - FD: $\{\text{n}\} \rightarrow \{\text{n}, \text{a}\}$
- $\text{Item}(\text{item}, \text{desc})$
 - FD: $\{\text{i}\} \rightarrow \{\text{i}, \text{d}\}$

ER design is subjective and can have many E + Rs
FDs: sanity checks + deeper understanding of schema

Same situation could happen with a relationship set

Finding functional dependencies

Rep

RepNum	LastName	FirstName	Street	City	State	PostalCode	Commission	PayClass	Rate
15	Campos	Rafael	724 Vinca Dr.	Grove	CA	90092	\$23,457.50	1	0.06
30	Gradey	Megan	632 Liatris St.	Fultton	CA	90085	\$41,317.00	2	0.08
45	Tian	Hui	1785 Tyler Ave.	Northfield	CA	90098	\$27,789.25	1	0.06
60	Sefton	Janet	267 Oakley St.	Congaree	CA	90097	\$0.00	1	0.06

- RepNum \rightarrow LastName, FirstName ?
- LastName \rightarrow Street, City, State?
- PayClass \rightarrow Rate?

Find all functional dependencies

- StudentNum (student number)
- StudentLast (student last name)
- StudentFirst (student first name)
- HighSchoolNum (number of the high school from which the student graduated)
- HighSchoolName (name of the high school from which the student graduated)
- AdvisorNum (number of the student's advisor)
- AdvisorLast (last name of the student's advisor)
- AdvisorFirst (first name of the student's advisor)

Student numbers, high school numbers, and advisor numbers are unique; no two students have the same number, no two high schools have the same number, and no two advisors have the same number. Use this information to determine the functional dependencies in the Student relation.

Inferring FD

- $\text{ename} \rightarrow \text{ejob}$, $\text{ejob} \rightarrow \text{esal}$; $\Rightarrow \text{ename} \rightarrow \text{esal}$
- Armstrong's Axioms (X, Y, Z are sets of attributes):
 - Reflexivity: If $Y \subseteq X$, then $X \rightarrow Y$
 - Augmentation: If $X \rightarrow Y$, then $XZ \rightarrow YZ$ for any Z
 - Transitivity: If $X \rightarrow Y$ and $Y \rightarrow Z$, then $X \rightarrow Z$
- Additional rules (derivable):
 - Union: If $X \rightarrow Y$ and $X \rightarrow Z$, then $X \rightarrow YZ$
 - Decomposition: If $X \rightarrow YZ$, then $X \rightarrow Y$ and $X \rightarrow Z$
- Set of all FD = closure of F , denoted as F^+
- AA sound: only generates FD in F^+
- AA complete: repeated application generates all FD in F^+

Keys and Superkeys

A superkey is a set of attributes A_1, \dots, A_n s.t.
for *any other* attribute B in R ,
we have $\{A_1, \dots, A_n\} \rightarrow B$

I.e. all attributes are
functionally determined by a
superkey

A key is a *minimal* superkey

Meaning that no subset
of a key is also a superkey

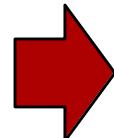
Decomposition

- Replace a relation with two or more relations
- Problems with decomposition
 1. Some queries become more expensive. (more joins)
 2. **Lossless Join:** Can we reconstruct the original relation from instances of the decomposed relations?
 3. **Dependency Preservation:** Checking some dependencies may require joining the instances of the decomposed relations.

Lossless Join Decompositions

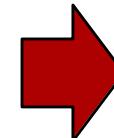
- Relation R, FDs F: Decomposed to X, Y
- Lossless-Join decomposition if:
 $\Pi_X(r) \bowtie \Pi_Y(r) = r$ for every instance r of R
- Note, $r \subseteq \Pi_X(r) \bowtie \Pi_Y(r)$ is always true,
not vice versa, unless the join is lossless
- Can generalize to three more relations

A	B	C
1	2	3
4	5	6
7	2	8



A	B
1	2
4	5
7	2

B	C
2	3
5	6
2	8



A	B	C
1	2	3
4	5	6
7	2	8
1	2	8
7	2	3

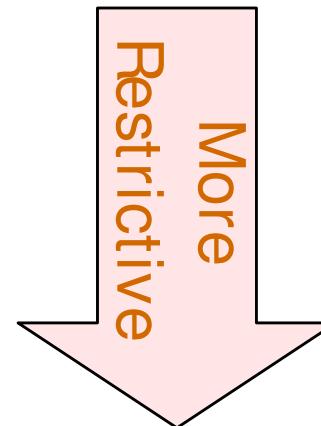
Lossless Join ...

- Relation R, FDs F: Decomposed to X, Y
 - Test: lossless-join w.r.t. F if and only if the closure of F contains:
 - $X \cap Y \rightarrow X$, or
 - $X \cap Y \rightarrow Y$i.e. attributes common to X and Y contain a key for either X or Y
 - Also, given FD: $X \rightarrow Y$ and $X \cap Y = \emptyset$, the decomposition into R-Y and XY is lossless
 - **X is a key in XY, and appears in both**

Normal Forms

- Is any refinement is needed!
- Normal Forms: guarantees that certain kinds of problems won't occur

- 1 NF : Atomic values
- 2 NF : Historical
- 3 NF : ...
- BCNF : Boyce-Codd Normal Form



- Role of FDs in detecting redundancy:
 - Relation R with 3 attributes, ABC.
 - No ICs (FDs) hold \Rightarrow no redundancy.
 - $A \rightarrow B \Rightarrow$ 2 or more tuples with the same A value, redundantly have the same B value!

First Normal Form (1NF)

- A relation (table) is in first normal form (1NF) when it does not contain repeating groups (multiple entries for a single record)

Orders

OrderNum	OrderDate	ItemNum	NumOrdered
51608	10/12/2015	CD33	5
51610	10/12/2015	KL78	25
		TR40	10
51613	10/13/2015	DL51	5
51614	10/13/2015	FD11	1
51617	10/15/2015	NL89	4
		TW35	3
51619		FD11	2
51623	10/15/2015	DR67	5
		FH24	12
		KD34	10
51625	10/16/2013	MT03	8

Unnormalized

Orders (OrderNum, OrderDate, (ItemNum, NumOrdered))

Orders

OrderNum	OrderDate	ItemNum	NumOrdered
51608	10/12/2015	CD33	5
51610	10/12/2015	KL78	25
51610	10/12/2015	TR40	10
51613	10/13/2015	DL51	5
51614	10/13/2015	FD11	1
51617	10/15/2015	NL89	4
51617	10/15/2015	TW35	3
51619	10/15/2015	FD11	2
51623	10/15/2015	DR67	5
51623	10/15/2015	FH24	12
51623	10/15/2015	KD34	10
51625	10/16/2015	MT03	8

1NF

Orders (OrderNum, OrderDate, ItemNum, NumOrdered)

Second Normal Form (2NF)

- A column is a nonkey column when it is not a part of the primary key (PK)
- A table (relation) is in second normal form (2NF) when it is in 1NF and no nonkey column is dependent on only a portion of the primary key

Orders

OrderNum	OrderDate	ItemNum	Description	NumOrdered	QuotedPrice
51608	10/12/2015	CD33	Wood Block Set (48 piece)	5	\$86.99
51610	10/12/2015	KL78	Pick Up Sticks	25	\$10.95
51610	10/12/2015	TR40	Tic Tac Toe	10	\$13.99
51613	10/13/2015	DL51	Classic Railway Set	5	\$104.95
51614	10/13/2015	FD11	Rocking Horse	1	\$124.95
51617	10/15/2015	NL89	Wood Block Set (62 piece)	4	\$115.99
51617	10/15/2015	TW35	Fire Engine	3	\$116.95
51619	10/15/2015	FD11	Rocking Horse	2	\$121.95
51623	10/15/2015	DR67	Giant Star Brain Teaser	5	\$29.95
51623	10/15/2015	FH24	Puzzle Gift Set	12	\$36.95
51623	10/15/2015	KD34	Pentominoes Brain Teaser	10	\$13.10
51625	10/16/2015	MT03	Zauberkasten Brain Teaser	8	\$45.79

Orders (OrderNum, OrderDate, ItemNum, Description, NumOrdered, QuotedPrice)

This table has the following functional dependencies:

