University of Edinburgh
School of Informatics
INFR11199 - Advanced Database Systems (Spring 2024)

Tutorial Sheet 1

1. (Single-Table SQL) Consider a database of published scientific papers with the following schema, where each primary key is underlined:

   Conference(<u>conference_id</u>, conference_name, organiser)

   Paper(<u>paper_id</u>, title, field, citations, year_published, conference_id)

   Ownership(<u>paper_id</u>, <u>researcher_id</u>)

   Researcher(<u>researcher_id</u>, researcher_name, affiliation, email)

   Write a SQL query for each task below.

   (a) Find the 10 papers with the highest numbers of citations, ordered from most to least. Break ties by paper title in alphabetical order.

   (b) Find the name and email for every researcher whose affiliation starts with the string 'University of'.

   (c) Find the total number of published papers per research field.

   (d) Find the total number of published papers per research field. Do not report fields with less than 10 papers.

   (e) Find the research field with the highest number of papers published after the year 2020. Assume there are no ties.

2. (Multi-Table SQL) Consider the same database schema from the previous question.

   Write a SQL query for each task below.

   (a) Find the name of all conferences that featured database papers in 2024.

   (b) Find the name of the conference that featured the paper with the highest number of citations. Assume there is only one such paper.

   (c) For each researcher, find the researcher name and the highest citation count of one of their paper. Include researchers that have not published a paper.

3. (CQ Minimization) Consider the CQ

$$Q(x,y) \text{ :- } R(y,x,z), R(w,v,z), T(x,z), T(x,a), R(y,x,a)$$

where x, y, z, v, w are variables and a is a constant value. Compute the minimal CQ of Q.

4. (Acyclicity) Consider the CQ

$$Q(x,y) \text{ :- } T(x,y,z), R(y,z), P(y,v,w), R(z,u).$$

Prove that Q is acyclic. In particular, apply the GYO-reduction to the hypergraph $H(Q)$ of Q and show that this leads to the empty graph. Give also the obtained join tree of $H(Q)$.

University of Edinburgh
School of Informatics
INFR11199 - Advanced Database Systems (Spring 2024)

Tutorial Sheet 1

1. (Single-Table SQL) Consider a database of published scientific papers with the following schema, where each primary key is underlined:

   Conference(<u>conference_id</u>, conference_name, organiser)

   Paper(<u>paper_id</u>, title, field, citations, year_published, conference_id)

   Ownership(<u>paper_id</u>, <u>researcher_id</u>)

   Researcher(<u>researcher_id</u>, researcher_name, affiliation, email)

   Write a SQL query for each task below.

   (a) Find the 10 papers with the highest numbers of citations, ordered from most to least. Break ties by paper title in alphabetical order.

   Solution:

   ```
   SELECT *
   FROM Paper
   ORDER BY citations DESC, title ASC
   LIMIT 10;
   ```

   (b) Find the name and email for every researcher whose affiliation starts with the string 'University of'.

   Solution:

   ```
   SELECT researcher_name, email
   FROM Researcher
   WHERE affiliation LIKE 'University of%';
   ```

(c) Find the total number of published papers per research field.

Solution:

```
SELECT field, COUNT(*)
FROM Paper
GROUP BY field;
```

(d) Find the total number of published papers per research field. Do not report fields with less than 10 papers.

Solution:

```
SELECT field, COUNT(*)
FROM Paper
GROUP BY field
HAVING COUNT(*) >= 10;
```

(e) Find the research field with the highest number of papers published after the year 2020. Assume there are no ties.

order by default is ASC

Solution:

```
SELECT field
FROM Paper
WHERE year_published > 2000
GROUP BY field
ORDER BY COUNT(*) DESC
LIMIT 1;
```

2. (Multi-Table SQL) Consider the same database schema from the previous question. Write a SQL query for each task below.

(a) Find the name of all conferences that featured database papers in 2024.

Solution:

```
SELECT conference_name
FROM Conference INNER JOIN Paper
  ON Conference.conference_id = Paper.conference_id
WHERE field = 'databases' AND year_published = 2024
GROUP BY conference_id, conference_name;
```

The same query can be expressed using an alternative join syntax:

```
SELECT conference_name
FROM Conference, Paper
WHERE Conference.conference_id = Paper.conference_id AND
      field = 'databases' AND year_published = 2024
GROUP BY conference_id, conference_name;
```

If a conference publishes multiple database papers in 2024, we need to make sure the conference_name appears only once in the result. However, we cannot use 'DISTINCT conference_name' to resolve this, because if there are 2 conferences that have different conference_ids but share the same conference_name, we need to make sure the conference_name shows up in the results twice (once for each unique conference_id), which is why we include the GROUP BY clause.

(b) Find the name of the conference that featured the paper with the highest number of citations. Assume there is only one such paper.

Solution:

```
SELECT conference_name
FROM Conference INNER JOIN Paper
  ON Conference.conference_id = Paper.conference_id
ORDER BY citations DESC
LIMIT 1;
```

(c) For each researcher, find the researcher name and the highest citation count of one of their paper. Include researchers that have not published a paper.

Solution:

```
SELECT researcher_name, MAX(citations)
FROM Researcher LEFT OUTER JOIN
    (Paper INNER JOIN Ownership
      ON Paper.paper_id = Ownership.paper_id)
  ON Researcher.researcher_id = Ownership.researcher_id
GROUP BY Researcher.researcher_id, researcher_name;
```

Same idea as (a) researcher_name may not be unique

3. (CQ Minimization) Consider the CQ

$$Q(x,y) \text{ :- } R(y,x,z), R(w,v,z), T(x,z), T(x,a), R(y,x,a)$$

where x, y, z, v, w are variables and a is a constant value. Compute the minimal CQ of Q.

Solution: We are going to apply the Minimization algorithm discussed in the lecture. Recall that this is a non-deterministic algorithm since the order in which the atoms of Q are considered is not predetermined. On the other hand, we can consider the atoms of Q in any order since the algorithm always computes the same minimal CQ (up to variable renaming). We are going to consider the atoms in Q from left to right.

- The first step is to check whether there is a query homomorphism from

  $$Q(x,y) \text{ :- } R(y,x,z), R(w,v,z), T(x,z), T(x,a), R(y,x,a)$$

  to

  $$Q(x,y) \text{ :- } R(w,v,z), T(x,z), T(x,a), R(y,x,a).$$

  This is indeed the case witnessed by the following query homomorphism

  $$h \;=\; \{x \mapsto x, y \mapsto y, z \mapsto a, w \mapsto y, v \mapsto x, a \mapsto a\}.$$

  It is easy to verify that

  $$h(\{R(y,x,z), R(w,v,z), T(x,z), T(x,a), R(y,x,a)\}) \subseteq \\ \{R(w,v,z), T(x,z), T(x,a), R(y,x,a)\}.$$

  Thus, we can remove the atom $R(y,x,z)$.

- The second step is to check whether there is a query homomorphism from

  $$Q(x,y) \text{ :- } R(w,v,z), T(x,z), T(x,a), R(y,x,a)$$

  to

  $$Q(x,y) \text{ :- } T(x,z), T(x,a), R(y,x,a).$$

  This is indeed the case witnessed by the following query homomorphism

  $$h \;=\; \{x \mapsto x, y \mapsto y, z \mapsto a, w \mapsto y, v \mapsto x, a \mapsto a\}.$$

  It is easy to verify that

  $$h(\{R(w,v,z), T(x,z), T(x,a), R(y,x,a)\}) \subseteq \{T(x,z), T(x,a), R(y,x,a)\}.$$

  Thus, we can remove the atom $R(w,v,z)$.

- The third step is to check whether there is a query homomorphism from

  $$Q(x,y) \text{ :- } T(x,z), T(x,a), R(y,x,a)$$

to

$$Q(x, y) \text{ :- } T(x, a), R(y, x, a).$$

This is indeed the case witnessed by the following query homomorphism

$$h \;=\; \{x \mapsto x, y \mapsto y, z \mapsto a, a \mapsto a\}.$$

It is easy to verify that

$$h(\{T(x, z), T(x, a), R(y, x, a)\}) \subseteq \{T(x, a), R(y, x, a)\}.$$

Thus, we can remove the atom $T(x, z)$.

- The fourth step is to check whether there is a query homomorphism from

  $$Q(x, y) \text{ :- } T(x, a), R(y, x, a)$$

  to

  $$Q(x, y) \text{ :- } R(y, x, a).$$

  It is clear that this is not the case since there is no way to map the atom $T(x, a)$ to $R(y, x, a)$. Therefore, $T(x, a)$ cannot be eliminated.

- The fifth and final step is to check whether there is a query homomorphism from

  $$Q(x, y) \text{ :- } T(x, a), R(y, x, a)$$

  to

  $$Q(x, y) \text{ :- } T(x, a).$$

  As in the previous step, it is clear that there is no way to map the atom $R(y, x, a)$ to $T(x, a)$, and thus, there is no query homomorphism. Hence, $R(y, x, a)$ cannot be eliminated.

Consequently, the minimal CQ of Q is

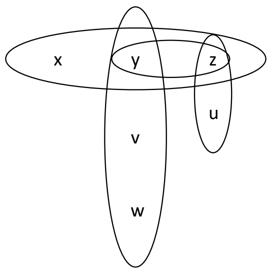
$$Q(x, y) \text{ :- } T(x, a), R(y, x, a).$$

4. (Acyclicity) Consider the CQ

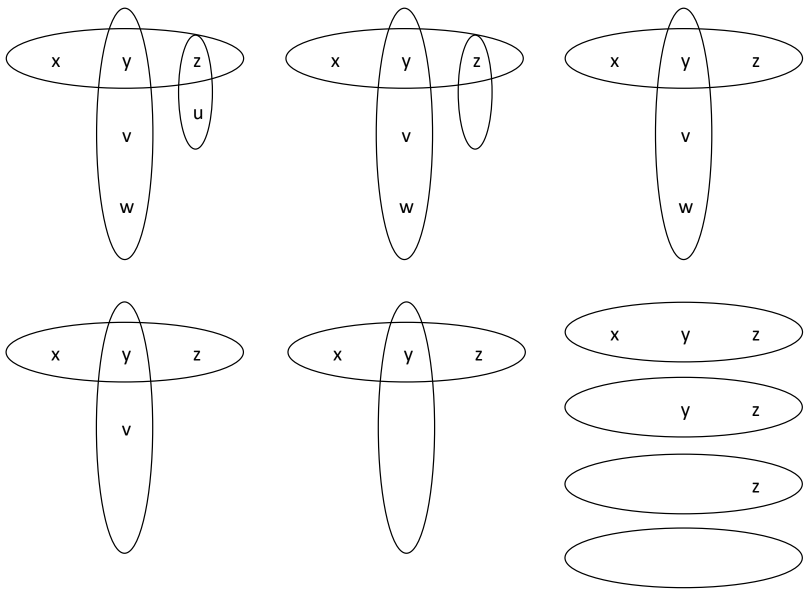
$$Q(x, y) \text{ :- } T(x, y, z), R(y, z), P(y, v, w), R(z, u).$$

Prove that Q is acyclic. In particular, apply the GYO-reduction to the hypergraph $H(Q)$ of Q and show that this leads to the empty graph. Give also the obtained join tree of $H(Q)$.

Solution: We first need to construct the hypergraph $H(Q)$ of Q, which follows:



We then apply the GYO-reduction on $H(Q)$. Here are the intermediate hypergraphs until we get the empty hypergraph:



The root of the obtained join tree is the set $\{x, y, z\}$, which has as children the sets $\{y, v, w\}$, $\{y, z\}$ and $\{z, u\}$.

SQL Quiz Time

POLL QUESTION

Student(sid,name,dept,age)

What is this query asking for?

```
SELECT S.name, MIN(S.age) AS dept_age
  FROM Student S
 GROUP BY S.dept
```

1. Get the student names and the minimum age in their department
2. Get the student names and the minimum age across all departments
3. Get the name and age of the student with the minimum age in their department
4. This is not a legal SQL query

POLL QUESTION

Student(sid, name, dept, age)

What is this query asking for?

```
SELECT S.name, MIN(S.age) AS dept_age
  FROM Student S
 GROUP BY S.dept
```

1. Get the student names and the minimum age in their department
2. Get the student names and the minimum age across all departments
3. Get the name and age of the student with the minimum age in their department
4. This is not a legal SQL query ✓

POLL QUESTION

Student(sid,name,dept,age)

Find the student with the highest age

A

```
SELECT MAX(S.age)
  FROM Student S
```

B

```
SELECT S.*, MAX(S.age)
  FROM Student S
```

C

```
SELECT S.*
  FROM Student S
 ORDER BY S.age
 LIMIT 1
```

POLL QUESTION

Student(sid, name, dept, age)

Find the student with the highest age

A

```
SELECT MAX(S.age)
  FROM Student S
```

✗

The highest age only

B

```
SELECT S.*, MAX(S.age)
  FROM Student S
```

✗

Invalid query

C

```
SELECT S.*
  FROM Student S
 ORDER BY S.age
 LIMIT 1
```

✗

The student with the lowest age

Poll Question

Student(sid, name, dept, age)
Enrolled(sid, cid, grade)

Find the CS students who have taken at least one course

A

```
SELECT DISTINCT S.*
  FROM Student S, Enrolled E
 WHERE S.dept = ’CS’
```

B

```
SELECT DISTINCT S.*
  FROM Student S
 WHERE S.dept = 'CS' AND ( SELECT COUNT(E.cid)
                             FROM Enrolled E ) >= 1
```

C

```
SELECT DISTINCT S.*
  FROM Student S NATURAL JOIN Enrolled E
 WHERE S.dept = ‘CS’
```

POLL QUESTION

`Student(sid, name, dept, age)`
`Enrolled(sid, cid, grade)`

Find the CS students who have taken at least one course

A

```
SELECT DISTINCT S.*
  FROM Student S, Enrolled E
 WHERE S.dept = ’CS’
```

✗ Missing join condition S.sid = E.sid

B

```
SELECT DISTINCT S.*
  FROM Student S
 WHERE S.dept = 'CS' AND ( SELECT COUNT(E.cid)
                             FROM Enrolled E ) >= 1
```

✗ Missing correlation S.sid = E.sid in the subquery

C

```
SELECT DISTINCT S.*
  FROM Student S NATURAL JOIN Enrolled E
 WHERE S.dept = ‘CS’
```

✓ DISTINCT S.* selects unique students without duplicates and is not Select *

Correct

Poll Question

Student(sid, name, dept, age)
Enrolled(sid, cid, grade)

Find the students who have not taken INF-11199

A

```
SELECT S.* FROM Student S
 WHERE S.sid NOT IN (SELECT E.* FROM Enrolled E
                      WHERE E.cid = ‘INF-11199’)
```

B

```
SELECT S.* FROM Student S
 WHERE NOT EXISTS (SELECT E.* FROM Enrolled E
                    WHERE E.cid = ‘INF-11199’)
```

C

```
SELECT S.* FROM Student S
 WHERE S.sid != ALL (SELECT E.sid FROM Enrolled E
                      WHERE E.cid = ‘INF-11199’)
```

POLL QUESTION

Student(sid, name, dept, age)
Enrolled(sid, cid, grade)

Find the students who have **not** taken INF-11199

A

```
SELECT S.* FROM Student S
 WHERE S.sid NOT IN (SELECT E.* FROM Enrolled E
                     WHERE E.cid = 'INF-11199')
```

✗ Invalid query. The inner query must return a bag of E.sid values

B

```
SELECT S.* FROM Student S
 WHERE NOT EXISTS (SELECT E.* FROM Enrolled E
                   WHERE E.cid = 'INF-11199')
```

✗ Missing correlation S.sid = E.sid in the subquery

C

```
SELECT S.* FROM Student S
 WHERE S.sid != ALL (SELECT E.sid FROM Enrolled E
                     WHERE E.cid = 'INF-11199')
```

✓ Correct

Poll Question

`Student(sid, name, dept, age)`

Find all the students with the highest age

A

```
SELECT S.*
  FROM Student S
 WHERE S.age > ALL
   (SELECT S2.age
       FROM Student S2
   )
```

B

```
SELECT S.*
  FROM Student S
 WHERE S.age > ANY
   (SELECT S2.age
       FROM Student S2
   )
```

C

```
SELECT S.*
  FROM Student S
 WHERE NOT EXISTS
   (SELECT S2.age
       FROM Student S2
      WHERE S2.age > S.age
   )
```

Poll Question

Student(sid, name, dept, age)

Find all the students with the highest age

A

```
SELECT S.*
  FROM Student S
 WHERE S.age > ALL
   (SELECT S2.age
      FROM Student S2
   )
```

✗

Always empty.
`S.age >= ALL (…)`
would be correct

B

```
SELECT S.*
  FROM Student S
 WHERE S.age > ANY
   (SELECT S2.age
      FROM Student S2
   )
```

✗

All except the lowest-age students

C

```
SELECT S.*
  FROM Student S
 WHERE NOT EXISTS
   (SELECT S2.age
      FROM Student S2
     WHERE S2.age > S.age
   )
```

✓

No other student has a higher age

Poll Question

Student(sid, name, dept, age)

A

```
SELECT S.*
  FROM Student S
 WHERE S.age =
   (SELECT MAX(S2.age)
      FROM Student S2)
```

B

```
SELECT S.*
  FROM Student S
 ORDER BY S.age DESC
 LIMIT 1
```

C

```
SELECT S.*
  FROM Student S
 WHERE S.age >= ALL
   (SELECT S2.age
      FROM Student S2)
```

Which queries always return the same students (in any order)?

1) None
2) A and B
3) B and C
4) A and C
5) All

Poll Question

`Student(sid, name, dept, age)`

A

```
SELECT S.*
  FROM Student S
 WHERE S.age =
   (SELECT MAX(S2.age)
       FROM Student S2)
```

B

```
SELECT S.*
  FROM Student S
 ORDER BY S.age DESC
 LIMIT 1
```

C

```
SELECT S.*
  FROM Student S
 WHERE S.age >= ALL
   (SELECT S2.age
       FROM Student S2)
```

Which queries always return the same students (in any order)?

1) None
2) A and B
3) B and C
4) A and C ✓
5) All

A and **C** return *all* students with the highest age.

B returns *one* student with the highest age.

Pool Question

Student(sid, name, dept, age)
Enrolled(sid, cid, grade)
Course(cid, name, year)

```
SELECT S.name FROM Student S
WHERE NOT EXISTS ( SELECT C.cid FROM Course C
                   WHERE NOT EXISTS ( SELECT 42 FROM Enrolled E
                                      WHERE E.cid = C.cid
                                        AND E.sid = S.sid ) )
```

What is this query asking for?

1) Names of students that have taken at least one course
2) Names of students that have taken no course
3) Names of students that have taken all courses
4) Names of students that have not taken all courses
5) Nobody really knows

POOL QUESTION

Student(sid, name, dept, age)

Enrolled(sid, cid, grade)

Course(cid, name, year)

```
SELECT S.name FROM Student S
WHERE NOT EXISTS ( SELECT C.cid FROM Course C
                   WHERE NOT EXISTS ( SELECT 42 FROM Enrolled E
                                      WHERE E.cid = C.cid
                                        AND E.sid = S.sid ) )
```

What is this query asking for?

1) Names of students that have taken at least one course
2) Names of students that have taken no course
3) Names of students that have taken all courses ✓
4) Names of students that have not taken all courses
5) Nobody really knows

This query effectively filters and lists names of students who are enrolled in every course offered. It uses a double negation approach:

It first finds any courses that a student is not enrolled in.
Then, it filters out any students who have such a course.

Congratulations!

University of Edinburgh
School of Informatics
INFR11199 - Advanced Database Systems (Spring 2024)

Tutorial Sheet 2

1. (Files, Pages, Records) Consider the following relation:

```
CREATE TABLE Customer (
  customer_id INTEGER PRIMARY KEY,    -- cannot be NULL!
  age INTEGER NOT NULL,
  name VARCHAR(10) NOT NULL,
  address VARCHAR(20) NOT NULL
);
```

Assume that `INTEGER`s are 4 bytes long and `VARCHAR(n)` can be up to n bytes long.

(a) As the records are variable length, we will need a *record header* in the record. How big is the record header? You may assume pointers are 4 bytes long, and that the record header only contains pointers for variable-length values.

(b) Including the record header, what is the smallest possible record size (in bytes) in this schema? Note: `NULL` is treated as a special value by SQL, and an empty string `VARCHAR` is different from `NULL`, just like how a 0 `INTEGER` value is also different from `NULL`.

(c) Including the record header, what is the largest possible record size (in bytes) in this schema?

(d) Suppose we are storing these records using a slotted page layout with variable length records. The page header contains an integer storing the record count and a pointer to free space, as well as a slot directory storing, for each record, a pointer and length. What is the *maximum* number of records that we can fit on a 8KB page?

(e) Suppose we stored the maximum number of records on a page, and then deleted one record. Now we want to insert another record. Are we guaranteed to be able to do this? Explain why or why not.

(f) Now suppose we deleted 3 records. Without reorganizing any of the records on the page, we would like to insert another record. Are we guaranteed to be able to do this? Explain why or why not.

2. (Buffer Management) We are given a buffer pool with 4 pages, which is empty to begin with. Answer the following questions given this access pattern:

A B C D E B A D C A E C

(a) What is the hit rate for MRU?

(b) For MRU, which pages are in the buffer pool after this sequence of accesses?

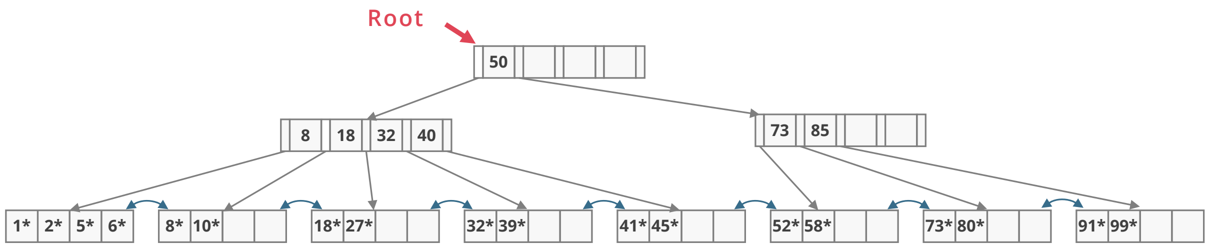
(c) What is the hit rate for clock? Assume the clock hand initially points at the first frame.

(d) For clock, which pages are in the buffer pool after this sequence of accesses?

(e) Which pages have their reference bits set?

(f) Which page is the hand of the clock pointing to?

3. (B+ Tree) Consider the following B+ tree index of order $d = 2$:



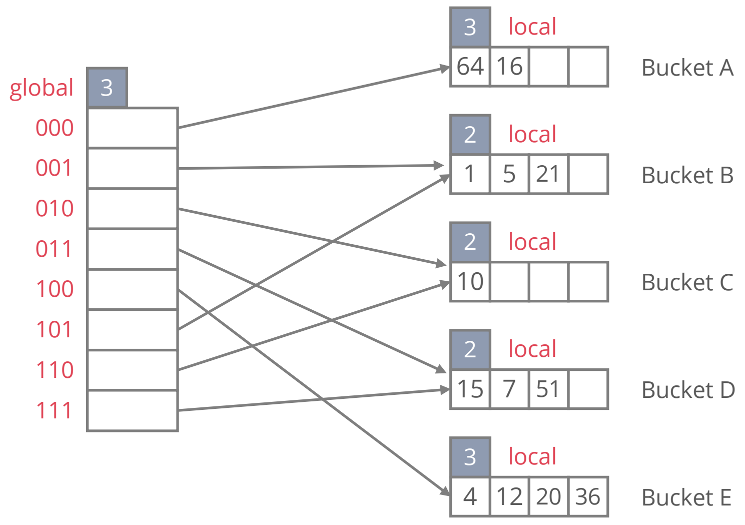
(a) Show the B+ tree that would result from inserting an index entry with key 9 into this tree.

(b) Show the B+ tree that would result from inserting an index entry with key 3 *into the original tree*. Assume no redistribution between siblings. How many page reads and page writes does the insertion require? Justify your answer.

(c) Show the B+ tree that would result from deleting the index entry with key 8 *from the original tree*, assuming that the left sibling is checked for possible redistribution.

(d) Show the B+ tree that would result from deleting the index entry with key 8 *from the original tree*, assuming that the right sibling is checked for possible redistribution.

4. (Extendible Hashing) Consider the following Extendible Hashing index:



(a) Draw the index after inserting an entry with hash value 68.

(b) Draw the index after inserting entries with hash values 17 and 69 into the original index.

University of Edinburgh
School of Informatics
INFR11199 - Advanced Database Systems (Spring 2024)

Tutorial Sheet 2

1. (Files, Pages, Records) Consider the following relation:

```
CREATE TABLE Customer (
  customer_id INTEGER PRIMARY KEY,    -- cannot be NULL!
  age INTEGER NOT NULL,
  name VARCHAR(10) NOT NULL,
  address VARCHAR(20) NOT NULL
);
```

Assume that `INTEGER`s are 4 bytes long and `VARCHAR(n)` can be up to n bytes long.

(a) As the records are variable length, we will need a *record header* in the record. How big is the record header? You may assume pointers are 4 bytes long, and that the record header only contains pointers for variable-length values.

> **Solution:** 8 bytes. In the record header, we need one pointer for each variable length value. In this schema, those are just the two VARCHARs, so we need 2 pointers, each 4 bytes.

(b) Including the record header, what is the smallest possible record size (in bytes) in this schema? Note: `NULL` is treated as a special value by SQL, and an empty string `VARCHAR` is different from `NULL`, just like how a 0 `INTEGER` value is also different from `NULL`.

> **Solution:** 16 bytes $(= 8 + 4 + 4 + 0 + 0)$
> 8 for the record header, 4 for each of the integers, and 0 for each of the `VARCHAR`s.

(c) Including the record header, what is the largest possible record size (in bytes) in this schema?

Solution: 46 bytes (= 8 + 4 + 4 + 10 + 20)

(d) Suppose we are storing these records using a slotted page layout with variable length records. The page header contains an integer storing the record count and a pointer to free space, as well as a slot directory storing, for each record, a pointer and length. What is the *maximum* number of records that we can fit on a 8KB page?

Solution: 341 records (= (8192 - 4 - 4) / (16 + 4 + 4))
We start out with 8192 bytes of space on the page. We subtract 4 bytes that are used for the record count, and another 4 for the pointer to free space. This leaves us with $8192 - 4 - 4$ bytes that we can use to store records and their slots. A record takes up 16 bytes of space at minimum (from the previous questions), and for each record we also need to store a slot with a pointer (4 bytes) and a length (4 bytes). Thus, we need $16 + 4 + 4$ bytes of space for each record and its slot.

(e) Suppose we stored the maximum number of records on a page, and then deleted one record. Now we want to insert another record. Are we guaranteed to be able to do this? Explain why or why not.

Solution: No, we deleted 16 bytes but the record we want to insert may be up to 46 bytes.

(f) Now suppose we deleted 3 records. Without reorganizing any of the records on the page, we would like to insert another record. Are we guaranteed to be able to do this? Explain why or why not.

Solution: No, there are 48 free bytes but they may be fragmented – there might not be 46 contiguous bytes.

2. (Buffer Management) We are given a buffer pool with 4 pages, which is empty to begin with. Answer the following questions given this access pattern:

A B C D E B A D C A E C

(a) What is the hit rate for MRU?

Solution: 4/12.
A (miss), B (miss), C (miss), D (miss), E (miss, D out), B (hit), A (hit), D (miss, A out), C (hit), A (miss, C out), E (hit), C (miss, E out)

(b) For MRU, which pages are in the buffer pool after this sequence of accesses?

Solution: D, B, A, C.

(c) What is the hit rate for clock? Assume the clock hand initially points at the first frame.

Solution: 5/12.
after ABCD: A(1), B(1), C(1), D(1), hand=A
after E: E(1), B(0), C(0), D(0), hand=B
after B: E(1), B(1), C(0), D(0), hand=B, HIT
after A: E(1), B(0), A(1), D(0), hand=D
after D: E(1), B(0), A(1), D(1), hand=D, HIT
after C: E(0), C(1), A(1), D(0), hand=A
after A: E(0), C(1), A(1), D(0), hand=A, HIT
after E: E(1), C(1), A(1), D(0), hand=A, HIT
after C: E(1), C(1), A(1), D(0), hand=A, HIT

(d) For clock, which pages are in the buffer pool after this sequence of accesses?

Solution: A, C, D, E.

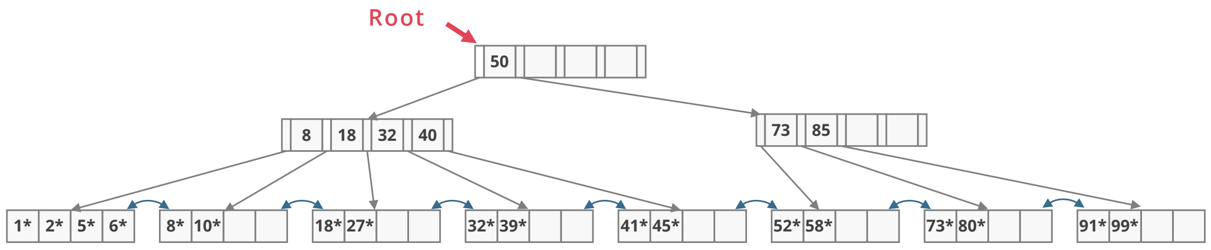
(e) Which pages have their reference bits set?

Solution: A, C, E.

(f) Which page is the hand of the clock pointing to?

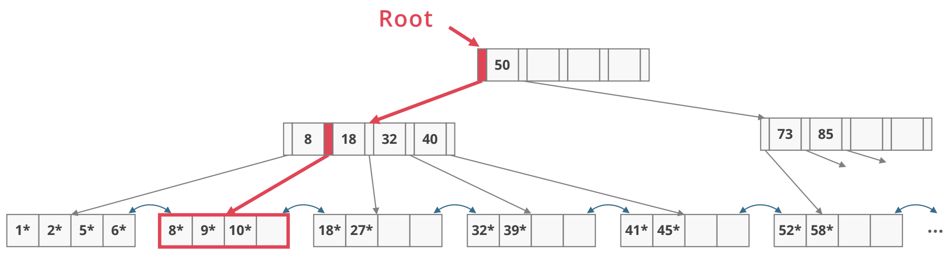
Solution: A. On a page hit, the clock hand does not move.

3. (B+ Tree) Consider the following B+ tree index of order $d = 2$:



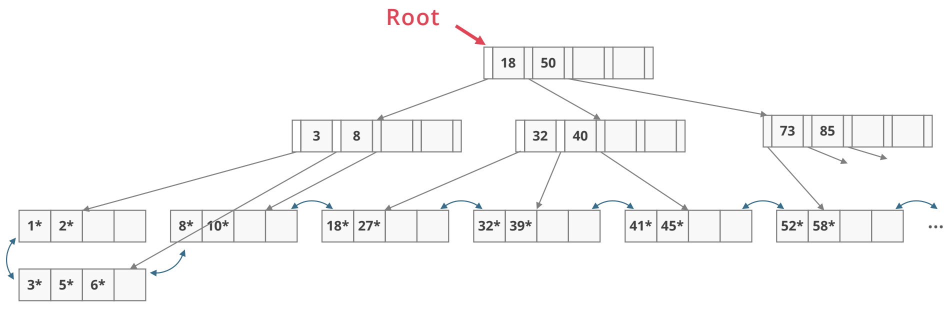
(a) Show the B+ tree that would result from inserting an index entry with key 9 into this tree.

Solution: The index entry with key 9 is inserted on the second leaf page. The resulting tree is shown below:



(b) Show the B+ tree that would result from inserting an index entry with key 3 *into the original tree*. Assume no redistribution between siblings. How many page reads and page writes does the insertion require? Justify your answer.

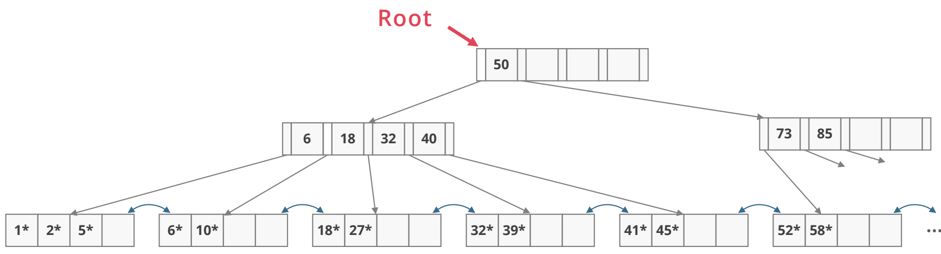
Solution: The index entry with key 3 goes on the first leaf page. As this page can accommodate at most four index entries ($d = 2$), the page is split. The lowest key of the new leaf is copied to the ancestor, which is also full and needs to be split. The middle key of the ancestor (after adding key 3, that would be key 18) is pushed to the root. The result is shown below:



The insertion requires 4 page reads, 6 page writes, and allocation of 2 new pages: read the root, read its left child, read the leftmost leaf, create a new leaf, write the leftmost leaf, write the new leaf, read and write the leaf containing keys 8 and 10 (to update its left pointer), create a new inner page, write the root's left child, write the new inner page, write the root.

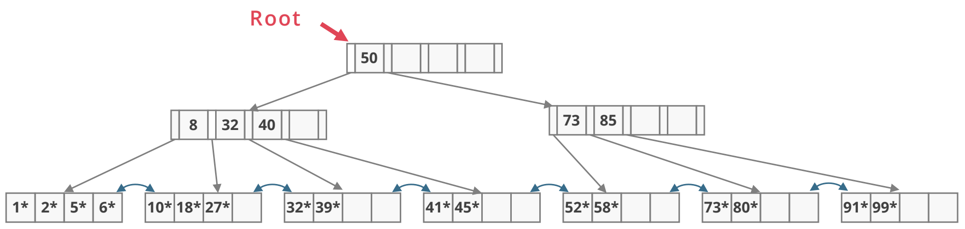
(c) Show the B+ tree that would result from deleting the index entry with key 8 *from the original tree*, assuming that the left sibling is checked for possible redistribution.

Solution: The index entry with key 8 is deleted, resulting in a leaf page with fewer than two index entries. The left sibling is checked for redistribution. Since the sibling has more than two index entries, the remaining keys are redistributed between the two leaves. The resulting tree is shown below:

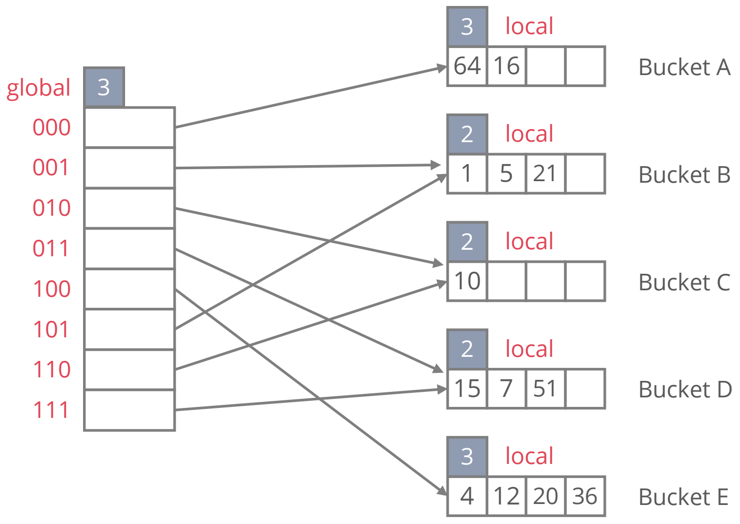


(d) Show the B+ tree that would result from deleting the index entry with key 8 *from the original tree*, assuming that the right sibling is checked for possible redistribution.

Solution: The index entry with key 8 is deleted from the corresponding leaf page. The right sibling has the minimum number of keys, therefore the two siblings merge. The key in the ancestor which distinguishes between the newly merged leaves is deleted. The resulting tree is shown below:

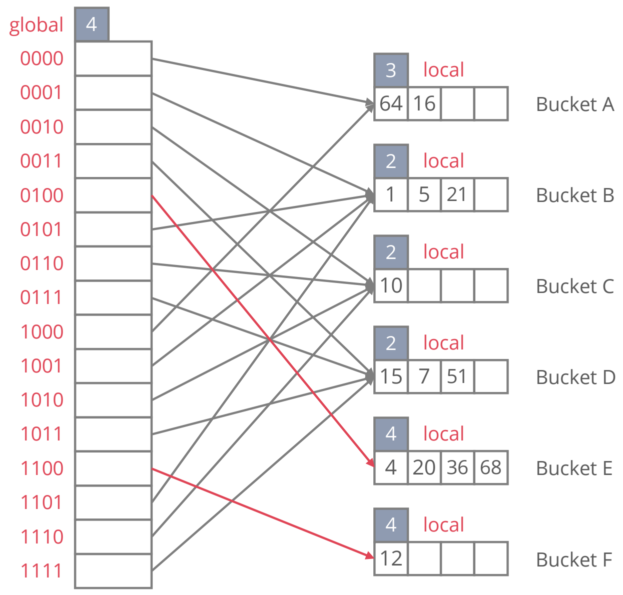


4. (Extendible Hashing) Consider the following Extendible Hashing index:



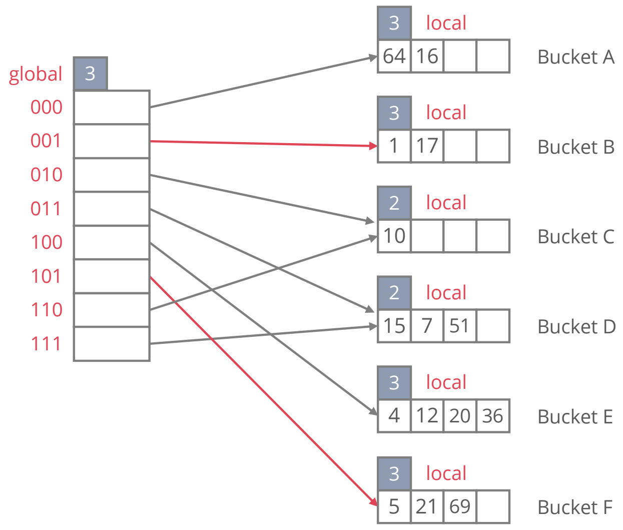
(a) Draw the index after inserting an entry with hash value 68.

Solution: After inserting an entry with hash value 68:



(b) Draw the index after inserting entries with hash values 17 and 69 into the original index.

Solution: After inserting entries with hash values 17 and 69:



Advanced Database Systems

Spring 2024

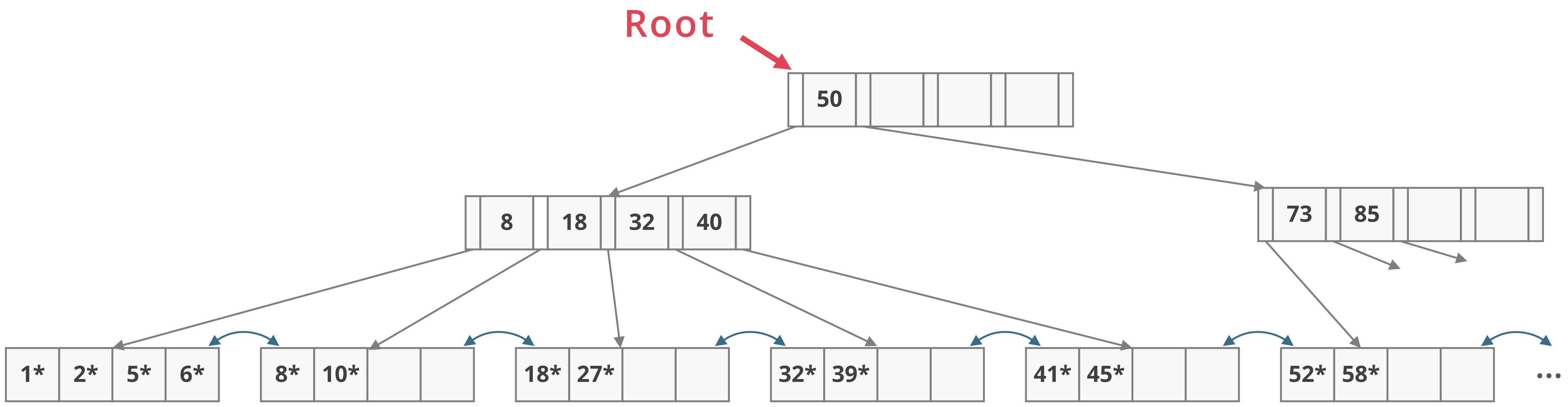
Tutorial 2

Question 3

B+ Tree: Search for 39

Find key = 39

- Find split on each node
- Follow pointer to next node

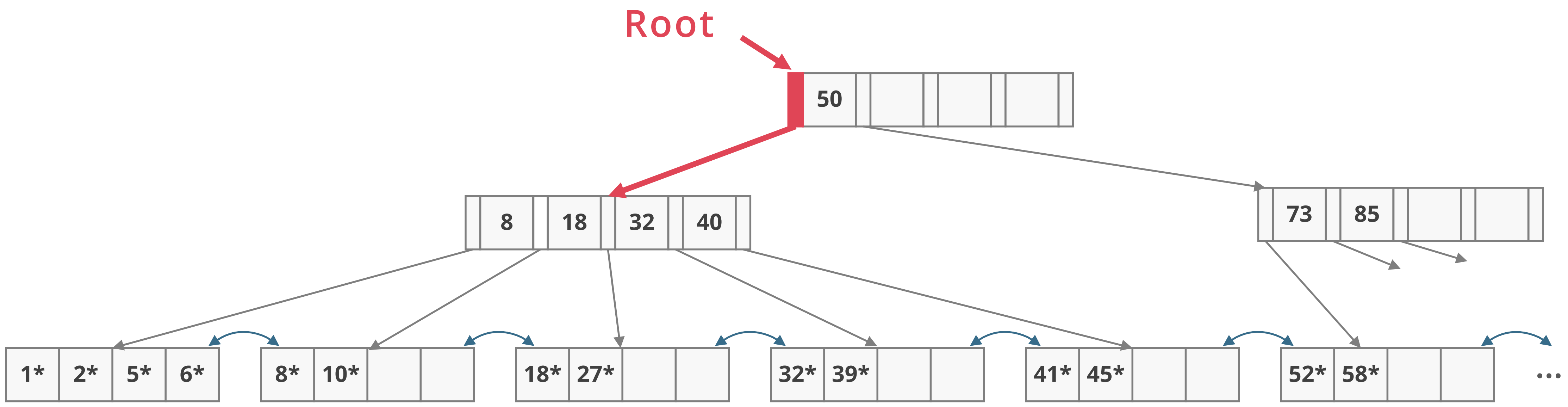


B+ Tree: Search for 39

Find key = 39

- Find split on each node
- Follow pointer to next node

Use binary search on each page

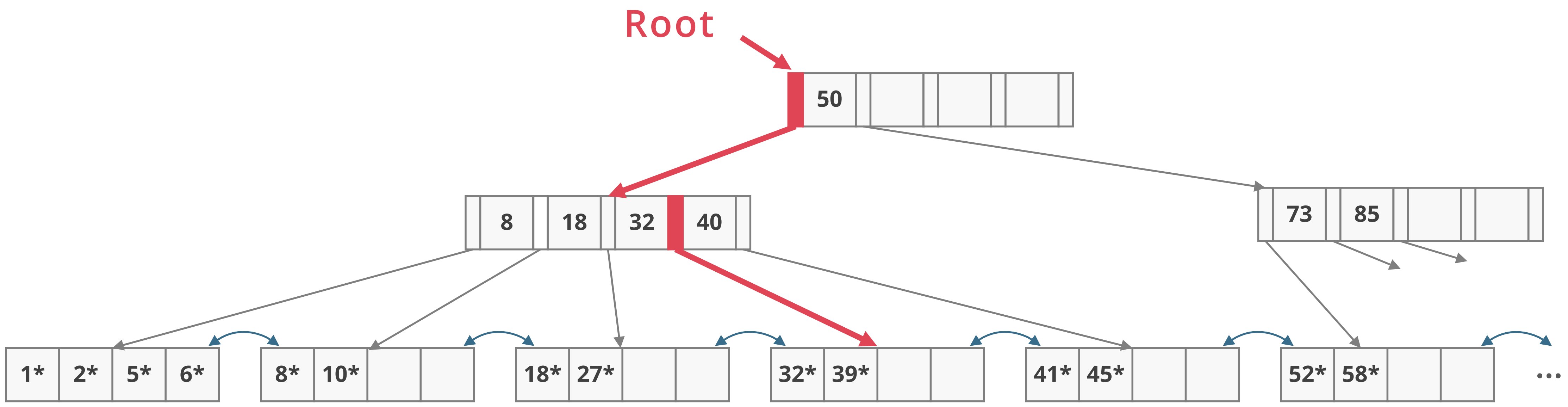


B+ Tree: Search for 39

Find key = 39

- Find split on each node
- Follow pointer to next node

Use binary search on each page

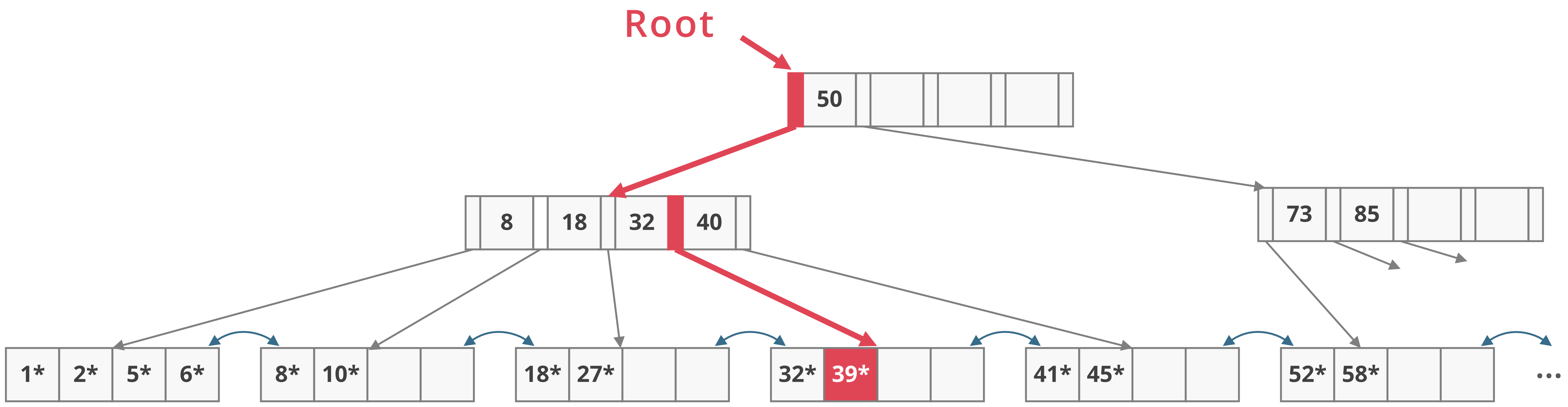


B+ Tree: Search for 39

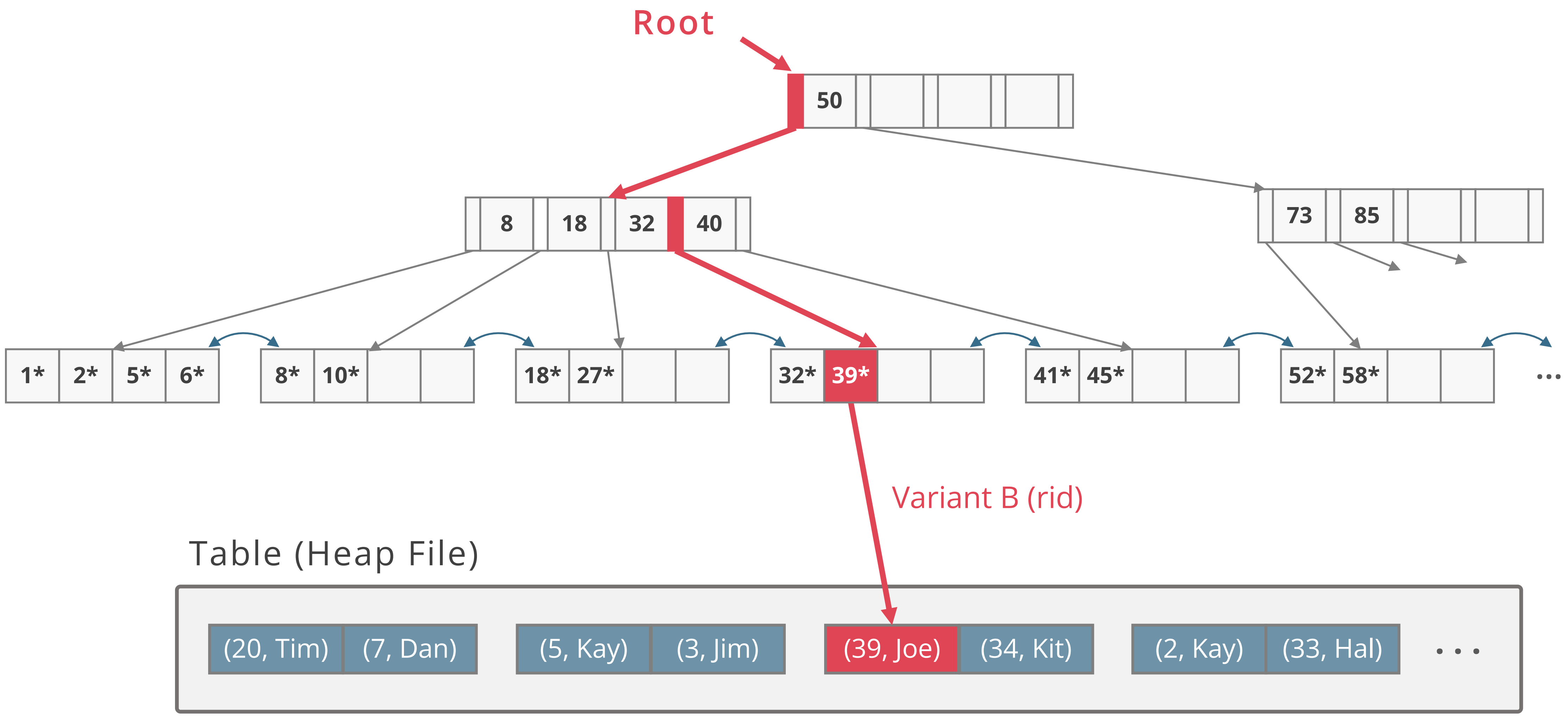
Find key = 39

- Find split on each node
- Follow pointer to next node

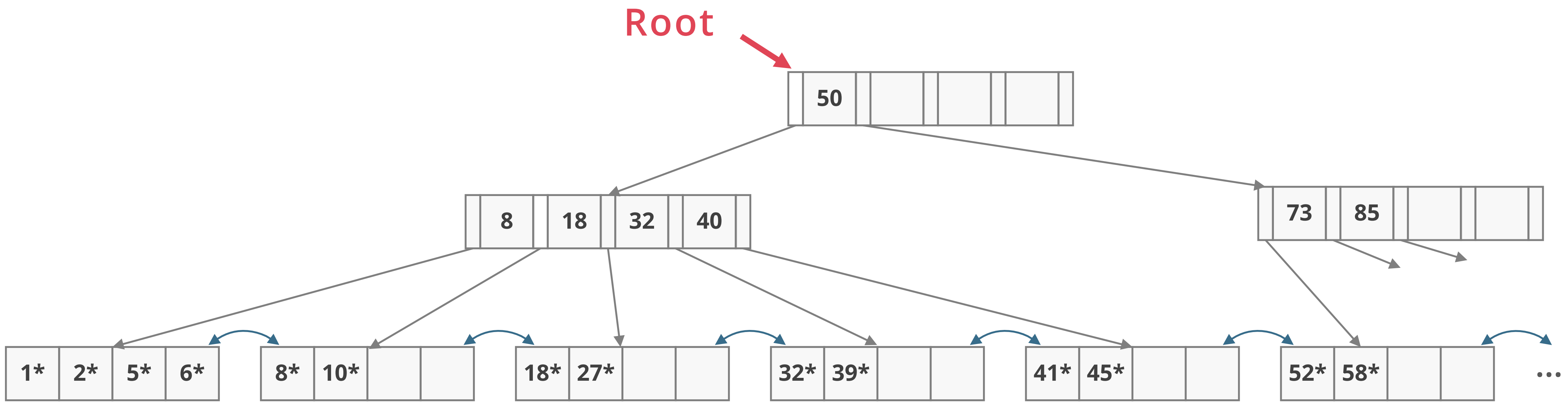
Use binary search on each page



B+ Tree: Search for 39



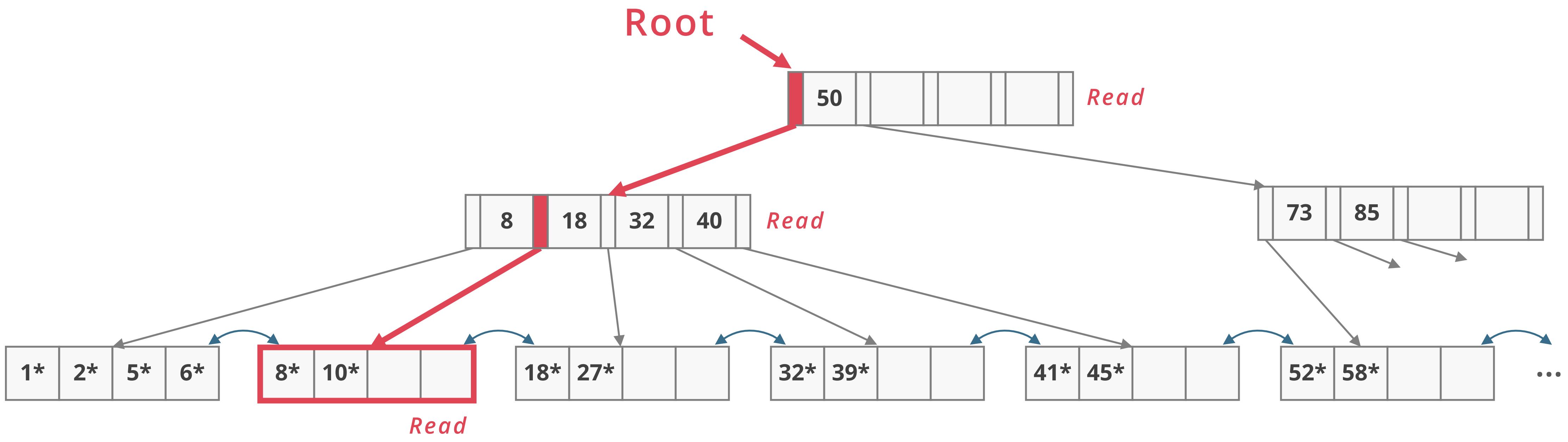
B+ Tree: Insert Entry 9*



I/O Total (so far): 0

B+ Tree: Insert Entry 9*

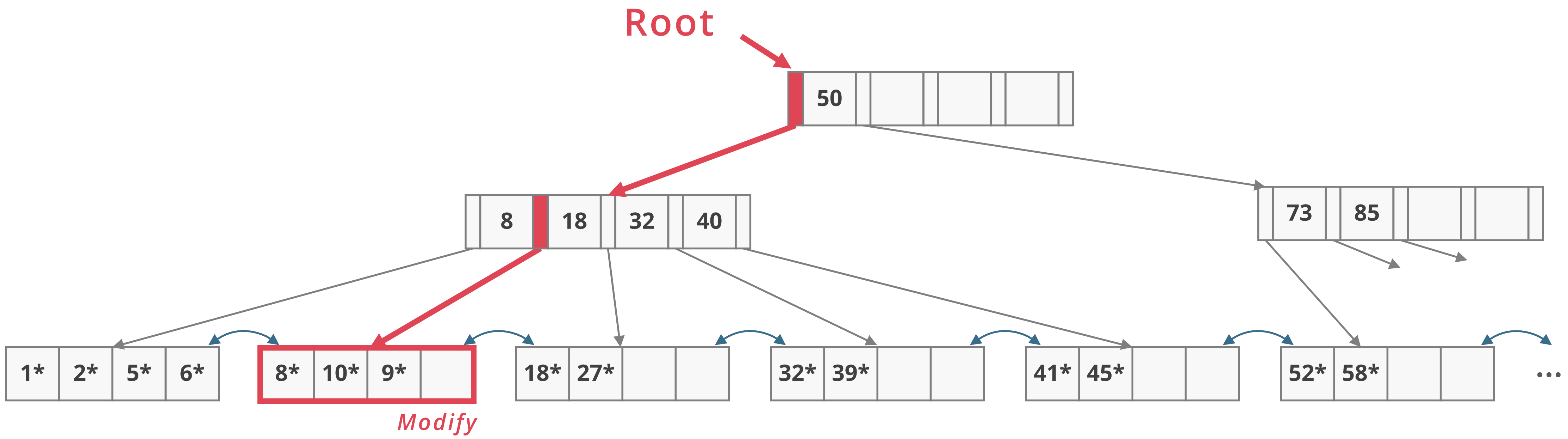
Find the correct leaf node



I/O Total (so far): 3

B+ Tree: Insert Entry 9*

If there is room in leaf, just add the entry

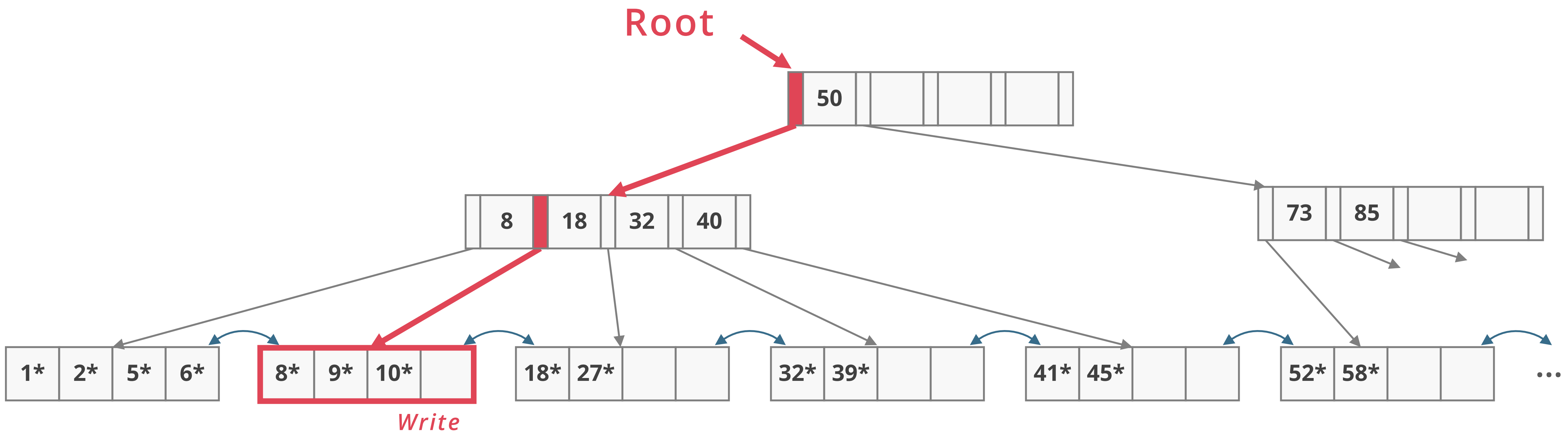


I/O Total (so far): 3

B+ Tree: Insert Entry 9*

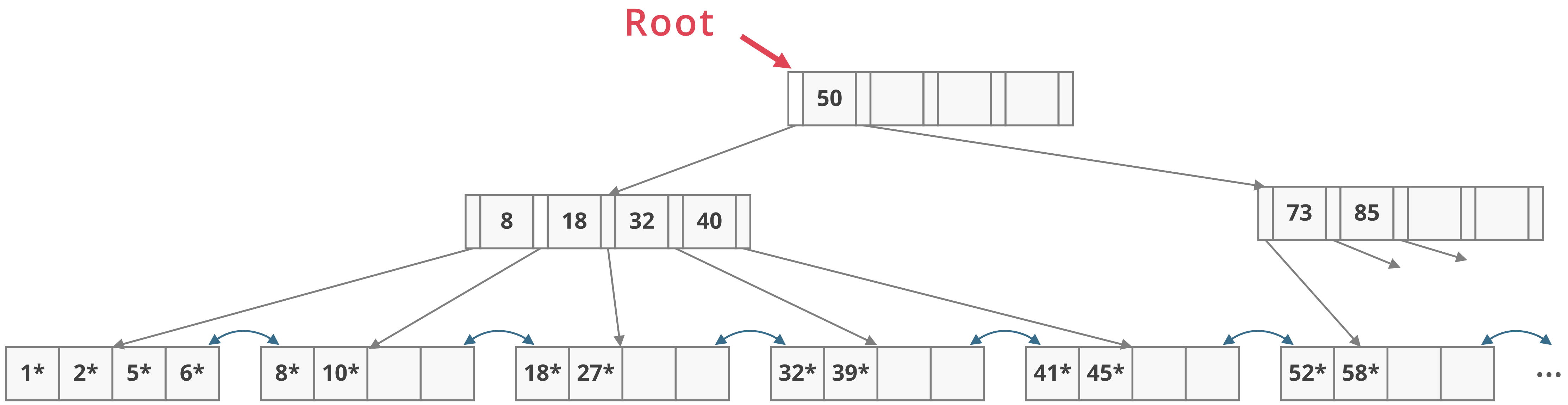
If there is room in leaf, just add the entry

