RELATIONAL OPERATORS #2

CS 564- Fall 2018

ACKs: *Jeff Naughton, Jignesh Patel, AnHai Doan*

WHAT IS THIS LECTURE ABOUT?

Algorithms for relational operators:

- joins
- set operators
- aggregation

JOINS

JOIN OPERATOR

Algorithms for equijoin:

```
SELECT *
FROM   R, S
WHERE  R.a = S.a
```

Why can't we compute it as cartesian product?

JOIN ALGORITHMS

Algorithms for equijoin:

- nested loop join
- block nested loop join
- index nested loop join
- block index nested loop join
- sort merge join
- hash join

NESTED LOOP JOIN (1)

- for each page P_R in **R**
  - for each page P_S in **S**
    - join the tuples on P_R with the tuples in P_S

I/O = $M_R + M_S \cdot M_R$

- M_R = number of pages in **R**
- M_S = number of pages in **S**

Observe that we ignore the cost of writing the output to disk!

NESTED LOOP JOIN (2)

- Which relation should be the outer relation in the loop?
  - The smaller of the two relations
- How many buffer pages do we need?
  - only 3 pages suffice

BLOCK NESTED LOOP JOIN (1)

Assume B buffer pages

- for each block of B-2 pages from **R**
  - for each page P_S in $\boldsymbol{S}$
    - join the tuples from the block with the tuples in P_S

$$\text{I/O} = M_R + M_S \cdot \left\lceil \frac{M_R}{B-2} \right\rceil$$

BLOCK NESTED LOOP JOIN (2)

- To increase CPU efficiency, create an in-memory hash table for each block
  - what will be the key of the hash table?

- What happens if **R** fits in memory?
  - The I/O cost is only $M_R + M_S$!

NLJ VS BNLJ

Example:

- M_R = 500 pages
- M_S = 1000 pages
- 100 tuples / page
- $B = 12$

NLJ I/O = 500 + 500*1,000 = **500,500**

BNLJ I/O = $500 + \frac{500 * 1{,}000}{12-2}$ = **50,500**

The difference in I/O cost in an order of magnitude!

INDEX NESTED LOOP JOIN

S has an index on the join attribute

- for each page P_R in **R**
  - for each tuple r in **R**
    - probe the index of **S** to retrieve any matching tuples

I/O = $M_R + |R| \cdot I^*$

- I^* is the I/O cost of searching an index, and depends on the type of index and whether it is clustered or not

BLOCK INDEX NESTED LOOP JOIN

- for each block of B-2 pages in **R**
  - sort the tuples in the block
  - for each tuple r in the block
    - probe the index of **S** to retrieve any matching tuples

- Why do we need to sort here?

SORT MERGE JOIN

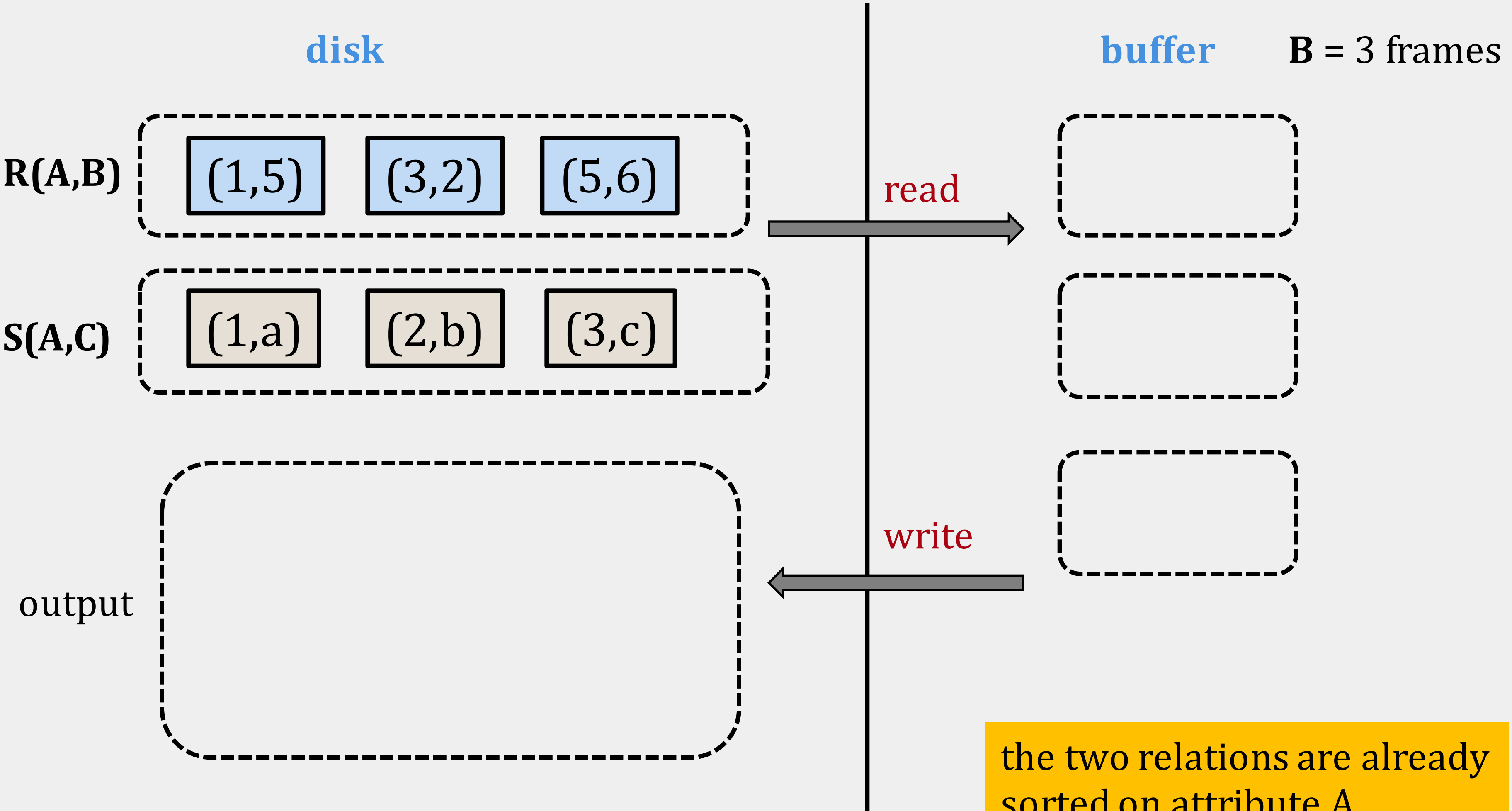
SORT MERGE JOIN: BASIC VERSION

The basic version:

