Discussion 6

Iterators and Joins

Announcements

Vitamin 6 (Iterators and Joins) due **Monday, March 4 at 11:59pm**

Project 3 (Joins and QO) part 1 due **Wednesday, March 6 at 11:59pm**

Agenda

I. Iterators
II. Joins
    A. SNLJ
    B. PNLJ
    C. BNLJ
    D. INLJ
    E. Sort-Merge Join
    F. Grace Hash Join
III. Worksheet

Iterators

Iterators

- Interface for **iterating** through data (making a single pass through the data)
- Important methods of an iterator are:
  - hasNext: is there another piece of data left
  - next: get the next piece of data
- Using an iterator is kind going through a linked list - no support for random access
  - You can’t (efficiently) say: fetch me the 50th item, then 30th, then 70th

Iterators

- Recall: relational operators operate on relations and return relations
- We can implement this as: operate on an iterator (of the input relation) and return an iterator (of the output)
  - Optionally choose if we wish to **materialize** the output relation (write it to disk) or **stream** it to the next operator

Iterators

- If we have: $\pi_{id}(\sigma_{name > \text{'A'}}(R))$
  - σ operator takes iterator over R, returns iterator that filters out tuples that don't satisfy predicate
  - π operator takes in iterator from σ operator and returns tuples with only the id field
  - Materializing the relation returned from $\sigma_{name > \text{'A'}}(R)$ not needed: π makes only one pass over the data
    - Only need one page of R in memory at once

Joins

Joins

- We’ll be looking at inner (equi) joins
  - Algorithms can be pretty easily extended to left/right outer joins
  - Full joins require more thought - not in scope
  - Some algorithms work for non-equi-joins, others don’t
- A join is: taking one relation, and matching each tuple with tuples from another relation
- The **join condition/predicate** determines what rows in the other relation match to a row in the first relation

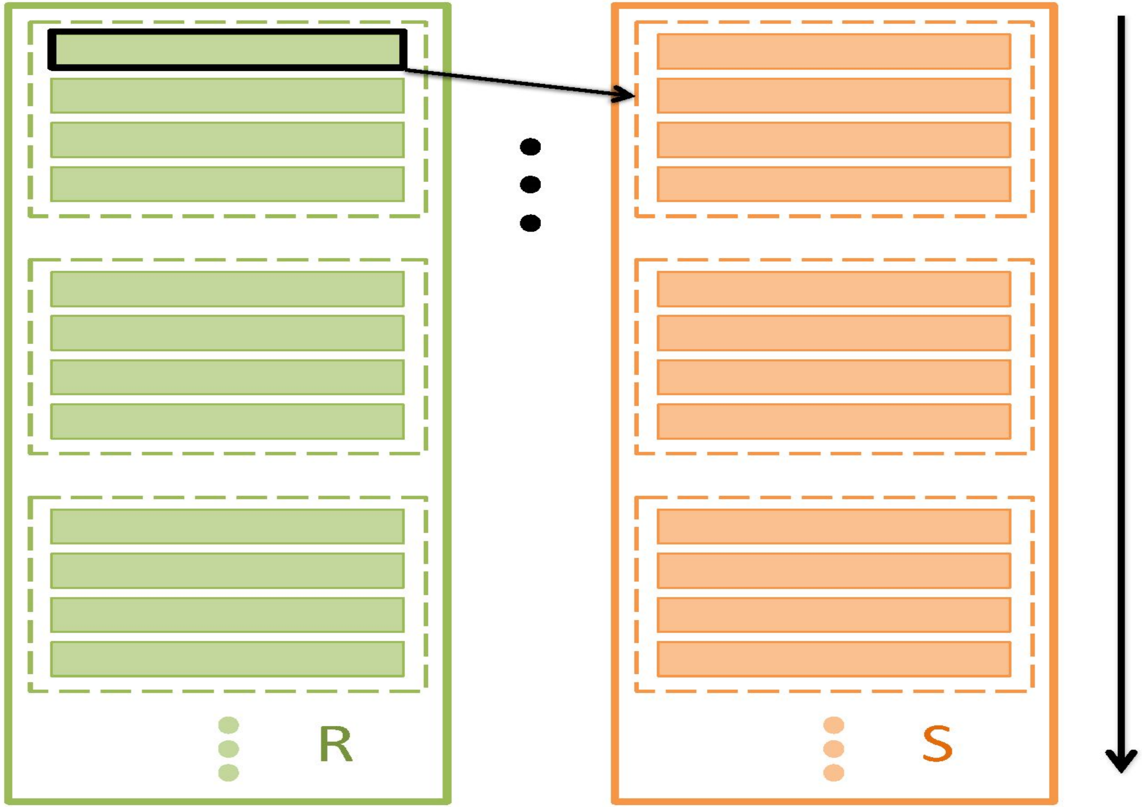
Joins

- Bit of notation:
  - [R] = number of pages in R
  - p_R = number of records per page in R
  - |R| = number of records in R (the **cardinality** of R)
    - $|R| = p_R * [R]$
- We typically *exclude* the final write's I/O cost
  - Don't add the cost of writing the joined output to disk
    - We might decide to stream it to the next operator instead of materializing results!

Simple Nested Loop Join (SNLJ)

- Direct translation of the definition of join into code
- To perform the join $R \bowtie_{\theta} S$, just take each row in R, and scan through S to find the matching rows!
  - for each row r in R:
    - for each row s in S:
      - if θ(r, s): output r joined with s

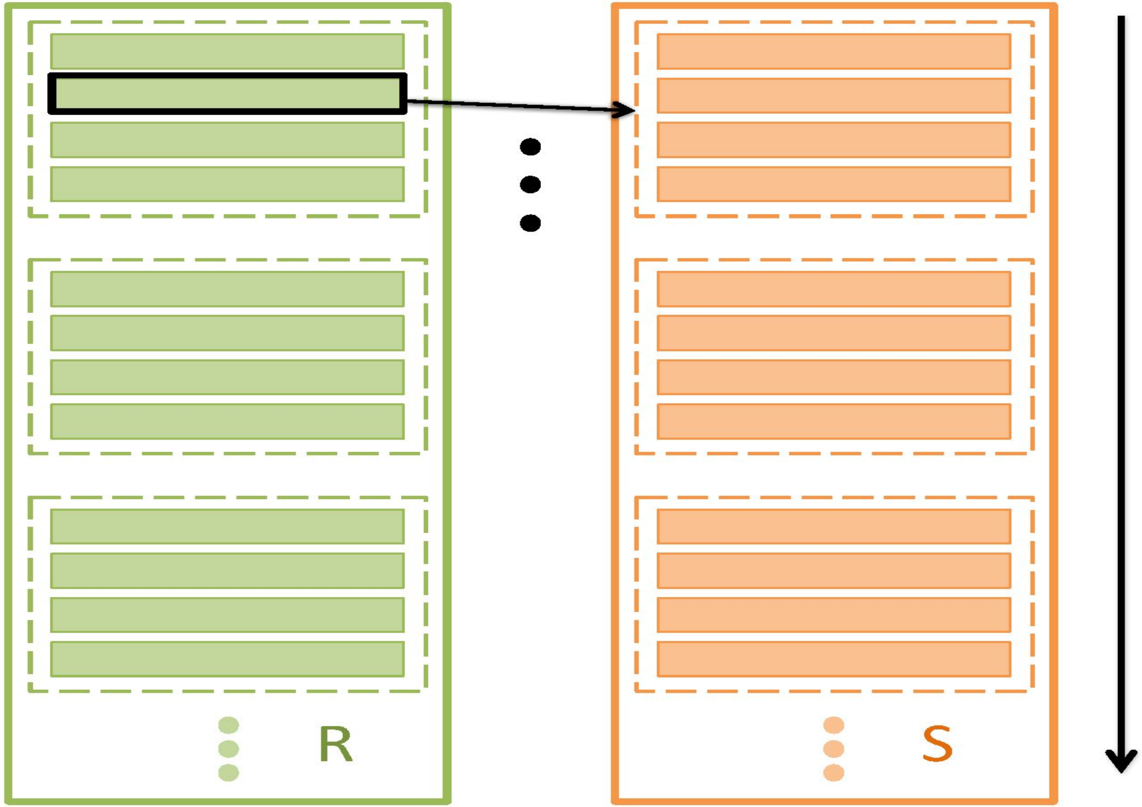
Simple Nested Loop Join



First iteration of outer loop…

Compare with all tuples in S
…and add matches to result

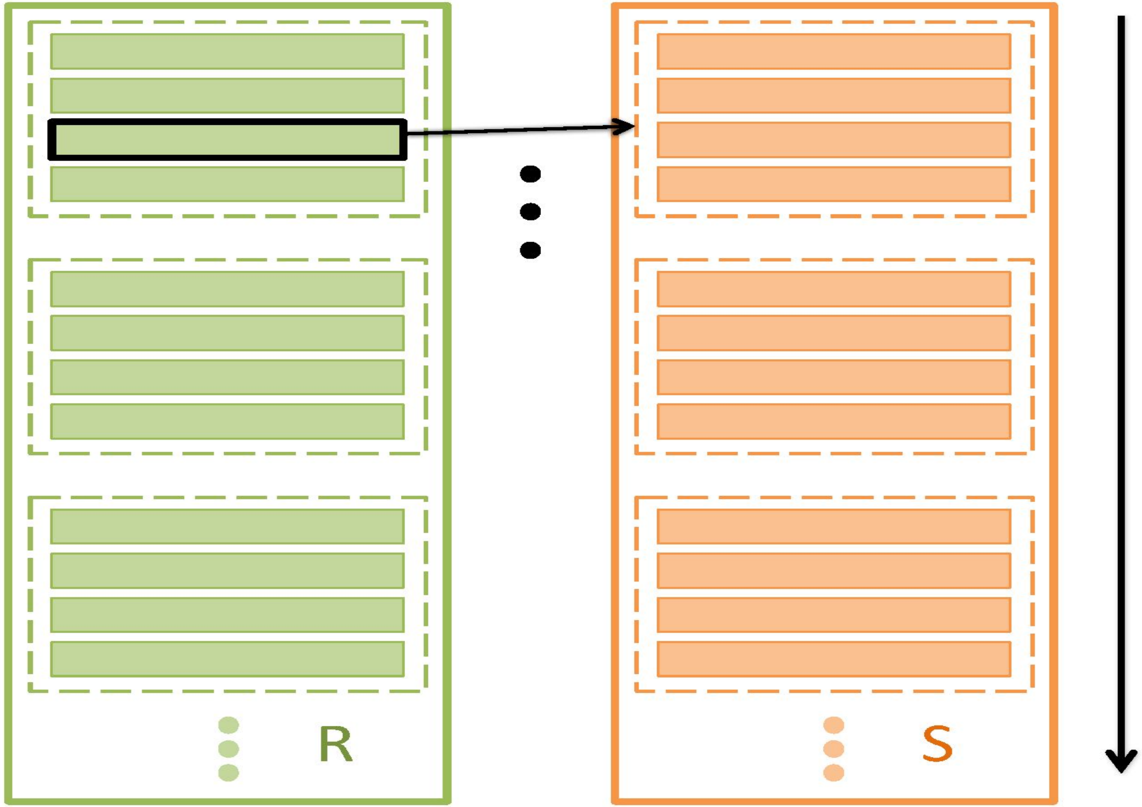
Simple Nested Loop Join



Second iteration of outer loop…

Compare with all tuples in S
…and add matches to result

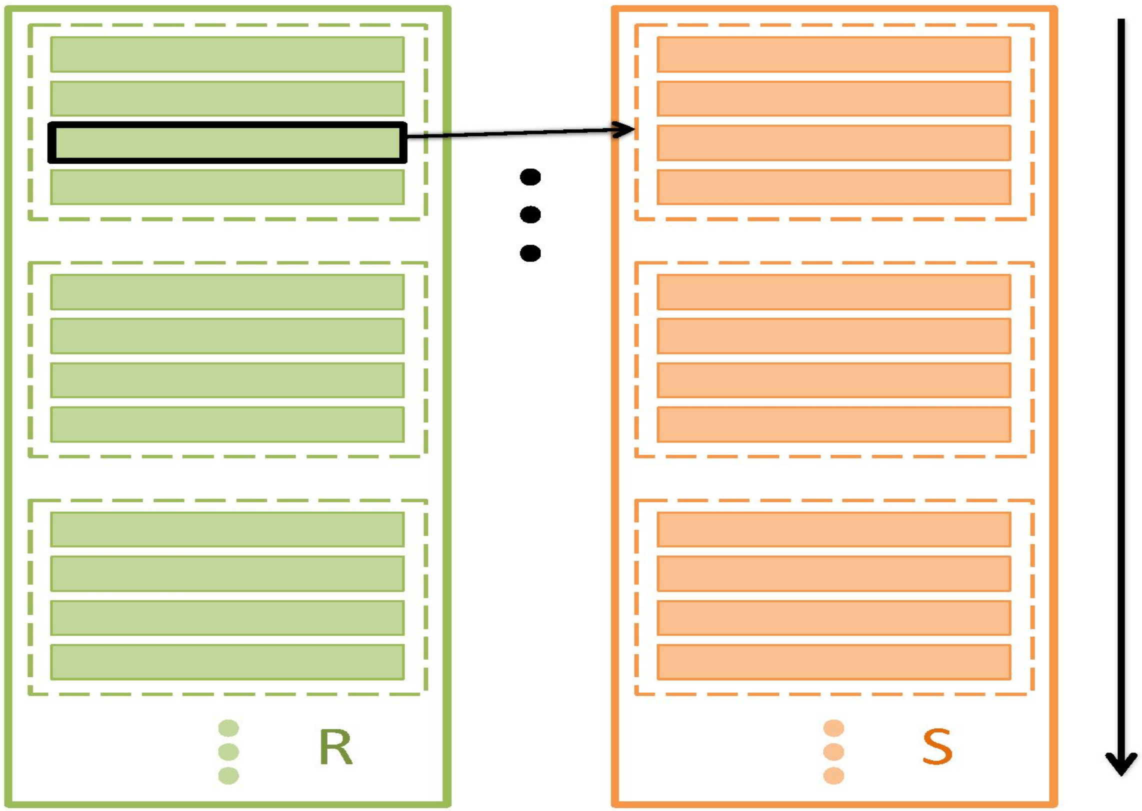
Simple Nested Loop Join



Cost =

$[R] + p_R[R][S]$

Simple Nested Loop Join



Which relation should we pick as R and S respectively?

Cost =

$[R] + p_R[R][S]$

Worksheet Q1a

How many disk I/Os are needed to perform a simple nested loop join?

```
Companies: (company_id, industry, ipo_date)
Nyse: (company_id, date, trade, quantity)
```

- 20 pages of memory
- Join Companies and NYSE on C.company_id = N.company_id
- company_id is the primary key for Companies
- For every tuple in Companies, assume there are 4 matching tuples in NYSE
- [N] = 100 pages, p_N = 100 tuples per page
- [C] = 50 pages, p_C = 50 tuples per page
- Unclustered B+ indexes on C.company_id and N.company_id
- For both indexes, assume it takes 2 I/Os to access a leaf

Worksheet Q1a

How many disk I/Os are needed to perform a simple nested loop join?

$C \bowtie N$

Cost is $[C] + |C| * [N] = [C] + p_C [C] [N]$

$= 50 + 50 * 50 * 100 = 250{,}050$ I/Os

$N \bowtie C$

Cost is $[N] + |N| * [C] = [N] + p_N [N] [C]$

$= 100 + 100 * 100 * 50 = 500{,}100$ I/Os

```
Companies: (company_id, industry, ipo_date)
Nyse: (company_id, date, trade, quantity)
```

- 20 pages of memory
- We want to join Companies and NYSE on C.company_id = N.company_id
- company_id is the primary key for Companies
- For every tuple in Companies, assume there are 4 matching tuples in NYSE
- $[N] = 100$ pages, $p_N = 100$ tuples per page
- $[C] = 50$ pages, $p_C = 50$ tuples per page
- Unclustered B+ indexes with height 1 on C.company_id and N.company_id

I/O cost for SNLJ: min(500,100, 250,050) = **250,050 I/Os**

Page Nested Loop Join (PNLJ)

- Can we do better?
  - We scan S for every row in R, but we had to load an entire page of R into memory to get that row!
  - Instead of finding the rows in S that match a row in R, do the check for *all* rows in a page in R at once

Page Nested Loop Join (PNLJ)

- SNLJ
  - for each row r in R:
    - for each row s in S:
      - if θ(r, s): output r joined with s

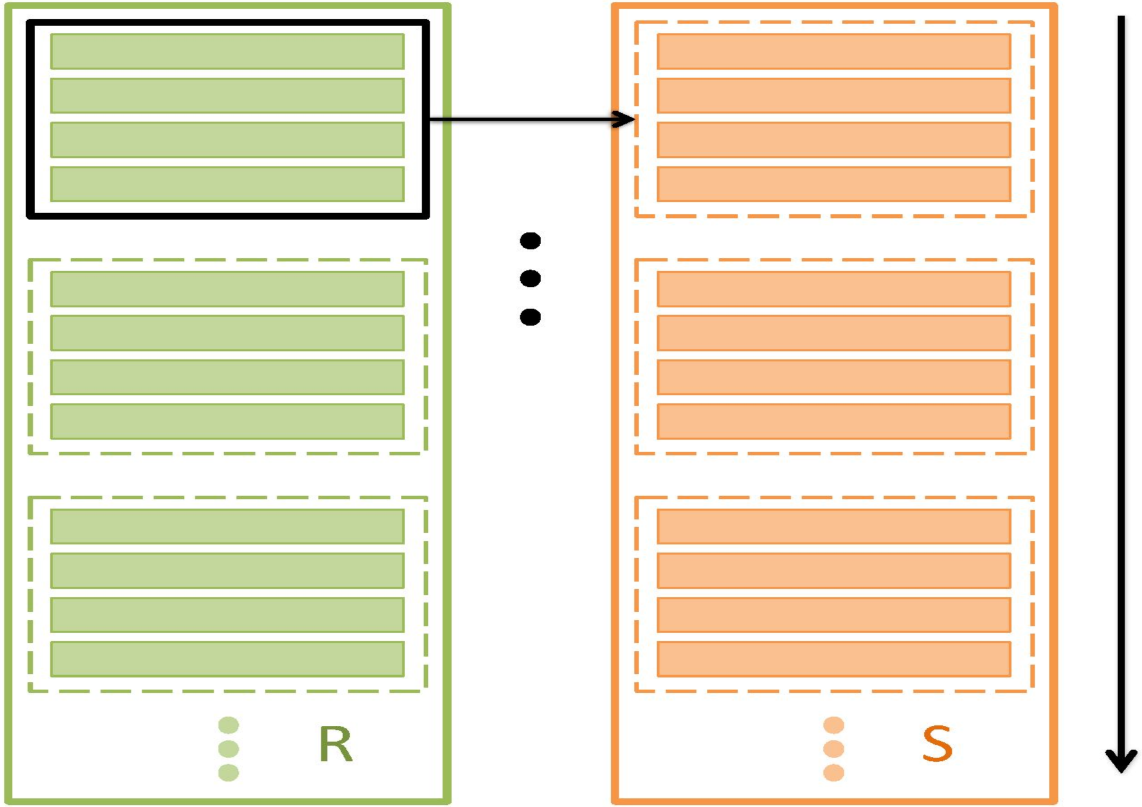
Page Nested Loop Join (PNLJ)

- SNLJ (but with page fetches written out explicitly)
    - for each page P_R in R:
        - for each row r in P_R:
            - for each page P_S in S:
                - for each row s in P_S:
                    - if θ(r, s): output r joined with s

Page Nested Loop Join (PNLJ)

- **P**NLJ
  - for each page P_R in R:
    - **for each page P_S in S:**
      - **for each row r in P_R:**
        - for each row s in P_S:
          - if $\theta(r, s)$: output r joined with s

