Lecture 15: The Details of Joins

(and bonus!)

What you will learn about in this section

1. How to choose between BNLJ, SMJ

2. HJ versus SMJ

3. Buffer Manager Detail (PS#3!)

4. ACTIVITY: Buffer Manager DETAIL!

Nested Loop Joins

Notes

- We are again considering "IO aware" algorithms: ***care about disk IO***

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

- Note also that we omit ceilings in calculations… good exercise to put back in!

Block Nested Loop Join (BNLJ)

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

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

Cost:

$P(R)$

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

Block Nested Loop Join (BNLJ)

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

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

Cost:

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

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

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

This line is called $\frac{P(R)}{B-1}$ *times; the loop iterates over the entire relation S* $\frac{P(R)}{B-1}$ *times (ceiling!)*

Block Nested Loop Join (BNLJ)

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

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

Cost:

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

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

BNLJ can also handle non-equality constraints

Block Nested Loop Join (BNLJ)

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

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

Cost:

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

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

BNLJ: Some quick facts.

- We use B+1 buffer pages as:
  - 1 page for S
  - 1 page for output
  - B-1 Pages for R

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

- If P(R) <= B-1 then we do one pass over S, and we run in time P(R) + P(S) + OUT.
  - Note: This is **optimal** for our cost model!
  - Thus, if min {P(R), P(S)} <= B-1 we should **always** use BNLJ
    - We use this at the end of **hash join.** *We define end condition, one of the buckets is smaller than B-1!*

1. Sort-Merge Join (SMJ)

Sort Merge Join (SMJ): Basic Procedure

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

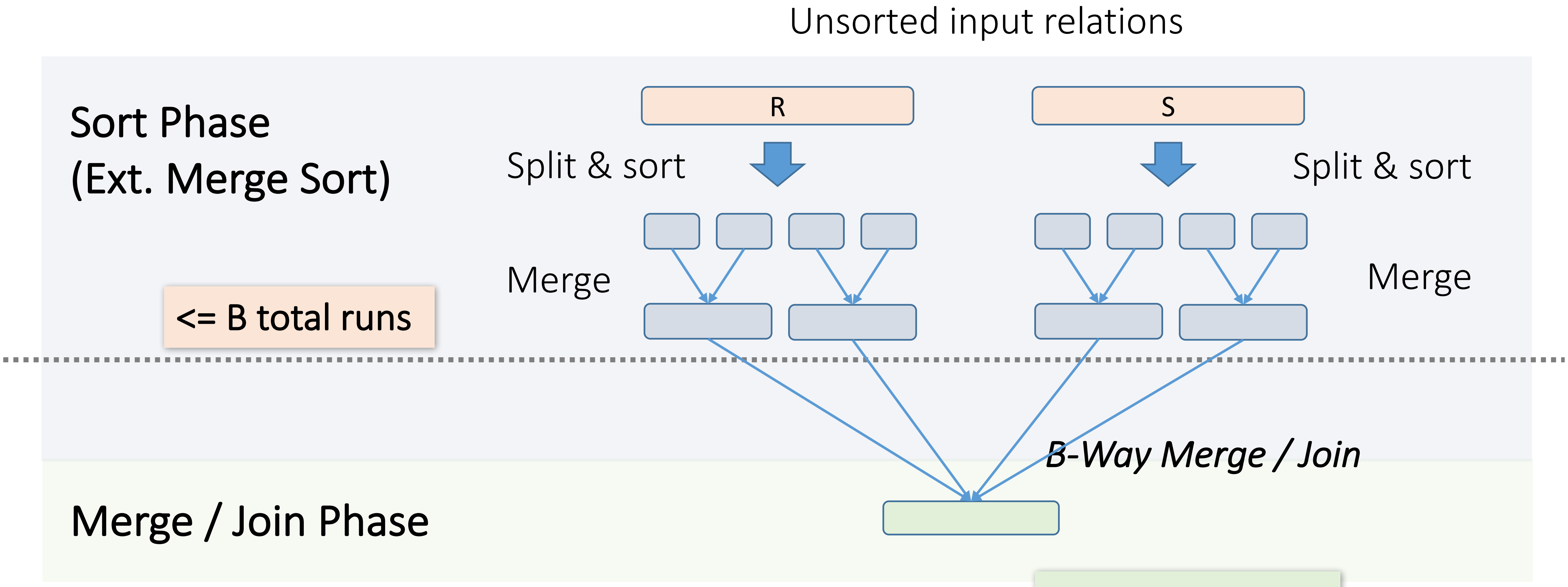
Note that we are only considering equality join conditions here

1. Sort R, S on A using ***external merge sort***
2. ***Scan*** sorted files and "merge"
3. *[May need to "backup"- if there are many duplicate join keys]*

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

Simple SMJ Optimization

Given *B+1* buffer pages



Joined output file created!

Simple SMJ Optimization

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

- On this last pass, we only do P(R) + P(S) + OUT IOs to complete the join!