- sort **R** and **S** on the join attribute (using external merge sort)
- read the sorted relations in the buffer and merge

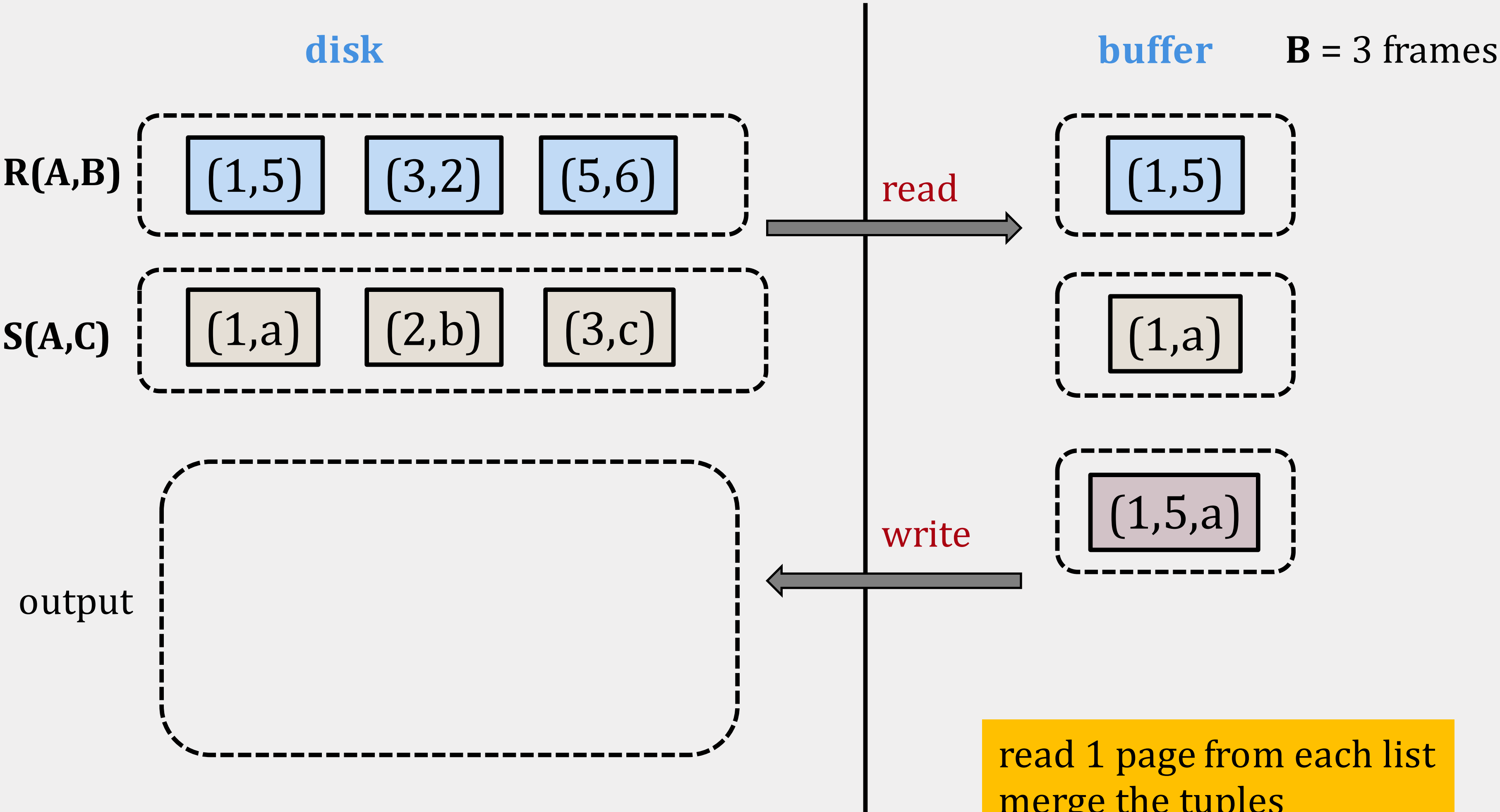
If **R**, **S** are already sorted on the join attribute we can skip the first step!

