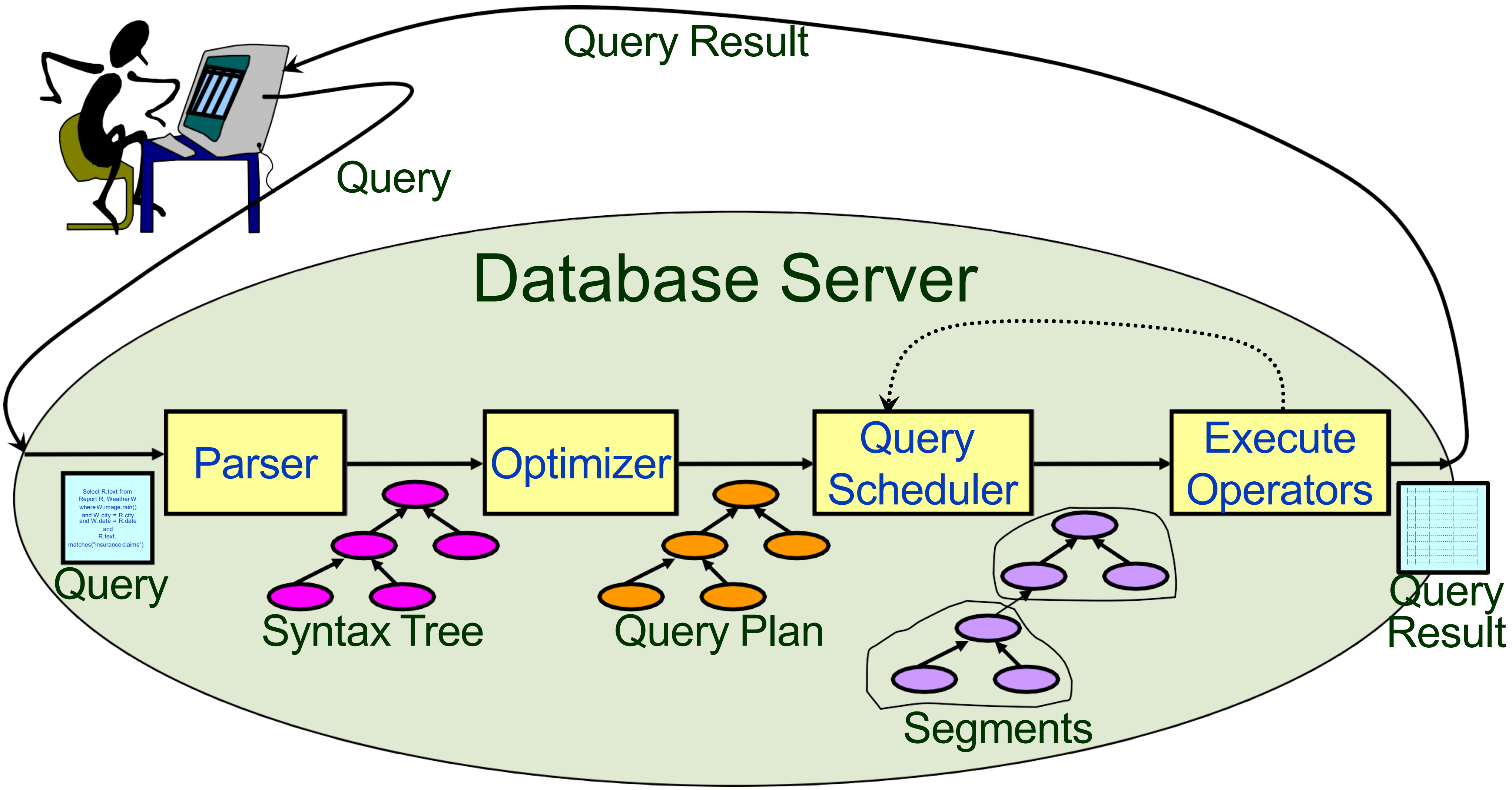
JOINS and QUERY PROCESSING

Life Cycle of a Query



```
Select R.text from
Report R, Weather W
where W.image.rain()
and W.city = R.city
and W.date = R.date
and
R.text.