- If we can initially split R and S into **B total runs each of length approx** 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?
  - $\frac{P(R)+P(S)}{B} \leq 2(B+1) \Rightarrow \sim \mathrm{P(R)} + \mathrm{P(S)} \leq 2B^2$
  - **Thus,** $\mathbf{max\{P(R), P(S)\}} \leq \boldsymbol{B^2}$ **is an approximate sufficient condition**

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

Bonus questions.

- Q1: Fast dog.
  - If max {P(R), P(S)} < B^2 then SMJ takes 3(P(R) + P(S)) + OUT
  - What is the similar condition to obtain 5(P(R) + P(S)) + OUT?
  - What is the condition for (2k+1)(P(R) + P(S)) + OUT
- Q2: BNLJ V. SMJ
  - Under what conditions will BNLJ outperform SMJ?
    - Size of R, S and # of buffer pages
- Discuss! And We'll put up a google form.

2. Hash Join (HJ)

Hash Join: High-level procedure

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

1. **Partition Phase:** Using one (shared) hash function $\boldsymbol{h_B}$ per pass partition R *and* S into ***B*** buckets.
   - Each phase creates B more buckets that are a factor of B smaller.
   - Repeatedly partition with a new hash function
   - Stop when all buckets for one relation are smaller than B-1 (Why?)

Each pass takes 2(P(R) + P(S))

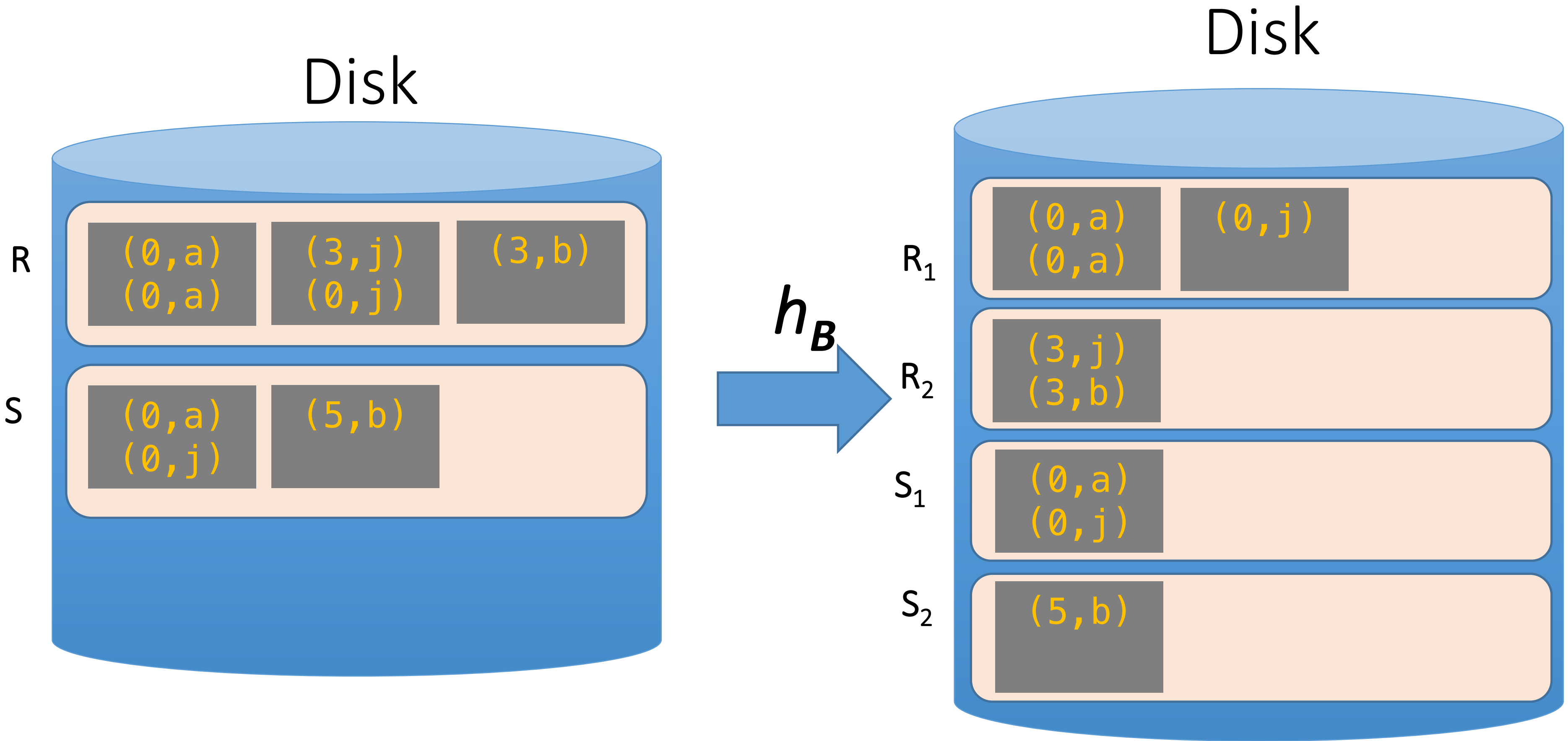
2. **Matching Phase:** Take pairs of buckets whose tuples have the same values for ***h***, and join these
   - Use BNLJ here for each matching pair.