READ AND MERGE



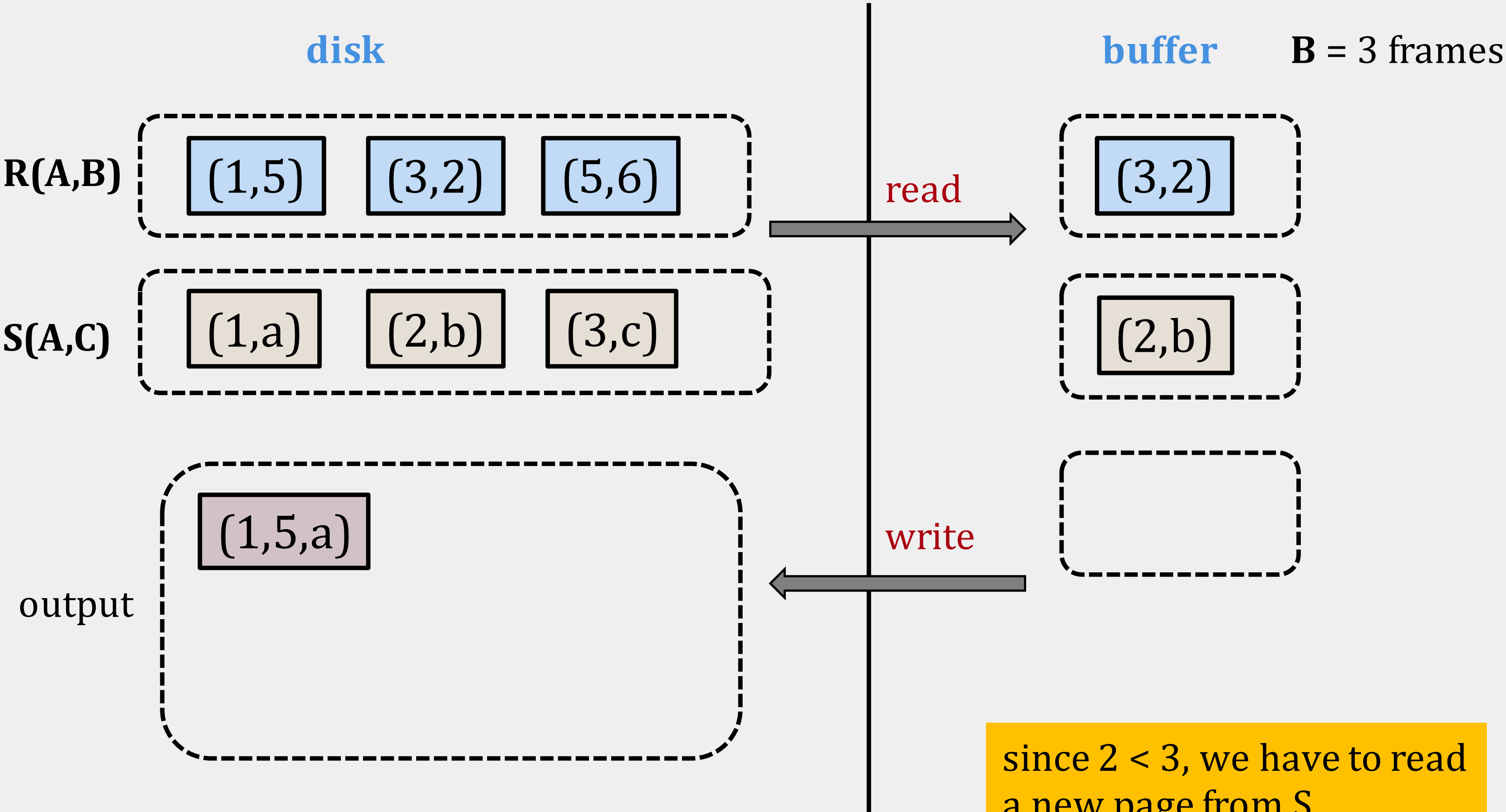
the two relations are already sorted on attribute A

READ AND MERGE



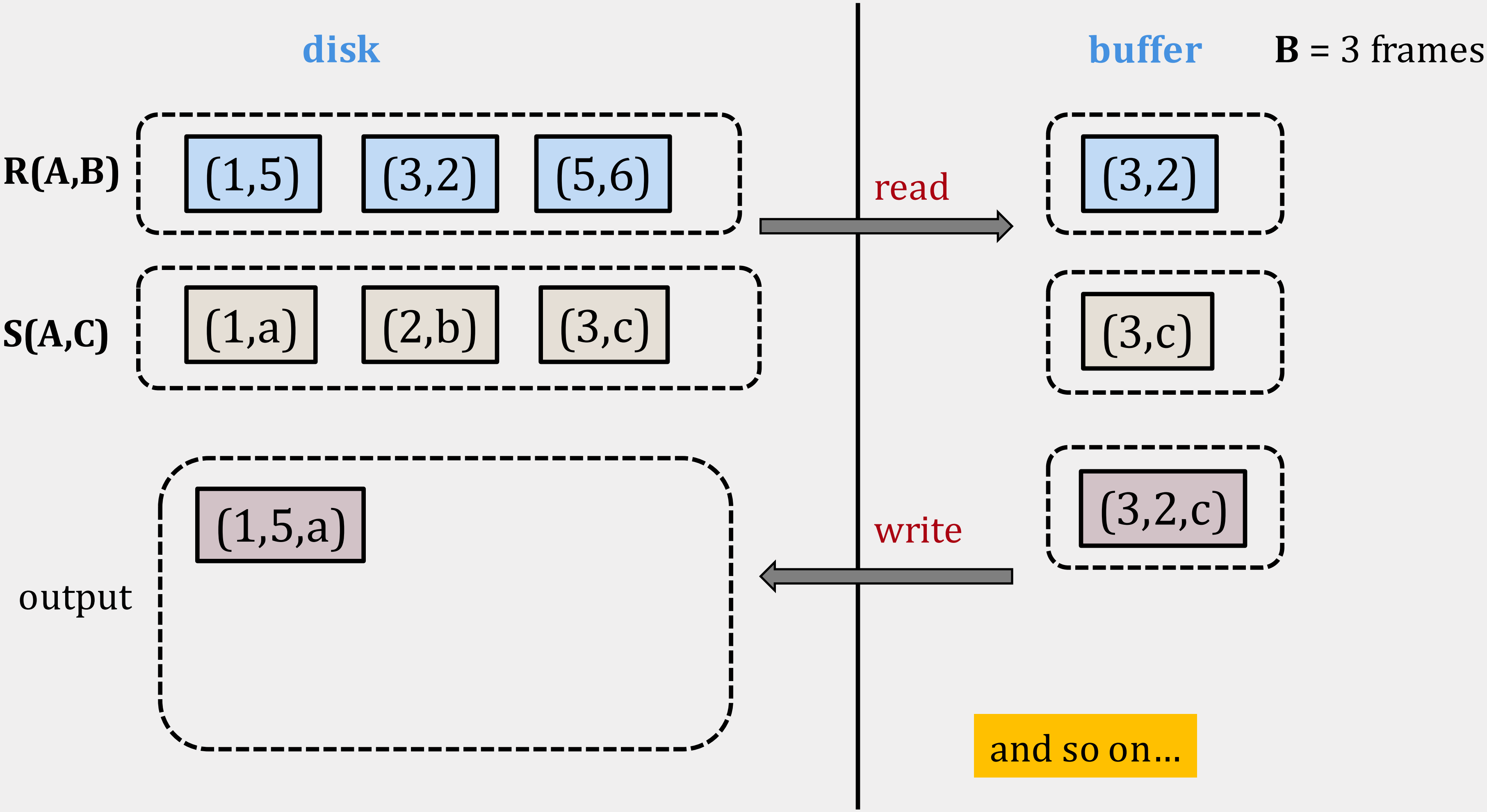
read 1 page from each list
merge the tuples

