Join Operations and Query Processing

CS 143 Introduction to Database Systems

TA: Jin Wang, Mingda Li shared slides (Edit by Jin)

02/14/2020

Announcement

- Midterm will be returned to you in lecture next Tuesday
- Project 2 is released
  - Due: March 8, Cut-off: March 10.
- Tentative schedule for remaining discussions
  - Week 7: Spark, Scala, Project 2
  - Week 8: ER diagram, Function Dependency
  - Week 9: Transaction and Concurrency Control
  - Week 10: Final Review

Outline

- Project 2 Overview
- Join Operations
- Query Processing

Project 2: Task

- Get known to the Apache Spark system
- Implement disk-based hash partition
- Implement UDF caching for both disk and in-memory settings
- Implement Hash-based aggregation (next week)

Project 2: Tips

- Get start as soon as possible
- Read the instructions *spark.md* and *setup.md* carefully
- Do not get overwhelmed by the bunch of codes
    - You only need to implement some functions specified by us
- If you do not have the experience of **Scala**, refer to tutorials provided in spec
- More details will be covered in discussion next week

Outline

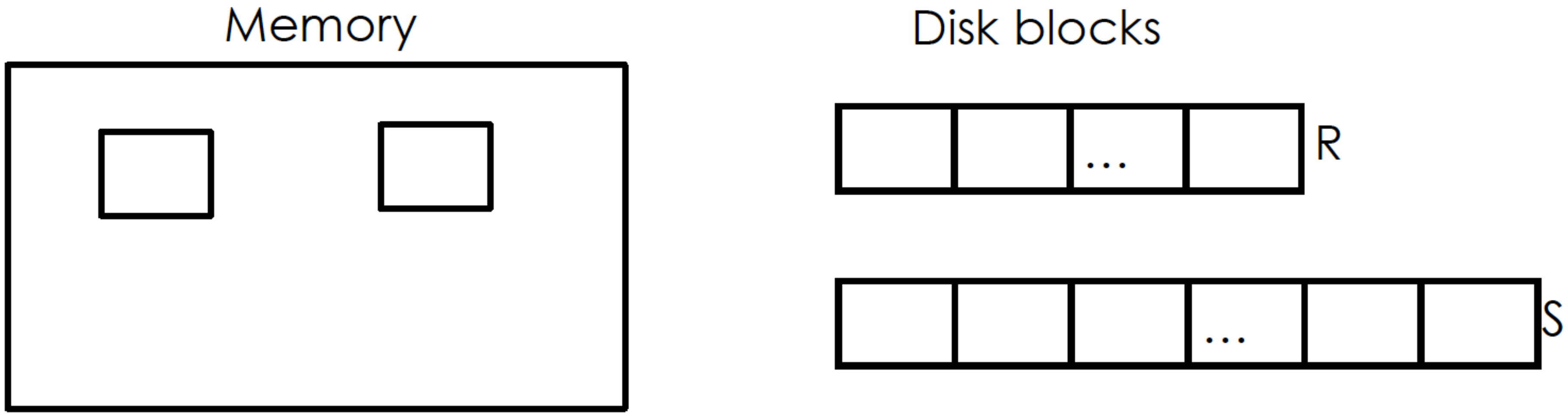
- Project 2 Overview
- Join Operations
- Query Processing

Join Operations: overview

- Types of Join operations [Physical Operator]
  - Nested-loop Join
  - Index Join
  - Sort Merge Join
  - Hash Join
- Cost Analysis
- Sample Question

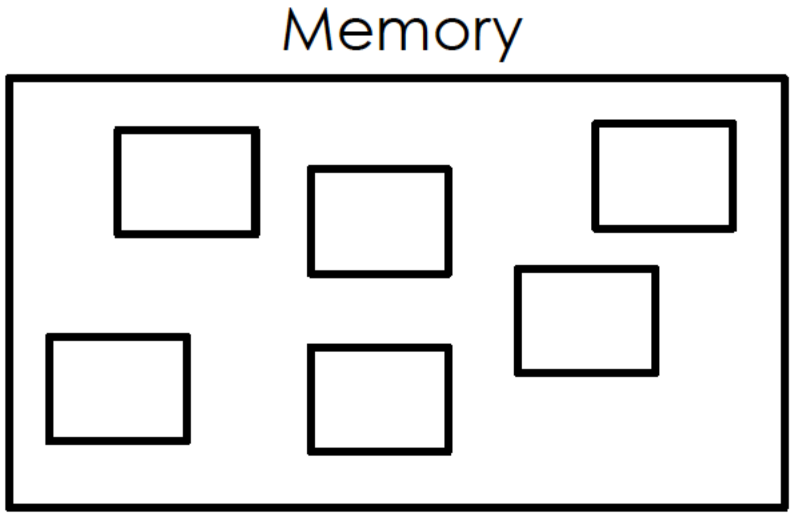
Cost Model

- Total number of disk blocks that have been read/written
  - Data are loaded from disk to memory in the unit of **block**
  - Count only when a disk block is being loaded into memory
  - Ignore the time of in-memory processing