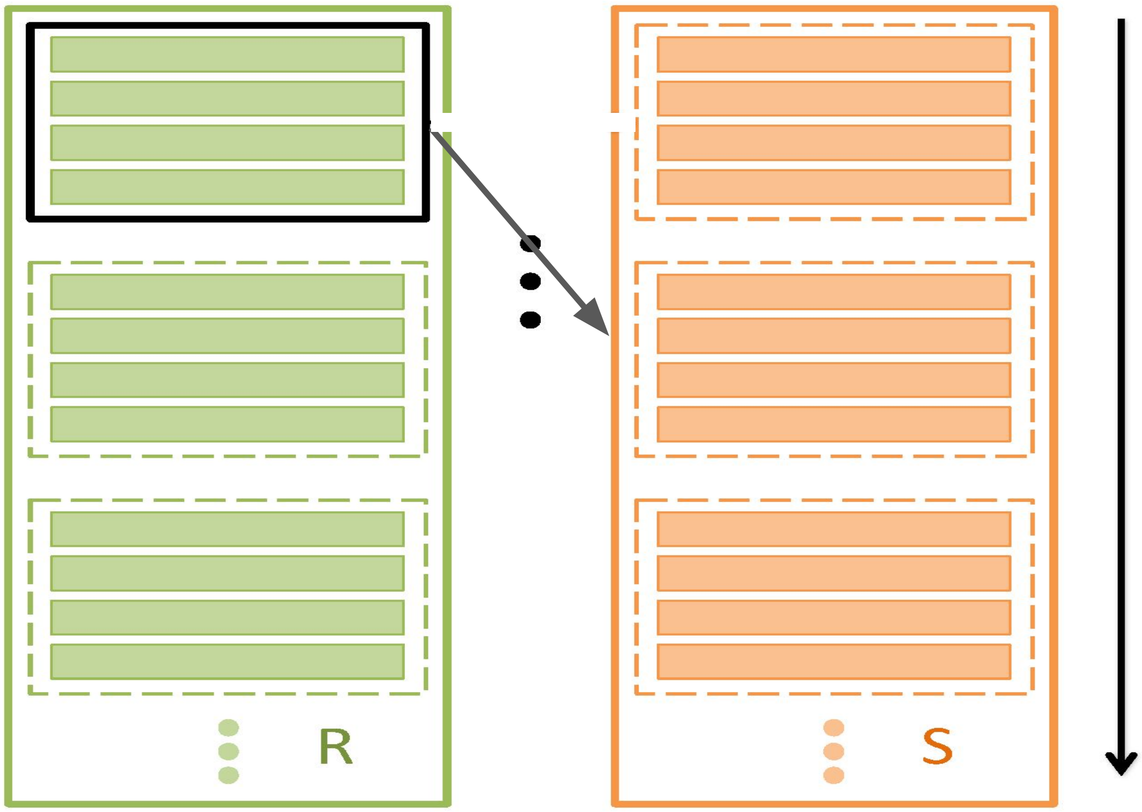
Page-Oriented Nested Loop Join



First iteration of outer loop…

Compare with all tuples in S
…and add matches to result

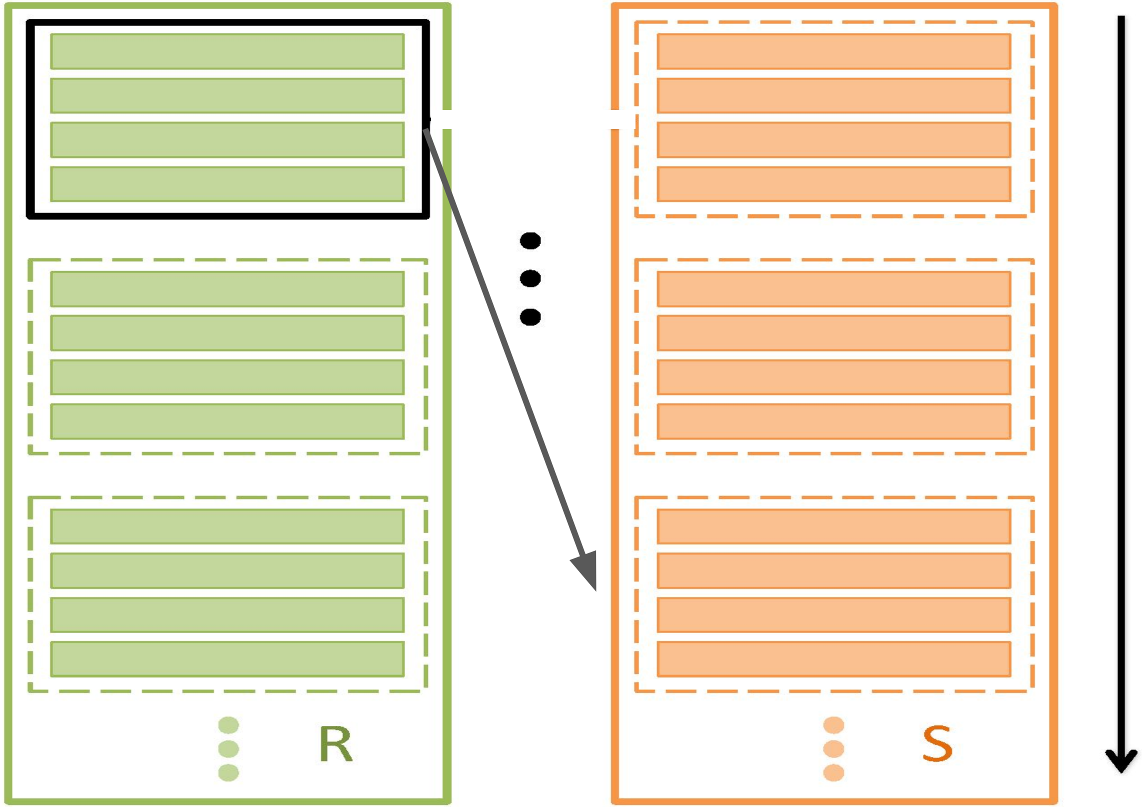
Page-Oriented Nested Loop Join



First iteration of outer loop…

Compare with all tuples in S
…and add matches to result

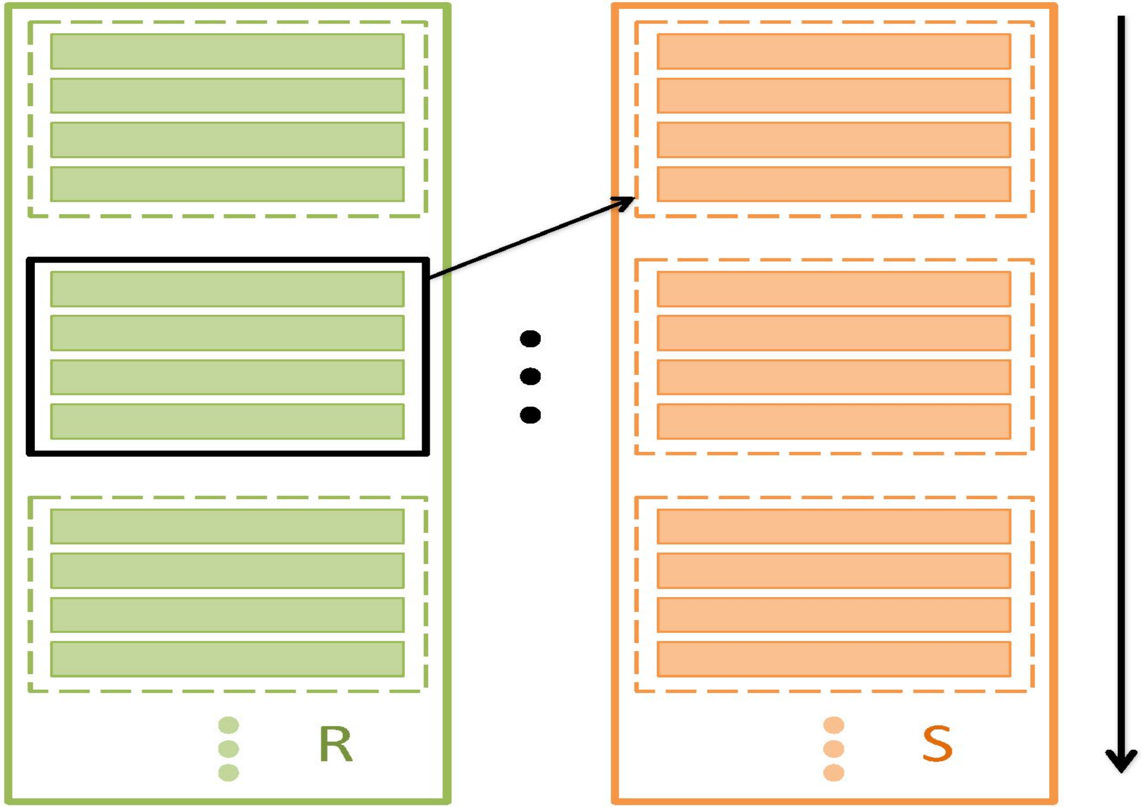
Page-Oriented Nested Loop Join



First iteration of outer loop…

Compare with all tuples in S
…and add matches to result

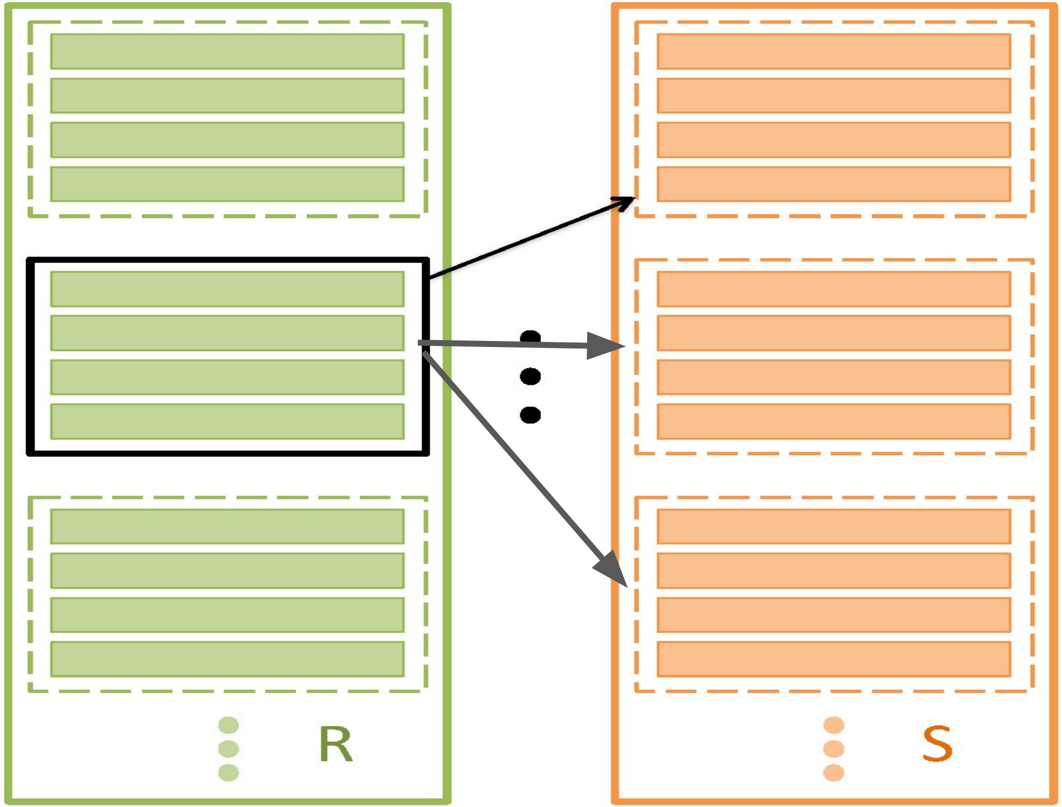
Page-Oriented Nested Loop Join



Second iteration of outer loop…

Compare with all tuples in S
…and add matches to result

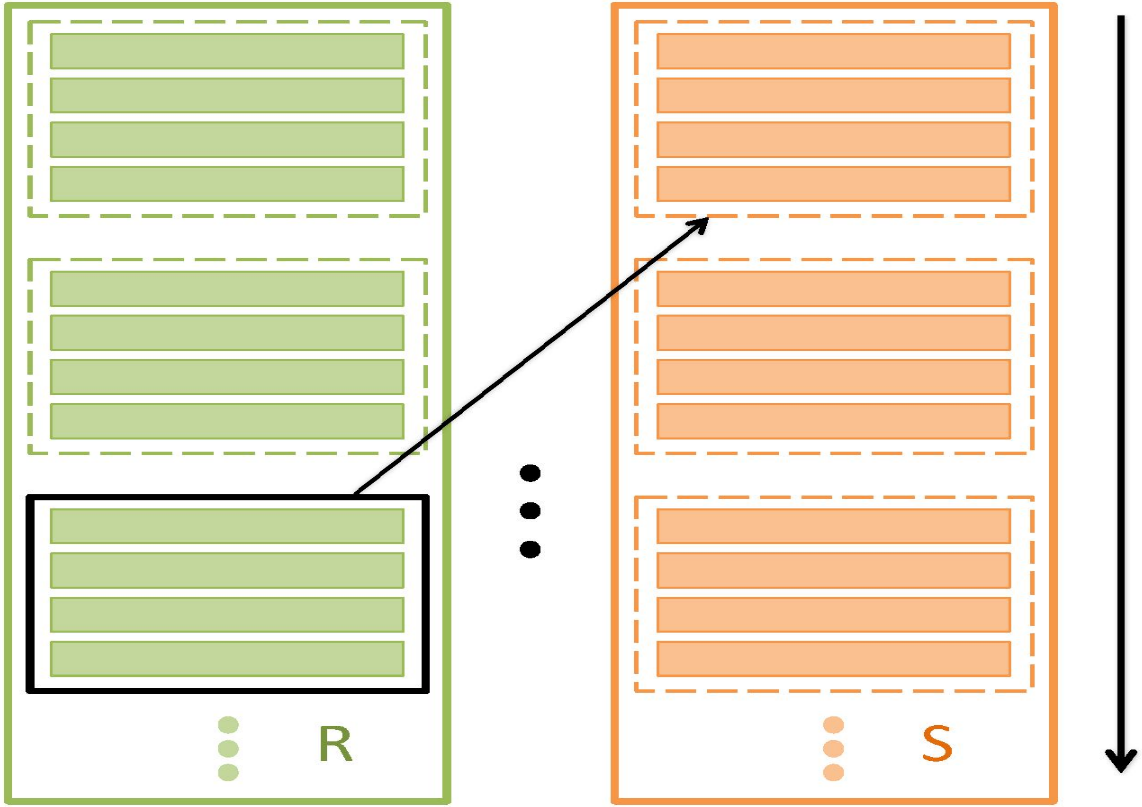
Page-Oriented Nested Loop Join



Second iteration of outer loop...

Compare with all tuples in S
...and add matches to result

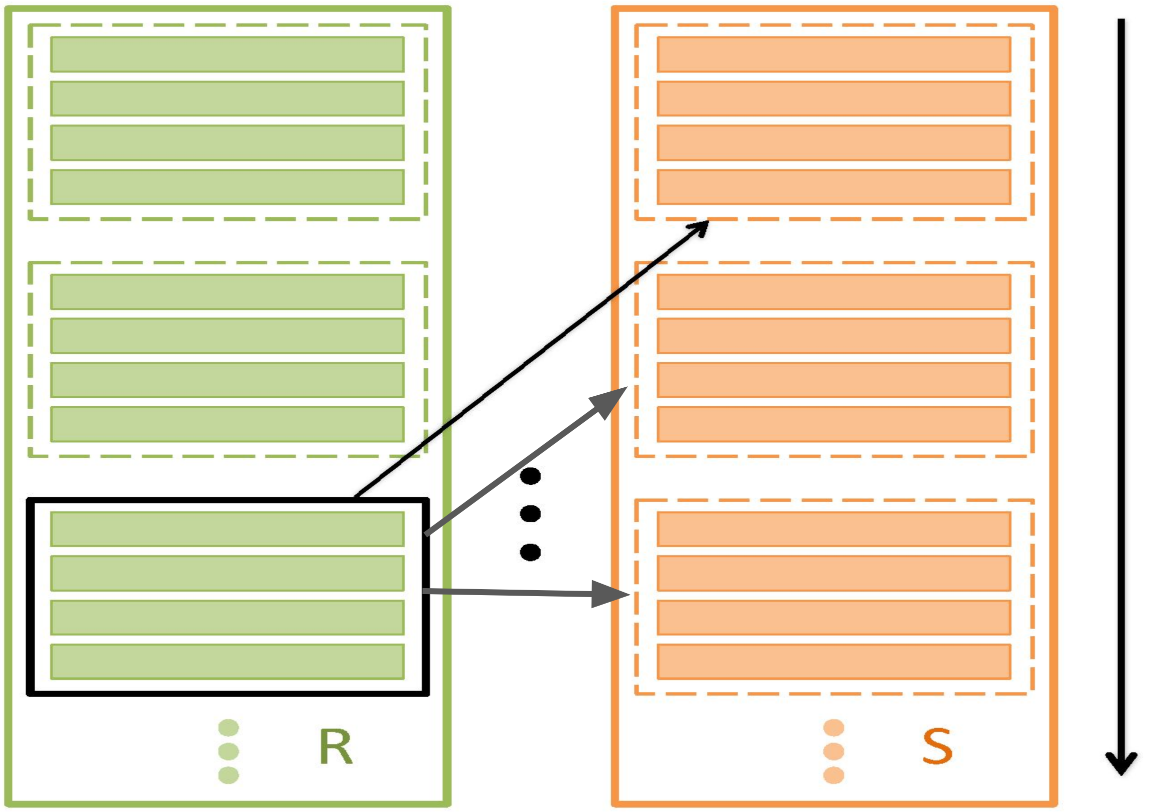
Page-Oriented Nested Loop Join



Cost =

[R] + [R][S]

Page-Oriented Nested Loop Join



Cost =
[R] + [R][S]

Block Nested Loop Join (BNLJ)

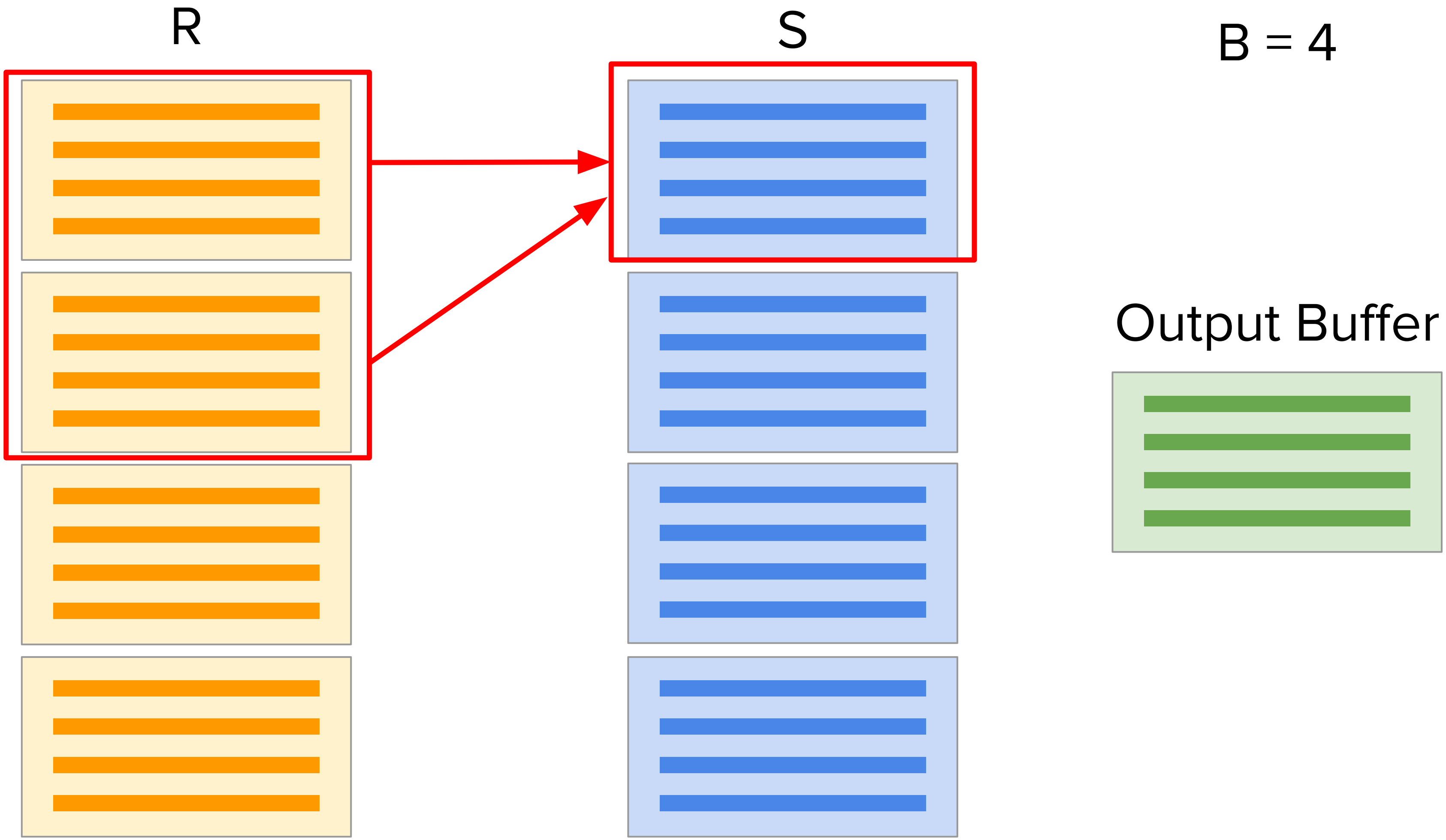
- Can we do even better?
  - We only use three page of memory for PNLJ (one buffer for R, one buffer for S, one output buffer), but we usually have more memory!
  - Instead of fetching one *page* of R at a time, why not fetch as many pages of R as we can fit (B - 2 pages)!

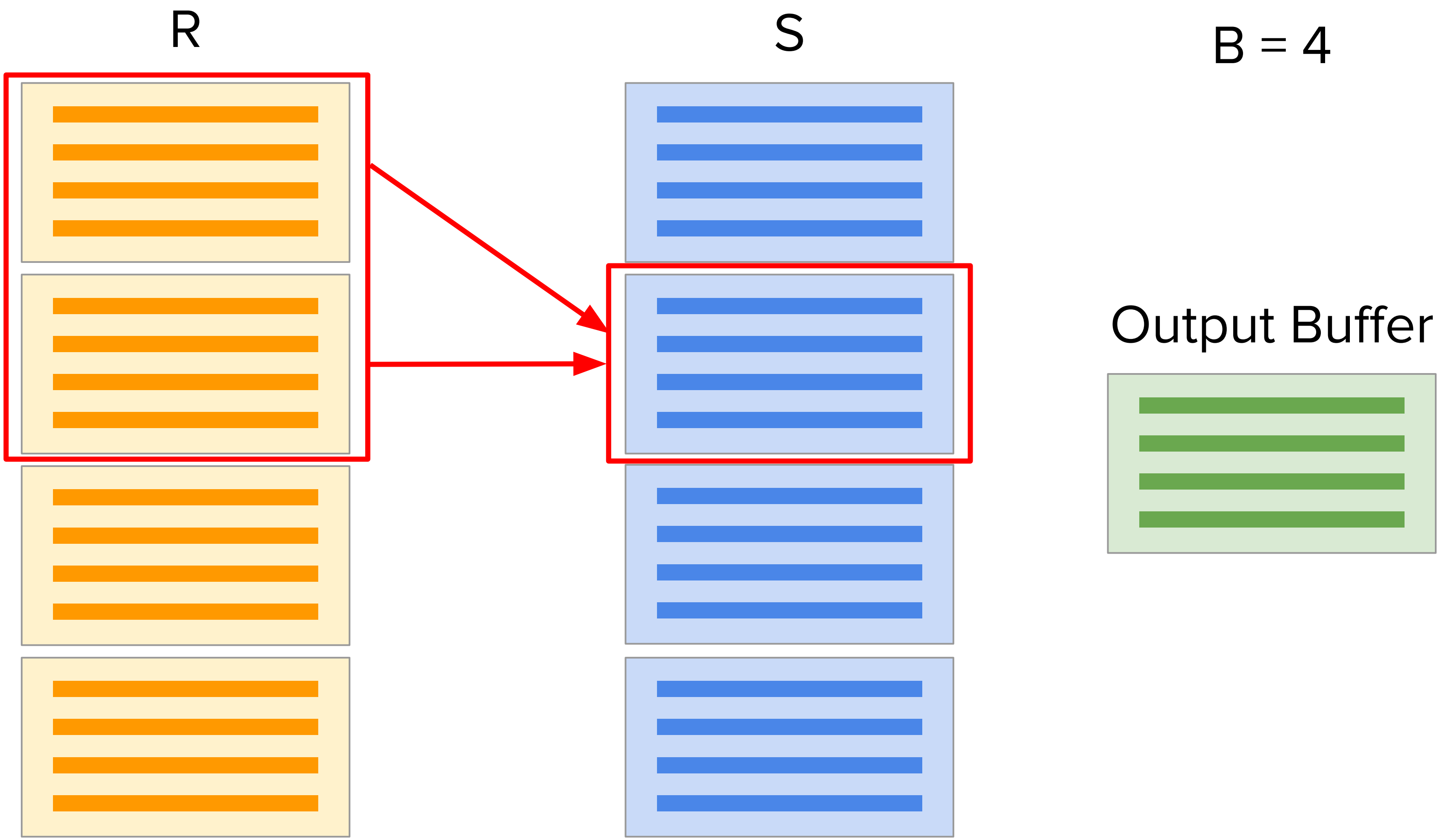
Block Nested Loop Join (BNLJ)

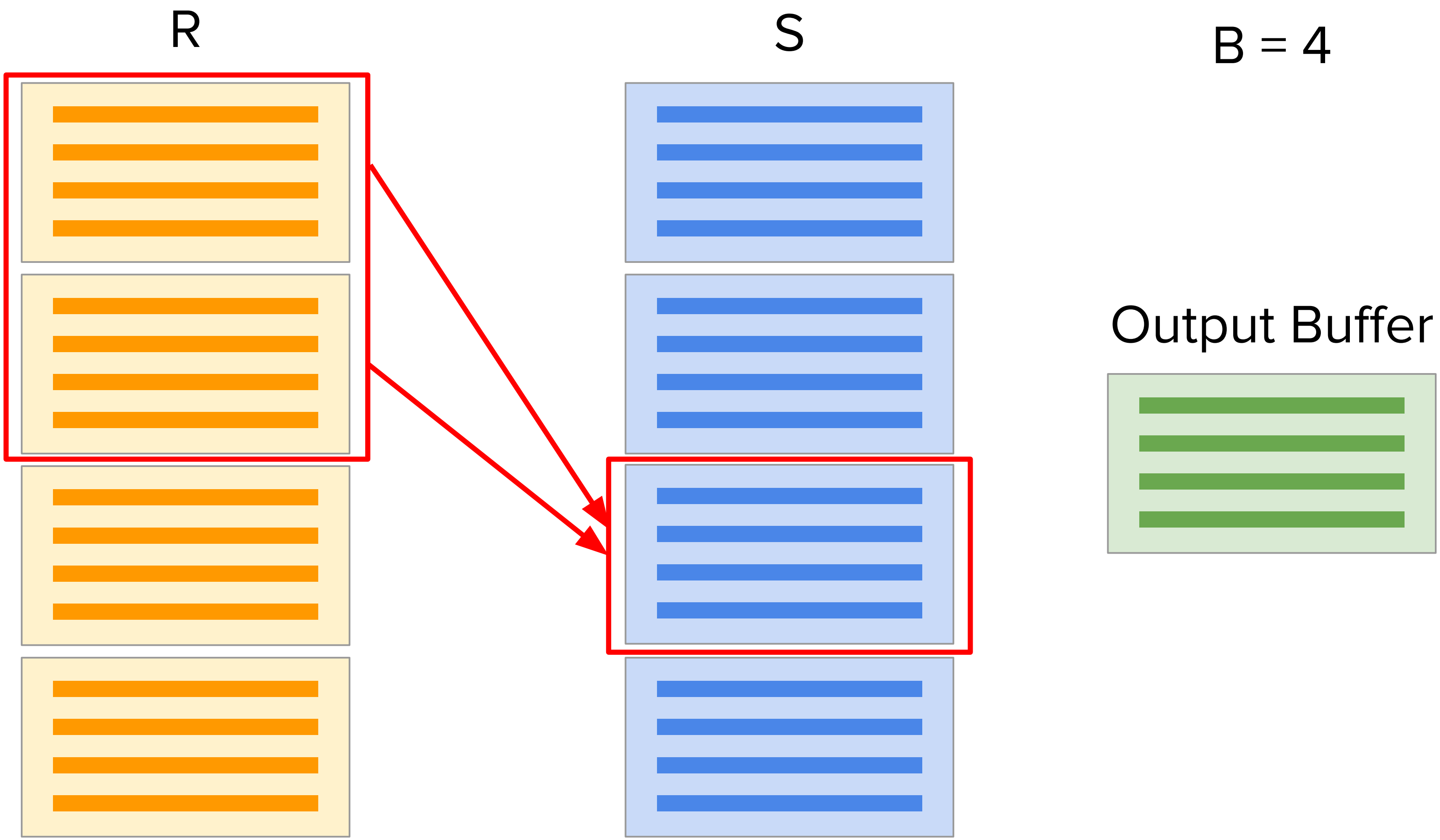
- PNLJ
  - for each page P_R in R:
    - for each page P_S in S:
      - for each row r in P_R:
        - for each row s in P_S:
          - if θ(r, s): output r joined with s