… and keep the leaf sorted



I/O Total (so far): 4

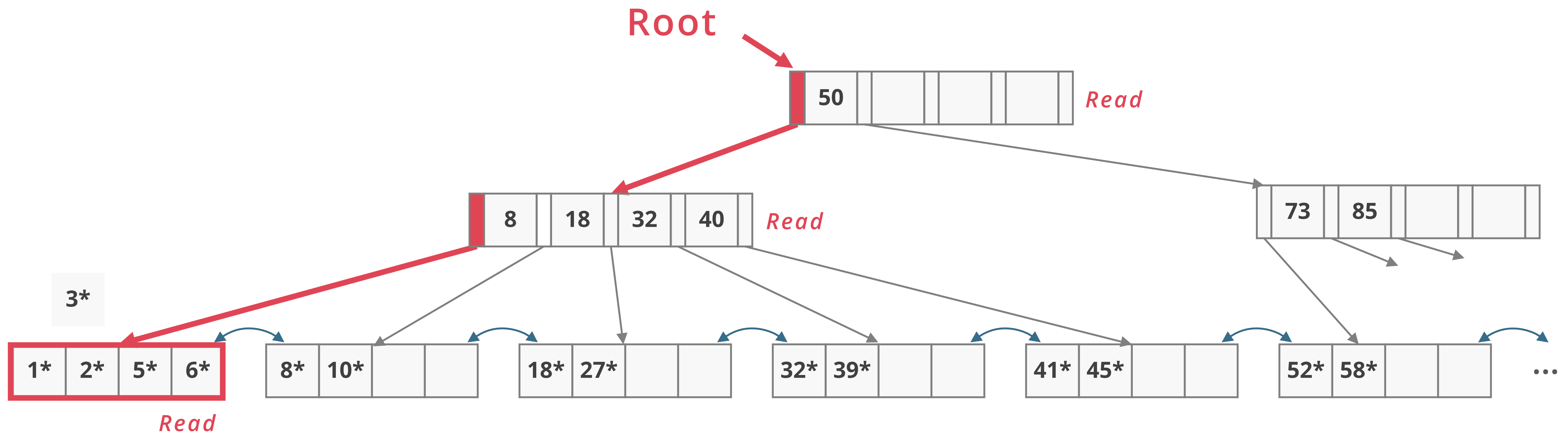
B+ Tree: Insert Entry 3*



I/O Total (so far): 0

B+ Tree: Insert Entry 3*

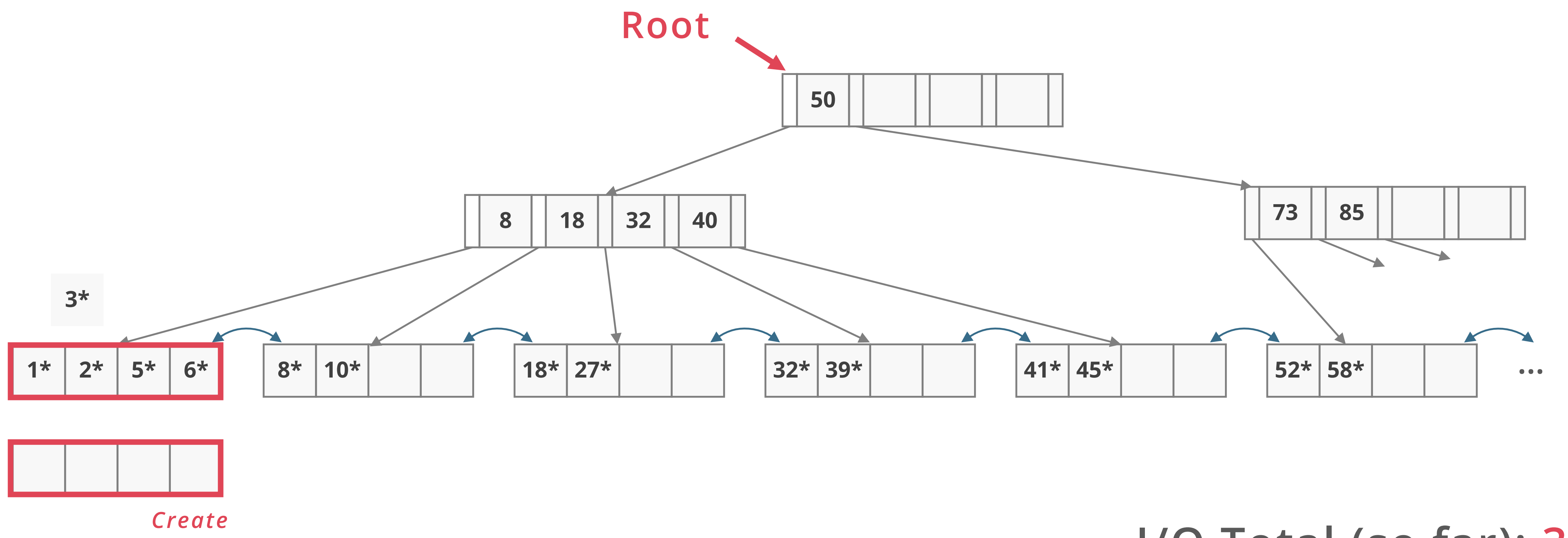
Find the correct leaf node



I/O Total (so far): 3

B+ Tree: Insert Entry 3*

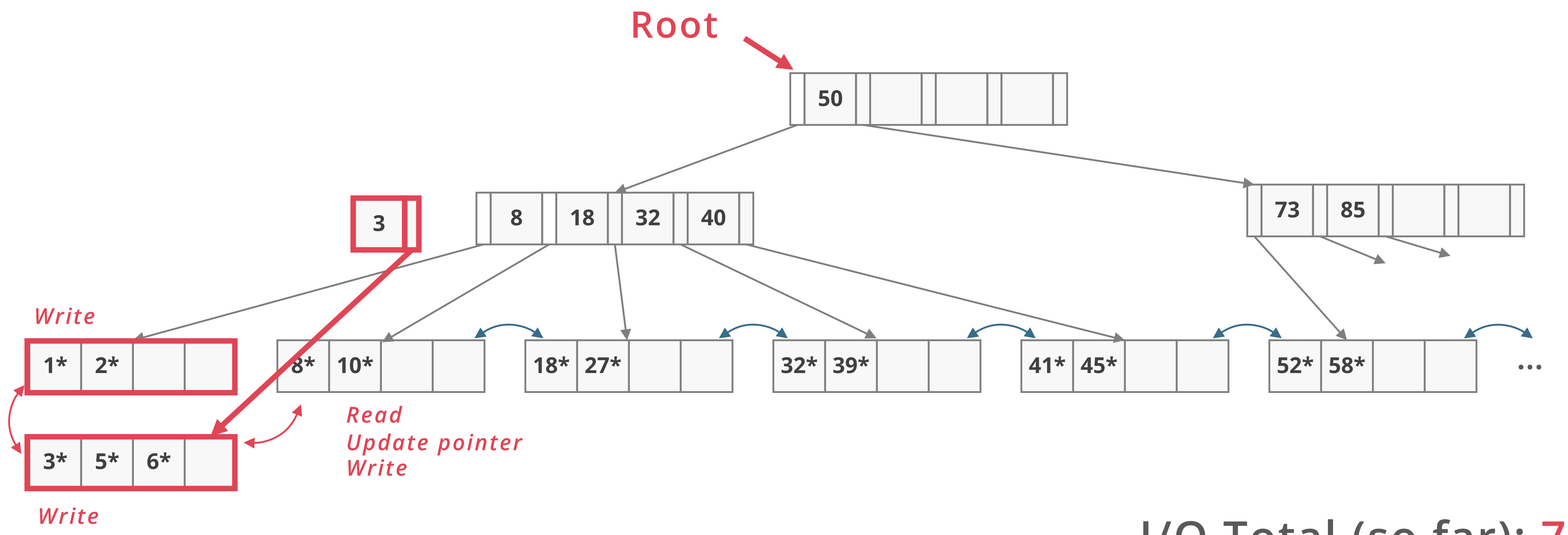
Split leaf if not enough room: into two leaves with d and d + 1 entries



I/O Total (so far): 3

B+ Tree: Insert Entry 3*

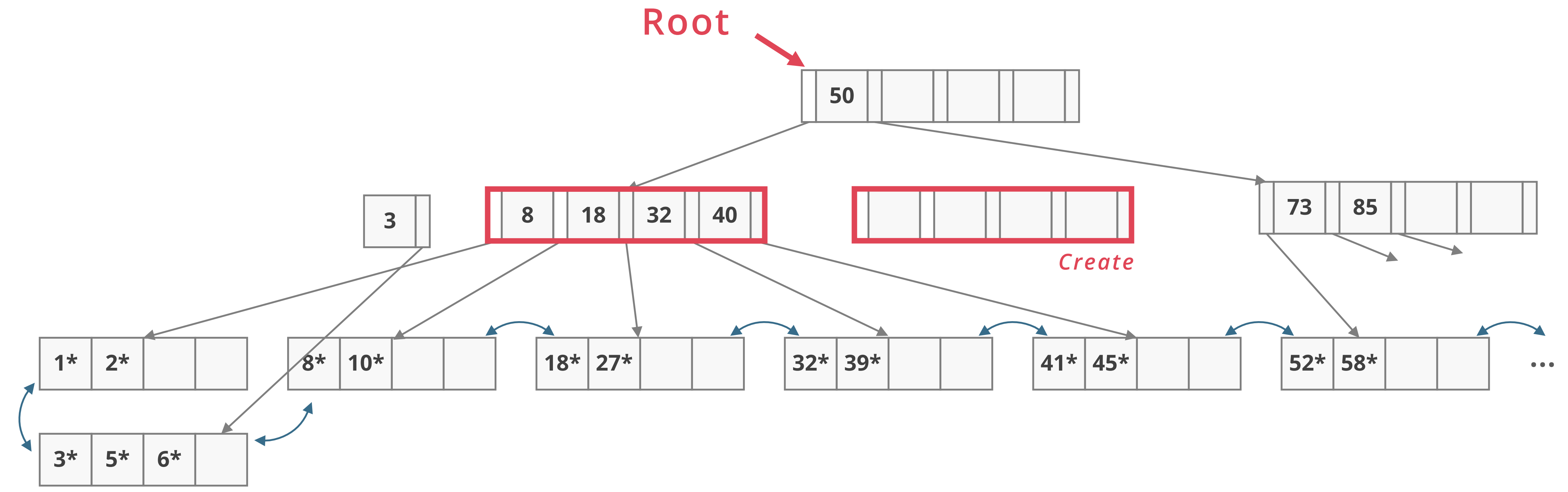
Copy up the middle key to inner node (since leaf nodes have data)



I/O Total (so far): 7

B+ Tree: Insert Entry 3*

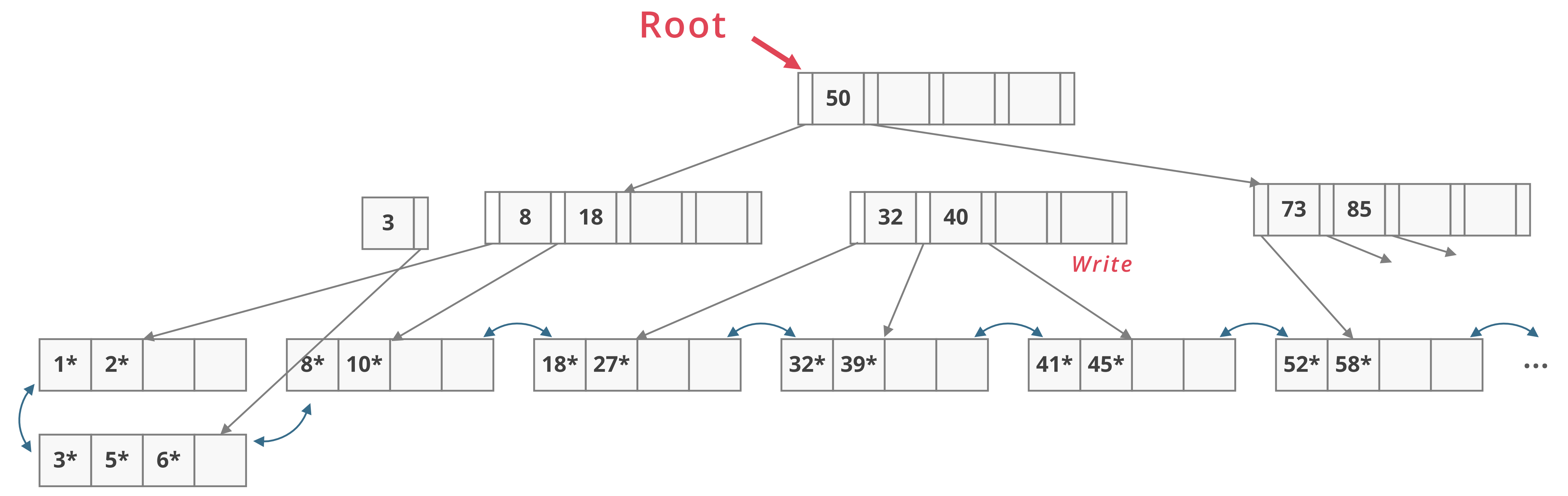
If inner node is full, split the inner node into two and **push** the middle key up



I/O Total (so far): 7

B+ Tree: Insert Entry 3*

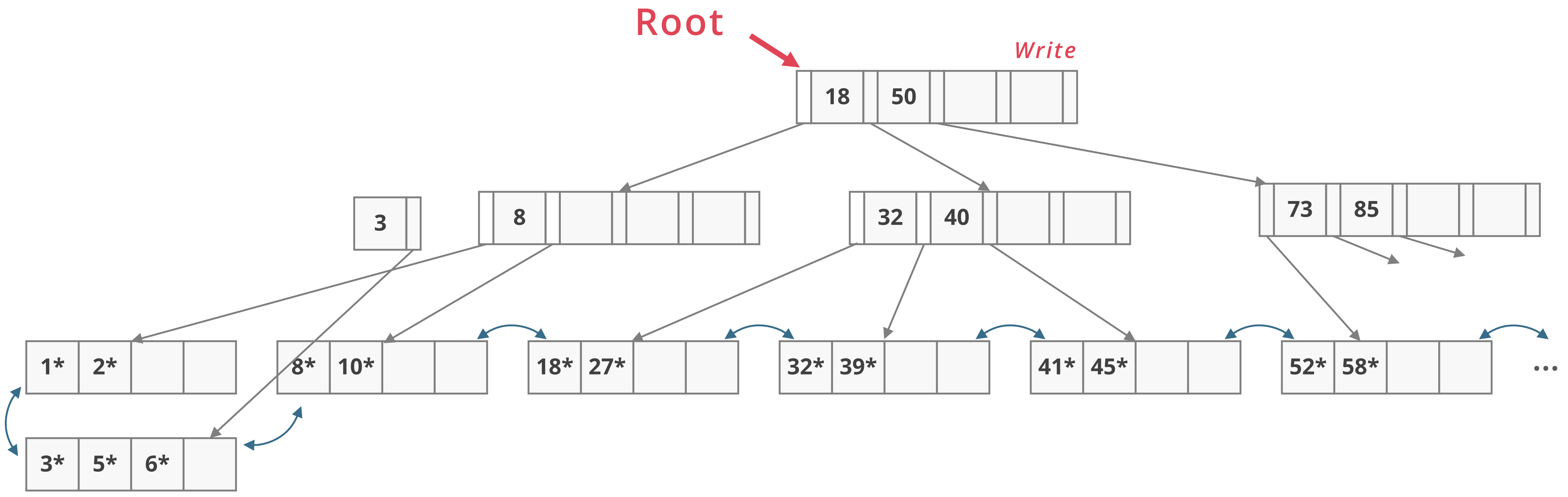
If inner node is full, split the inner node into two and **push** the middle key up



I/O Total (so far): 8

B+ Tree: Insert Entry 3*

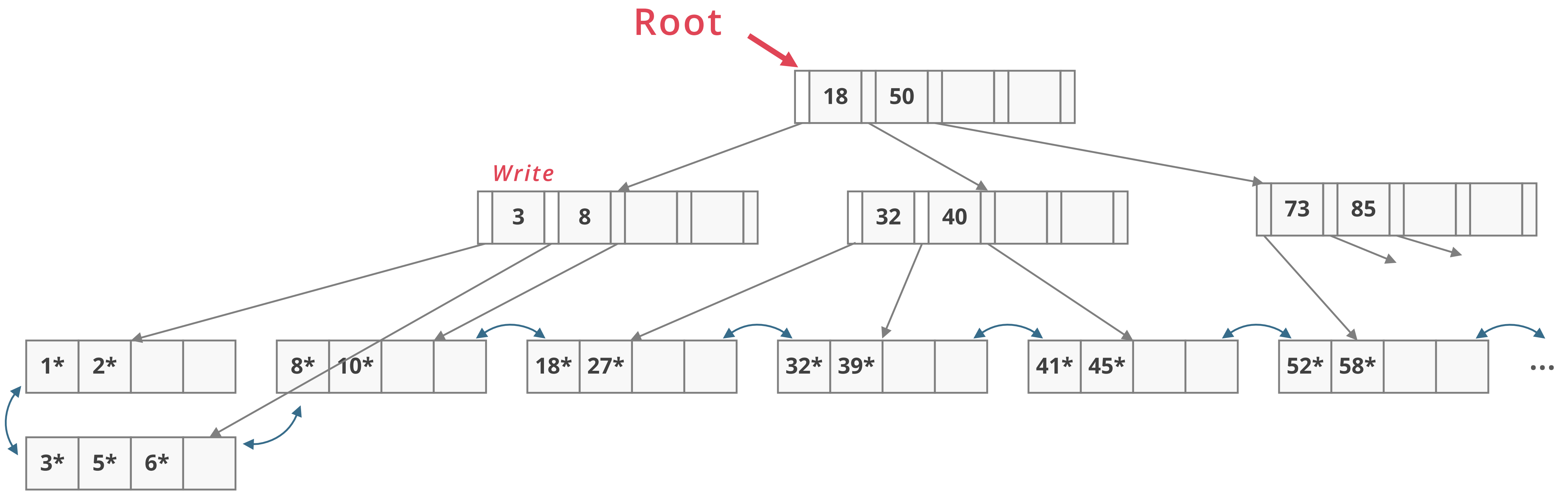
If inner node is full, split the inner node into two and **push** the middle key up



I/O Total (so far): 9

B+ Tree: Insert Entry 3*

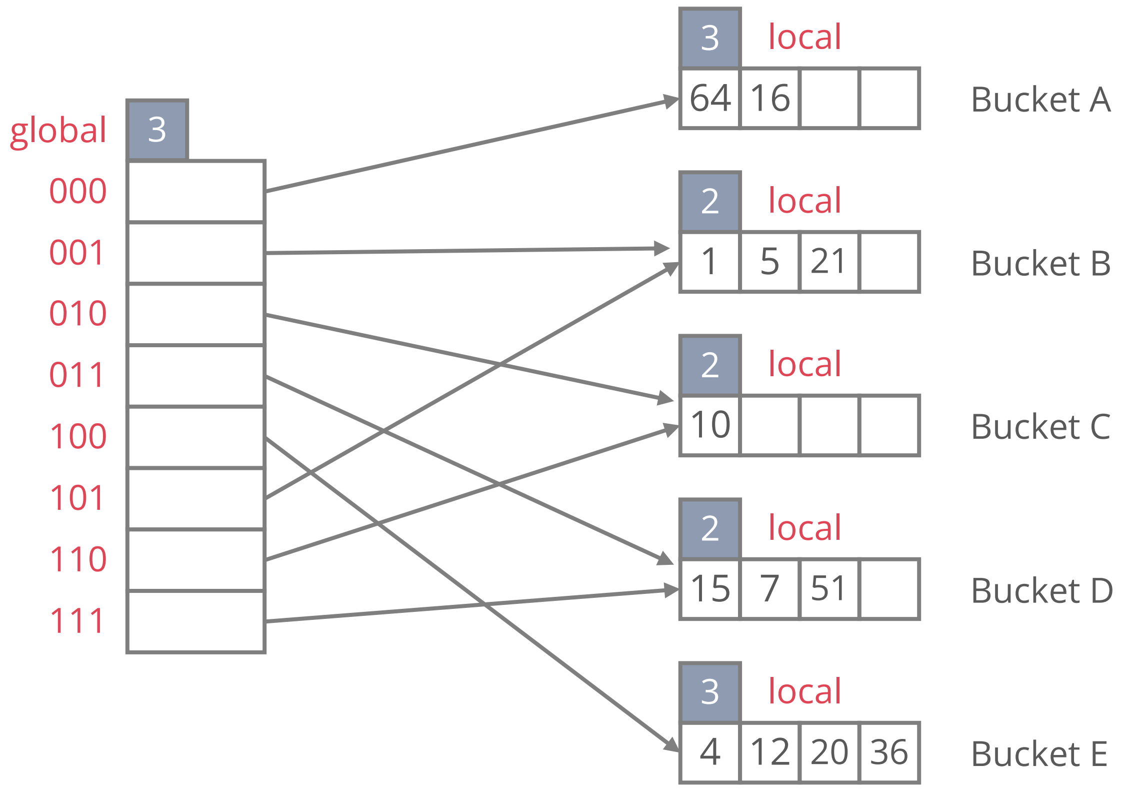
If inner node is full, split the inner node into two and **push** the middle key up



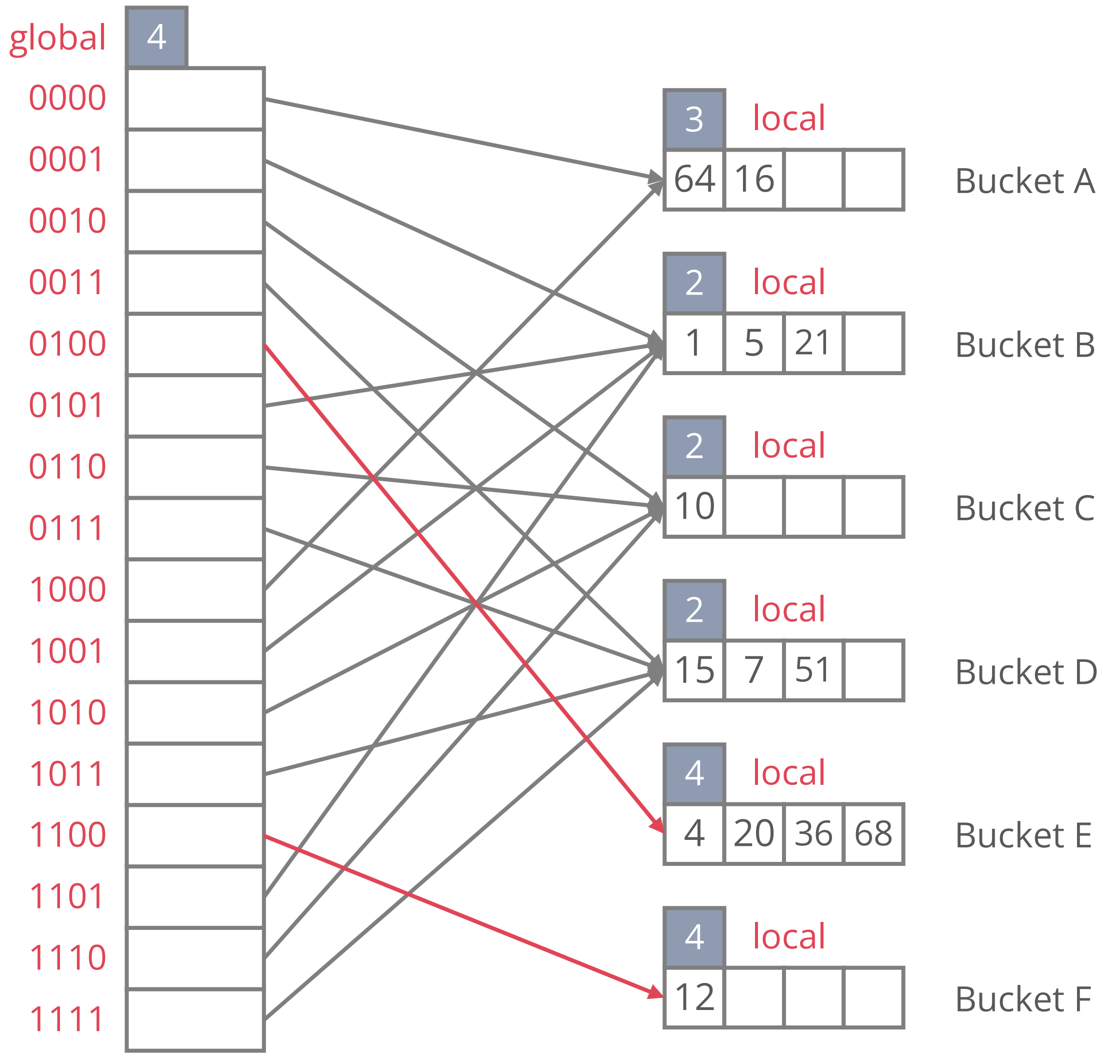
I/O Total (so far): 10

Question 4

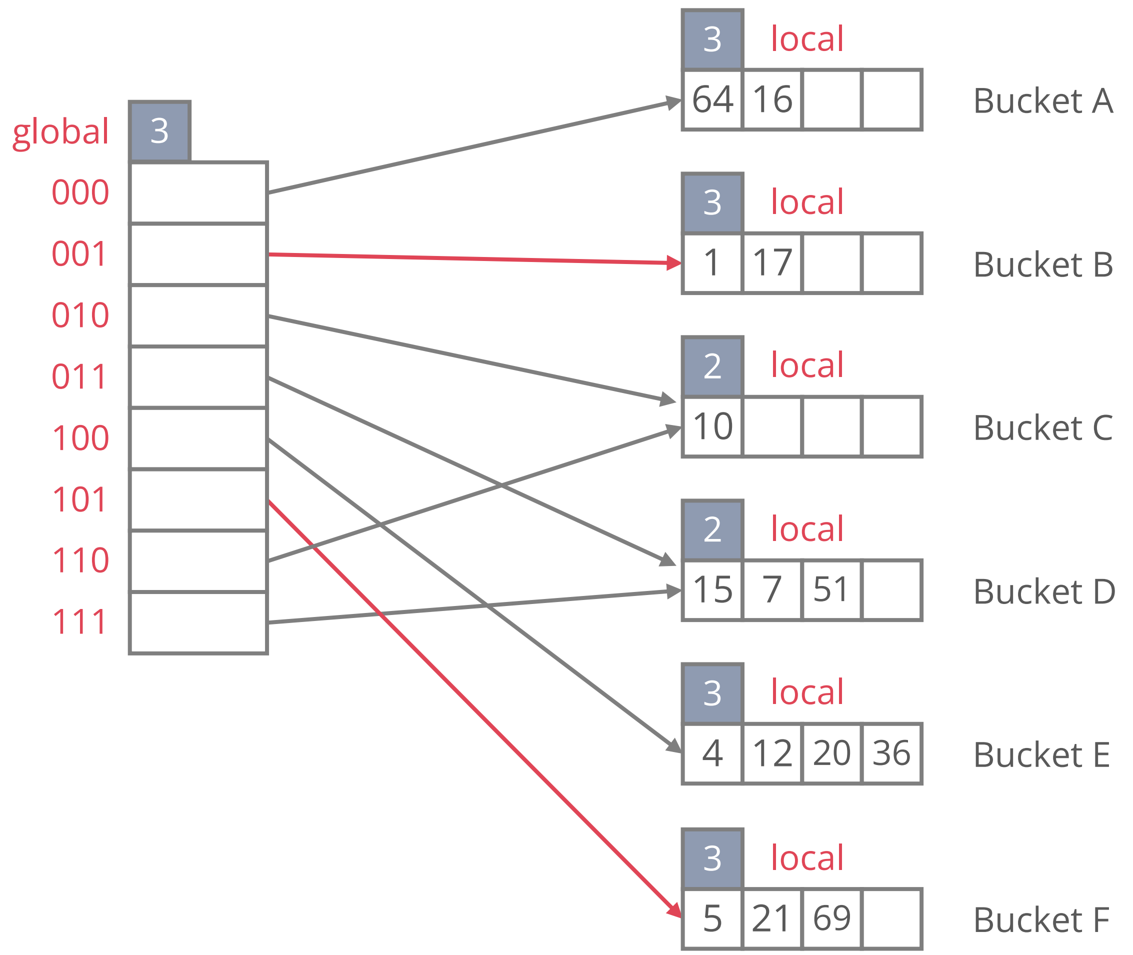
Extendible Hashing



Insert Entry with Hash Value 68



Insert Entries with Hash Values 17 & 69



University of Edinburgh
School of Informatics
INFR11199 - Advanced Database Systems (Spring 2024)

Tutorial Sheet 3

1. (Sorting and Hashing) Suppose the size of a page is 4 KB, and the size of the memory buffer is 1 MB (1024 KB).

   (a) We have a relation of size 800 KB. How many page I/Os are required to sort this relation and write the sorted relation back to disk?

   (b) We have a relation of size 5000 KB. How many page I/Os are required to sort this relation and write the sorted relation back to disk?

   (c) What is the size of the largest relation that would need two passes to sort?

   (d) What is the size of the largest relation we can possibly hash in two passes (i.e., with just one partitioning phase)?

   (e) Suppose we have a relation of size 3000 KB. We are executing a `DISTINCT` query on a column `age`, which has only two distinct values, evenly distributed. Would sorting or hashing be better here, and why?

   (f) Now suppose we were executing a `GROUP BY` on `age` instead. Would sorting or hashing be better here, and why?

2. (Joins) Consider the following database of students and assignment submissions and the SQL query:

```
CREATE TABLE Students (
  student_id INTEGER PRIMARY KEY,
  ...
);
CREATE TABLE Assignments(
  assignment_number INTEGER,
  student_id INTEGER REFERENCES Students(student_id),
  ...
);
SELECT *
  FROM Students, Assignments
 WHERE Students.student_id = Assignments.student_id;
```

Assume the following:

- `Students` has 20 pages, with 200 records per page
- `Assignments` has 40 pages, with 250 records per page.

(a) What is the I/O cost of a simple nested loop join for `Students` $\bowtie$ `Assignments`?

(b) What is the I/O cost of a simple nested loop join for `Assignments` $\bowtie$ `Students`?

(c) What is the I/O cost of a block nested loop join for `Students` $\bowtie$ `Assignments`? Assume our buffer size is $B = 12$ pages.

(d) What is the I/O cost of a block nested loop join for `Assignments` $\bowtie$ `Students`? Assume our buffer size is $B = 12$ pages.

(e) What is the I/O cost of an Index-Nested Loop Join for `Students` on $\bowtie$ `Assignments`?

Assume we have a *clustered* variant B index on `Assignments.student_id`, in the form of a height 2 B+ tree. Assume that: index (non-leaf) nodes and leaf pages are not cached; all hits are on the same leaf page; and all hits are also on the same data page. only one matching page

(f) Now assume we have an *unclustered* variant B index on `Assignments.student_id`, in the form of a height 2 B+ tree. Assume that index node pages and leaf pages are never cached, and we only need to read the relevant leaf page once for each record of `Students`, and all hits are on the same leaf page.

What is the I/O cost of an Index-Nested Loop Join for `Students` $\bowtie$ `Assignments`?

Hint: The foreign key in `Assignments` may play a role in how many accesses we do per record. no info on total number of matching tuples

(g) What is the cost of an *unoptimized* sort-merge join for `Students` $\bowtie$ `Assignments`? Assume we have $B = 12$ buffer pages.

(h) What is the cost of an *optimized* sort-merge join for `Students` $\bowtie$ `Assignments`? Assume we have $B = 12$ buffer pages.

(i) In the previous question, we had a buffer of $B = 12$ pages. If we shrank B enough, the answer we got might change.
How small can the buffer B be without changing the I/O cost answer we got?

(j) What is the I/O cost of Grace Hash Join on these tables?
Assume uniform hash partitioning and a buffer pool consisting of $B = 6$ pages.

University of Edinburgh
School of Informatics
INFR11199 - Advanced Database Systems (Spring 2024)

Tutorial Sheet 3

1. (Sorting and Hashing) Suppose the size of a page is 4 KB, and the size of the memory buffer is 1 MB (1024 KB).

   (a) We have a relation of size 800 KB. How many page I/Os are required to sort this relation and write the sorted relation back to disk?

   Solution: 400 (= 200 + 200).
   200 to read in, 200 to write out. Since the relation is small enough to completely fit into the buffer, we only need to read it in, sort it (no I/Os required for sorting), then write the sorted pages back to disk.

   (b) We have a relation of size 5000 KB. How many page I/Os are required to sort this relation and write the sorted relation back to disk?

   Solution: 5000 (= 2 * 1250 * number of passes).
   2 passes. 5000 KB with 4KB per page means 1250 pages are needed to store the relation. We have B = 1024 / 4 = 256 pages in our buffer.
   Number of Passes = $1 + \lceil log_{255} \lceil 1250/256 \rceil \rceil = 2$.

   (c) What is the size of the largest relation that would need two passes to sort?

   Solution: 261,120 KB. (255 * 256 pages).

   (d) What is the size of the largest relation we can possibly hash in two passes (i.e., with just one partitioning phase)?

   Solution: 261,120 KB.

(e) Suppose we have a relation of size 3000 KB. We are executing a `DISTINCT` query on a column `age`, which has only two distinct values, evenly distributed. Would sorting or hashing be better here, and why?

> **Solution:** Hashing, which allows us to remove duplicates early on and potentially improve performance (in this case, we might be able to finish in 1 pass, instead of 2 for sorting).

(f) Now suppose we were executing a `GROUP BY` on `age` instead. Would sorting or hashing be better here, and why?

> **Solution:** Sorting because hashing won't work; each partition is larger than memory, so no amount of hash partitioning will suffice.

2. (Joins) Consider the following database of students and assignment submissions and the SQL query:

```
CREATE TABLE Students (
  student_id INTEGER PRIMARY KEY,
  ...
);
CREATE TABLE Assignments(
  assignment_number INTEGER,
  student_id INTEGER REFERENCES Students(student_id),
  ...
);
SELECT *
  FROM Students, Assignments
 WHERE Students.student_id = Assignments.student_id;
```

Assume the following:

- `Students` has 20 pages, with 200 records per page
- `Assignments` has 40 pages, with 250 records per page.

(a) What is the I/O cost of a simple nested loop join for `Students` $\bowtie$ `Assignments`?

> **Solution:** 160,020 I/Os.
> The cost of a SNLJ is: #pages(S) + #records(S) · #pages(A).
> Plugging in the numbers gives us $20 + (20 \cdot 200) \cdot 40 = 160,020$ I/Os.

(b) What is the I/O cost of a simple nested loop join for `Assignments` $\bowtie$ `Students`?

Solution: 200,040 I/Os.
The cost of a SNLJ is: #pages(A) + #records(A) · #pages(S).
Plugging in the numbers gives us $40 + (40 \cdot 250) \cdot 20 = 200,040$ I/Os.

(c) What is the I/O cost of a block nested loop join for `Students` ⋈ `Assignments`? Assume our buffer size is $B = 12$ pages.

Solution: 100 I/Os.
Since `Students` is the outer table, we calculate the number of blocks of `Students`: #pages(S) / (B-2) = 20 / 10 = 2. Thus, the final cost is #pages(S) plus 2 passes through all pages(A), or $20 + 2 \cdot 40 = 100$ I/Os.

(d) What is the I/O cost of a block nested loop join for `Assignments` ⋈ `Students`? Assume our buffer size is $B = 12$ pages.

Solution: 120 I/Os.
Since `Assignments` is the outer table, we calculate the number of blocks of `Assignments`: #pages(A) / (B-2) = 40 / 10 = 4. Thus, the final cost is #pages(A) plus 4 passes through all pages(S), or $40 + 4 \cdot 20 = 120$ I/Os.

(e) What is the I/O cost of an Index-Nested Loop Join for `Students` on ⋈ `Assignments`?

Assume we have a *clustered* variant B index on `Assignments.student_id`, in the form of a height 2 B+ tree. Assume that: index (non-leaf) nodes and leaf pages are not cached; all hits are on the same leaf page; and all hits are also on the same data page.

Solution: 16,020 I/Os.
The formula is #pages(S) + #records(S) * (cost of index lookup).
The cost of index lookup is 3 I/Os to access the leaf, and 1 I/O to access the data page for all matching records.
So the total cost is $20 + 4000 \cdot 4 = 16,020$ I/Os.

(f) Now assume we have an *unclustered* variant B index on `Assignments.student_id`, in the form of a height 2 B+ tree. Assume that index node pages and leaf pages are never cached, and we only need to read the relevant leaf page once for each record of `Students`, and all hits are on the same leaf page.

What is the I/O cost of an Index-Nested Loop Join for `Students` ⋈ `Assignments`?

Hint: The foreign key in `Assignments` may play a role in how many accesses we do per record.

Solution: 22,020 I/Os.
The formula is #pages(S) + #records(S) * (cost of index lookup).
This time though, the cost of index lookup is 3 I/Os to access the leaf, and 1 I/O to access the data page for *each matching record*.
How many records match per key? We actually haven't told you! But, we do know that we will eventually have to access each record exactly once (since each `Assignments` is foreign-keyed on a `student_id` – so there will be #records(A) = 10,000 data page lookups, one for each row.
So the total cost is $20 + 4000 \cdot 3 + 10000 = 22,020$ I/Os.

(g) What is the cost of an *unoptimized* sort-merge join for `Students` $\bowtie$ `Assignments`? Assume we have $B = 12$ buffer pages.

Solution: 300 I/Os.
The formula is (cost of sorting S) + (cost of sorting A) + #pages(S) + #pages(A).
For sorting S: The first pass will make two runs, which is mergeable in one merge pass; thus, we need two passes.
For sorting A: The first pass will make four runs, which is mergeable in one merge pass; thus, we need two passes.
Thus the total cost is $(2 \cdot 2\#pages(S)) + (2 \cdot 2\#pages(A)) + \#pages(S) + \#pages(A) = 5(\#pages(S) + \#pages(A)) = 5 \cdot 60 = 300$ I/Os.

(h) What is the cost of an *optimized* sort-merge join for `Students` $\bowtie$ `Assignments`? Assume we have $B = 12$ buffer pages.

Solution: 180 I/Os.
The difference from the above question is that we will skip the last write in the external sorting phase, and the initial read in the sort-merge phase. For this to be possible, all the runs of S and A in the last phase of external sorting should be able to fit into memory together. From the previous question, we know there are 2 + 4 = 6 runs, which fits just fine in our buffer of 12 pages.
The total cost is $300 - 2\#pages(S) - 2\#pages(A) = 300 - 120 = 180$ I/Os.

(i) In the previous question, we had a buffer of $B = 12$ pages. If we shrank B enough, the answer we got might change.
How small can the buffer B be without changing the I/O cost answer we got?

Solution: The restriction for optimized sort-merge join is that the number of final runs of S and A can both fit in memory simultaneously (i.e., the

number of runs of S + the number of runs of $A \leq B - 1$). We had 2 + 4 runs last time, which fit comfortably in $12 - 1$ buffer pages (recall that one page is reserved for output).
What about $B = 11$? We would still have $2 + 4 \leq 11 - 1$ runs.
What about $B = 10$? We would still have $2 + 4 \leq 10 - 1$ runs.
What about $B = 9$? Now we have 3 runs for S and 5 runs for A, which just exactly fits in $9 - 1$ buffer pages.
Since 9 buffer pages fits perfectly, any smaller would force more merge passes and thus more I/Os.

(j) What is the I/O cost of Grace Hash Join on these tables?
Assume uniform hash partitioning and a buffer pool consisting of $B = 6$ pages.

Solution: 180 I/Os.
For Grace Hash Join, we have to walk through what the partition sizes are like for each phase, one phase at a time. In the partitioning phase, we will proceed as in external hashing. We will load in 1 page a time and hash it into $B - 1 = 5$ partitions. This means the 20 pages of S get split into 4 pages per partition, and the 40 pages of A get split into 8 pages per partition.

Do we need to recursively partition? No! Remember that the stopping condition is that any table's partition fits in $B - 2 = 4$ buffer pages; the partitions of S satisfy this.

In the hash joining phase, the I/O cost is simply the total number of pages across all partitions – we read all of these in exactly once.

Thus the final I/O cost is 20 + 20 for partitioning S, 40 + 40 for partitioning A, and 20 + 40 for the hash join, for a total cost of 180 I/Os.