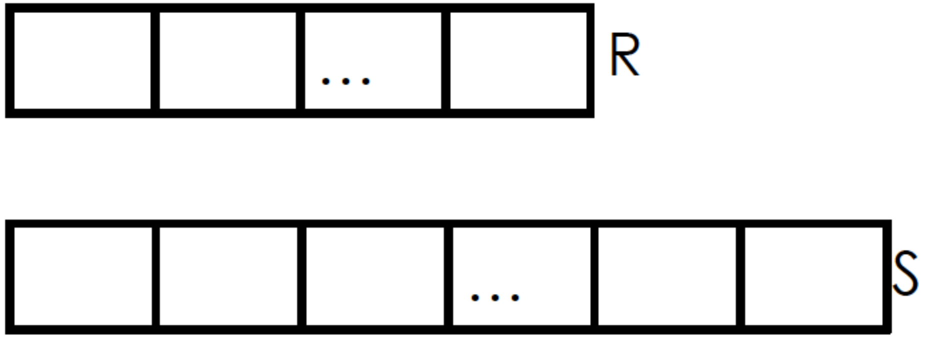


Running Example Setup

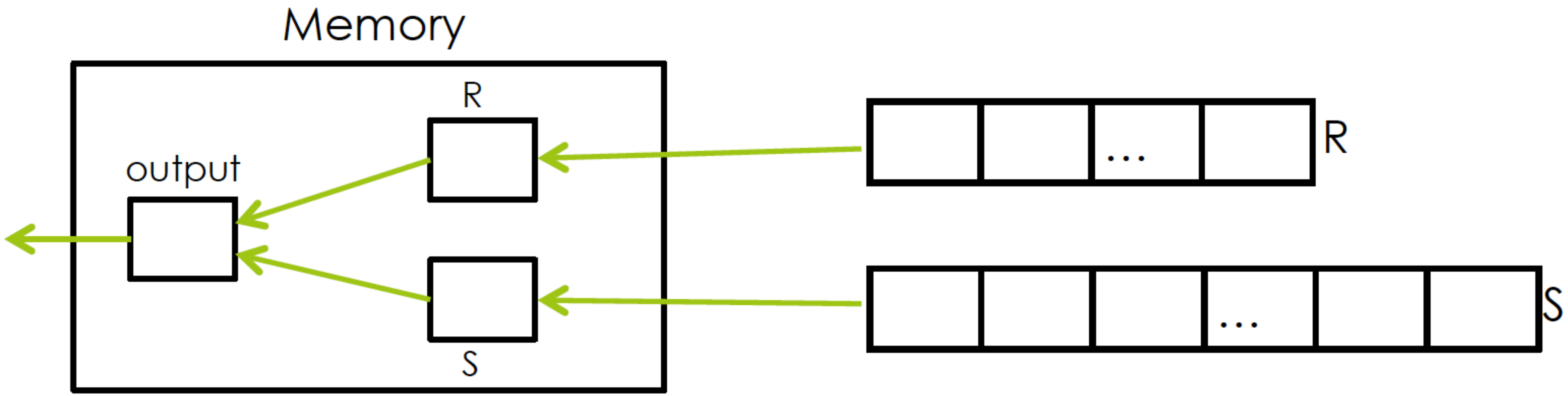
- Two tables R, S
- R has 1,000 tuples; S has 10,000 tuples
- 10 tuples/block (b_R=100 blocks, b_S=1,000 blocks)
- Memory buffer: M=22 blocks