matches("insurance claims")
```

Problem Statement

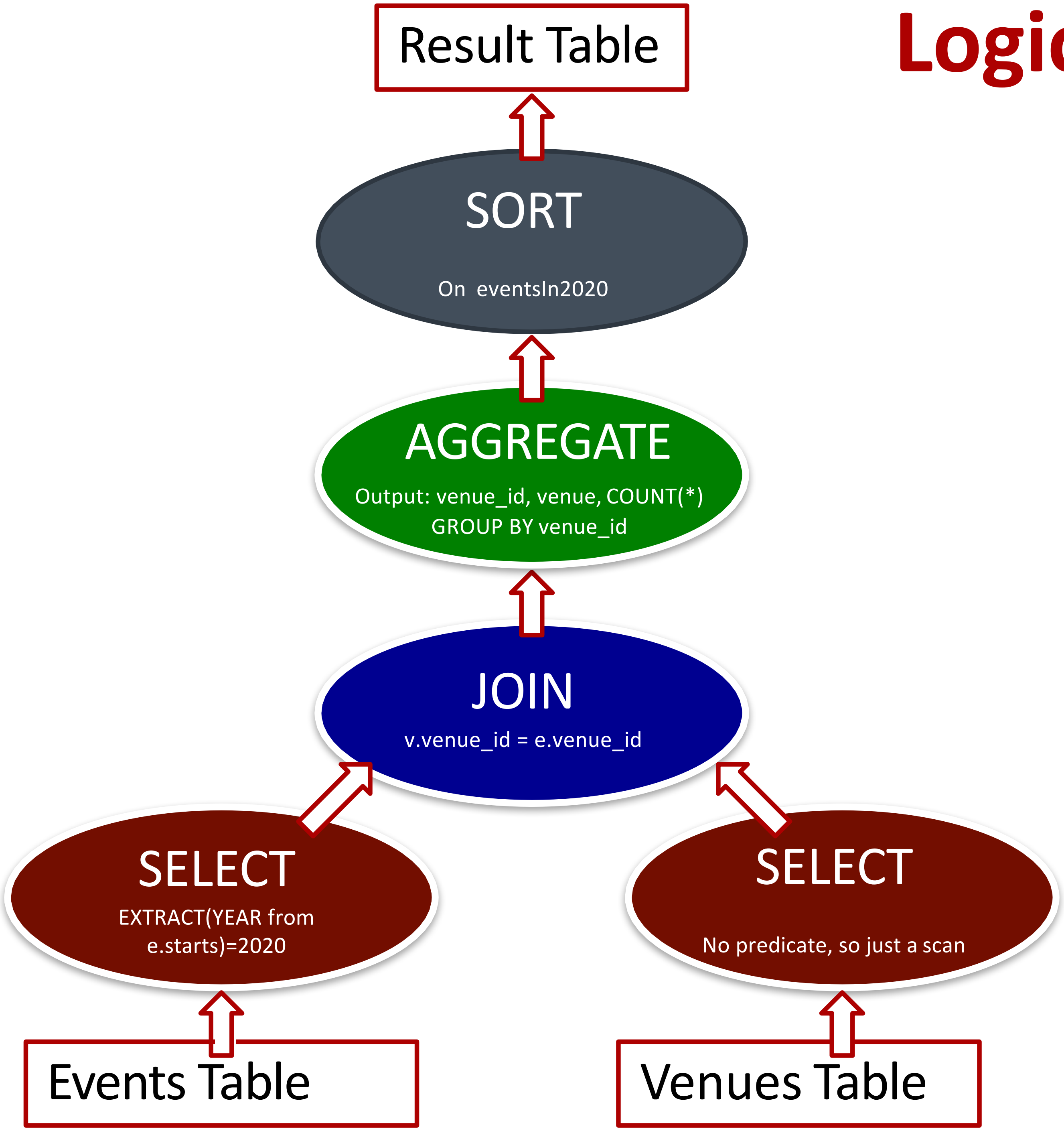
- Run the following query:

```
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 Agricultutre	4
…		…

Logical Query Plan

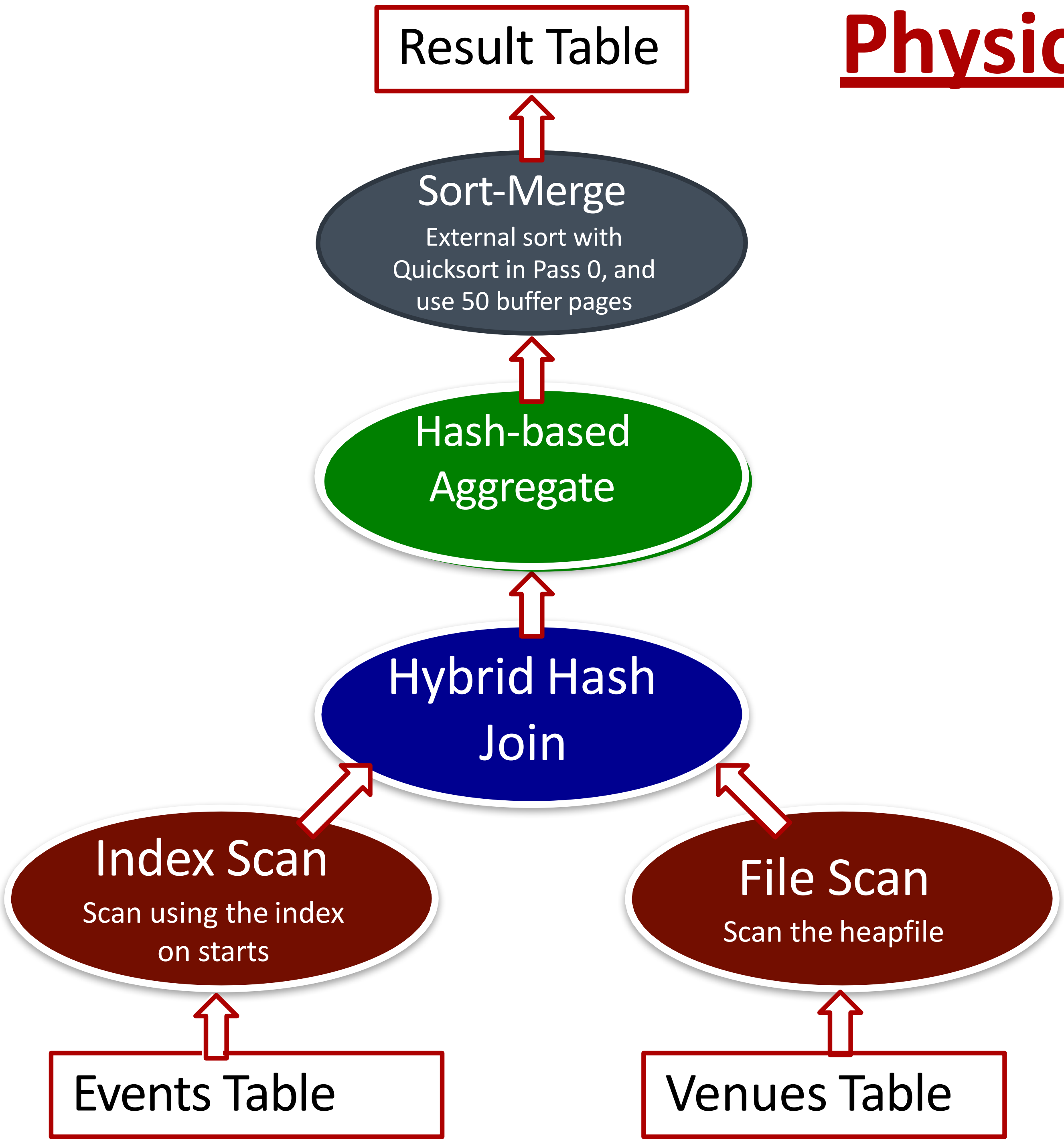


Here the ovals are logical operators. There are many different algorithms for each of these operators.

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 = number of selected tuples/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

# data pages
= 1M/100 = 10K pages
# leaf idx pgs
= 1M / (500 * 0.67)
~ 3K pages

	1% Selection	10% Selection
Clustered	30 + 100	300 + 1000
Non-Clustered	30 + 10,000	300 + 100,000
NC + Sort Rids	30 + (~ 10,000)	300 + (~ 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 and p2) or p3
- Convert to Conjunctive Normal Form(CNF)
  (p1 or p3) and (p2 or p3)
- An index ***matches*** a predicate
  - Index can be used to evaluate the predicate

Index Matching

- B+-tree index on <a, b, c>

		Hash Idx
– a=5 and b= 3?	●	●
– a > 5 and b < 3	●	●
– b=3	●	●
– a=7 and b=5 and c=4	●	●
and **d>4**	**(primary conjunct)**	
– a=7 and c=5	●	●

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

Index Matching

- A predicate could match more than 1 index
- Hash index on <a, b> and B+tree index on <a, c>
  - a=7 and b=5 and c=4   Which index?
  - Option1: Use either (or a file scan!)
    - Check selectivity of the primary conjuct
  - 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 <a> and Hash index on <b>
  - a=7 **or** b>5 Which index?