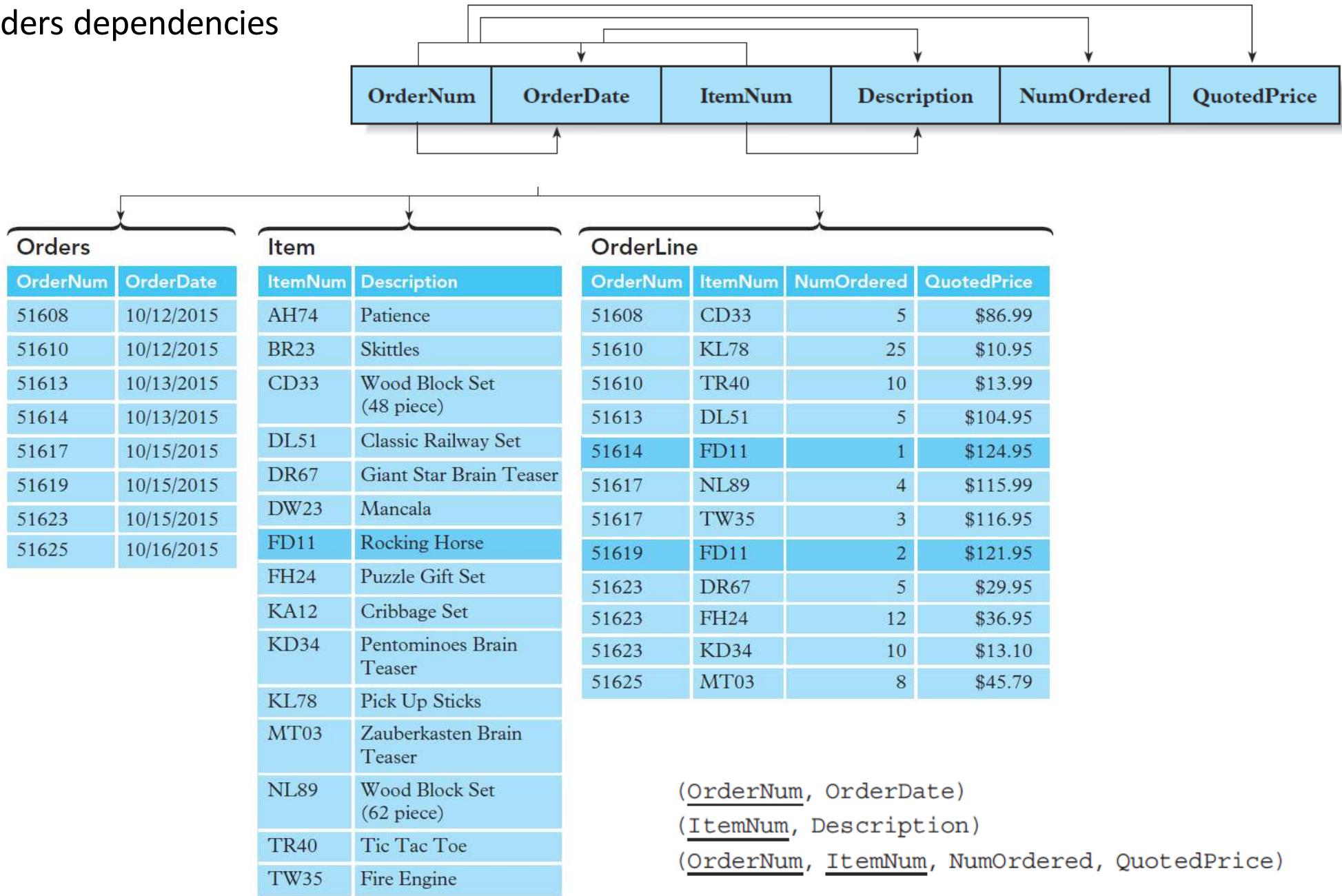
$\text{OrderNum} \rightarrow \text{OrderDate}$

$\text{ItemNum} \rightarrow \text{Description}$

$\text{OrderNum}, \text{ItemNum} \rightarrow \text{NumOrdered}, \text{QuotedPrice}, \text{OrderDate}, \text{Description}$

2NF (cont)

Orders dependencies



Conversion to 2NF

Boyce-Codd Normal Form (BCNF)

- Relation R with FDs F is in BCNF if, for all $X \rightarrow A$ in F^+

- $A \subseteq X$ (trivial FD), or
- X is a super key

i.e. all non-trivial FDs over R are key constraints.

- No redundancy in R (at least none that FDs detect)
- Most desirable normal form
- Customer table is not in BCNF since RepNum is not a key

Customer

CustomerNum	CustomerName	Balance	CreditLimit	RepNum	LastName	FirstName
126	Toys Galore	\$1,210.25	\$7,500.00	15	Campos	Rafael
260	Brookings Direct	\$575.00	\$10,000.00	30	Gradey	Megan
334	The Everything Shop	\$2,345.75	\$7,500.00	45	Tian	Hui
386	Johnson's Department Store	\$879.25	\$7,500.00	30	Gradey	Megan
440	Grove Historical Museum Store	\$345.00	\$5,000.00	45	Tian	Hui
502	Cards and More	\$5,025.75	\$5,000.00	15	Campos	Rafael
586	Almondton General Store	\$3,456.75	\$15,000.00	45	Tian	Hui
665	Cricket Gift Shop	\$678.90	\$7,500.00	30	Gradey	Megan
713	Cress Store	\$4,234.60	\$10,000.00	15	Campos	Rafael
796	Unique Gifts	\$124.75	\$7,500.00	45	Tian	Hui
824	Kline's	\$2,475.99	\$15,000.00	30	Gradey	Megan
893	All Season Gifts	\$935.75	\$7,500.00	15	Campos	Rafael

Customer (CustomerNum, CustomerName, Balance, CreditLimit, RepNum, LastName, FirstName)

The functional dependencies in this table are as follows:

$\text{CustomerNum} \rightarrow \text{CustomerName}, \text{Balance}, \text{CreditLimit}, \text{RepNum}, \text{LastName}, \text{FirstName}$
 $\text{RepNum} \rightarrow \text{LastName}, \text{FirstName}$

Customer (CustomerNum, CustomerName, Balance, CreditLimit, RepNum)

Rep (RepNum, LastName, FirstName)

Third Normal Form (3NF)

- Relation R with FDs F is in **3NF** if, for all $X \rightarrow A$ in F^+

If $\{A_1, \dots, A_n\} \rightarrow B$ is a *non-trivial* FD in R

then $\{A_1, \dots, A_n\}$ is a superkey for R **OR**

B is part of some key of R (**prime attribute**)

- Minimality of a key (not superkey) is crucial!

- BCNF implies 3NF
- e.g.: Sailor (Sailor, Boat, Date, CreditCrd)
 - SBD \rightarrow SBDC, S \rightarrow C (**not 3NF**)
 - If C \rightarrow S, then CBD \rightarrow SBDC (i.e. CBD is also a key). Now in 3NF!
 - Note redundancy in (S, C); 3NF permits this
 - Compromise used when BCNF not achievable, or perf. Consideration
- Lossless-join, dependency-preserving decomposition of R into a collection of 3NF relations always possible.

Multi-Value Dependencies (MVDs)

A multi-value dependency (MVD) is another type of dependency that could hold in our data, ***which is not captured by FDs***

Formal definition:

Given a relation R having attribute set A , and two sets of attributes $X, Y \subseteq A$

The ***multi-value dependency (MVD)*** $X \twoheadrightarrow Y$ holds on R if

for any tuples $t_1, t_2 \in R$ s.t. $t_1[X] = t_2[X]$, there exists a tuple t_3 s.t.:

$$t_1[X] = t_2[X] = t_3[X]$$

$$t_1[Y] = t_3[Y]$$

$$t_2[A \setminus Y] = t_3[A \setminus Y]$$

Where $A \setminus B$ means “elements of set A not in set B ”

Note that this can be avoided all together if each repeated group is put in its own table.

MVDs: Movie Theatre Example

Movie_theater	film_name	snack
Cinema 1	Star Trek: The Wrath of Kahn	Kale Chips
Cinema 1	Star Trek: The Wrath of Kahn	Burrito
Cinema 1	Lord of the Rings: Concatenated & Extended Edition	Kale Chips
Cinema 1	Lord of the Rings: Concatenated & Extended Edition	Burrito
Marcus	Star Wars: The Boba Fett Prequel	Ramen
Marcus	Star Wars: The Boba Fett Prequel	Plain Pasta

Are there any functional dependencies that might hold here?

No...

And yet it seems like there is some pattern / dependency...

MVDs: Movie Theatre Example

Movie_theater	film_name	snack
Cinema 1	Star Trek: The Wrath of Kahn	Kale Chips
Cinema 1	Star Trek: The Wrath of Kahn	Burrito
Cinema 1	Lord of the Rings: Concatenated & Extended Edition	Kale Chips
Cinema 1	Lord of the Rings: Concatenated & Extended Edition	Burrito
Marcus	Star Wars: The Boba Fett Prequel	Ramen
Marcus	Star Wars: The Boba Fett Prequel	Plain Pasta

For a given movie theatre...