P(R) + P(S) + OUT

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

Hash Join: High-level procedure

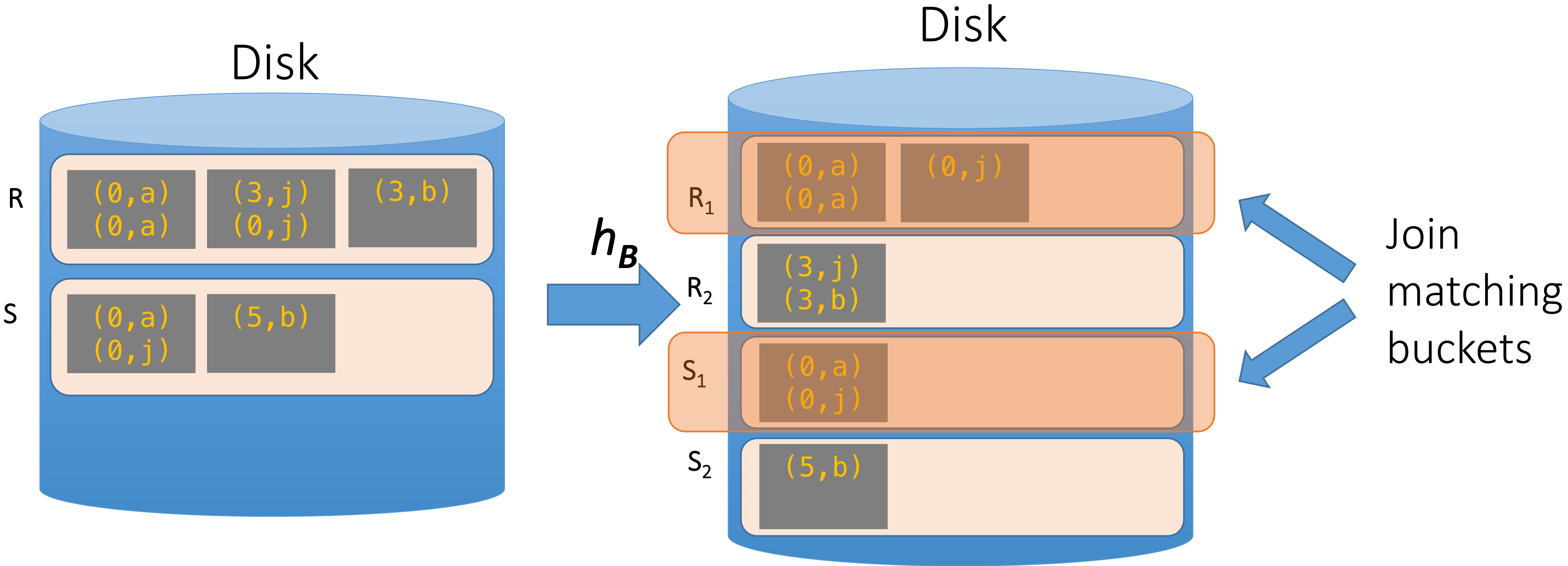
1. Partition Phase: Using one (shared) hash function $\boldsymbol{h_B}$, partition R *and* S into ***B*** buckets