  - Neither! File scan required for b>5
- Hash index on <a> and B+-tree on <b>
  - 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 <a> and B+-tree on <b>
  - (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 <u>and</u> eliminate duplicates
- Algorithms
  – Sorting: Sort on <u>all</u> 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 used^2 > |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 Thousand IOs ~= <u>1.4 hours</u>**

BNLJ: Cost = 500 + $\frac{500*1000}{10}$ = **50 *Thousand* IOs ~= <u>0.14 hours</u>**

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

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:

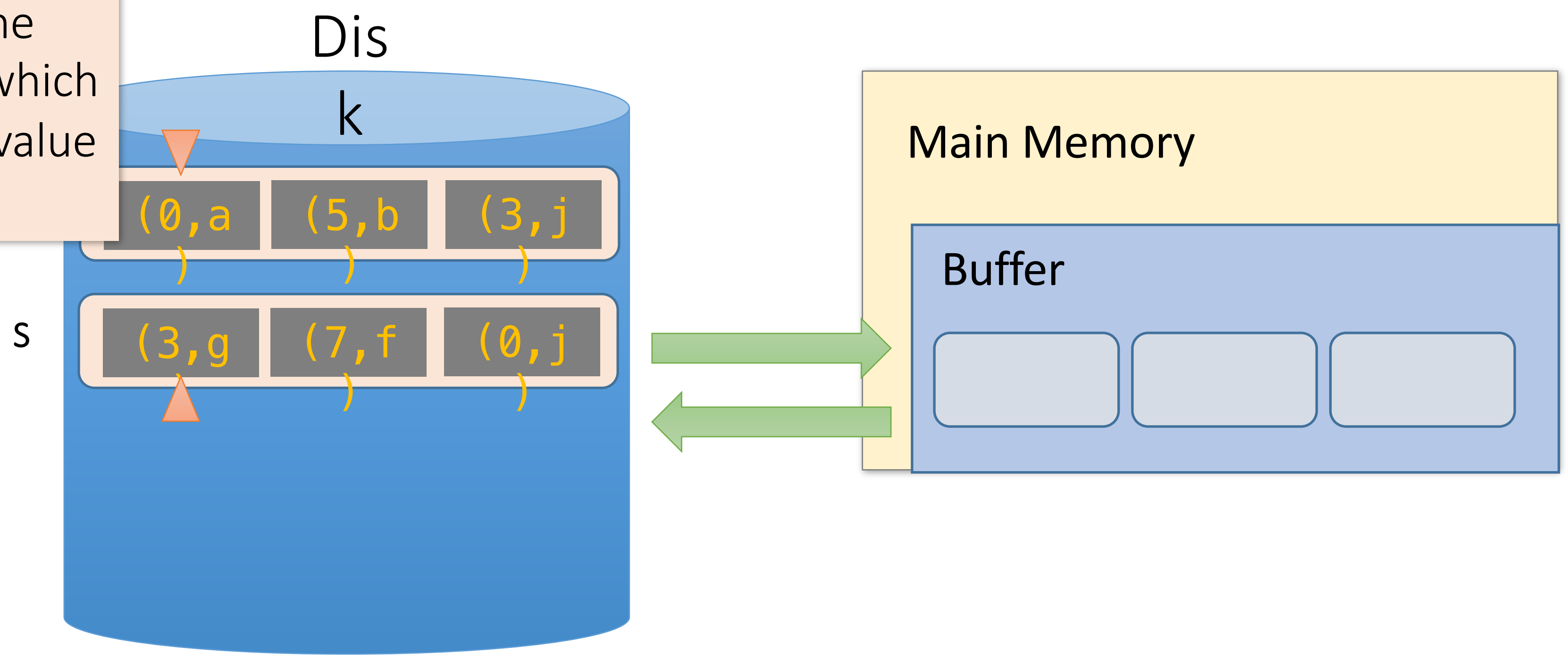
•There is no need to hold either table in memory (unlike the hash joins)