MVDs: Movie Theatre Example

Movie_theater	film_name	snack
Cinema 1	Star Trek: The Wrath of Kahn	Kale Chips
Cinema 1	Star Trek: The Wrath of Kahn	Burrito
Cinema 1	Lord of the Rings: Concatenated & Extended Edition	Kale Chips
Cinema 1	Lord of the Rings: Concatenated & Extended Edition	Burrito
Marcus	Star Wars: The Boba Fett Prequel	Ramen
Marcus	Star Wars: The Boba Fett Prequel	Plain Pasta

For a given movie theatre...

Given a set of movies and snacks...

MVDs: Movie Theatre Example

Movie_theater	film_name	snack
Cinema 1	Star Trek: The Wrath of Kahn	Kale Chips
Cinema 1	Star Trek: The Wrath of Kahn	Burrito
Cinema 1	Lord of the Rings: Concatenated & Extended Edition	Kale Chips
Cinema 1	Lord of the Rings: Concatenated & Extended Edition	Burrito
Marcus	Star Wars: The Boba Fett Prequel	Ramen
Marcus	Star Wars: The Boba Fett Prequel	Plain Pasta

For a given movie theatre...

Given a set of movies and snacks...

Any movie / snack combination is possible!

MVDs: Movie Theatre Example

Movie_theater	film_name	snack
t ₁	Cinema 1	Star Trek: The Wrath of Kahn
	Cinema 1	Burrito
	Cinema 1	Kale Chips
t ₂	Cinema 1	Lord of the Rings: Concatenated & Extended Edition
	Cinema 1	Burrito
Marcus	Star Wars: The Boba Fett Prequel	Ramen
Marcus	Star Wars: The Boba Fett Prequel	Plain Pasta

More formally, we write $\{A\} \twoheadrightarrow \{B\}$ if for any tuples t_1, t_2 s.t. $t_1[A] = t_2[A]$

MVDs: Movie Theatre Example

Movie_theater	film_name	snack
t ₁	Cinema 1	Star Trek: The Wrath of Kahn
t ₃	Cinema 1	Star Trek: The Wrath of Kahn
	Cinema 1	Burrito
	Lord of the Rings: Concatenated & Extended Edition	Kale Chips
t ₂	Cinema 1	Lord of the Rings: Concatenated & Extended Edition
	Burrito	
	Star Wars: The Boba Fett Prequel	Ramen
	Star Wars: The Boba Fett Prequel	Plain Pasta

More formally, we write $\{A\} \twoheadrightarrow \{B\}$ if for any tuples t_1, t_2 s.t. $t_1[A] = t_2[A]$ there is a tuple t_3 s.t.

- $t_3[A] = t_1[A]$

MVDs: Movie Theatre Example

	Movie_theater	film_name	snack
t_1	Cinema 1	Star Trek: The Wrath of Kahn	Kale Chips
t_3	Cinema 1	Star Trek: The Wrath of Kahn	Burrito
	Cinema 1	Lord of the Rings: Concatenated & Extended Edition	Kale Chips
t_2	Cinema 1	Lord of the Rings: Concatenated & Extended Edition	Burrito
	Marcus	Star Wars: The Boba Fett Prequel	Ramen
	Marcus	Star Wars: The Boba Fett Prequel	Plain Pasta

More formally, we write $\{A\} \twoheadrightarrow \{B\}$ if for any tuples t_1, t_2 s.t. $t_1[A] = t_2[A]$ there is a tuple t_3 s.t.

- $t_3[A] = t_1[A]$
- $t_3[B] = t_1[B]$

MVDs: Movie Theatre Example

	Movie_theater (A)	film name (B)	Snack (C)
t_1	Cinema 1	Star Trek: The Wrath of Kahn	Kale Chips
t_3	Cinema 1	Star Trek: The Wrath of Kahn	Burrito
	Cinema 1	Lord of the Rings: Concatenated & Extended Edition	Kale Chips
t_2	Cinema 1	Lord of the Rings: Concatenated & Extended Edition	Burrito
	Marcus	Star Wars: The Boba Fett Prequel	Ramen
	Marcus	Star Wars: The Boba Fett Prequel	Plain Pasta

More formally, we write $\{A\} \twoheadrightarrow \{B\}$ if for any tuples t_1, t_2 s.t. $t_1[A] = t_2[A]$ there is a tuple t_3 s.t.

- $t_3[A] = t_1[A]$
- $t_3[B] = t_1[B]$
- and $t_3[R \setminus B] = t_2[R \setminus B]$

Where $R \setminus B$ is “R minus B” i.e. the attributes of R not in B

MVDs: Movie Theatre Example

	Movie_theater (A)	film name (B)	Snack (C)
t ₂	Cinema 1	Star Trek: The Wrath of Kahn	Kale Chips
	Cinema 1	Star Trek: The Wrath of Kahn	Burrito
t ₃	Cinema 1	Lord of the Rings: Concatenated & Extended Edition	Kale Chips
	Cinema 1	Lord of the Rings: Concatenated & Extended Edition	Burrito
	Marcus	Star Wars: The Boba Fett Prequel	Ramen
	Marcus	Star Wars: The Boba Fett Prequel	Plain Pasta

This expresses a sort of dependency (= data redundancy) that we *can't* express with FDs

*Actually, it expresses conditional independence (between film and snack given movie theatre)!

Summary of Normal Forms

Normal Form	Meaning/Required Conditions
1NF	No repeating groups
2NF	1NF and no nonkey column dependent on only a portion of the primary key
3NF	2NF and the only determinants are candidate keys (BCNF) or part of a candidate key
4NF	3NF and no multivalued dependencies

For FDs, BCNF is the normal form

A tradeoff for insert performance: 3NF

For today

- Convert the following table to third normal form. In this table, StudentNum determines Student-Name, NumCredits, AdvisorNum, and AdvisorName. AdvisorNum determines AdvisorName. CourseNum determines Description. The combination of StudentNum and CourseNum determines Grade.

Student (StudentNum, StudentName, NumCredits, AdvisorNum, AdvisorName, (CourseNum, Description, Grade))

Full text search

Full text search

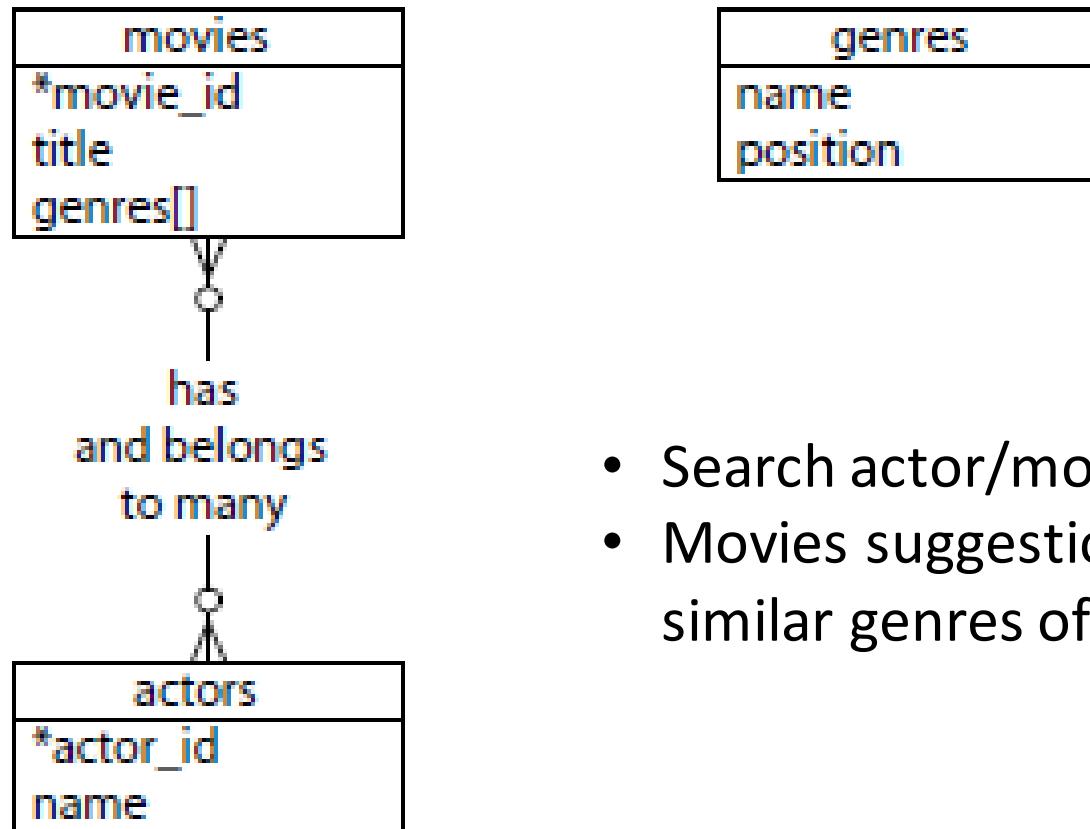
- Capability to identify natural-language *documents* that satisfy a *query*, and optionally to sort them by relevance to the query.
- Find all documents containing given *query terms* and return them in order of their *similarity* to the query.
- query and similarity are flexible terms and their definition depend on the specific application.
- Search frameworks exist:
 - Lucene (<http://lucene.apache.org/>)
 - Elasticsearch
(<https://www.elastic.co/products/elasticsearch>)