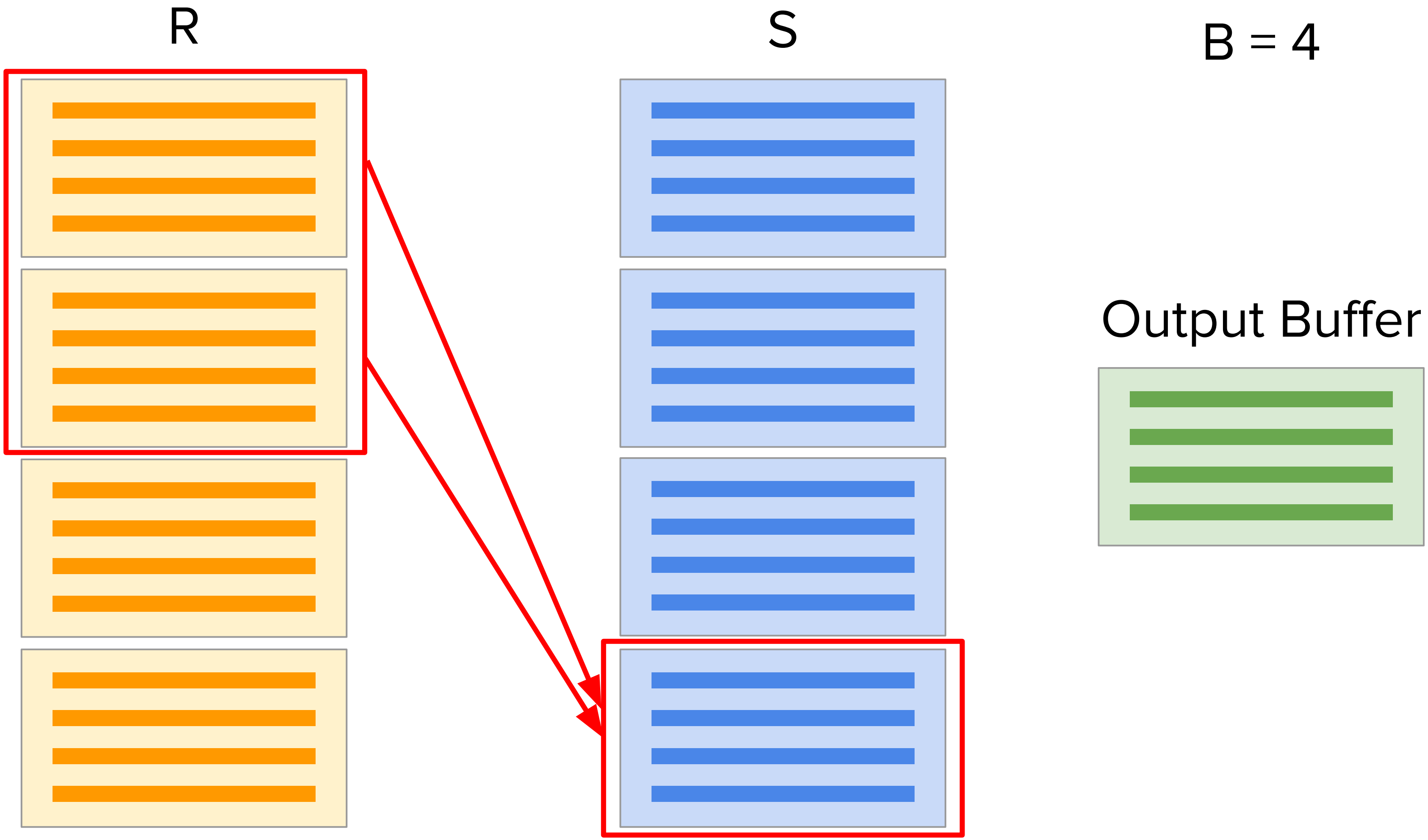
Block Nested Loop Join (BNLJ)

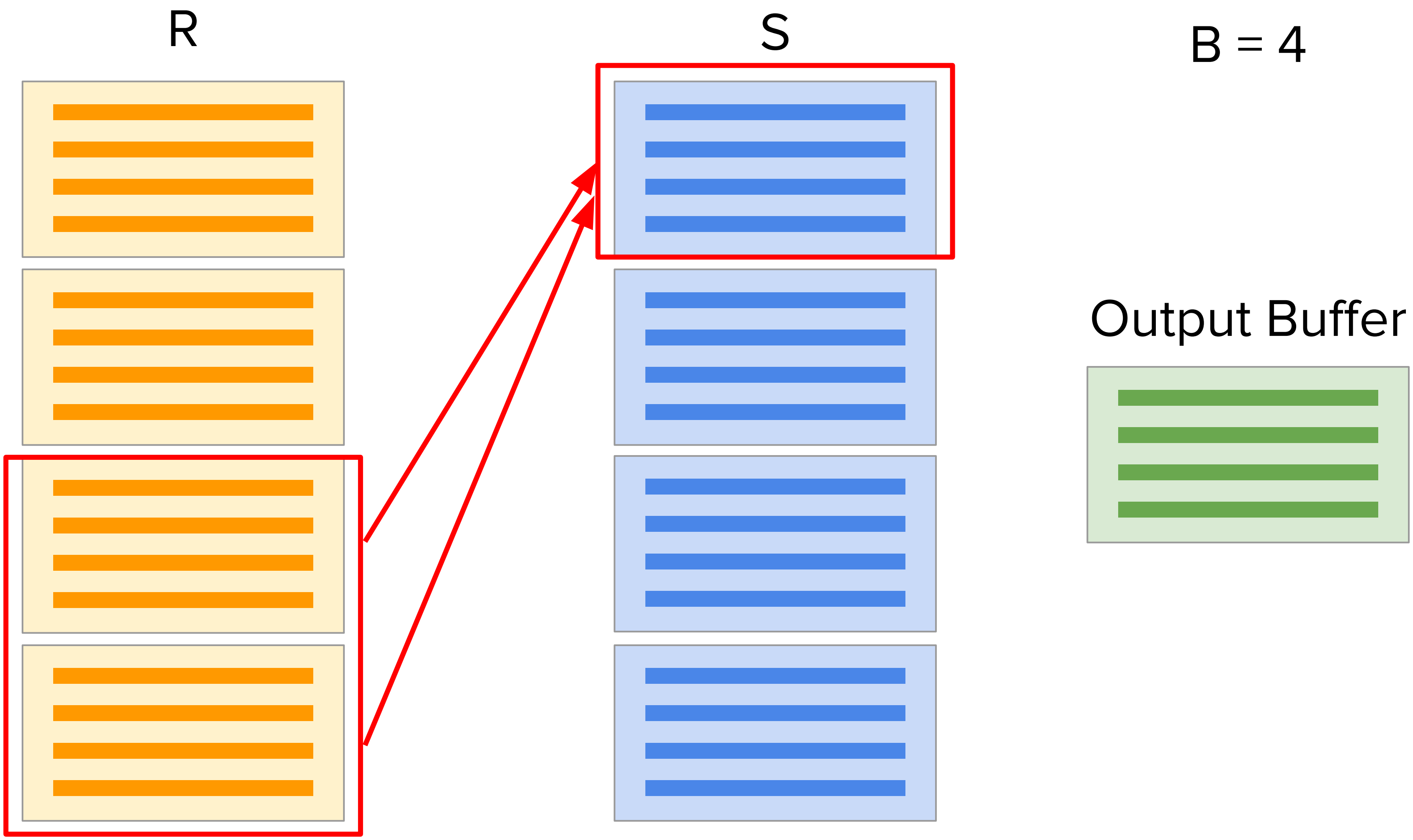
- **B**NLJ
  - **for each block of B - 2 pages $C_R = \{P_1, P_2, \ldots, P_{B-2}\}$ in R:**
    - for each page P_S in S:
      - **for each row r in C_R:**
        - for each row s in P_S:
          - if θ(r, s): output r joined with s

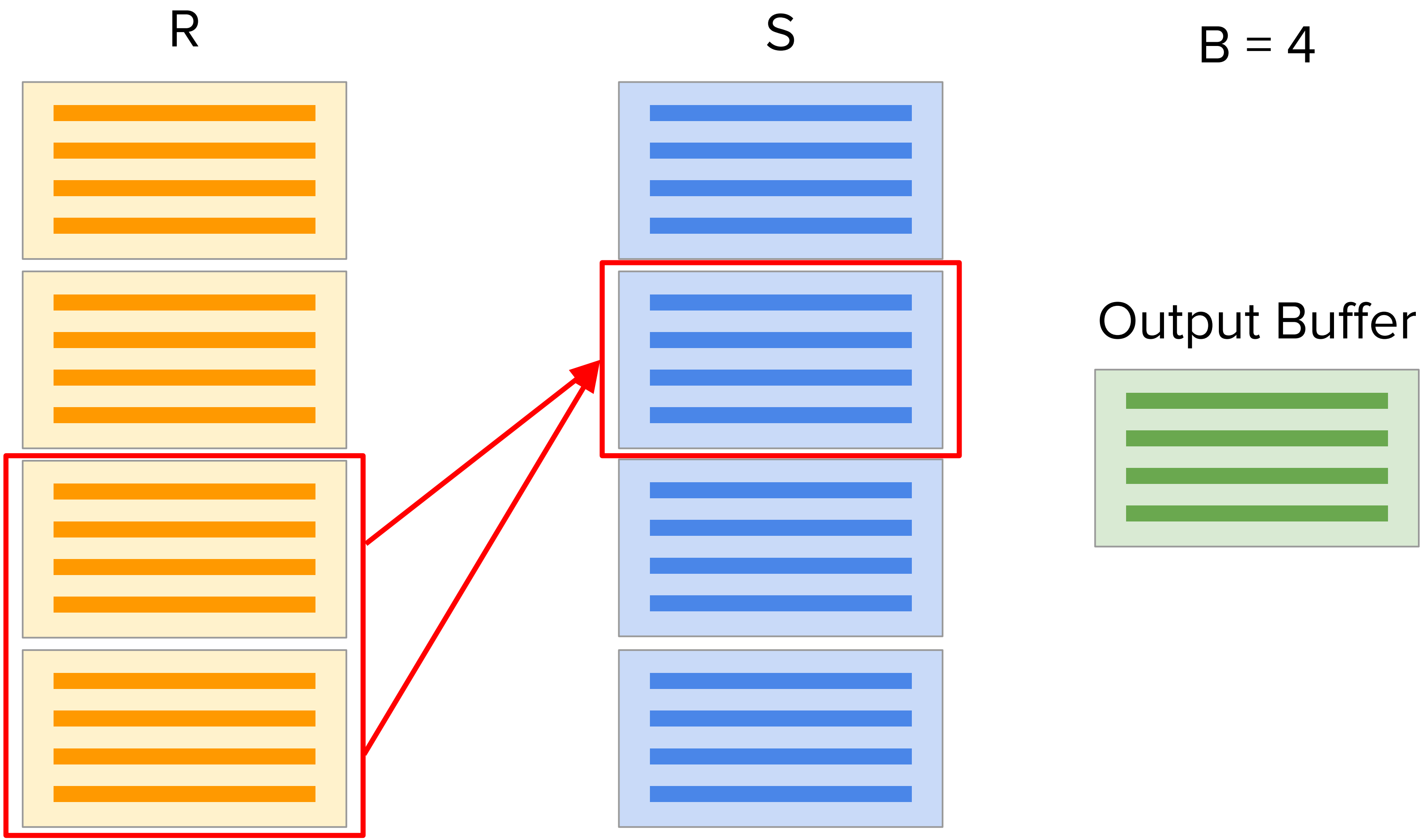

R
S
B = 4
Output Buffer

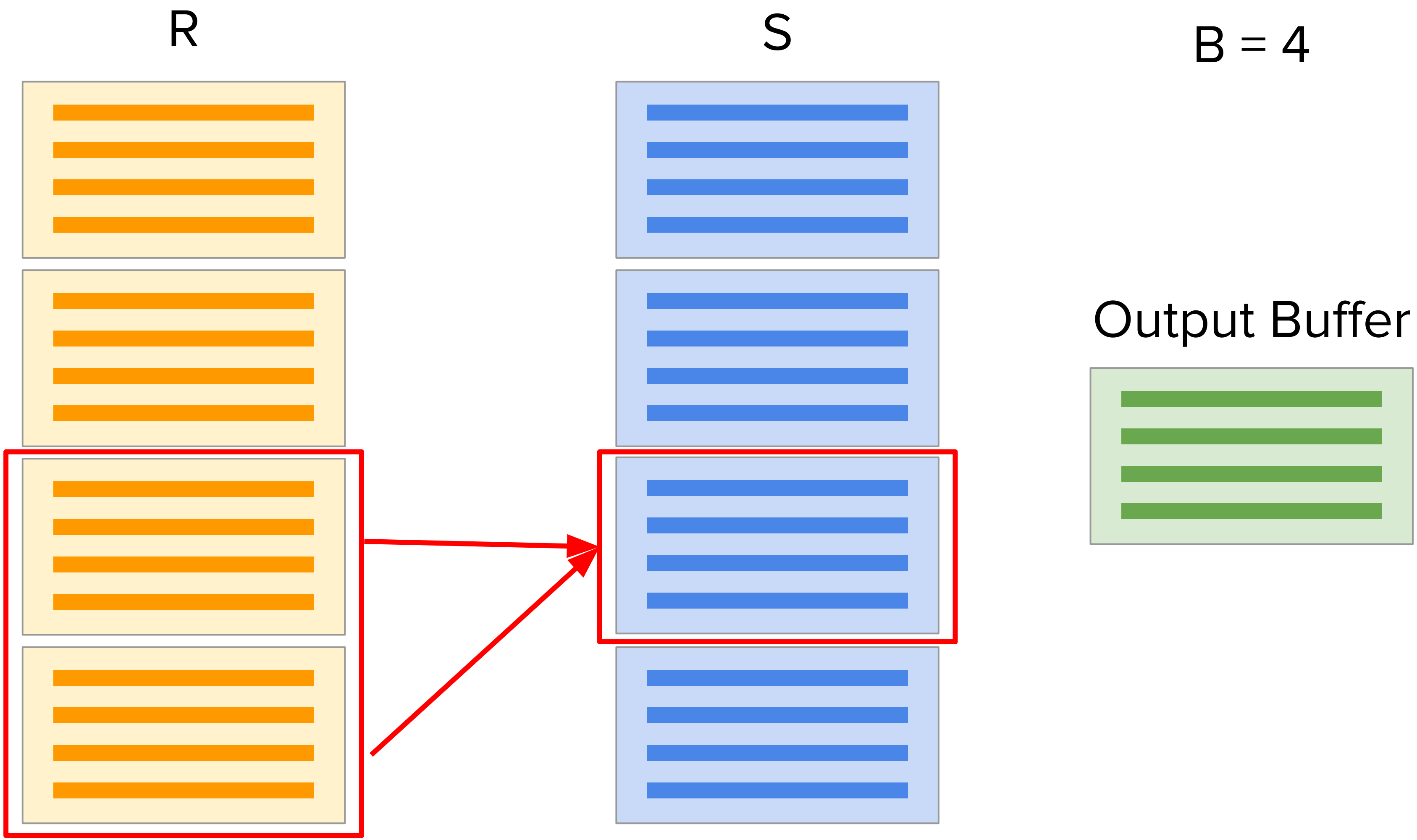

R
S
B = 4
Output Buffer

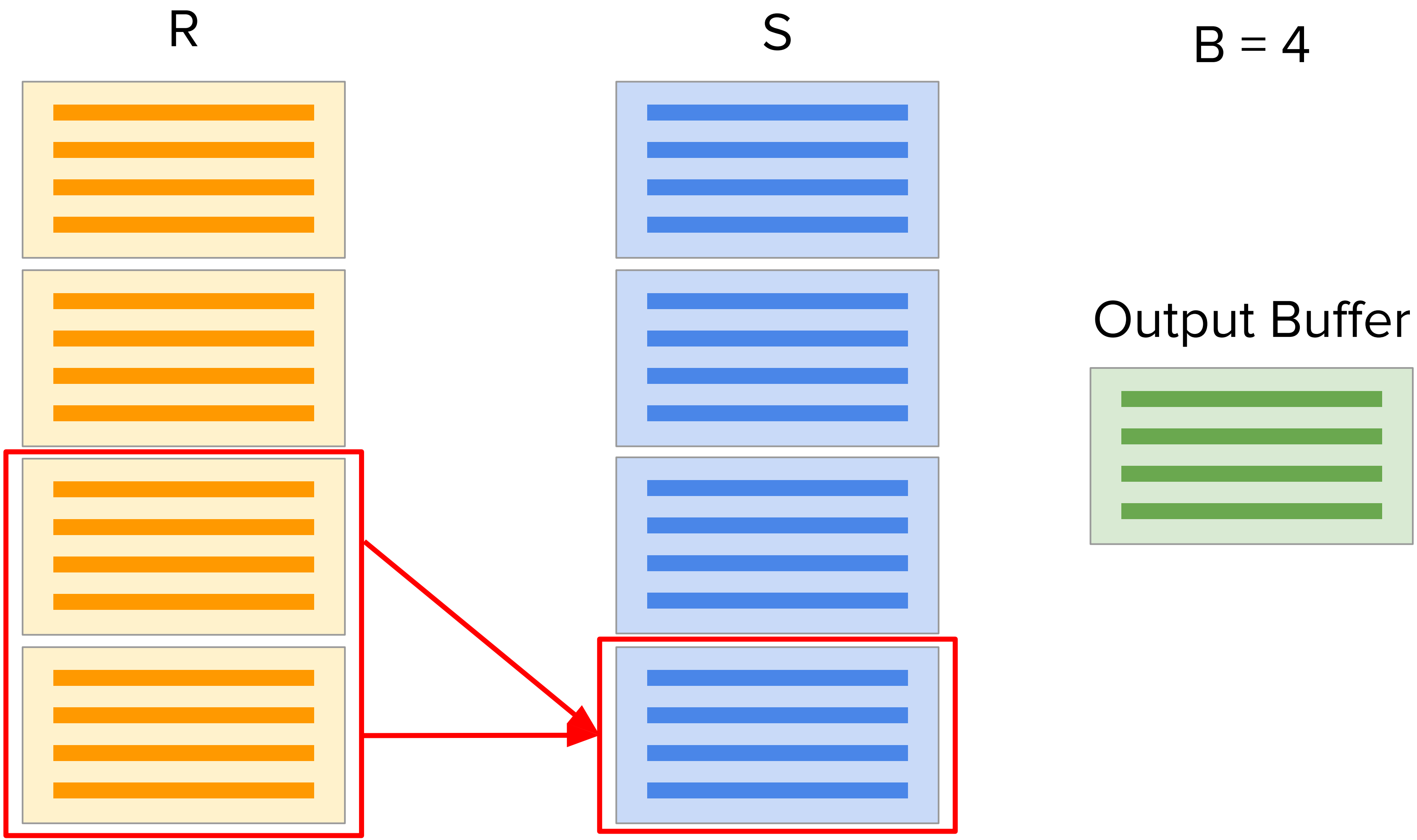

R
S
B = 4
Output Buffer


R
S
B = 4
Output Buffer

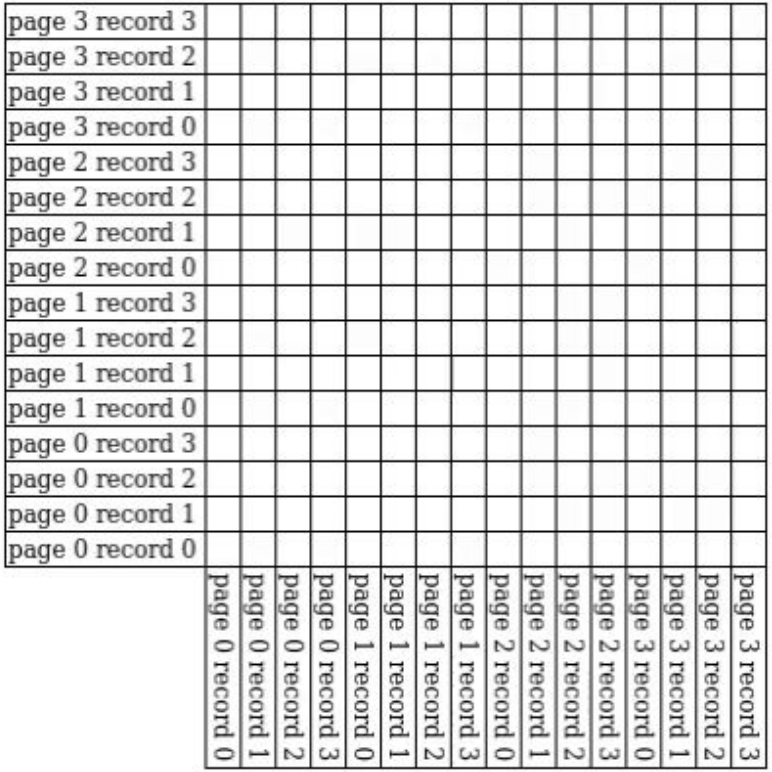

R
S
B = 4
Output Buffer


R
S
B = 4
Output Buffer


R
S
B = 4
Output Buffer


R
S
B = 4
Output Buffer

BNLJ



leftRecordIterator: Iterator over left block
rightRecordIterator: Iterator over right page
leftRecord:
rightRecord :

Cost of BNLJ?

$$[R] + (\#\ \text{blocks in } R) * [S]$$

$$= [R] + \lceil {}^{[R]}/_{chunksize} \rceil * [S]$$

$$\boxed{= [R] + \lceil {}^{[R]}/_{(B-2)} \rceil [S]}$$

Worksheet Q1b

How many disk I/Os are needed to perform a block nested loop join?

```
Companies: (company_id, industry, ipo_date)
Nyse: (company_id, date, trade, quantity)
```

- 20 pages of memory
- We want to join Companies and NYSE on C.company_id = N.company_id
- company_id is the primary key for Companies
- For every tuple in Companies, assume there are 4 matching tuples in NYSE
- [N] = 100 pages, p_N = 100 tuples per page
- [C] = 50 pages, p_C = 50 tuples per page
- Unclustered B+ indexes with height 1 on C.company_id and N.company_id

Worksheet Q1b

How many disk I/Os are needed to perform a block nested loop join?