•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: $\mathrm{R} \bowtie S\ on\ A$ with 3 page buffer

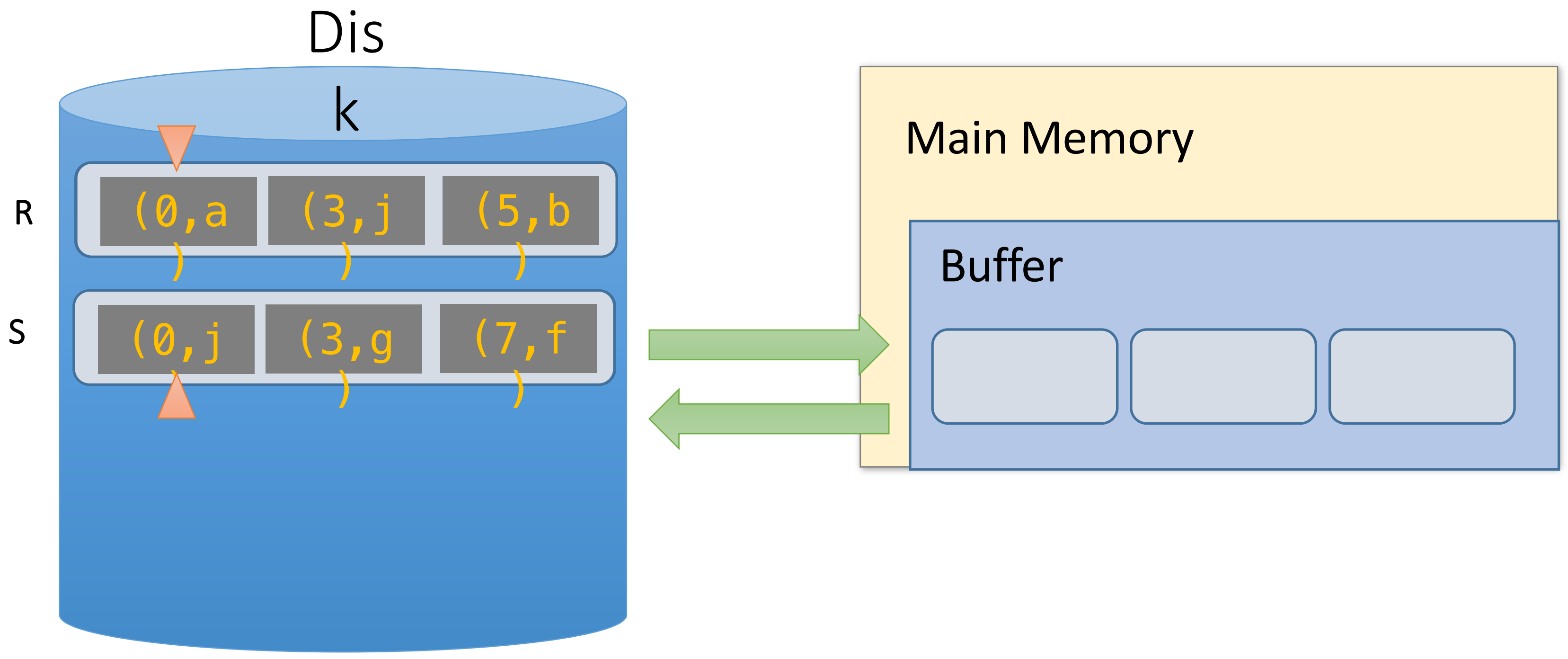
- For simplicity: Let each page be ***one tuple***, and let the first value be A

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



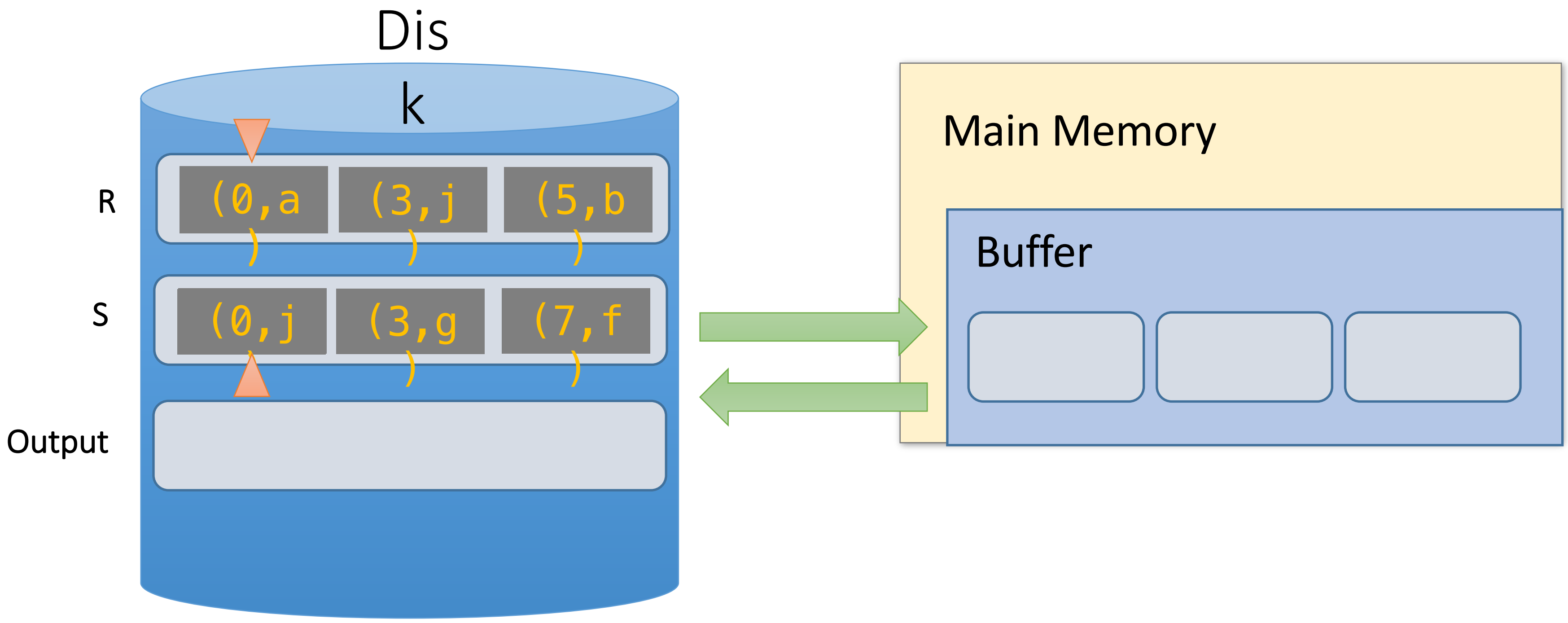
SMJ Example: $\mathrm{R} \bowtie S\ on\ A$ with 3 page buffer

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