Text search in Postgres

Documents are *preprocessed* to support text search:

- *Parsing documents into tokens*, e.g., numbers, words, complex words, email addresses.
PostgreSQL uses a *parser* to perform this step. You can write your own.
- *Converting tokens into lexemes*. A lexeme is a *normalized* string/token so that different forms of the same word map to the same lexeme. This step also typically eliminates *stop words*, common words that are useless for searching. PostgreSQL uses *dictionaries* to perform this step. Several available.
- *Storing preprocessed documents optimized for searching*. For example, each document can be represented as a sorted array of lexemes.
- Two indexes to speed up full text searches: GiST (Generalized Search Tree) and GIN (Generalized Inverted Index).
- Contributed packages
 - <http://www.postgresql.org/docs/current/static/contrib.html>
 - We'll see tablefunc, dict_xsyn, fuzzystmatch, pg_trgm, cube

A movie query system



- Search actor/movie names
- Movies suggestions based on similar genres of movies

- Standard string matches:
 - LIKE and regular expression
 - `SELECT title FROM movies WHERE title ILIKE 'stardust%';`
 - `SELECT title FROM movies WHERE title ILIKE 'stardust_%';`

Fuzzy search

- Approximate string matching
- Regular expressions (regex)
 - POSIX style
 - All movies that do not start with ‘The’:
 - `SELECT COUNT(*) FROM movies WHERE title !~* '^the.*';`
 - `!` -> not matching
 - `~` regular expression match
 - `*` case insensitive
- *Index columns for pattern matching by using a pattern_ops operator class index*
 - `CREATE INDEX movies_title_pattern ON movies (lower(title) text_pattern_ops);`
 - Also available: `varchar_patterns_ops`, `bpchar_pattern_ops`, and `name_pattern_ops`

Levenshtein string comparison

- Edit distance: steps required to change one string into another: count character updates, insertions, and deletions
 - `SELECT levenshtein('bat', 'fads');`
 - Distance = 3
 - Case changes cost a point too, convert all strings to the same case before querying:
 - ```
SELECT movie_id, title
FROM movies
WHERE levenshtein(lower(title),
lower('a hard day nght')) <= 3;
```

# Trigrams

- A trigram is a group of three consecutive characters taken from a string
  - `SELECT show_trgm('Avatar');`
- To query: count the number of matching trigrams, return the most similar
- Great for movie titles because they have similar lengths
- Generalized Index Search Tree (GiST), generic API
  - `CREATE INDEX movies_title_trigram ON movies USING gist (title gist_trgm_ops);`
- Now query:
  - `SELECT title FROM movies WHERE title % 'Avatre';`

# Full-text

- *@@ Text-search matching operator:*
  - ```
SELECT title FROM movies  
WHERE title @@ 'night & day';
```
- Special datatypes to split string into an array of tokens:
 - TSVECTOR to represent text data
 - TSQUERY to represent search predicates
 - Equivalent to above query:
 - ```
SELECT title FROM movies WHERE
to_tsvector(title) @@ to_tsquery('english',
'night & day');
```

# TSVector and TSQuery

- *Tokens on a tsvector are called lexemes*
- ```
SELECT to_tsvector('A Hard Day''s Night'),  
       to_tsquery('english', 'night & day');
```
- *Number is the position of the lexeme in the text*
- *Stop words are removed*
- ```
SELECT * FROM movies WHERE title @@
 to_tsquery('english', 'a');
```
- *Simple dictionary: breaks up strings by nonword  
characters and makes them lowercase*
- ```
SELECT to_tsvector('simple', 'A Hard Day''s Night');
```
- *Support for many languages*
 - ```
SELECT to_tsvector('spanish', 'que haces?');
```

# Indexing lexemes

- *GIN – Generalized Inverted index, like GiST, it's an index API*
- ```
CREATE INDEX movies_title_searchable ON movies
USING gin(to_tsvector('english', title));
```
- ```
EXPLAIN SELECT * FROM movies WHERE title @@ 'night & day';
```
- ```
EXPLAIN SELECT * FROM movies WHERE
to_tsvector('english',title) @@ 'night & day';
```

Example “Collection”

Four sentences from the Wikipedia entry for *tropical fish*

- S_1 Tropical fish include fish found in tropical environments around the world, including both freshwater and salt water species.
- S_2 Fishkeepers often use the term tropical fish to refer only those requiring fresh water, with saltwater tropical fish referred to as marine fish.
- S_3 Tropical fish are popular aquarium fish, due to their often bright coloration.
- S_4 In freshwater fish, this coloration typically derives from iridescence, while salt water fish are generally pigmented.

Simple Inverted Index

and	1	only	2
aquarium	3	pigmented	4
are	3	popular	3
around	4	refer	2
as	1	referred	2
both	1	requiring	2
bright	3	salt	1
coloration	3	saltwater	4
derives	4	species	1
due	3	term	2
environments	1	the	1
fish	1	their	2
fishkeepers	2	this	3
found	1	those	4
fresh	2	to	2
freshwater	1	tropical	3
from	4	typically	1
generally	4	use	2
in	1	water	2
include	4	while	3
including	1	with	4
iridescence	1	world	2
marine	2		1
often	2		
	3		

Inverted Index

- Each index term (posting) is associated with an *inverted list*
 - Contains lists of documents, or lists of word occurrences in documents, and other information
 - Lists are usually *document-ordered* (sorted by document number)

Metaphones

- *Algorithms to create string representation of word sounds*
- *Select metaphone ('Modern databases', 9);*
- *SELECT * FROM actors WHERE name = 'Broos Wils'; /*No matches*/*
- *SELECT * FROM actors WHERE name % 'Broos Wils'; /*Using trigrams*/*
- *SELECT * FROM actors WHERE metaphone(name, 6) = metaphone('Broos Wils', 6); /*Using metaphone*/*

More fuzzymatch functions

- *dmetaphone*: double metaphone
- *dmetaphone_alt*: alternative name pronunciations
- *soundex*: old algorithm (1880) to compare American surnames
- ```
SELECT name, dmetaphone(name),
dmetaphone_alt(name), metaphone(name, 8),
soundex(name) FROM actors;
```
- One of the most flexible aspects of metaphones is that their outputs are just strings

# For today...

Write a query that combines AT LEAST TWO of the string match functions we covered today.

For example, use the trigram operator against metaphone() outputs and then order the results by the lowest Levenshtein distance.

E.g. “Get me names that sound the most like Robin Williams”

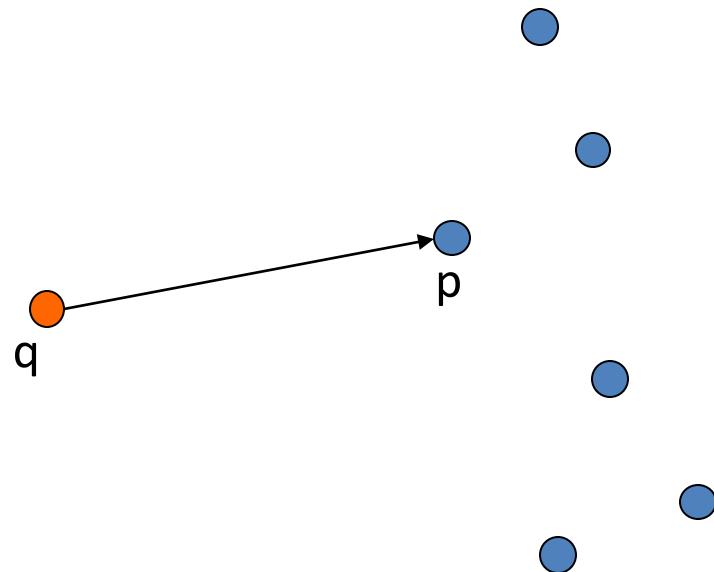
Submit your sql statement in ICON

# Multidimensional queries

# Nearest Neighbor (NN) Search

*Data is often modeled as a set of  $n$  points in a multidimensional space ( $R^d$ )*