SELECT * FROM R, S WHERE R.C=S.C

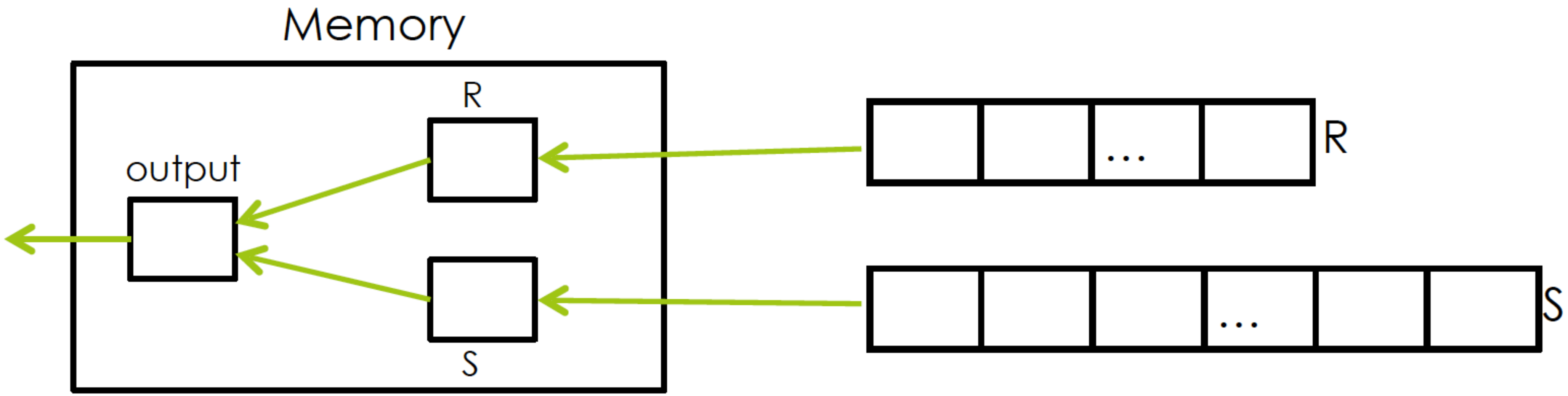


Nested-Loop Join



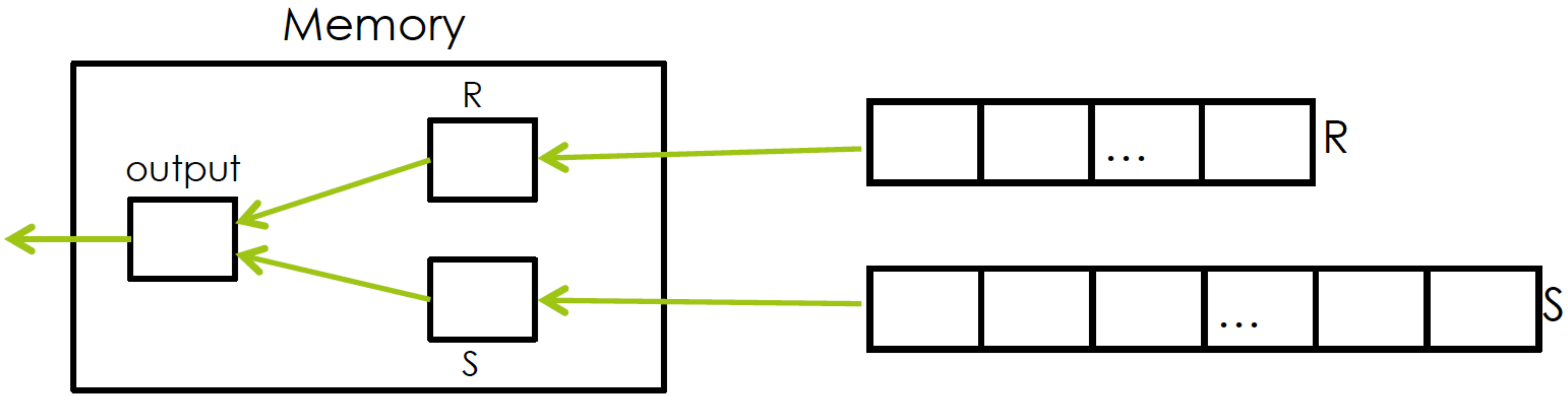
- Read a block from R at a time
  - For **each tuple** in R, compare it with **each tuple** in S
- Cost: $b_R + |R| b_S = 100 + 1000 * 1000 = 1000100$

Nested-Loop Join



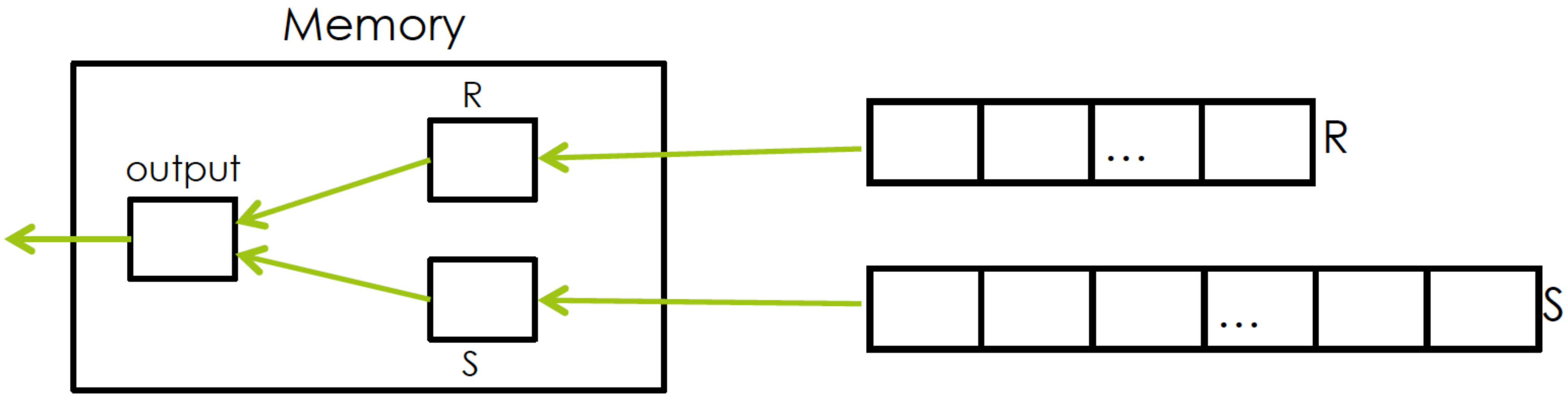
- What if we read S first?
- Cost: $b_S + |S|b_R = 1000+10000*100=1001000$
- Worse results!