Advanced Database Systems

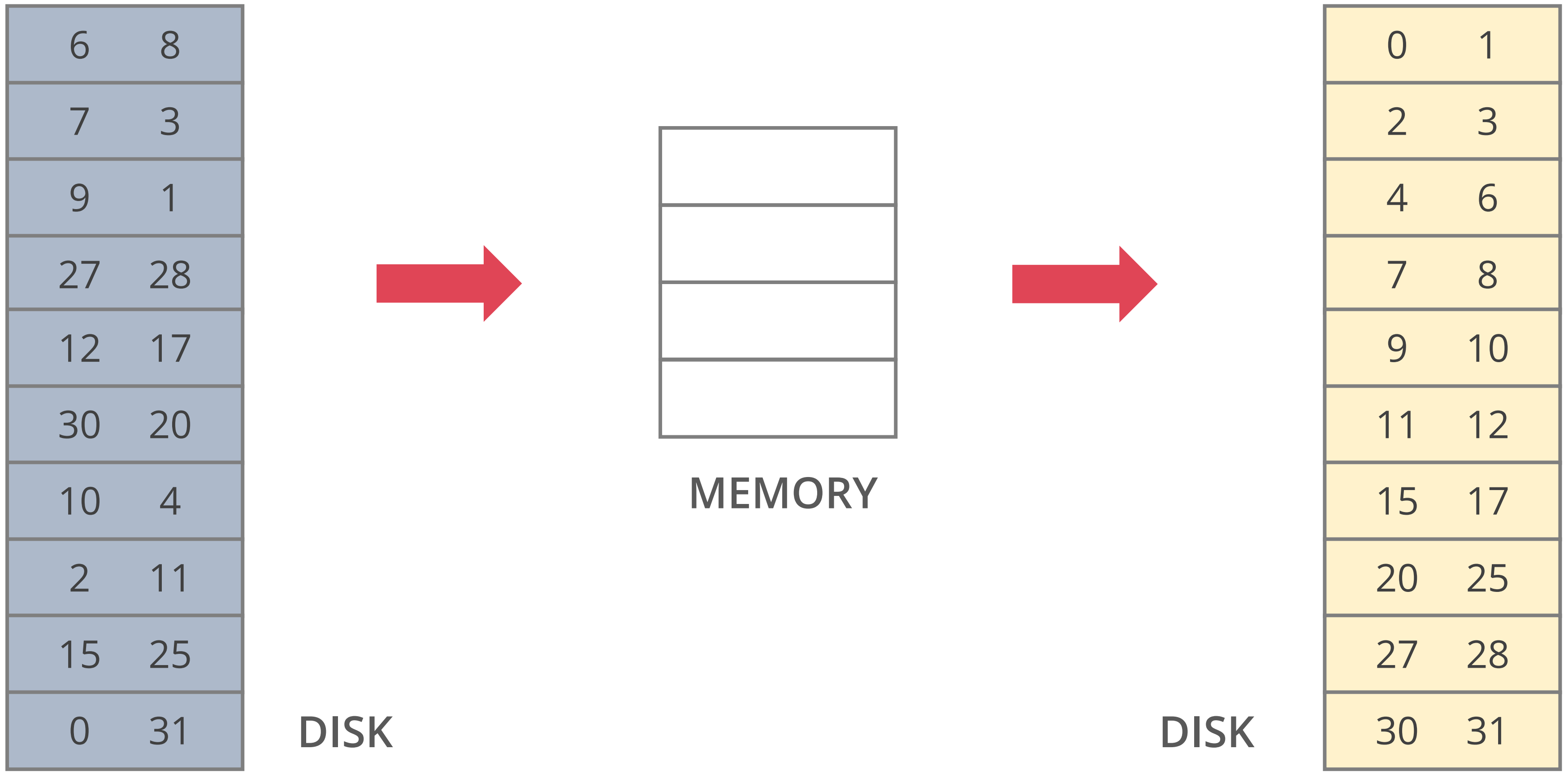
Spring 2024

Tutorial 3

External Sorting

General External Merge Sort

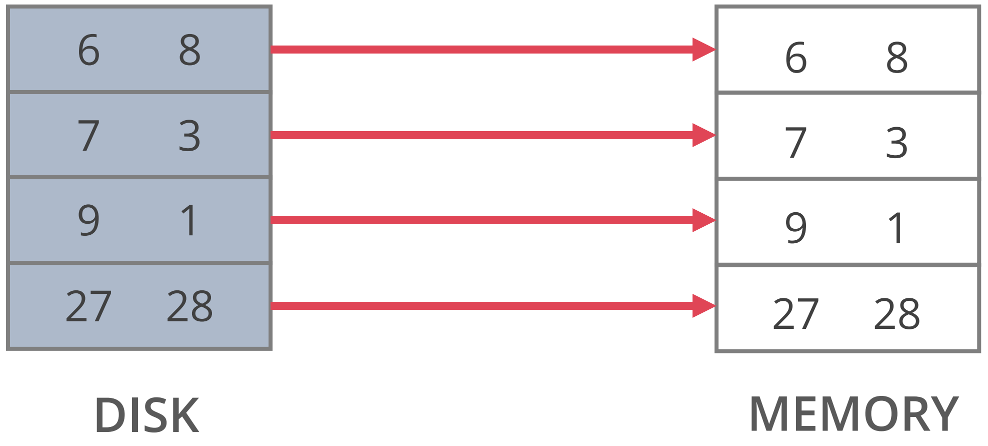
Goal: sort records on 10 data pages using 4 buffer pages



General External Merge Sort

Pass 0, Run 1

Read 4 pages into memory

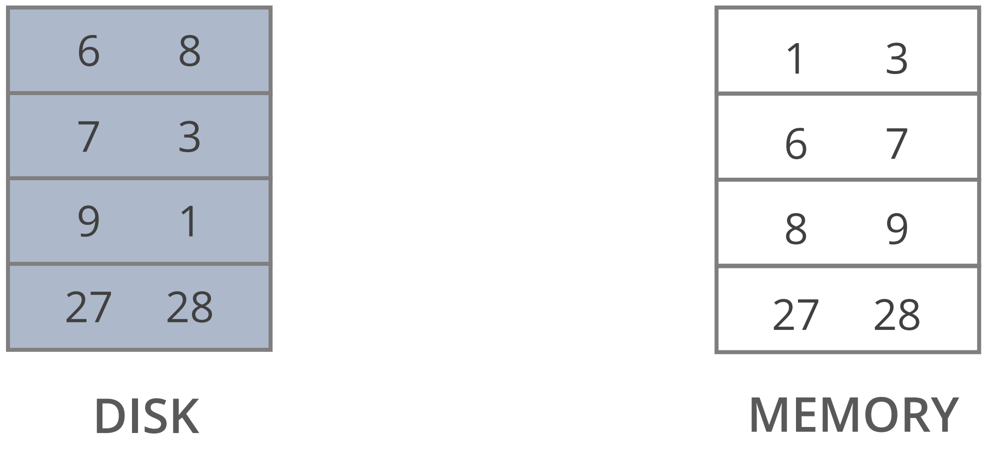


I/O Total (so far): 4

General External Merge Sort

Pass 0, Run 1

In-memory sort

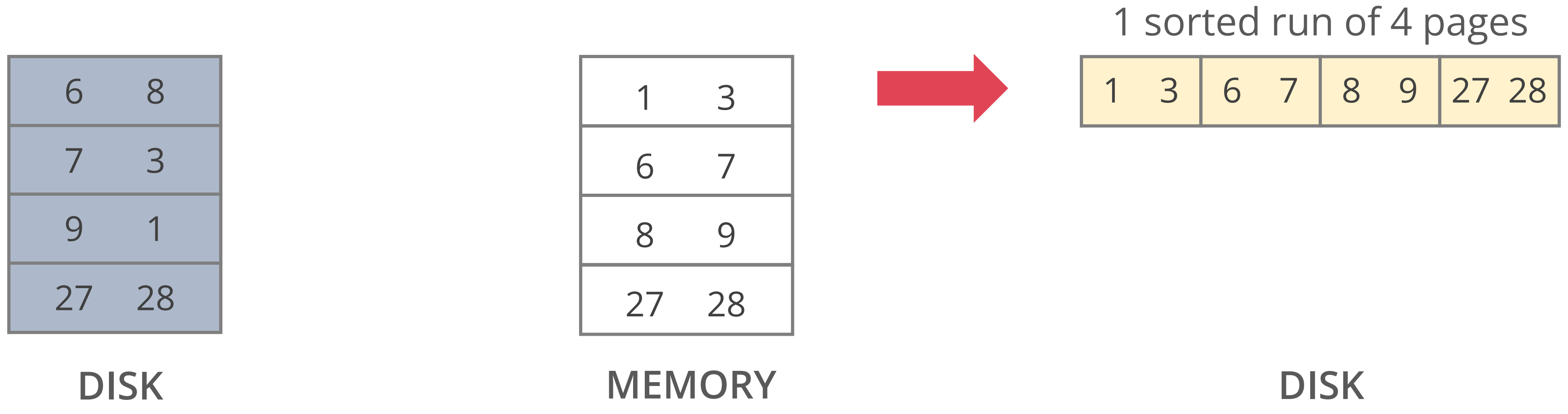


I/O Total (so far): 4

General External Merge Sort

Pass 0, Run 1

Write 4 sorted pages to disk

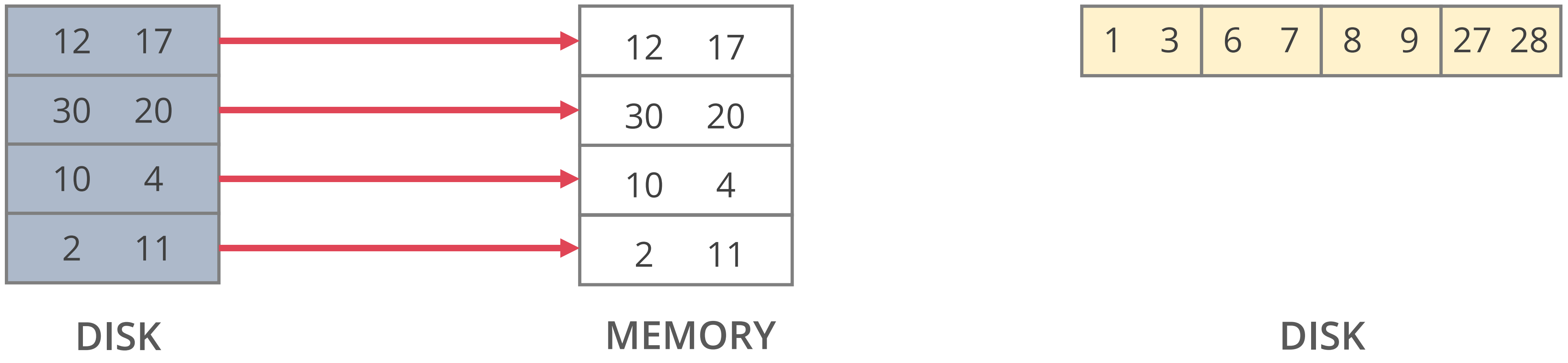


I/O Total (so far): 8

General External Merge Sort

Pass 0, Run 2

Read 4 pages into memory

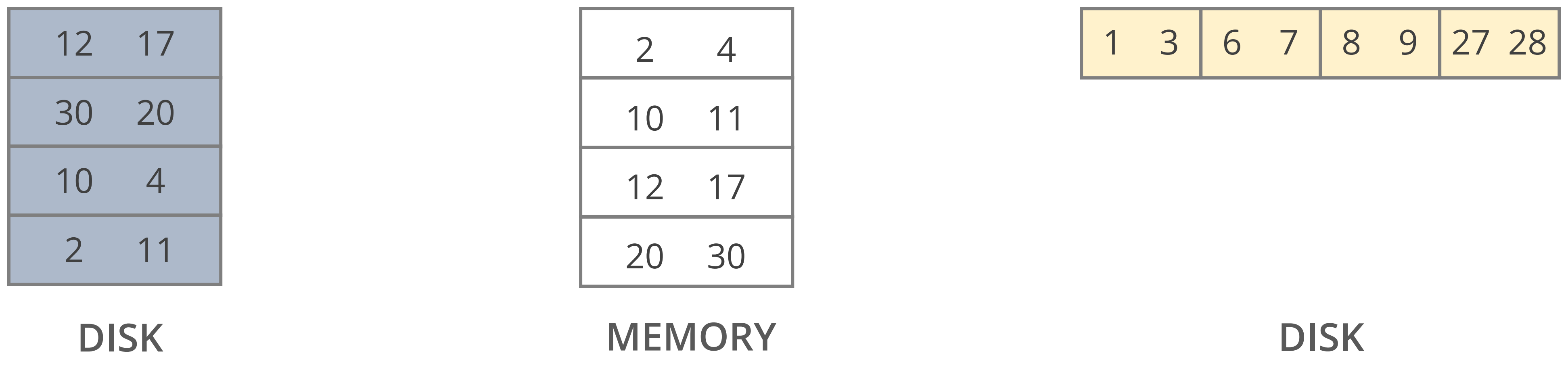


I/O Total (so far): 12

General External Merge Sort

Pass 0, Run 2

In-memory sort

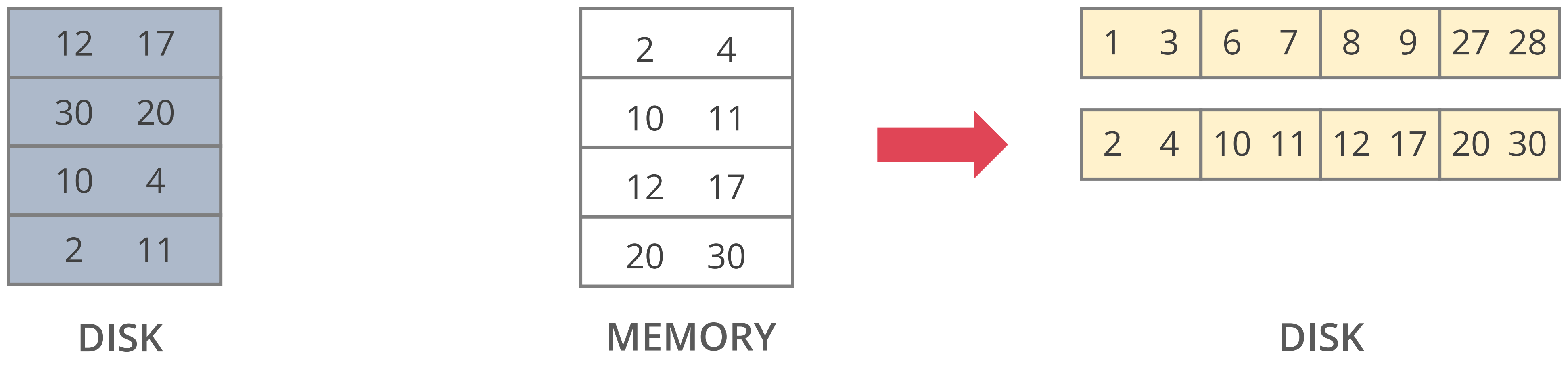


I/O Total (so far): 12

General External Merge Sort

Pass 0, Run 2

Write 4 sorted pages to disk

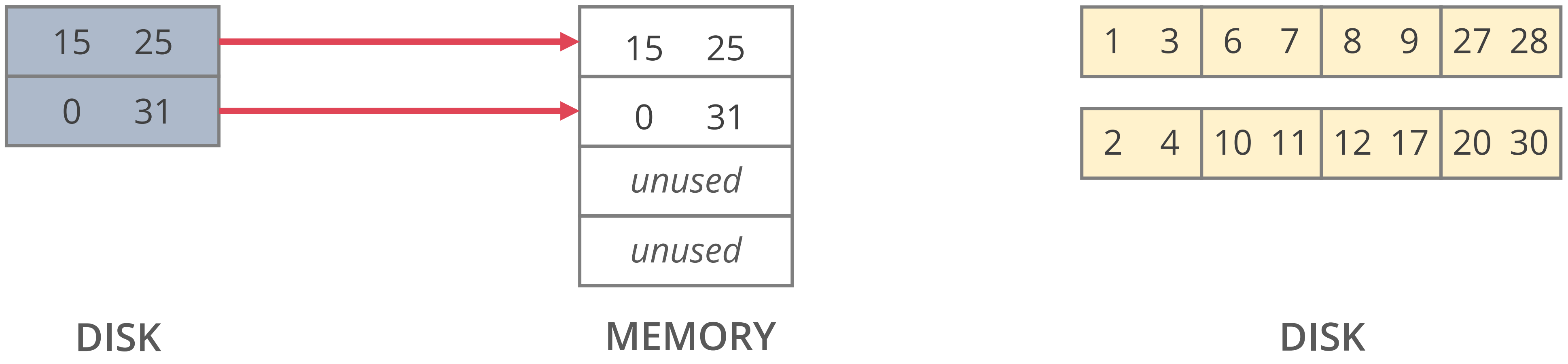


I/O Total (so far): 16

General External Merge Sort

Pass 0, Run 3

Read 2 pages into memory

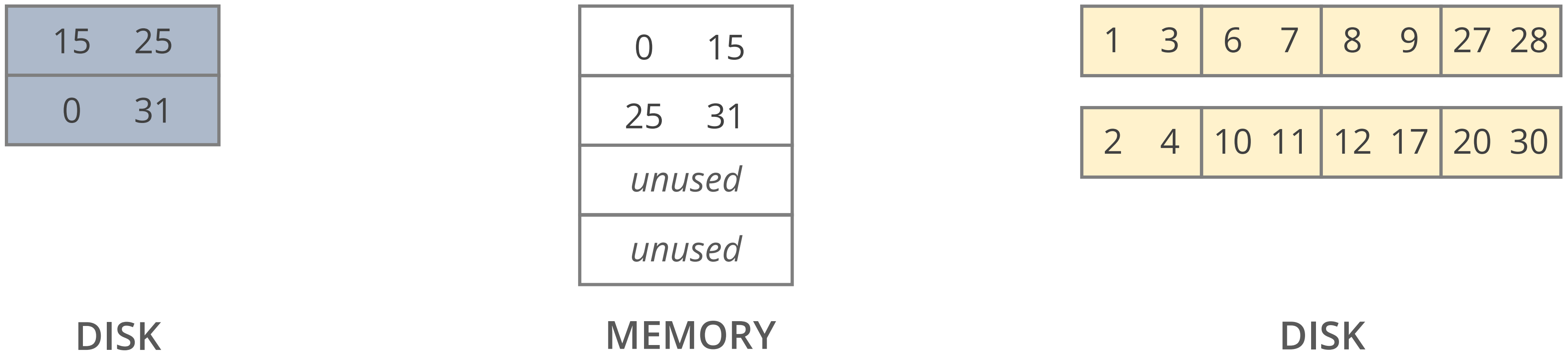


I/O Total (so far): 18

General External Merge Sort

Pass 0, Run 3

In-memory sort

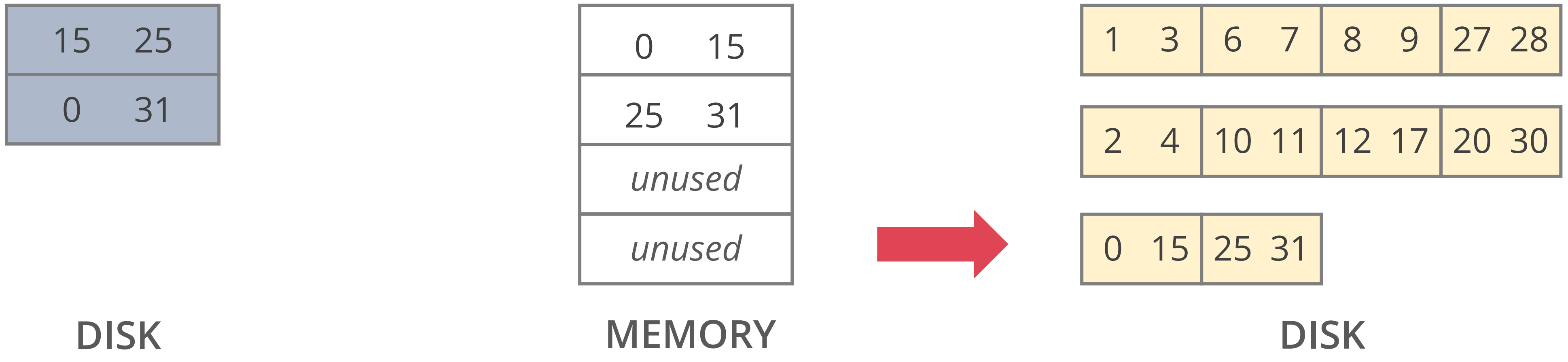


I/O Total (so far): 18

General External Merge Sort

Pass 0, Run 3

Write 2 sorted pages to disk

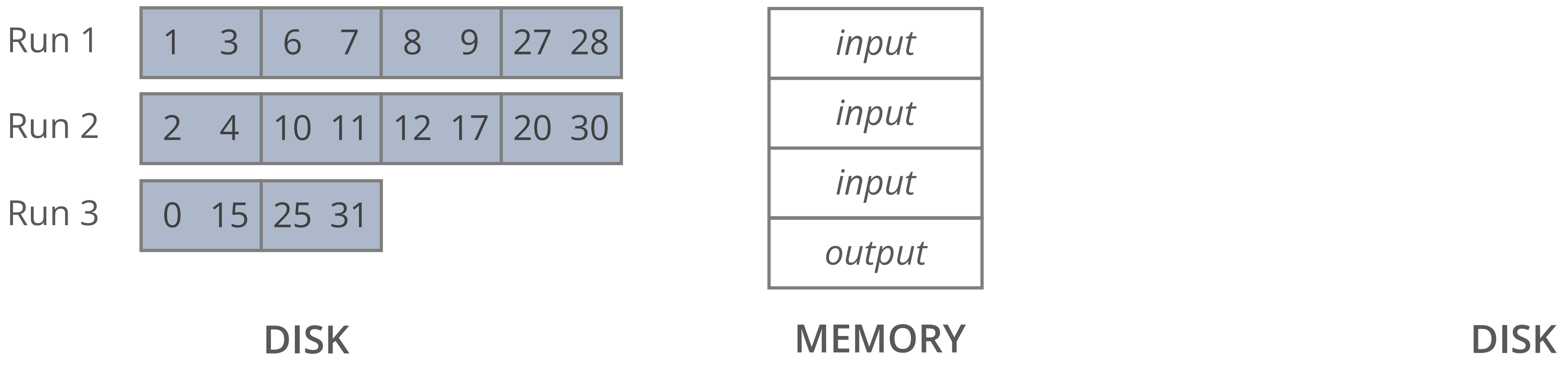


I/O Total (so far): 20

General External Merge Sort

Pass 1

Read 3 sorted runs into memory, write 1 sorted run to disk



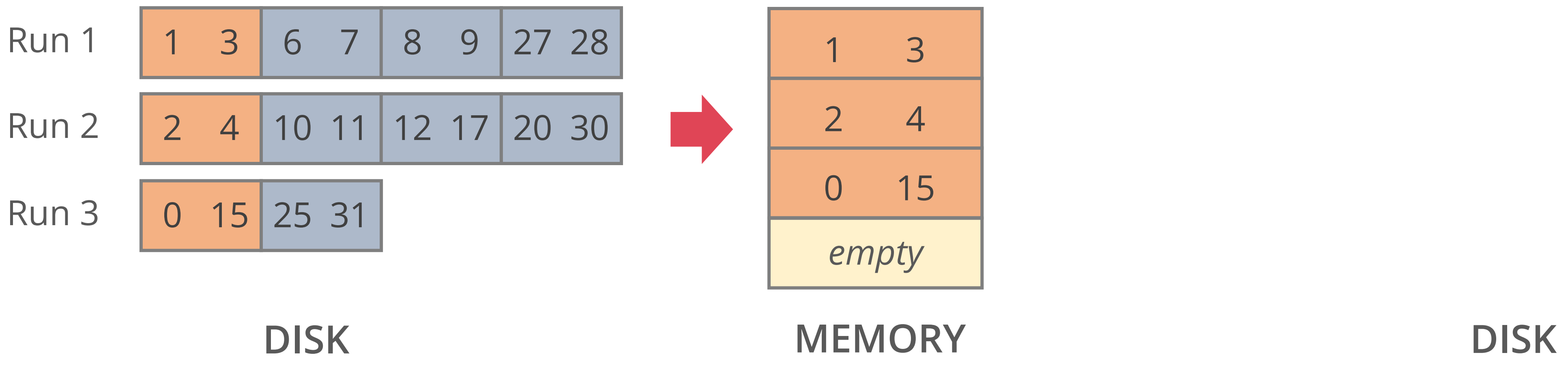
Reserve ***B-1*** input buffers and ***1*** output buffer. Load one page from each run at a time.