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

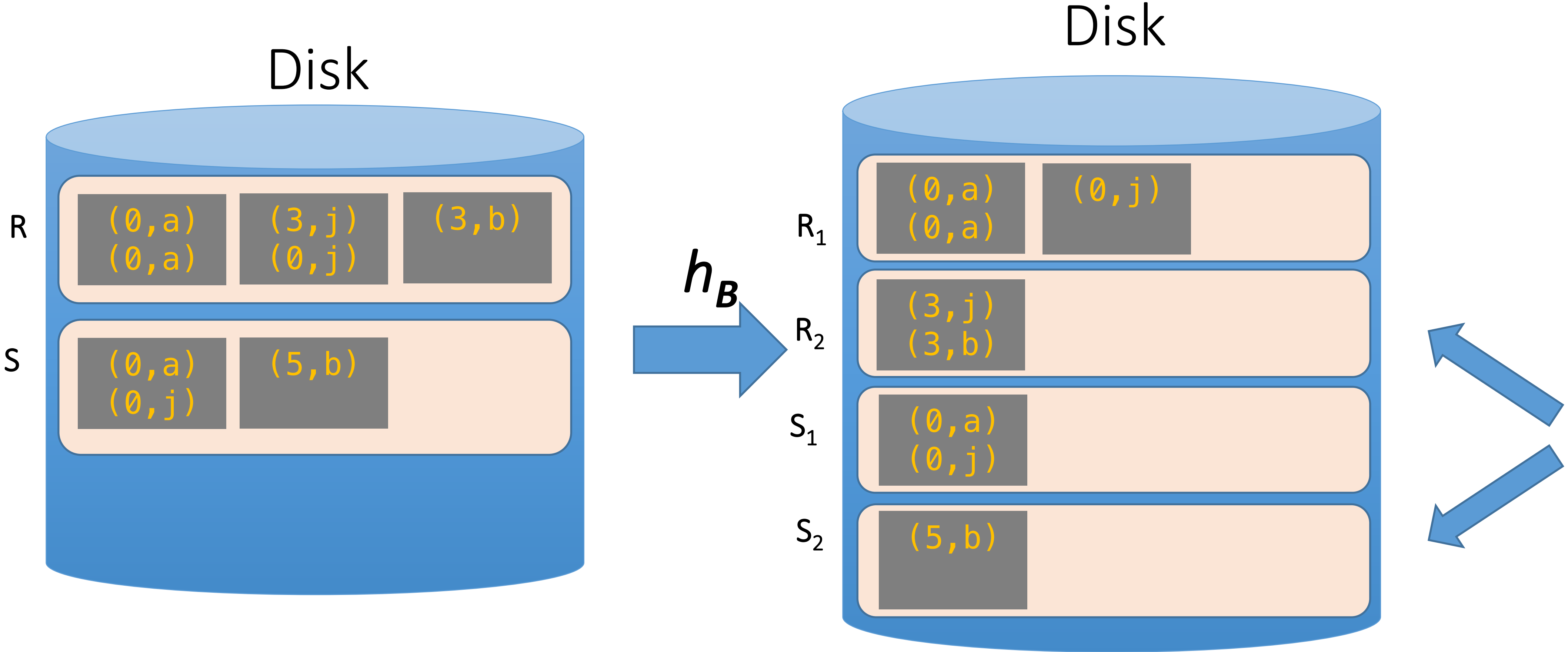
Hash Join: High-level procedure

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



Hash Join: High-level procedure

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



Don’t have to join the others! E.g. these!

Bonus questions #2

- Q1: Fast little dog.
  - If min {P(R), P(S)} < B^2 then HJ takes 3(P(R) + P(S)) + OUT
  - What is the similar condition to obtain 5(P(R) + P(S)) + OUT?
  - What is the condition for (2k+1)(P(R) + P(S)) + OUT
- Q2: SMJ V. HJ
  - Under what conditions will HJ outperform SMJ?
  - Under what conditions will SMJ outperform SMJ?
    - Size of R, S and # of buffer pages
- Discuss! And We'll put up a google form.

Sort-Merge v. Hash Join

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

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

- ***"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

- I will ask you to compute costs on the final and PS
  - Walk through the algorithms, you'll be able to compute the costs!

- Memory sizes key in hash versus sort join
  - Hash Join = Little dog (depends on smaller relation)

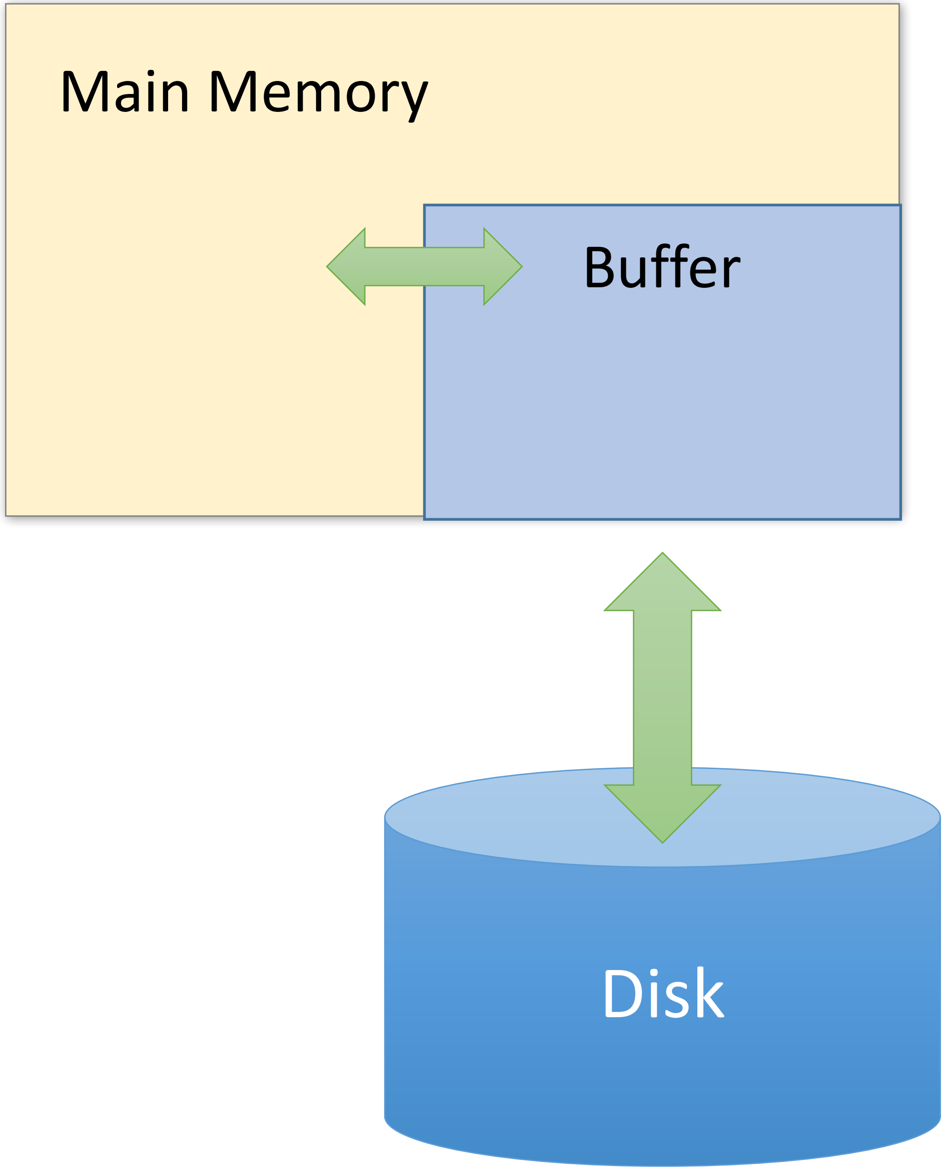
- Skew is a major factor (more on PS)