Block Nested-Loop Join



- Read a block from R at a time
  - For **each block** in R, compare it with **each block** in S
- Cost: $b_S + b_R * b_S = 1000 + 100 * 1000 = 101000$

Block Nested-Loop Join

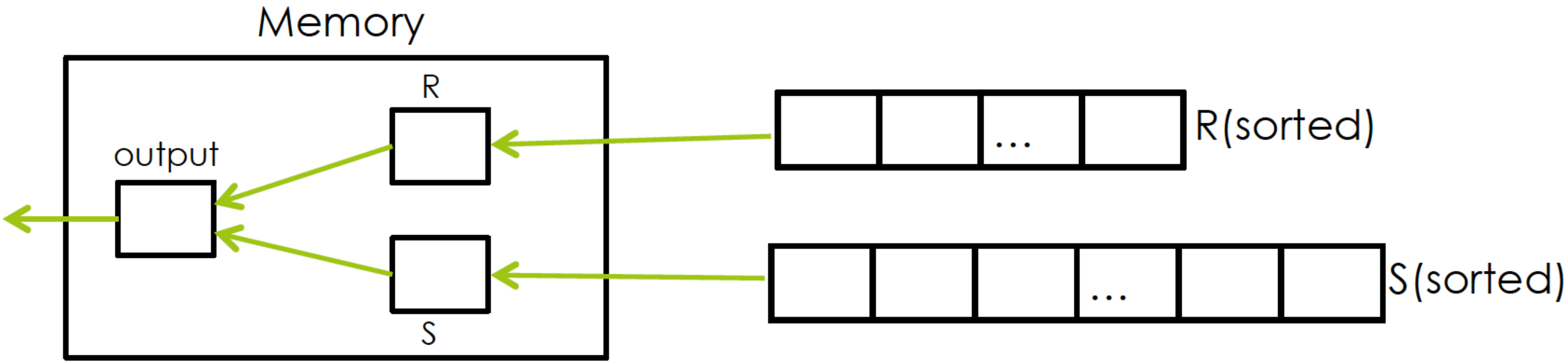


- What if we read R first?
- Cost: $b_R + b_S * b_R = 100 + 100 * 1000 = 100100 < 101000$
- Summary: Use the smaller table on the left (or outer loop)

Sort-Merge Join

- Two stage algorithm
  - Sort stage: Sort R and S – required **external merge-sort** process
  - Merge stage: Merge sorted R and S

Sort-Merge Join



- Read R and S blocks one block at a time
  - Merge…
- Cost: $b_R + b_S = 1000 + 100 = 1100$