B = 20, block size = B - 2 = 18

C ⋈ N

Cost is [C] + ⌈[C] / B - 2⌉ * [N]

= 50 + ⌈50 / 18⌉ * 100 = 350 I/Os

N ⋈ C

Cost is [N] + ⌈[N] / B - 2⌉ * [C]

= 100 + ⌈100 / 18⌉ * 50 = 400 I/Os

I/O cost for BNLJ: min(350, 400) I/Os = **350 I/Os**

```
Companies: (company_id, industry, ipo_date)
Nyse: (company_id, date, trade, quantity)
```

- 20 pages of memory
- We want to join Companies and NYSE on C.company_id = N.company_id
- company_id is the primary key for Companies
- For every tuple in Companies, assume there are 4 matching tuples in NYSE
- [N] = 100 pages, p_N = 100 tuples per page
- [C] = 50 pages, p_C = 50 tuples per page
- Unclustered B+ indexes with height 1 on C.company_id and N.company_id

Index Nested Loop Join (INLJ)

- A join is essentially:
  - for each row r in R:
    - for each row s in S that satisfies θ(r, s):
      - output r joined with s

Index Nested Loop Join (INLJ)

- An *index on S* allows us to do the inner loop efficiently!
  - for each row r in R:
    - for each row s in S that satisfies θ(r, s) ***(found using the index)***:
      - output r joined with s

Index Nested Loop Join (INLJ)

- What's the I/O cost?
  - [R] + |R| * cost to find matching S tuples
    - [R] from scanning through R
  - Cost to find matching S tuples:
    - Alternative 1: cost to traverse root to leaf + read all the leaves with matching tuples
    - Alternative 2/3: cost of retrieving RIDs (similar to Alternative 1) + cost to fetch actual records
      - 1 I/O per **page** if clustered, 1 I/O per **tuple** if not

Index Nested Loop Join (INLJ)

- What's the I/O cost?
  - [R] + |R| * cost to find matching S tuples
    - [R] from scanning through R
  - **If we have no index,** then the only way to search for matching S tuples is by scanning all of S → SNLJ, PNLJ, BNLJ, etc.
    - Cost to find matching S tuples is then [S], giving us the formula for SNLJ cost

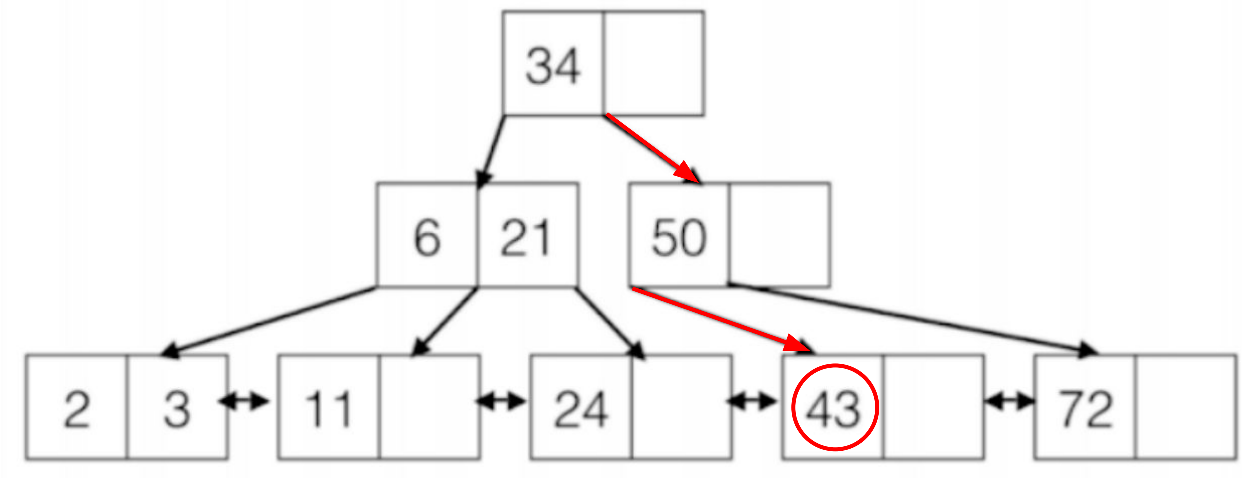
Index Nested Loop Join (INLJ)

R.col

43
5
11

Output

Index on S.col

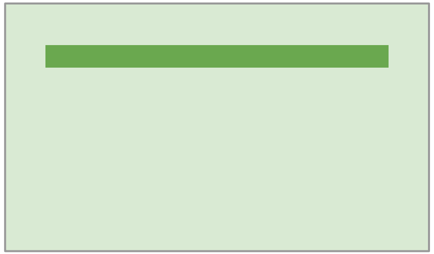


Index Nested Loop Join (INLJ)

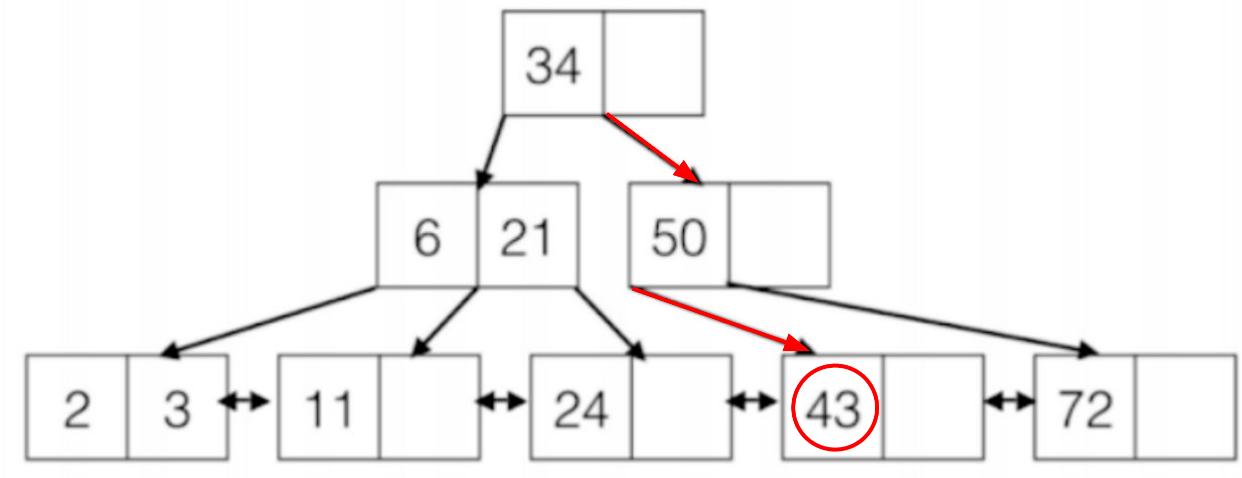
R.col

43
5
11

Output



Index on S.col



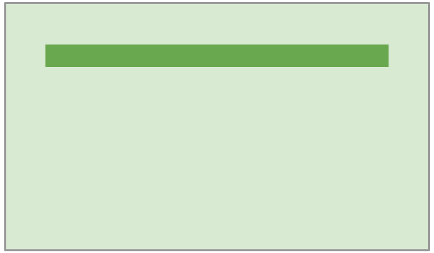
Index Nested Loop Join (INLJ)

R.col

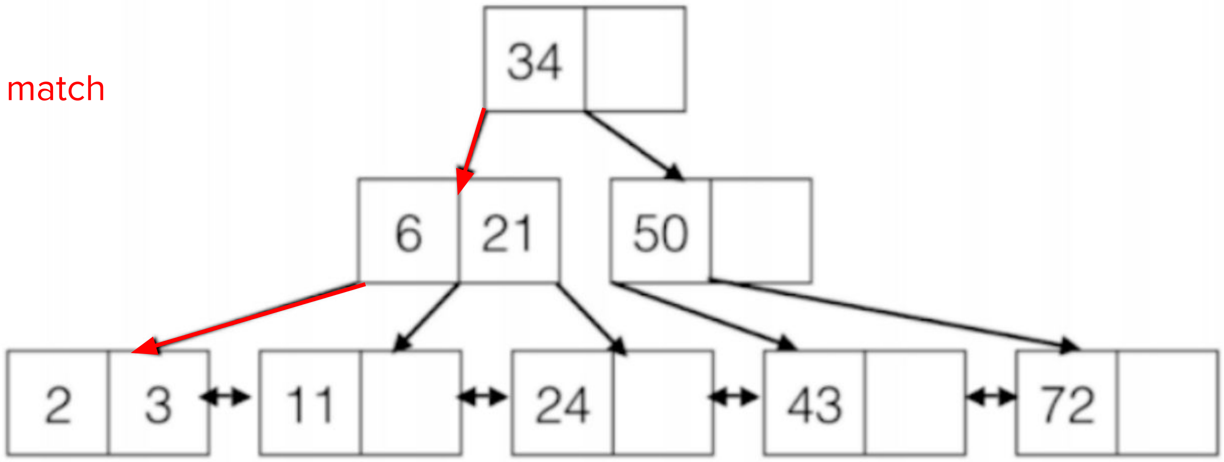
43
5
11

not a match

Output



Index on S.col

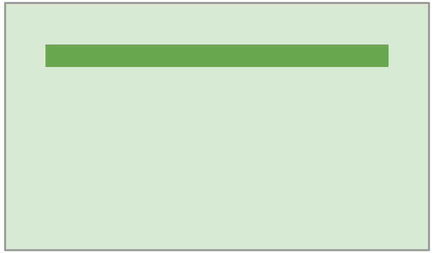


Index Nested Loop Join (INLJ)

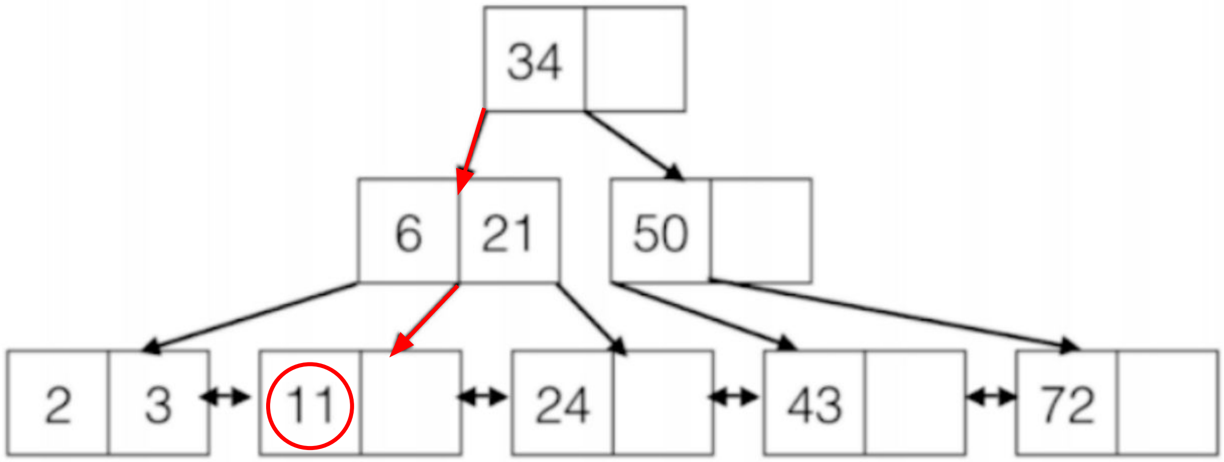
R.col

43
5
11

Output



Index on S.col

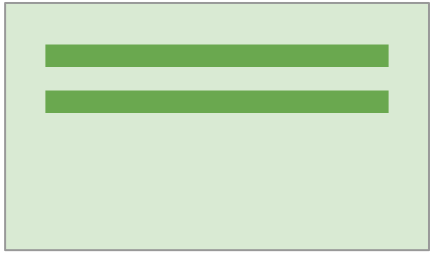


Index Nested Loop Join (INLJ)

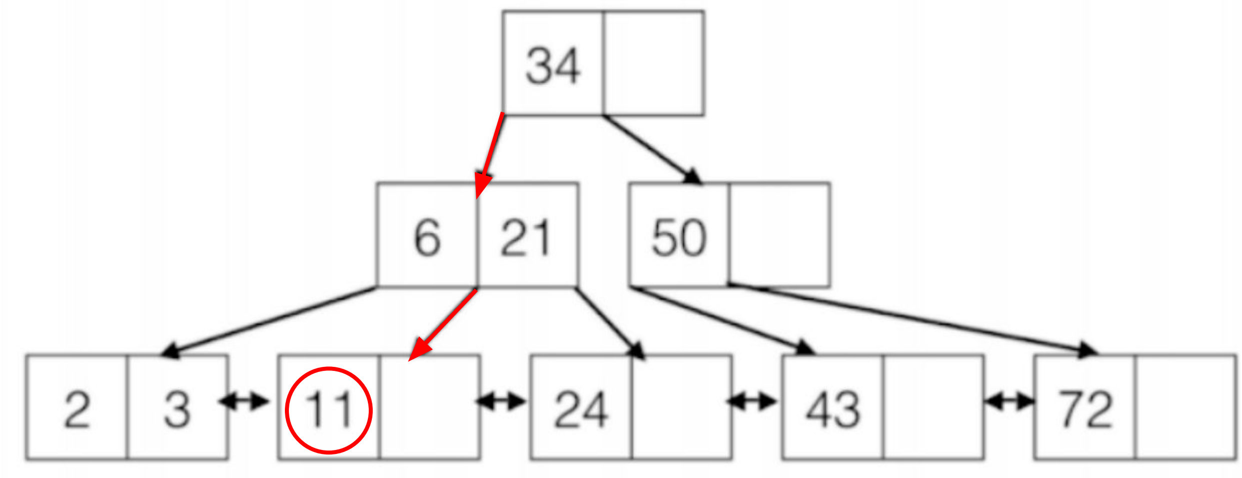
R.col

43
5
11

Output



Index on S.col



Worksheet Q1c

How many disk I/Os are needed to perform an index nested loop join?

```
Companies: (company_id, industry, ipo_date)
Nyse: (company_id, date, trade, quantity)
```

- 20 pages of memory
- We want to join Companies and NYSE on C.company_id = N.company_id
- company_id is the primary key for Companies
- For every tuple in Companies, assume there are 4 matching tuples in NYSE
- [N] = 100 pages, p_N = 100 tuples per page
- [C] = 50 pages, p_C = 50 tuples per page
- Unclustered alternative 3 B+ indexes with height 1 on C.company_id and N.company_id. Throughout the problem assume no index nodes are cached.

Worksheet Q1c

How many disk I/Os are needed to perform an index nested loop join?

C ⋈ N

Cost is [C] + |C| * cost of searching N

= 50 + (50 * 50) * (2 + 4) = 15,050 I/Os

N ⋈ C

Cost is [N] + |N| * cost of searching C

= 100 + (100 * 100) * (2 + 1) = 30,100 I/Os

I/O cost: min(30,100, 15,050) = **15,050 I/Os**

```
Companies: (company_id, industry, ipo_date)
Nyse: (company_id, date, trade, quantity)
```

- 20 pages of memory
- We want to join Companies and NYSE on C.company_id = N.company_id
- company_id is the primary key for Companies
- For every tuple in Companies, assume there are 4 matching tuples in NYSE
- [N] = 100 pages, p_N = 100 tuples per page
- [C] = 50 pages, p_C = 50 tuples per page
- Unclustered alternative 3 B+ indexes with height 1 on C.company_id and N.company_id. Throughout the problem assume no index nodes are cached.

Worksheet Q1d

Now assume the index on NYSE.company_id is clustered. What is the cost of an index nested loop join using companies as the outer relation?

```
Companies: (company_id, industry, ipo_date)
Nyse: (company_id, date, trade, quantity)
```

- 20 pages of memory
- We want to join Companies and NYSE on C.company_id = N.company_id
- company_id is the primary key for Companies
- For every tuple in Companies, assume there are 4 matching tuples in NYSE
- [N] = 100 pages, p_N = 100 tuples per page
- [C] = 50 pages, p_C = 50 tuples per page
- Unclustered alternative 3 B+ indexes with height 1 on C.company_id and N.company_id. Throughout the problem assume no index nodes are cached.

Worksheet Q1d

Now assume the index on NYSE.company_id is clustered. What is the cost of an index nested loop join using companies as the outer relation?

```
Companies: (company_id, industry, ipo_date)
Nyse: (company_id, date, trade, quantity)
```

- 20 pages of memory
- We want to join Companies and NYSE on C.company_id = N.company_id
- company_id is the primary key for Companies
- For every tuple in Companies, assume there are 4 matching tuples in NYSE
- [N] = 100 pages, p_N = 100 tuples per page
- [C] = 50 pages, p_C = 50 tuples per page
- Unclustered alternative 3 B+ indexes with height 1 on C.company_id and N.company_id. Thoughout the problem assume no index nodes are cached.

$C \bowtie N$

Cost is [C] + |C| * cost of searching N

= 50 + 50 * 50 * (2 + *# pages of matching tuples*)

= 50 + 50 * 50 * (2 + ceil(# matches /p_N))

= 50 + 50 * 50 * (2 + ceil(4/100)) = **7550 I/Os**

Sort-Merge Join (SMJ)

- What if we process the data a bit before we join things together?
  - For example, sort both relations first! Then we can join them efficiently
  - In some cases, we might even have one of the relations already sorted on the right key, and then we don't even have to spend time sorting it!

Sort-Merge Join (SMJ)

- First step: **sort** both R and S (with external sorting)
- Second step: **merge** matching tuples from R and S together
  - We do this efficiently by moving iterators over sorted R and sorted S in lockstep: move the iterator with the smaller key
    - We know that this key is smaller than *all* remaining key values in the other relation, so we're completely done joining that tuple!

Sort-Merge Join (SMJ)

- First step: **sort** both R and S (with external sorting)
- Second step: **merge** matching tuples from R and S together
  - Need a bit more care than this: we might have multiple rows in R matching with multiple rows in S
    - **Mark** the first matching row in S, match tuples with the first matching row in R, then **reset** the iterator to the mark so we can go through the rows in S again for the second matching row in R

Sort-**Merge** Join

```
while not done {
  while (r < s) { advance r }
  while (r > s) { advance s }

  mark s // save start of “block”
  while (r == s) {
    // Outer loop over r
    while (r == s) {
      // Inner loop over s
      yield <r, s>
      advance s
    }
    reset s to mark
    advance r
  }
}
```

sid	sname
22	dustin
28	yuppy
31	lubber
31	lubber2
44	guppy
57	rusty

sid	bid
28	103
28	104
31	101
31	102
42	142
58	107

Sort-**Merge** Join

```
while not done {
  while (r < s) { advance r }
  while (r > s) { advance s }

  mark s // save start of "block"
  while (r == s) {
    // Outer loop over r
    while (r == s) {
      // Inner loop over s
      yield <r, s>
      advance s
    }
    reset s to mark
    advance r
  }
}
```

sid	sname
22	dustin
28	yuppy
31	lubber
31	lubber2
44	guppy
57	rusty

sid	bid
28	103
28	104
31	101
31	102
42	142
58	107

Sort-Merge Join

```
while not done {
  while (r < s) { advance r }
  while (r > s) { advance s }

  mark s // save start of “block”
  while (r == s) {
    // Outer loop over r
    while (r == s) {
      // Inner loop over s
      yield <r, s>
      advance s
    }
    reset s to mark
    advance r
  }
}
```

sid	sname
22	dustin
28	yuppy
31	lubber
31	lubber2
44	guppy
57	rusty

sid	bid
28	103
28	104
31	101
31	102
42	142
58	107

Sort-**Merge** Join

```
while not done {
  while (r < s) { advance r }
  while (r > s) { advance s }

  mark s // save start of “block”
  while (r == s) {
    // Outer loop over r
    while (r == s) {
      // Inner loop over s
      yield <r, s>
      advance s
    }
    reset s to mark
    advance r
  }
}
```

sid	sname
22	dustin
28	yuppy
31	lubber
31	lubber2
44	guppy
57	rusty

sid	bid
28	103
28	104
31	101
31	102
42	142
58	107

sid	sname	bid
28	yuppy	103

Sort-Merge Join

```
while not done {
  while (r < s) { advance r }
  while (r > s) { advance s }

  mark s // save start of "block"
  while (r == s) {
    // Outer loop over r
    while (r == s) {
      // Inner loop over s
      yield <r, s>
      advance s
    }
    reset s to mark
    advance r
  }
}
```

sid	sname
22	dustin
28	yuppy
31	lubber
31	lubber2
44	guppy
57	rusty

sid	bid
28	103
28	104
31	101
31	102
42	142
58	107

sid	sname	bid
28	yuppy	103

Sort-Merge Join

```
while not done {
  while (r < s) { advance r }
  while (r > s) { advance s }

  mark s // save start of “block”
  while (r == s) {
    // Outer loop over r
    while (r == s) {
      // Inner loop over s
      yield <r, s>
      advance s
    }
    reset s to mark
    advance r
  }
}
```

sid	sname
22	dustin
28	yuppy
31	lubber
31	lubber2
44	guppy
57	rusty

sid	bid
28	103
28	104
31	101
31	102
42	142
58	107

sid	sname	bid
28	yuppy	103
28	yuppy	104

Sort-**Merge** Join

```
while not done {
  while (r < s) { advance r }
  while (r > s) { advance s }

  mark s // save start of "block"
  while (r == s) {
    // Outer loop over r
    while (r == s) {
      // Inner loop over s
      yield <r, s>
      advance s
    }
    reset s to mark
    advance r
  }
}
```

sid	sname
22	dustin
28	yuppy
31	lubber
31	lubber2
44	guppy
57	rusty

sid	bid
28	103
28	104
31	101
31	102
42	142
58	107

sid	sname	bid
28	yuppy	103
28	yuppy	104

Sort-Merge Join

```
while not done {
  while (r < s) { advance r }
  while (r > s) { advance s }

  mark s // save start of "block"
  while (r == s) {
    // Outer loop over r
    while (r == s) {
      // Inner loop over s
      yield <r, s>
      advance s
    }
    reset s to mark
    advance r
  }
}
```

sid	sname
22	dustin
28	yuppy
31	lubber
31	lubber2
44	guppy
57	rusty

sid	bid
28	103
28	104
31	101
31	102
42	142
58	107

sid	sname	bid
28	yuppy	103
28	yuppy	104

Sort-**Merge** Join

```
while not done {
  while (r < s) { advance r }
  while (r > s) { advance s }

  mark s // save start of “block”
  while (r == s) {
    // Outer loop over r
    while (r == s) {
      // Inner loop over s
      yield <r, s>
      advance s
    }
    reset s to mark
    advance r
  }
}
```

sid	sname
22	dustin
28	yuppy
31	lubber
31	lubber2
44	guppy
57	rusty

sid	bid
28	103
28	104
31	101
31	102
42	142
58	107

sid	sname	bid
28	yuppy	103
28	yuppy	104

Sort-Merge Join

```
while not done {
  while (r < s) { advance r }
  while (r > s) { advance s }

  mark s // save start of "block"
  while (r == s) {
    // Outer loop over r
    while (r == s) {
      // Inner loop over s
      yield <r, s>
      advance s
    }
    reset s to mark
    advance r
  }
}
```

sid	sname
22	dustin
28	yuppy
31	lubber
31	lubber2
44	guppy
57	rusty

sid	bid
28	103
28	104
31	101
31	102
42	142
58	107

sid	sname	bid
28	yuppy	103
28	yuppy	104

Sort-**Merge** Join

```
while not done {
  while (r < s) { advance r }
  while (r > s) { advance s }

  mark s // save start of “block”
  while (r == s) {
    // Outer loop over r
    while (r == s) {
      // Inner loop over s
      yield <r, s>
      advance s
    }
    reset s to mark
    advance r
  }
}
```

sid	sname
22	dustin
28	yuppy
31	lubber
31	lubber2
44	guppy
57	rusty

sid	bid
28	103
28	104
31	101
31	102
42	142
58	107

sid	sname	bid
28	yuppy	103
28	yuppy	104

Sort-**Merge** Join

```
while not done {
  while (r < s) { advance r }
  while (r > s) { advance s }

  mark s // save start of "block"
  while (r == s) {
    // Outer loop over r
    while (r == s) {
      // Inner loop over s
      yield <r, s>
      advance s
    }
    reset s to mark
    advance r
  }
}
```

sid	sname
22	dustin
28	yuppy
31	lubber
31	lubber2
44	guppy
57	rusty

sid	bid
28	103
28	104
31	101
31	102
42	142
58	107

sid	sname	bid
28	yuppy	103
28	yuppy	104

Sort-Merge Join

```
while not done {
  while (r < s) { advance r }
  while (r > s) { advance s }

  mark s // save start of "block"
  while (r == s) {
    // Outer loop over r
    while (r == s) {
      // Inner loop over s
      yield <r, s>
      advance s
    }
    reset s to mark
    advance r
  }
}
```

sid	sname
22	dustin
28	yuppy
31	lubber
31	lubber2
44	guppy
57	rusty

sid	bid
28	103
28	104
31	101
31	102
42	142
58	107

sid	sname	bid
28	yuppy	103
28	yuppy	104

Sort-**Merge** Join

```
while not done {
  while (r < s) { advance r }
  while (r > s) { advance s }

  mark s // save start of "block"
  while (r == s) {
    // Outer loop over r
    while (r == s) {
      // Inner loop over s
      yield <r, s>
      advance s
    }
    reset s to mark
    advance r
  }
}
```

sid	sname
22	dustin
28	yuppy
31	lubber
31	lubber2
44	guppy
57	rusty

sid	bid
28	103
28	104
31	101
31	102
42	142
58	107

sid	sname	bid
28	yuppy	103
28	yuppy	104
31	lubber	101

Sort-Merge Join

```
while not done {
  while (r < s) { advance r }
  while (r > s) { advance s }

  mark s // save start of "block"
  while (r == s) {
    // Outer loop over r
    while (r == s) {
      // Inner loop over s
      yield <r, s>
      advance s
    }
    reset s to mark
    advance r
  }
}
```

sid	sname
22	dustin
28	yuppy
31	lubber
31	lubber2
44	guppy
57	rusty

sid	bid
28	103
28	104
31	101
31	102
42	142
58	107

sid	sname	bid
28	yuppy	103
28	yuppy	104
31	lubber	101

Sort-**Merge** Join

```
while not done {
  while (r < s) { advance r }
  while (r > s) { advance s }

  mark s // save start of “block”
  while (r == s) {
    // Outer loop over r
    while (r == s) {
      // Inner loop over s
      yield <r, s>
      advance s
    }
    reset s to mark
    advance r
  }
}
```

sid	sname
22	dustin
28	yuppy
31	lubber
31	lubber2
44	guppy
57	rusty

sid	bid
28	103
28	104
31	101
31	102
42	142
58	107

sid	sname	bid
28	yuppy	103
28	yuppy	104
31	lubber	101
31	lubber	102

Sort-**Merge** Join

```
while not done {
  while (r < s) { advance r }
  while (r > s) { advance s }

  mark s // save start of "block"
  while (r == s) {
    // Outer loop over r
    while (r == s) {
      // Inner loop over s
      yield <r, s>
      advance s
    }
    reset s to mark
    advance r
  }
}
```

sid	sname
22	dustin
28	yuppy
31	lubber
31	lubber2
44	guppy
57	rusty

sid	bid
28	103
28	104
31	101
31	102
42	142
58	107

sid	sname	bid
28	yuppy	103
28	yuppy	104
31	lubber	101
31	lubber	102

Sort-**Merge** Join

```
while not done {
  while (r < s) { advance r }
  while (r > s) { advance s }

  mark s // save start of "block"
  while (r == s) {
    // Outer loop over r
    while (r == s) {
      // Inner loop over s
      yield <r, s>
      advance s
    }
    reset s to mark
    advance r
  }
}
```

sid	sname
22	dustin
28	yuppy
31	lubber
31	lubber2
44	guppy
57	rusty

sid	bid
28	103
28	104
31	101
31	102
42	142
58	107

sid	sname	bid
28	yuppy	103
28	yuppy	104
31	lubber	101
31	lubber	102

Sort-**Merge** Join

```
while not done {
  while (r < s) { advance r }
  while (r > s) { advance s }

  mark s // save start of “block”
  while (r == s) {
    // Outer loop over r
    while (r == s) {
      // Inner loop over s
      yield <r, s>
      advance s
    }
    reset s to mark
    advance r
  }
}
```

sid	sname
22	dustin
28	yuppy
31	lubber
31	lubber2
44	guppy
57	rusty

sid	bid
28	103
28	104
31	101
31	102
42	142
58	107

sid	sname	bid
28	yuppy	103
28	yuppy	104
31	lubber	101
31	lubber	102

Sort-Merge Join

```
while not done {
  while (r < s) { advance r }
  while (r > s) { advance s }

  mark s // save start of "block"
  while (r == s) {
    // Outer loop over r
    while (r == s) {
      // Inner loop over s
      yield <r, s>
      advance s
    }
    reset s to mark
    advance r
  }
}
```

sid	sname
22	dustin
28	yuppy
31	lubber
31	lubber2
44	guppy
57	rusty

sid	bid
28	103
28	104
31	101
31	102
42	142
58	107

sid	sname	bid
28	yuppy	103
28	yuppy	104
31	lubber	101
31	lubber	102
31	lubber2	101