*Given a query point  $q$ , a nearest neighbor query will find the nearest neighbor  $p$  of  $q$ , i.e. the closest point given a distance metric*

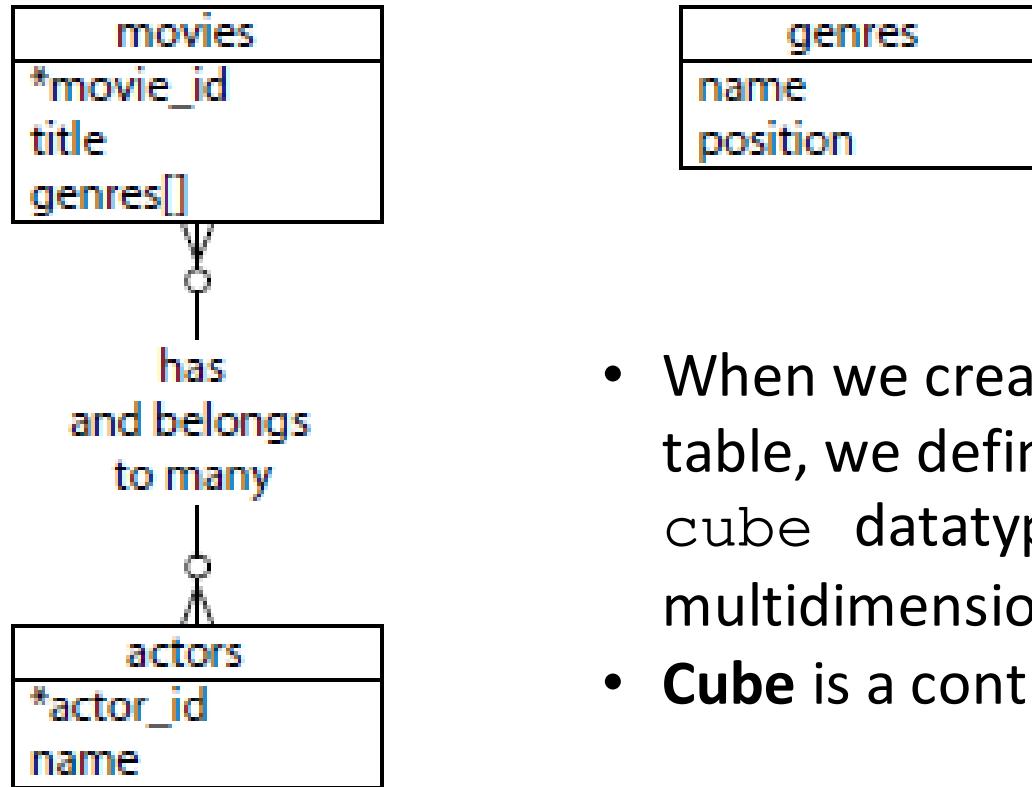


# Nearest neighbor applications

(Longer list available in Wikipedia)

- Pattern recognition
- Statistical classification
- Computer vision
- Content-based image retrieval
- Recommendation systems
- Internet marketing
- Spell checking
- Plagiarism detection
- Similarity scores
- Cluster analysis
- Chemical similarity

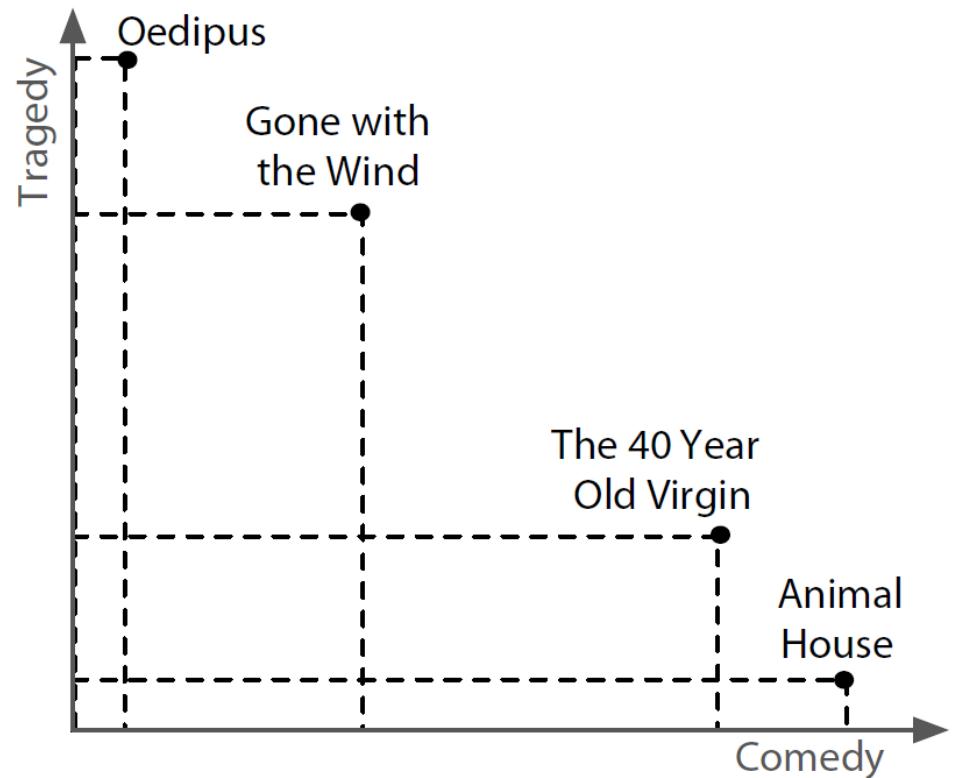
# Our movie system



- Each value for genres in movies is a point in 18-dimensional space with each dimension representing a genre
- Why does it make sense? Movies can span multiple genres, many movies are not 100% comedy or 100% tragedy – they are in between
- When we created the movies table, we defined genres as a cube datatype, i.e. a multidimensional vector
- **Cube** is a contributed package

# Movies recommendation

- We'll find similar movies by finding NN
- Figure shows four movies in a 2-dimensional space
- If your favorite movie is *Animal House*, you'd probably would like to see *The 40-year-old virgin* more than *Oedipus*
- We'll use 18 genres, but the principle will be the same



# Genres in our movie system

- Each genre is scored from 0-10 (totally arbitrary)
  - 0 means nonexistent
  - 10 being the strongest
- What is the genre vector of *Star wars*?
  - `SELECT title, genre FROM movies WHERE lower(title)='star wars'`
  - `(0,7,0,0,0,0,0,0,7,0,0,0,0,10,0,0,0)`
- Each position maps to an entry in the genres table:
  - `SELECT * FROM genres;`

# The cube package

- <https://www.postgresql.org/docs/10/cube.html>
- We can decrypt the genre values by using  
cube\_ur\_coord

cube\_ur\_coord(cube, integer)

Returns the *n*-th coordinate value for the upper right corner of the cube.

- ```
SELECT name,
       cube_ur_coord('(0,7,0,0,0,0,0,0,7,0,0,0,0,10,0,0,0)', position)
    as score FROM genres g
   WHERE cube_ur_coord('(0,7,0,0,0,0,0,0,7,0,0,0,0,10,0,0,0)', position) > 0;
```

Select movies with a particular genre

- *SELECT title, genre
FROM movies WHERE
cube_ur_coord(genre,
15)>5*
- *SELECT title, genre
FROM movies
WHERE
cube_ur_coord(genre,
(select position from
genres where
name='Comedy'))>5;*

name	position
Action	1
Adventure	2
Animation	3
Comedy	4
Crime	5
Disaster	6
Documentary	7
Drama	8
Eastern	9
Fantasy	10
History	11
Horror	12
Musical	13
Romance	14
SciFi	15
Sport	16
Thriller	17
Western	18

Nearest neighbor search

- We'll use function
 `cube_distance(point1, point2)`
- *Find the distance of all movies to the Star Wars genre vector, nearest first:*

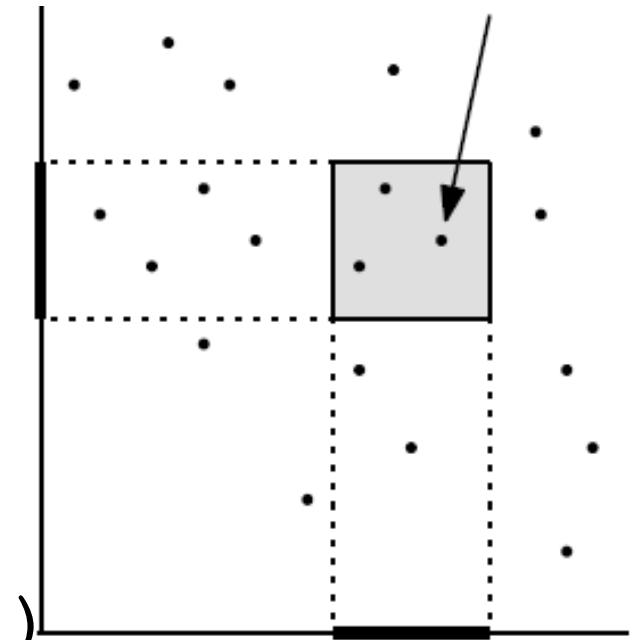
```
SELECT *,  
       cube_distance(genre,  
                      '(0, 7, 0, 0, 0, 0, 0, 0, 7, 0, 0, 0, 0, 10, 0, 0, 0)')  
    dist  
   FROM movies  
 ORDER BY dist;
```

Nearest neighbor performance

- What is the cost of a NN query?
- What about the movies_genres_cube cube index we created last lecture?
- We can rewrite the NN query as a range query:
- Near neighbor: *find one/all points within distance r from q*
- The index can be used to reduce the number of points we need to look at
- Any downside?