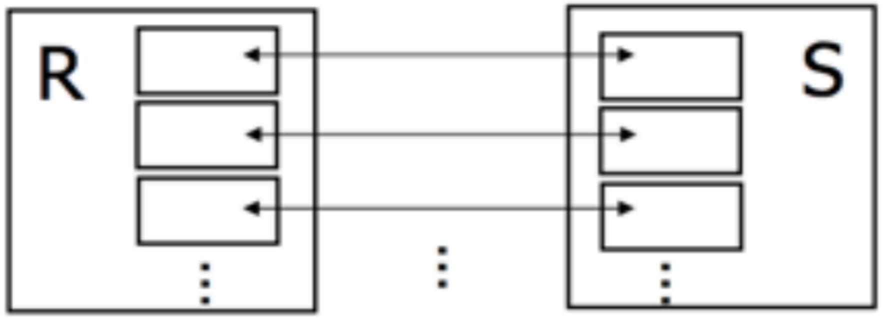
Hash Join

- Hash function h(v), range 1 → k

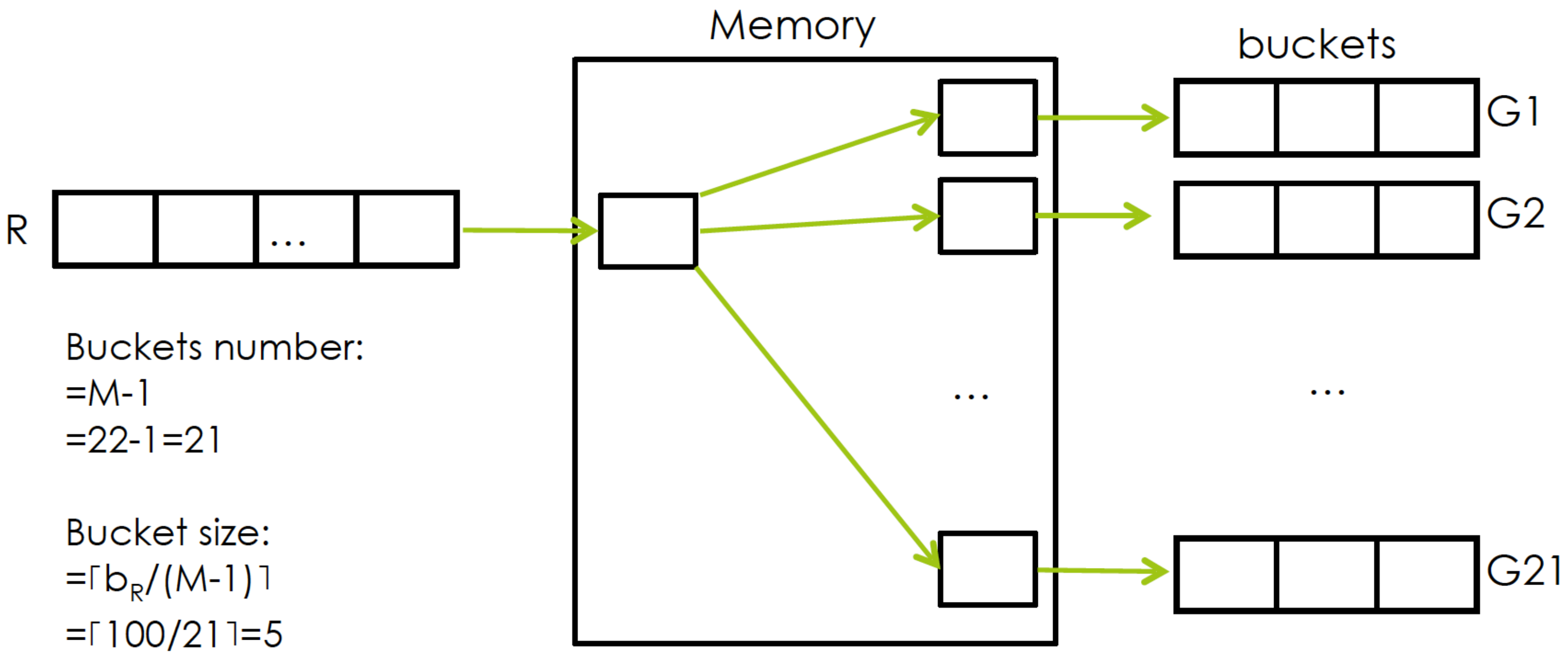
<u>Algorithm</u>

(1) Hashing stage (bucketizing): hash tuples into buckets
- Hash R tuples into G1,…,Gk buckets
- Hash S tuples into H1,…,Hk buckets

(2) Join stage: join tuples in matching buckets
- For i = 1 to k do
  match tuples in Gi, Hi buckets