READ AND MERGE

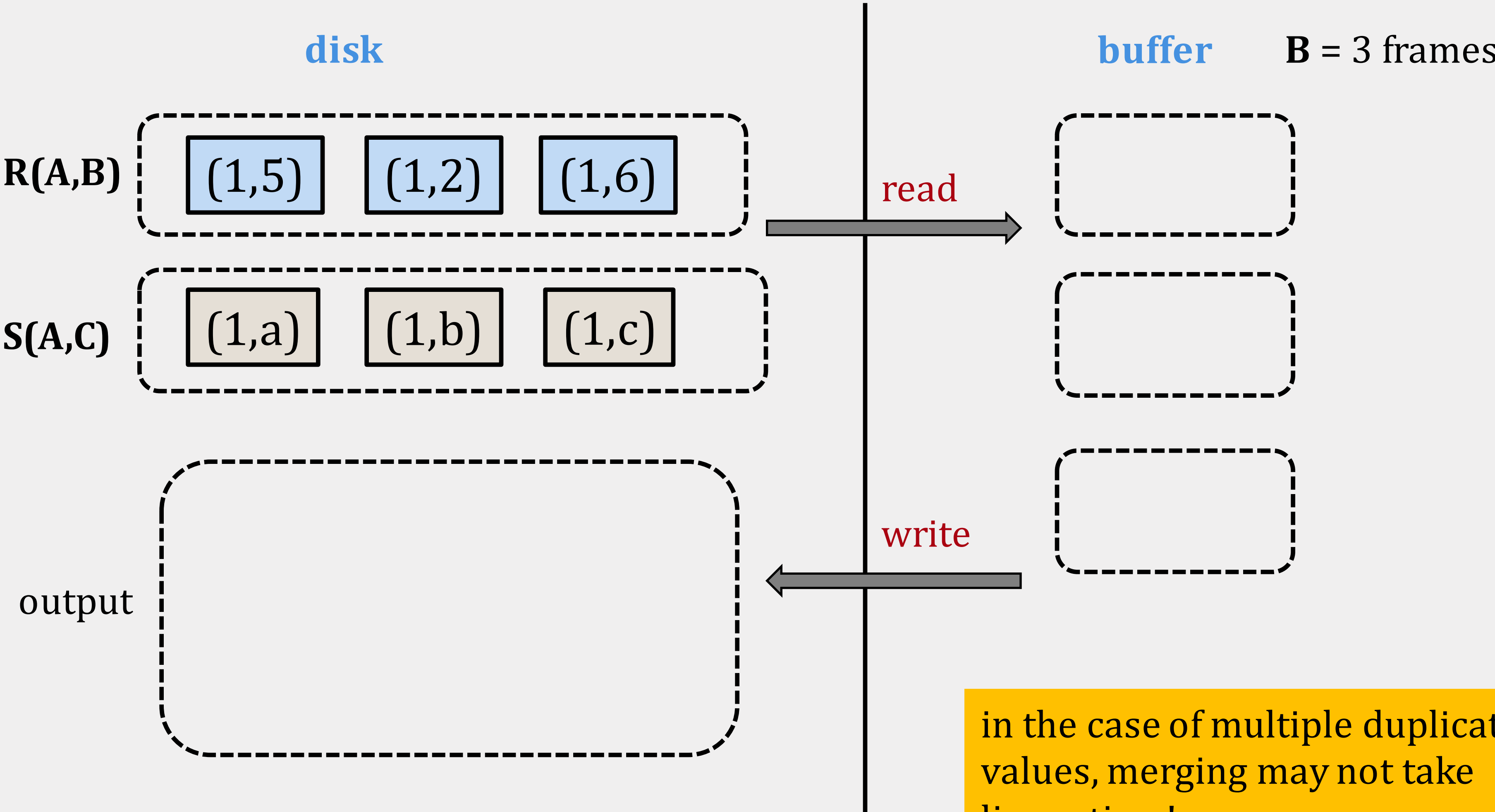


since 2 < 3, we have to read a new page from S

READ AND MERGE

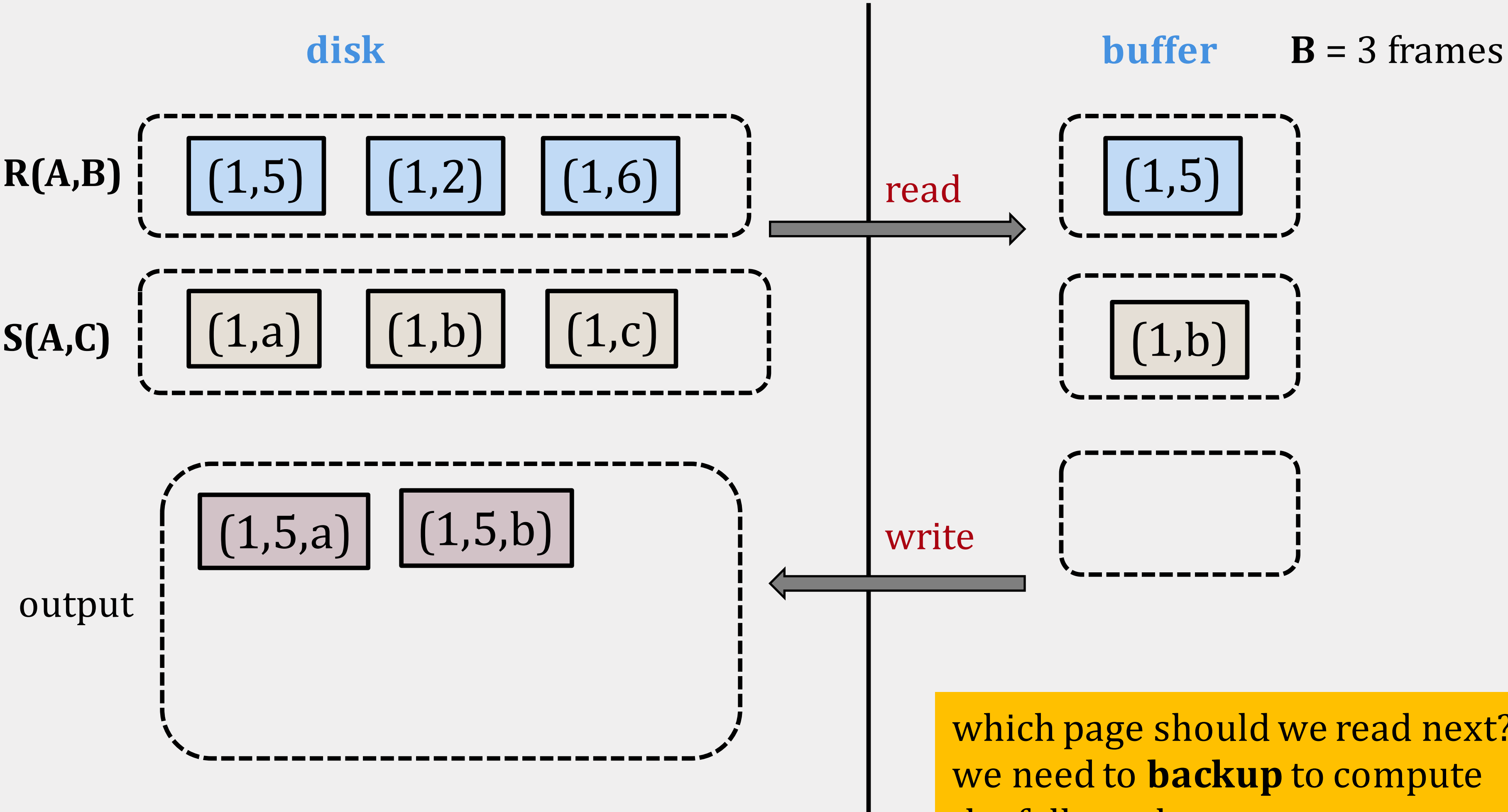


SORTING WITH DUPLICATES



in the case of multiple duplicate values, merging may not take linear time!

SORTING WITH DUPLICATES



which page should we read next? we need to **backup** to compute the full result

SMJ: I/O COST

- If there is no backup, the I/O cost of read + merge is only $M_R + M_S$
- If there is backup, in the worst case the I/O cost could be $M_R * M_S$
  - this happens when there is a *single* join value

Total I/O cost ~ $sort(R) + sort(S) + \ M_R + M_S$

SORT MERGE JOIN: OPTIMIZED

- Generate sorted runs of size $\sim 2B$ for **R** and **S**
- Merge the sorted runs for **R** and **S**
  - while merging check for the join condition and output the join tuples