Sort-**Merge** Join

```
while not done {
  while (r < s) { advance r }
  while (r > s) { advance s }

  mark s // save start of “block”
  while (r == s) {
    // Outer loop over r
    while (r == s) {
      // Inner loop over s
      yield <r, s>
      advance s
    }
    reset s to mark
    advance r
  }
}
```

sid	sname
22	dustin
28	yuppy
31	lubber
31	lubber2
44	guppy
57	rusty

sid	bid
28	103
28	104
31	101
31	102
42	142
58	107

sid	sname	bid
28	yuppy	103
28	yuppy	104
31	lubber	101
31	lubber	102
31	lubber2	101

Sort-Merge Join

```
while not done {
  while (r < s) { advance r }
  while (r > s) { advance s }

  mark s // save start of “block”
  while (r == s) {
    // Outer loop over r
    while (r == s) {
      // Inner loop over s
      yield <r, s>
      advance s
    }
    reset s to mark
    advance r
  }
}
```

sid	sname
22	dustin
28	yuppy
31	lubber
31	lubber2
44	guppy
57	rusty

sid	bid
28	103
28	104
31	101
31	102
42	142
58	107

sid	sname	bid
28	yuppy	103
28	yuppy	104
31	lubber	101
31	lubber	102
31	lubber2	101
31	lubber2	102

Sort-**Merge** Join

```
while not done {
  while (r < s) { advance r }
  while (r > s) { advance s }

  mark s // save start of "block"
  while (r == s) {
    // Outer loop over r
    while (r == s) {
      // Inner loop over s
      yield <r, s>
      advance s
    }
    reset s to mark
    advance r
  }
}
```

sid	sname
22	dustin
28	yuppy
31	lubber
31	lubber2
44	guppy
57	rusty

sid	bid
28	103
28	104
31	101
31	102
42	142
58	107

sid	sname	bid
28	yuppy	103
28	yuppy	104
31	lubber	101
31	lubber	102
31	lubber2	101
31	lubber2	102

Sort-**Merge** Join

```
while not done {
  while (r < s) { advance r }
  while (r > s) { advance s }

  mark s // save start of “block”
  while (r == s) {
    // Outer loop over r
    while (r == s) {
      // Inner loop over s
      yield <r, s>
      advance s
    }
    reset s to mark
    advance r
  }
}
```

sid	sname
22	dustin
28	yuppy
31	lubber
31	lubber2
44	guppy
57	rusty

sid	bid
28	103
28	104
31	101
31	102
42	142
58	107

sid	sname	bid
28	yuppy	103
28	yuppy	104
31	lubber	101
31	lubber	102
31	lubber2	101
31	lubber2	102

Sort-Merge Join

```
while not done {
  while (r < s) { advance r }
  while (r > s) { advance s }

  mark s // save start of "block"
  while (r == s) {
    // Outer loop over r
    while (r == s) {
      // Inner loop over s
      yield <r, s>
      advance s
    }
    reset s to mark
    advance r
  }
}
```

sid	sname
22	dustin
28	yuppy
31	lubber
31	lubber2
44	guppy
57	rusty

sid	bid
28	103
28	104
31	101
31	102
42	142
58	107

sid	sname	bid
28	yuppy	103
28	yuppy	104
31	lubber	101
31	lubber	102
31	lubber2	101
31	lubber2	102

Sort-**Merge** Join

```
while not done {
   while (r < s) { advance r }
   while (r > s) { advance s }

   mark s // save start of "block"
   while (r == s) {
      // Outer loop over r
      while (r == s) {
         // Inner loop over s
         yield <r, s>
         advance s
      }
      reset s to mark
      advance r
   }
}
```

sid	sname
22	dustin
28	yuppy
31	lubber
31	lubber2
44	guppy
57	rusty

sid	bid
28	103
28	104
31	101
31	102
42	142
58	107

sid	sname	bid
28	yuppy	103
28	yuppy	104
31	lubber	101
31	lubber	102
31	lubber2	101
31	lubber2	102

Sort-**Merge** Join

```
while not done {
  while (r < s) { advance r }
  while (r > s) { advance s }

  mark s // save start of "block"
  while (r == s) {
    // Outer loop over r
    while (r == s) {
      // Inner loop over s
      yield <r, s>
      advance s
    }
    reset s to mark
    advance r
  }
}
```

sid	sname
22	dustin
28	yuppy
31	lubber
31	lubber2
44	guppy
57	rusty

sid	bid
28	103
28	104
31	101
31	102
42	142
58	107

sid	sname	bid
28	yuppy	103
28	yuppy	104
31	lubber	101
31	lubber	102
31	lubber2	101
31	lubber2	102

Sort-**Merge** Join

```
while not done {
  while (r < s) { advance r }
  while (r > s) { advance s }

  mark s // save start of "block"
  while (r == s) {
    // Outer loop over r
    while (r == s) {
      // Inner loop over s
      yield <r, s>
      advance s
    }
    reset s to mark
    advance r
  }
}
```

sid	sname
22	dustin
28	yuppy
31	lubber
31	lubber2
44	guppy
57	rusty

sid	bid
28	103
28	104
31	101
31	102
42	142
58	107

sid	sname	bid
28	yuppy	103
28	yuppy	104
31	lubber	101
31	lubber	102
31	lubber2	101
31	lubber2	102

Sort-Merge Join

```
while not done {
  while (r < s) { advance r }
  while (r > s) { advance s }

  mark s // save start of “block”
  while (r == s) {
    // Outer loop over r
    while (r == s) {
      // Inner loop over s
      yield <r, s>
      advance s
    }
    reset s to mark
    advance r
  }
}
```

sid	sname
22	dustin
28	yuppy
31	lubber
31	lubber2
44	guppy
57	rusty

sid	bid
28	103
28	104
31	101
31	102
42	142
58	107

sid	sname	bid
28	yuppy	103
28	yuppy	104
31	lubber	101
31	lubber	102
31	lubber2	101
31	lubber2	102

Sort-Merge Join

```
while not done {
  while (r < s) { advance r }
  while (r > s) { advance s }

  mark s // save start of "block"
  while (r == s) {
    // Outer loop over r
    while (r == s) {
      // Inner loop over s
      yield <r, s>
      advance s
    }
    reset s to mark
    advance r
  }
}
```

sid	sname
22	dustin
28	yuppy
31	lubber
31	lubber2
44	guppy
57	rusty

sid	bid
28	103
28	104
31	101
31	102
42	142
58	107

sid	sname	bid
28	yuppy	103
28	yuppy	104
31	lubber	101
31	lubber	102
31	lubber2	101
31	lubber2	102

Sort-**Merge** Join

```
while not done {
  while (r < s) { advance r }
  while (r > s) { advance s }

  mark s // save start of "block"
  while (r == s) {
    // Outer loop over r
    while (r == s) {
      // Inner loop over s
      yield <r, s>
      advance s
    }
    reset s to mark
    advance r
  }
}
```

sid	sname
22	dustin
28	yuppy
31	lubber
31	lubber2
44	guppy
57	rusty

sid	bid
28	103
28	104
31	101
31	102
42	142
58	107

sid	sname	bid
28	yuppy	103
28	yuppy	104
31	lubber	101
31	lubber	102
31	lubber2	101
31	lubber2	102

Sort-**Merge** Join

```
while not done {
  while (r < s) { advance r }
  while (r > s) { advance s }

  mark s // save start of "block"
  while (r == s) {
    // Outer loop over r
    while (r == s) {
      // Inner loop over s
      yield <r, s>
      advance s
    }
    reset s to mark
    advance r
  }
}
```

sid	sname
22	dustin
28	yuppy
31	lubber
31	lubber2
44	guppy
57	rusty

sid	bid
28	103
28	104
31	101
31	102
42	142
58	107

sid	sname	bid
28	yuppy	103
28	yuppy	104
31	lubber	101
31	lubber	102
31	lubber2	101
31	lubber2	102

Sort-**Merge** Join

```
while not done {
  while (r < s) { advance r }
  while (r > s) { advance s }

  mark s // save start of "block"
  while (r == s) {
    // Outer loop over r
    while (r == s) {
      // Inner loop over s
      yield <r, s>
      advance s
    }
    reset s to mark
    advance r
  }
}
```

sid	sname
22	dustin
28	yuppy
31	lubber
31	lubber2
44	guppy
57	rusty

sid	bid
28	103
28	104
31	101
31	102
42	142
58	107

sid	sname	bid
28	yuppy	103
28	yuppy	104
31	lubber	101
31	lubber	102
31	lubber2	101
31	lubber2	102

Sort-Merge Join (SMJ)

- I/O cost?
  - Cost of sorting R
  - Cost of sorting S
  - The merge step: [R] + [S]
    - Only one pass (if we assume there aren’t a lot of duplicates)

Sort-Merge Join (SMJ)

- An optimization we can sometimes make
  - Recall materialization (write + read) is expensive
  - We only have to (assuming no duplicate values in R) make *one* pass through the sorted relation → we don't *need* the sorted relations to be materialized!
  - In the final merge pass of sorting both relations, instead of writing the sorted relations to disk, we can stream them into the second part of SMJ!
    - Reduces I/O cost by 2*([R] + [S])!

Sort-Merge Join (SMJ)

- An optimization we can sometimes make
  - In the final merge pass of sorting both relations, instead of writing the sorted relations to disk, we can stream them into the second part of SMJ!
    - Since we are iterating over R and S anyway, we can begin outputting what will join from the two relations
    - We have to be able to fit the input buffers of the last merge pass of sorting R *and* sorting S in memory, as well as have one output buffer for joined tuples
    - Need: # runs in last merge pass for R + # runs in last merge pass for S $\leq$ B - 1

Worksheet Q1e

In the average case, how many disk I/Os are needed to perform a sort-merge join (unoptimized/optimized)?

```
Companies: (company_id, industry, ipo_date)
Nyse: (company_id, date, trade, quantity)
```

- 20 pages of memory
- [N] = 100 pages, p_N = 100 tuples per page
- [C] = 50 pages, p_C = 50 tuples per page

Worksheet Q1e

How many disk I/Os are needed to perform a sort-merge join (unoptimized/optimized)?

```
Companies: (company_id, industry, ipo_date)
Nyse: (company_id, date, trade, quantity)
```

- 20 pages of memory
- [N] = 100 pages, p_N = 100 tuples per page
- [C] = 50 pages, p_C = 50 tuples per page

Unoptimized:

Sorting N:

Pass 1 - ceil(100/20) = 5 sorted runs of 20 pages each

Pass 2 - ceil(5/19) = 1 sorted run of 100 pages each

Total I/Os: 4 * (100 pages) = 400 I/Os

Sorting C:

Pass 1 - ceil(50/20) = 3 sorted runs of 20 pages, 20 pages, and 10 pages

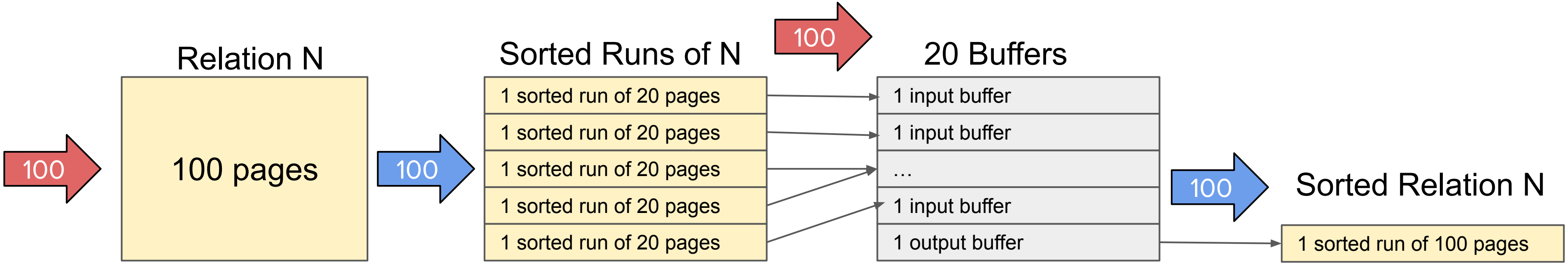
Pass 2 - ceil(3/19) = 1 sorted run of 50 pages

Total I/Os: 4 * (50 pages) = 200 I/Os

Merging: [C] + [N] = 150 I/Os

Total SMJ I/Os: 200 + 400 + 150 = 750 I/Os

Unoptimized SMJ

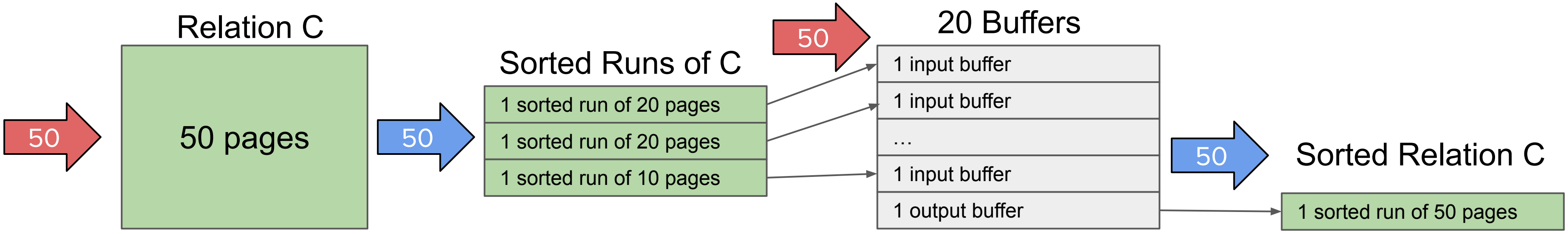


I/Os:

- Sorting N: 400
- Sorting C: ??
- Merging: ??

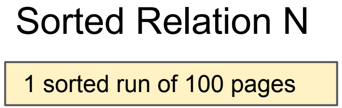
Read, Write, Merge

Unoptimized SMJ



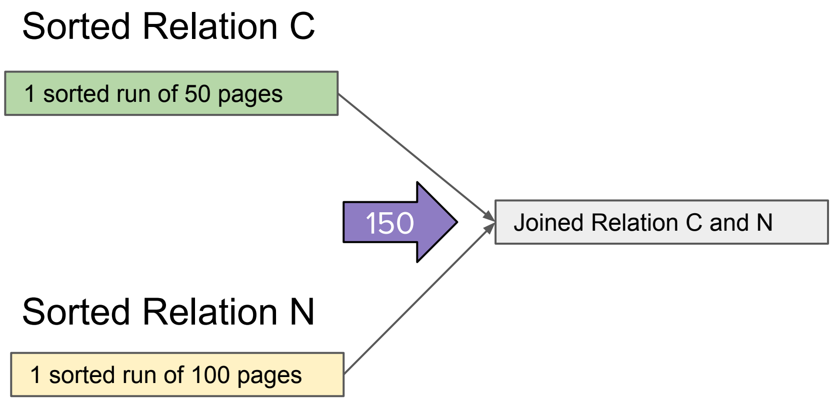
I/Os:

- Sorting N: 400
- Sorting C: 200
- Merging: ??



Read, Write, Merge

Unoptimized SMJ



I/Os: 750

- Sorting N: 400
- Sorting C: 200
- Merging: 150

Read, Write, Merge

Worksheet Q1e

How many disk I/Os are needed to perform a sort-merge join (unoptimized/optimized)?

```
Companies: (company_id, industry, ipo_date)
Nyse: (company_id, date, trade, quantity)
```

- 20 pages of memory
- [N] = 100 pages, p_N = 100 tuples per page
- [C] = 50 pages, p_C = 50 tuples per page

Can we perform the SMJ optimization?

Sorting N:

Pass 0 - ceil(100/20) = 5 sorted runs of 20 pages each

Pass 1 - ceil(5/19) = 1 sorted run of 100 pages each

Total I/Os: 4 * (100 pages) = 400 I/Os

Sorting C:

Pass 0 - ceil(50/20) = 3 sorted runs of 20 pages, 20 pages, and 10 pages

Pass 1 - ceil(3/19) = 1 sorted run of 50 pages

Total I/Os: 4 * (50 pages) = 200 I/Os

Worksheet Q1e

```
Companies: (company_id, industry, ipo_date)
Nyse: (company_id, date, trade, quantity)
```

- 20 pages of memory
- [N] = 100 pages, p_N = 100 tuples per page
- [C] = 50 pages, p_C = 50 tuples per page

How many disk I/Os are needed to perform a sort-merge join (unoptimized/optimized)?

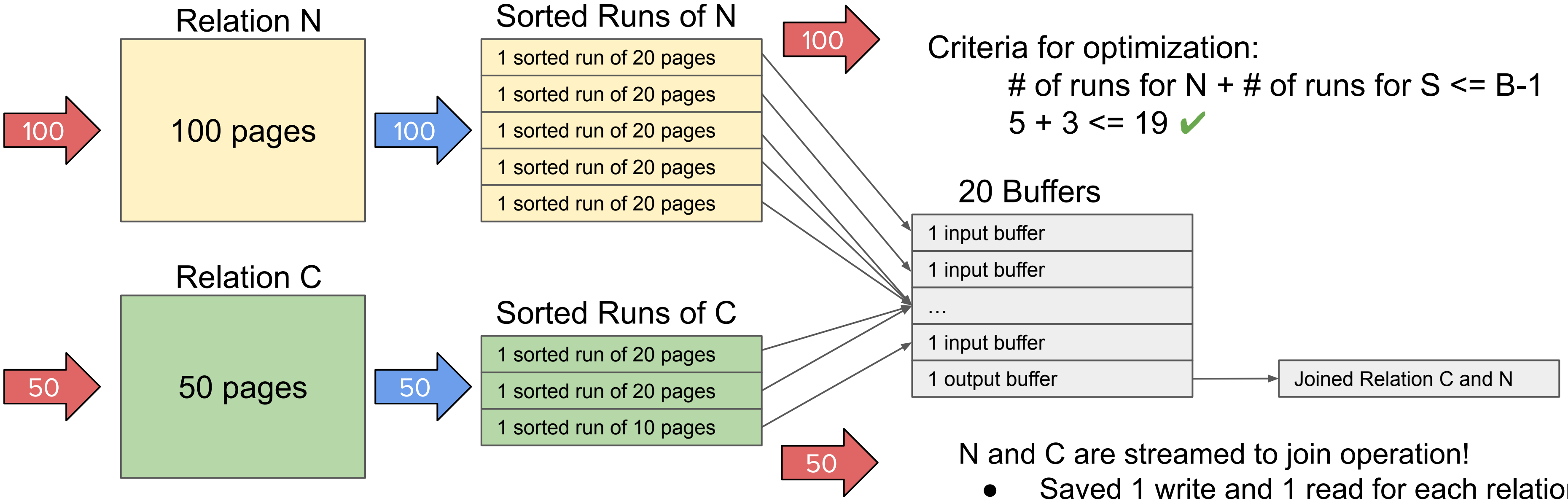
Can we perform the SMJ optimization?

Yes.

During the 2nd to last pass, we produce 5 sorted runs of N and 3 sorted runs of C. Since the number of runs of C + the number of runs of N ≤ 20 - 1, we can optimize sort merge join and combine the last sorting pass and final merging pass to save 2 * ([C] + [N]) I/Os.

Total I/Os = 750 - 2(50+100) = 450 I/Os

Optimized SMJ



Criteria for optimization:

# of runs for N + # of runs for S <= B-1

5 + 3 <= 19 ✔

N and C are streamed to join operation!