- Message: The database can compute IO costs, and these are different than a traditional system.

3. Buffer Manager

The Buffer

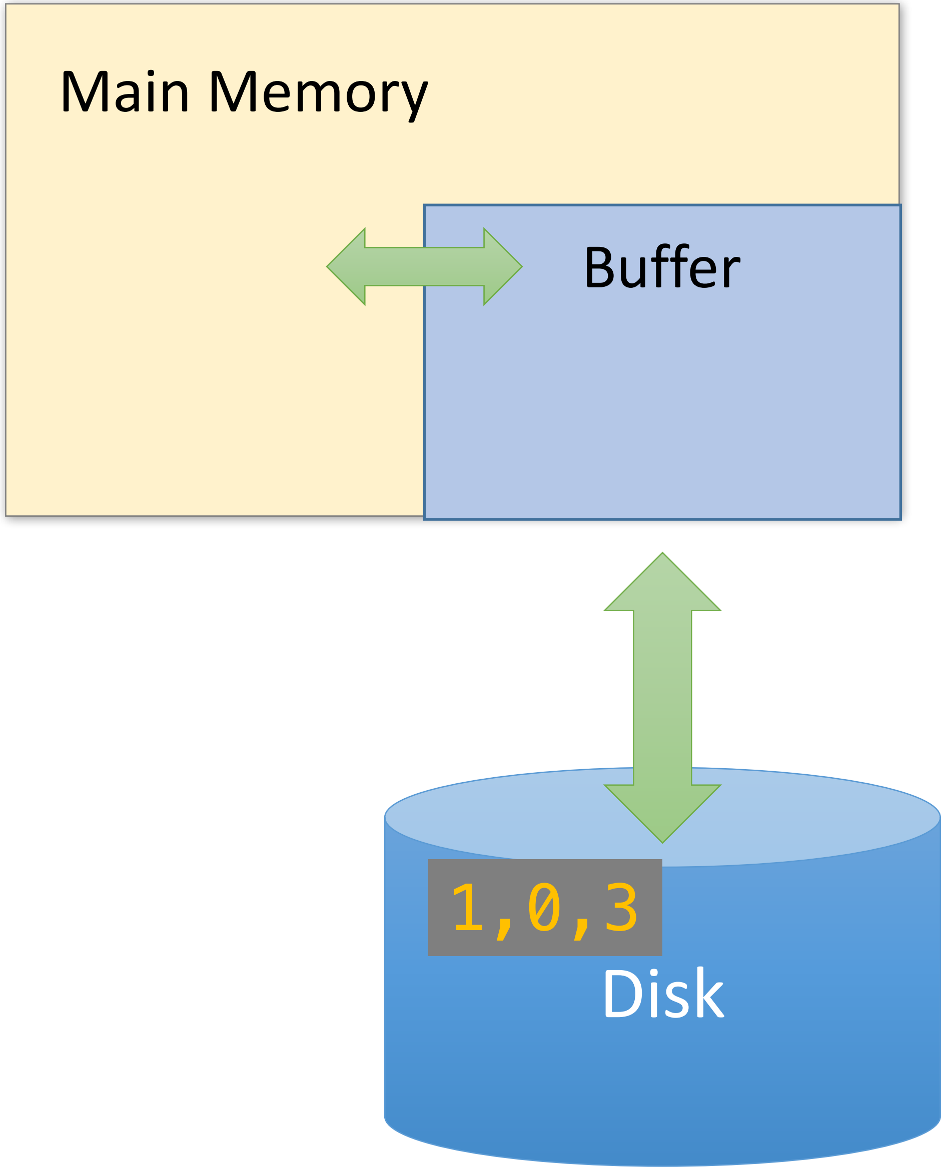
- A **buffer** is a region of physical memory used to store *temporary data*
    - *In this lecture:* a region in main memory used to store **intermediate data between disk and processes**
- *Key idea:* Reading / writing to disk is slow- need to cache data!