I/O cost $\sim 3(M_R + M_S)$

But how much memory do we need for this to happen?

SMJ: MEMORY ANALYSIS

- In the first phase, we create runs of length ~2B
- Hence, the number of runs is $\frac{M_R+M_S}{2B}$
- To perform a k-way merge, we need k+1 buffer pages, so:

$$\frac{M_R+M_S}{2B} \leq B - 1 \text{ or } B^2 \geq \max\{M_S, M_R\}$$

If B^2 is larger than the **maximum** number of pages of the two relations, then SMJ has I/O cost ~ $3(M_R + M_S)$

HASH JOIN

HASH FUNCTION REFRESHER

- We will use a hash function *h* to map values of the join attribute (A) into buckets [1, B-1]
- Tuple *t* is then hashed to bucket *h(t.A)*

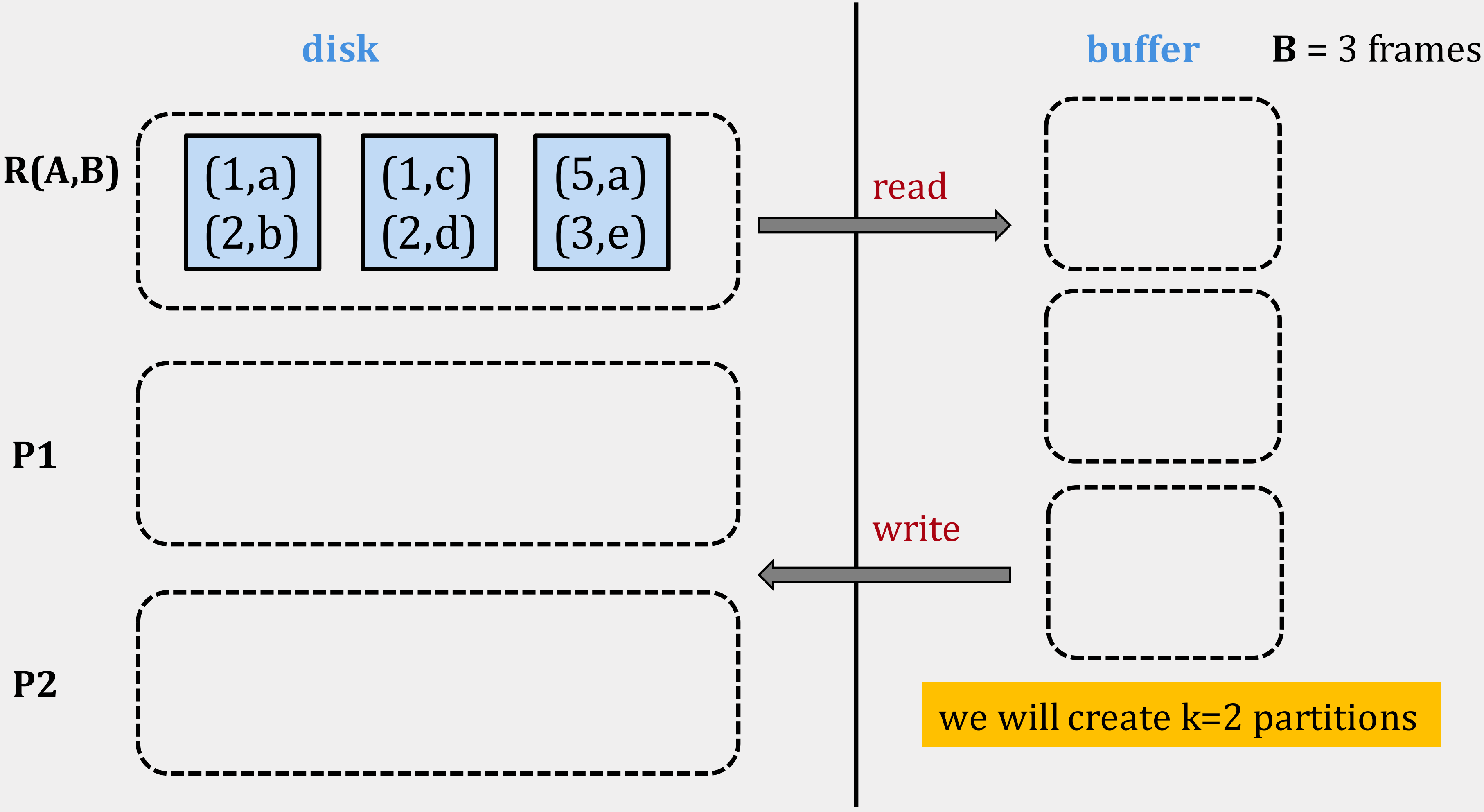
- A hash **collision** occurs when x != y but h(x) = h(y)
- Note however that it will **never** happen that x = y but h(x) != h(y)