- Saved 1 write and 1 read for each relation
- i.e. saves 2([C] + [N]) I/Os

I/Os: cost(unoptimized SMJ) - 2([C] + [N])

= 750 - 2(150)

= 450

Read, Write

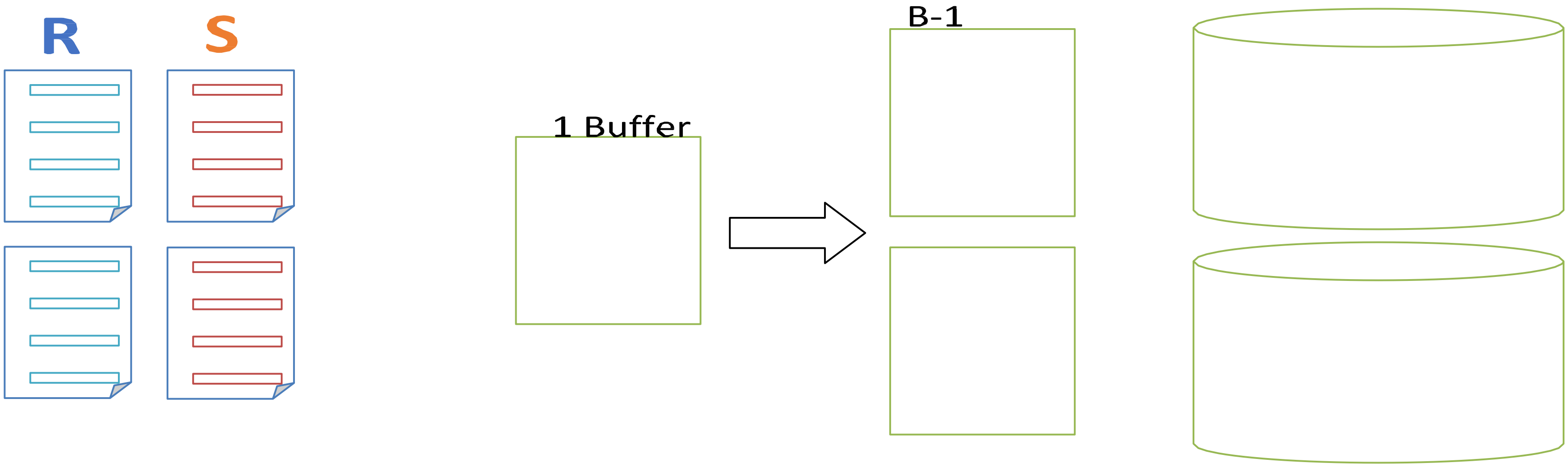
Grace Hash Join

- Same idea as SMJ, but let's build some hash tables instead
- Two passes: **partition** the data, then **build** an in-memory hash table and **probe** it
  - First, partition R and S into B - 1 partitions (like in external hashing), *using the same hash function*
    - All the tuples in R matching a tuple in S must be in the same partition → we can consider each partition independently

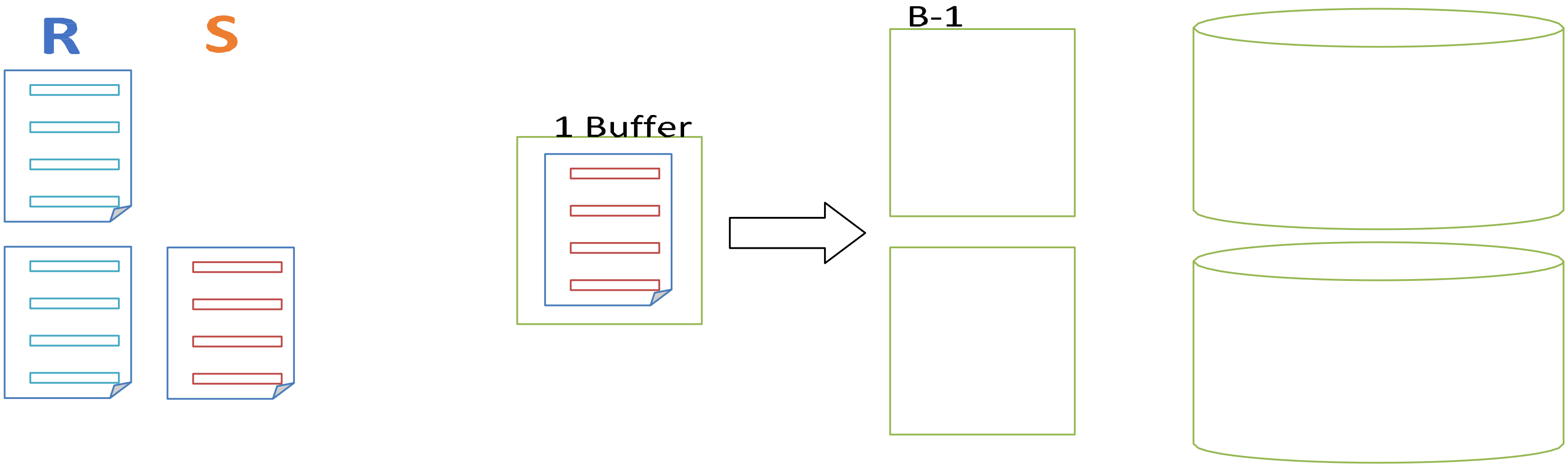
Grace Hash Join

- Same idea as SMJ, but let's build some hash tables instead
- Two passes: **partition** the data, then **build** an in-memory hash table and **probe** it
  - Then, build an in-memory hash table for a partition of R
  - We can use this in-memory hash table to find all the tuples in R that match a tuple in S
    - Stream in tuples of S, probe the hash table, output matching tuples