Near neighbor query

- *Define a bounding cube containing the query*
- *All points inside the cube are potential NN*
- Use `cube_enlarge(cube, r, n)` to increase the size of the cube by the specified radius r in at least n dimensions.
- Create a two-dimensional square around point (1,1) of one unit: the lower-left point would be at (0,0), and the upper-right point at (2,2):
`SELECT cube_enlarge(' (1,1)', 1, 2);`



Recommending movies

Find all movies within a distance of 5-unit cube from the *Star Wars* genre point

Contains operator @>: a @> b returns whether cube a contains cube b

```
SELECT title,  
cube_distance(genre,  
'(0,7,0,0,0,0,0,0,7,0,0,0,0,0,10,0,0,0)' ) dist FROM  
movies WHERE  
cube_enlarge('(0,7,0,0,0,0,0,0,0,0,0,0,0,0,10,0,0,0  
)' ::cube, 5, 18)  
@> genre ORDER BY dist;
```

Recommending movies

Using a subselect, we can get the genre by movie name and perform our calculations against that genre using a table alias.

```
SELECT m.movie_id, m.title
FROM movies m,
(SELECT genre, title FROM movies WHERE title =
'Mad Max') s
WHERE cube_enlarge(s.genre, 5, 18) @> m.genre
AND s.title <> m.title
ORDER BY cube_distance(m.genre, s.genre)
LIMIT 10;
```

For today...

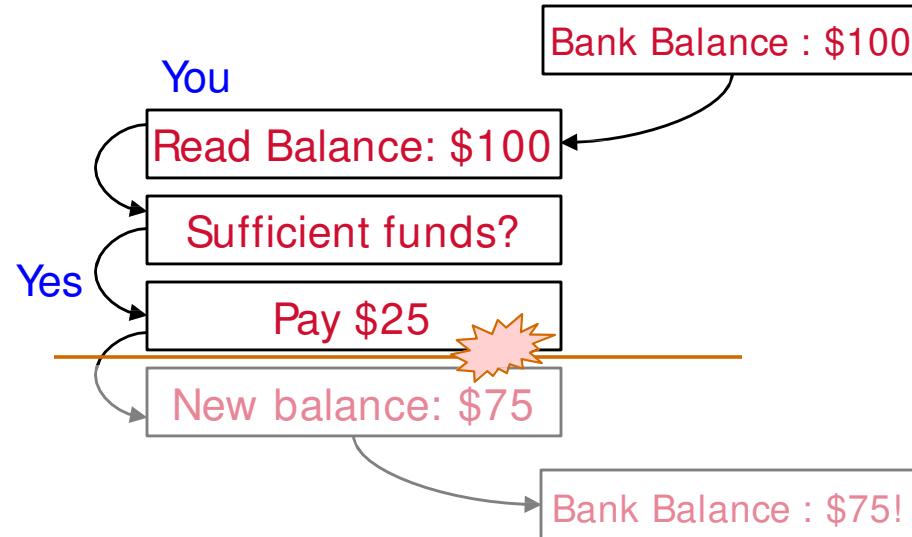
Create a stored procedure that enables you to input a movie title or an actor's name and then receive the top five suggestions based on either movies the actor has starred in or films with similar genres.

Include two calls to your function.

TRANSACTION MANAGEMENT

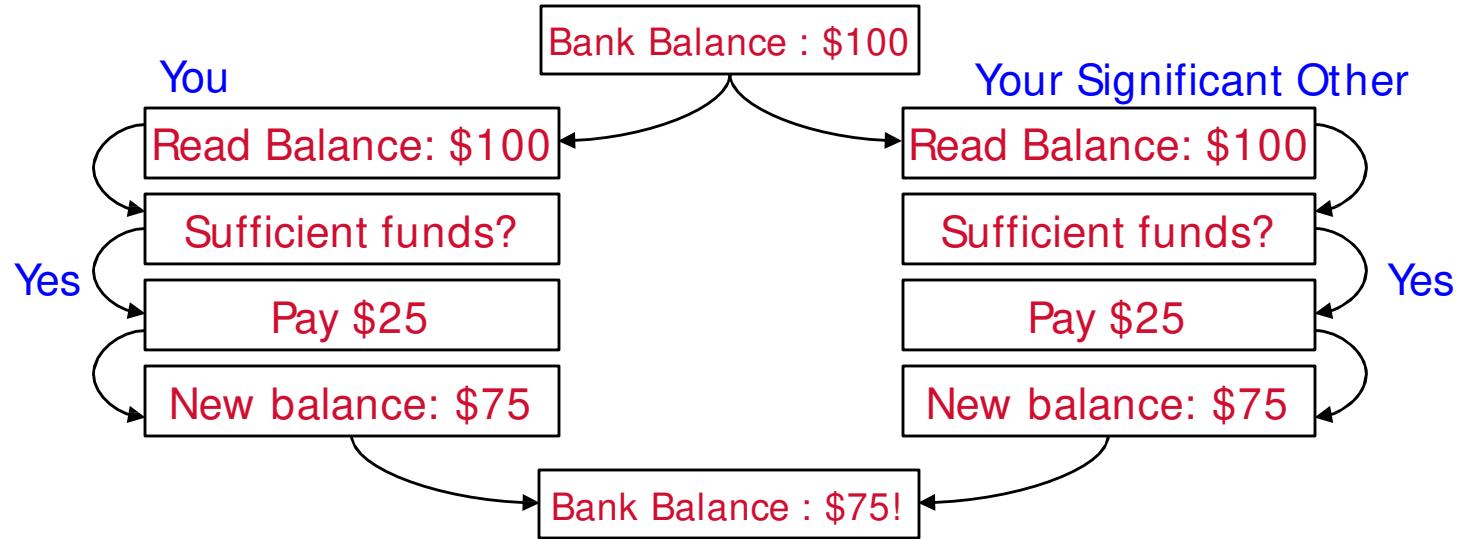
Transaction Management

```
Read (A);  
Check (A > $25);  
Pay ($25);  
A = A - 25;  
Write (A);
```



Transaction Management

```
Read (A);  
Check (A > $25);  
Pay ($25);  
A = A - 25;  
Write (A);
```



- Inconsistency
 - Interleaving actions of different user programs
 - System crash/user abort/...
- Provide the users an illusion of a single-user system
 - Could insist on admitting only one query into the system at any time
 - lower utilization: CPU/IO overlap
 - long running queries starve other queries

What is a Transaction?

- Collection of operations that form a single logical unit
 - A sequence of many actions considered to be one atomic unit of work
- Logical unit:
 - `begin transaction (SQL) end transaction`
- Operations:
 - Read (X), Write (X): Assume R/W on tuples (can be relaxed)
 - Special actions: `begin, commit, abort`
- Desirable Property: Must leave the DB in a consistent state
 - (DB is consistent when the transaction begins)
 - Consistency: DBMS only enforces integrity constraints (IC) specified by the user
 - DBMS does not understand any other semantics of the data

The ACID Properties



Xact. Mgmt.
(logging)

• • • **A**tomicity: All actions in the Xact happen, or none happen.



• • • **C**onsistency: Consistent DB + consistent Xact \Rightarrow consistent DB



Concurrency Ctrl.
(locking)

• • • **I**solation: Execution of one Xact is isolated from that of other Xacts.



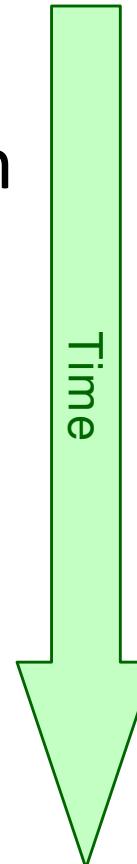
Recovery Mgmt.
(WAL, ...)