HASH JOIN: OVERVIEW

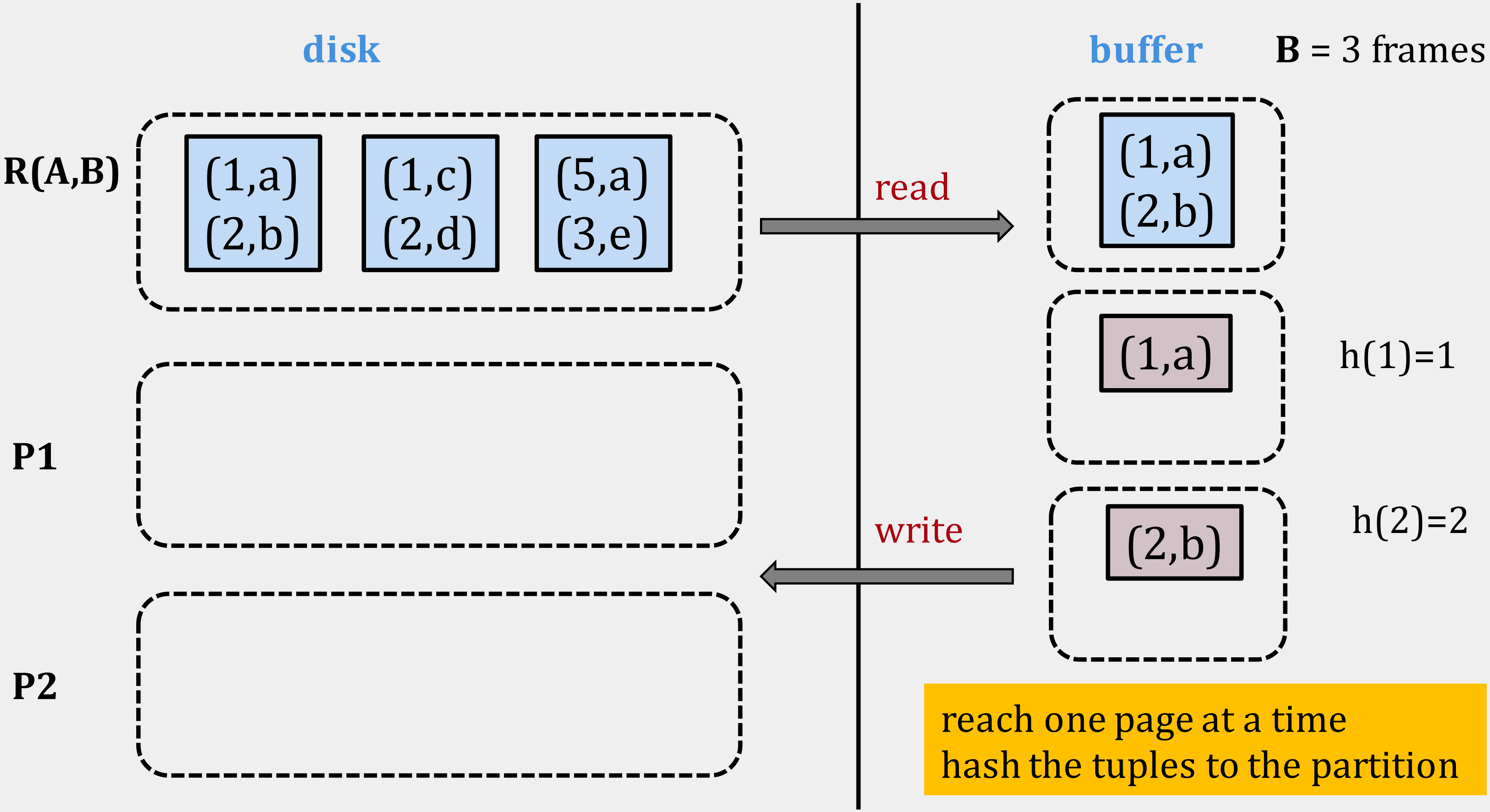
Start with a hash function h on the join attribute

- **Partition phase:** partition **R** and **S** into k partitions using h
- **Matching phase:** join each partition of **R** with the corresponding (same hash value) partition of **S** using BNLJ