The (Simplified) Buffer

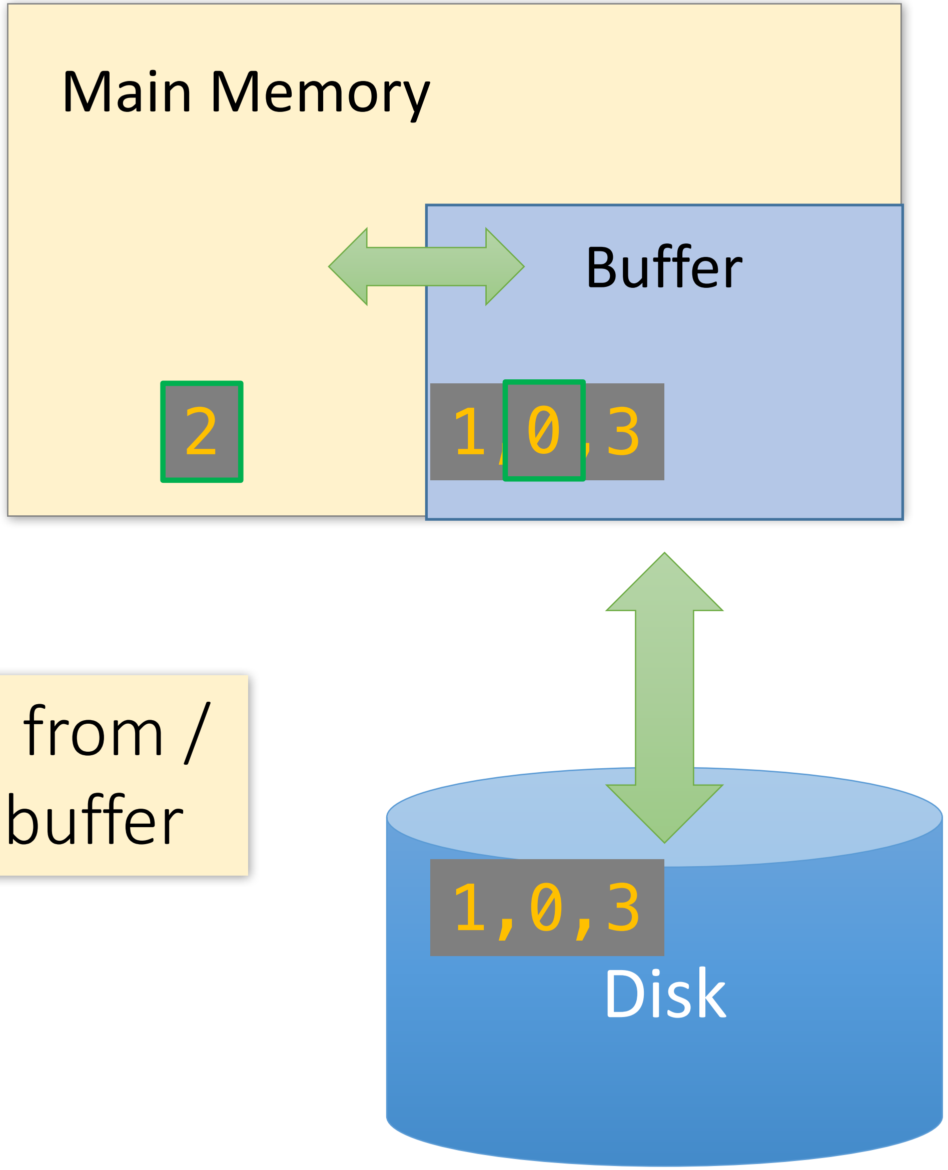
- In this class: We'll consider a buffer located in **main memory** that operates over **pages** and **files**:

  - **Read(page):** Read page from disk -> buffer *if not already in buffer*



The (Simplified) Buffer

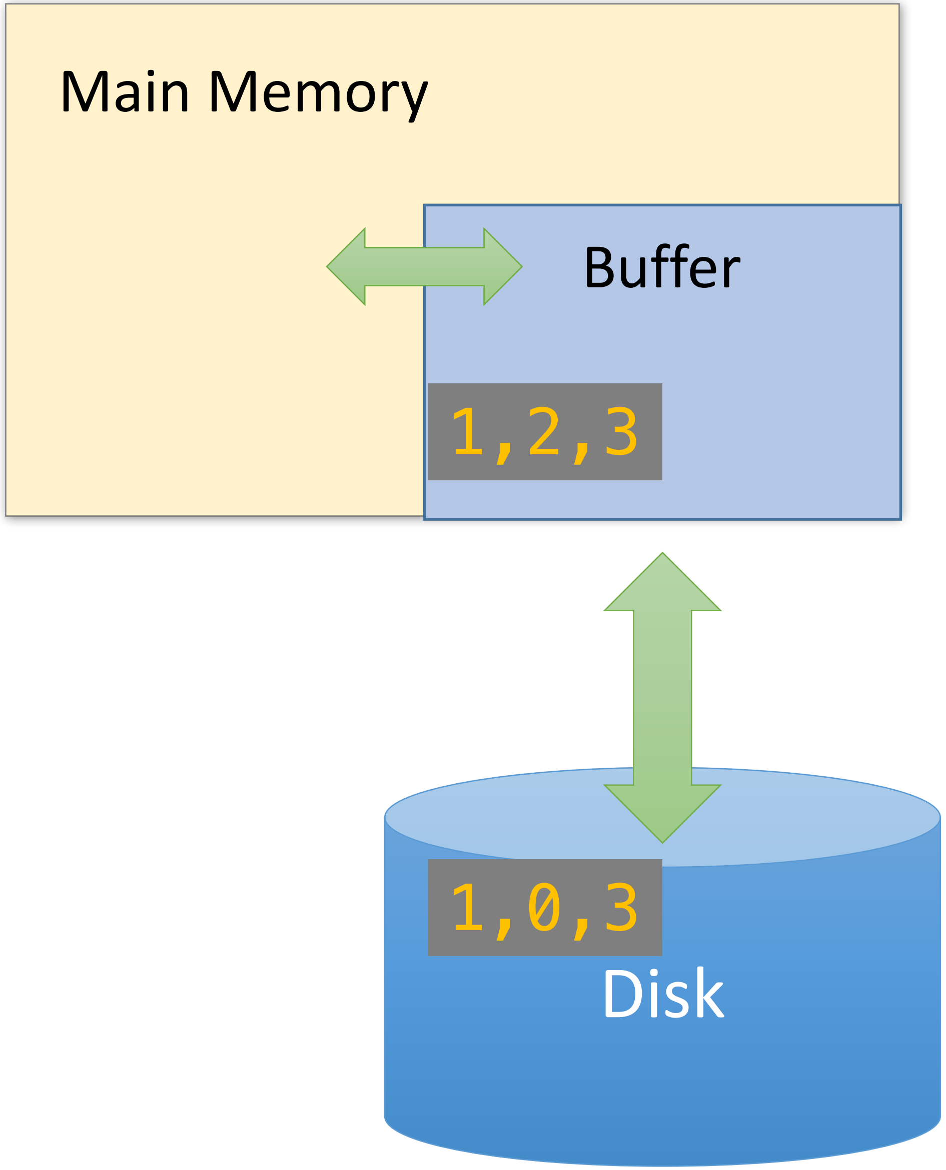
- In this class: We'll consider a buffer located in **main memory** that operates over **pages** and **files**:
  - **Read(page):** Read page from disk -> buffer *if not already in buffer*



Processes can then read from / write to the page in the buffer

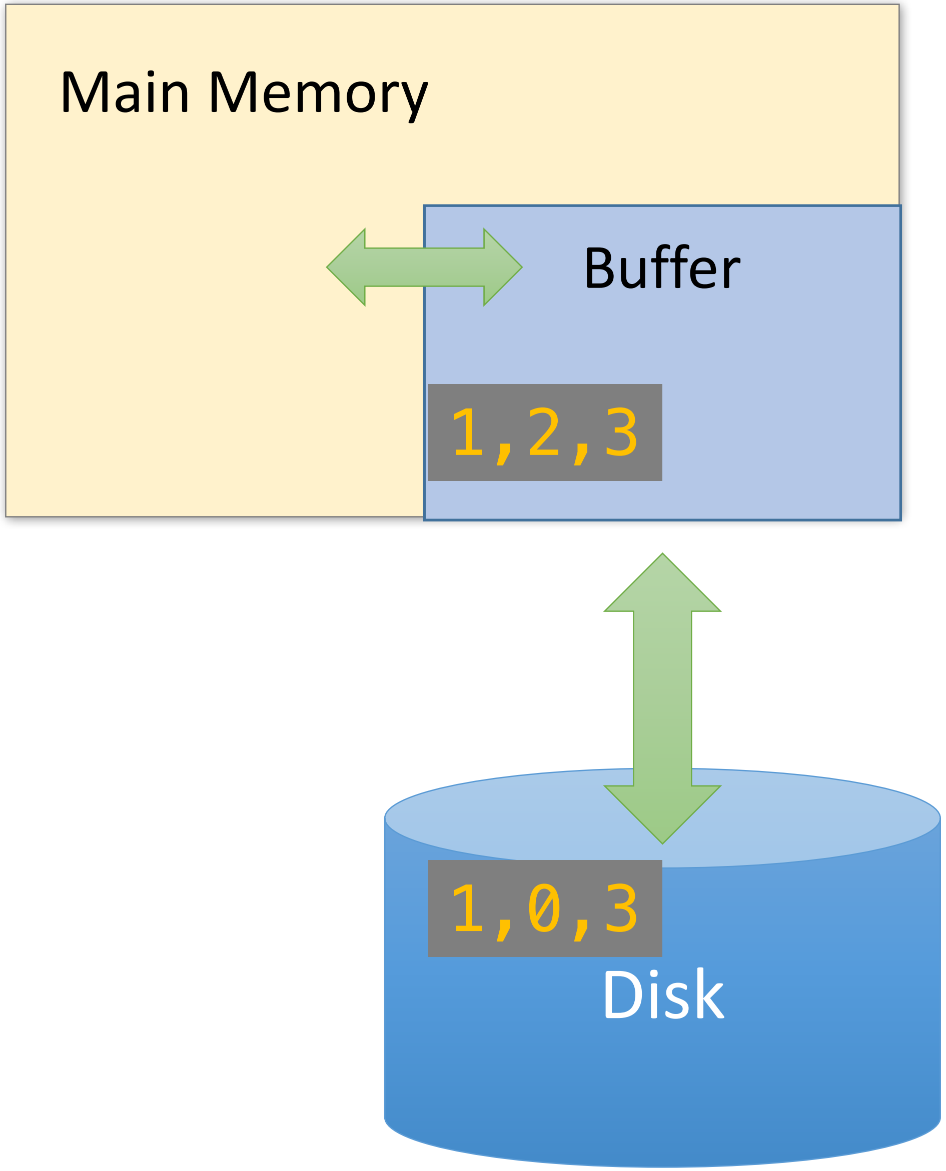
The (Simplified) Buffer

- In this class: We'll consider a buffer located in **main memory** that operates over **pages** and **files**:
  - **<u>Read(page):</u>** Read page from disk -> buffer *if not already in buffer*
  - **<u>Flush(page):</u>** Evict page from buffer & write to disk