Store sorted results in output buffer. Write to disk when output buffer is full

General External Merge Sort

Pass 1

Read 3 sorted runs into memory, write 1 sorted run to disk

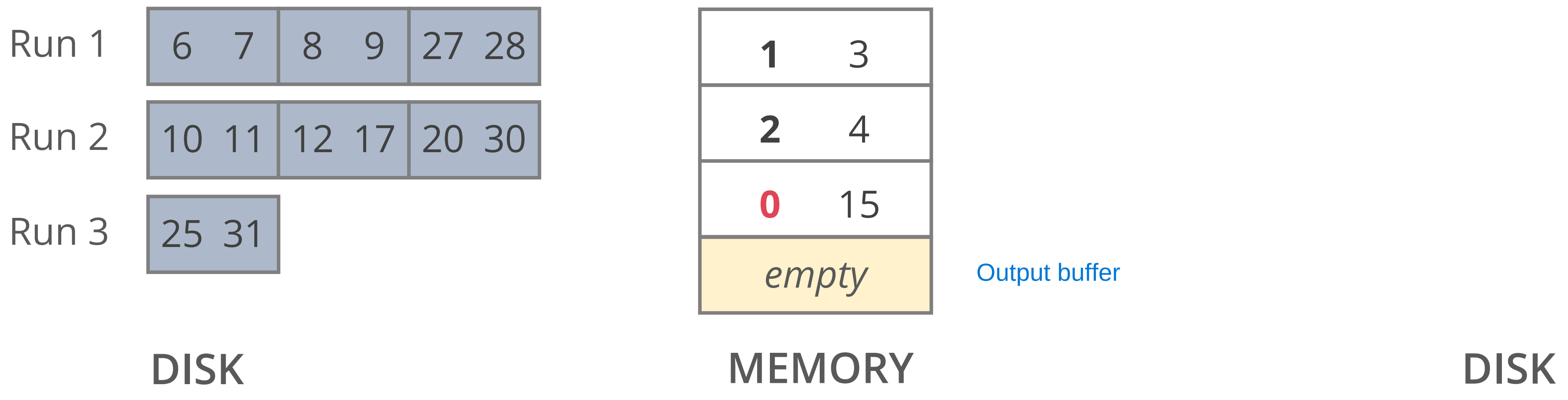


I/O Total (so far): 23

General External Merge Sort

Pass 1

Read 3 sorted runs into memory, write 1 sorted run to disk

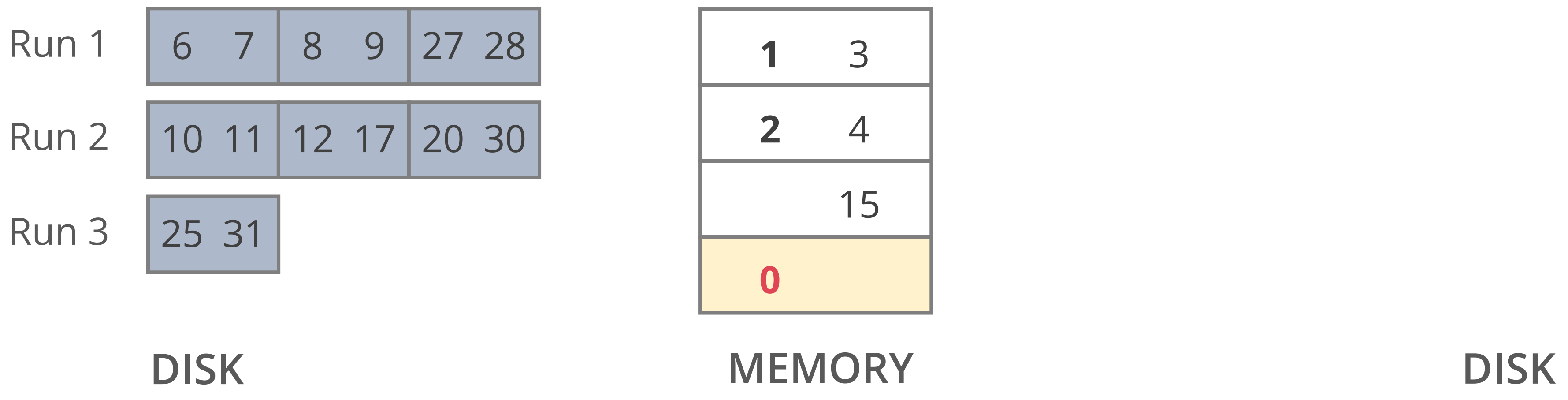


I/O Total (so far): 23

General External Merge Sort

Pass 1

Read 3 sorted runs into memory, write 1 sorted run to disk

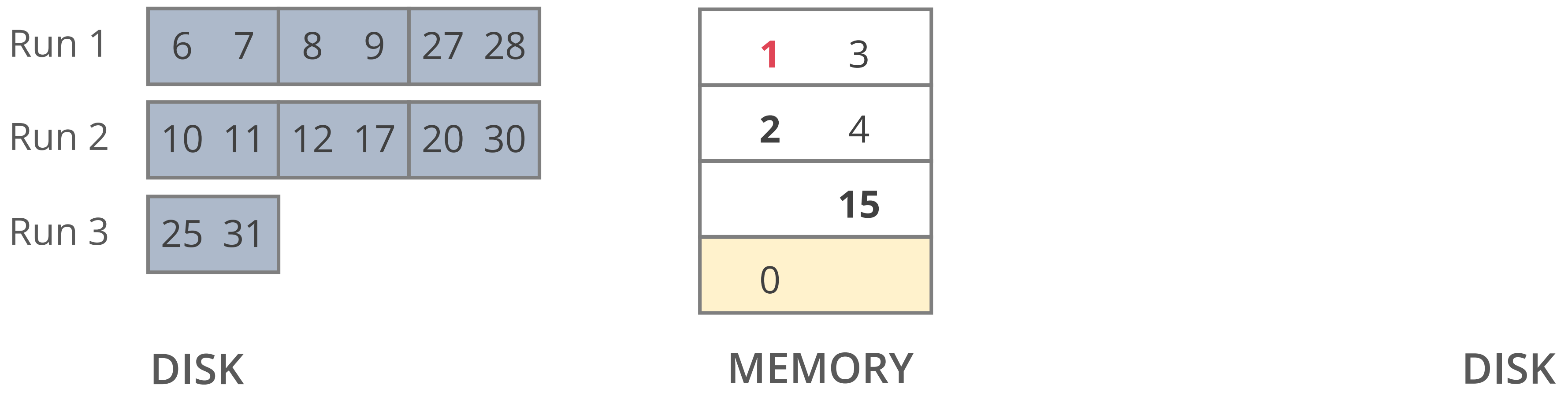


I/O Total (so far): 23

General External Merge Sort

Pass 1

Read 3 sorted runs into memory, write 1 sorted run to disk

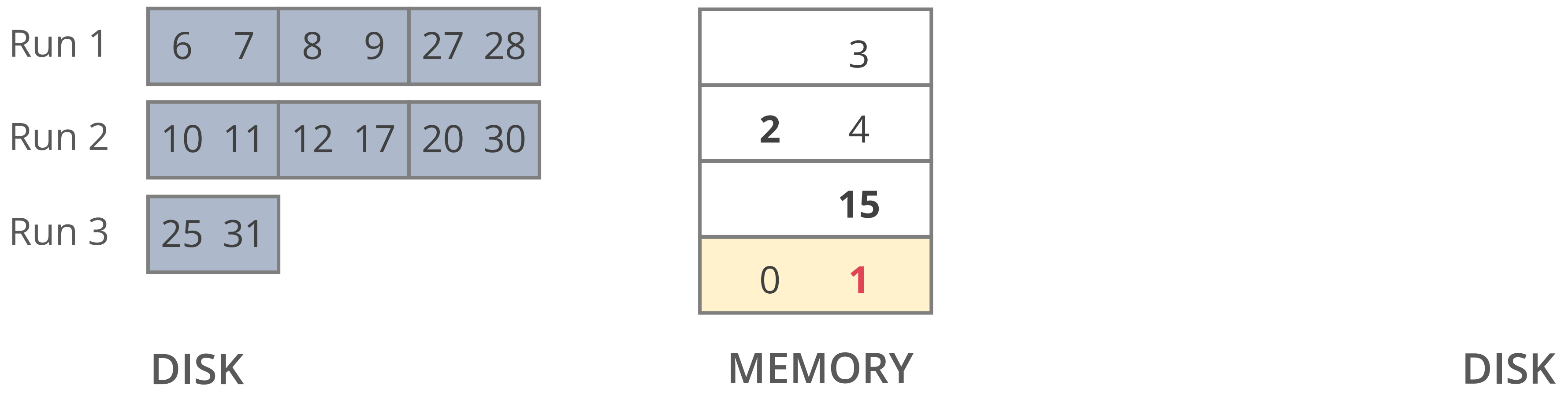


I/O Total (so far): 23

General External Merge Sort

Pass 1

Read 3 sorted runs into memory, write 1 sorted run to disk

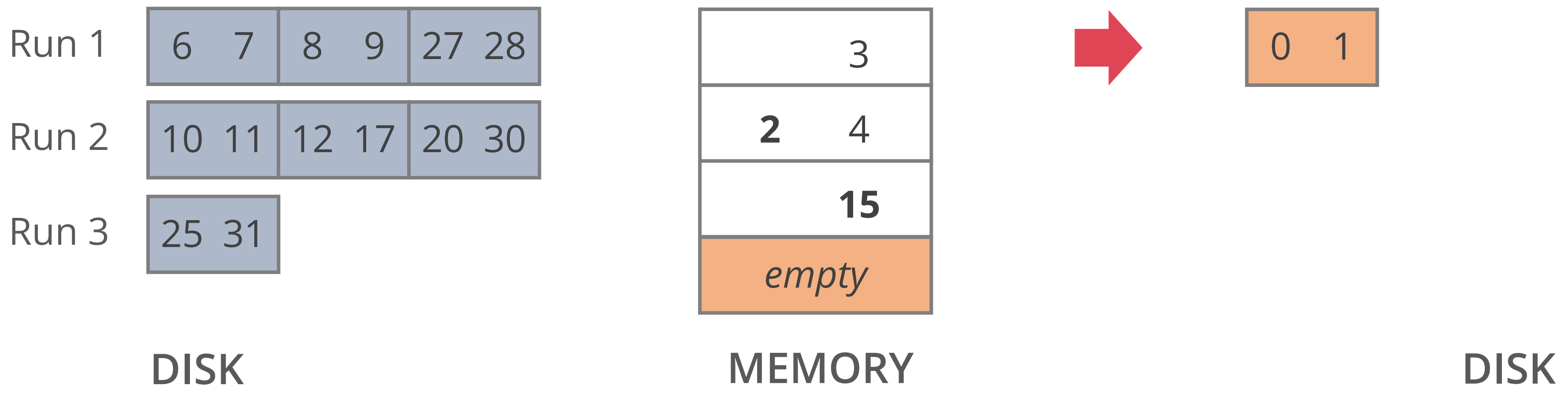


I/O Total (so far): 23

General External Merge Sort

Pass 1

Read 3 sorted runs into memory, write 1 sorted run to disk

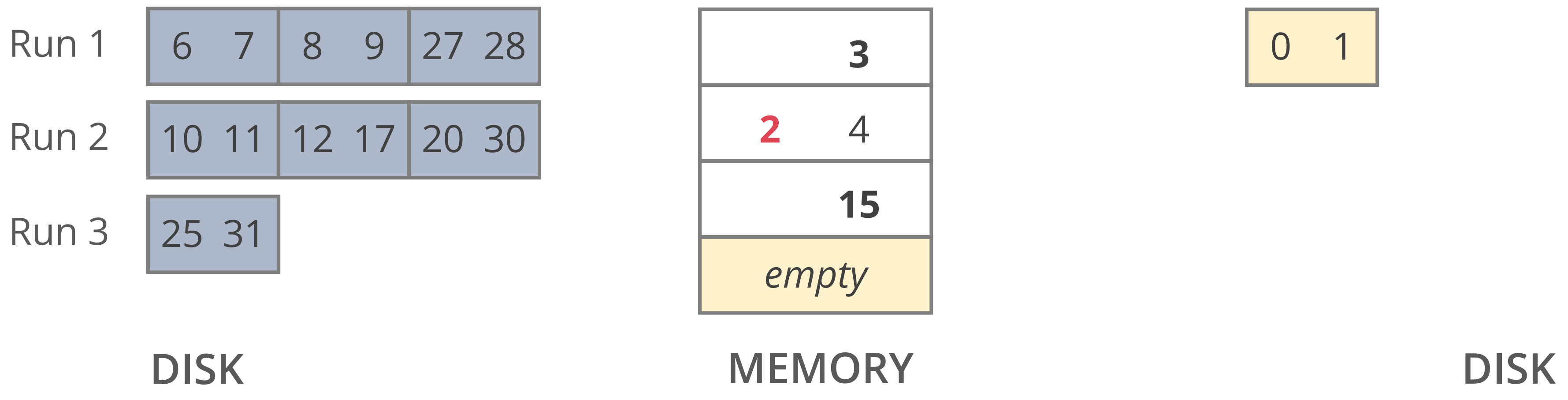


I/O Total (so far): 24

General External Merge Sort

Pass 1

Read 3 sorted runs into memory, write 1 sorted run to disk

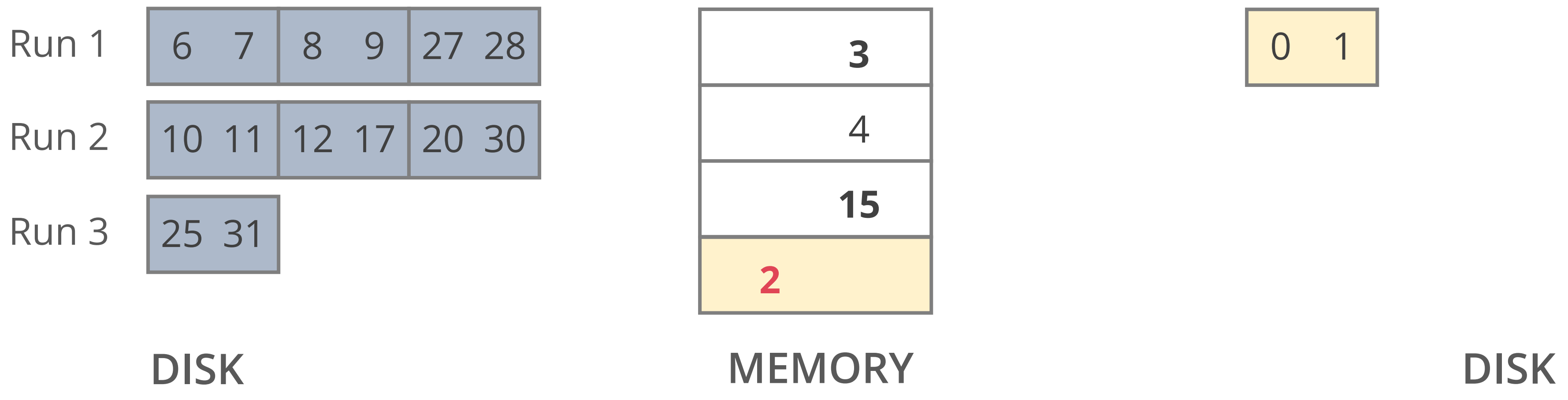


I/O Total (so far): 24

General External Merge Sort

Pass 1

Read 3 sorted runs into memory, write 1 sorted run to disk

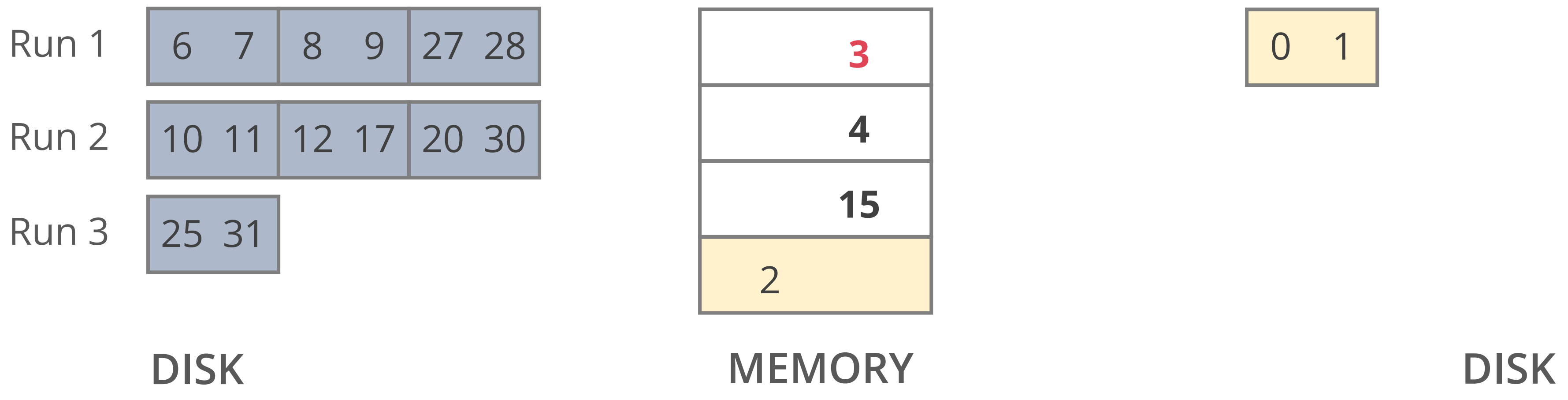


I/O Total (so far): 24

General External Merge Sort

Pass 1

Read 3 sorted runs into memory, write 1 sorted run to disk

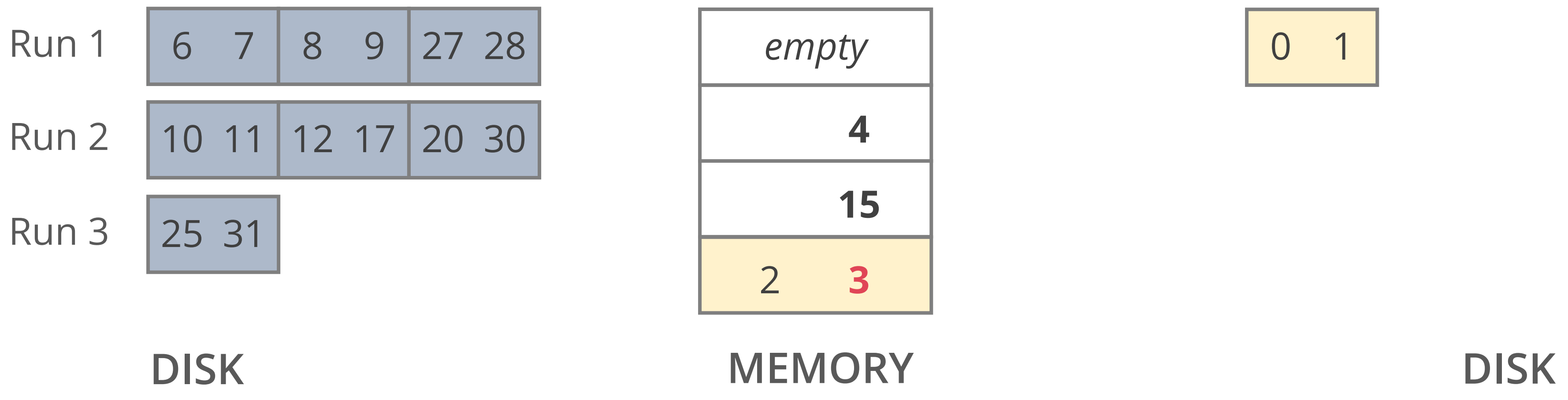


I/O Total (so far): 24

General External Merge Sort

Pass 1

Read 3 sorted runs into memory, write 1 sorted run to disk

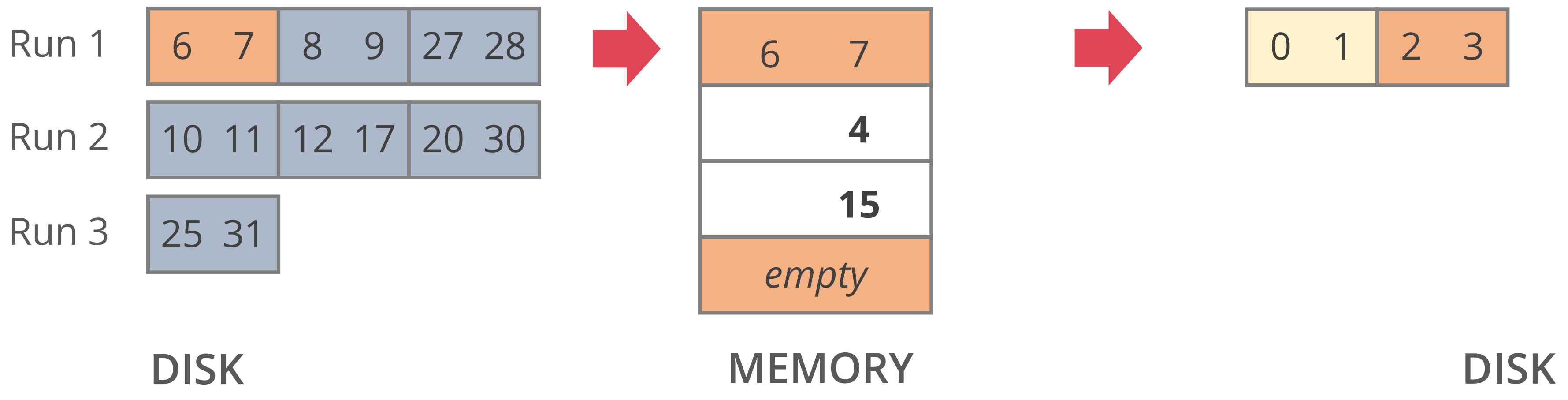


I/O Total (so far): 24

General External Merge Sort

Pass 1

Read 3 sorted runs into memory, write 1 sorted run to disk

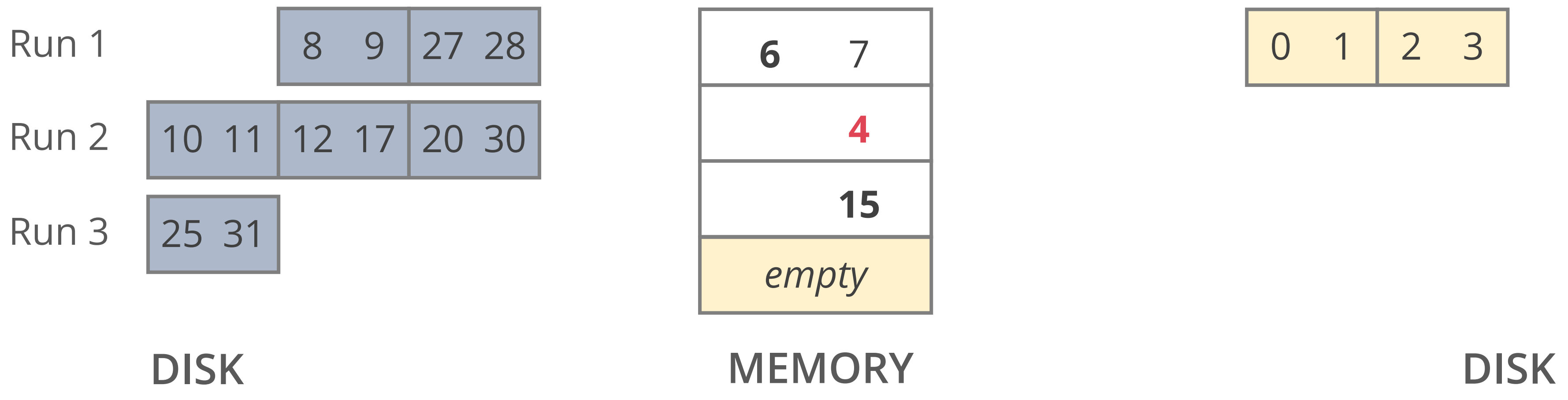


I/O Total (so far): 26

General External Merge Sort

Pass 1

Read 3 sorted runs into memory, write 1 sorted run to disk

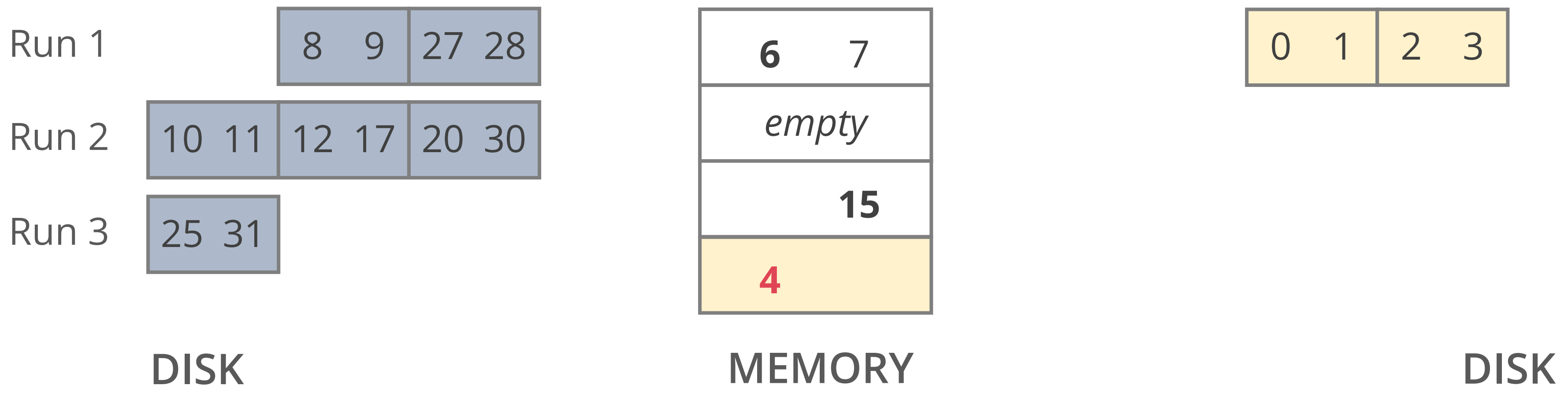


I/O Total (so far): 26

General External Merge Sort

Pass 1

Read 3 sorted runs into memory, write 1 sorted run to disk

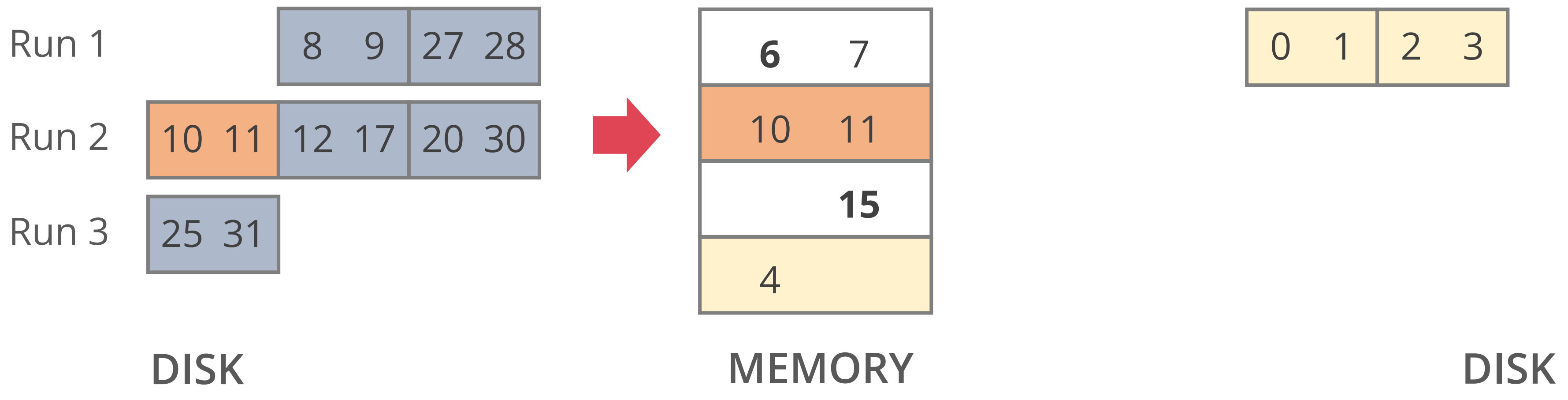


I/O Total (so far): 26

General External Merge Sort

Pass 1

Read 3 sorted runs into memory, write 1 sorted run to disk

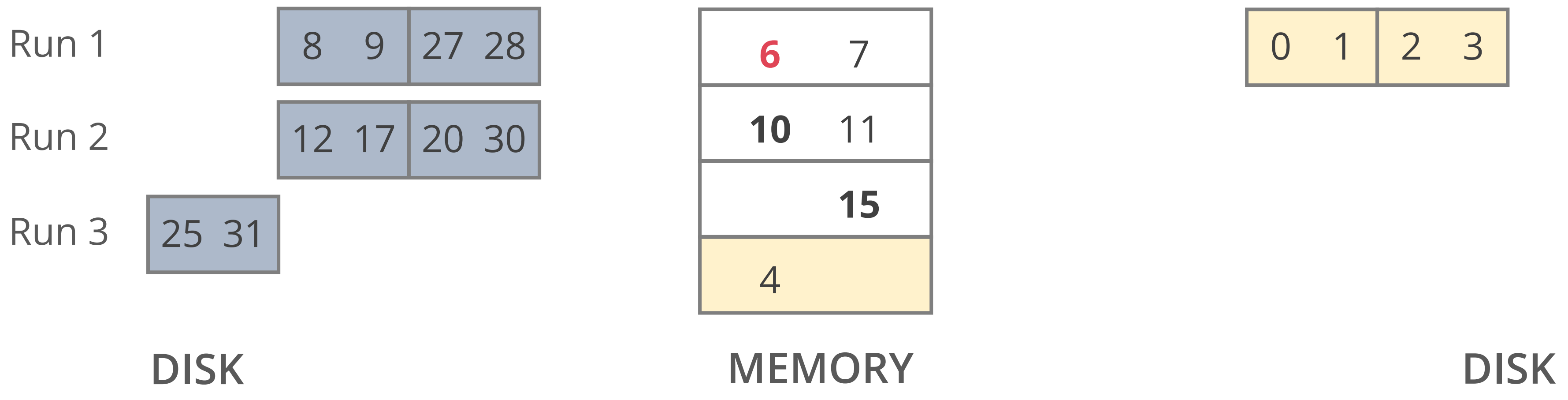


I/O Total (so far): 27

General External Merge Sort

Pass 1

Read 3 sorted runs into memory, write 1 sorted run to disk

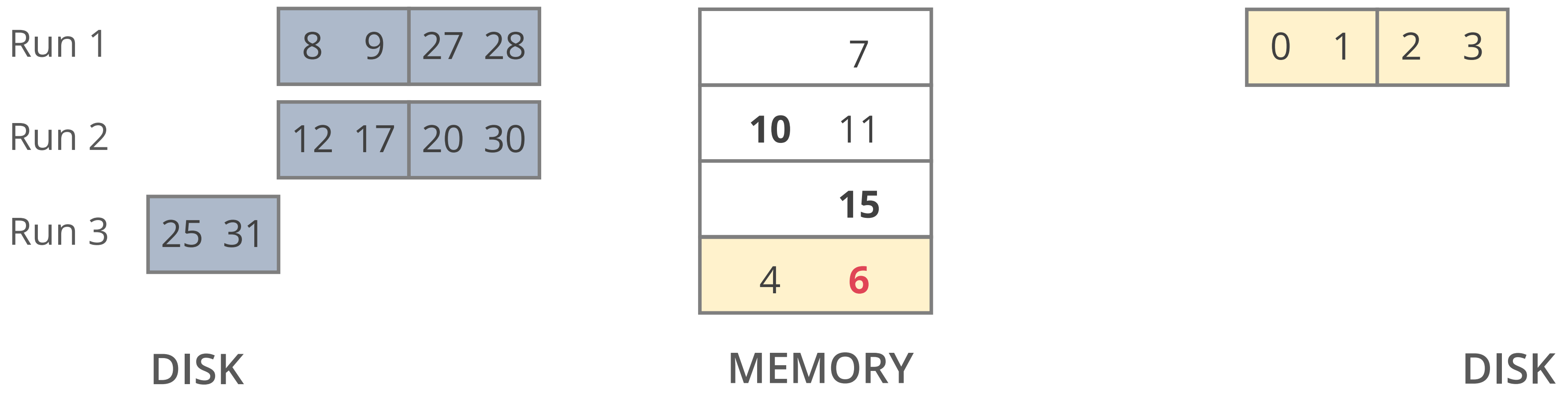


I/O Total (so far): 27

General External Merge Sort

Pass 1

Read 3 sorted runs into memory, write 1 sorted run to disk

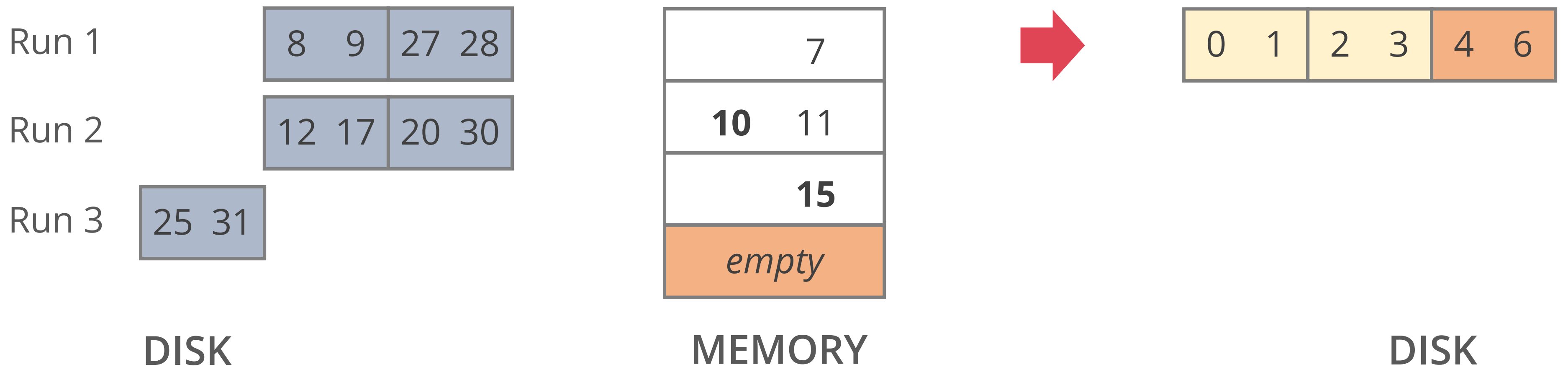


I/O Total (so far): 27

General External Merge Sort

Pass 1

Read 3 sorted runs into memory, write 1 sorted run to disk

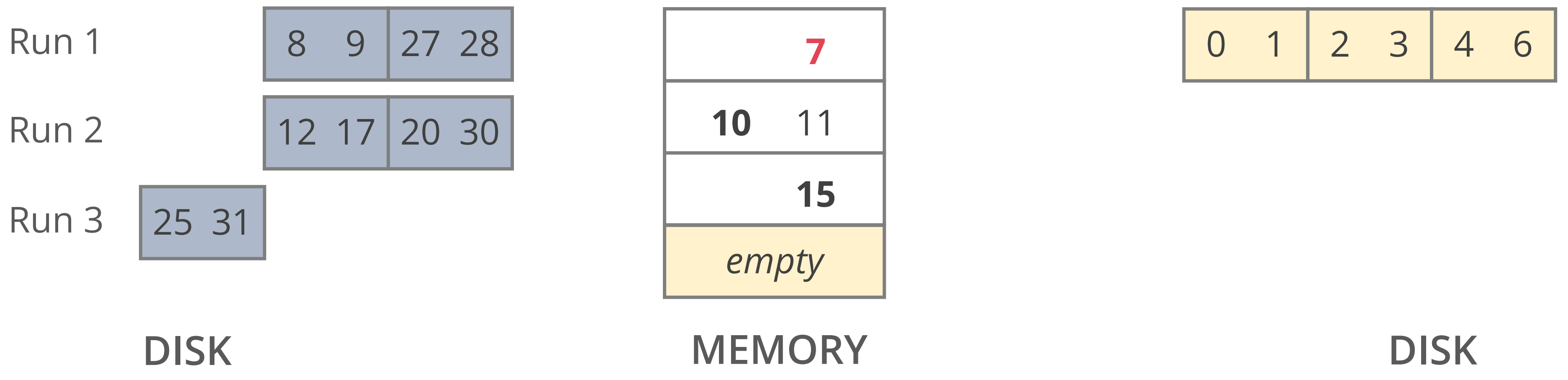


I/O Total (so far): 28

General External Merge Sort

Pass 1

Read 3 sorted runs into memory, write 1 sorted run to disk

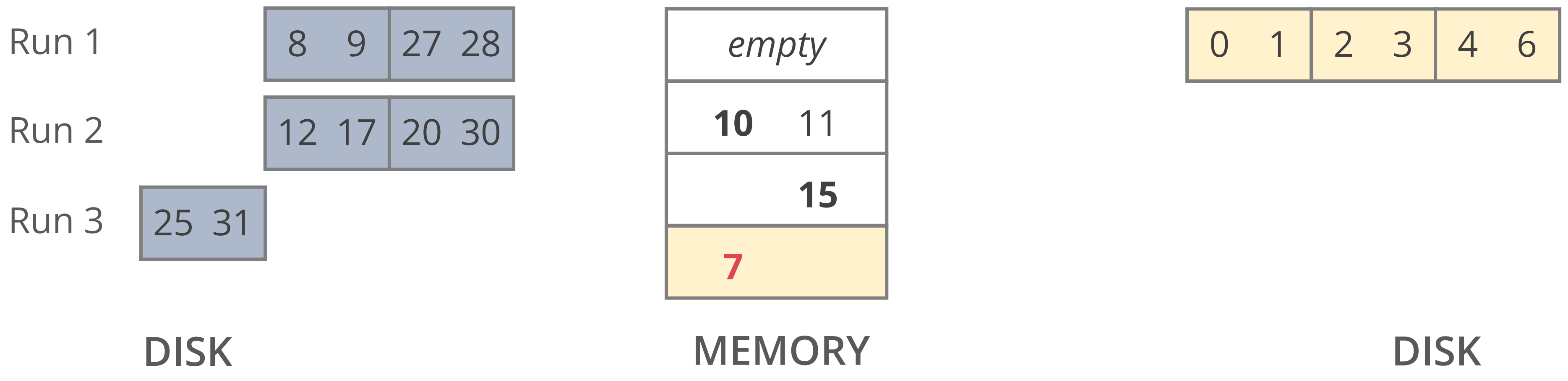


I/O Total (so far): 28

General External Merge Sort

Pass 1

Read 3 sorted runs into memory, write 1 sorted run to disk

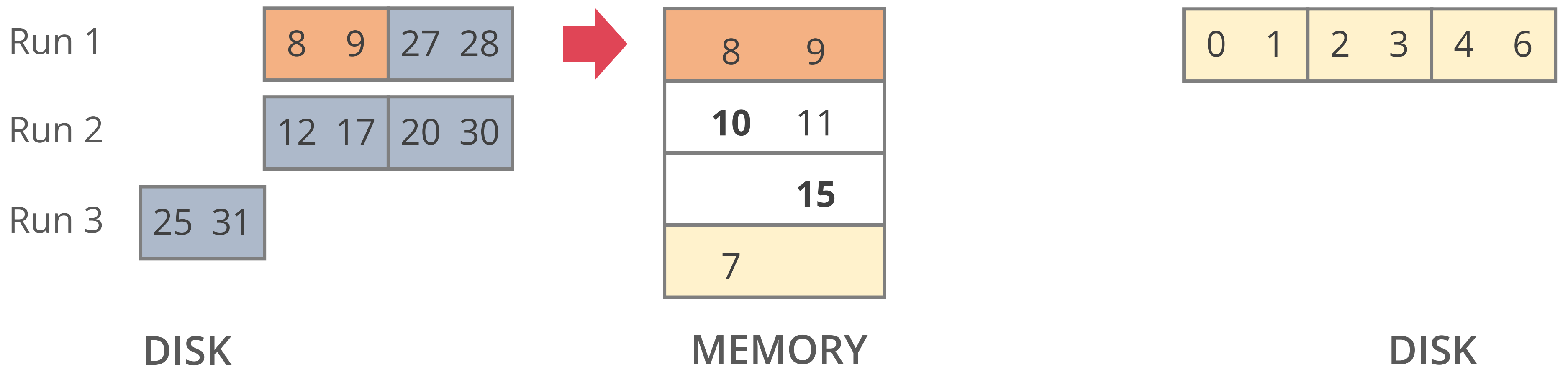


I/O Total (so far): 28

General External Merge Sort

Pass 1

Read 3 sorted runs into memory, write 1 sorted run to disk

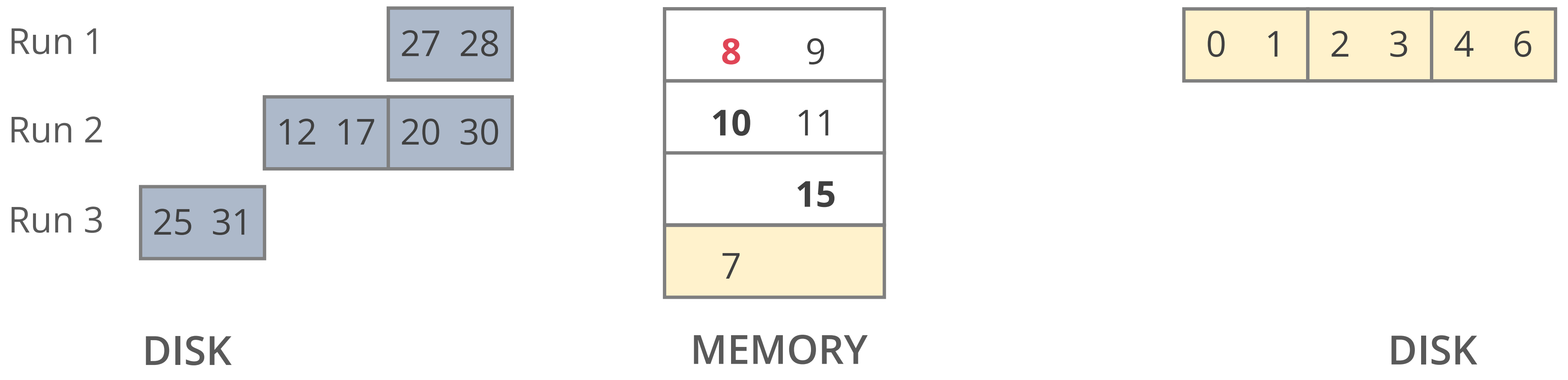


I/O Total (so far): 29

General External Merge Sort

Pass 1

Read 3 sorted runs into memory, write 1 sorted run to disk

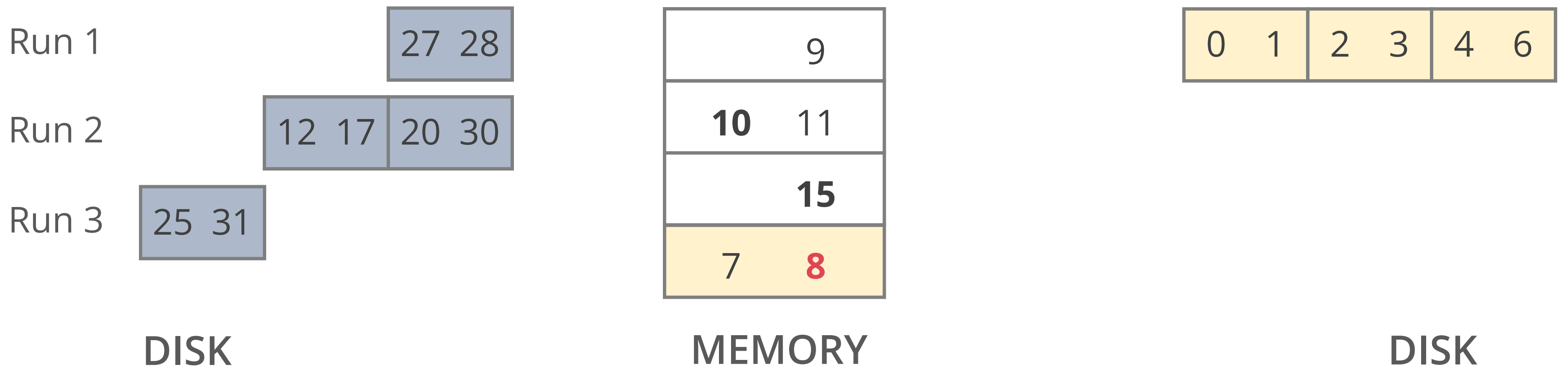


I/O Total (so far): 29

General External Merge Sort

Pass 1

Read 3 sorted runs into memory, write 1 sorted run to disk

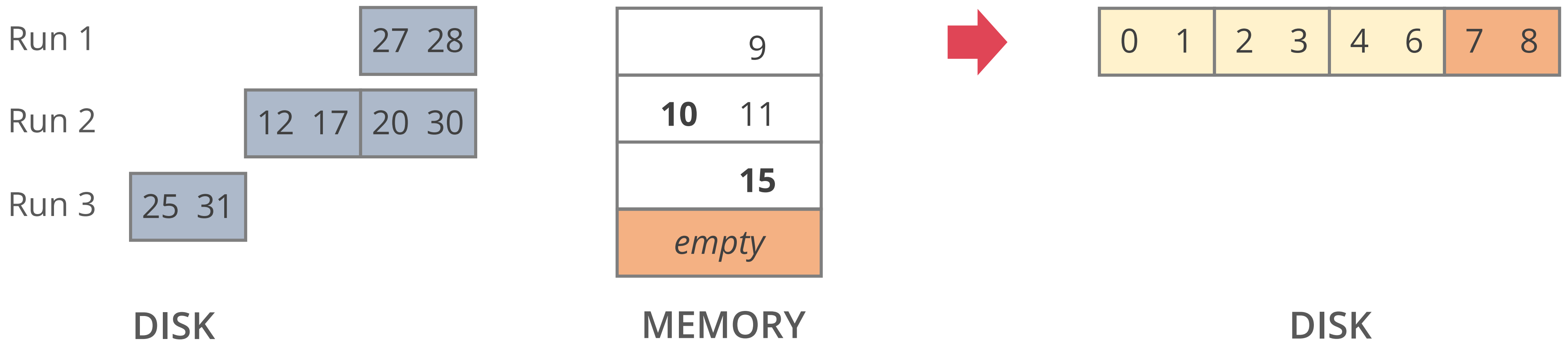


I/O Total (so far): 29

General External Merge Sort

Pass 1

Read 3 sorted runs into memory, write 1 sorted run to disk

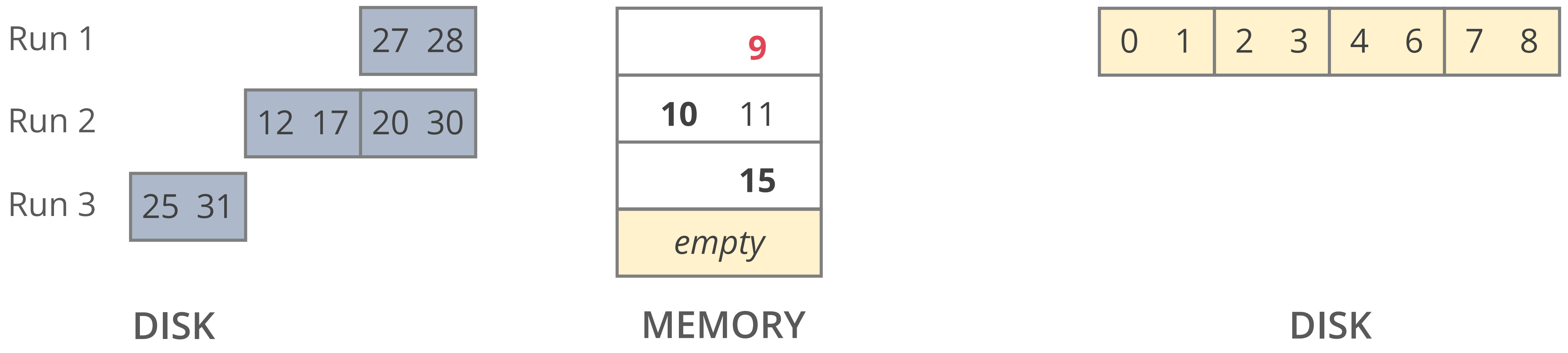


I/O Total (so far): 30

General External Merge Sort

Pass 1

Read 3 sorted runs into memory, write 1 sorted run to disk

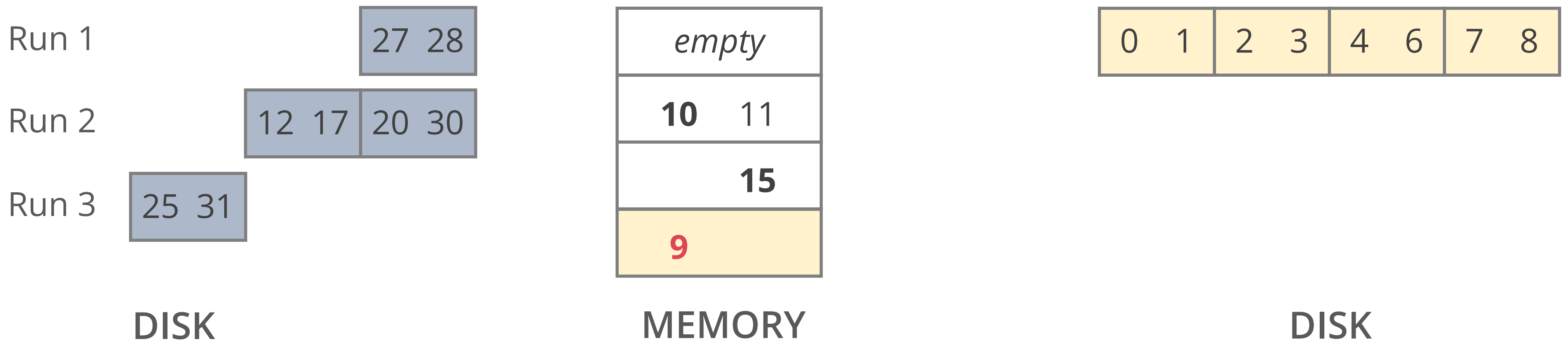


I/O Total (so far): 30

General External Merge Sort

Pass 1

Read 3 sorted runs into memory, write 1 sorted run to disk

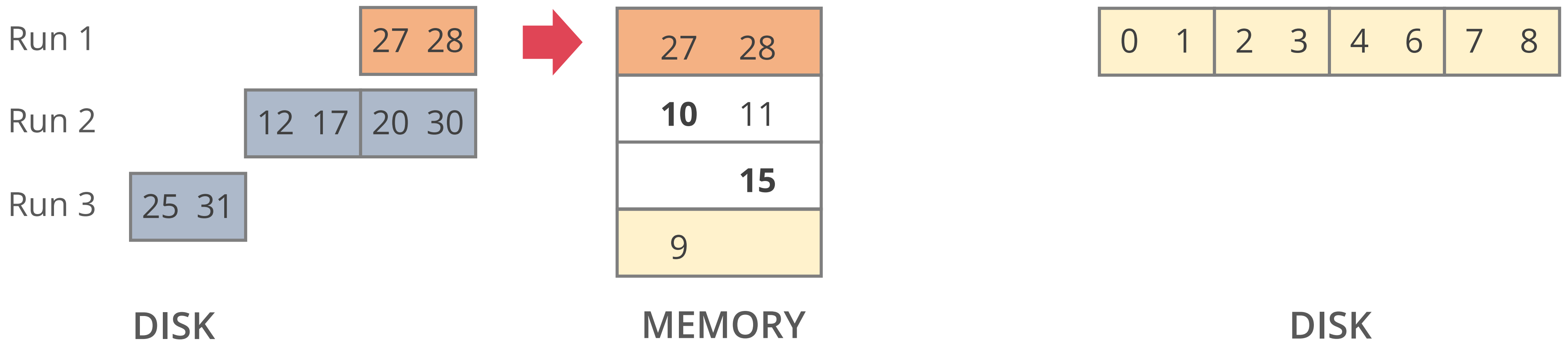


I/O Total (so far): 30

General External Merge Sort

Pass 1

Read 3 sorted runs into memory, write 1 sorted run to disk

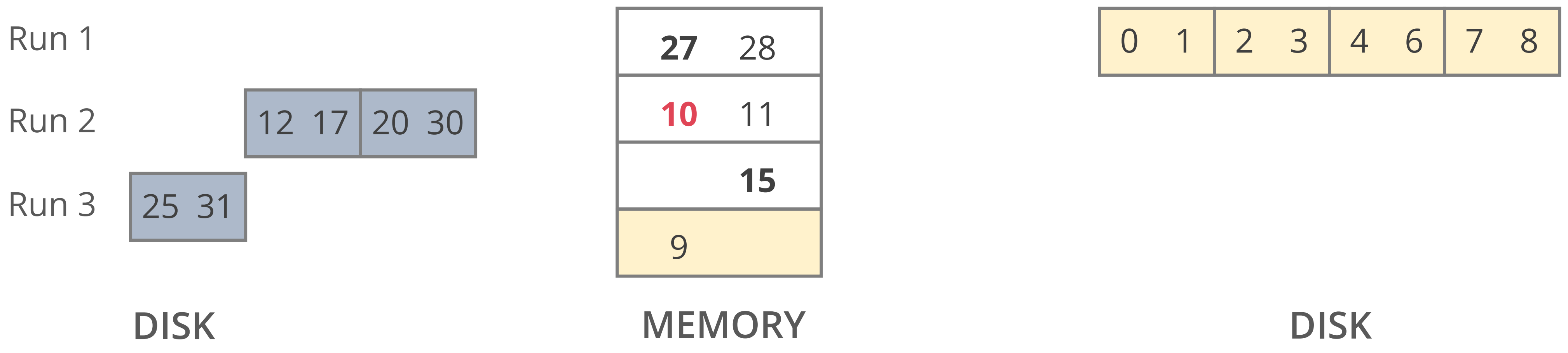


I/O Total (so far): 31

General External Merge Sort

Pass 1

Read 3 sorted runs into memory, write 1 sorted run to disk

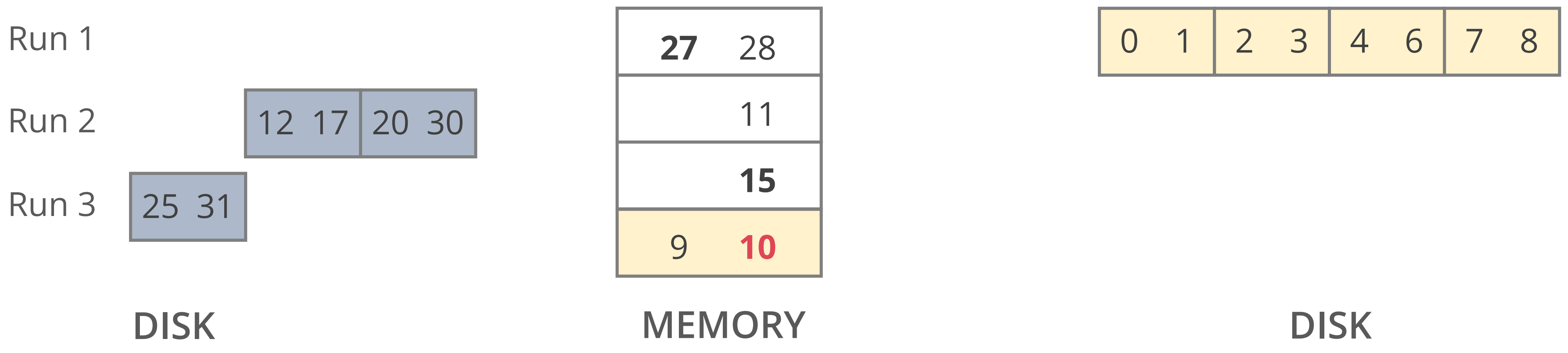


I/O Total (so far): 31

General External Merge Sort

Pass 1

Read 3 sorted runs into memory, write 1 sorted run to disk

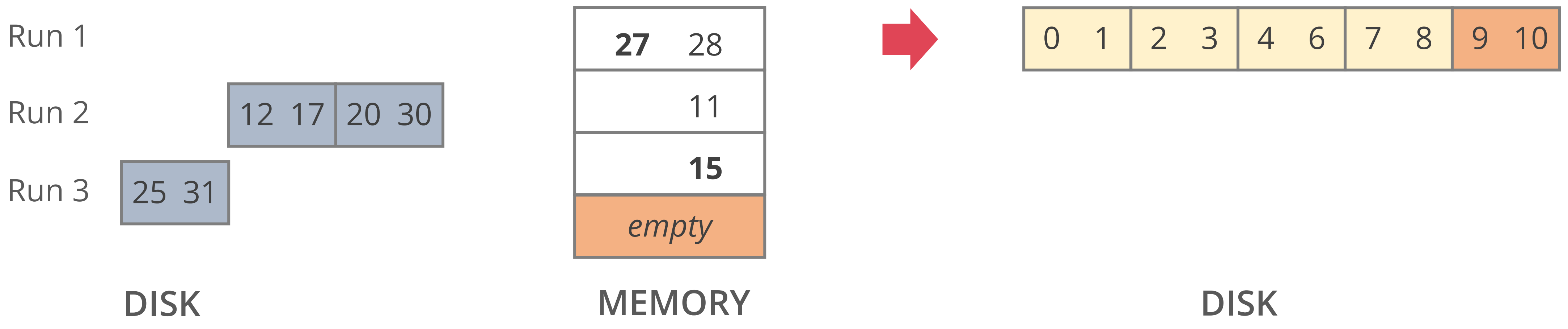


I/O Total (so far): 31

General External Merge Sort

Pass 1

Read 3 sorted runs into memory, write 1 sorted run to disk

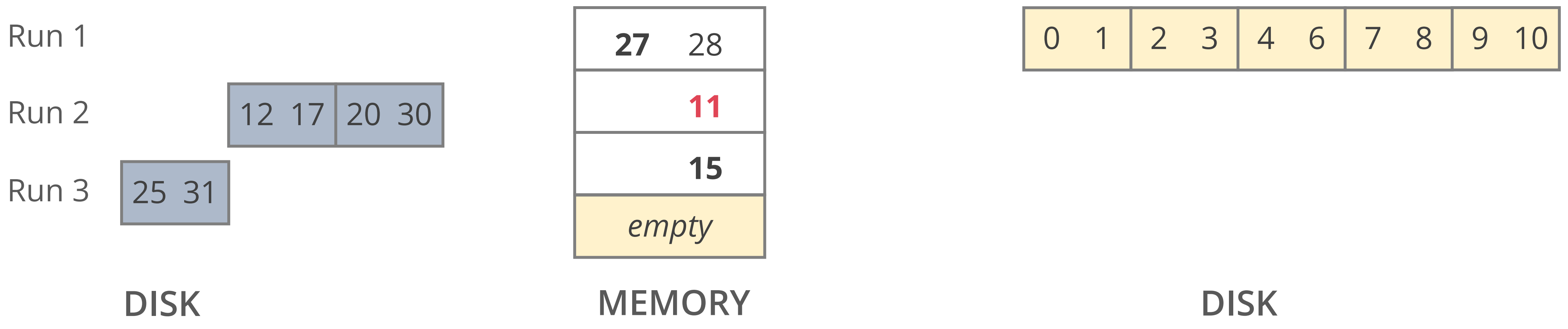


I/O Total (so far): 32

General External Merge Sort

Pass 1

Read 3 sorted runs into memory, write 1 sorted run to disk

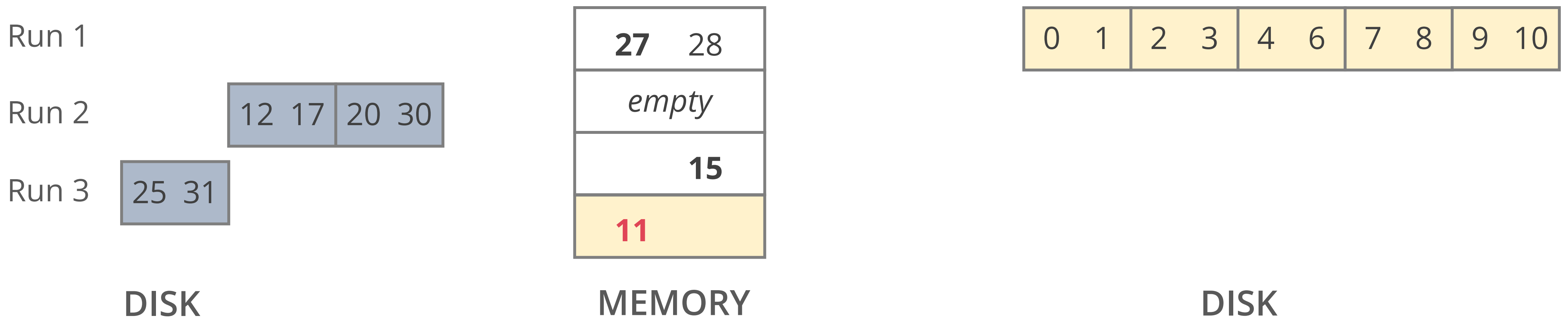


I/O Total (so far): 32

General External Merge Sort

Pass 1

Read 3 sorted runs into memory, write 1 sorted run to disk

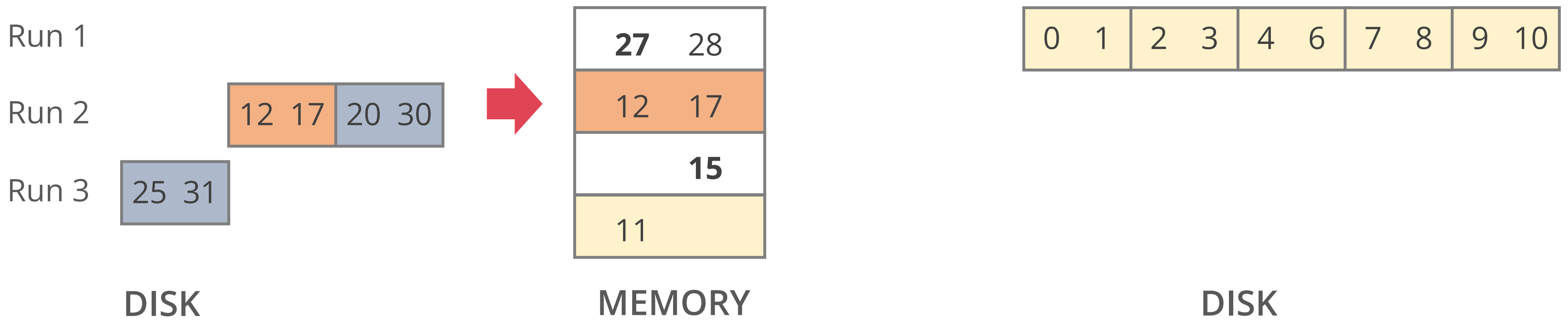


I/O Total (so far): 32

General External Merge Sort

Pass 1

Read 3 sorted runs into memory, write 1 sorted run to disk

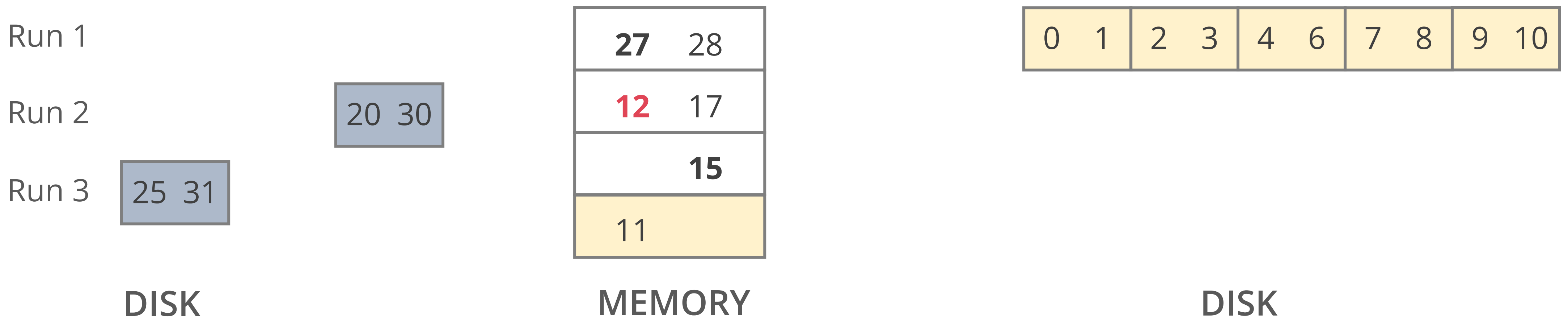


I/O Total (so far): 33

General External Merge Sort

Pass 1

Read 3 sorted runs into memory, write 1 sorted run to disk

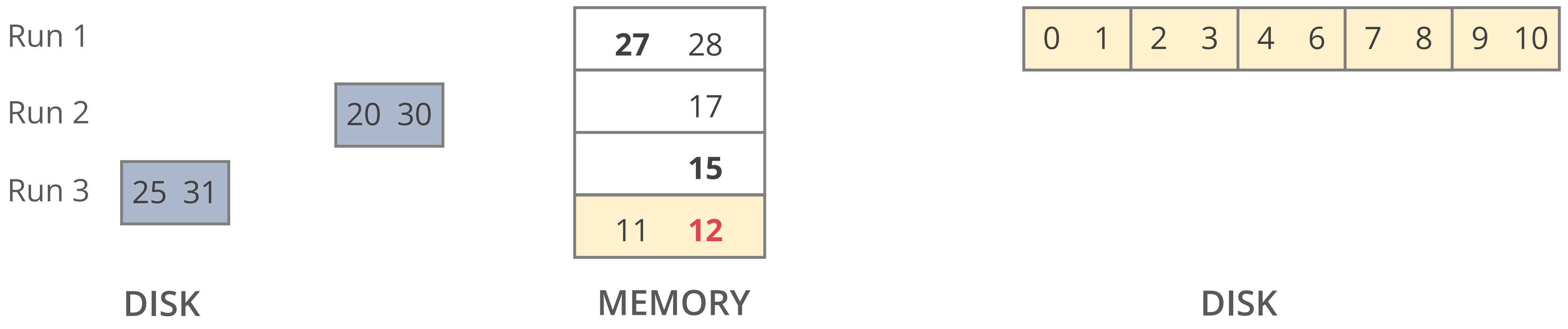


I/O Total (so far): 33

General External Merge Sort

Pass 1

Read 3 sorted runs into memory, write 1 sorted run to disk

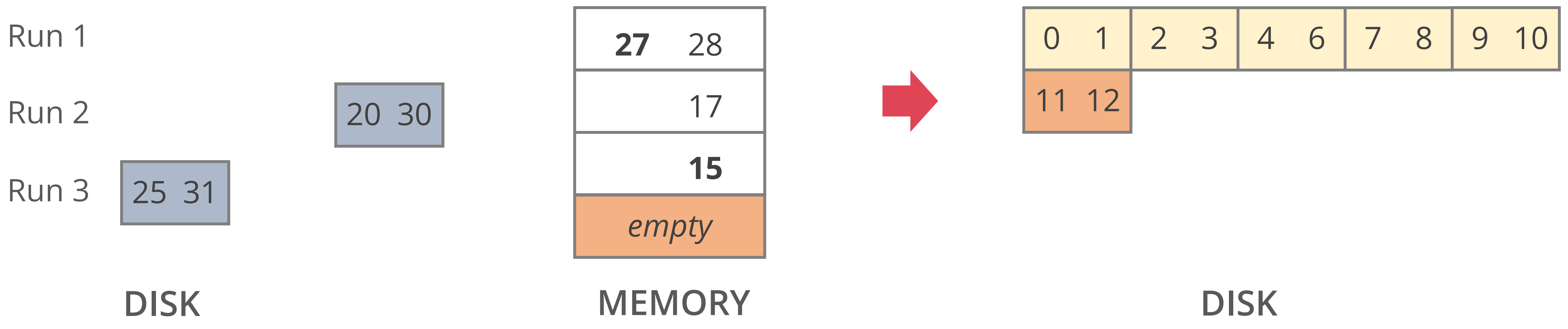


I/O Total (so far): 33

General External Merge Sort

Pass 1

Read 3 sorted runs into memory, write 1 sorted run to disk

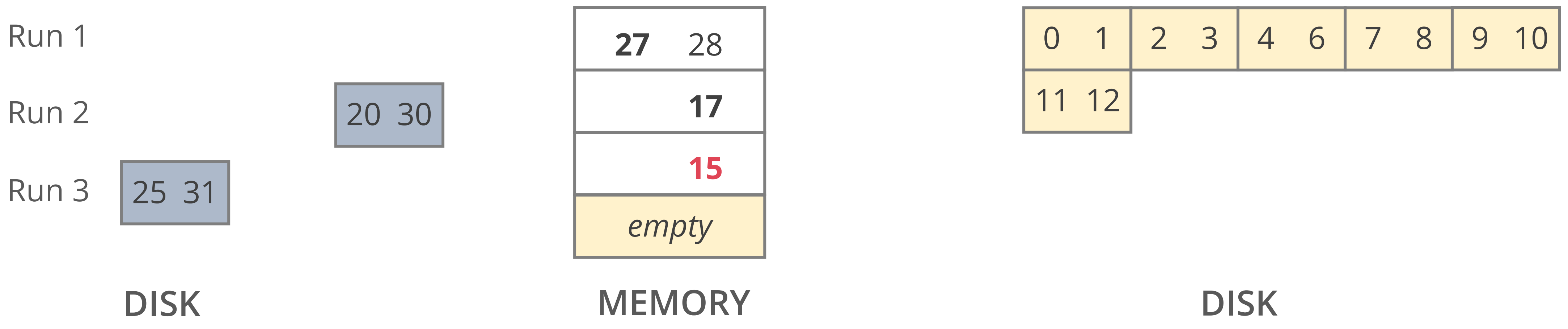


I/O Total (so far): 34

General External Merge Sort

Pass 1

Read 3 sorted runs into memory, write 1 sorted run to disk

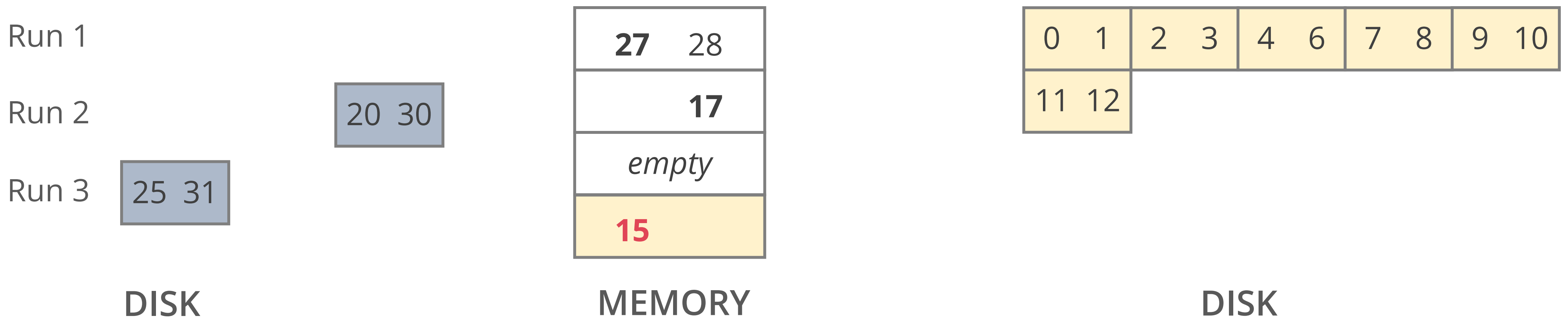


I/O Total (so far): 34

General External Merge Sort

Pass 1

Read 3 sorted runs into memory, write 1 sorted run to disk

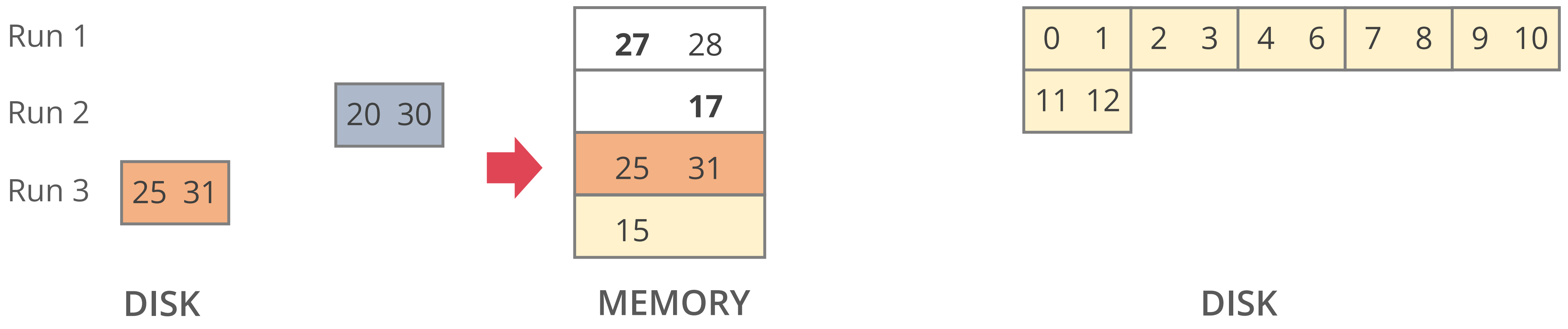


I/O Total (so far): 34

General External Merge Sort

Pass 1