PARTITION PHASE

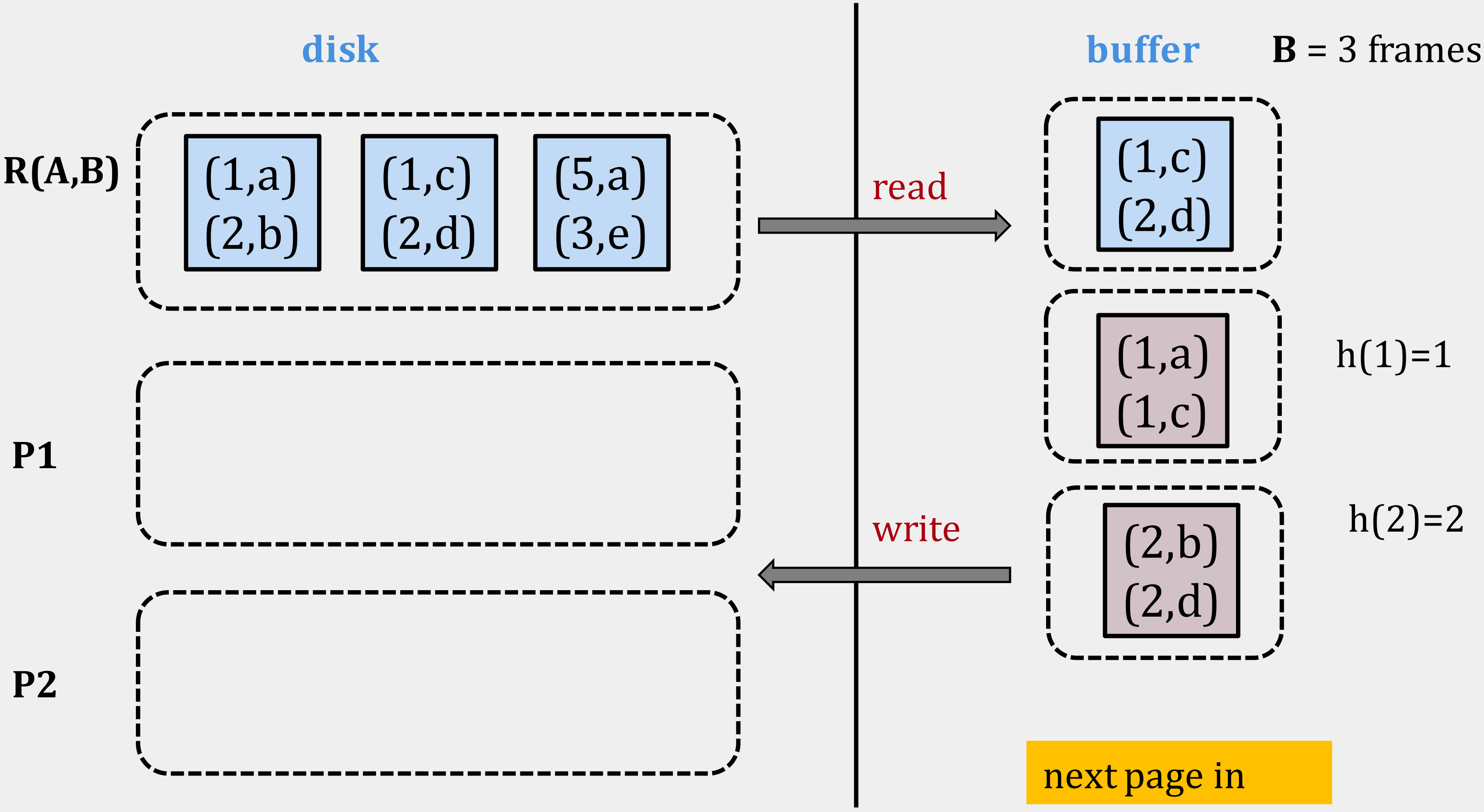


PARTITION PHASE

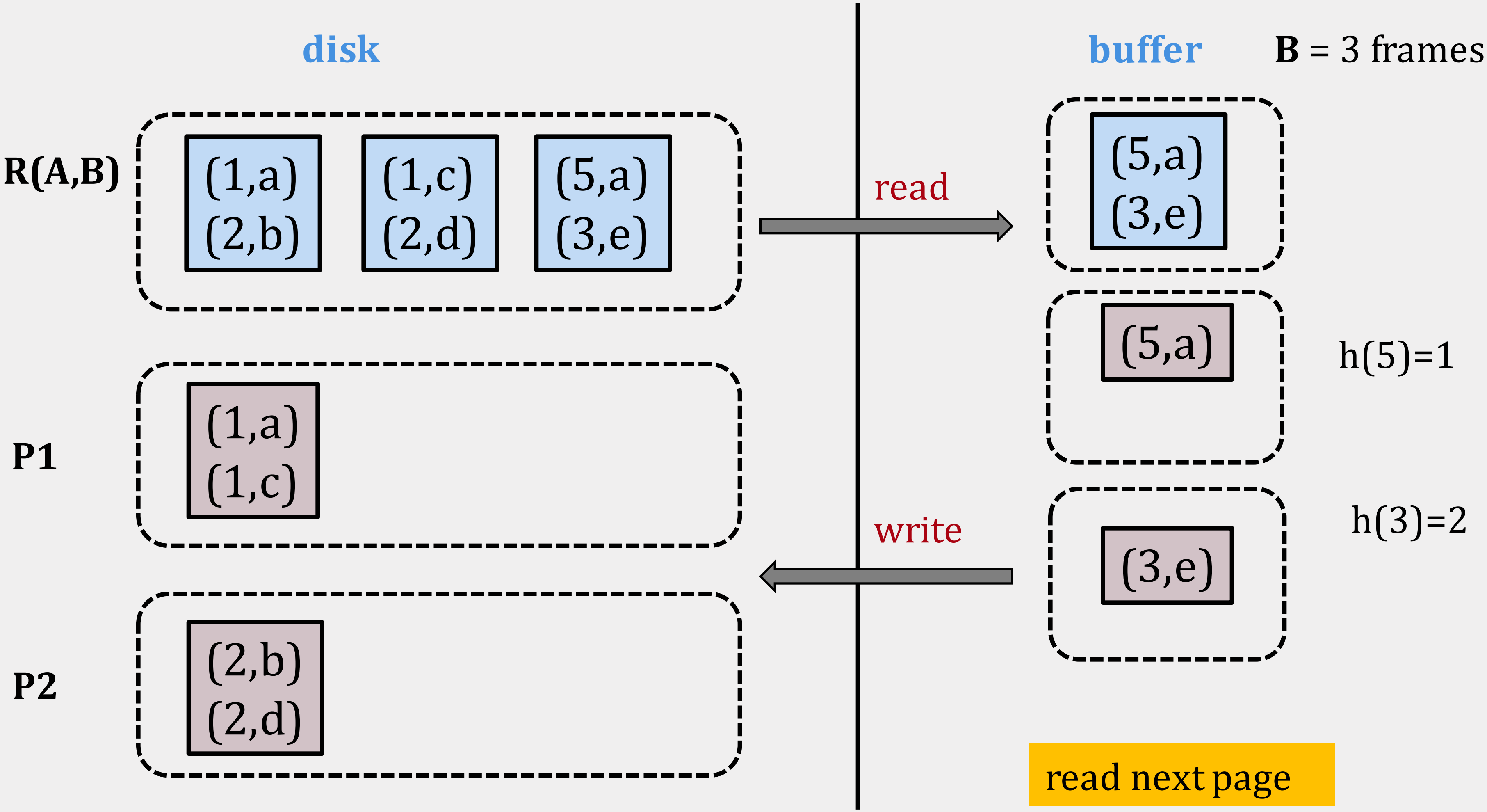


reach one page at a time
hash the tuples to the partition

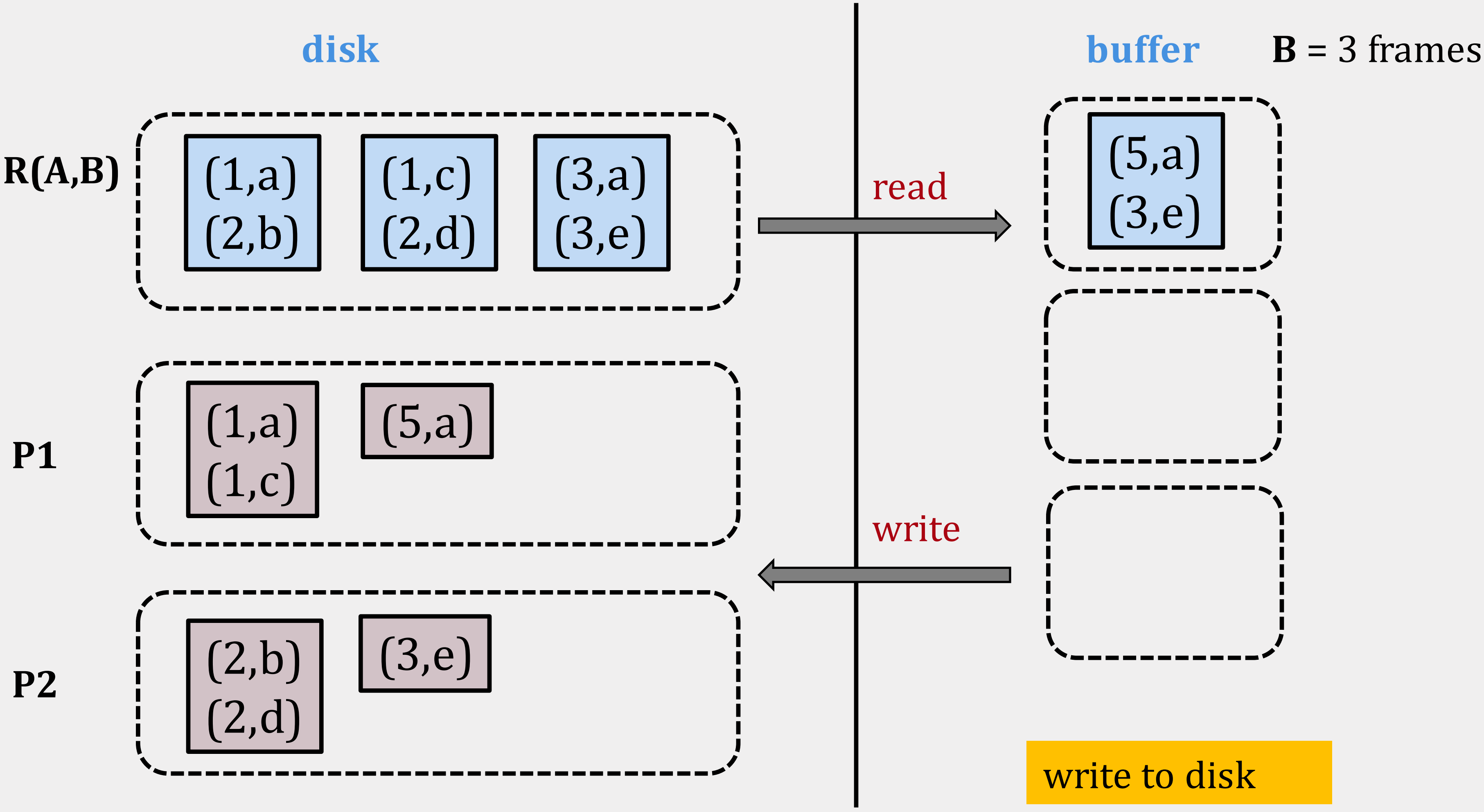
PARTITION PHASE



PARTITION PHASE



PARTITION PHASE



BUCKET SIZE

- We can create up to k = B-1 partitions in one pass
- How big are the buckets we create?
  - Ideally, each bucket has ~ M/(B-1) pages
  - but hash collisions can occur!
  - or we may have many duplicate values on the join attribute (skew)

- In the matching phase, we join two buckets from R, S with the same hash value
  - We want to do this in linear time using BNLJ, so we must guarantee that each bucket from one of the two relations is at most B-1 pages

HJ: I/O COST

- Suppose $M_R \leq M_S$
- The partition phase gives buckets of size ~ M_R/B
- To make BNLJ run in one pass we need to make sure that:

$$\frac{M_R}{B} \leq B - 2 \text{ or equivalently: } B^2 \geq M_R$$

If B^2 is larger than the **minimum** number of pages of the two relations, then HJ has I/O cost ~ $3(M_R + M_S)$

COMPARISON OF JOIN ALGORITHMS

Hash Join **vs** Block Nested Loop Join

- the same if smaller table fits into memory
- otherwise, hash join is much better

COMPARISON OF JOIN ALGORITHMS

Hash Join **vs** Sort Merge Join

- Suppose $M_R > M_S$
- To do a two-pass join, SMJ needs $B > \sqrt{M_R}$
  - the I/O cost is: $3(M_R + M_S)$
- To do a two-pass join, HJ needs $B > \sqrt{M_S}$