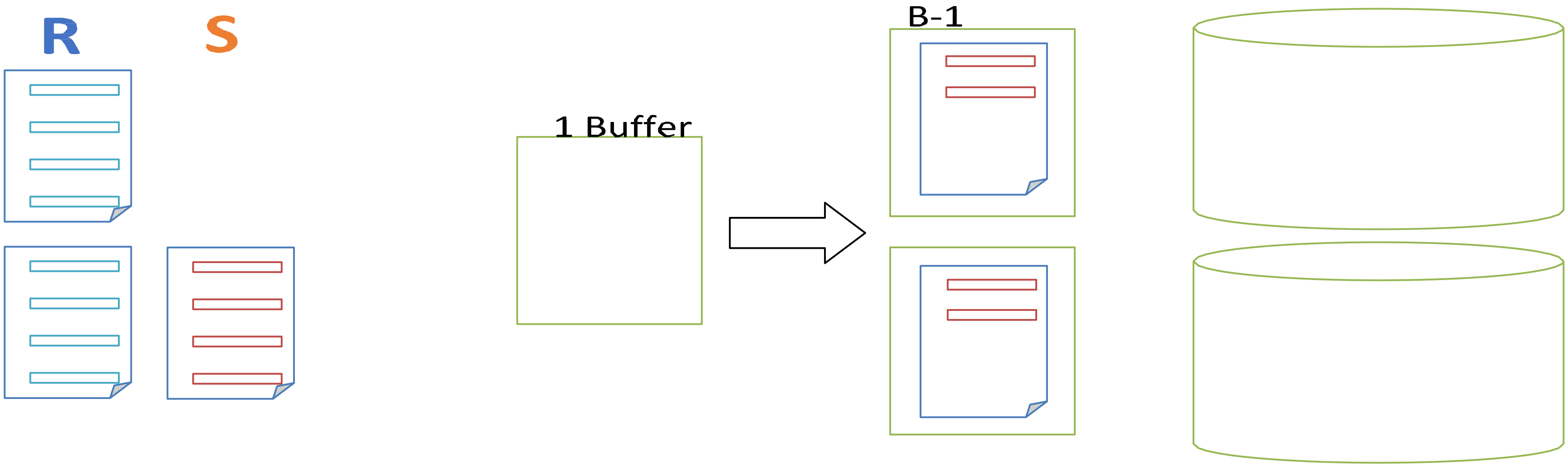
Grace Hash Join: ***Partition***



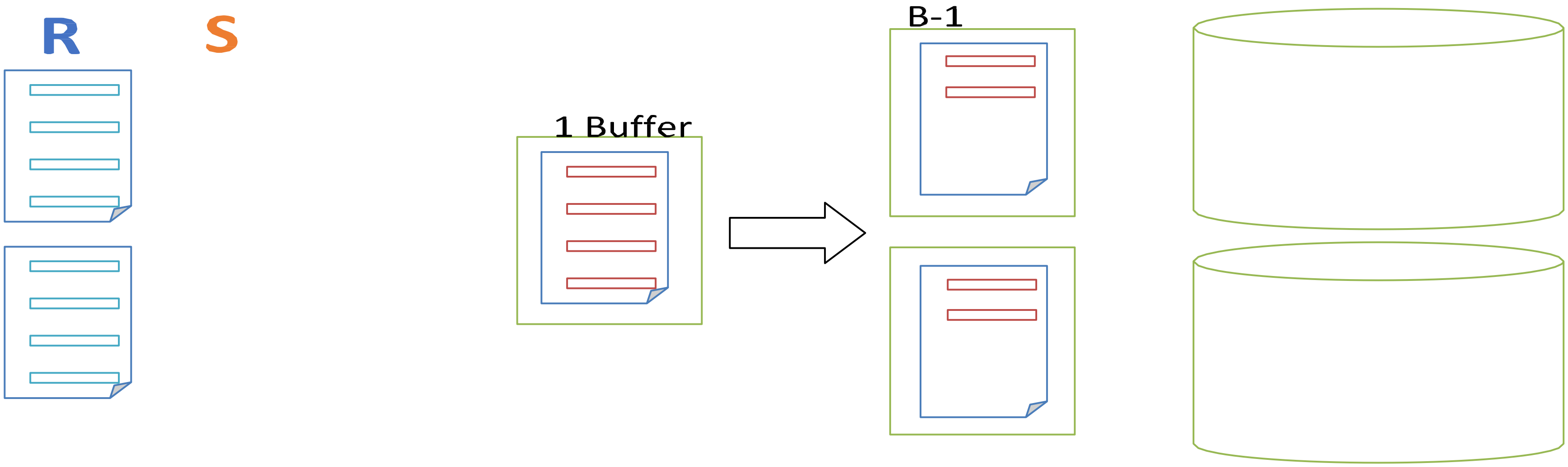
Grace Hash Join: ***Partition***



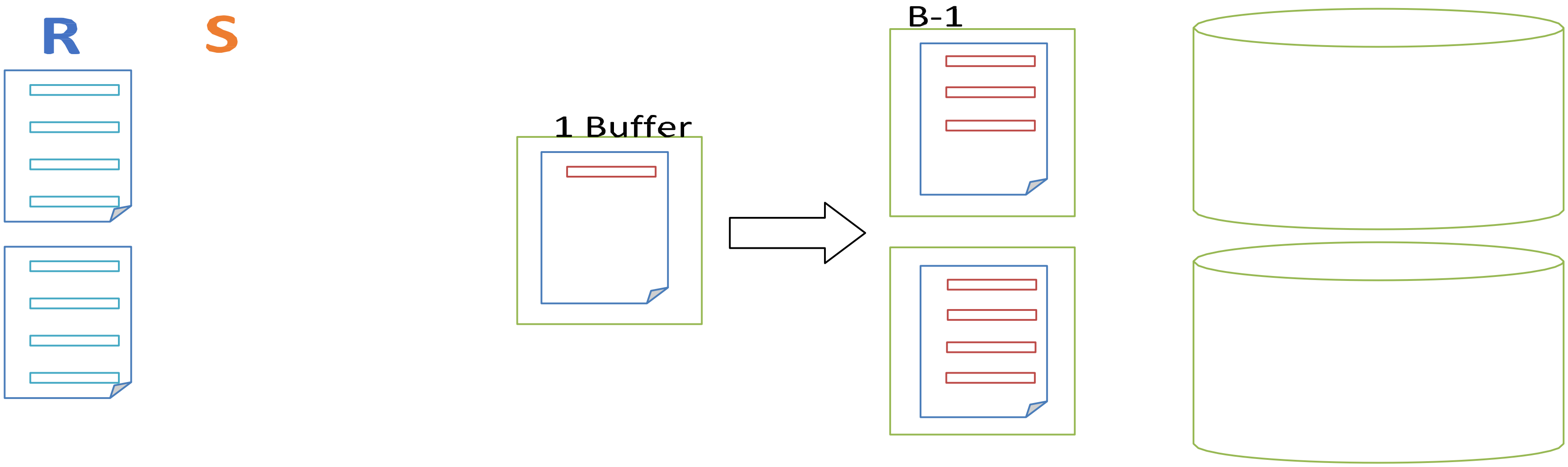
Grace Hash Join: *Partition*



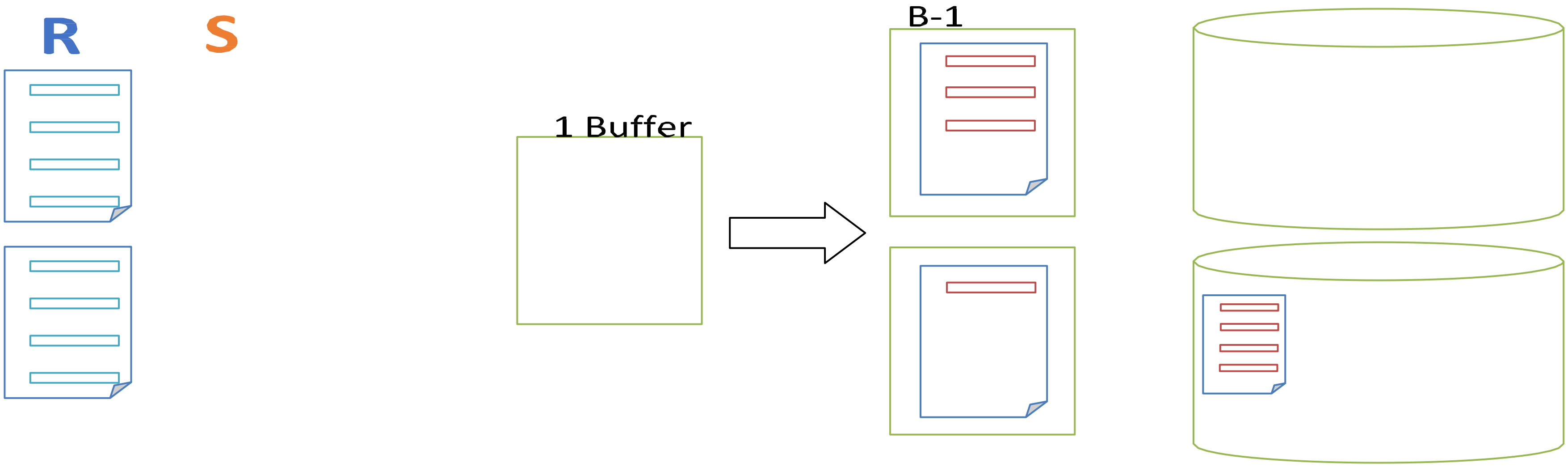
Grace Hash Join: *Partition*



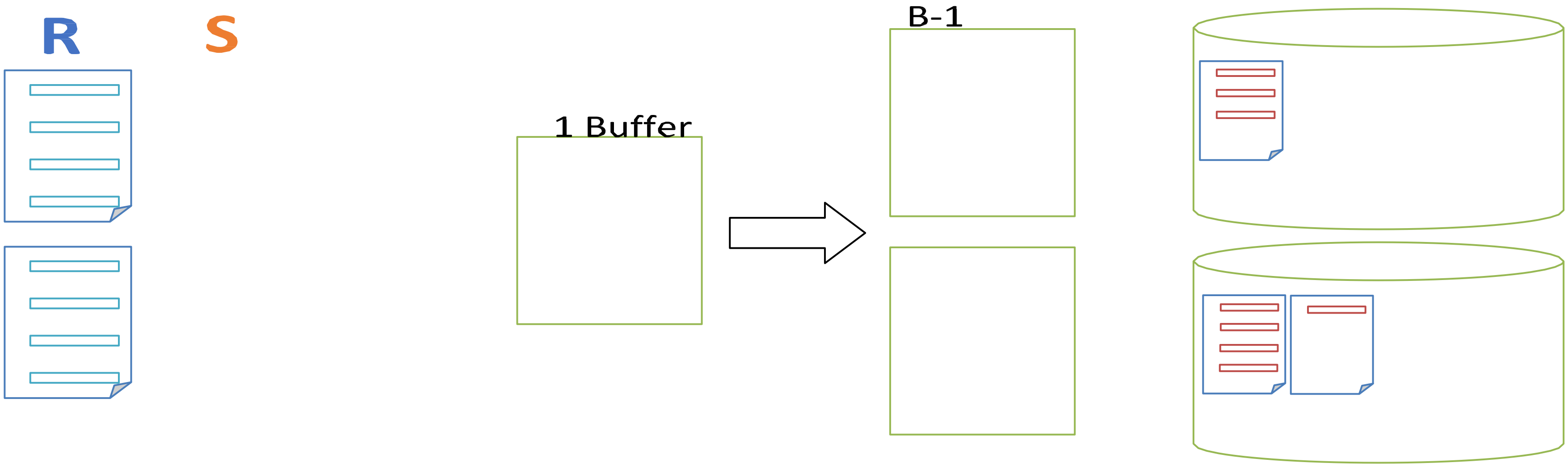
Grace Hash Join: ***Partition***



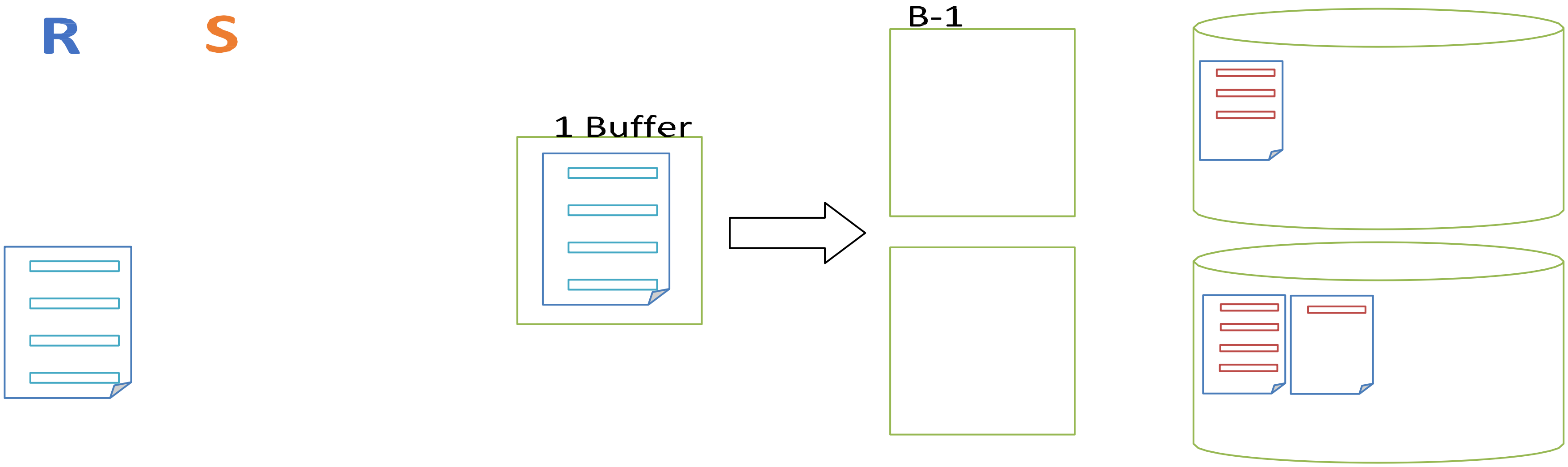
Grace Hash Join: *Partition*



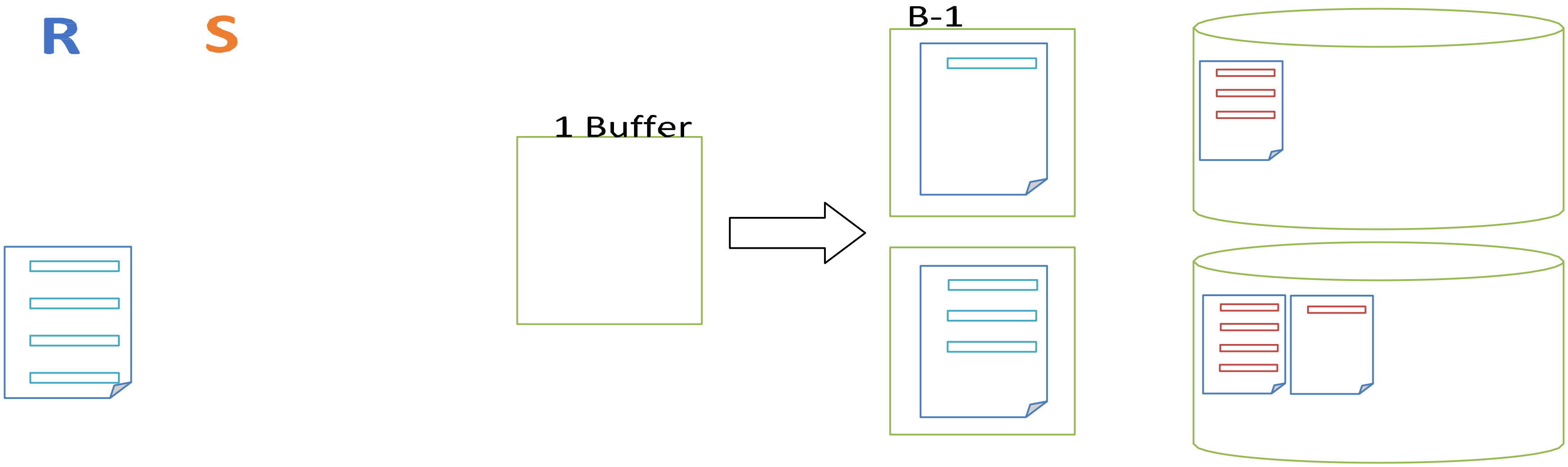
Grace Hash Join: *Partition*



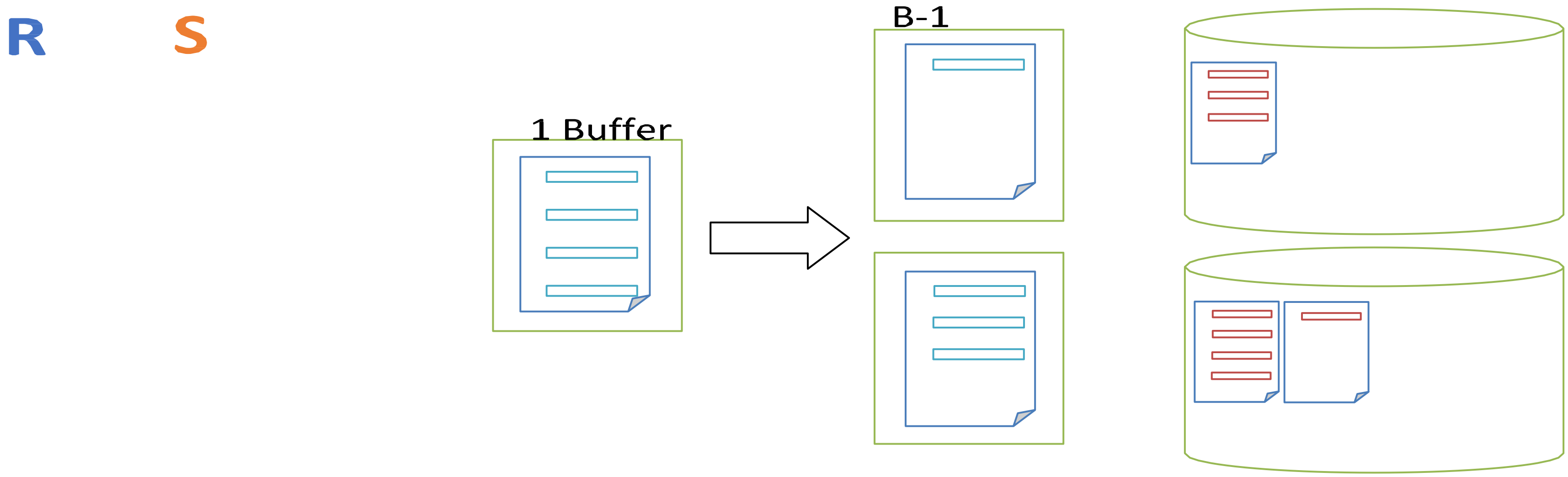
Grace Hash Join: ***Partition***



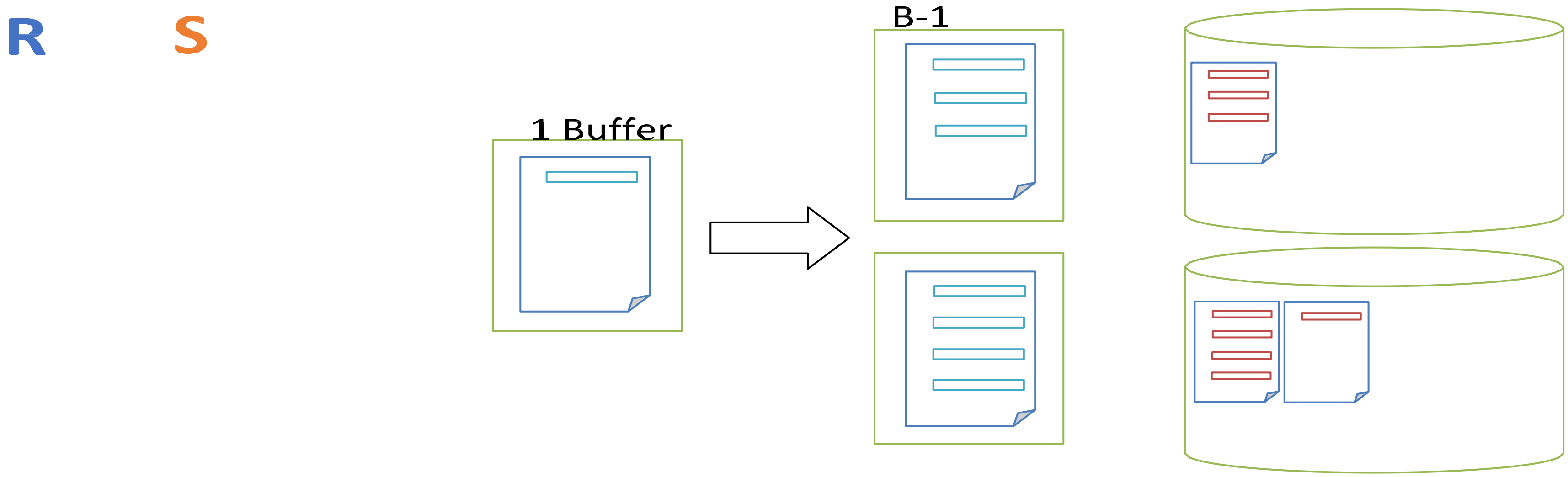
Grace Hash Join: ***Partition***



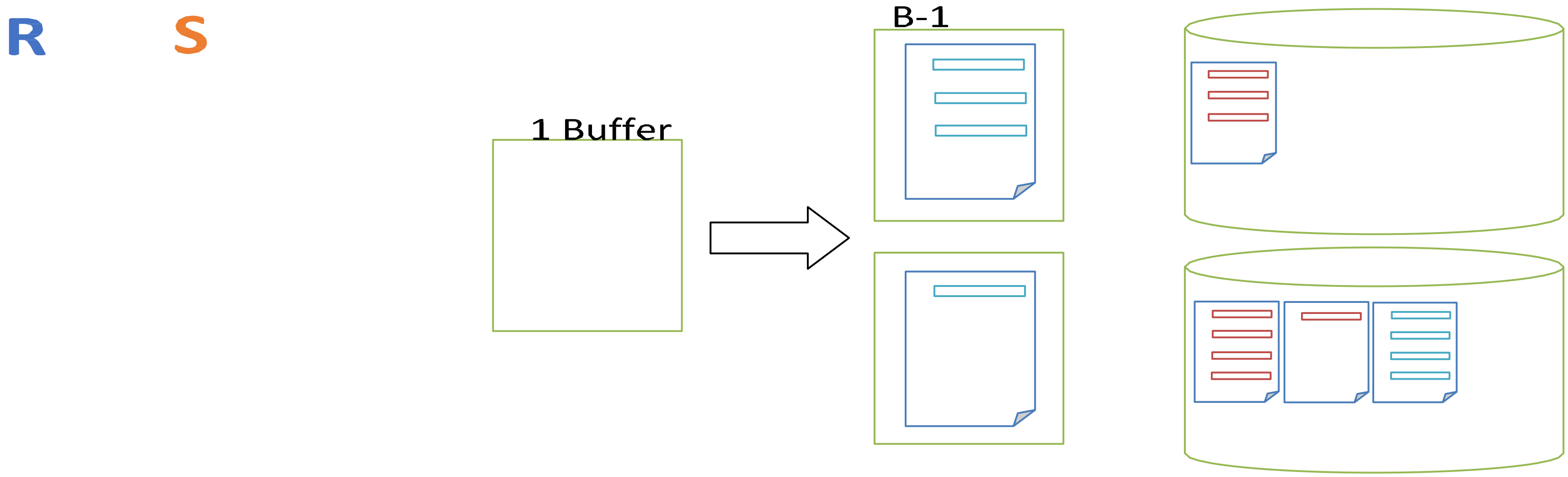
Grace Hash Join: ***Partition***



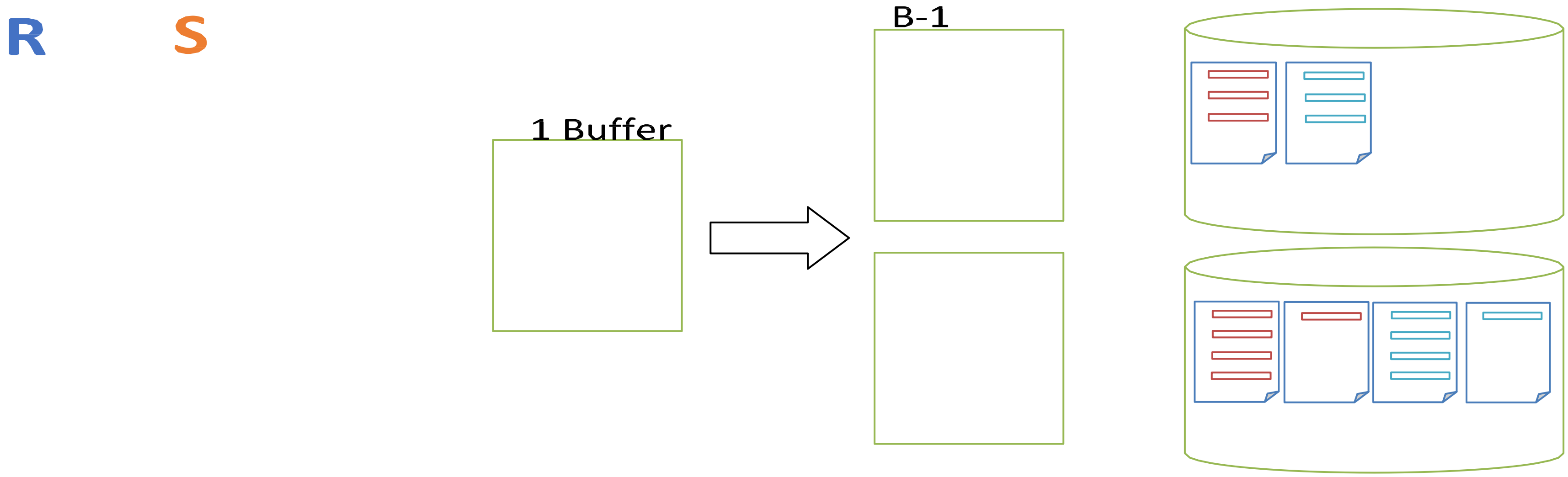
Grace Hash Join: ***Partition***



Grace Hash Join: ***Partition***



Grace Hash Join: *Partition*



Grace Hash Join

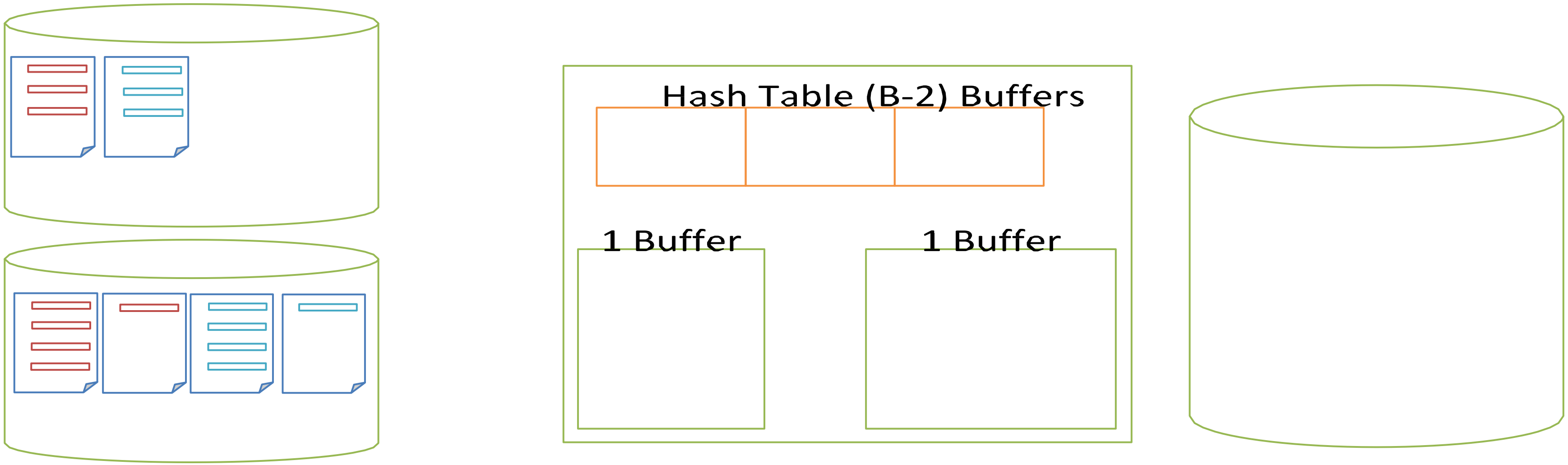
- We need partitions of R (but not S) to fit in B - 2 pages
  - 1 page reserved for streaming S partition
  - 1 page reserved for streaming output
- What if partitions of R are too big?
  - If S is smaller, do $S \bowtie_{\theta} R$ instead
  - Recursively partition! Make sure that for any partition of R you recursively partition, the matching S partition is also recursively partitioned!

Grace Hash Join

Pass 2: Build and Probe

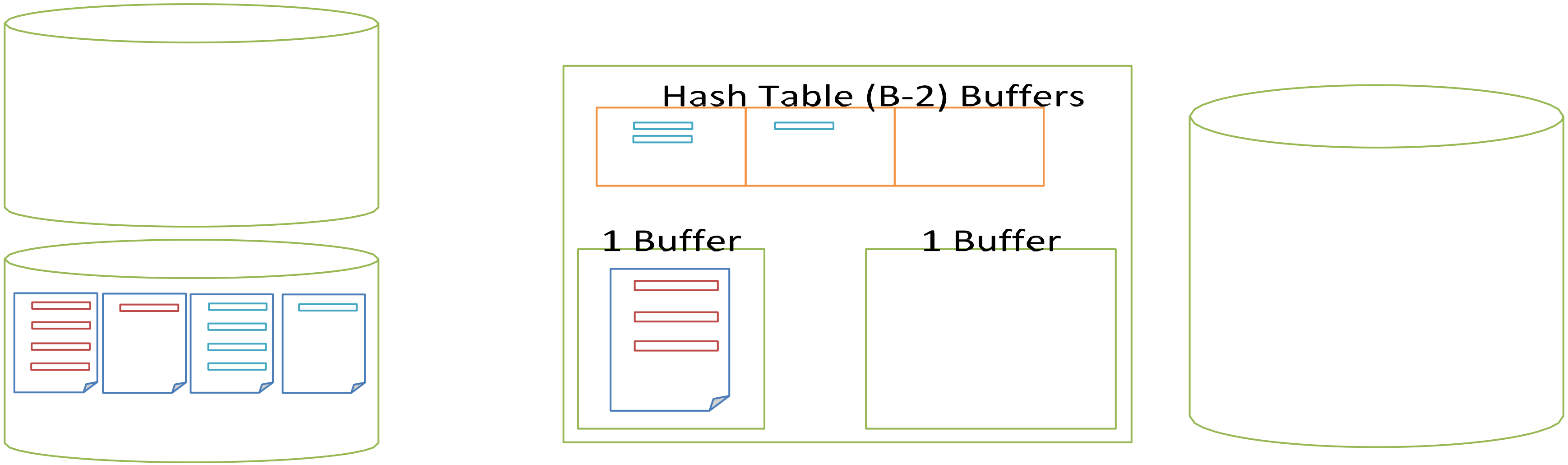
- Build an in-memory hash table for a partition of R
- Stream in tuples of S, probe the hash table, output matching tuples

Grace Hash Join: ***Build*** ***& Probe***



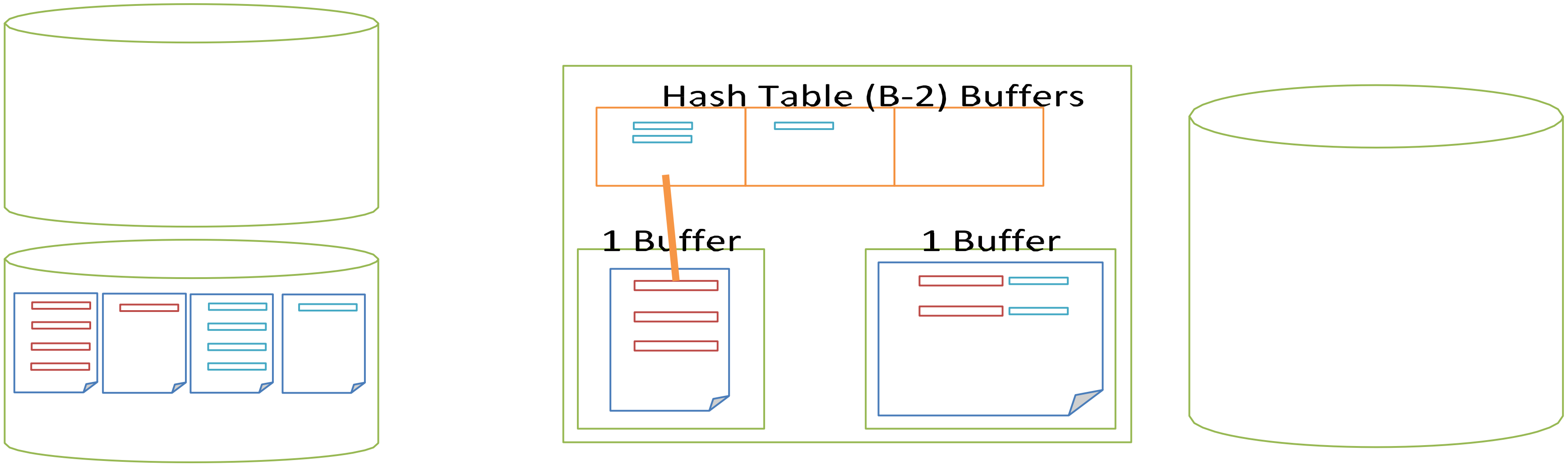
Build in-memory hash table of **R** and stream in tuples of **S**

Grace Hash Join: ***<u>Build</u>* & *Probe***



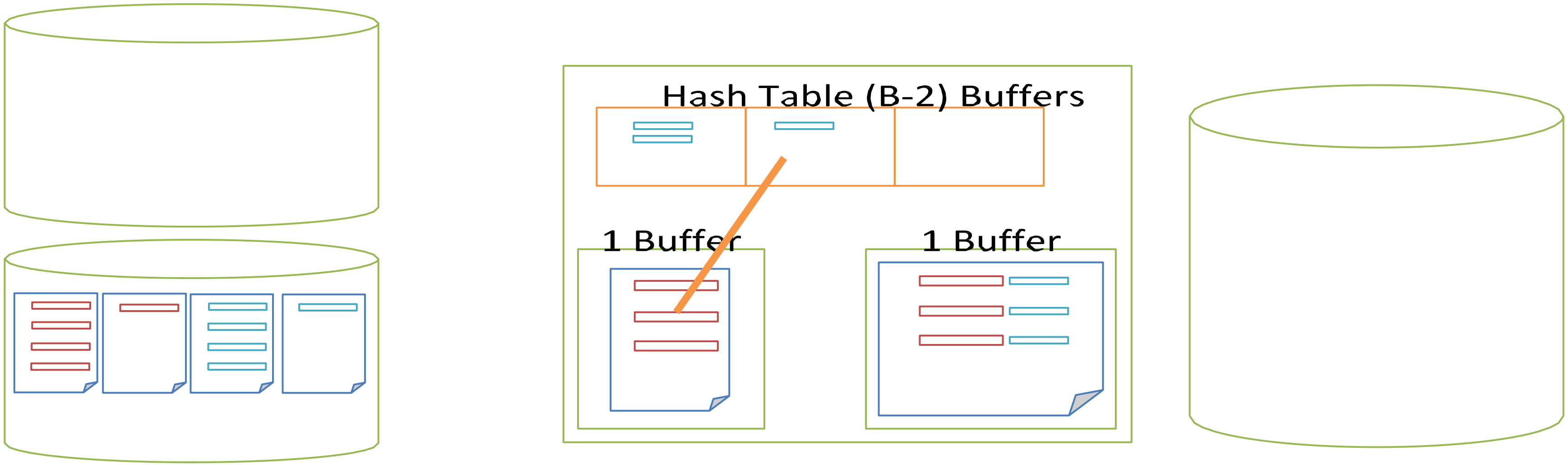
Build in-memory hash table of **R** and stream in tuples of **S**

Grace Hash Join: ***Build*** & ***<u>Probe</u>***



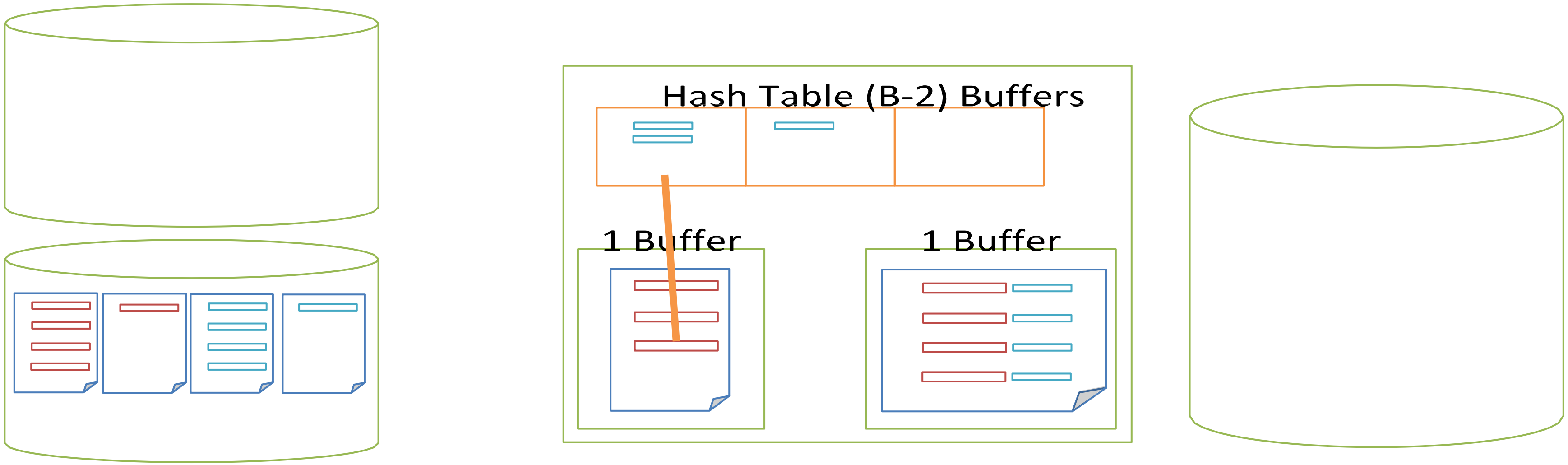
Probe in-memory hash table of **R** and stream out matching tuples of **S** and **R**

Grace Hash Join: ***Build & Probe***



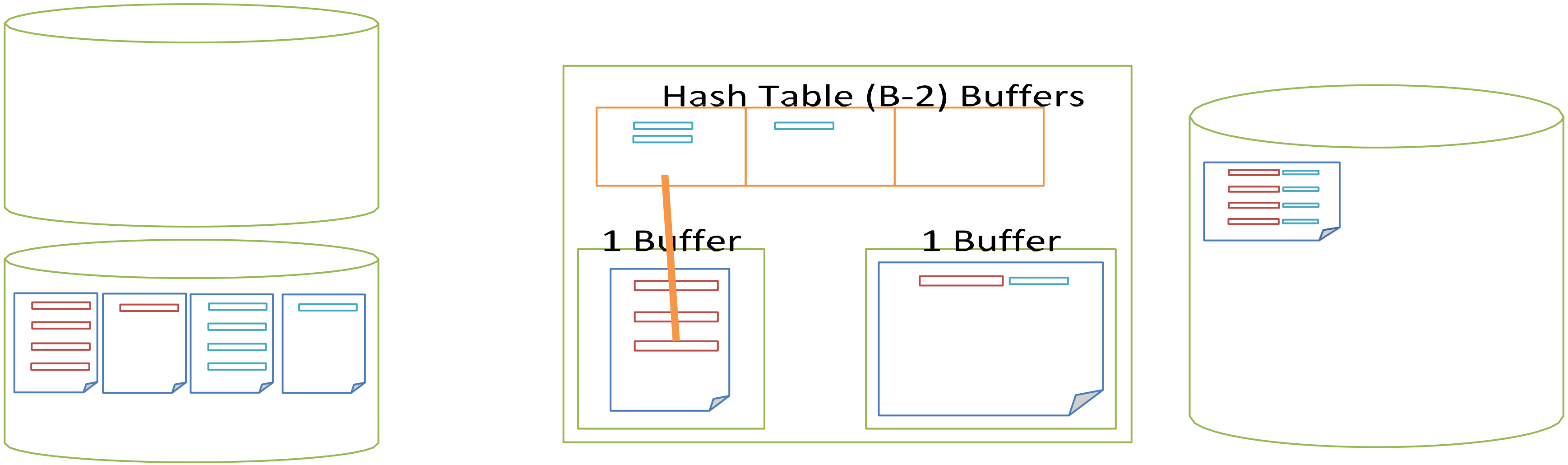
Probe in-memory hash table of **R** and stream out matching tuples of **S** and **R**

Grace Hash Join: ***Build*** & ***<u>Probe</u>***



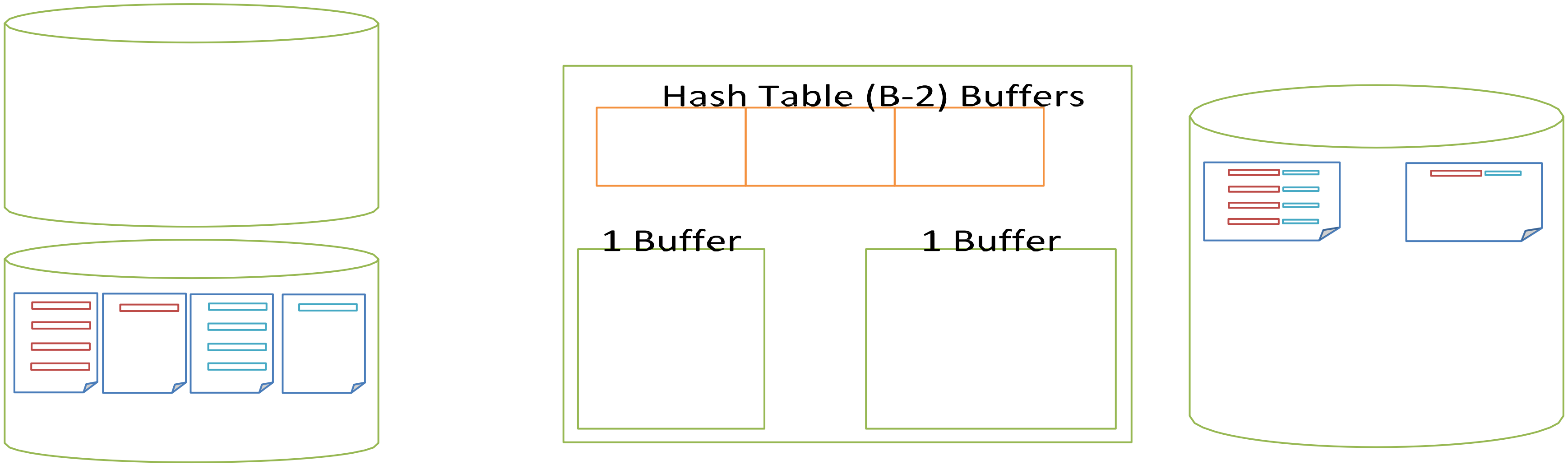
Probe in-memory hash table of **R** and stream out matching tuples of **S** and **R**

Grace Hash Join: ***Build*** **&** ***<u>Probe</u>***



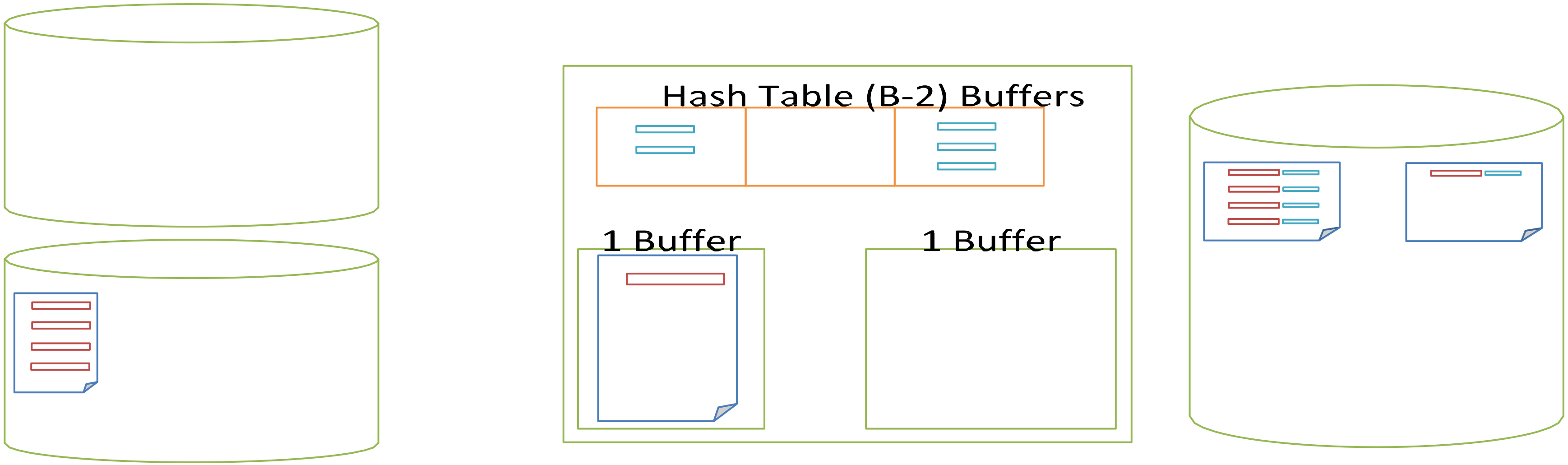
Probe in-memory hash table of **R** and stream out matching tuples of **S** and **R**