Hash Join – hashing stage



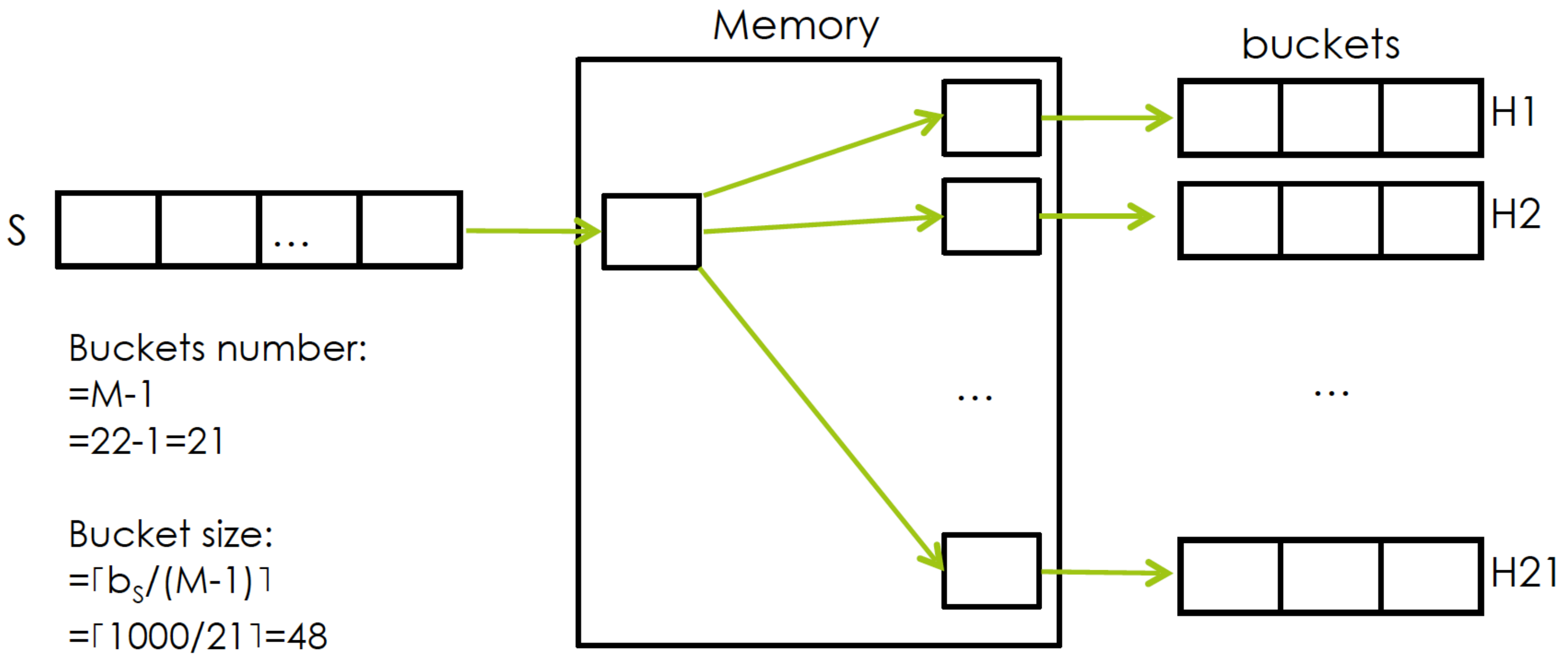
Buckets number:
=M-1
=22-1=21

Bucket size:
$=\lceil b_R/(M-1) \rceil$
$=\lceil 100/21 \rceil=5$

Block read: $b_R=100$
Block write: $\sim b_R=100$

Total:$\sim 2b_R=200$

Hash Join – hashing stage



Buckets number:
=M-1
=22-1=21

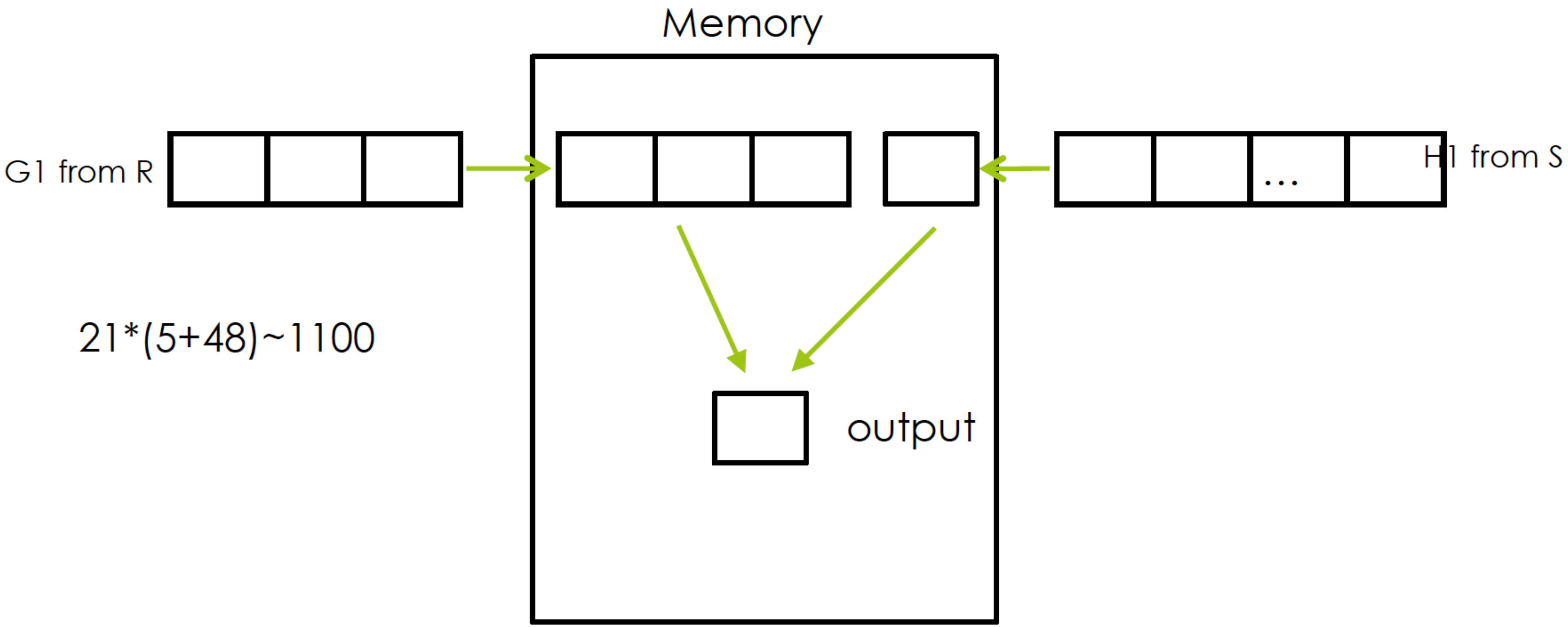
Bucket size:
$=\lceil b_S/(M-1)\rceil$
$=\lceil 1000/21\rceil=48$