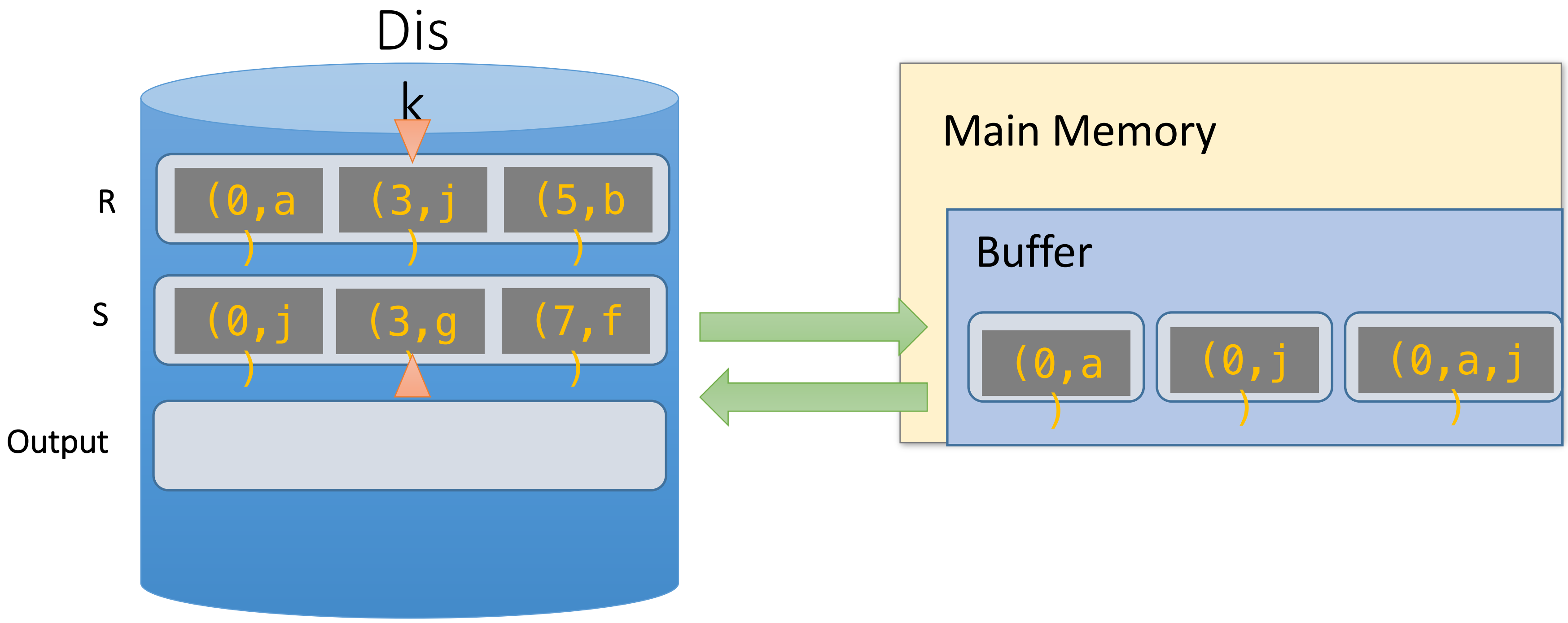
SMJ Example: $\mathrm{R} \bowtie S\ on\ A$ with 3 page buffer

2. Scan and “merge” on join key!



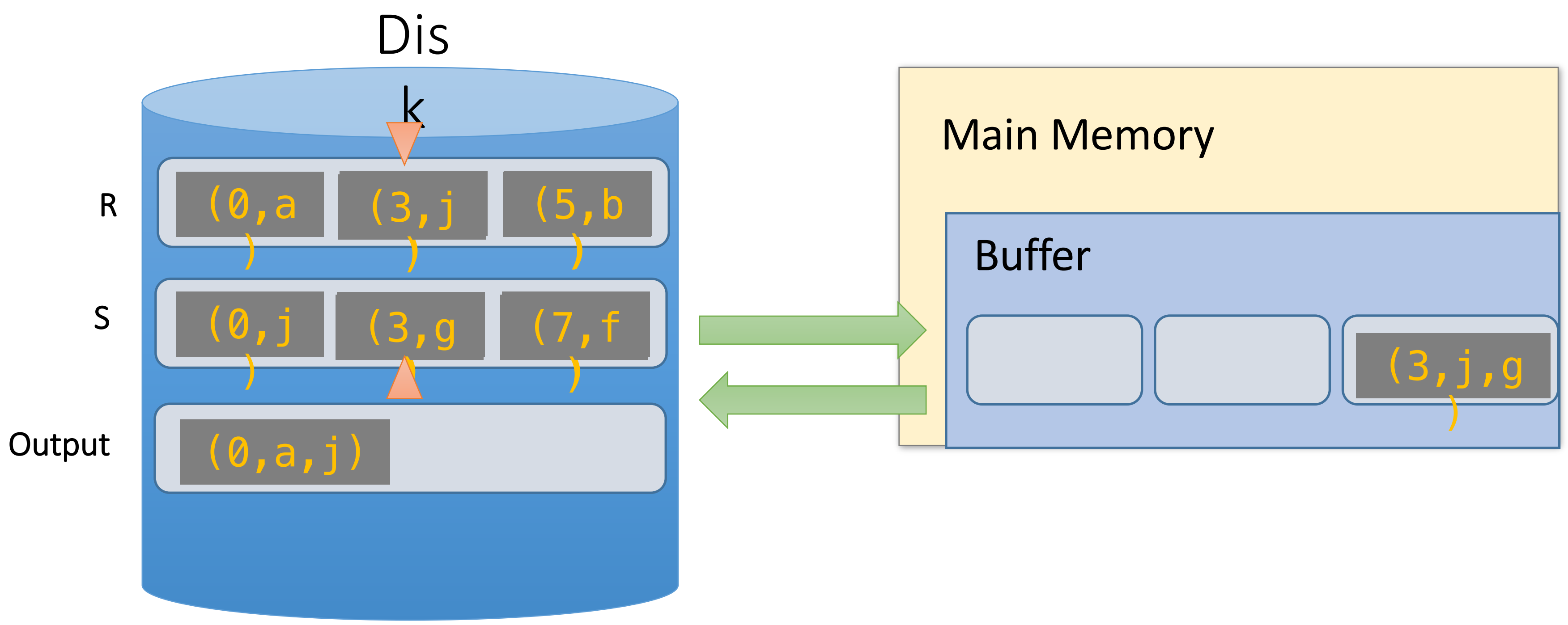
SMJ Example: $\mathrm{R} \bowtie S\ on\ A$ with 3 page buffer

2. Scan and “merge” on join key!



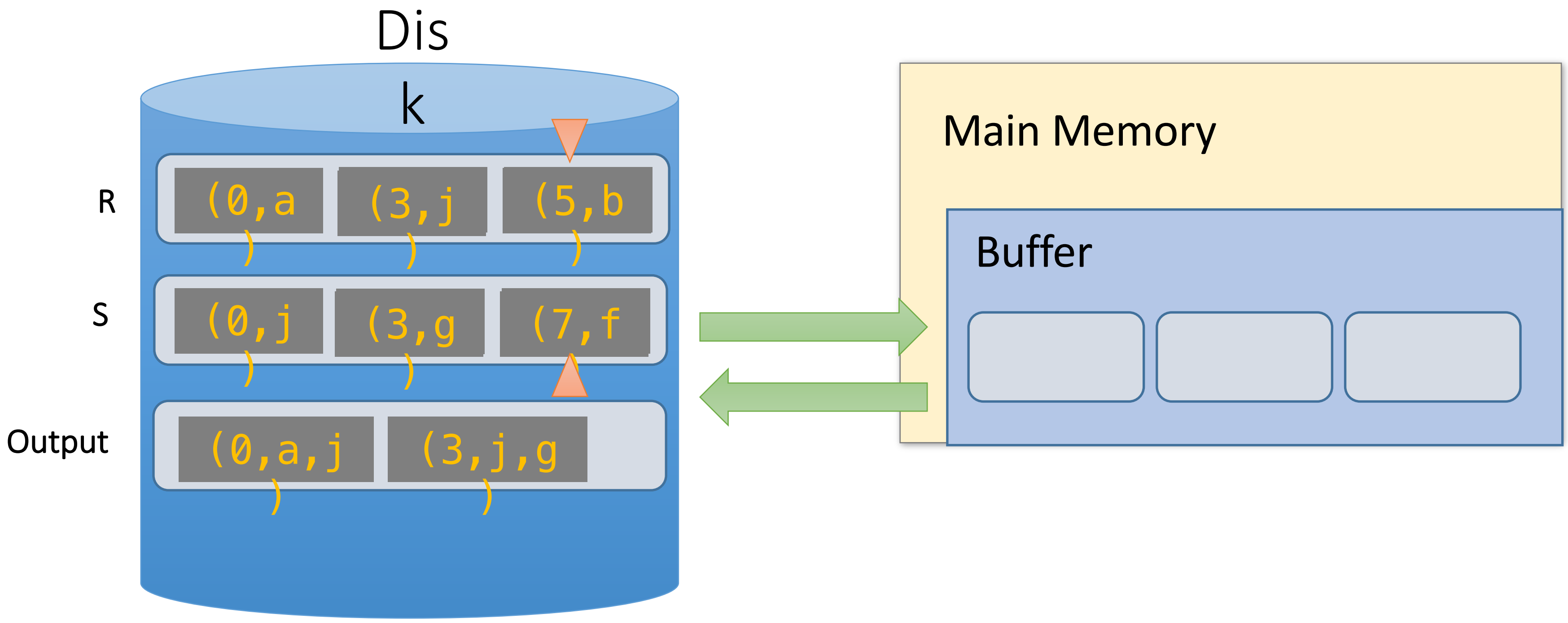
SMJ Example: $\mathrm{R} \bowtie S\ on\ A$ with 3 page buffer