Read 3 sorted runs into memory, write 1 sorted run to disk

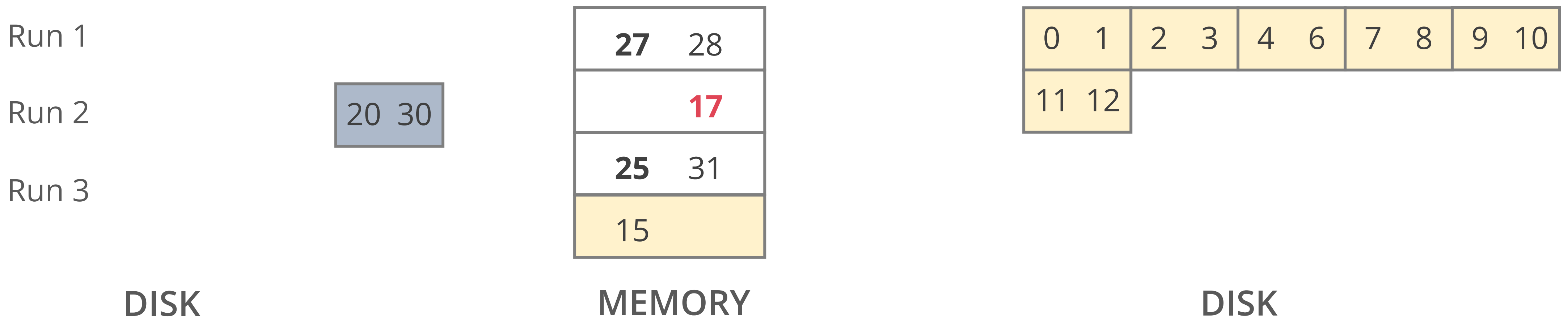


I/O Total (so far): 35

General External Merge Sort

Pass 1

Read 3 sorted runs into memory, write 1 sorted run to disk

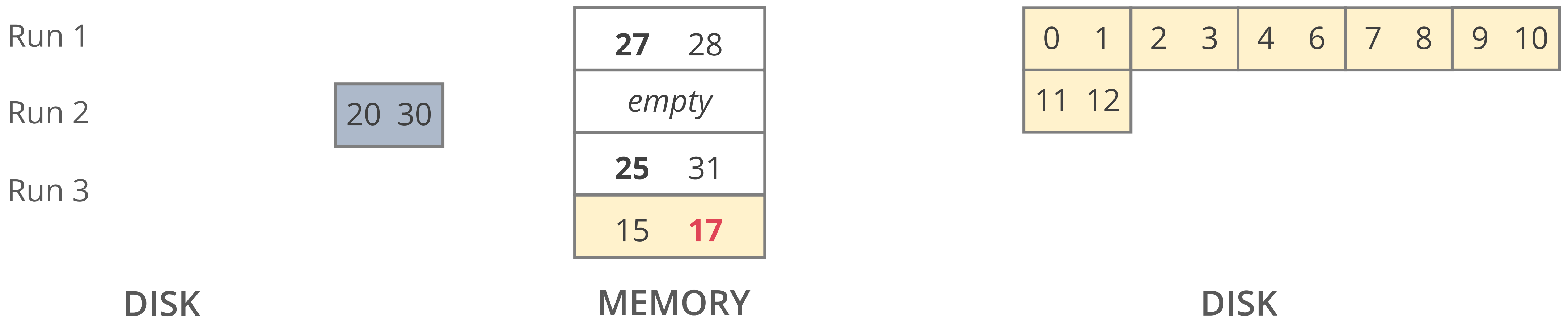


I/O Total (so far): 35

General External Merge Sort

Pass 1

Read 3 sorted runs into memory, write 1 sorted run to disk

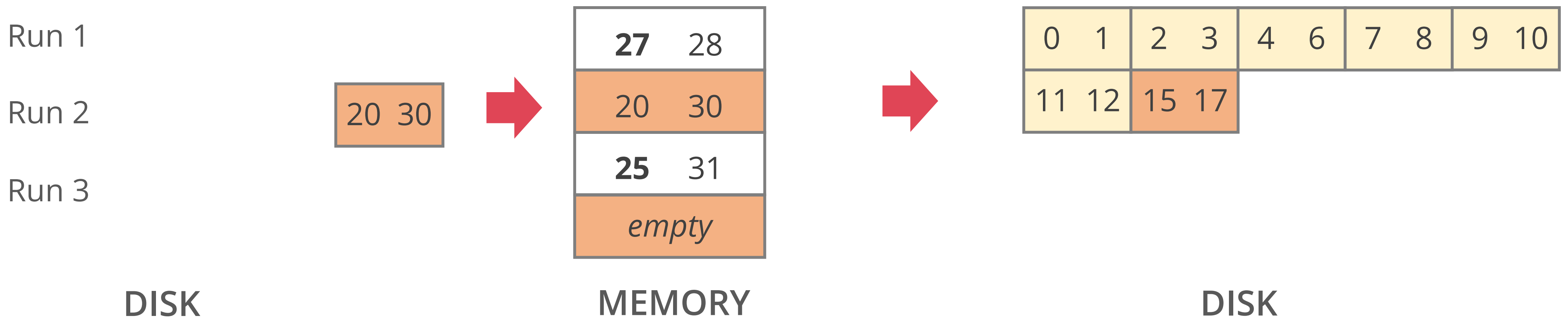


I/O Total (so far): 35

General External Merge Sort

Pass 1

Read 3 sorted runs into memory, write 1 sorted run to disk

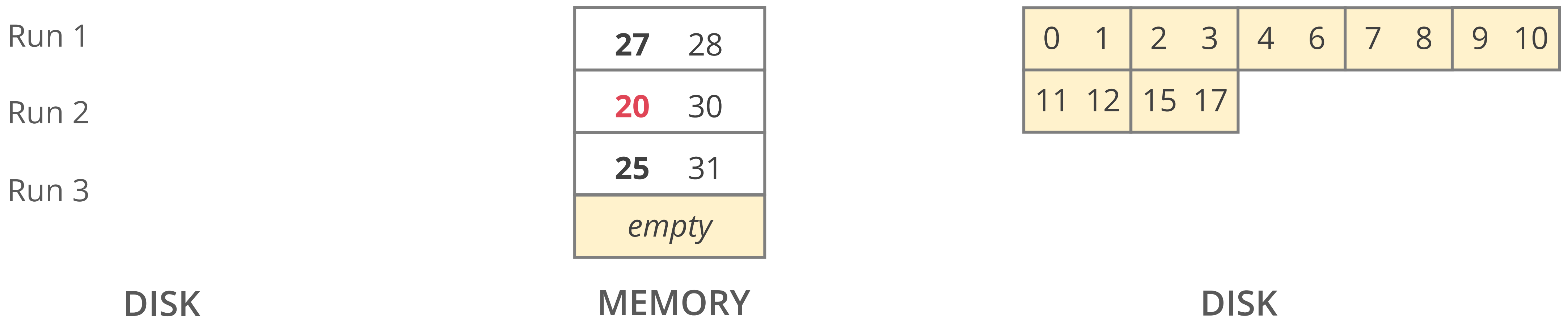


I/O Total (so far): 37

General External Merge Sort

Pass 1

Read 3 sorted runs into memory, write 1 sorted run to disk

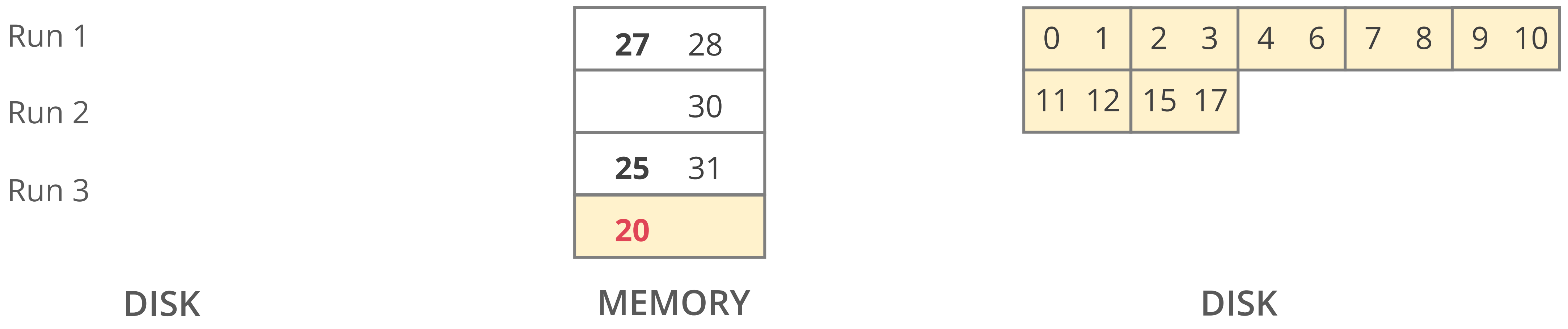


I/O Total (so far): 37

General External Merge Sort

Pass 1

Read 3 sorted runs into memory, write 1 sorted run to disk

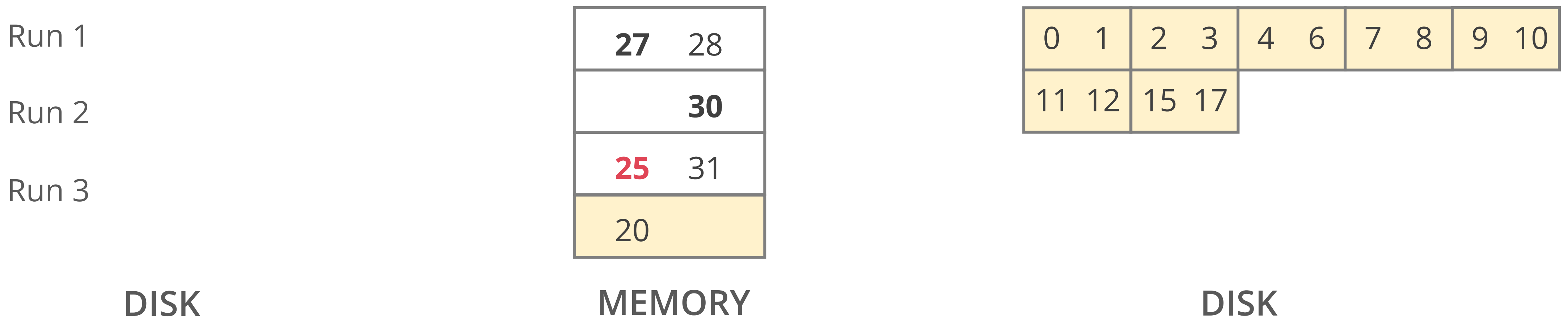


I/O Total (so far): 37

General External Merge Sort

Pass 1

Read 3 sorted runs into memory, write 1 sorted run to disk

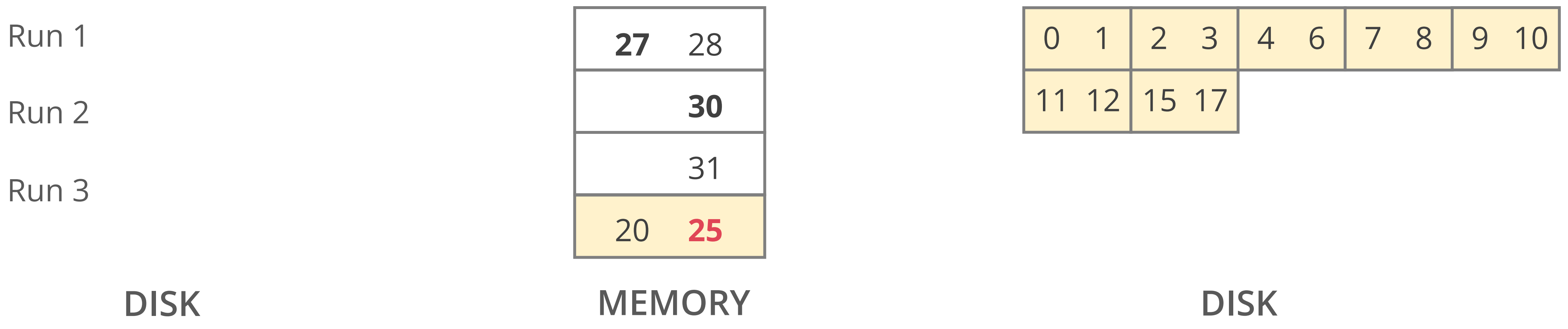


I/O Total (so far): 37

General External Merge Sort

Pass 1

Read 3 sorted runs into memory, write 1 sorted run to disk

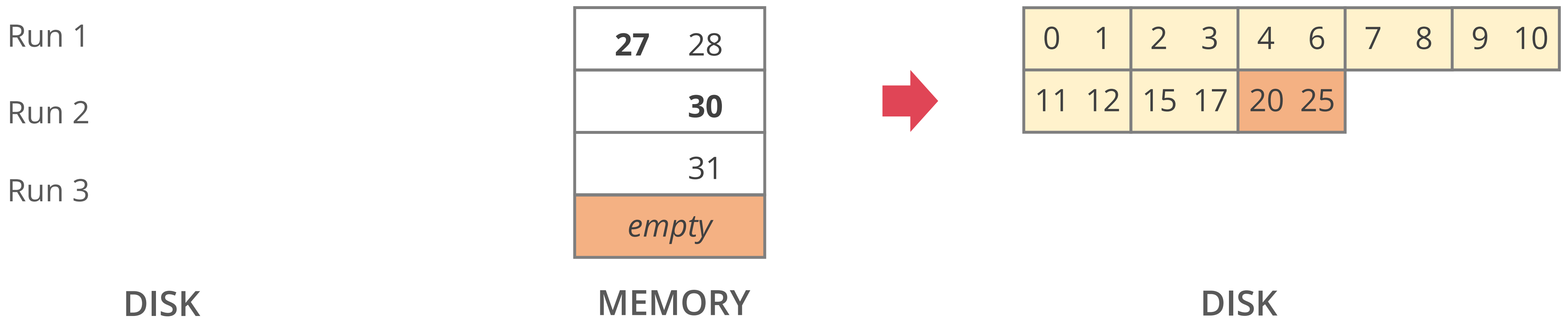


I/O Total (so far): 37

GENERAL EXTERNAL MERGE SORT

Pass 1

Read 3 sorted runs into memory, write 1 sorted run to disk

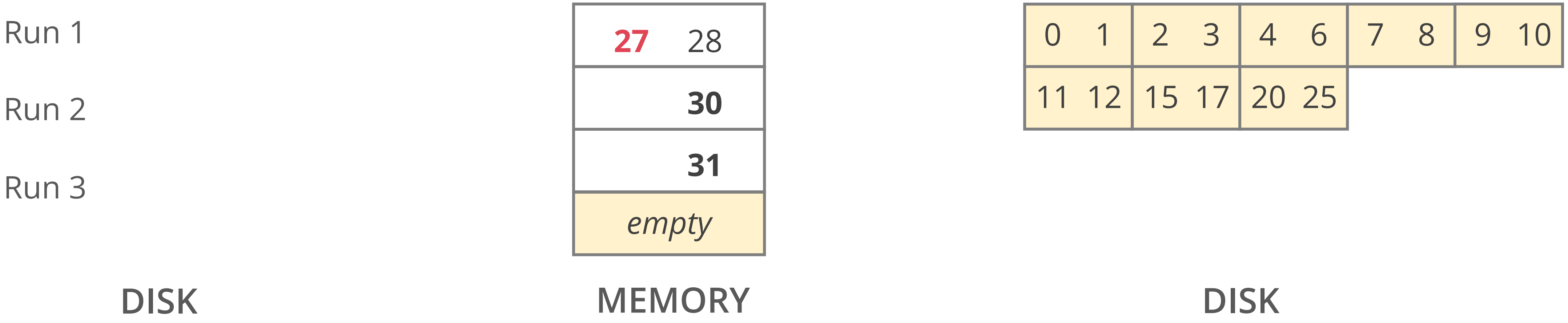


I/O Total (so far): 38

General External Merge Sort

Pass 1

Read 3 sorted runs into memory, write 1 sorted run to disk

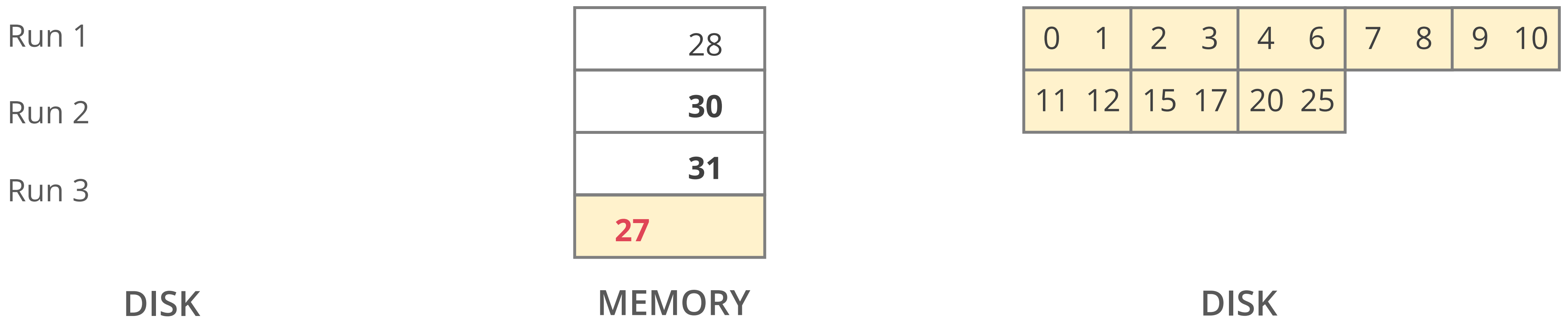


I/O Total (so far): 38

General External Merge Sort

Pass 1

Read 3 sorted runs into memory, write 1 sorted run to disk

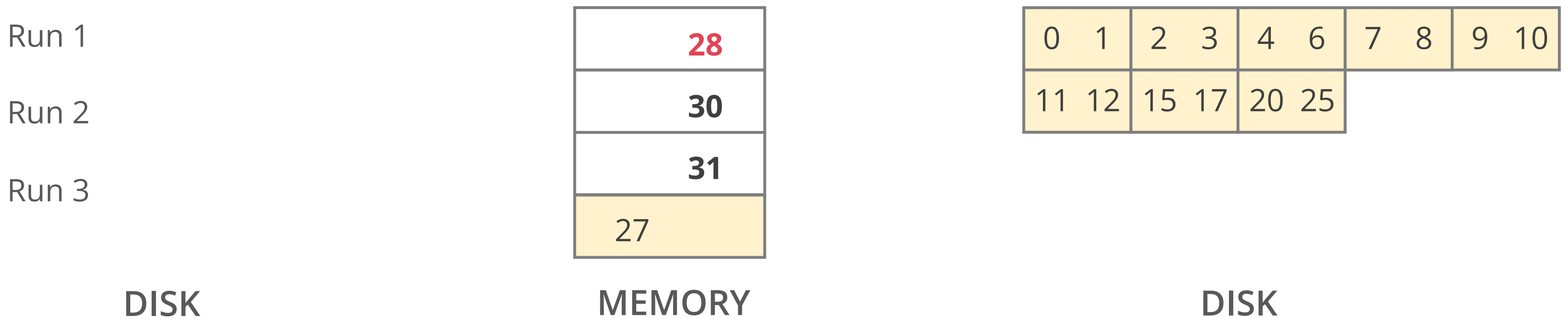


I/O Total (so far): 38

GENERAL EXTERNAL MERGE SORT

Pass 1

Read 3 sorted runs into memory, write 1 sorted run to disk

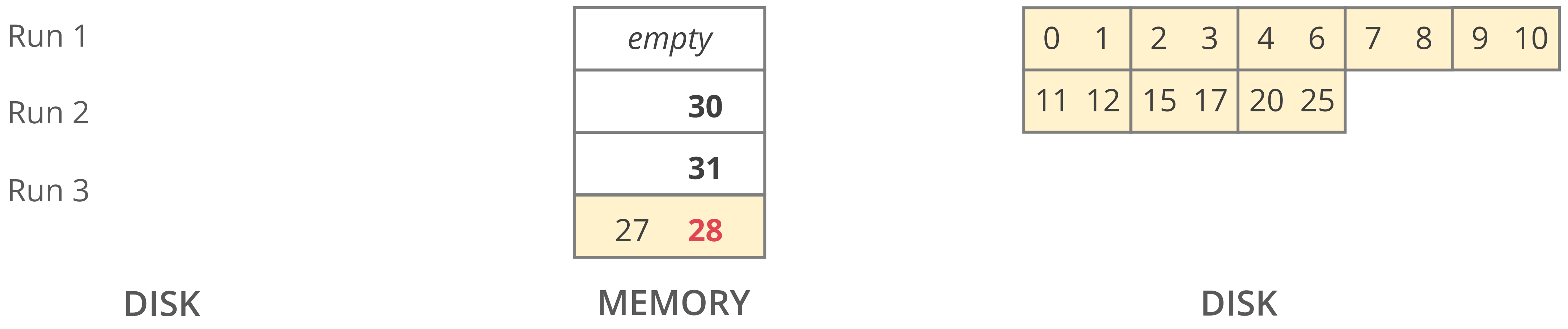


I/O Total (so far): 38

General External Merge Sort

Pass 1

Read 3 sorted runs into memory, write 1 sorted run to disk

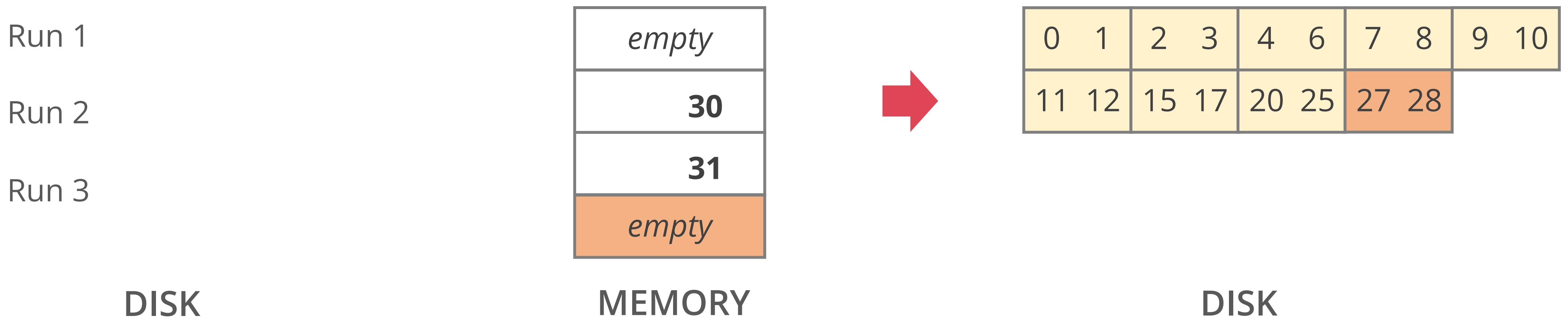


I/O Total (so far): 38

General External Merge Sort

Pass 1

Read 3 sorted runs into memory, write 1 sorted run to disk

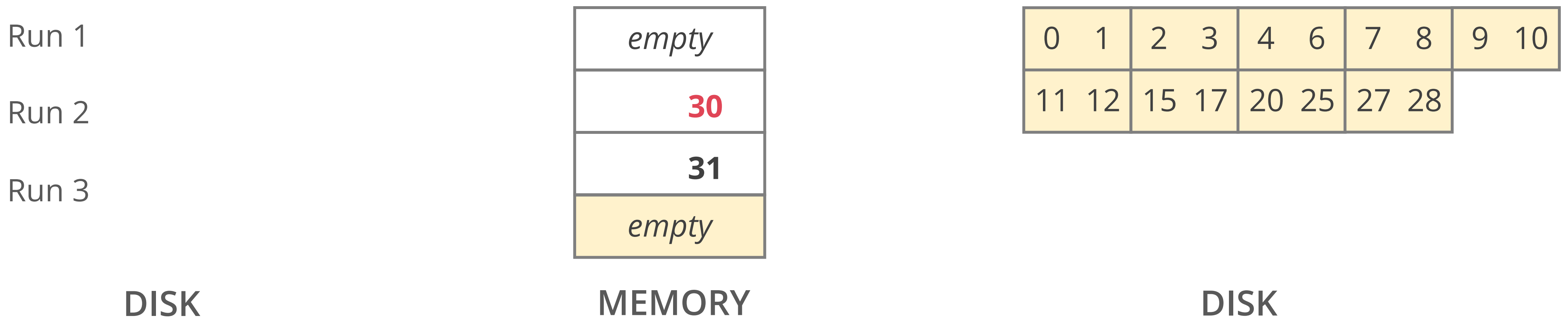


I/O Total (so far): 39

General External Merge Sort

Pass 1

Read 3 sorted runs into memory, write 1 sorted run to disk

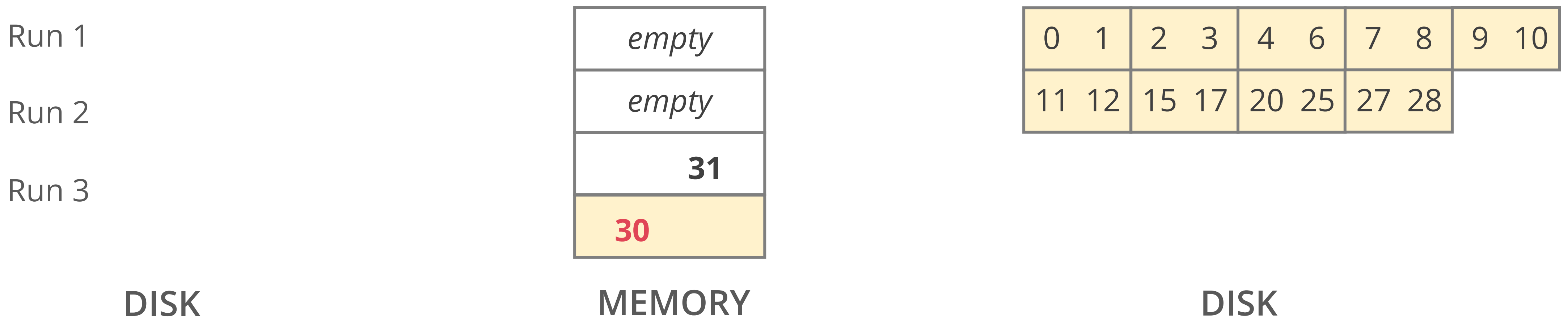


I/O Total (so far): 39

General External Merge Sort

Pass 1

Read 3 sorted runs into memory, write 1 sorted run to disk

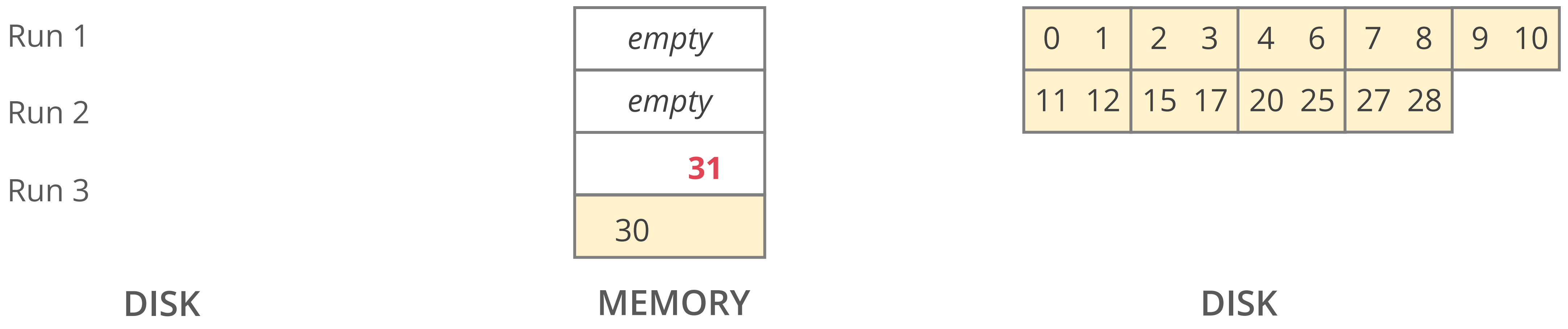


I/O Total (so far): 39

General External Merge Sort

Pass 1

Read 3 sorted runs into memory, write 1 sorted run to disk

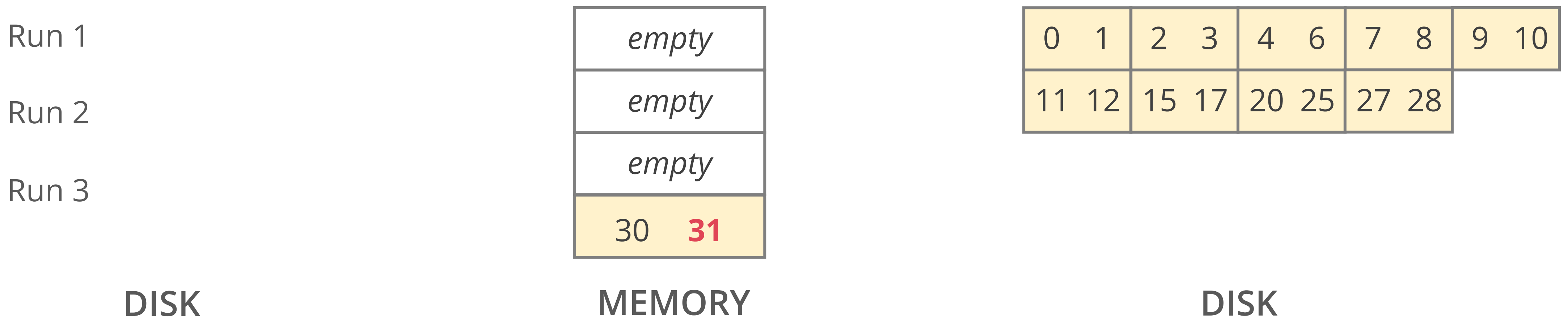


I/O Total (so far): 39

General External Merge Sort

Pass 1

Read 3 sorted runs into memory, write 1 sorted run to disk

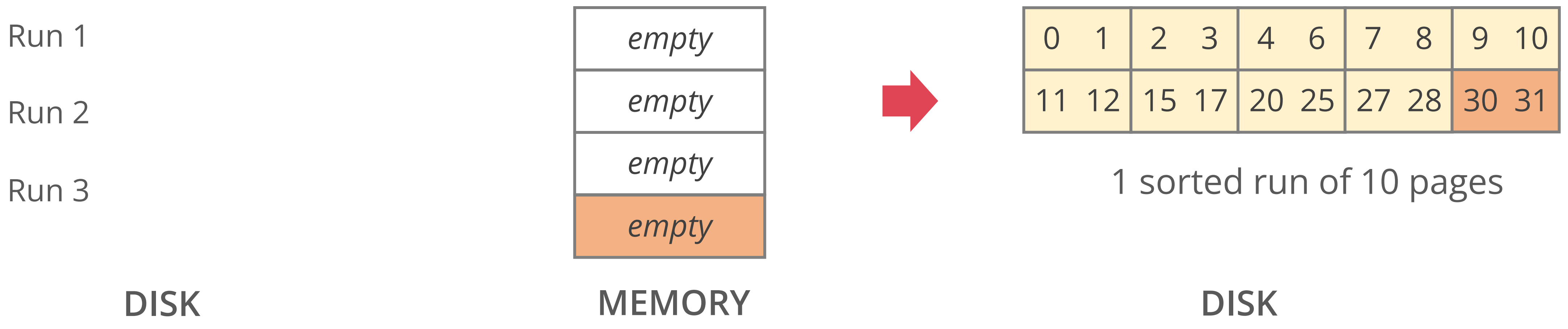


I/O Total (so far): 39

General External Merge Sort

Pass 1

Read 3 sorted runs into memory, write 1 sorted run to disk



I/O Total: 40

Sanity Check

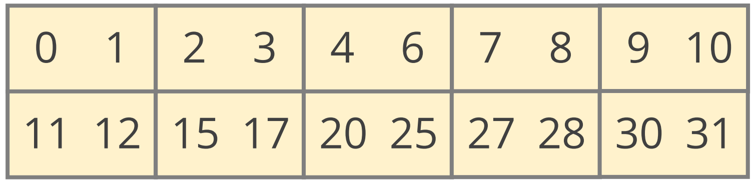
$N = 10, B = 4$

$\text{Cost} = 2N * (1 + \lceil \log_{B-1}(\lceil N/B \rceil) \rceil)$

$= 2 * 10 * (1 + \lceil \log_3(2.5) \rceil)$

$= 20 * (1 + 1)$

$= 40$ I/Os ✓



0 1	2 3	4 6	7 8	9 10
11 12	15 17	20 25	27 28	30 31

1 sorted run of 10 pages

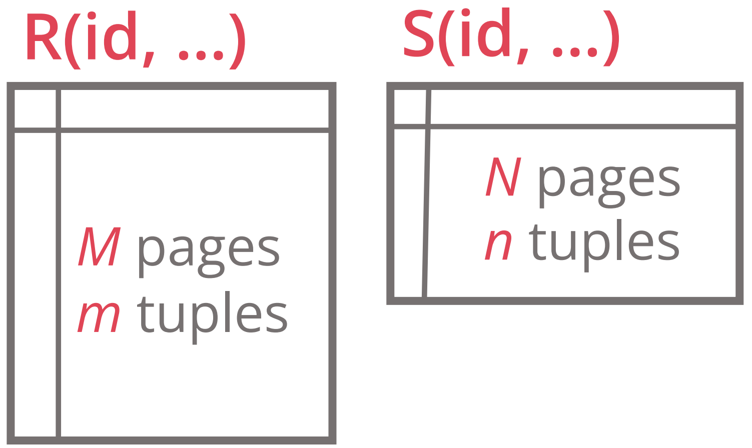
I/O Total: 40

Joins

Joins

Bit of notation:

- *M* = number of pages in ***R***
- *m* = number of records in ***R*** (the **cardinality** of ***R***)
- *N* = number of pages in ***S***
- *n* = number of records in ***S***



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!

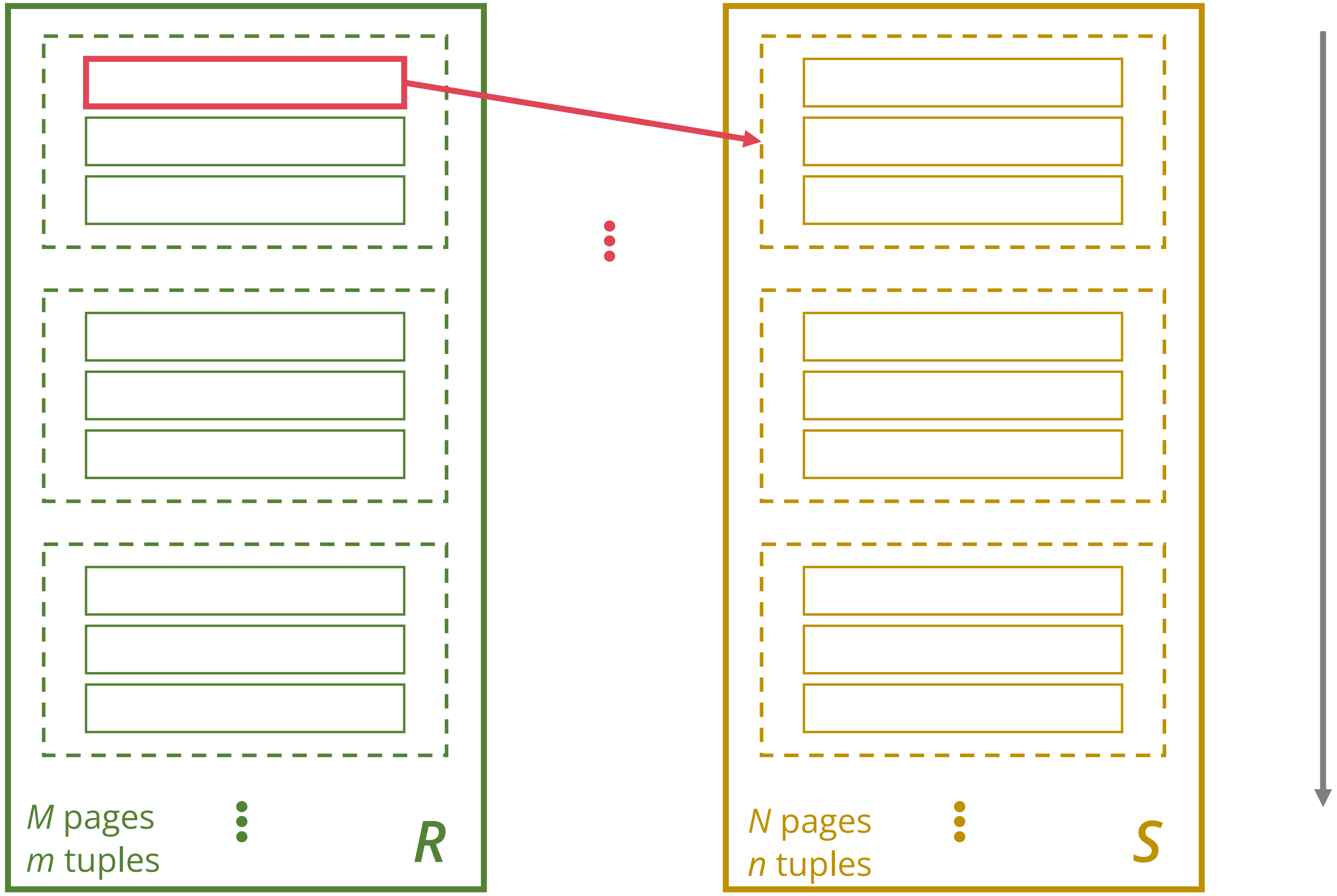
Simple Nested Loops Join (SNLJ)

Direct translation of the definition of join into code

To perform $R \bowtie_{\theta} S$, take each tuple in R and scan through S to find the matching S-tuples!

```
foreach tuple r ∈ R:
  foreach tuple s ∈ S:
    if $\theta$(r,s): output r joined with s
```

Simple Nested Loops Join (SNLJ)

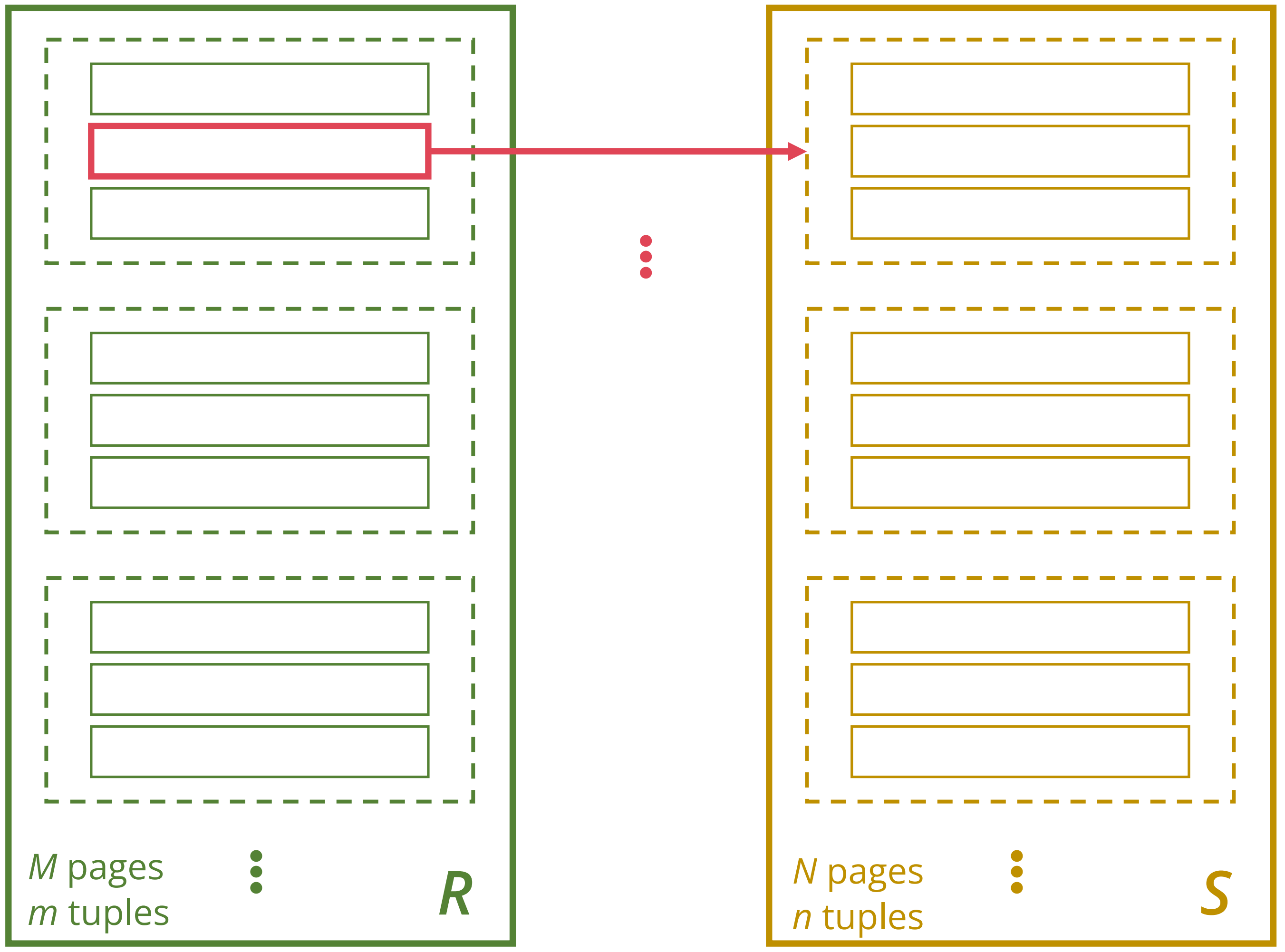


First iteration of outer loop…

Compare with all tuples in ***S***

… and add matches to result

Simple Nested Loops Join (SNLJ)

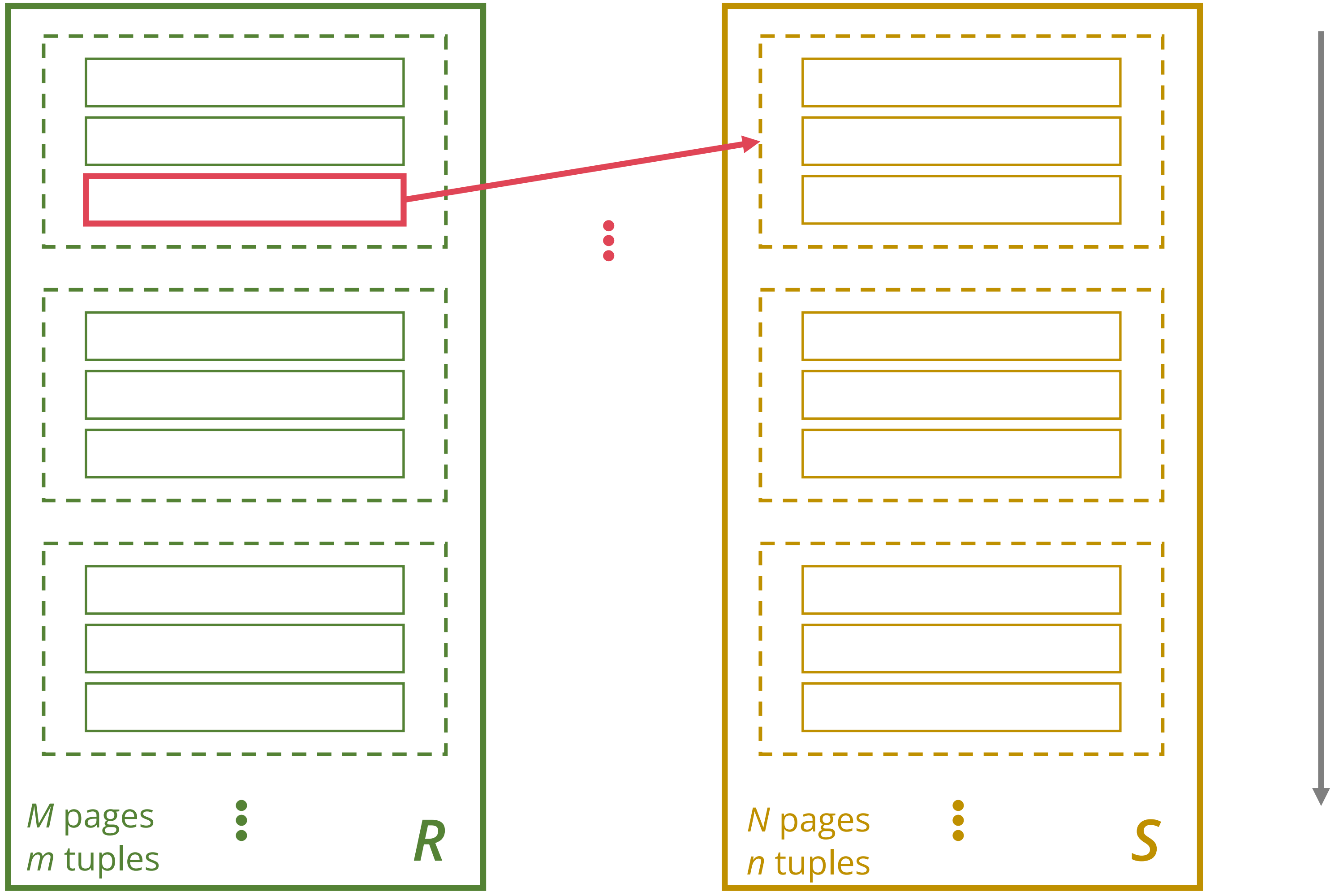


Second iteration of outer loop…

Compare with all tuples in ***S***

… and add matches to result

Simple Nested Loops Join (SNLJ)



Cost = $M + m \cdot N$

Flipping the order of R and S can change the I/O cost

Which relation to use as outer?

Page Nested Loops Join (PNLJ)

Can we do better?

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

Page Nested Loops Join (PNLJ)

SNLJ

```
foreach tuple r ∈ R:
  foreach tuple s ∈ S:
    if $\theta$(r,s): output r joined with s
```

Page Nested Loops Join (PNLJ)

SNLJ (but with page fetches written out explicitly)

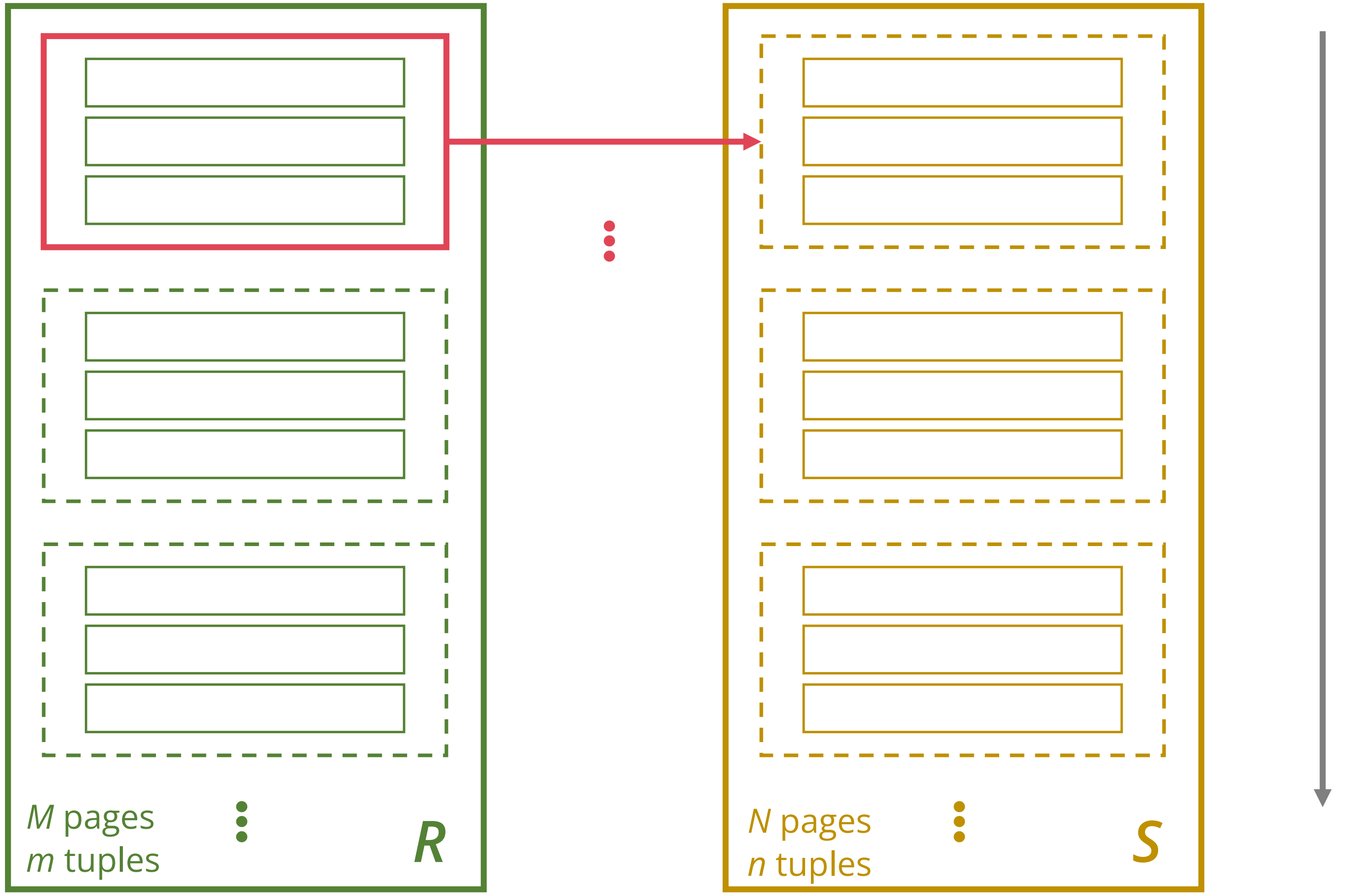
```
foreach page $P_R \in R$:
  foreach tuple $r \in P_R$:
    foreach page $P_S \in S$:
      foreach tuple $s \in P_S$:
        if $\theta(r,s)$: output r joined with s
```

Page Nested Loops Join (PNLJ)

PNLJ

```
foreach page $P_R$ ∈ R:
  foreach page $P_S$ ∈ S:                    flipped loops
    foreach tuple r ∈ $P_R$:
      foreach tuple s ∈ $P_S$:
        if $\theta$(r,s): output r joined with s
```

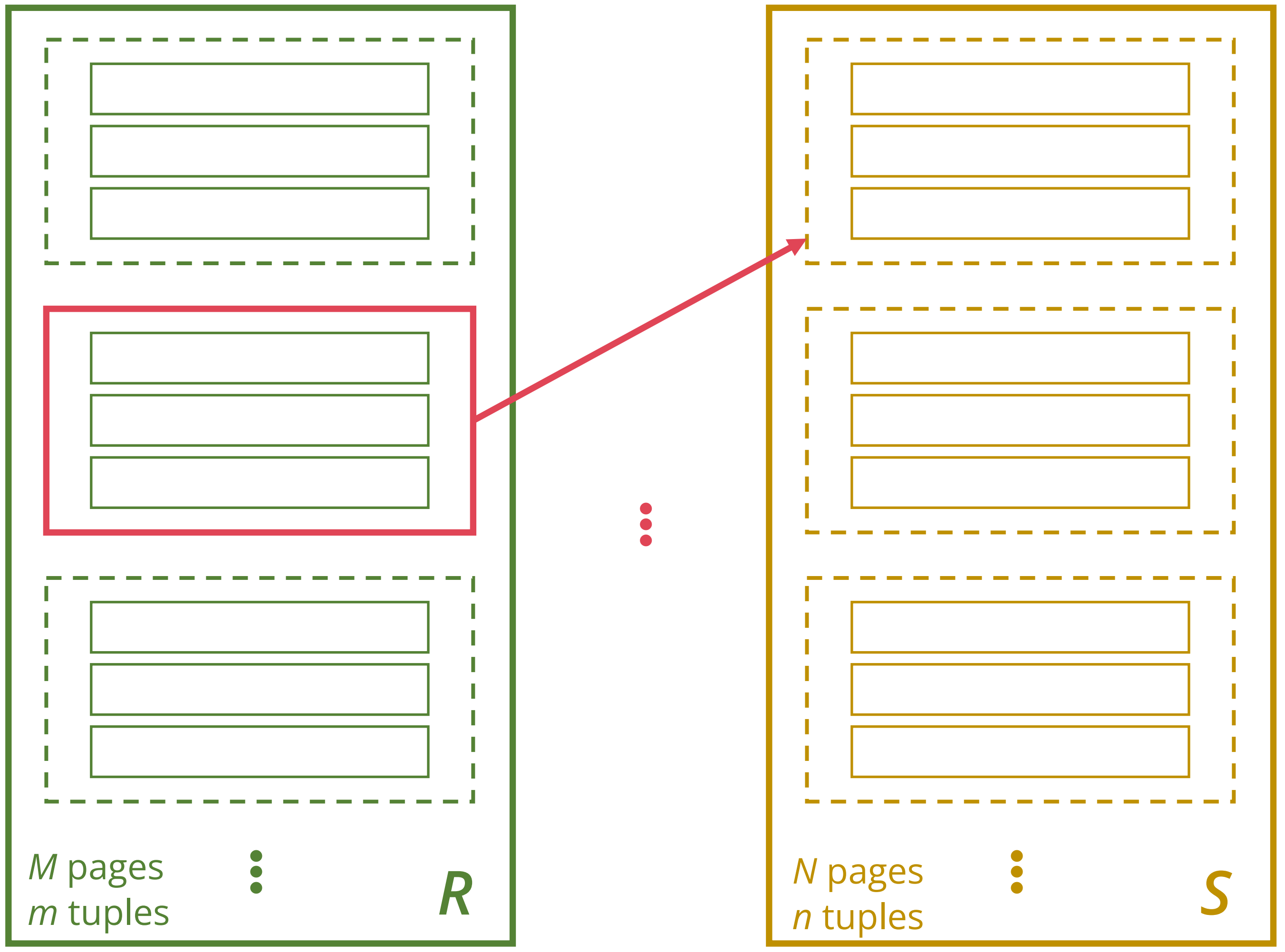
Page Nested Loops Join (PNLJ)



First iteration of outer loop…

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

Page Nested Loops Join (PNLJ)

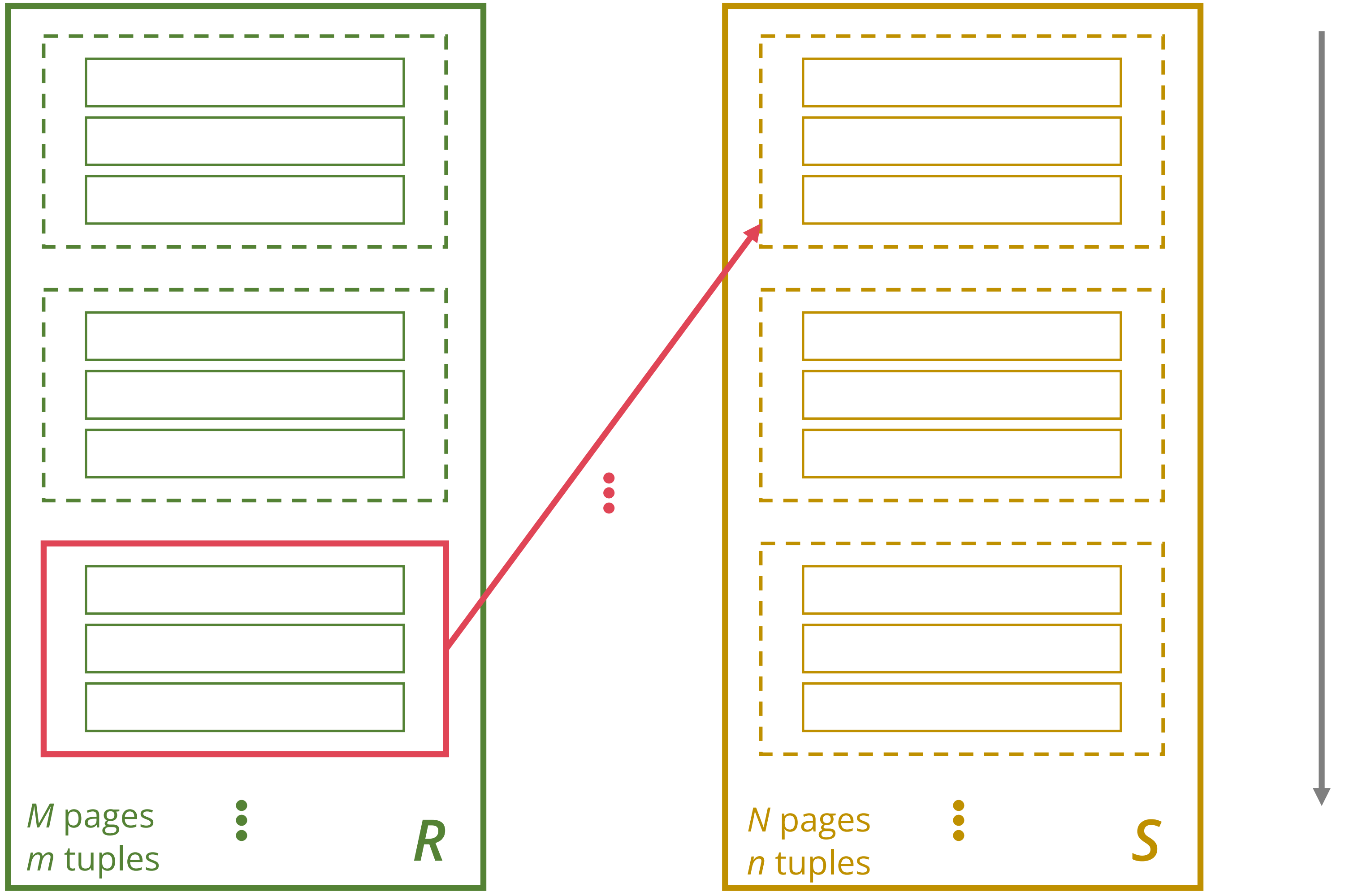


Second iteration of outer loop…

Compare with all tuples in ***S***

… and add matches to result

Page Nested Loops Join (PNLJ)



Cost = $M + M \cdot N$

Block Nested Loops 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 Loops Join (PNLJ)

PNLJ

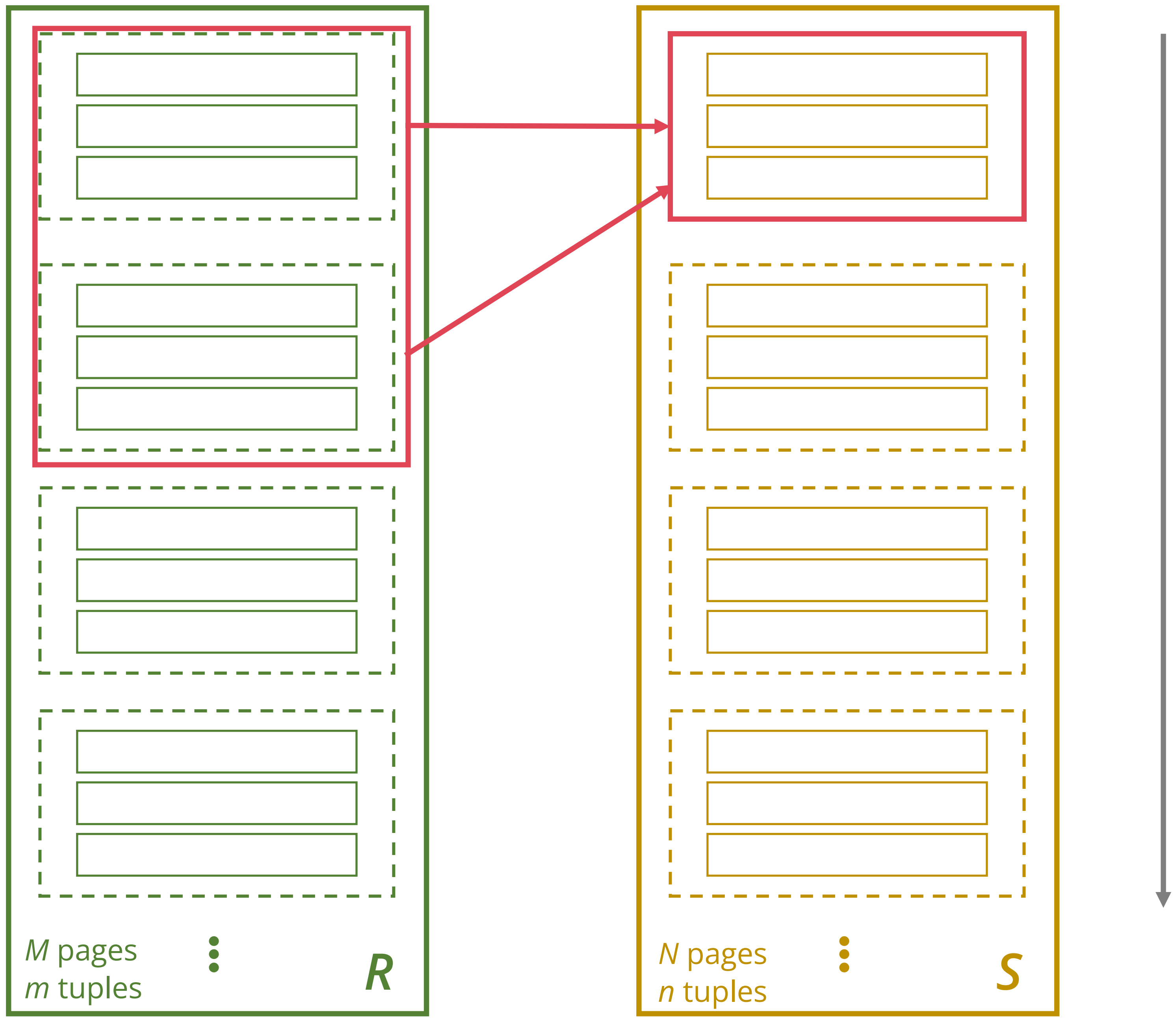
```
foreach page $P_R$ ∈ R:
  foreach page $P_S$ ∈ S:
    foreach tuple r ∈ $P_R$:
      foreach tuple s ∈ $P_S$:
        if $\theta$(r,s): output r joined with s
```

Block Nested Loops Join (PNLJ)

BNLJ

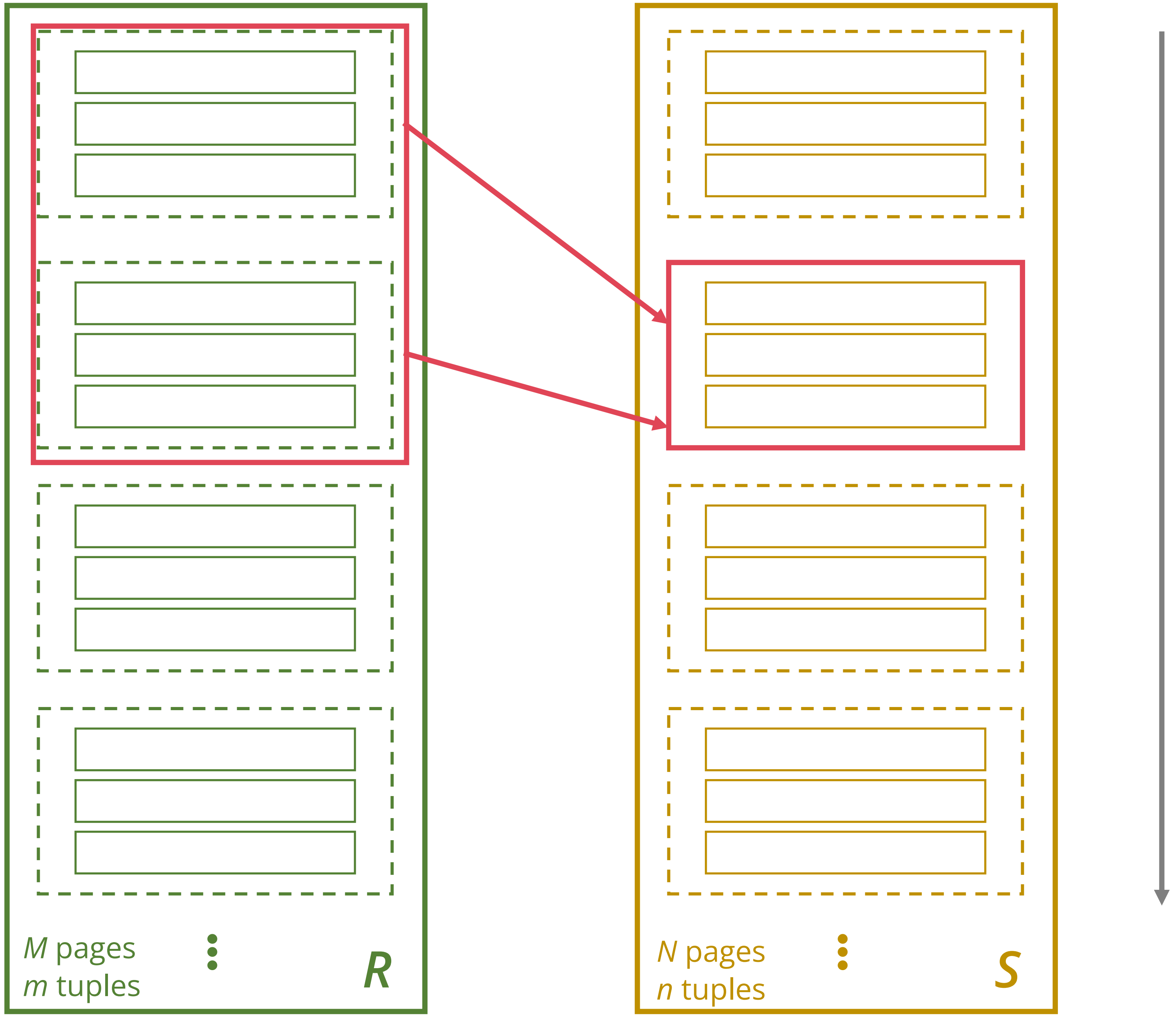
```
foreach block $C_R$ of B-2 pages ∈ R:
  foreach page $P_S$ ∈ S:
    foreach tuple r ∈ $C_R$:
      foreach tuple s ∈ $P_S$:
        if $\theta$(r,s): output r joined with s
```