  - the I/O cost is: $3(M_R + M_S)$

GENERAL JOIN CONDITIONS

- Equalities over multiple attributes
  - e.g., *R.sid=S.sid* **and** *R.rname=S.sname*
  - for Index Nested Loop
    - index on <*sid, sname*>
    - index on *sid* or *sname*
  - for SMJ and HJ, we can sort/hash on combination of join attributes

GENERAL JOIN CONDITIONS

- Inequality conditions
  - e.g., *R.rname* < *S.sname*
  - For BINL, we need (clustered) B+ tree index
  - SMJ and HJ not applicable
  - BNLJ likely to be the winner (why?)

Set Operations & Aggregation

SET OPERATIONS

- Intersection is a special case of a join
- Union and difference are similar
- Sorting:
  - sort both relations (on *all attributes*)
  - merge sorted relations eliminating duplicates
- Hashing:
  - partition R and S
  - build in-memory hash table for partition R_i
  - probe with tuples in S_i, add to table if not a duplicate

AGGREGATION: SORTING

- sort on group by attributes (if any)
- scan sorted tuples, computing running aggregate
  - max/min: max/min
  - average: sum, count
- when the group by attribute changes, output aggregate result
- **cost** = sorting cost

AGGREGATION: 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 happens if we have many groups?

AGGREGATION: INDEX

- Without grouping
  - Can use B+ tree on aggregate attribute(s)
- 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, the data entries/tuples are retrieved in group-by order