2. Scan and “merge” on join key!



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

2. Done!



SMJ: Total cost

- Cost of SMJ is **cost of sorting** R and S…
- Plus the **cost of scanning**: ~|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**: ~|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 h1 on the join key
2. Partition E into P partitions, using a hash function h1 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 h1

With B buffer pages, # partitions is ~B (for each V and E)

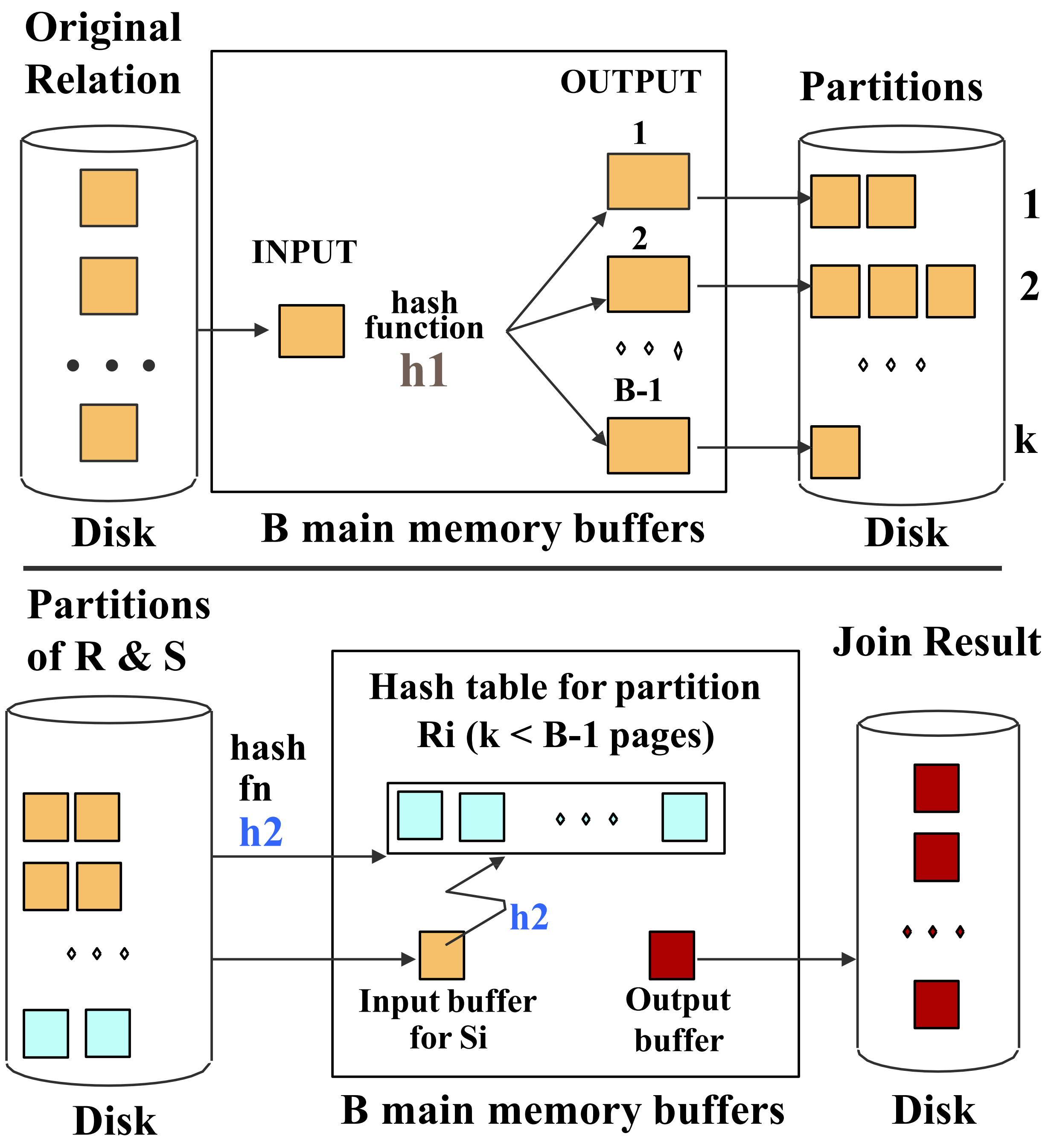
Each partition of V must fit in memory (with its hash table).

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:

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

Hash-Join



Original Relation
OUTPUT
Partitions
1
2
INPUT
hash function
h1
B-1
1
2
k
Disk
B main memory buffers
Disk
Partitions of R & S
Join Result
Hash table for partition Ri (k < B-1 pages)
hash fn
h2
h2
Input buffer for Si
Output buffer
Disk
B main memory buffers
Disk

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!

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

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 *<sid, sname>*
    - 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 $\times$ 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.