B = *4*



First iteration of outer loop…

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



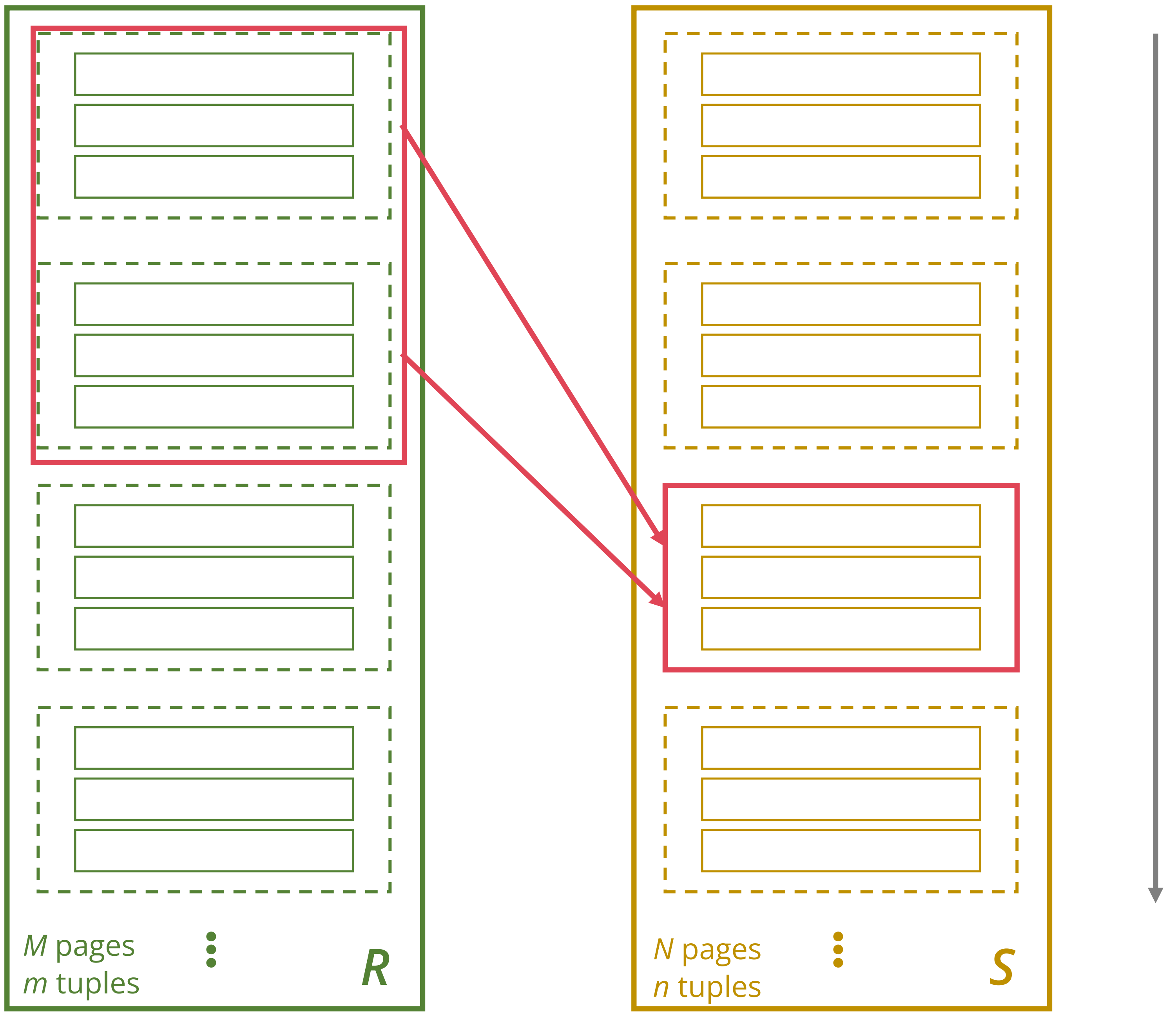
B* = *4

First iteration of outer loop…

Compare with all tuples in ***S***

… and add matches to result

B = *4*

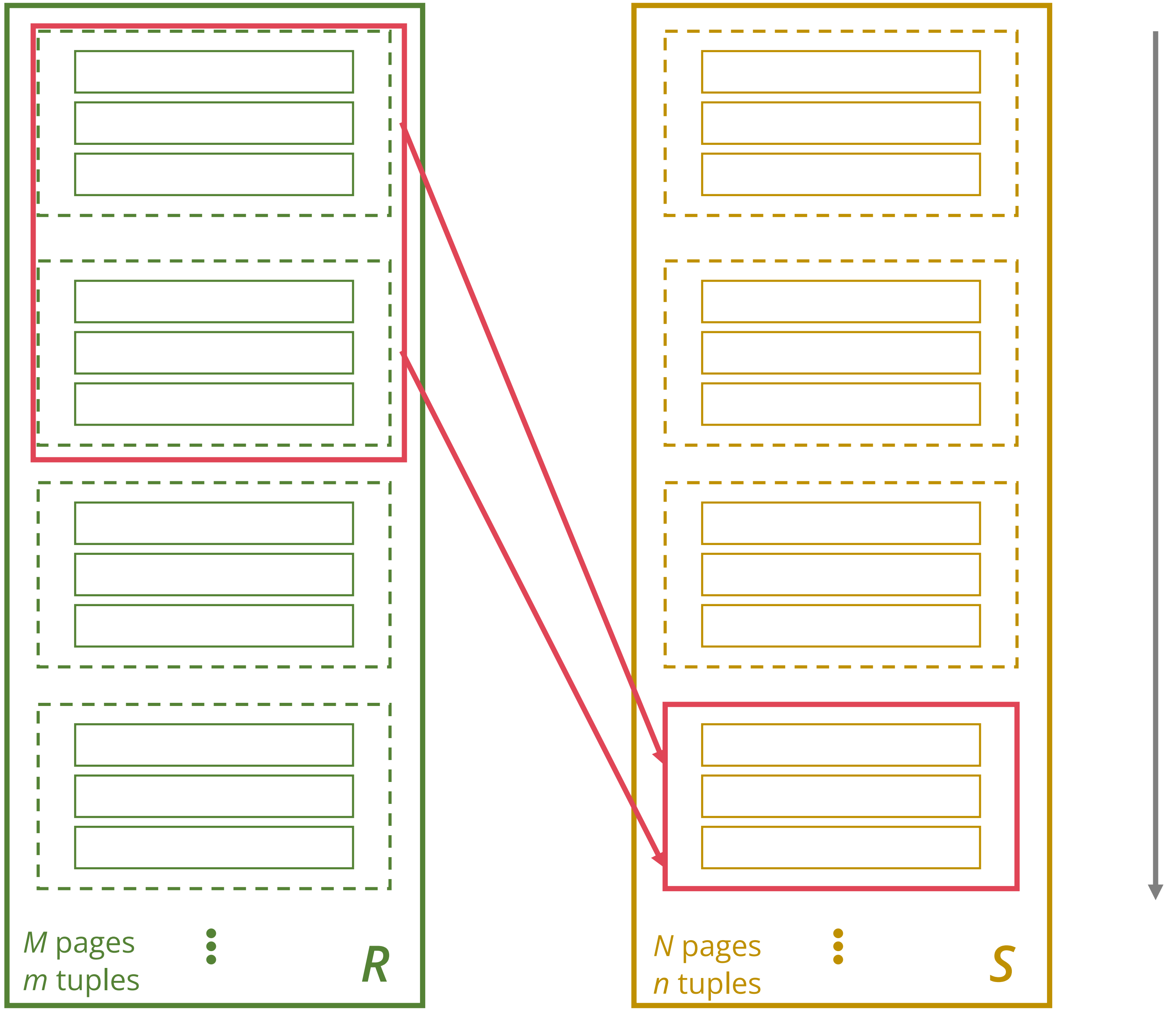


First iteration of outer loop…

Compare with all tuples in *S*

… and add matches to result

B = 4

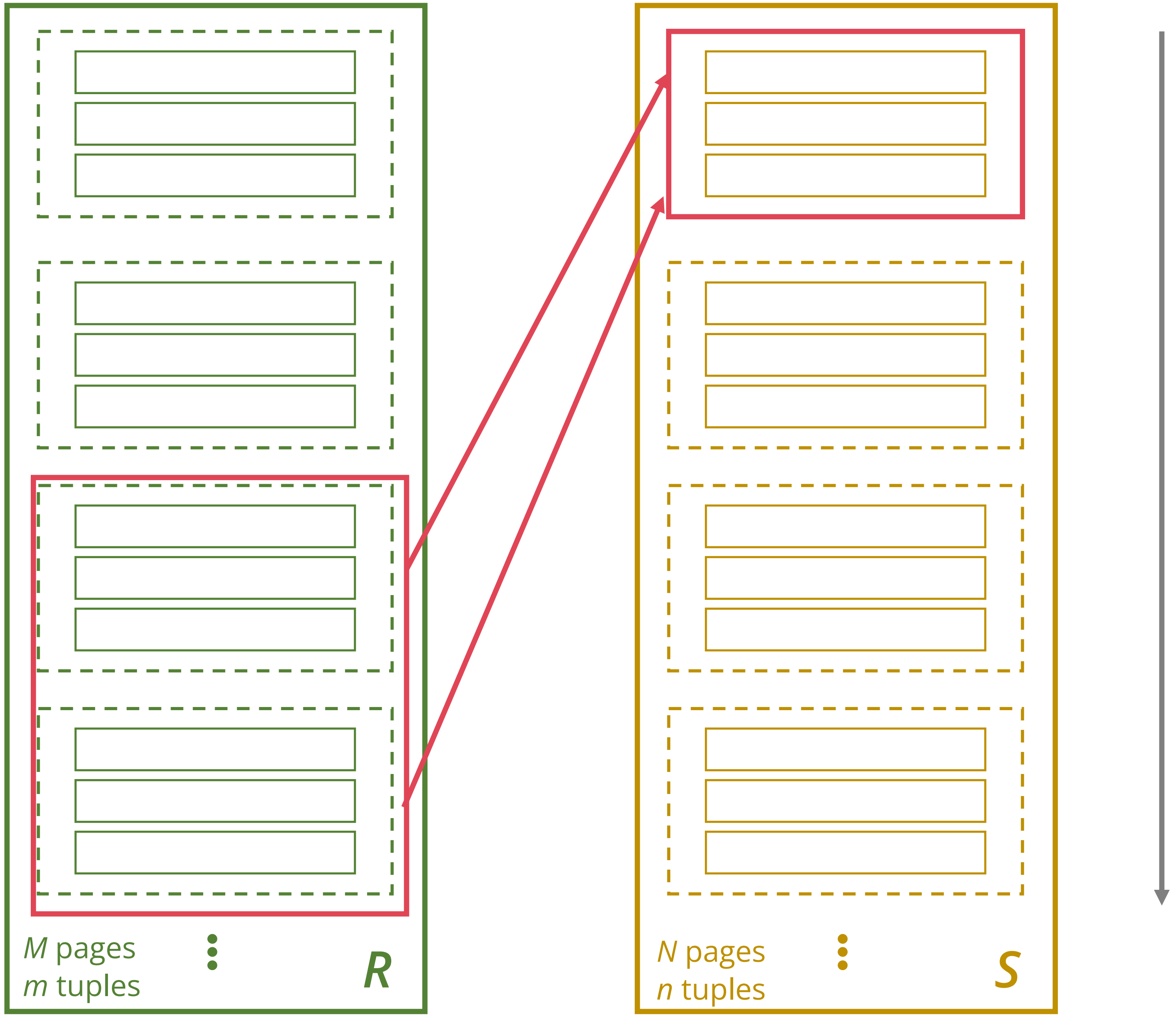


First iteration of outer loop…

Compare with all tuples in *S*

… and add matches to result

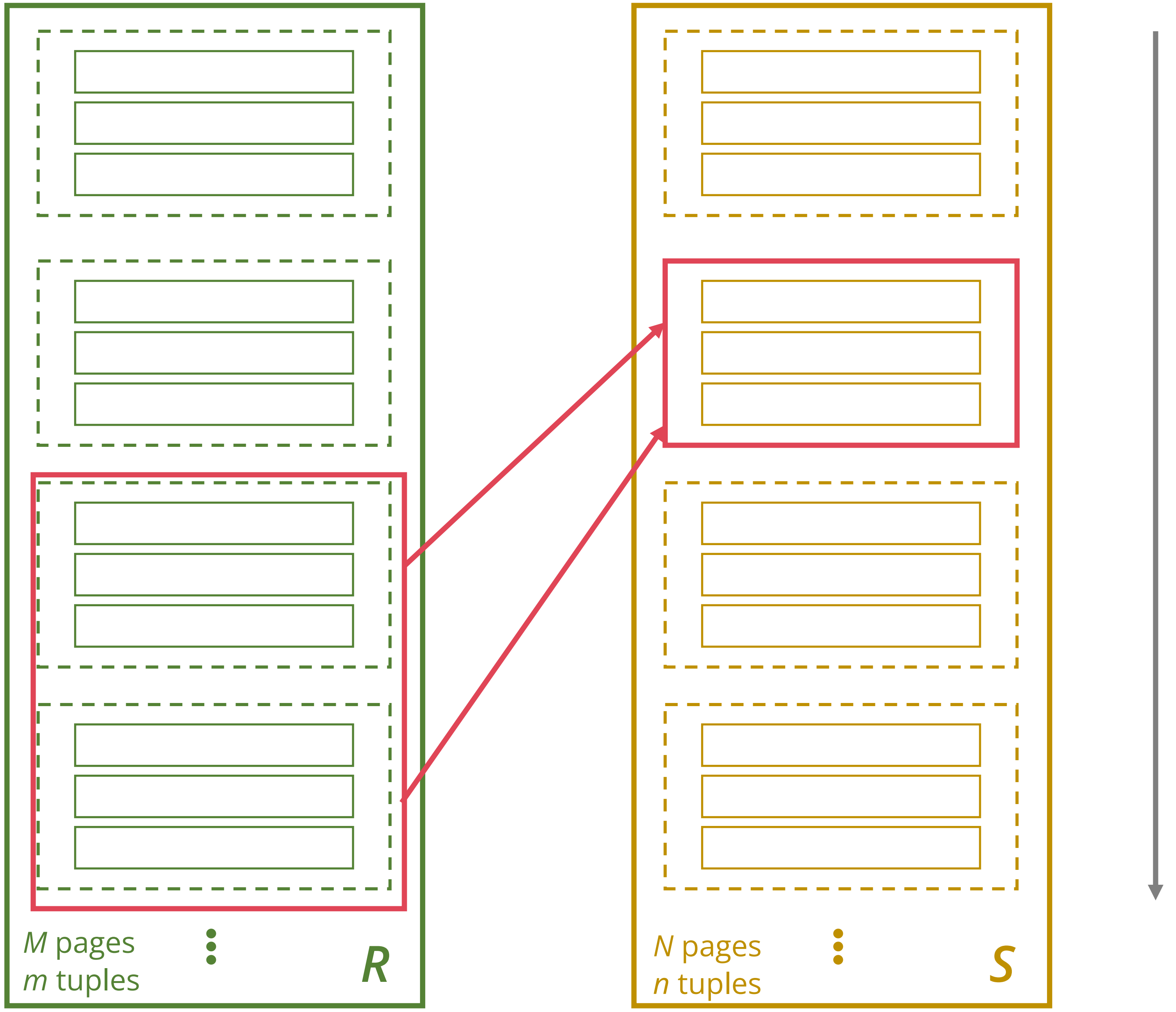
B = *4*



Second iteration of outer loop…

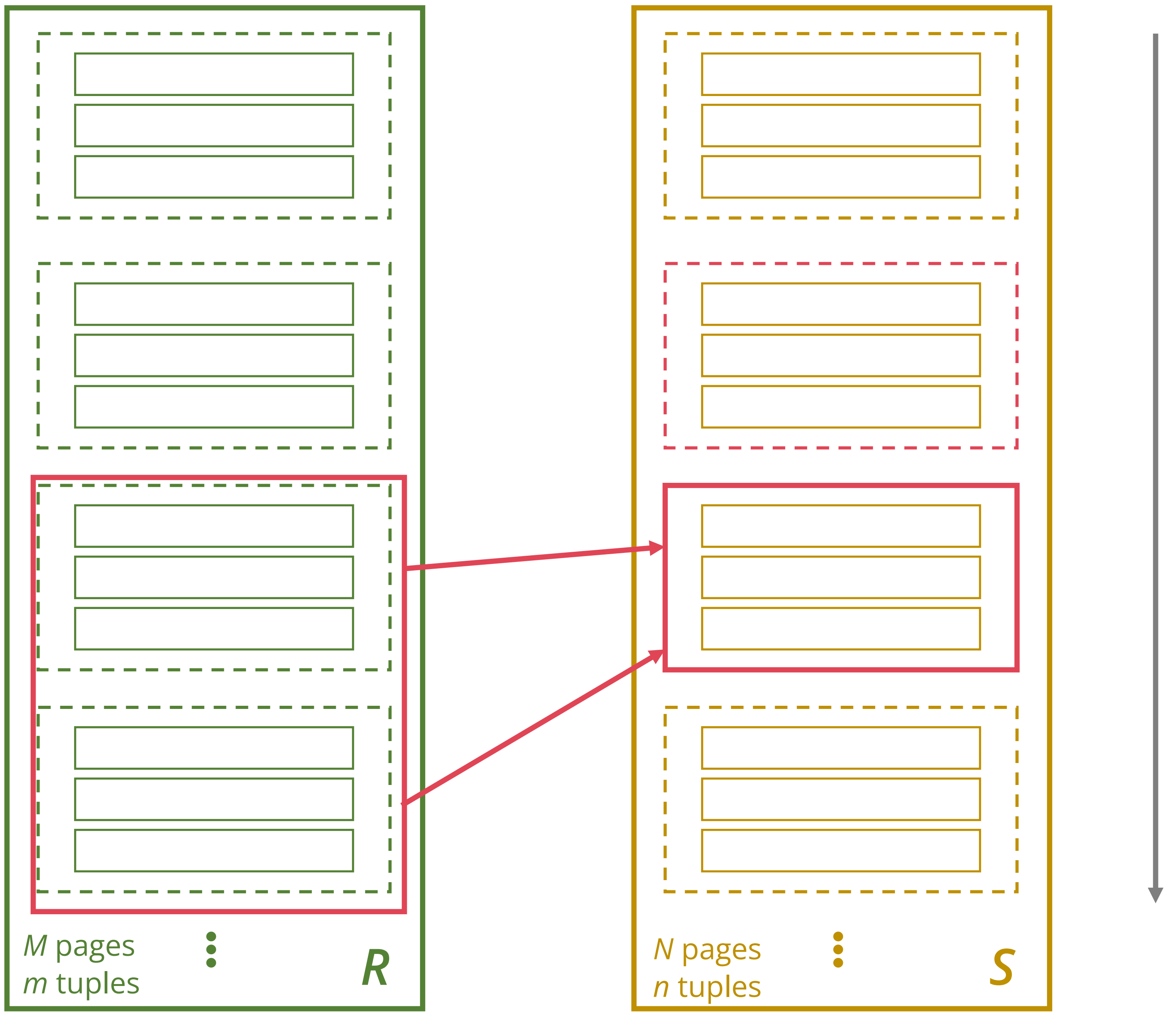
Compare with all tuples in *S*
… and add matches to result

B = *4*



Second iteration of outer loop…

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



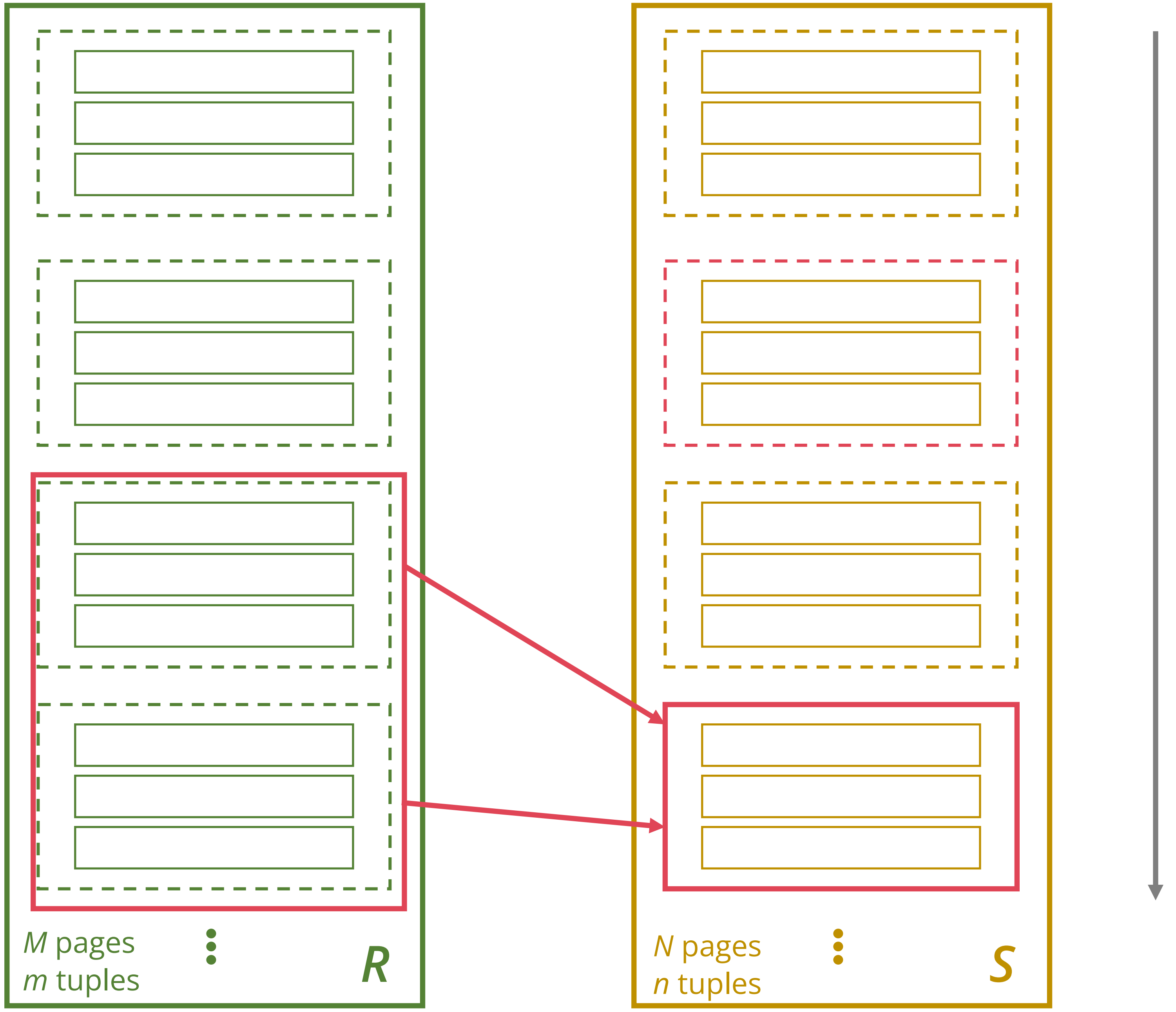
B = 4

Second iteration of outer loop…

Compare with all tuples in *S*

… and add matches to result

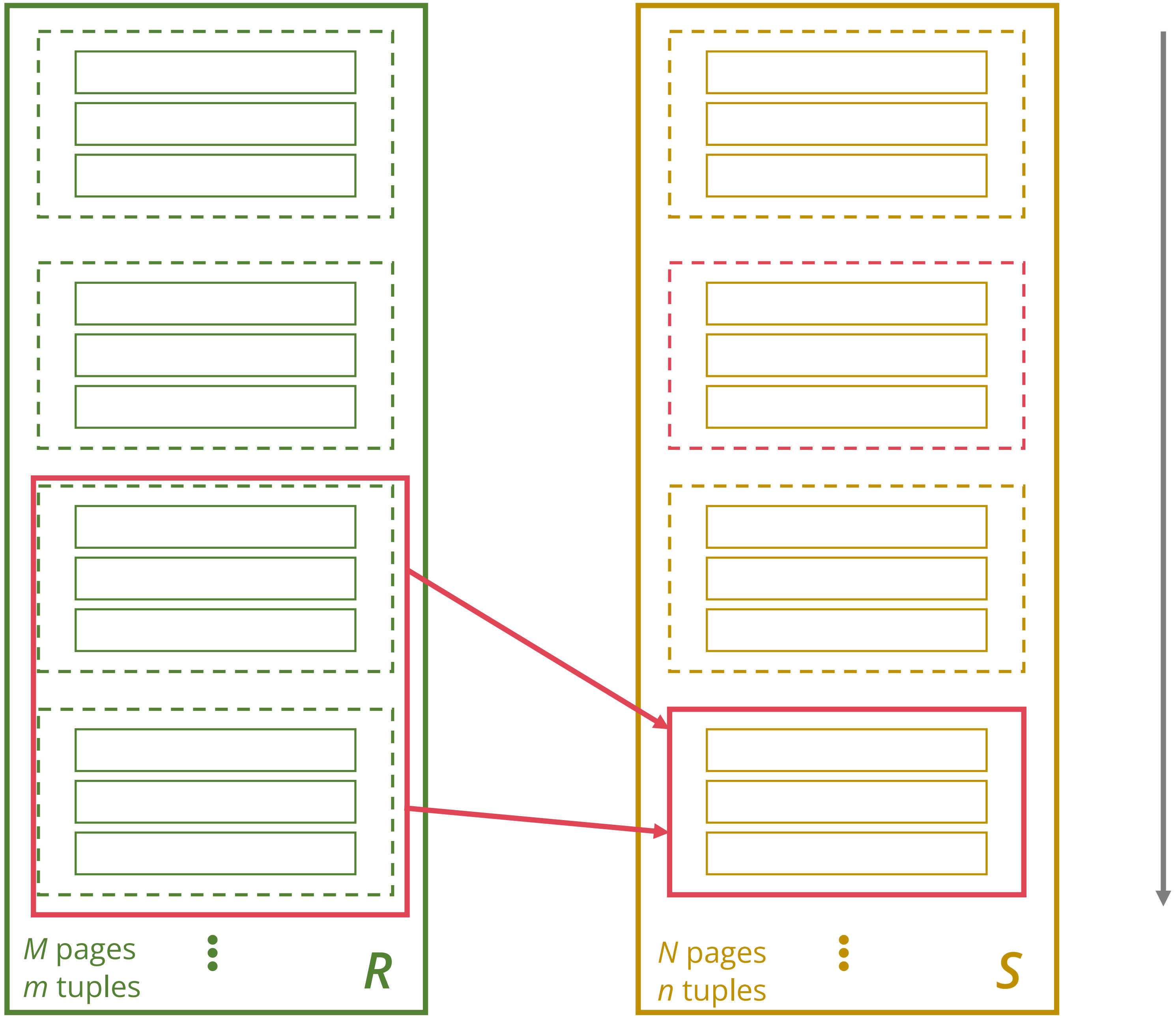
B = *4*



Second iteration of outer loop…

Compare with all tuples in *S*

… and add matches to result



$B = 4$

Cost

$= M + (\#\ \text{blocks in}\ R) \cdot N$

$= M + \lceil M / \text{chunk size} \rceil \cdot N$

$= M + \lceil M / B - 2 \rceil \cdot N$

Index Nested Loops Join (INLJ)

A join is essentially

```
foreach tuple r ∈ R:
  foreach tuple s ∈ S that satisfies $\theta$(r,s):
    output r joined with s
```

Index Nested Loops Join (INLJ)

An **index** on *S* allows us to do the inner loop efficiently!

```
foreach tuple r ∈ R:
  foreach tuple s ∈ S that satisfies θ(r,s):
  (found using the index)
    output r joined with s
```

Index Nested Loops Join (INLJ)

Cost = *M* + *m* * cost to find matching *S* tuples

- *M* from scanning through *R*

Cost to find matching *S* tuples via **tree index**

- Variant A: cost to traverse root to leaf + read all the leaves with matching tuples
- Variants B or C: cost of retrieving RIDs (similar to Variant A) + cost to fetch actual records
  - 1 I/O per **page** if clustered, 1 I/O per **matching tuple** if not

Index Nested Loops Join (INLJ)

Cost = *M* + *m* * cost to find matching *S* tuples

M from scanning through *R*

Cost to find matching *S* tuples via **hash index**

1-2 I/Os to reach the target bucket

Then scan pages in that bucket

May stop early if search key values are unique and we found a match!

Index Nested Loops Join (INLJ)

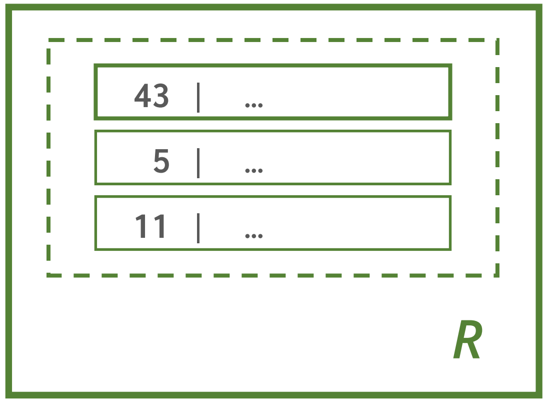
Cost = *M* + *m* * cost to find matching *S* tuples

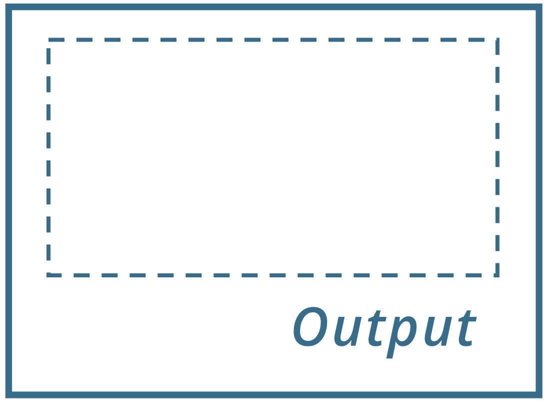
- *M* from scanning through *R*

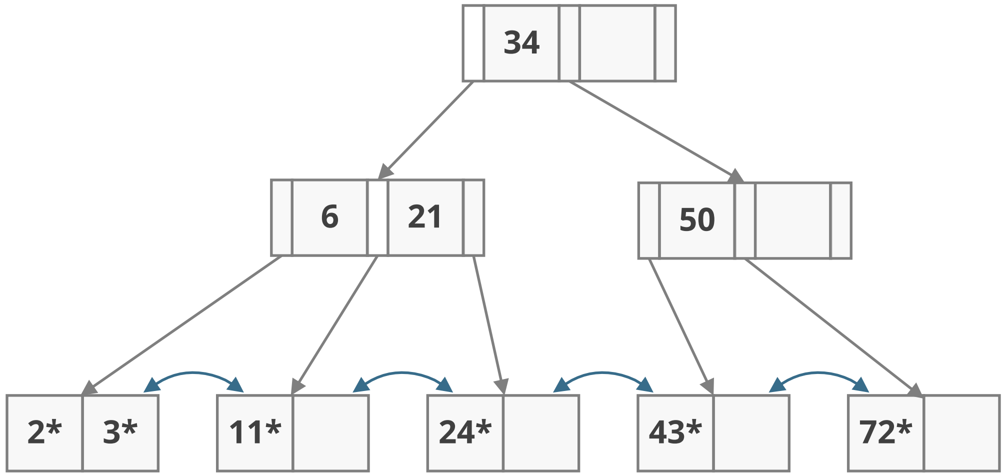
If we have **no index**

- Then the only way to search for matching *S* tuples is by scanning all of *S* ⇒ SNLJ
- Cost to find matching *S* tuples is then *N*, giving us the formula for SNLJ cost

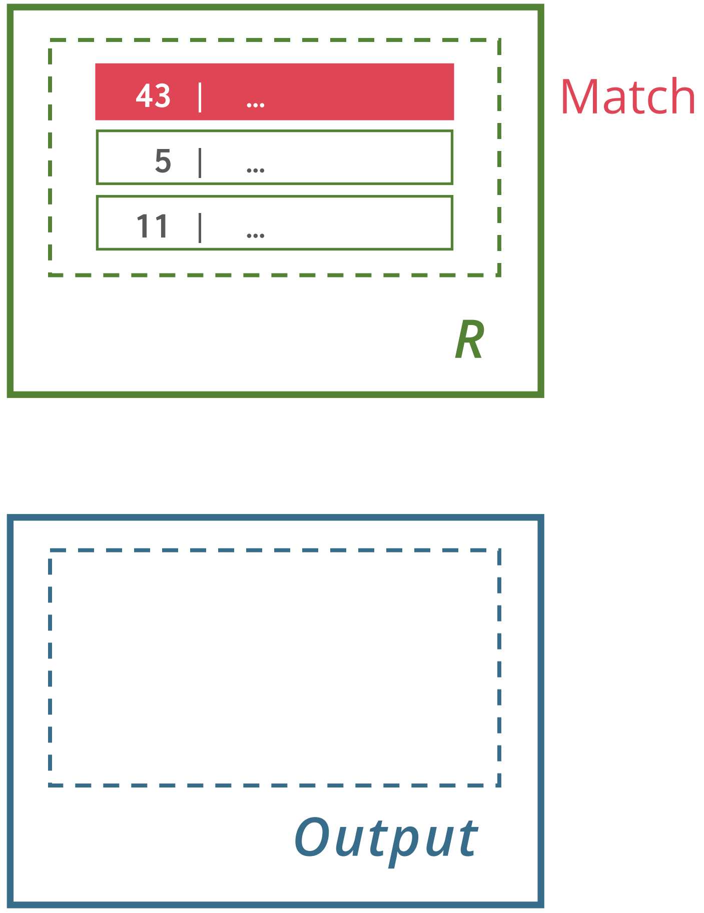
Index Nested Loops Join (INLJ)

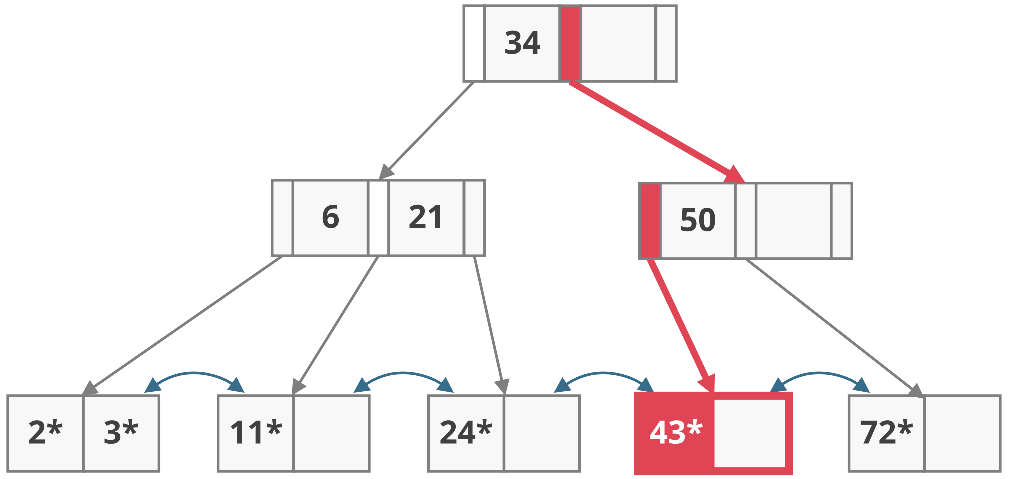




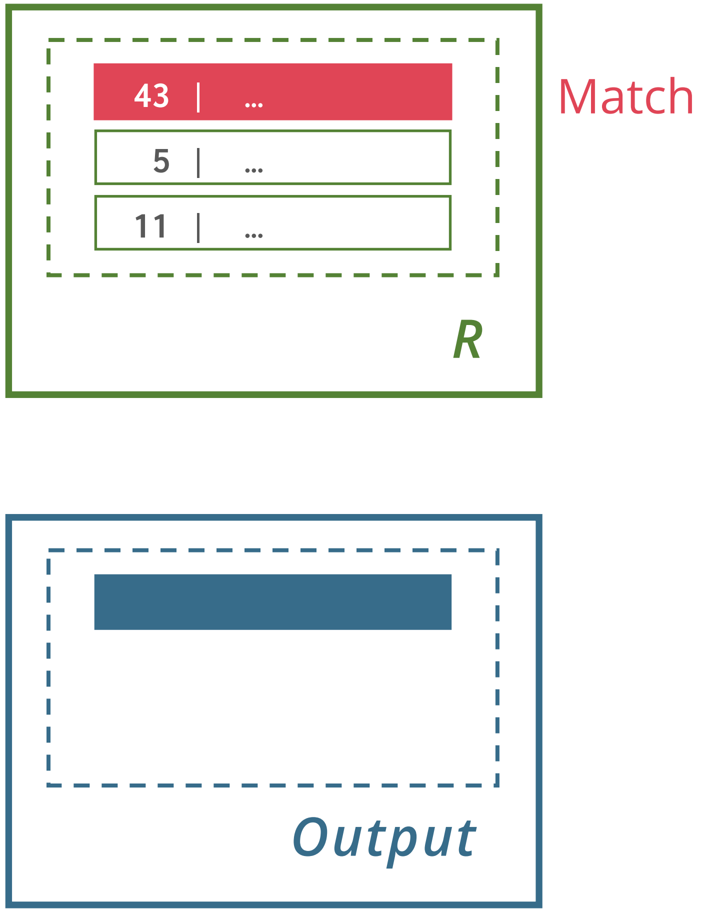


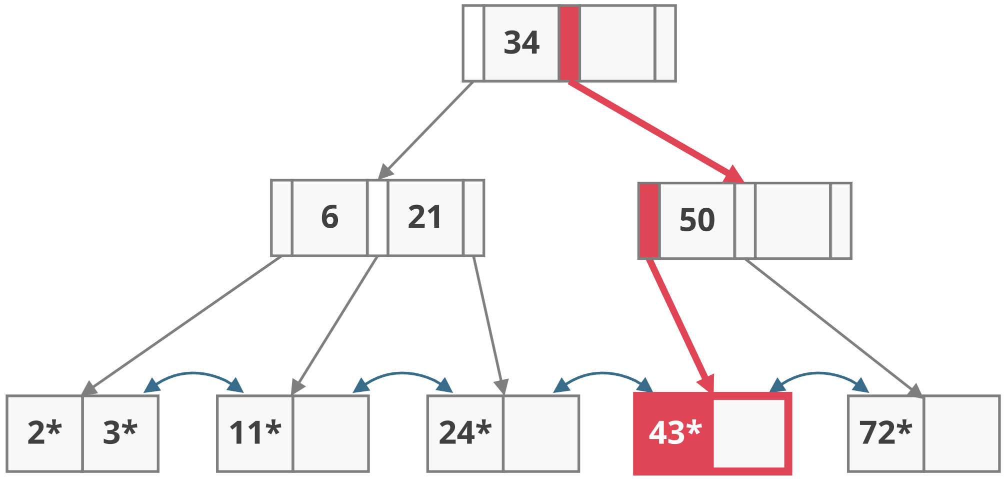
INDEX NESTED LOOPS JOIN (INLJ)



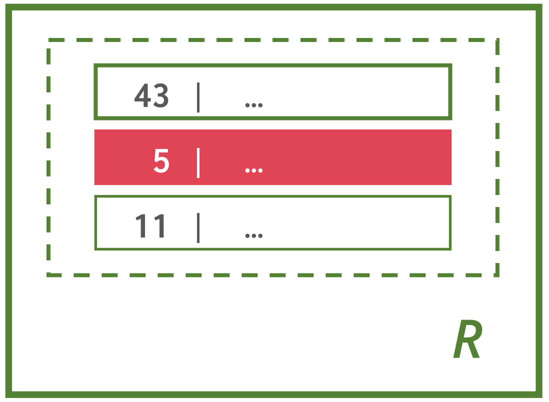


Index Nested Loops Join (INLJ)

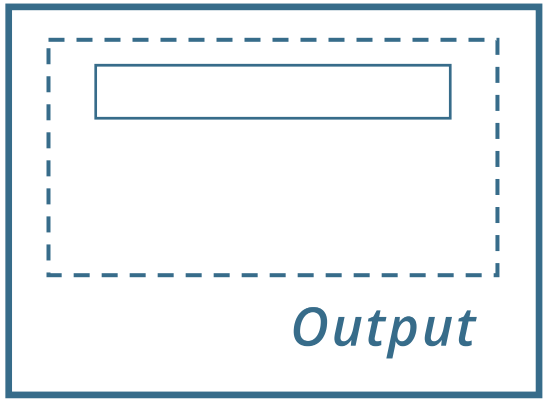


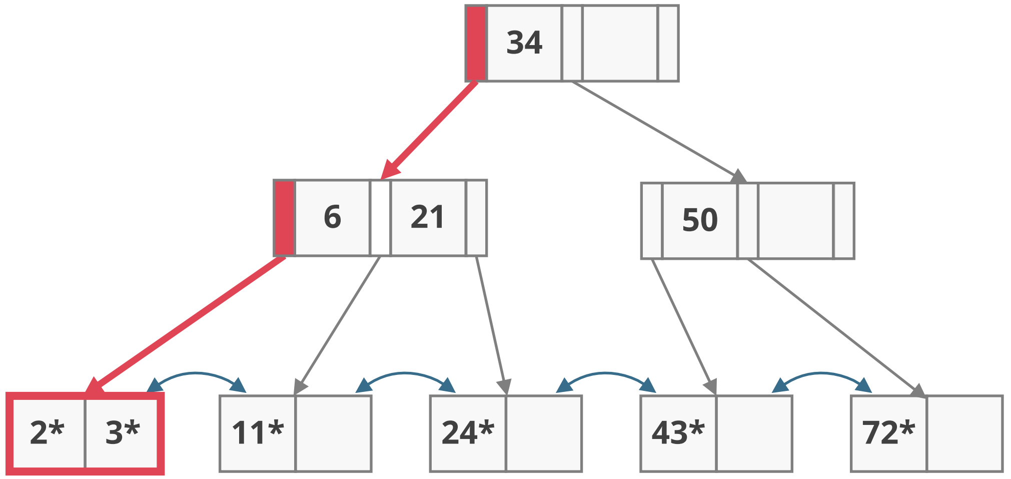


Index Nested Loops Join (INLJ)

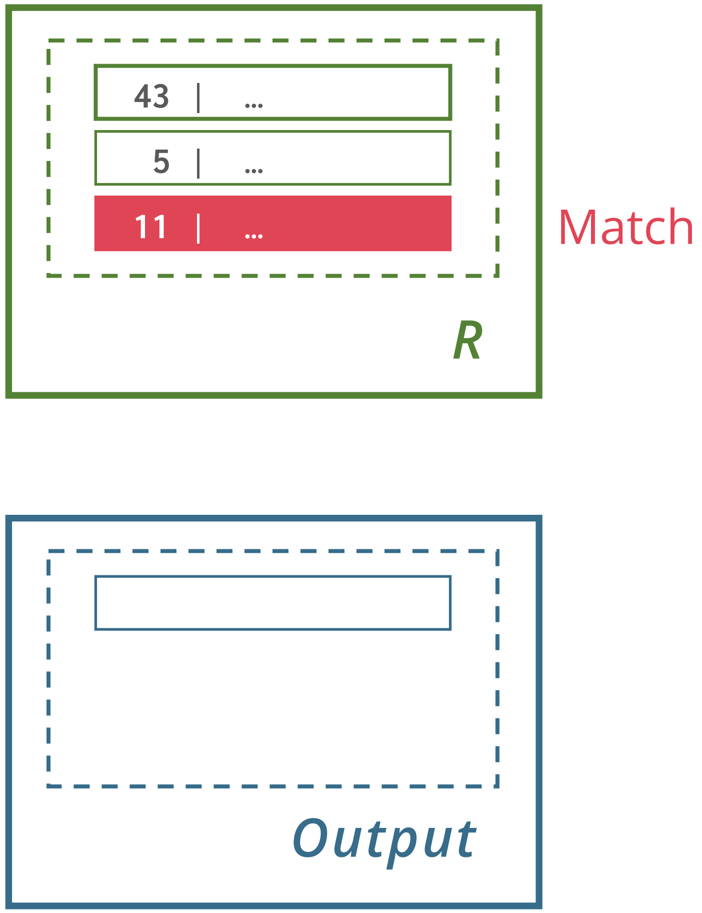


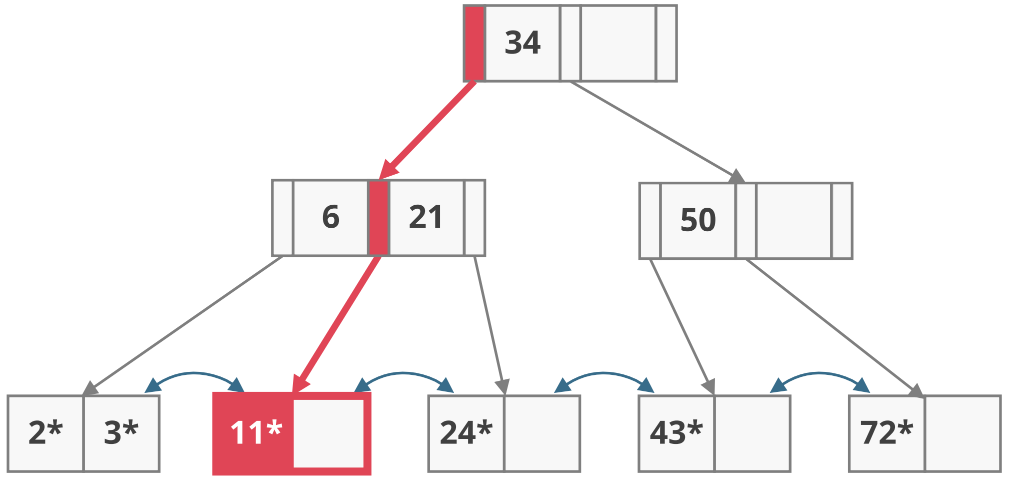
Not a match



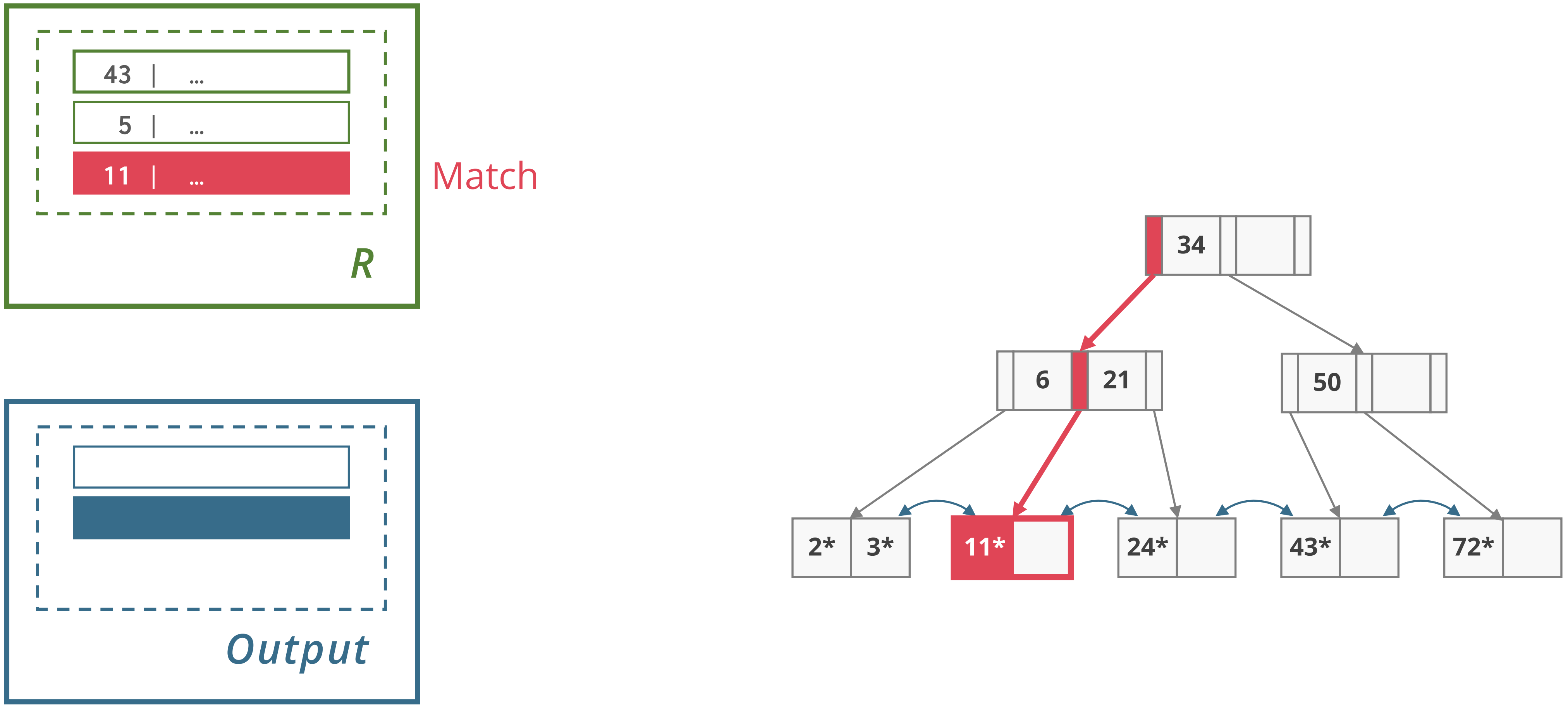


Index Nested Loops Join (INLJ)





Index Nested Loops Join (INLJ)



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 tuples in *R* matching with multiple tuples in *S*
    - **Mark** the first matching tuple in *S*
    - Match tuples with the first matching tuple in *R*,
    - Then **reset** the iterator to the mark
    - … so we can go through the tuples in *S* again for the second matching tuple in *R*

Sort Merge Join (SMJ)

```
while not done:
  while (r < s): advance r
  while (r > s): advance s

  mark s   // same start of "block"
  while (r == s):
    // outer loop over r
    while (r == s):
      // inner loop over s
      output r joined with s
      advance s
    reset s to mark
    advance r
```

sid	sname
22	John
28	Smith
31	Gold
31	Alvaro
44	McDonald
57	Gupta

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

Sort Merge Join (SMJ)

```
while not done:
  while (r < s): advance r
  while (r > s): advance s

  mark s   // same start of "block"
  while (r == s):
    // outer loop over r
    while (r == s):
      // inner loop over s
      output r joined with s
      advance s
    reset s to mark
    advance r
```

sid	sname
22	John
28	Smith
31	Gold
31	Alvaro
44	McDonald
57	Gupta

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

Sort Merge Join (SMJ)

```
while not done:
  while (r < s): advance r
  while (r > s): advance s

  mark s   // same start of "block"
  while (r == s):
    // outer loop over r
    while (r == s):
      // inner loop over s
      output r joined with s
      advance s
    reset s to mark
    advance r
```

sid	sname
22	John
28	Smith
31	Gold
31	Alvaro
44	McDonald
57	Gupta

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

Sort Merge Join (SMJ)

```
while not done:
  while (r < s): advance r
  while (r > s): advance s

  mark s   // same start of "block"
  while (r == s):
    // outer loop over r
    while (r == s):
      // inner loop over s
      output r joined with s
      advance s
    reset s to mark
    advance r
```

sid	sname
22	John
28	Smith
31	Gold
31	Alvaro
44	McDonald
57	Gupta

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

Sort Merge Join (SMJ)

```
while not done:
  while (r < s): advance r
  while (r > s): advance s

  mark s   // same start of “block”
  while (r == s):
    // outer loop over r
    while (r == s):
      // inner loop over s
      output r joined with s
      advance s
    reset s to mark
    advance r
```

sid	sname
22	John
28	Smith
31	Gold
31	Alvaro
44	McDonald
57	Gupta

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

sid	sname	bid
28	Smith	103

Sort Merge Join (SMJ)

```
while not done:
  while (r < s): advance r
  while (r > s): advance s

  mark s   // same start of "block"
  while (r == s):
    // outer loop over r
    while (r == s):
      // inner loop over s
      output r joined with s
      advance s
    reset s to mark
    advance r
```

sid	sname
22	John
28	Smith
31	Gold
31	Alvaro
44	McDonald
57	Gupta

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

sid	sname	bid
28	Smith	103

Sort Merge Join (SMJ)

```
while not done:
  while (r < s): advance r
  while (r > s): advance s

  mark s   // same start of "block"
  while (r == s):
    // outer loop over r
    while (r == s):
      // inner loop over s
      output r joined with s
      advance s
    reset s to mark
    advance r
```

sid	sname
22	John
28	Smith
31	Gold
31	Alvaro
44	McDonald
57	Gupta

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

sid	sname	bid
28	Smith	103
28	Smith	104

Sort Merge Join (SMJ)

```
while not done:
  while (r < s): advance r
  while (r > s): advance s

  mark s   // same start of "block"
  while (r == s):
    // outer loop over r
    while (r == s):
      // inner loop over s
      output r joined with s
      advance s
    reset s to mark
    advance r
```

sid	sname
22	John
28	Smith
31	Gold
31	Alvaro
44	McDonald
57	Gupta

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

sid	sname	bid
28	Smith	103
28	Smith	104

Sort Merge Join (SMJ)

```
while not done:
  while (r < s): advance r
  while (r > s): advance s

  mark s   // same start of "block"
  while (r == s):
    // outer loop over r
    while (r == s):
      // inner loop over s
      output r joined with s
      advance s
    reset s to mark
    advance r
```

sid	sname
22	John
28	Smith
31	Gold
31	Alvaro
44	McDonald
57	Gupta

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

sid	sname	bid
28	Smith	103
28	Smith	104

SORT MERGE JOIN (SMJ)

```
while not done:
  while (r < s): advance r
  while (r > s): advance s

  mark s   // same start of "block"
  while (r == s):
    // outer loop over r
    while (r == s):
      // inner loop over s
      output r joined with s
      advance s
    reset s to mark
    advance r
```

sid	sname
22	John
28	Smith
31	Gold
31	Alvaro
44	McDonald
57	Gupta

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

sid	sname	bid
28	Smith	103
28	Smith	104

Sort Merge Join (SMJ)

```
while not done:
  while (r < s): advance r
  while (r > s): advance s

  mark s   // same start of "block"
  while (r == s):
    // outer loop over r
    while (r == s):
      // inner loop over s
      output r joined with s
      advance s
    reset s to mark
    advance r
```

sid	sname
22	John
28	Smith
31	Gold
31	Alvaro
44	McDonald
57	Gupta

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

sid	sname	bid
28	Smith	103
28	Smith	104

Sort Merge Join (SMJ)

```
while not done:
  while (r < s): advance r
  while (r > s): advance s

  mark s   // same start of "block"
  while (r == s):
    // outer loop over r
    while (r == s):
      // inner loop over s
      output r joined with s
      advance s
    reset s to mark
    advance r
```

sid	sname
22	John
28	Smith
31	Gold
31	Alvaro
44	McDonald
57	Gupta

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

sid	sname	bid
28	Smith	103
28	Smith	104

Sort Merge Join (SMJ)

```
while not done:
  while (r < s): advance r
  while (r > s): advance s

  mark s    // same start of "block"
  while (r == s):
    // outer loop over r
    while (r == s):
      // inner loop over s
      output r joined with s
      advance s
    reset s to mark
    advance r
```

sid	sname
22	John
28	Smith
31	Gold
31	Alvaro
44	McDonald
57	Gupta

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

sid	sname	bid
28	Smith	103
28	Smith	104

Sort Merge Join (SMJ)

```
while not done:
  while (r < s): advance r
  while (r > s): advance s

  mark s    // same start of "block"
  while (r == s):
    // outer loop over r
    while (r == s):
      // inner loop over s
      output r joined with s
      advance s
    reset s to mark
    advance r
```

sid	sname
22	John
28	Smith
31	Gold
31	Alvaro
44	McDonald
57	Gupta

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

sid	sname	bid
28	Smith	103
28	Smith	104

Sort Merge Join (SMJ)

```
while not done:
  while (r < s): advance r
  while (r > s): advance s

  mark s   // same start of "block"
  while (r == s):
    // outer loop over r
    while (r == s):
      // inner loop over s
      output r joined with s
      advance s
    reset s to mark
    advance r
```

sid	sname
22	John
28	Smith
31	Gold
31	Alvaro
44	McDonald
57	Gupta

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

sid	sname	bid
28	Smith	103
28	Smith	104
31	Gold	101

Sort Merge Join (SMJ)

```
while not done:
  while (r < s): advance r
  while (r > s): advance s

  mark s   // same start of "block"
  while (r == s):
    // outer loop over r
    while (r == s):
      // inner loop over s
      output r joined with s
      advance s
    reset s to mark
    advance r
```

sid	sname
22	John
28	Smith
31	Gold
31	Alvaro
44	McDonald
57	Gupta

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

sid	sname	bid
28	Smith	103
28	Smith	104
31	Gold	101

Sort Merge Join (SMJ)

```
while not done:
  while (r < s): advance r
  while (r > s): advance s

  mark s   // same start of “block”
  while (r == s):
    // outer loop over r
    while (r == s):
      // inner loop over s
      output r joined with s
      advance s
    reset s to mark
    advance r
```

sid	sname
22	John
28	Smith
31	Gold
31	Alvaro
44	McDonald
57	Gupta

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

sid	sname	bid
28	Smith	103
28	Smith	104
31	Gold	101
31	Gold	102

Sort Merge Join (SMJ)

```
while not done:
  while (r < s): advance r
  while (r > s): advance s

  mark s   // same start of "block"
  while (r == s):
    // outer loop over r
    while (r == s):
      // inner loop over s
      output r joined with s
      advance s
    reset s to mark
    advance r
```

sid	sname
22	John
28	Smith
31	Gold
31	Alvaro
44	McDonald
57	Gupta

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

sid	sname	bid
28	Smith	103
28	Smith	104
31	Gold	101
31	Gold	102

Sort Merge Join (SMJ)

```
while not done:
  while (r < s): advance r
  while (r > s): advance s

  mark s   // same start of “block”
  while (r == s):
    // outer loop over r
    while (r == s):
      // inner loop over s
      output r joined with s
      advance s
    reset s to mark
    advance r
```

sid	sname
22	John
28	Smith
31	Gold
31	Alvaro
44	McDonald
57	Gupta

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

sid	sname	bid
28	Smith	103
28	Smith	104
31	Gold	101
31	Gold	102

Sort Merge Join (SMJ)

```
while not done:
  while (r < s): advance r
  while (r > s): advance s

  mark s   // same start of "block"
  while (r == s):
    // outer loop over r
    while (r == s):
      // inner loop over s
      output r joined with s
      advance s
    reset s to mark
    advance r
```

sid	sname
22	John
28	Smith
31	Gold
31	Alvaro
44	McDonald
57	Gupta

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

sid	sname	bid
28	Smith	103
28	Smith	104
31	Gold	101
31	Gold	102

Sort Merge Join (SMJ)

```
while not done:
  while (r < s): advance r
  while (r > s): advance s

  mark s   // same start of "block"
  while (r == s):
    // outer loop over r
    while (r == s):
      // inner loop over s
      output r joined with s
      advance s
    reset s to mark
    advance r
```

sid	sname
22	John
28	Smith
31	Gold
31	Alvaro
44	McDonald
57	Gupta

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

sid	sname	bid
28	Smith	103
28	Smith	104
31	Gold	101
31	Gold	102
31	Alvaro	101

Sort Merge Join (SMJ)

```
while not done:
  while (r < s): advance r
  while (r > s): advance s

  mark s   // same start of "block"
  while (r == s):
    // outer loop over r
    while (r == s):
      // inner loop over s
      output r joined with s
      advance s
    reset s to mark
    advance r
```

sid	sname
22	John
28	Smith
31	Gold
31	Alvaro
44	McDonald
57	Gupta

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

sid	sname	bid
28	Smith	103
28	Smith	104
31	Gold	101
31	Gold	102
31	Alvaro	101

Sort Merge Join (SMJ)

```
while not done:
  while (r < s): advance r
  while (r > s): advance s

  mark s   // same start of "block"
  while (r == s):
    // outer loop over r
    while (r == s):
      // inner loop over s
      output r joined with s
      advance s
    reset s to mark
    advance r
```

sid	sname
22	John
28	Smith
31	Gold
31	Alvaro
44	McDonald
57	Gupta

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

sid	sname	bid
28	Smith	103
28	Smith	104
31	Gold	101
31	Gold	102
31	Alvaro	101
31	Alvaro	102