Grace Hash Join: ***Build*** & ***Probe***



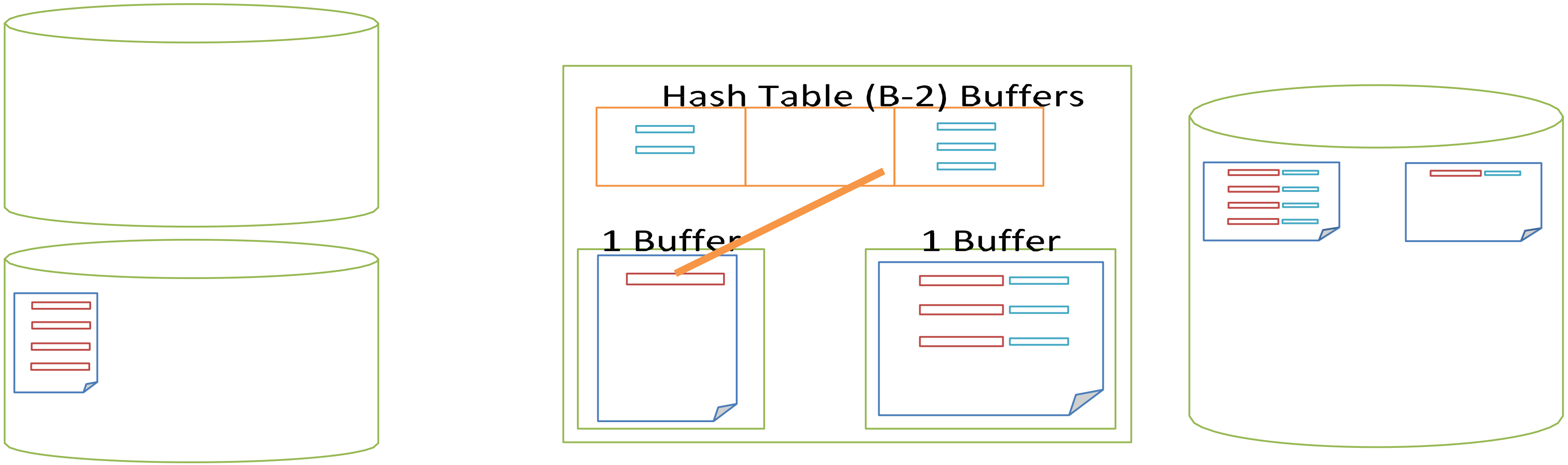
Build in-memory hash table of **R** and stream in tuples of **S**

Grace Hash Join: ***Build*** & ***Probe***



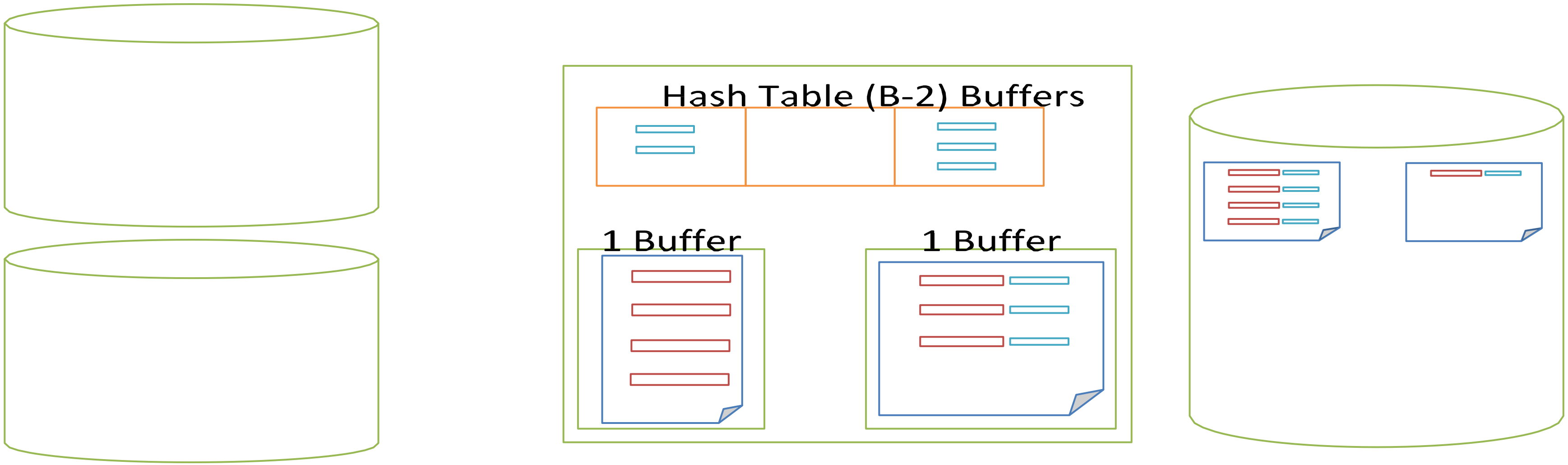
Build in-memory hash table of **R** and stream in tuples of **S**

Grace Hash Join: ***Build*** **&** ***<u>Probe</u>***



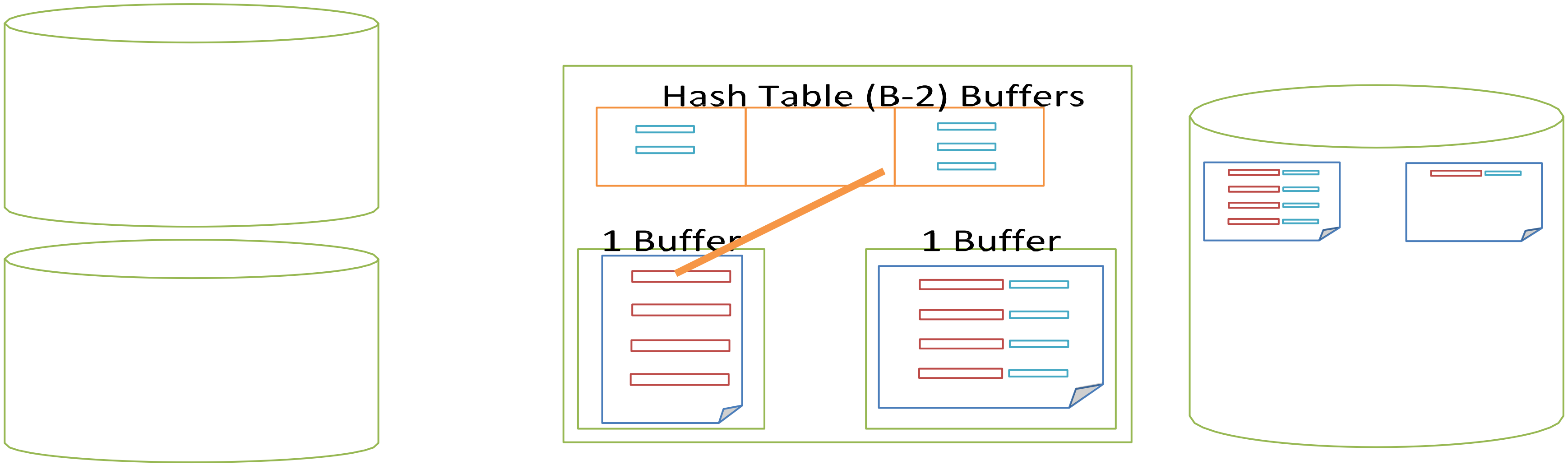
Probe in-memory hash table of **R** and stream out matching tuples of **S** and **R**

Grace Hash Join: ***Build & Probe***



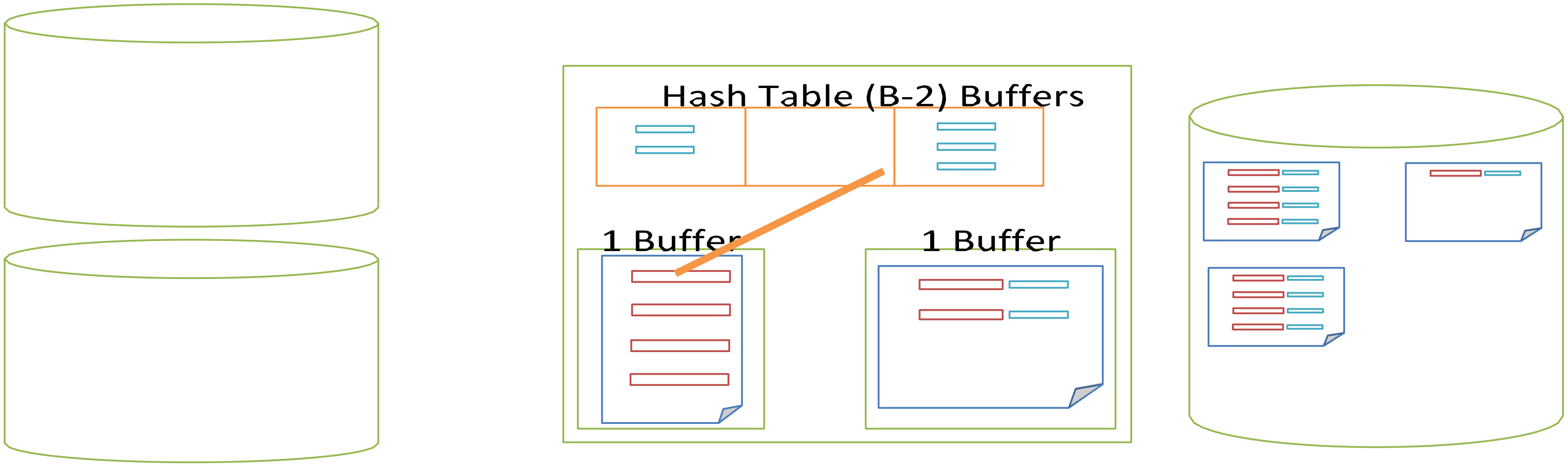
Probe in-memory hash table of **R** and stream out matching tuples of **S** and **R**

Grace Hash Join: ***Build & <u>Probe</u>***



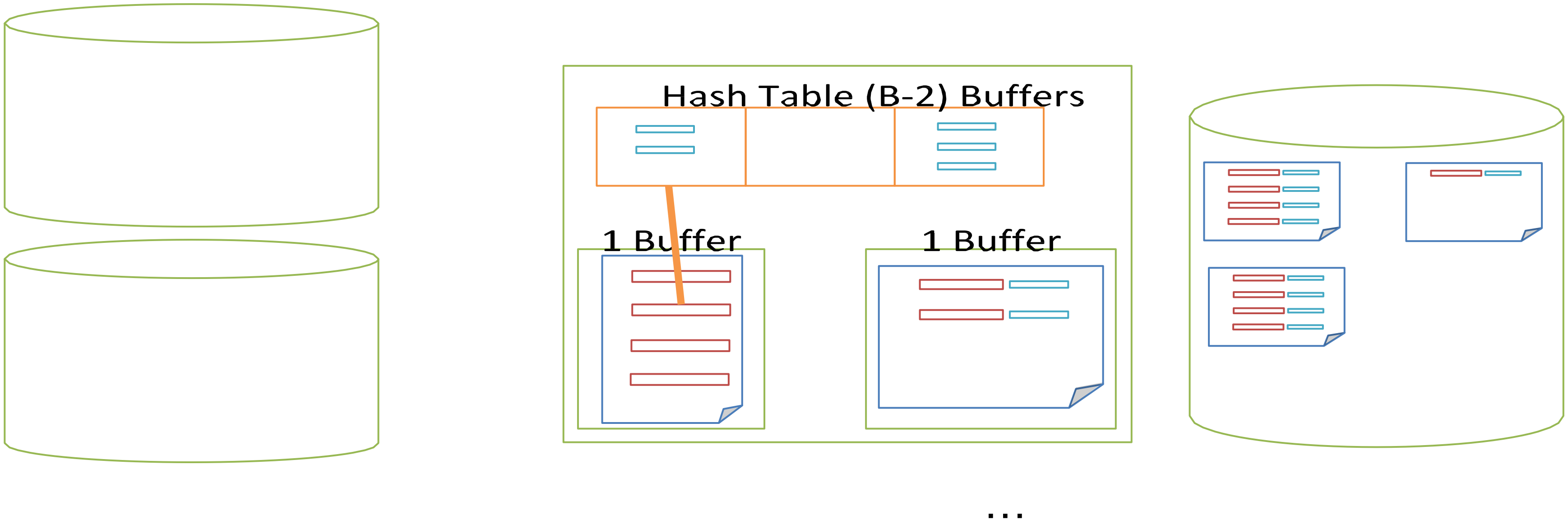
Probe in-memory hash table of **R** and stream out matching tuples of **S** and **R**

Grace Hash Join: ***Build & <u>Probe</u>***



Probe in-memory hash table of **R** and stream out matching tuples of **S** and **R**

Grace Hash Join: ***Build*** & ***<u>Probe</u>***



Probe in-memory hash table of **R** and stream out matching tuples of **S** and **R**

Worksheet

Worksheet Q2a

If we had 10 buffer pages, how many partitioning phases would we require for grace hash join?

- 2 tables: Catalog and Transactions
- [C] = 100 pages, p_C = 20 tuples per page
- [T] = 50 pages, p_T = 50 tuples per page
- Assume the hash functions uniformly distribute the data for both tables.

Worksheet Q2a

If we had 10 buffer pages, how many partitioning phases would we require for grace hash join?

- 2 tables: Catalog and Transactions
- [C] = 100 pages, p_C = 20 tuples per page
- [T] = 50 pages, p_T = 50 tuples per page
- Assume the hash functions uniformly distribute the data for both tables.

T is smaller, so we need its partitions to be at most B - 2 = 8 pages. After 1 partitioning pass, we have partitions of size 6, which is <= 8 so we only need **1 partitioning pass**.

Worksheet Q2b

What is the IO cost for the grace hash join then? Assume uniform partitioning.

- 2 tables: Catalog and Transactions
- [C] = 100 pages, p_C = 20 tuples per page
- [T] = 50 pages, p_T = 50 tuples per page
- Assume the hash functions uniformly distribute the data for both tables.

Worksheet Q2b

What is the IO cost for the grace hash join then? Assume uniform partitioning.

- 2 tables: Catalog and Transactions
- [C] = 100 pages, p_C = 20 tuples per page
- [T] = 50 pages, p_T = 50 tuples per page
- Assume the hash functions uniformly distribute the data for both tables.

We need 1 partitioning pass.

Partitioning phase:

ceil([C]/(B - 1)) = 12 pages per partition for C, 12(9) pages in total after partitioning

ceil([T]/(B - 1)) = 6 pages per partition for T, 6(9) pages in total after partitioning

Partitioning IOs: 100 I/Os to read from Catalog + 12(9) to write for Catalog + 50 I/Os to read from Transactions + 6(9) to write for Transactions = 312 I/Os

Probing phase: 12(9) + 6(9) = 162 I/Os to read from Catalog and Transactions

Total: 312 + 162 = 474 I/Os

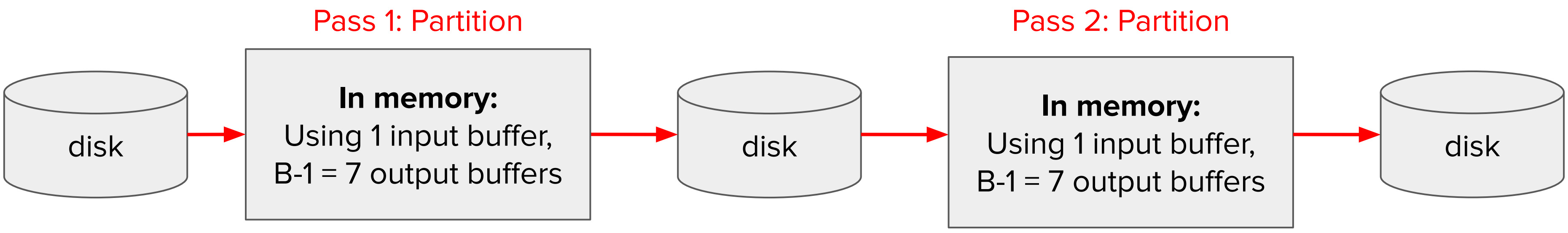
Worksheet Q2c

If we only had 8 buffer pages, how many partitioning phases would there be?

- 2 tables: Catalog and Transactions
- [C] = 100 pages, p_C = 20 tuples per page
- [T] = 50 pages, p_T = 50 tuples per page
- Assume the hash functions uniformly distribute the data for both tables.

Worksheet Q2c,d

- B = 8
- [C] = 100 pages
- [T] = 50 pages
- Assume the hash functions uniformly distribute the data for both tables.



Read into memory

100 pages of C	50 pages of T

Write to disk   Read into memory

7 partitions, each with

15 pages of C	8 pages of T

The partitions for neither table fit in B-2 = 6 pages, so we must **recursively partition**.

Write to disk

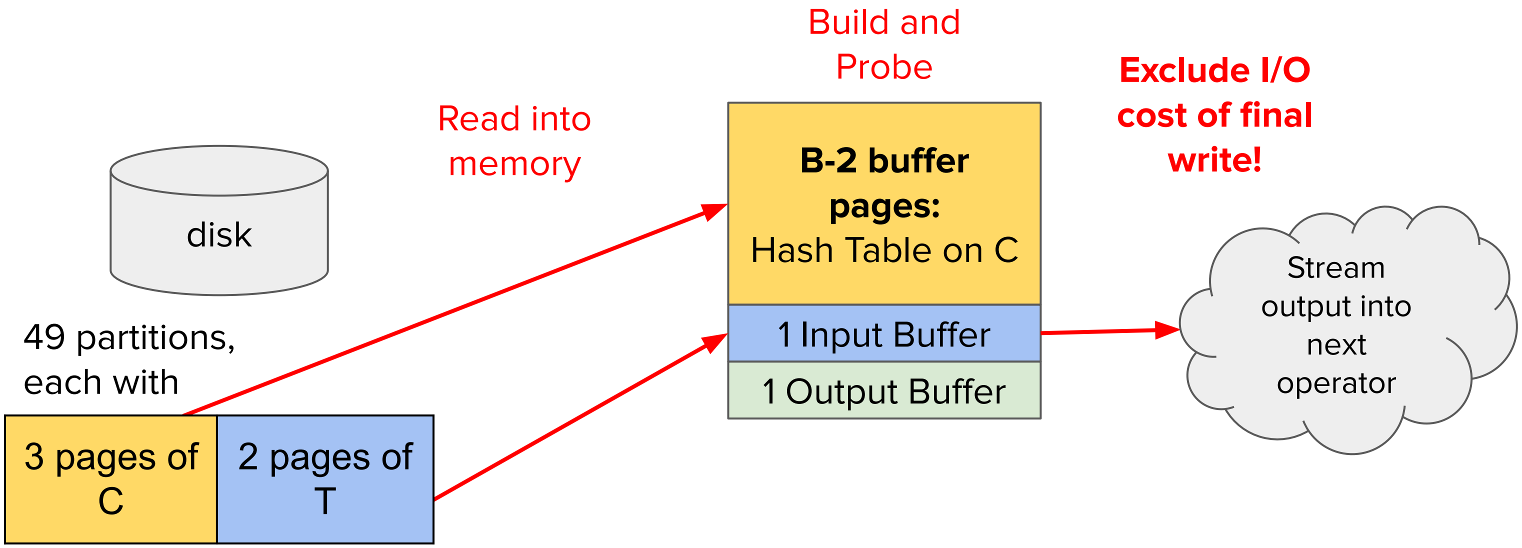
49 partitions, each with

3 pages of C	2 pages of T

The partitions for at least 1 table fit in B-2 = 6 pages, so we can enter the Build and Probe phase.

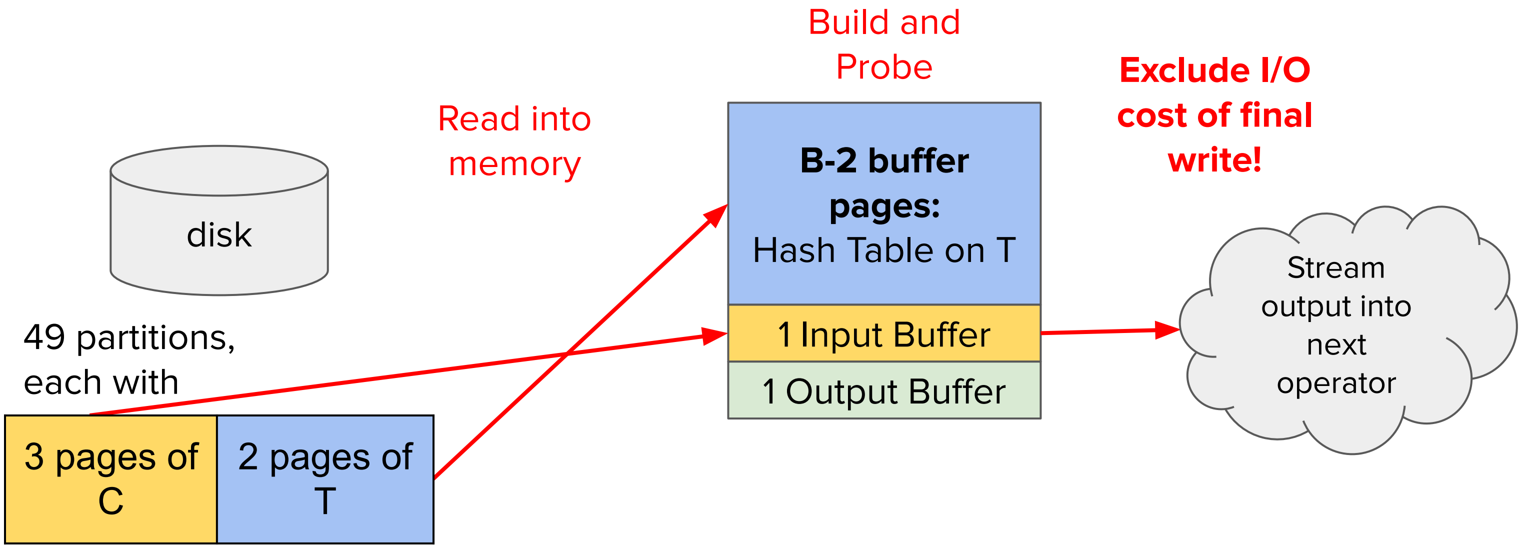
Worksheet Q2c,d

- B = 8
- [C] = 100 pages
- [T] = 50 pages
- Assume the hash functions uniformly distribute the data for both tables.



Worksheet Q2c,d

- B = 8
- [C] = 100 pages
- [T] = 50 pages
- Assume the hash functions uniformly distribute the data for both tables.



Note: We can alternatively build a hash table on T and probe C since partitions for either relation fit in B-2 pages.

Worksheet Q2c

If we only had 8 buffer pages, how many partitioning phases would there be?

- 2 tables: Catalog and Transactions
- [C] = 100 pages, p_C = 20 tuples per page
- [T] = 50 pages, p_T = 50 tuples per page
- Assume the hash functions uniformly distribute the data for both tables.

T is smaller, so we need its partitions to be at most B - 2 = 6 pages. After 1 partitioning pass, we have partitions of size 8, which is too big to fit in B-2 buffer pages. We need a second partitioning pass. 8 / 7 = 1.1→2 pages, which is small enough to fit in B-2 buffer pages.

Therefore, we need **2 passes** in total.

Worksheet Q2d

What will be the IO cost?

- 2 tables: Catalog and Transactions
- [C] = 100 pages, p_C = 20 tuples per page
- [T] = 50 pages, p_T = 50 tuples per page
- Assume the hash functions uniformly distribute the data for both tables.

Worksheet Q2d

What will be the IO cost?

- 2 tables: Catalog and Transactions
- [C] = 100 pages, p_C = 20 tuples per page
- [T] = 50 pages, p_T = 50 tuples per page
- Assume the hash functions uniformly distribute the data for both tables.

Partitioning phase:

ceil([C]/(B - 1)) = 15 pages per partition for C

ceil([T]/(B - 1)) = 8 pages per partition for T

ceil([C]/(B - 1)) = 3 pages per partition for second pass for C

ceil([T]/(B - 1)) = 2 pages per partition for second pass for T

	Read 1st	Write 1st	Read 2nd	Write 2nd
Partitioning IOs:	[100 + 50] +	[15(7) + 8(7)] +	[15(7) + 8(7)] +	[3(49) + 2(49)] = 717 I/Os

Build and Probe Phase: 3(49) + 2(49) = 245 IOs

Total: 717 + 245 = **962 I/Os**

Attendance Link

https://cs186berkeley.net/attendance