Block read: $b_S=1000$

Block write: $\sim b_S=1000$

Total: $\sim 2b_S=2000$

Hash Join – join stage



21*(5+48)~1100

Total cost of two stages: 200+2000+1100 =3300

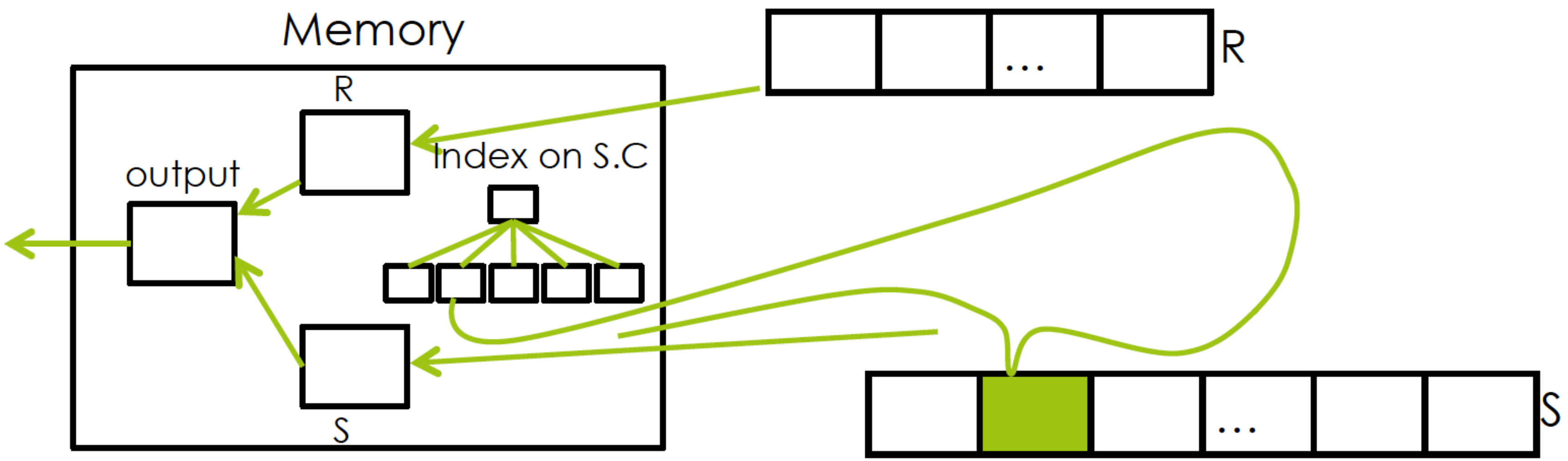
Index Join

- What if we have an index built on the join key of S?
- Index join cost:
  - I/O for R scanning : b_R
  - I/O for index look up: C
  - I/O for tuple read from S: J

General cost: $b_R + |R| \cdot (C + J)$

- C average index look up cost
- J matching tuples in S for every R tuple
- $|R|$ tuples in R