Sort Merge Join (SMJ)

```
while not done:
  while (r < s): advance r
  while (r > s): advance s

  mark s   // same start of "block"
  while (r == s):
    // outer loop over r
    while (r == s):
      // inner loop over s
      output r joined with s
      advance s
    reset s to mark
    advance r
```

sid	sname
22	John
28	Smith
31	Gold
31	Alvaro
44	McDonald
57	Gupta

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

sid	sname	bid
28	Smith	103
28	Smith	104
31	Gold	101
31	Gold	102
31	Alvaro	101
31	Alvaro	102

Sort Merge Join (SMJ)

```
while not done:
  while (r < s): advance r
  while (r > s): advance s

  mark s   // same start of “block”
  while (r == s):
    // outer loop over r
    while (r == s):
      // inner loop over s
      output r joined with s
      advance s
    reset s to mark
    advance r
```

sid	sname
22	John
28	Smith
31	Gold
31	Alvaro
44	McDonald
57	Gupta

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

sid	sname	bid
28	Smith	103
28	Smith	104
31	Gold	101
31	Gold	102
31	Alvaro	101
31	Alvaro	102

Sort Merge Join (SMJ)

```
while not done:
  while (r < s): advance r
  while (r > s): advance s

  mark s   // same start of "block"
  while (r == s):
    // outer loop over r
    while (r == s):
      // inner loop over s
      output r joined with s
      advance s
    reset s to mark
    advance r
```

sid	sname
22	John
28	Smith
31	Gold
31	Alvaro
44	McDonald
57	Gupta

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

sid	sname	bid
28	Smith	103
28	Smith	104
31	Gold	101
31	Gold	102
31	Alvaro	101
31	Alvaro	102

Sort Merge Join (SMJ)

```
while not done:
  while (r < s): advance r
  while (r > s): advance s

  mark s   // same start of "block"
  while (r == s):
    // outer loop over r
    while (r == s):
      // inner loop over s
      output r joined with s
      advance s
    reset s to mark
    advance r
```

sid	sname
22	John
28	Smith
31	Gold
31	Alvaro
44	McDonald
57	Gupta

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

sid	sname	bid
28	Smith	103
28	Smith	104
31	Gold	101
31	Gold	102
31	Alvaro	101
31	Alvaro	102

Sort Merge Join (SMJ)

```
while not done:
  while (r < s): advance r
  while (r > s): advance s

  mark s   // same start of "block"
  while (r == s):
    // outer loop over r
    while (r == s):
      // inner loop over s
      output r joined with s
      advance s
    reset s to mark
    advance r
```

sid	sname
22	John
28	Smith
31	Gold
31	Alvaro
44	McDonald
57	Gupta

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

sid	sname	bid
28	Smith	103
28	Smith	104
31	Gold	101
31	Gold	102
31	Alvaro	101
31	Alvaro	102

Sort Merge Join (SMJ)

```
while not done:
  while (r < s): advance r
  while (r > s): advance s

  mark s   // same start of "block"
  while (r == s):
    // outer loop over r
    while (r == s):
      // inner loop over s
      output r joined with s
      advance s
    reset s to mark
    advance r
```

sid	sname
22	John
28	Smith
31	Gold
31	Alvaro
44	McDonald
57	Gupta

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

sid	sname	bid
28	Smith	103
28	Smith	104
31	Gold	101
31	Gold	102
31	Alvaro	101
31	Alvaro	102

Sort Merge Join (SMJ)

```
while not done:
  while (r < s): advance r
  while (r > s): advance s

  mark s   // same start of "block"
  while (r == s):
    // outer loop over r
    while (r == s):
      // inner loop over s
      output r joined with s
      advance s
    reset s to mark
    advance r
```

sid	sname
22	John
28	Smith
31	Gold
31	Alvaro
44	McDonald
57	Gupta

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

sid	sname	bid
28	Smith	103
28	Smith	104
31	Gold	101
31	Gold	102
31	Alvaro	101
31	Alvaro	102

Sort Merge Join (SMJ)

```
while not done:
  while (r < s): advance r
  while (r > s): advance s

  mark s   // same start of “block”
  while (r == s):
    // outer loop over r
    while (r == s):
      // inner loop over s
      output r joined with s
      advance s
    reset s to mark
    advance r
```

sid	sname
22	John
28	Smith
31	Gold
31	Alvaro
44	McDonald
57	Gupta

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

sid	sname	bid
28	Smith	103
28	Smith	104
31	Gold	101
31	Gold	102
31	Alvaro	101
31	Alvaro	102

Sort Merge Join (SMJ)

```
while not done:
  while (r < s): advance r
  while (r > s): advance s

  mark s   // same start of "block"
  while (r == s):
    // outer loop over r
    while (r == s):
      // inner loop over s
      output r joined with s
      advance s
    reset s to mark
    advance r
```

sid	sname
22	John
28	Smith
31	Gold
31	Alvaro
44	McDonald
57	Gupta

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

sid	sname	bid
28	Smith	103
28	Smith	104
31	Gold	101
31	Gold	102
31	Alvaro	101
31	Alvaro	102

Sort Merge Join (SMJ)

```
while not done:
    while (r < s): advance r
    while (r > s): advance s

    mark s    // same start of "block"
    while (r == s):
        // outer loop over r
        while (r == s):
            // inner loop over s
            output r joined with s
            advance s
        reset s to mark
        advance r
```

sid	sname
22	John
28	Smith
31	Gold
31	Alvaro
44	McDonald
57	Gupta

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

sid	sname	bid
28	Smith	103
28	Smith	104
31	Gold	101
31	Gold	102
31	Alvaro	101
31	Alvaro	102

Sort Merge Join (SMJ)

Total cost:

- Cost of sorting ***R***
- Cost of sorting ***S***
- Cost of merging: $M + N$
  - Only one pass (if we assume there aren’t a lot of duplicates)

Sort Merge Join (SMJ)

We can be sometimes smarter about SMJ

Observation:

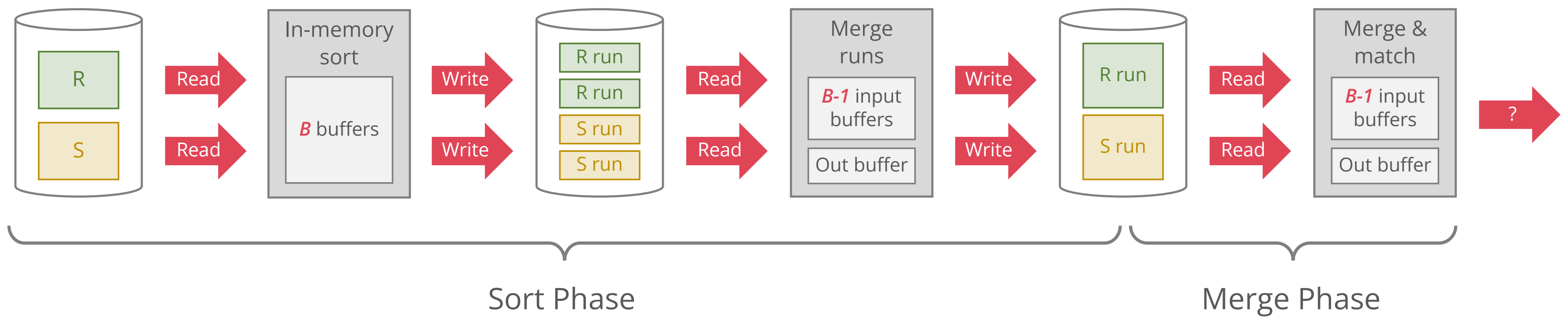
We make **one pass** through each sorted relation (assuming no duplicate values in ***R***)

⇒ We do **not need** the sorted relations to be materialized!

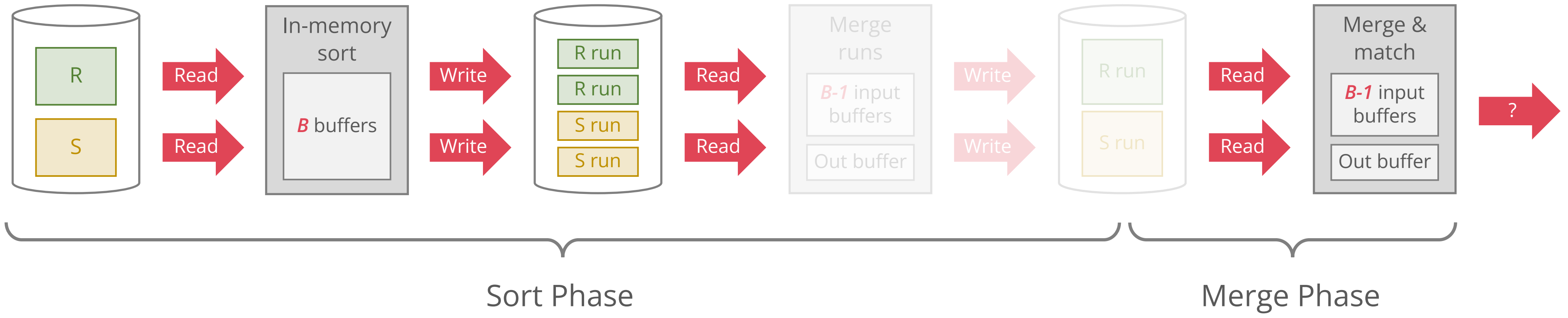
Optimisation:

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!

Sort Merge Join (SMJ)



Sort Merge Join (SMJ)



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***) ≤ ***B – 1***

Reduces I/O cost by ***2 · (M + N)***!

Grace Hash Join

Similar idea as SMJ, but let's build some hash tables instead

Two passes: **partition** the data, then **build** a hash table and **probe** it

- Partition ***R*** and ***S*** into ***B - 1*** partitions (like in external hashing) using 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

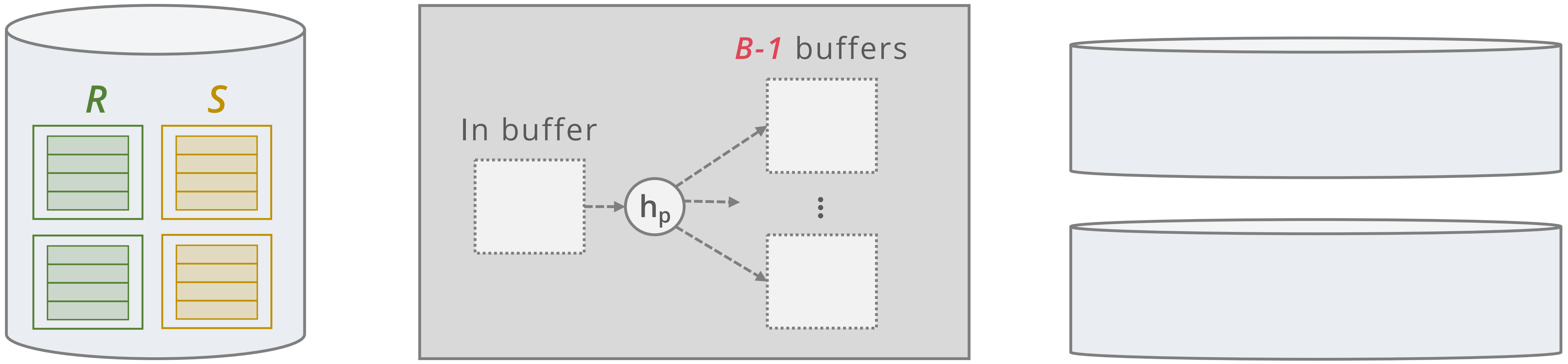
Grace Hash Join

Similar idea as SMJ, but let’s build some hash tables instead

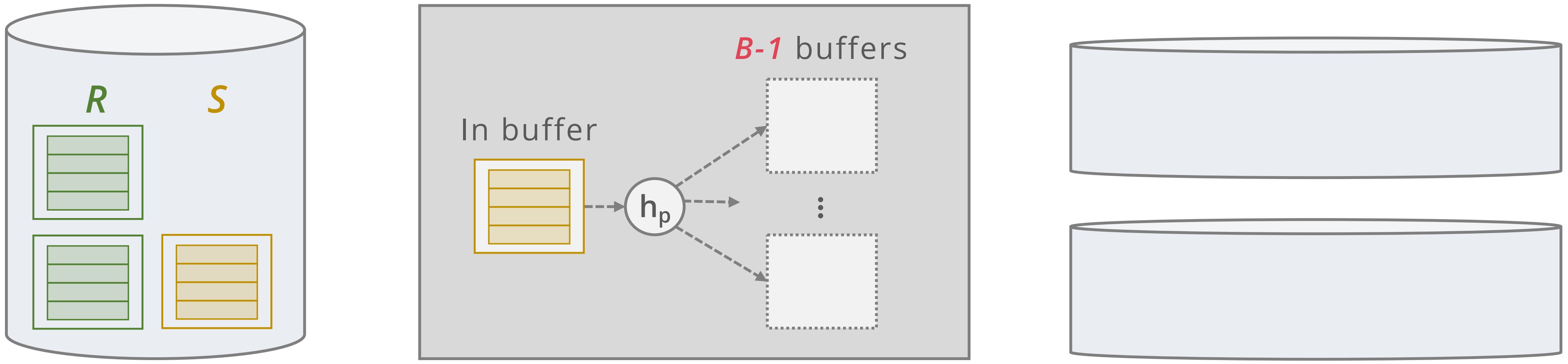
Two passes: **partition** the data, then **build** a hash table and **probe** it

- 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

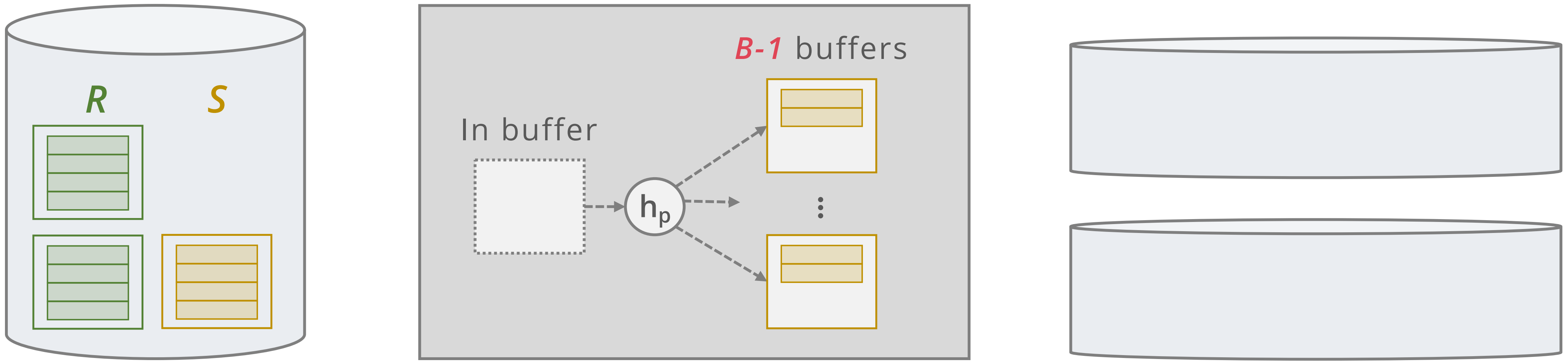
Grace Hash Join: Partition



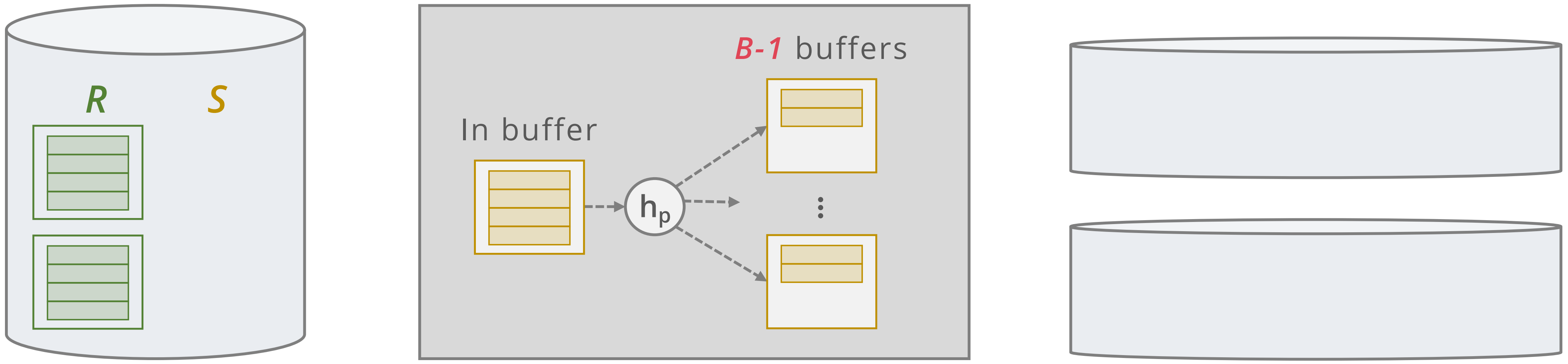
GRACE HASH JOIN: PARTITION



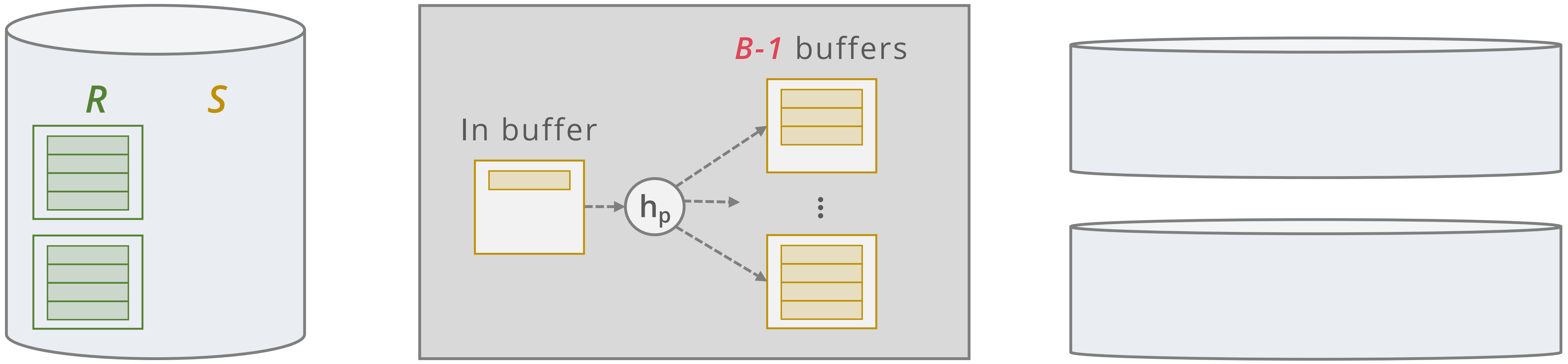
Grace Hash Join: Partition



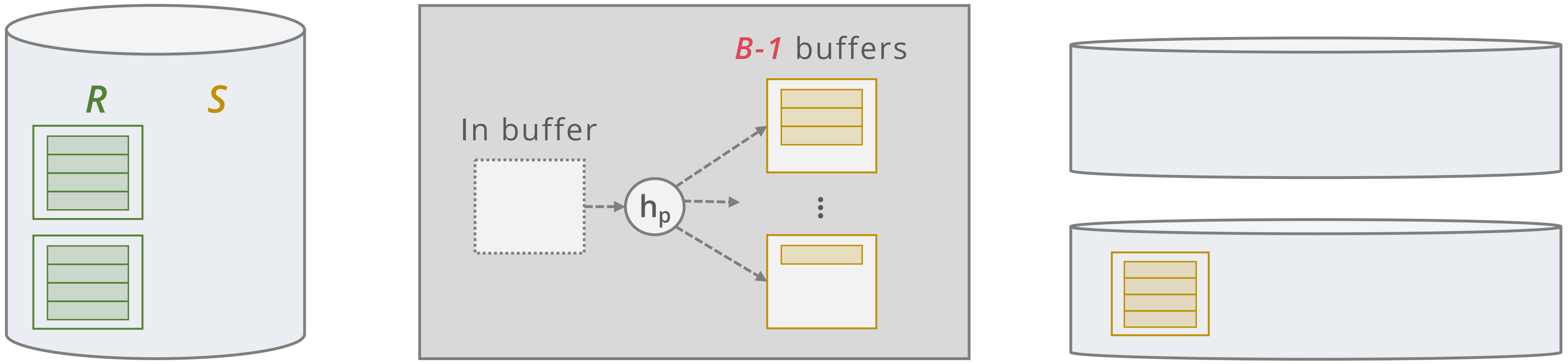
Grace Hash Join: Partition



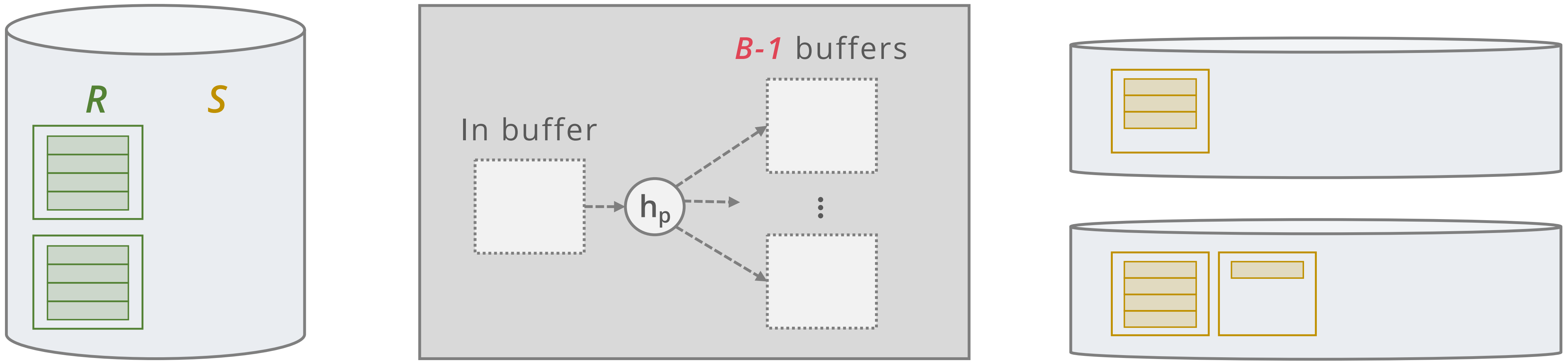
Grace Hash Join: Partition



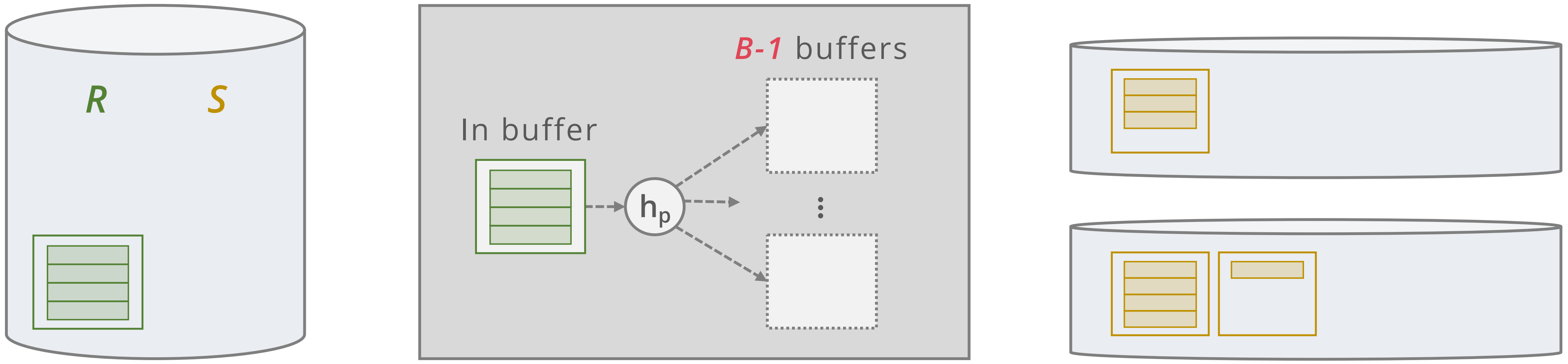
Grace Hash Join: Partition



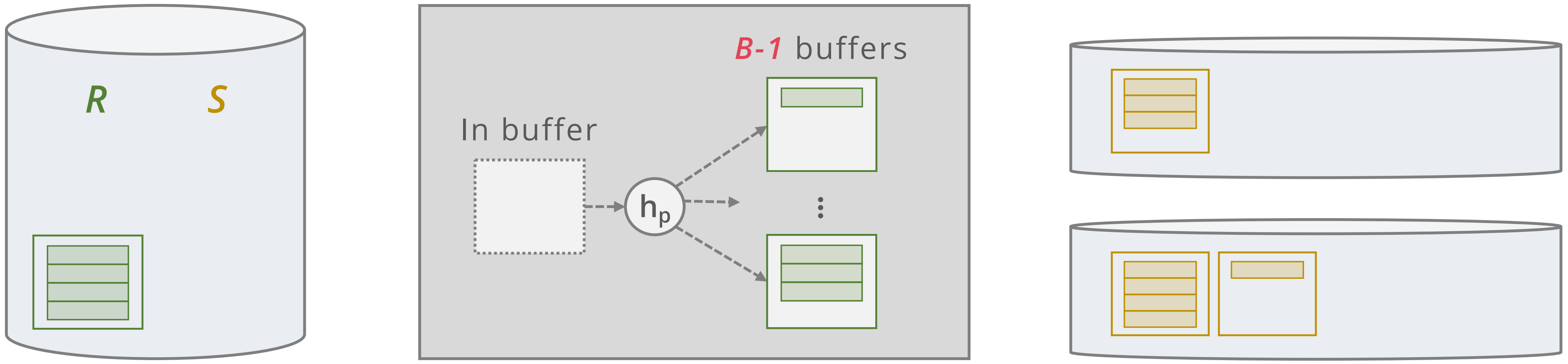
Grace Hash Join: Partition



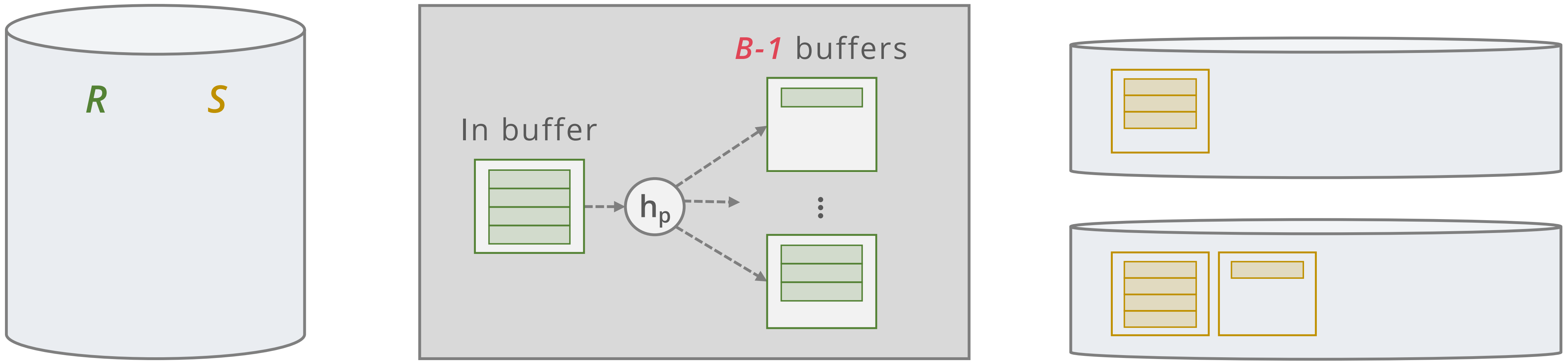
Grace Hash Join: Partition



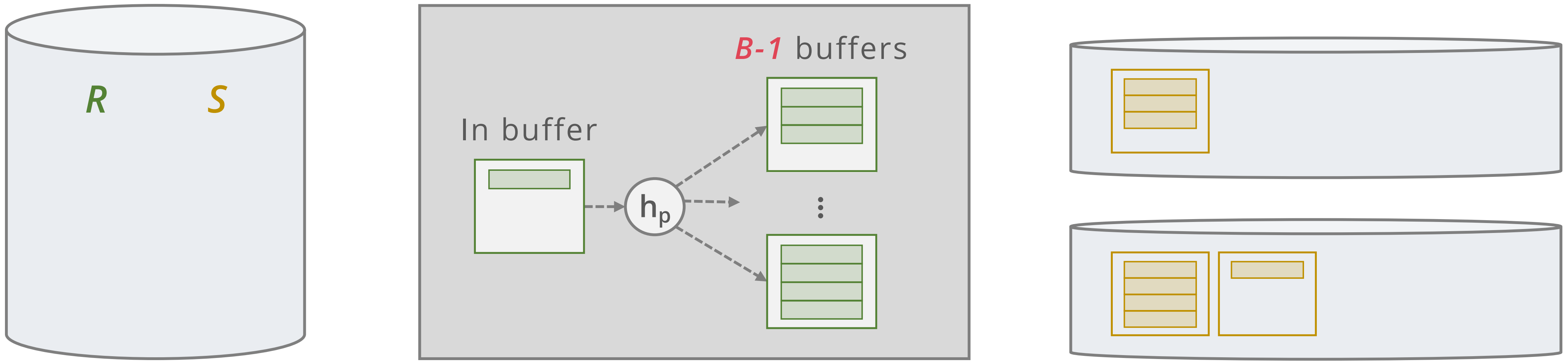
Grace Hash Join: Partition



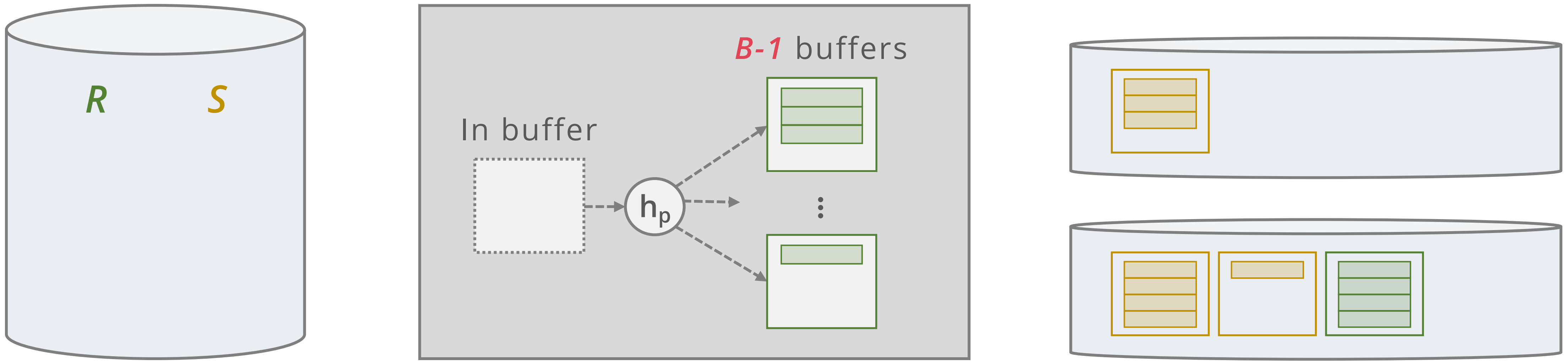
GRACE HASH JOIN: PARTITION



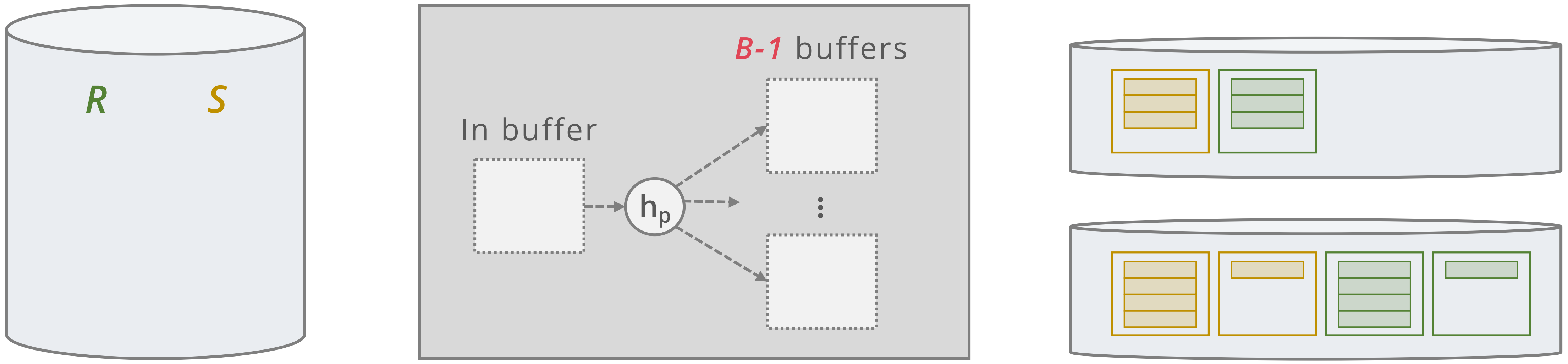
Grace Hash Join: Partition



Grace Hash Join: Partition



Grace Hash Join: Partition



Grace Hash Join

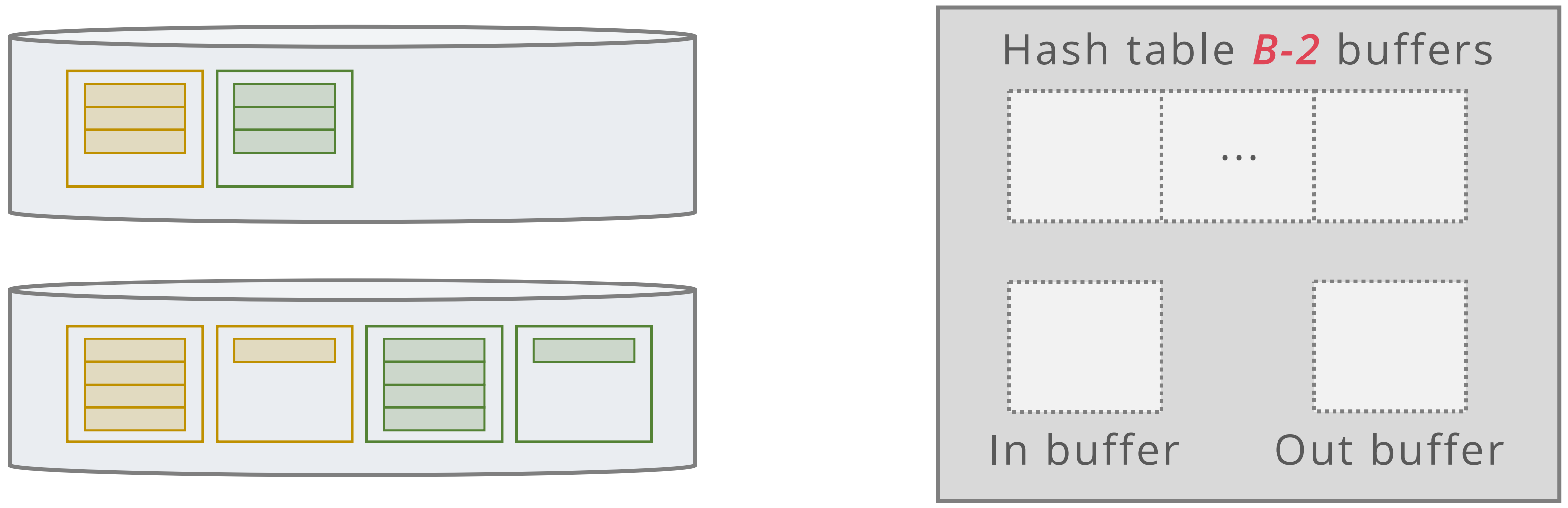
We need partitions of ***R*** (but not ***S***!) to fit in ***B* - 2** pages

- 1 page reserved for streaming ***S*** partition
- 1 page reserved for streaming output

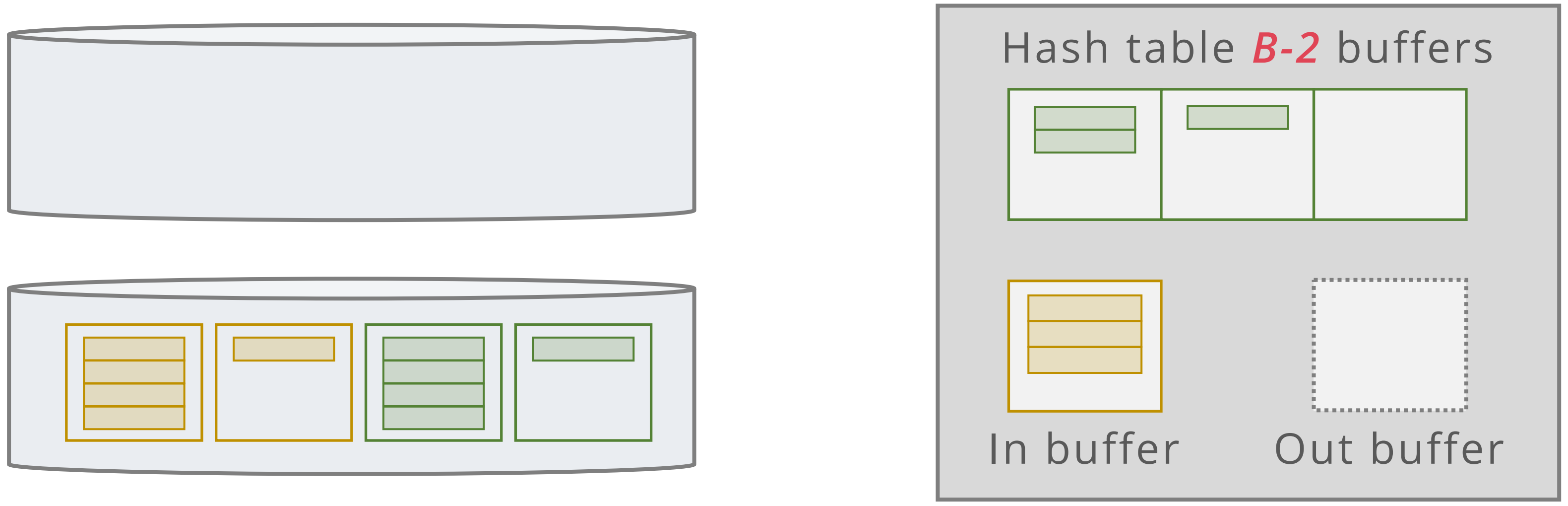
What if partitions of ***R*** are too big?

- 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!

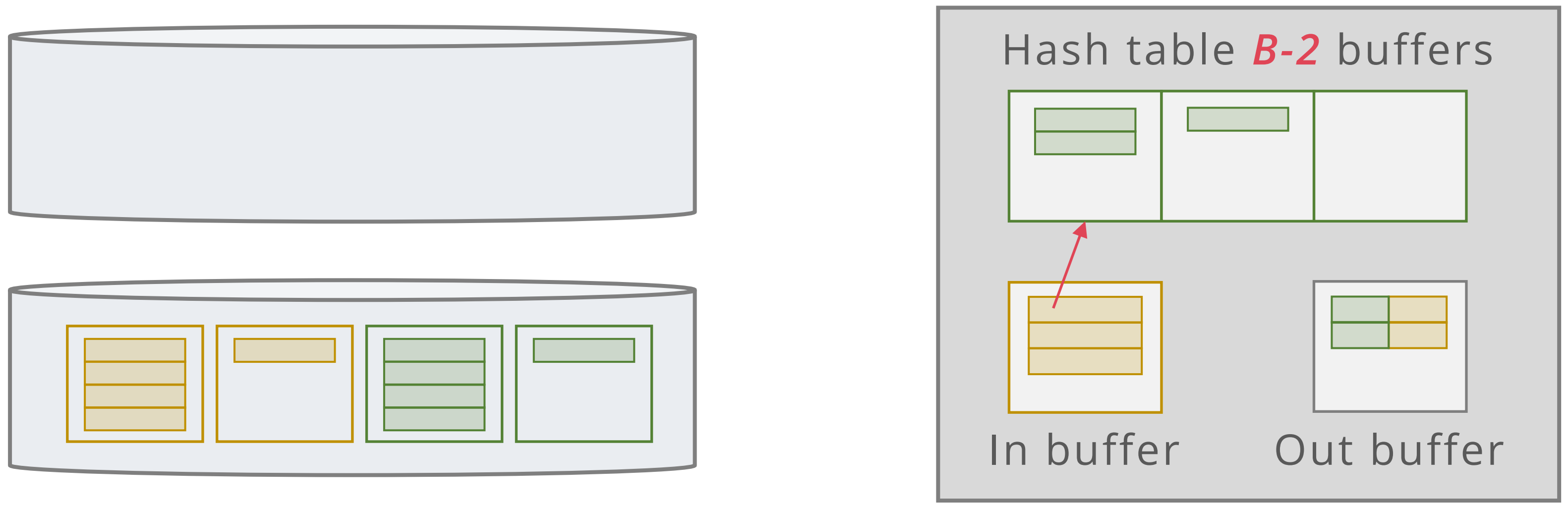
Grace Hash Join: Build & Probe



GRACE HASH JOIN: BUILD & PROBE

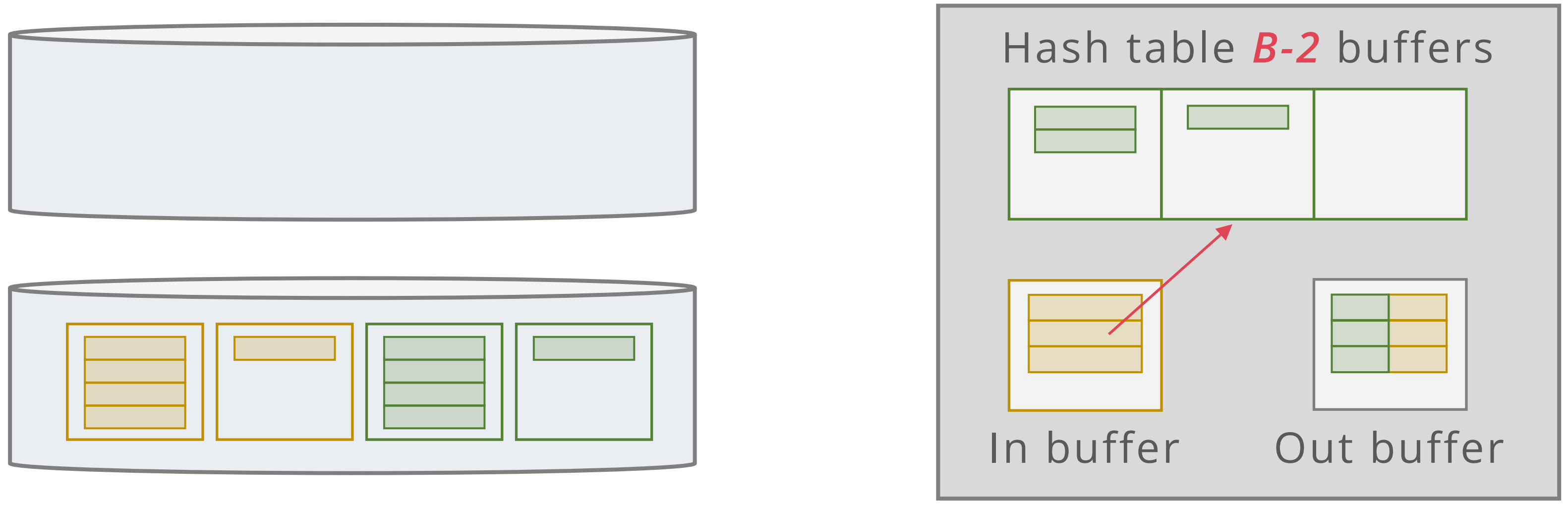


Grace Hash Join: Build & Probe



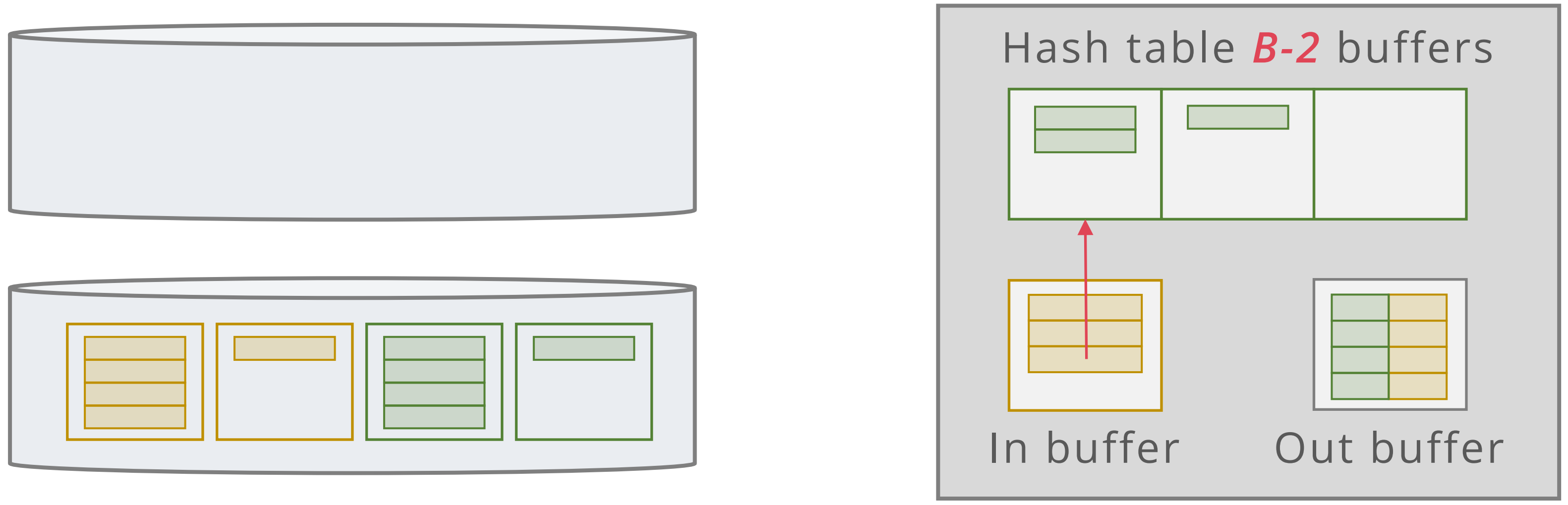
2 matches

Grace Hash Join: Build & Probe



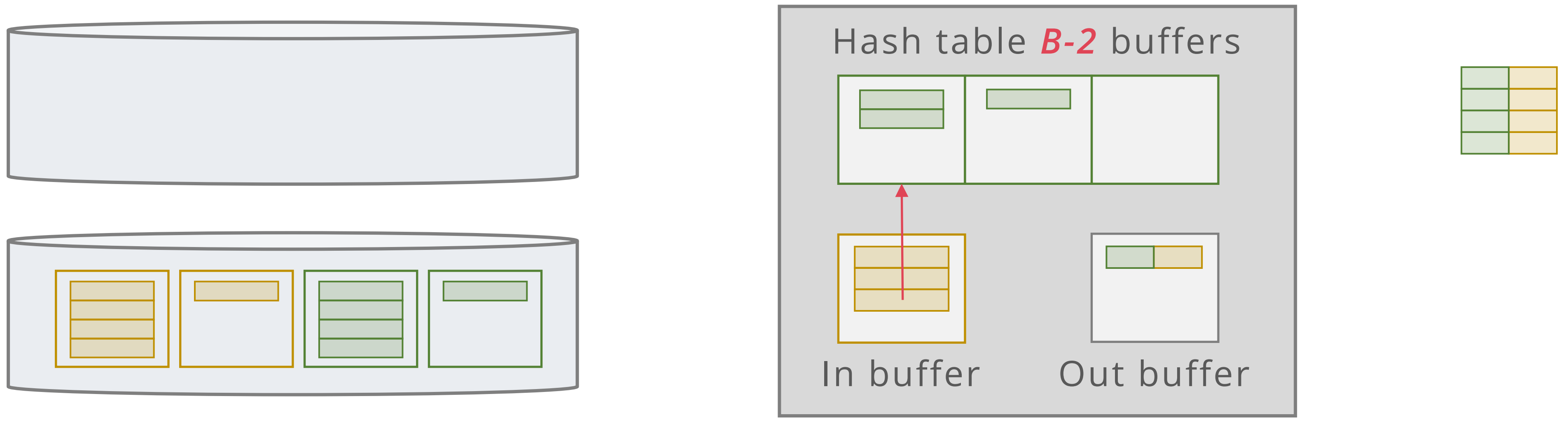
1 match

Grace Hash Join: Build & Probe



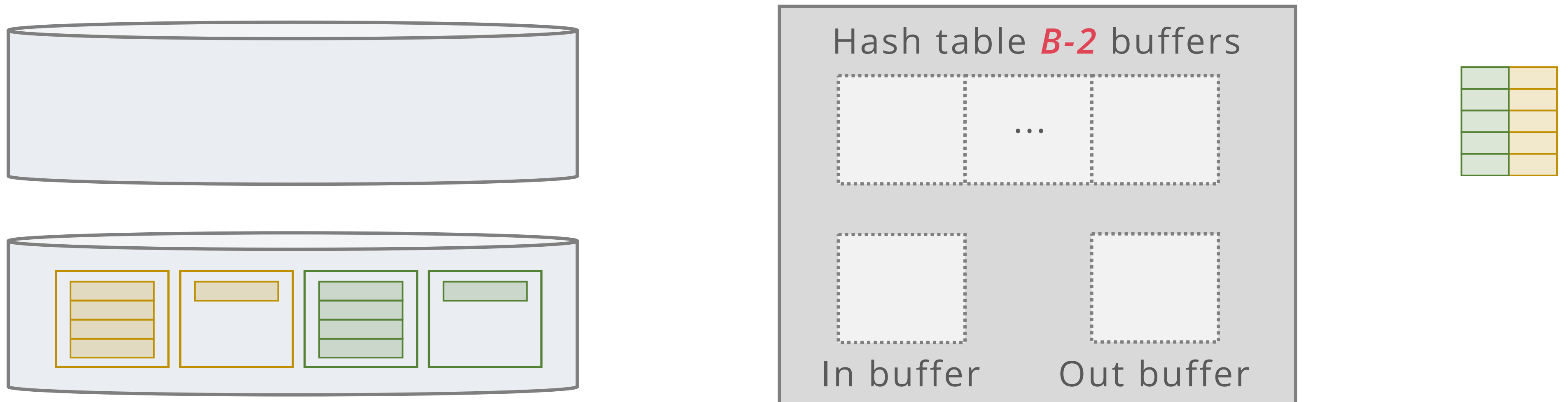
2 matches

Grace Hash Join: Build & Probe



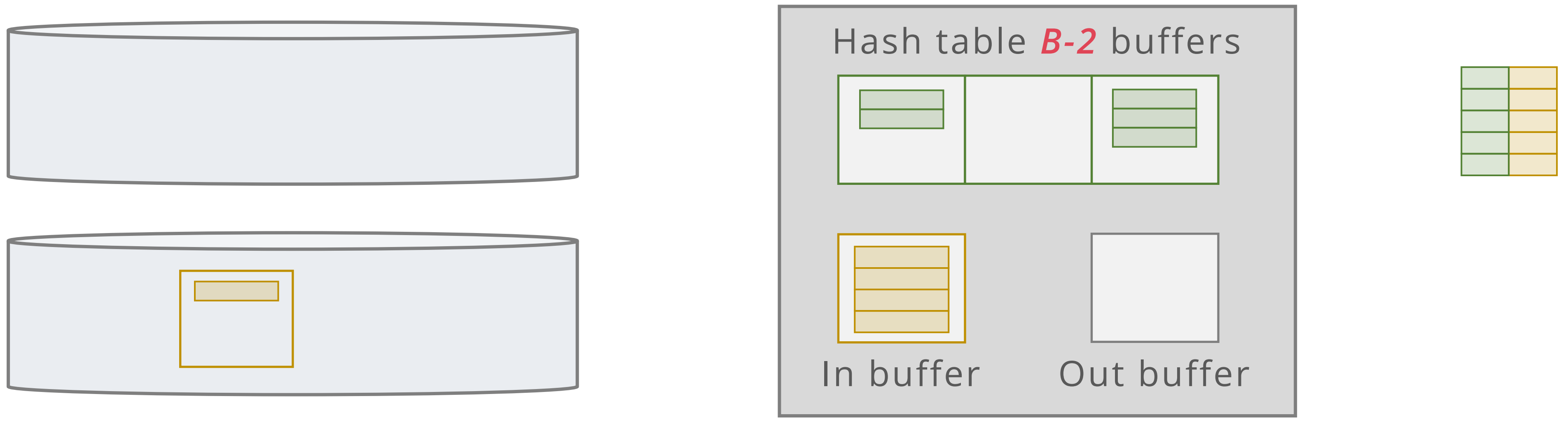
2 matches

Grace Hash Join: Build & Probe

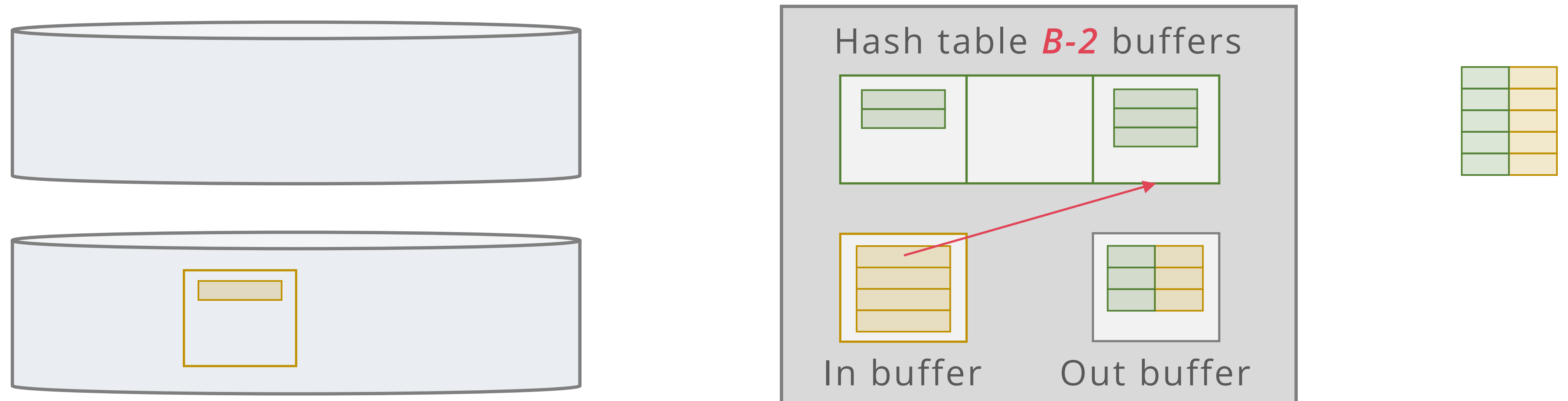


Flushing output buffer not strictly necessary, can reuse for next pair of partitions