• • • **D**urability: If a Xact commits, its effects persist.

```
Begin  
  Read (A);  
  A = A - 25;  
  Write (A);  
  Read (B);  
  B = B + 25;  
  Write (B);  
Commit
```

Schedules

- **Schedule**: An interleaving of actions from a set of Xacts, where the actions of any one Xact are in the original order.
 - Actions of Xacts *as seen by the DB*
 - *Complete schedule* : each Xact ends in commit or abort
 - *Serial schedule* : No interleaving of actions from different Xacts.
- Initial State + Schedule → Final State



I1	I2
begin	
R(A)	
W(A)	
	begin
	R(B)
	W(B)
R(C)	
W(C)	
	commit
abort	

Acceptable Schedules

- One sensible “isolated, consistent” schedule:
 - Run Xacts one at a time (serial schedule)
- Serializable schedules:
 - Final state is what *some complete* serial schedule of *committed* transactions would have produced.
 - Can different serial schedules have different final states?
 - Yes, all are “OK”!
 - Aborted Xacts?
 - ignore them for a little while (made to ‘disappear’ using logging)
 - Other external actions (besides R/W to DB)
 - e.g. print a computed value, fire a missile, ...
 - Assume (for this class) these values are written to the DB, and can be undone

Serializability Violations

- @Start (A,B) = (1000, 100)
 - End (990, 210)
- T1→T2:
 - (900, 200) → (990, 220)
- T2→T1:
 - (1100, 110) → (1000, 210)
- **W-R conflict:** Dirty read
 - Could lead to a non-serializable execution
- **Also R-W and W-W conflicts**

<i>T1: Transfer \$100 from A to B</i>	<i>T2: Add 10% interest to A & B</i>
begin	
	begin
R(A) / A -= 100	
W(A)	
	R(A) / A * = 1.1
	W(A)
	R(B) / B * = 1.1
	W(B)
	commit
R(B) / B += 100	
W(B)	
commit	

Database Inconsistent

More Conflicts

- **RW Conflicts (Unrepeatable Read)**
 - $R_{T_2}(X) \rightarrow W_{T_1}(X)$, T1 overwrites what T2 read.
 - $R_{T_2}(X) \rightarrow W_{T_1}(X) \rightarrow R_{T_2}(X)$. T2 sees a different X value!
- **WW Conflicts (Overwriting Uncommitted Data)**
 - T2 overwrites what T1 wrote.
 - E.g. : Students in the same group get the same project grade.
 - $T_P: W(X=A), W(Y=A)$ $T_{TA}: W(X=B), W(Y=B)$
 - $W_P(X=A) \rightarrow W_{TA}(X=B) \rightarrow W_{TA}(Y=B) \rightarrow W_P(Y=A)$
[Note: no reads]
 - Usually occurs in conjunction with other anomalies.

Now, Aborted Transactions

- Serializable schedule: Equivalent to a serial schedule of *committed* Xacts.
 - as if aborted Xacts *never happened*.
- Two Issues:
 - How does one undo the effects of a Xact?
 - Logging/recovery
 - What if another Xact sees these effects??
 - Must undo that Xact as well!

Cascading Aborts

- Abort of T1 requires abort of T2!
 - Cascading Abort

T1	T2
begin	
R(A)	
W(A)	
	begin
	R(A)
	W(A)
	commit
abort	

Cascading Aborts

- Abort of T1 requires abort of T2!
 - **Cascading Abort**
- Consider commit of T2
 - Can we undo T2?
- *Recoverable* schedule: Commit only after all xacts that supply dirty data have committed.

T1	T2
begin	
R(A)	
W(A)	
	begin
	R(A)
	W(A)
commit	
	commit

Cascading Aborts

- *ACA (avoids cascading abort) schedule*

- Transaction only reads committed data
- One in which cascading abort cannot arise.
- Schedule is also recoverable

T1	T2
begin	
R(A)	
W(A)	
	begin
	R(A)
	W(A)
abort	
	commit

T1	T2
begin	
R(A)	
W(A)	
commit	
	begin
	R(A)
	W(A)
	Commit

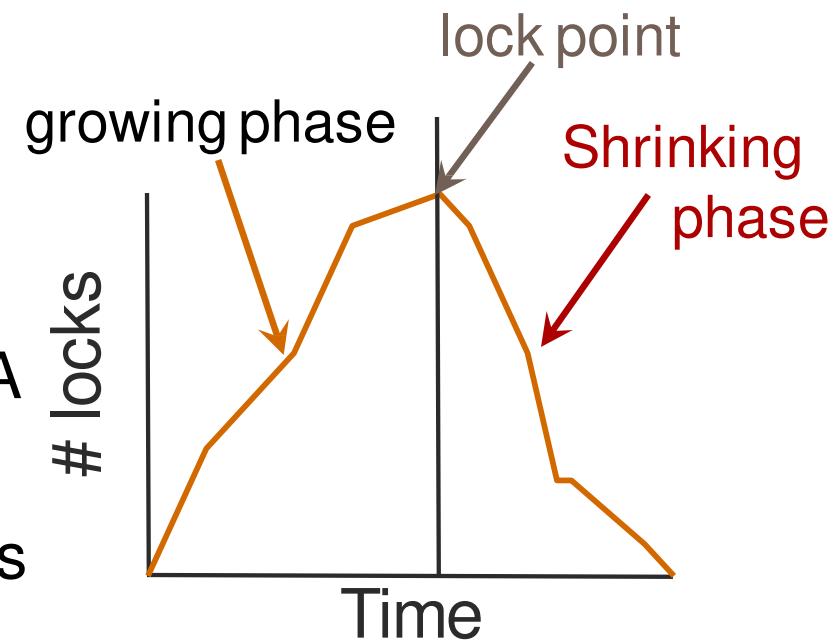
Locking: A Technique for C. C.

- Concurrency control usually done via locking.
- Lock info maintained by a “lock manager”:
 - Stores (XID, RID, Mode) triples.
 - This is a simplistic view; suffices for now.
 - Mode $\in \{S,X\}$
 - Lock compatibility table:
- If a Xact can't get a lock
 - Suspended on a wait queue
- When are locks acquired?
 - Buffer manager call!

	--	S	X
--	✓	✓	✓
S	✓	✓	
X	✓		

Two-Phase Locking (2PL)

- **2PL:**
 - If T wants to read (modify) an object, first obtains an S (X) lock
 - If T releases any lock, it can acquire no new locks!
 - **Guarantees serializability! Why?**
- **Strict 2PL:**
 - Hold all locks until end of Xact
 - Guarantees serializability, and ACA too!
 - Note ACA schedules are always recoverable



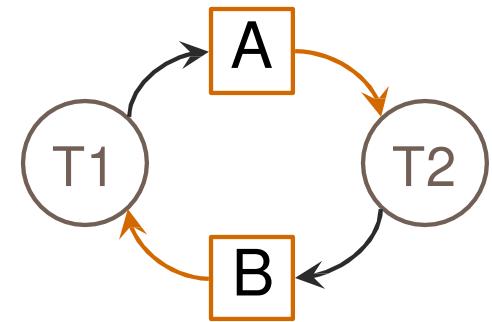
Schedule with Locks

<i>T1: Transfer \$100 from A to B</i>	<i>T2: Add 10% interest to A & B</i>
begin	
	begin
R(A) / A -= 100	
W(A)	
	R(A) / A * = 1.1
	W(A)
	R(B) / B * = 1.1
	W(B)
	commit
R(B) / B += 100	
W(B)	
commit	

<i>T1</i>	<i>T2</i>
begin	
	begin
X(A)	
R(A)	
W(A)	
	X(A) – Wait!
X(B)	
R(B)	
W(B)	
U_x(A), U_x(B)/commit	
	R(A)
	W(A)
	...