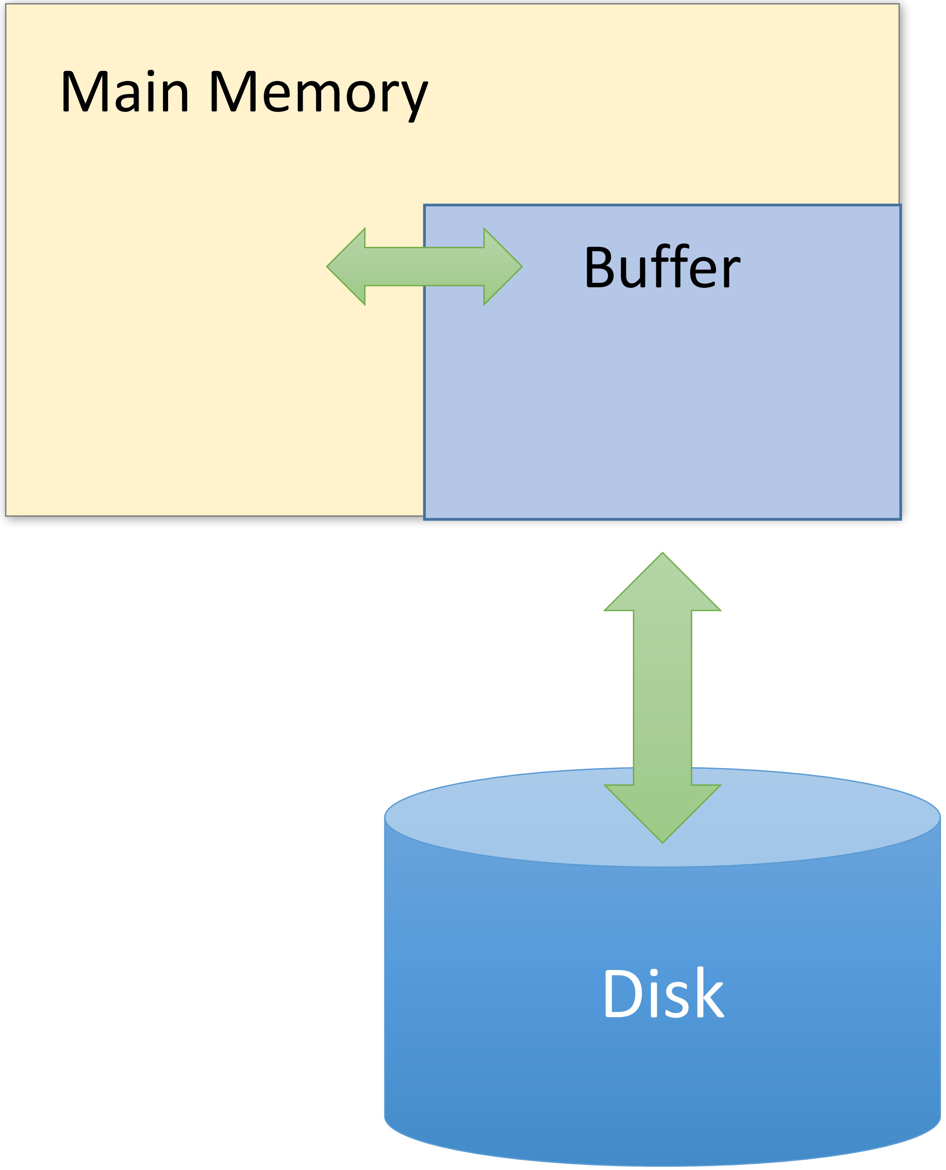
The (Simplified) Buffer

- In this class: We'll consider a buffer located in **main memory** that operates over **pages** and **files**:
  - **Read(page):** Read page from disk -> buffer *if not already in buffer*
  - **Flush(page):** Evict page from buffer & write to disk
  - **Release(page):** Evict page from buffer *without* writing to disk



Managing Disk: The DBMS Buffer

- Database maintains its own buffer
  - Why? The OS already does this…
  - DB knows more about access patterns.
    - Watch for how this shows up! (cf. *Sequential Flooding*)
  - Recovery and logging require ability to **flush** to disk.



The Buffer Manager

- A **buffer manager** handles supporting operations for the buffer:
  - Primarily, handles & executes the “replacement policy”
    - i.e. finds a page in buffer to flush/release if buffer is full and a new page needs to be read in
  - DBMSs typically implement their own buffer management routines

Activity 15!