Grace Hash Join: Build & Probe

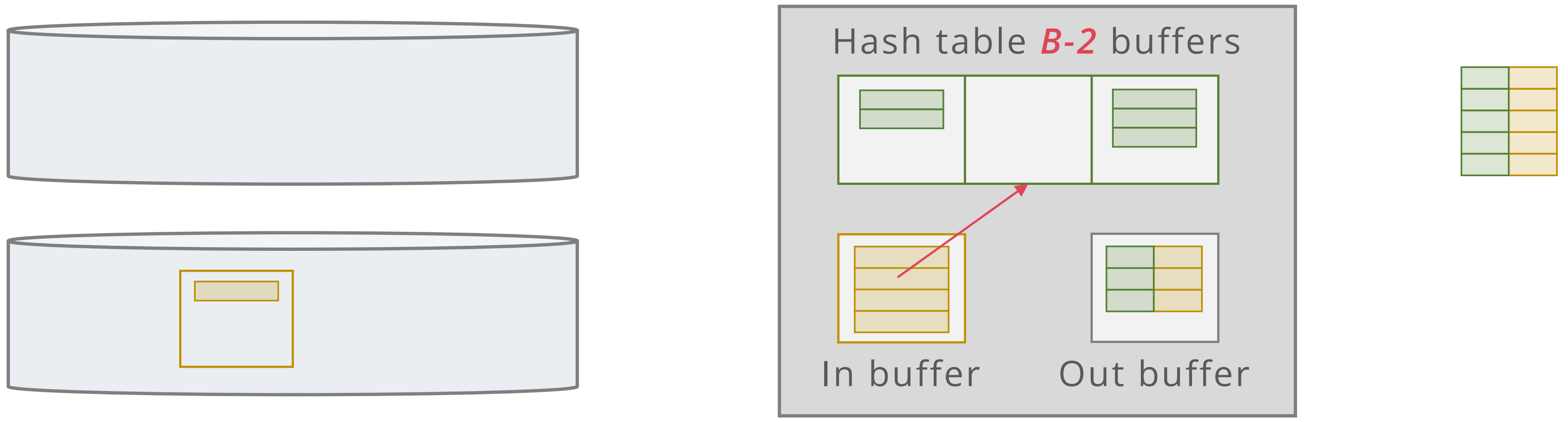


Grace Hash Join: Build & Probe



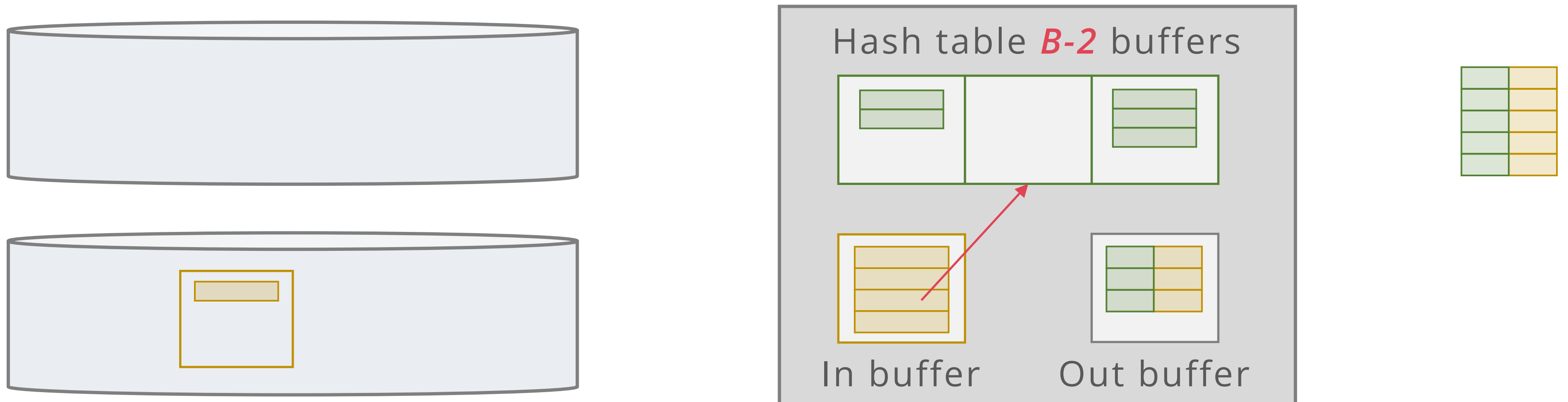
3 matches

Grace Hash Join: Build & Probe



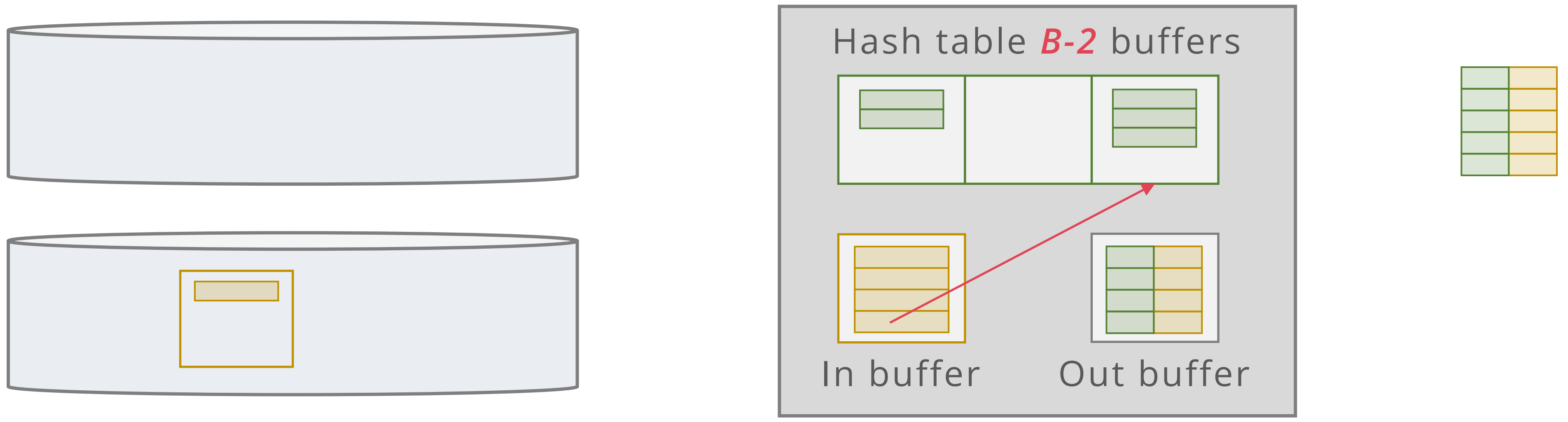
No match

Grace Hash Join: Build & Probe



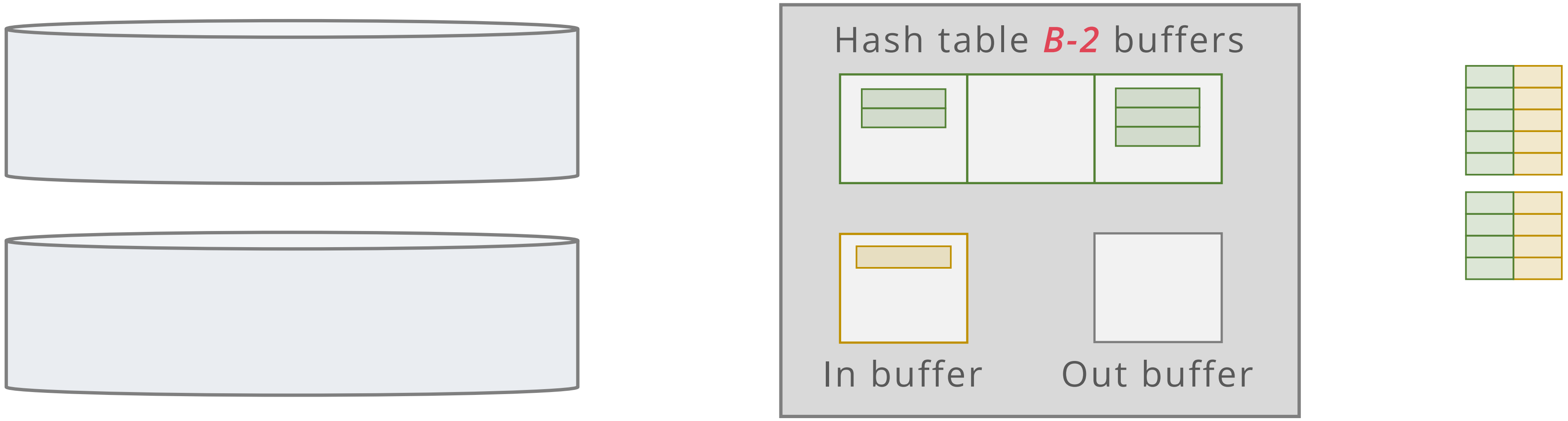
No match

Grace Hash Join: Build & Probe

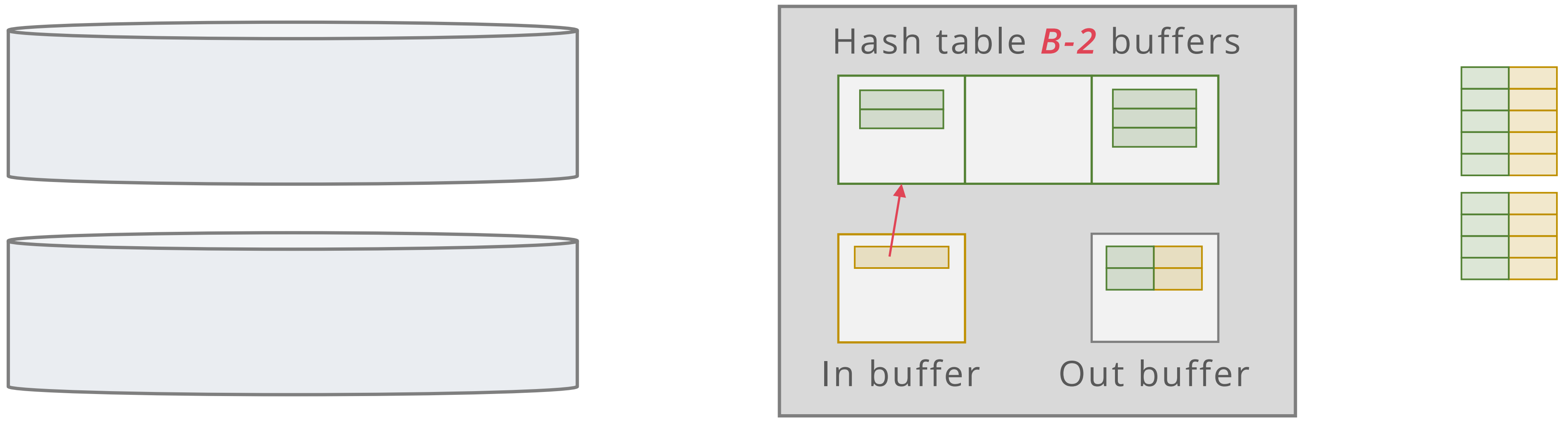


1 match

GRACE HASH JOIN: BUILD & PROBE

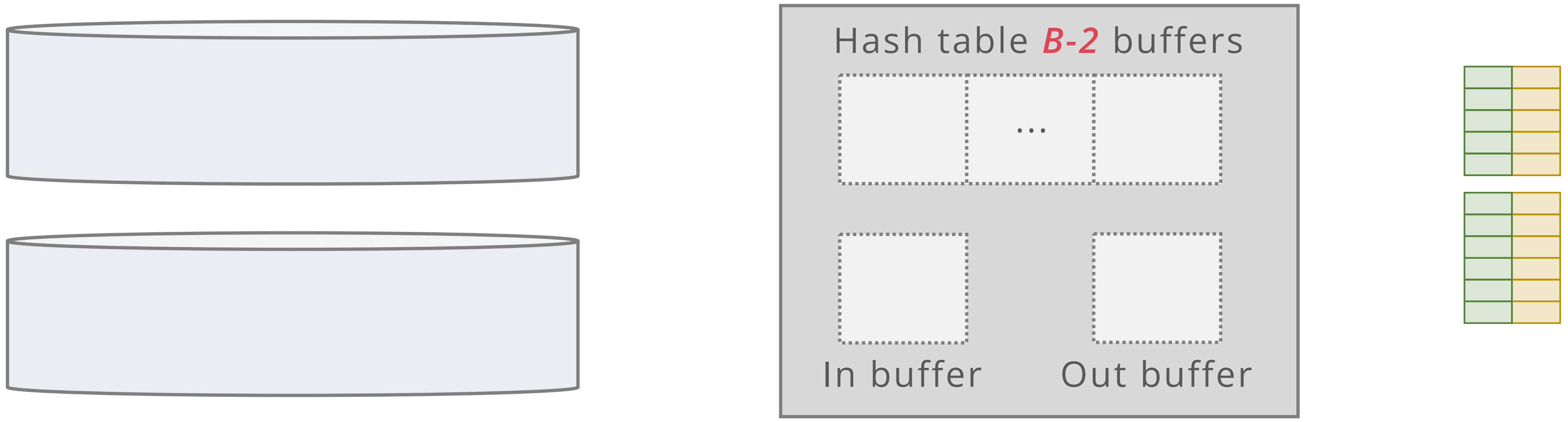


GRACE HASH JOIN: BUILD & PROBE



2 matches

Grace Hash Join: Build & Probe



University of Edinburgh
School of Informatics
INFR11199 - Advanced Database Systems (Spring 2024)

Tutorial Sheet 4 - Playing with PostgreSQL

The purpose of this practical sheet is to familiarise you with the query execution engine of PostgreSQL. In particular, you will analyse a few queries and answer questions regarding their performance when turning different knobs of the execution engine. To answer the questions, you might find the following documentation links useful:

- Documentation of `EXPLAIN ANALYZE`:
  `https://www.postgresql.org/docs/14/sql-explain.html`.
- Making sense of the `EXPLAIN ANALYZE` output:
  `https://www.postgresql.org/docs/14/performance-tips.html`.
- PostgreSQL query planner documentation:
  `https://www.postgresql.org/docs/14/runtime-config-query.html`.
- How to create an index:
  `https://www.postgresql.org/docs/14/sql-createindex.html`.
- The system table `pg_class`:
  `https://www.postgresql.org/docs/current/catalog-pg-class.html`.

Prerequisites:

- Install PostgreSQL on your machine and start a PostgreSQL server (plenty of instructions online on how to do this, e.g., `http://postgresguide.com/setup/install.html`; any version will work). Make sure the command-line tool `psql` is working and you can use it to create tables and run queries.
- Download the bay-area-bike-sharing dataset from the course webpage. Unzip the archive and import the data into PostgreSQL using the provided scripts (e.g., by typing the command `psql < import.sql`).

1. **EXPLAIN and ANALYZE**

   For the following questions consider the query below:

   ```
   SELECT * FROM trip WHERE bike_id = 10;
   ```

   (a) Provide the PostgreSQL execution plan of the query and the SQL statement you use to generate the result.

   Based on the execution plan:

   i. What was the estimated cost (in arbitrary units)? ____________
   ii. What was the total runtime (in ms)? ____________
   iii. What was the estimated number of tuples to be output? ____________
   iv. What was the actual number of output tuples? ____________

(b) Create an index on the attribute `bike_id` on the table `trip`. Provide the SQL statement for that and the new execution plan of the query.

Based on the execution plan:

i. What was the estimated cost (in arbitrary units)? Faster than without index ______

ii. What was the total runtime (in ms)? ______

(c) Use the table `pg_class` to answer the following questions.

i. How many pages are used to store the index you created on table `trip`? Provide the answer and the query you use to generate the answer.

ii. How many tuples are in the index you created on column `bike_id`? Provide the answer and the query you use to generate the answer.

Without running ANALYZE, the pg_class does not have info about new deletions in the tuples

iii. How many tuples are in the table `weather`, according to `pg_class`?

iv. In the table `weather`, delete all records of which date is earlier than '2013-10-01'. Provide the SQL statement you use.

v. After deletion, rerun your query from step 3. Is the new result equal to the result of running `SELECT COUNT(*) FROM weather`?

◯ Yes ◯ No

vi. `ANALYZE` is a Postgres function used to collect statistics about a database. You want to use it especially after considerable number of modifications happen to that database. Run `ANALYZE`, and then rerun your query from step 3 again. Is the new result equal to the result of running `SELECT COUNT(*) FROM weather`?

◯ Yes ◯ No

2. **Using Indexes**

In this question, we will learn the conditions under which indexes may or may not be used by the query optimizer.

(a) Create an index on the column `start_station_name` on the table `trip`. Provide the SQL command you use.

(b) For each of those queries, answer Yes if the index you created on `trip.start_station_name` was used in the execution plan, or No otherwise:

i.
```
SELECT * FROM trip
 WHERE start_station_name like 'San';
```

◯ Yes ◯ No

ii.
```
SELECT * FROM trip
 WHERE start_station_name like '%San';
```

◯ Yes ◯ No

iii.
```
SELECT * FROM trip
 WHERE start_station_name BETWEEN 'San Francisco' AND 'San Jose'
   AND end_station_name > 'San';
```

◯ Yes ◯ No

iv.
```
SELECT * FROM trip
 WHERE start_station_name BETWEEN 'San␣Francisco' AND 'San␣Jose'
    OR end_station_name > 'San';
```

◯ Yes ◯ No

(c) Make sure you still have an index on the column `trip.bike_id` (you can verify this using `\di` in `psql`). For each of those queries, answer which indexes are used in their execution plans.

i.
```
SELECT * FROM trip
 WHERE start_station_name BETWEEN 'San␣Francisco' AND 'San␣Jose'
   AND bike_id < 10;
```

- ◯ Only the index on `start_station_name` was used.
- ◯ Only the index on `bike_id` was used.
- ◯ Both indexes were used.
- ◯ None of the indexes were used.

ii.
```
SELECT * FROM trip
 WHERE start_station_name BETWEEN 'San␣Francisco' AND 'San␣Jose'
   AND bike_id < 500;
```

- ◯ Only the index on `start_station_name` was used.
- ◯ Only the index on `bike_id` was used.
- ◯ Both indexes were used.
- ◯ None of the indexes were used.

iii.
```
SELECT * FROM trip
 WHERE start_station_name BETWEEN 'San␣Francisco' AND 'San␣Jose'
   AND bike_id BETWEEN 500 AND 510;
```

- ◯ Only the index on `start_station_name` was used.
- ◯ Only the index on `bike_id` was used.
- ◯ Both indexes were used.
- ◯ None of the indexes were used.

iv.
```
SELECT * FROM trip
 WHERE start_station_name > 'San␣Francisco'
   AND bike_id < 500;
```

- ◯ Only the index on `start_station_name` was used.
- ◯ Only the index on `bike_id` was used.
- ◯ Both indexes were used.
- ◯ None of the indexes were used.

(d) Answer the questions below for the query:

```
SELECT * FROM trip
 WHERE bike_id BETWEEN 10 AND 20;
```

i. Was the index on `bike_id` used? ◯ Yes ◯ No

ii. What percentage of the total records in the table `trip` was returned? Provide a percent and retain two significant figures. __________

(e) Answer the questions below for the query:

```
SELECT * FROM trip
 WHERE bike_id > 10;
```

i. Was the index on `bike_id` used? ◯ Yes ◯ No

ii. What percentage of the total records in the table `trip` was returned? Provide a percent and retain two significant figures. __________

(f) Answer the questions below for the query:

```
SELECT * FROM trip
 WHERE bike_id > 10 ORDER BY start_time;
```

i. Which method was used for sorting? __________
ii. Where did the sorting happen? ◯ Memory ◯ Disk
iii. How much space was used for sorting? __________
iv. What was the total runtime (in ms)? __________

(g) Display PostgreSQL working memory with `SHOW work_mem;`. Increase PostgreSQL working memory with the command `SET work_mem = '128MB';`. For the same query from part vi., answer the following questions:

i. Which method was used for sorting? __________
ii. Where did the sorting happen? ◯ Memory ◯ Disk
iii. How much space was used for sorting? __________
iv. What was the total runtime (in ms)? __________

(h) Execute the command `RESET work_mem;` to get PostgreSQL working memory back to the default value (or your answers for the next questions will turn out wrong).

3. **Joins**

In this question, we will learn about different methods used by PostgreSQL for executing joins. Make sure you reset `work_mem` to its default value (i.e., `RESET work_mem;`).

Answer the following questions based on the query below:

```
SELECT trip.*, station.city
  FROM trip, station
 WHERE trip.start_station_id = station.station_id
   AND bike_id < 200;
```

(a) Provide the query plan for the above query.

Based on the execution plan:

i. Which join method was used? ____________

ii. What was the estimated cost (in arbitrary units)? ____________

iii. What was the total runtime (in ms)? ____________

(b) Execute the command `SET enable_hashjoin = false;` to disable hash joins. Provide the new query plan.

Based on the execution plan:

i. Which join method was used? ____________

ii. What was the estimated cost (in arbitrary units)? ____________

iii. What was the total runtime (in ms)? ____________

(c) Execute the command `SET enable_mergejoin = false;` to disable merge joins. Provide the new query plan.

Based on the execution plan:

i. Which join method was used? ____________

ii. What was the estimated cost (in arbitrary units)? ____________

iii. What was the total runtime (in ms)? ____________

(d) Execute the command `SET enable_indexscan = false; SET enable_bitmapscan = false;` to disable index scans. Give the new plan.

Based on the execution plan:

i. Which join method was used? ____________

ii. What was the estimated cost (in arbitrary units)? ____________

iii. What was the total runtime (in ms)? ____________

(e) Execute these commands to re-enable the different joins.

```
RESET enable_mergejoin;
RESET enable_hashjoin;
RESET enable_indexscan;
RESET enable_bitmapscan;
```

University of Edinburgh
School of Informatics
INFR11199 - Advanced Database Systems (Spring 2024)

Tutorial Sheet 4 - Playing with PostgreSQL

The purpose of this practical sheet is to familiarise you with the query execution engine of PostgreSQL. In particular, you will analyse a few queries and answer questions regarding their performance when turning different knobs of the execution engine. To answer the questions, you might find the following documentation links useful:

- Documentation of `EXPLAIN ANALYZE`:
  `https://www.postgresql.org/docs/14/sql-explain.html`.
- Making sense of the `EXPLAIN ANALYZE` output:
  `https://www.postgresql.org/docs/14/performance-tips.html`.
- PostgreSQL query planner documentation:
  `https://www.postgresql.org/docs/14/runtime-config-query.html`.
- How to create an index:
  `https://www.postgresql.org/docs/14/sql-createindex.html`.
- The system table `pg_class`:
  `https://www.postgresql.org/docs/current/catalog-pg-class.html`.

Prerequisites:

- Install PostgreSQL on your machine and start a PostgreSQL server (plenty of instructions online on how to do this, e.g., `http://postgresguide.com/setup/install.html`; any version will work). Make sure the command-line tool `psql` is working and you can use it to create tables and run queries.
- Download the bay-area-bike-sharing dataset from the course webpage. Unzip the archive and import the data into PostgreSQL using the provided scripts (e.g., by typing the command `psql < import.sql`).

1. **EXPLAIN and ANALYZE**

   For the following questions consider the query below:

```
SELECT * FROM trip WHERE bike_id = 10;
```

   (a) Provide the PostgreSQL execution plan of the query and the SQL statement you use to generate the result.

Solution:

```
EXPLAIN ANALYZE SELECT * FROM trip WHERE bike_id = 10;

QUERY PLAN
------------------------------------------------------------------------
Gather  (cost=1000.00..14191.37 rows=790 width=80) (actual time
    =2.556..143.164 rows=248 loops=1)
   Workers Planned: 2
   Workers Launched: 2
   ->  Parallel Seq Scan on trip  (cost=0.00..13112.37 rows=329 width=80)
       (actual time=4.328..127.060 rows=83 loops=3)
         Filter: (bike_id = 10)
         Rows Removed by Filter: 223237
 Planning Time: 0.824 ms
 Execution Time: 143.286 ms
```

Based on the execution plan:

i. What was the estimated cost (in arbitrary units)?	**14191.87**
ii. What was the total runtime (in ms)?	**143.286**
iii. What was the estimated number of tuples to be output?	**790**
iv. What was the actual number of output tuples?	**248**

(b) Create an index on the attribute `bike_id` on the table `trip`. Provide the SQL statement for that and the new execution plan of the query.

Solution:

```
CREATE INDEX idx_bike_id ON trip(bike_id);

QUERY PLAN
-----------------------------------------------------------------------------
 Bitmap Heap Scan on trip  (cost=18.55..2424.94 rows=790 width=80) (actual
      time=0.432..1.039 rows=248 loops=1)
   Recheck Cond: (bike_id = 10)
   Heap Blocks: exact=234
   ->  Bitmap Index Scan on idx_bike_id  (cost=0.00..18.35 rows=790 width
       =0) (actual time=0.335..0.335 rows=248 loops=1)
         Index Cond: (bike_id = 10)
 Planning Time: 6.332 ms
 Execution Time: 1.154 ms
```

Based on the execution plan:

i. What was the estimated cost (in arbitrary units)? **2424.94**

ii. What was the total runtime (in ms)? **1.154**

(c) Use the table `pg_class` to answer the following questions.

i. How many pages are used to store the index you created on table `trip`? Provide the answer and the query you use to generate the answer.

Solution:

```
SELECT relpages FROM pg_class
WHERE relname = 'idx_bike_id';

relpages
----------
     1840
```

ii. How many tuples are in the index you created on column `bike_id`? Provide the answer and the query you use to generate the answer.

Solution:

```
SELECT reltuples FROM pg_class
WHERE relname = 'idx_bike_id';

 reltuples
-----------
    669959
```

iii. How many tuples are in the table `weather`, according to `pg_class`?

Solution:

```
SELECT reltuples FROM pg_class
WHERE relname = 'weather';

 reltuples
-----------
      3665
```

iv. In the table `weather`, delete all records of which date is earlier than '2013-10-01'. Provide the SQL statement you use.

Solution:

```
DELETE FROM weather WHERE date < '2013-10-01';
```

v. After deletion, rerun your query from step 3. Is the new result equal to the result of running `SELECT COUNT(*) FROM weather`?

○ Yes [✓ **No**]

vi. `ANALYZE` is a Postgres function used to collect statistics about a database. You want to use it especially after considerable number of modifications happen to that database. Run `ANALYZE`, and then rerun your query from step 3 again. Is the new result equal to the result of running `SELECT COUNT(*) FROM weather`?

[✓ **Yes**] ○ No

2. **Using Indexes**

In this question, we will learn the conditions under which indexes may or may not be used by the query optimizer.

(a) Create an index on the column `start_station_name` on the table `trip`. Provide the SQL command you use.

Solution:

```
CREATE INDEX idx_start_sta_name ON trip ( start_station_name );
```

(b) For each of those queries, answer Yes if the index you created on `trip.start_station_name` was used in the execution plan, or No otherwise:

i.
```
SELECT * FROM trip
 WHERE start_station_name like 'San';
```

[✓ **Yes**] ◯ No

ii.
```
SELECT * FROM trip
  WHERE start_station_name like '%San';
```

◯ Yes [✓ **No**]

iii.
```
SELECT * FROM trip
  WHERE start_station_name BETWEEN 'San␣Francisco' AND 'San␣Jose'
    AND end_station_name > 'San';
```

[✓ **Yes**] ◯ No

iv.
```
SELECT * FROM trip
  WHERE start_station_name BETWEEN 'San␣Francisco' AND 'San␣Jose'
     OR end_station_name > 'San';
```

◯ Yes [✓ **No**]

(c) Make sure you still have an index on the column `trip.bike_id` (you can verify this using `\di` in `psql`). For each of those queries, answer which indexes are used in their execution plans.

i.
```
SELECT * FROM trip
  WHERE start_station_name BETWEEN 'San␣Francisco' AND 'San␣Jose'
    AND bike_id < 10;
```

◯ Only the index on `start_station_name` was used.
[✓ **Only the index on `bike_id` was used.**]
◯ Both indexes were used.
◯ None of the indexes were used.

ii.
```
SELECT * FROM trip
  WHERE start_station_name BETWEEN 'San␣Francisco' AND 'San␣Jose'
    AND bike_id < 500;
```

[✓ **Only the index on `start_station_name` was used.**]
◯ Only the index on `bike_id` was used.
◯ Both indexes were used.
◯ None of the indexes were used.

iii.
```
SELECT * FROM trip
  WHERE start_station_name BETWEEN 'San␣Francisco' AND 'San␣Jose'
    AND bike_id BETWEEN 500 AND 510;
```

◯ Only the index on `start_station_name` was used.
◯ Only the index on `bike_id` was used.
[✓ **Both indexes were used.**]
◯ None of the indexes were used.

iv.
```
SELECT * FROM trip
  WHERE start_station_name > 'San␣Francisco'
    AND bike_id < 500;
```

◯ Only the index on start_station_name was used.
◯ Only the index on bike_id was used.
◯ Both indexes were used.
[✓ **None of the indexes were used.**]

(d) Answer the questions below for the query:

```sql
SELECT * FROM trip
 WHERE bike_id BETWEEN 10 AND 20;
```

i. Was the index on bike_id used? [✓ **Yes**] ◯ No

ii. What percentage of the total records in the table trip was returned? Provide a percent and retain two significant figures. 0.54% (3594 **out of** 669959)

(e) Answer the questions below for the query:

```sql
SELECT * FROM trip
 WHERE bike_id > 10;
```

i. Was the index on bike_id used? ◯ Yes [✓ **No**]

ii. What percentage of the total records in the table trip was returned? Provide a percent and retain two significant figures. 99.92% (669460 **out of** 669959)

If buffer memory more than relation, then in memory sorting can be used, else external merge sort has to be used

(f) Answer the questions below for the query:

```sql
SELECT * FROM trip
 WHERE bike_id > 10 ORDER BY start_time;
```

i. Which method was used for sorting? **external merge sort**
ii. Where did the sorting happen? ◯ Memory [✓ **Disk**]
iii. How much space was used for sorting? 22080**kB per worker**
iv. What was the total runtime (in ms)? 891.222 **ms (any number)**

(g) Display PostgreSQL working memory with `SHOW work_mem;`. Increase PostgreSQL working memory with the command `SET work_mem = '128MB';`. For the same query from part vi., answer the following questions:

i. Which method was used for sorting? **quick sort**
ii. Where did the sorting happen? [✓ **Memory**] ◯ Disk
iii. How much space was used for sorting? 115, 909**kB**
iv. What was the total runtime (in ms)? 454.680 **ms (any number)**

(h) Execute the command `RESET work_mem;` to get PostgreSQL working memory back to the default value (or your answers for the next questions will turn out wrong).

3. **Joins**

In this question, we will learn about different methods used by PostgreSQL for executing joins. Make sure you reset `work_mem` to its default value (i.e., `RESET work_mem;`).

Answer the following questions based on the query below:

```sql
SELECT trip.*, station.city
  FROM trip, station
 WHERE trip.start_station_id = station.station_id
   AND bike_id < 200;
```

(a) Provide the query plan for the above query.

Solution:

```
QUERY PLAN
------------------------------------------------------------------------
 Hash Join  (cost=1093.33..11604.43 rows=58107 width=92) (actual time
     =29.274..85.848 rows=58161 loops=1)
   Hash Cond: (trip.start_station_id = station.station_id)
   ->  Bitmap Heap Scan on trip  (cost=1090.75..11440.09 rows=58107 width
       =80) (actual time=29.108..52.497 rows=58161 loops=1)
         Recheck Cond: (bike_id < 200)
         Heap Blocks: exact=9541
         ->  Bitmap Index Scan on idx_bike_id  (cost=0.00..1076.23 rows
             =58107 width=0) (actual time=25.510..25.510 rows=58161 loops
             =1)
               Index Cond: (bike_id < 200)
   ->  Hash  (cost=1.70..1.70 rows=70 width=14) (actual time=0.126..0.126
       rows=70 loops=1)
         Buckets: 1024  Batches: 1  Memory Usage: 12kB
         ->  Seq Scan on station  (cost=0.00..1.70 rows=70 width=14) (
             actual time=0.023..0.068 rows=70 loops=1)
 Planning Time: 0.900 ms
 Execution Time: 89.936 ms
```

Based on the execution plan:

i. Which join method was used? **Hash join**

ii. What was the estimated cost (in arbitrary units)? **11604.43**

iii. What was the total runtime (in ms)? **89.936**

(b) Execute the command `SET enable_hashjoin = false;` to disable hash joins. Provide the new query plan.

Solution:

```
QUERY PLAN
------------------------------------------------------------------------
 Merge Join  (cost=18625.07..19497.02 rows=58107 width=92) (actual time
     =83.404..124.602 rows=58161 loops=1)
   Merge Cond: (trip.start_station_id = station.station_id)
   ->  Sort  (cost=18621.22..18766.49 rows=58107 width=80) (actual time
       =83.322..99.226 rows=58161 loops=1)
```

```
         Sort Key: trip.start_station_id
         Sort Method: external merge  Disk: 5400kB
         ->  Bitmap Heap Scan on trip  (cost=1090.75..11440.09 rows=58107
             width=80) (actual time=15.734..39.091 rows=58161 loops=1)
               Recheck Cond: (bike_id < 200)
               Heap Blocks: exact=9541
               ->  Bitmap Index Scan on idx_bike_id  (cost=0.00..1076.23
                   rows=58107 width=0) (actual time=13.722..13.722 rows
                   =58161 loops=1)
                     Index Cond: (bike_id < 200)
   ->  Sort  (cost=3.85..4.02 rows=70 width=14) (actual time=0.075..0.084
       rows=70 loops=1)
         Sort Key: station.station_id
         Sort Method: quicksort  Memory: 28kB
         ->  Seq Scan on station  (cost=0.00..1.70 rows=70 width=14) (
             actual time=0.016..0.033 rows=70 loops=1)
 Planning Time: 0.642 ms
 Execution Time: 134.899 ms
```

Based on the execution plan:

i. Which join method was used? **Sort-merge join**

ii. What was the estimated cost (in arbitrary units)? **19497.02**

iii. What was the total runtime (in ms)? **134.899**

(c) Execute the command `SET enable_mergejoin = false;` to disable merge joins. Provide the new query plan.

Solution:

```
QUERY PLAN
-----------------------------------------------------------------------
 Nested Loop  (cost=1090.90..20735.16 rows=58107 width=92) (actual time
     =19.123..179.892 rows=58161 loops=1)
   ->  Bitmap Heap Scan on trip  (cost=1090.75..11440.09 rows=58107 width
       =80) (actual time=19.095..43.152 rows=58161 loops=1)
         Recheck Cond: (bike_id < 200)
         Heap Blocks: exact=9541
         ->  Bitmap Index Scan on idx_bike_id  (cost=0.00..1076.23 rows
             =58107 width=0) (actual time=16.637..16.638 rows=58161 loops
             =1)
               Index Cond: (bike_id < 200)
   ->  Index Scan using station_pkey on station  (cost=0.14..0.16 rows=1
       width=14) (actual time=0.002..0.002 rows=1 loops=58161)
         Index Cond: (station_id = trip.start_station_id)
 Planning Time: 0.384 ms
 Execution Time: 185.202 ms
```

Based on the execution plan:

i. Which join method was used? **Nested loops join w/ index scans**

ii. What was the estimated cost (in arbitrary units)? **20735.16**

iii. What was the total runtime (in ms)? **185.202**

(d) Execute the command `SET enable_indexscan = false; SET enable_bitmapscan = false;` to disable index scans. Give the new plan.

Solution:

```
QUERY PLAN
--------------------------------------------------------------------------
 Nested Loop  (cost=0.00..78164.76 rows=58107 width=92) (actual time
     =0.101..595.171 rows=58161 loops=1)
   Join Filter: (trip.start_station_id = station.station_id)
   Rows Removed by Join Filter: 1637515
   ->  Seq Scan on trip  (cost=0.00..17997.49 rows=58107 width=80) (actual
        time=0.039..179.165 rows=58161 loops=1)
         Filter: (bike_id < 200)
         Rows Removed by Filter: 611798
   ->  Materialize  (cost=0.00..2.05 rows=70 width=14) (actual time
       =0.000..0.002 rows=29 loops=58161)
         ->  Seq Scan on station  (cost=0.00..1.70 rows=70 width=14) (
             actual time=0.016..0.048 rows=70 loops=1)
 Planning Time: 0.460 ms
 Execution Time: 599.250 ms
```

Based on the execution plan:

i. Which join method was used? **Nested loops join**

ii. What was the estimated cost (in arbitrary units)? **78164.76**

iii. What was the total runtime (in ms)? **599.250**

(e) Execute these commands to re-enable the different joins.

```
RESET enable_mergejoin;
RESET enable_hashjoin;
RESET enable_indexscan;
RESET enable_bitmapscan;
```

University of Edinburgh
School of Informatics
INFR11199 - Advanced Database Systems (Spring 2024)

Tutorial Sheet 5

1. (Transactions & Concurrency) Consider a database with objects X and Y and two transactions. Transaction 1 reads X and Y and then writes X and Y. Transaction 2 reads and writes X then reads and writes Y.

   (a) Create a schedule for these transactions that is *not* serializable. Explain why your schedule is not serializable.

   (b) Would your schedule be allowed under strict two-phase locking? Why or why not?

   Now consider the following schedule:

	1	2	3	4	5	6	7	8
T1				X(B)	X(A)			S(D)
T2		X(D)	S(E)					
T3	X(E)						S(B)	
T4						X(A)		

   (c) Draw the waits-for graph for this schedule.

   (d) Are there any transactions involved in deadlock?

   (e) Next, assume that T1 priority > T2 > T3 > T4. You are the database administrator and one of your users purchases an exclusive plan to ensure that their transaction, T2, runs to completion. Assuming the same schedule, what deadlock avoidance policy would you choose to make sure T2 commits?

2. (Parallel Query Processing) For each of the following scenarios, state whether it is an example of:

   - Inter-query parallelism
   - Intra-query, inter-operator parallelism
   - Intra-query, intra-operator parallelism

- No parallelism

(a) A query with a selection, followed by a projection, followed by a join, runs on a single machine with one thread.

(b) Same as before, but there is a second machine and a second query, running independently of the first machine and the first query.

(c) A query with a selection, followed by a projection, runs on a single machine with multiple threads; one thread is given to the selection and one thread is given to the projection.

(d) We have a single machine, and it runs recursive hash partitioning (for external hashing) with one thread.

(e) We have a multi-machine database, and we are running a join over it. For the join, we are running parallel sort-merge join.

3. (Parallel Query Processing) Suppose we have 4 machines, each with 10 buffer pages. Machine 1 has a `Students` table which consists of 100 pages. Each page is 1 KB, and it takes 1 second to send 1 KB of data across the network to another machine.

   (a) How long would it take to send the data over the network after we uniformly range partition the 100 pages? Assume that we can send data to multiple machines at the same time.

   (b) Next, imagine that there is another table, `Classes`, which is 10 pages. Using just one machine, how long would a BNLJ take if each disk access (read or write) takes 0.5 seconds?

   (c) Now assume that the `Students` table has already been uniformly range partitioned across the four machines, but `Classes` is only on Machine 1. How long would a broadcast join take if we perform BNLJ on each machine? Do not worry about the cost of combining the output of the machines.

   (d) Which algorithm performs better?

   (e) Knowing that the `Students` table was range partitioned, how can we improve the performance of the join even further?

4. (Two-Phase Commit with Logging) Suppose we have one coordinator and three participants. It takes 30ms for a coordinator to send messages to all participants; 5, 10, and 15ms for participant 1, 2, and 3 to send a message to the coordinator respectively; and 10ms for each machine to generate and flush a record. Assume that for the same message, each participant receives it from the coordinator at the same time.

   (a) Under 2PC with logging, how long does the whole 2PC process (from the beginning to the coordinator's final log flush) take for a successful commit in the best case?

(b) Now in the 2PC protocol, describe what happens if a participant receives a PREPARE message, replies with a YES vote, crashes, and restarts (All other participants also voted YES and didn't crash).

(c) In the 2PC protocol, suppose the coordinator sends PREPARE message to Participants 1 and 2. Participant 1 sends a "VOTE YES" message, and Participant 2 sends a "VOTE NO" message back to the coordinator.

   i. Before receiving the result of the commit/abort vote from the coordinator, Participant 1 crashes. Upon recovery, what actions does Participant 1 take (1) if we were not using presumed abort, and (2) if we were using presumed abort?

   ii. Before receiving the result of the commit/abort vote from the coordinator, Participant 2 crashes. Upon recovery, what actions does Participant 2 take (1) if we were not using presumed abort, and (2) if we were using presumed abort?

(d) In the 2PC protocol, suppose that the coordinator sends PREPARE messages to the participants and crashes before receiving any votes from the participants. Assuming that we are running 2PC with presumed abort, answer the following questions.

   i. What sequence of operations does the coordinator take after it recovers?

   ii. What sequence of operations does a participant who received the message and replied NO before the coordinator crashed take?

   iii. What sequence of operations does a participant who received the message and replied YES before the coordinator crashed take?

   iv. Let's say that the coordinator instead crashes after successfully receiving votes from all participants, with all participants voting YES except for one NO vote. Assuming the coordinator sees no records for this transaction in its log after coming back online, how does this affect the answers to parts (d).i-(d).iii.?

University of Edinburgh
School of Informatics
INFR11199 - Advanced Database Systems (Spring 2024)

Tutorial Sheet 5

1. (Transactions & Concurrency) Consider a database with objects X and Y and two transactions. Transaction 1 reads X and Y and then writes X and Y. Transaction 2 reads and writes X then reads and writes Y.

   (a) Create a schedule for these transactions that is *not* serializable. Explain why your schedule is not serializable.

   Solution: Here is an example of a schedule that is not serializable.

Transaction 1	Transaction 2
Read(X)	
Read(Y)	
	Read(X)
	Write(X)
	Read(Y)
Write(X)	
Write(Y)	
	Write(Y)

   In this example, T1 reads X before T2 writes X. However, T1 writes X after T2 reads/writes it. The schedule is thus not serializable since there is no equivalent serial schedule that would have the same result (neither T1-T2 nor T2-T1).

   (b) Would your schedule be allowed under strict two-phase locking? Why or why not?

   Solution: No, because strict 2PL ensures serializability. Keep in mind that strict 2PL only allows releasing locks at the end of a transaction. In the example schedule shown above, when Transaction 2 attempts to acquire an exclusive lock to write X, it will have to wait for Transaction 1 to release its lock on X, which will not happen until Transaction 1 commits. This will never happen, so this schedule is not possible under strict 2PL.

Now consider the following schedule:

	1	2	3	4	5	6	7	8
T1				X(B)	X(A)			S(D)
T2		X(D)	S(E)					
T3	X(E)						S(B)	
T4						X(A)		

(c) Draw the waits-for graph for this schedule.

Solution: The waits-for graph consists of the following edges:

- T1 $\rightarrow$ T2
- T2 $\rightarrow$ T3
- T3 $\rightarrow$ T1
- T4 $\rightarrow$ T1

(d) Are there any transactions involved in deadlock?

Solution: Yes, Transactions 1, 2, and 3 are deadlocked.

(e) Next, assume that T1 priority > T2 > T3 > T4. You are the database administrator and one of your users purchases an exclusive plan to ensure that their transaction, T2, runs to completion. Assuming the same schedule, what deadlock avoidance policy would you choose to make sure T2 commits?

Solution: Choose wait-die. Under wait-die, T3 and T4 abort after steps 6 and 7 because they are attempting to acquire a lock held by a transaction with higher priority. Afterwards, both T1 and T2 run to completion.

However, under wound-wait, T3 will be killed by T2 at step 3, and T2 will be killed by T1 at step 8. Since we want to make sure T2 commits, we should choose wait-die.

2. (Parallel Query Processing) For each of the following scenarios, state whether it is an example of:

- Inter-query parallelism
- Intra-query, inter-operator parallelism
- Intra-query, intra-operator parallelism
- No parallelism

(a) A query with a selection, followed by a projection, followed by a join, runs on a single machine with one thread.

> **Solution:** No parallelism. It might look like a pipeline, but at any given point in time there is only one thing happening, since there is only one thread.

(b) Same as before, but there is a second machine and a second query, running independently of the first machine and the first query.

> **Solution:** Inter-query parallelism.

(c) A query with a selection, followed by a projection, runs on a single machine with multiple threads; one thread is given to the selection and one thread is given to the projection.

> **Solution:** Intra-query, inter-operator parallelism.

(d) We have a single machine, and it runs recursive hash partitioning (for external hashing) with one thread.

> **Solution:** No parallelism, because there is only one machine and one thread. Don't confuse this with parallel hashing!

(e) We have a multi-machine database, and we are running a join over it. For the join, we are running parallel sort-merge join.

> **Solution:** Intra-query, intra-operator parallelism. We have a single query and a single operator, but that single operator is going to do multiple things at the same time (across different machines).

3. (Parallel Query Processing) Suppose we have 4 machines, each with 10 buffer pages. Machine 1 has a `Students` table which consists of 100 pages. Each page is 1 KB, and it takes 1 second to send 1 KB of data across the network to another machine.

(a) How long would it take to send the data over the network after we uniformly range partition the 100 pages? Assume that we can send data to multiple machines at the same time.

Solution: 25 seconds.
After we uniformly partition our data, Machine 1 will send 25 pages to Machines 2, 3, and 4. It will take 25 seconds to finish sending these pages to each machine if we send the pages to each machine at the same time.

(b) Next, imagine that there is another table, `Classes`, which is 10 pages. Using just one machine, how long would a BNLJ take if each disk access (read or write) takes 0.5 seconds?

Solution: 105 seconds.
BNLJ will require $10 + \lceil 10/8 \rceil \cdot 100 = 210$ I/Os, which takes 105 seconds.

(c) Now assume that the `Students` table has already been uniformly range partitioned across the four machines, but `Classes` is only on Machine 1. How long would a broadcast join take if we perform BNLJ on each machine? Do not worry about the cost of combining the output of the machines.

Solution: 40 seconds.
First, we must broadcast the `Classes` table to each machine, which will take 10 seconds (since we send the table to each machine at the same time). Next, we will perform BNLJ on each machine, which requires $10 + \lceil 10/8 \rceil \cdot 25 = 60$ I/Os, or 30 seconds. In total, the time required to send the data over the network and perform the join will be 40 seconds.

(d) Which algorithm performs better?

Solution: Broadcasting the `Classes` table and performing a parallel BNLJ runs faster than just using one machine, even with the additional time required to send the table over the network.

(e) Knowing that the `Students` table was range partitioned, how can we improve the performance of the join even further?

Solution: Since we know the `Students` table was range partitioned, we can also range partition the `Classes` table on the same column and only send the corresponding partitions to each machine. This should lower the network cost and decrease the number of disk I/Os.

4. (Two-Phase Commit with Logging) Suppose we have one coordinator and three participants. It takes 30ms for a coordinator to send messages to all participants; 5, 10, and 15ms for participant 1, 2, and 3 to send a message to the coordinator respectively; and 10ms for each machine to generate and flush a record. Assume that for the same message, each participant receives it from the coordinator at the same time.

   (a) Under 2PC with logging, how long does the whole 2PC process (from the beginning to the coordinator's final log flush) take for a successful commit in the best case?

   Solution: 130ms.
   Phase 1: 30 + 10 + 15 + 10 = 65ms
   Coordinator sends prepare message + Participant generates and flushes prepare record + max time it takes a participant to send a Yes message + Coordinator generates and flushes commit record
   Phase 2: 30 + 10 + 15 + 10 = 65ms
   Coordinator sends commit message + Participant generates and flushes commit record + max time it takes a participant to send an ACK message + Coordinator generates and flushes end record
   Total: 130ms

   (b) Now in the 2PC protocol, describe what happens if a participant receives a PREPARE message, replies with a YES vote, crashes, and restarts (All other participants also voted YES and didn't crash).

   Solution: In this scenario, upon restarting, the recovery process will find that the participant is in the prepared state for the transaction. It will periodically try to contact the coordinator site to find out how the transaction should be resolved. In our scenario, the final outcome of the transaction is a commit. Because the coordinator cannot end a committed transaction until it receives final ACKs from all nodes, it will correctly respond with a COMMIT message to the inquiry. The participant will then be able to properly commit the transaction. It will write (and flush) a COMMIT record and send an ACK message to the coordinator.

   (c) In the 2PC protocol, suppose the coordinator sends PREPARE message to Participants 1 and 2. Participant 1 sends a "VOTE YES" message, and Participant 2 sends a "VOTE NO" message back to the coordinator.

      i. Before receiving the result of the commit/abort vote from the coordinator, Participant 1 crashes. Upon recovery, what actions does Participant 1 take (1) if we were not using presumed abort, and (2) if we were using presumed abort?

Solution: Upon recovery, Participant 1 will find a PREPARE message for this transaction in their log. In both cases, Participant 1 sends an inquiry to the coordinator for the status of this transaction and receives a message that the transaction was aborted. Participant 1 will abort the transaction locally.

ii. Before receiving the result of the commit/abort vote from the coordinator, Participant 2 crashes. Upon recovery, what actions does Participant 2 take (1) if we were not using presumed abort, and (2) if we were using presumed abort?

Solution: If running 2PC without presumed abort, Participant 2 sees a (flushed) abort record for this transaction in their log and sends the "VOTE NO" message back to the coordinator.
If running 2PC with presumed abort, the abort record may not have reached disk before crashing, thus it's missing upon recovery. Participant 2 will abort the transaction locally, without sending any messages to the coordinator.

(d) In the 2PC protocol, suppose that the coordinator sends PREPARE messages to the participants and crashes before receiving any votes from the participants. Assuming that we are running 2PC with presumed abort, answer the following questions.

i. What sequence of operations does the coordinator take after it recovers?

Since no log, recovery process should step in

Solution: After the coordinator restarts, the recovery process will find that a transaction was executing at the time of the crash and that no commit log record had been written (remember that the coordinator does not write any log records before sending PREPARE messages). The recovery process will abort the transaction locally (not sending an ABORT message) by undoing its actions, if any, and writing an abort record.

ii. What sequence of operations does a participant who received the message and replied NO before the coordinator crashed take?

Solution: If the participant sent a NO vote, it knows that the transaction will be aborted because a NO vote acts like a veto. The participant does not care if the coordinator crashes or not. It aborts the local effects of the transaction.

iii. What sequence of operations does a participant who received the message and replied YES before the coordinator crashed take?

Solution: If the participant sent a YES vote, it cannot make any unilateral decisions. If the participant notices the failure of the coordinator (for example by using a timeout), it hands the transaction over to the recovery process. The recovery process will find that it is in the prepared state for the transaction. It will periodically try to contact the coordinator site to find out how the transaction should be resolved. As we discussed above, after the coordinator recovers, it will abort the transaction and will answer "abort" upon receiving an inquiry message. The participant will then abort the transaction.

iv. Let's say that the coordinator instead crashes after successfully receiving votes from all participants, with all participants voting YES except for one NO vote. Assuming the coordinator sees no records for this transaction in its log after coming back online, how does this affect the answers to parts (d).i-(d).iii.?

Solution: Note: the coordinator's log may contain no information about the transaction or an ABORT record (because the abort record does not need to be flushed immediately with presumed abort). In this question, we assume the coordinator crashed without flushing its ABORT log record.
No change for part (d).i. because the coordinator likewise sees no log records for the transaction. The recovery process will abort the transaction locally, and since there is no information about the participant IDs in the log, the coordinator cannot send abort records to the participants.
No change for part (d).ii. - with presumed abort, even if the participant itself crashes after sending the NO vote, the participant will proceed with abort since that is the presumption given no log records.
No change for part (d).iii. - with presumed abort, when the coordinator comes back online and sees no information about the transaction in its log, the transaction is unilaterally aborted. This change is communicated when participants who voted YES ping the coordinator to find out how the transaction should be resolved, and the coordinator will respond with an ABORT message.
Note: the number of records written/flushed to disk with presumed abort is fewer than without presumed abort. However, with successful transactions, the number of flushed records will be the same.