Deadlocks

$X_{T1}(B), X_{T2}(A), S_{T1}(A), S_{T2}(B)$



- Deadlocks can cause the system to wait forever.
- Need to detect deadlock and break, or prevent deadlocks
- Simple mechanism: timeout and abort
- More sophisticated methods exist

Conflict Serializability & Graphs

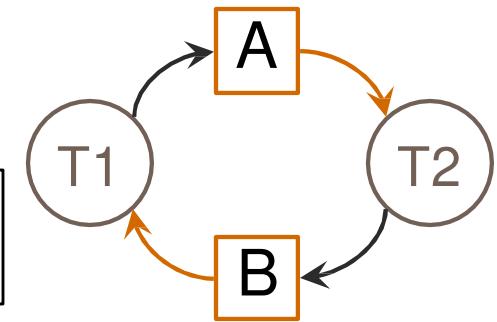
Theorem: A schedule is conflict serializable iff its precedence graph is acyclic

Theorem: 2PL ensures that the precedence graph will be acyclic

- Why Strict 2PL?
 - Guarantees ACA
 - read only committed values
 - How? Write locks until EOT
 - No WW or WR => on abort replace original value

Deadlocks

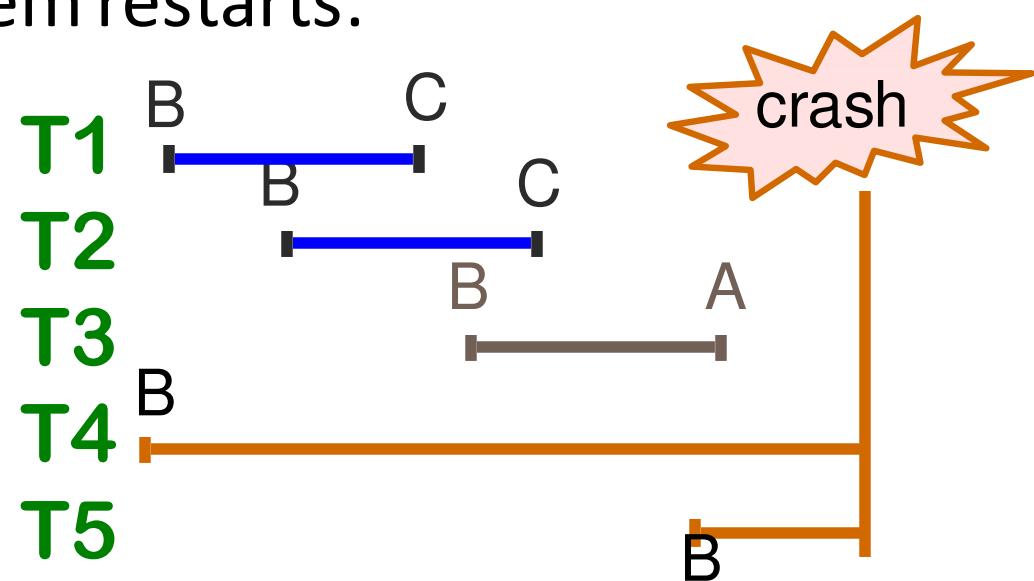
$X_{T1}(B), X_{T2}(A), S_{T1}(A), S_{T2}(B)$



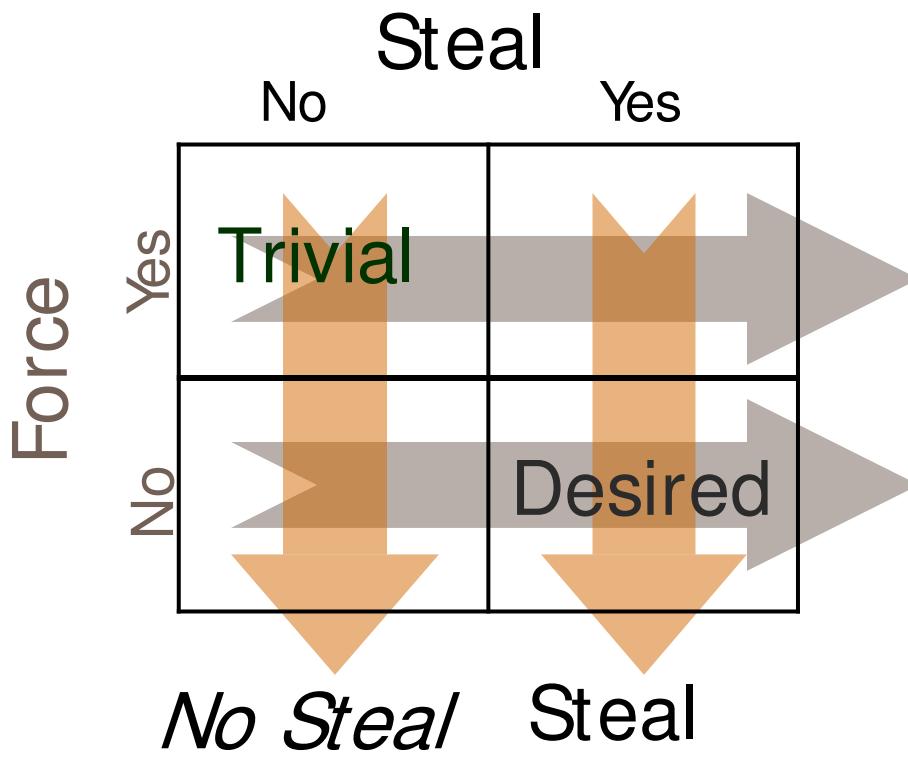
- Deadlocks can cause the system to wait forever.
- Need to detect deadlock and break, or prevent deadlocks
- Detect deadlock
 - Draw a lock graph. Cycles implies a deadlock
- Alternative ways of dealing with deadlock
 - Break Deadlock
 - On each lock request “update the lock graph”. If a cycle is detected, abort one of the transactions. The aborted transaction is restarted after waiting for a time-out interval.
 - Prevent deadlock
 - Assign priorities to the transactions. If a transaction, T1, requests a lock that is being held by another transaction, T2, with a lower priority, then T1 “snatches” the lock from T2 by aborting T2 (which frees up the lock on the resource). T2 is then restarted again after a time-out.

Ensuring Atomicity & Durability

- Atomicity:
 - Transactions may abort (“Rollback”) -> no effects should be seen
- Durability:
 - What if DBMS stops running? (Causes?)
- Desired Behavior after system restarts:
 - T1, T2 & T3 should be durable.
 - T4 & T5 should be aborted



Buffer Pool: Sharing & Writing



*Poor throughput,
but works*

- Page being stolen (and flushed) was modified by an uncommitted Xact T
- If T aborts, how is atomicity enforced?
- Soln: Remember old value (logs). Use this to UNDO

Force

- *Poor response time, but durable*

No Force

- Crash before a page is flushed to disk
- Soln: Force a short summary @ commit (logs). Use this to REDO

Basic Idea: Logging

- Record information, for every change, in a *log*.
 - Sequential writes to log (put it on a separate disk).
 - Stored in stable storage to survive system crash
 - disk mirroring
 - Each record has a log sequence number (LSN)
 - Log record contains:
 - <prevLSN, XID, type, ... >
 - and additional control info (which we'll see soon)
 - Note: the log records for a transaction are chained by prevLSN

Write-Ahead Logging (WAL)

- The Write-Ahead Logging Protocol:
 1. Must force the log record for an update before the corresponding data page gets to storage.
 2. Must write all log records for a Xact before commit.
- #1 guarantees Atomicity.
- #2 guarantees Durability.

If DB says TX **commits**, TX effect **remains** after database crash

DB can **undo actions** and help us with **atomicity**

Normal Execution of a Xact

- Series of **reads & writes**, followed by **commit** or **abort**.
 - Updates are “in place”: i.e., data on disk is overwritten
 - We will assume that write is atomic on disk.
 - In practice, additional details to deal with non-atomic writes.
- Strict 2PL.
- STEAL, NO-FORCE buffer management, with **Write-Ahead Logging**.

The ACID Properties

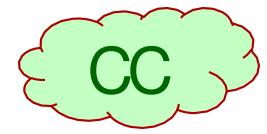


Xact. Mgmt.
(logging)

- Atomicity: All actions in the Xact happen, or none happen.



- Consistency: Consistent DB + consistent Xact \Rightarrow consistent DB



Concurrency Ctrl.
(locking)

- Isolation: Execution of one Xact is isolated from that of other Xacts.



Recovery Mgmt.
(WAL, ...)

- Durability: If a Xact commits, its effects persist.

Postgres

- PostgreSQL transactions follow ACID compliance
- Atomicity & Durability: WAL logging, Continuous Archiving and Point-in-Time Recovery (PITR)
- Consistency: Multiversion Concurrency Control, MVCC
- Isolation: *Serializable Snapshot* Isolation (SSI)
- Every command we've executed in psql has been implicitly wrapped in a transaction
- Explicit transaction:

```
BEGIN TRANSACTION;  
    DELETE FROM events;  
ROLLBACK;  
SELECT * FROM events;
```

END; or **COMMIT;** Commits the current transaction

Install MongoDB

We will be using the free MongoDB Community Edition Database server.

<https://www.mongodb.com/try/download/community>

Download a build for your OS

After download completes, follow the Installation instructions

<https://docs.mongodb.com/manual/installation/>

From a terminal window in your computer, run the mongoDB server (mongod).

Then run the mongo shell client (mongo) -

<https://docs.mongodb.com/manual/mongo/>

Take a screenshot of your terminal window and upload to ICON in the dropbox.

To exit the shell, type **quit()** or use the <Ctrl-C> shortcut