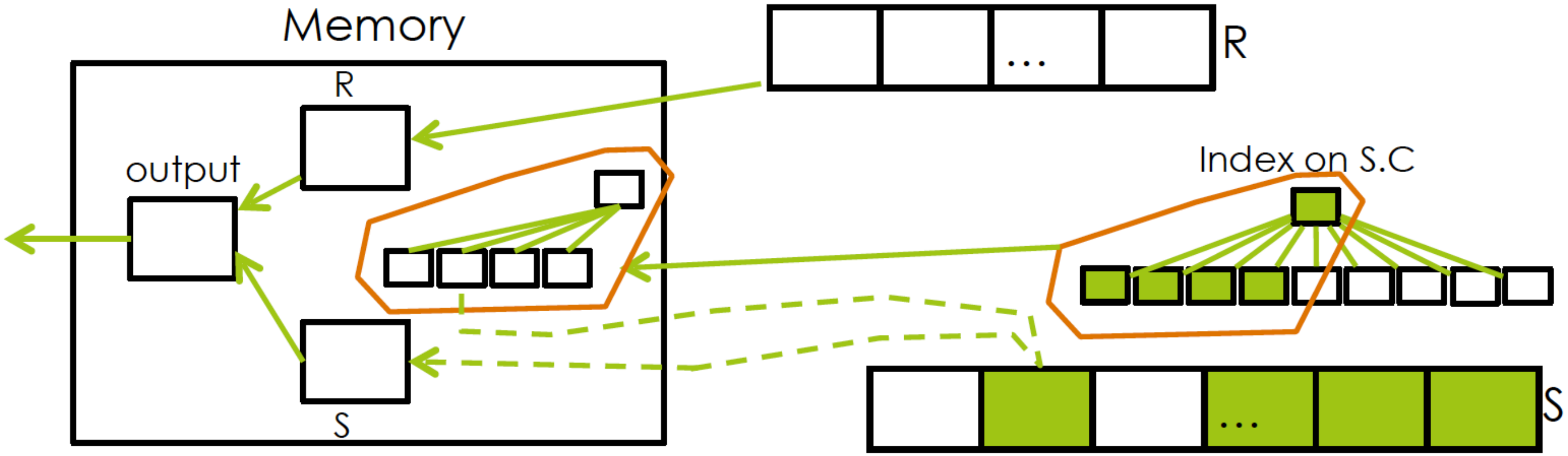
Index Join



Example 1
15 blocks for index (1 root, 14 leaf), read index into memory first
On average, 1 matching S tuples per an R tuple

How many IO?
index: 15
C: B+ tree: 0 $\quad 15+b_R+|R|*(0+1)$
J: S tuple: 1 $\quad 15+100+1000*(0+1)=1115$

Index Join



Example 1
40 blocks for index (1 root, 39 leaf), read partial index into memory first
On average, 10 matching S tuples per an R tuple

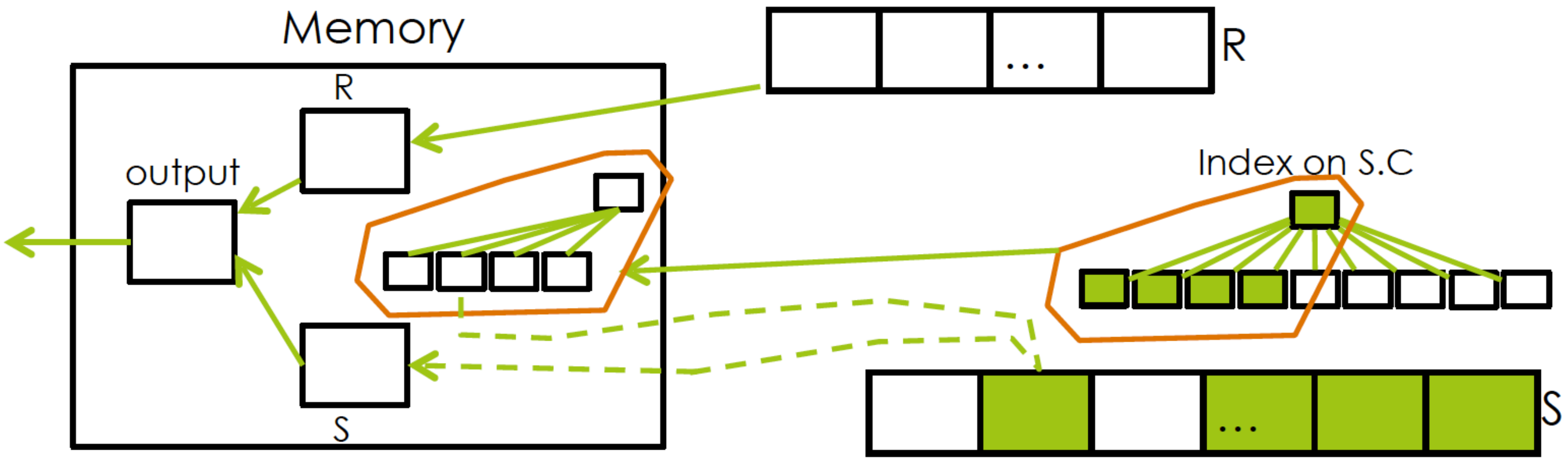
How many IO?
M-3: partial index (1 root, 18 leaf): 19
C: B+ tree: 0*(18/39)+1*(21/39)=0.5

J: S tuple: 10

$(M-3)+b_R+|R|*(C+J)$

19+100+1000*(0.5+10)=10619

Index Join Summary



read partial index into memory first

How many IO? $b_R + |R| * (C+J)$

M-3: partial index (negligible)

C: average index look up cost

J: matching tuples in S for every R tuple

|R|: tuples in R

Summary of Join Algorithm

- Nested-loop join OK for “small” relations (relative to memory size)
- Hash join usually best for equi-join if relations not sorted and no index
- Merge join for sorted relations
- Sort merge join good for non-equi-join
- Consider index join if index exists
- DBMS maintains statistics on data (for query optimization)

Outline

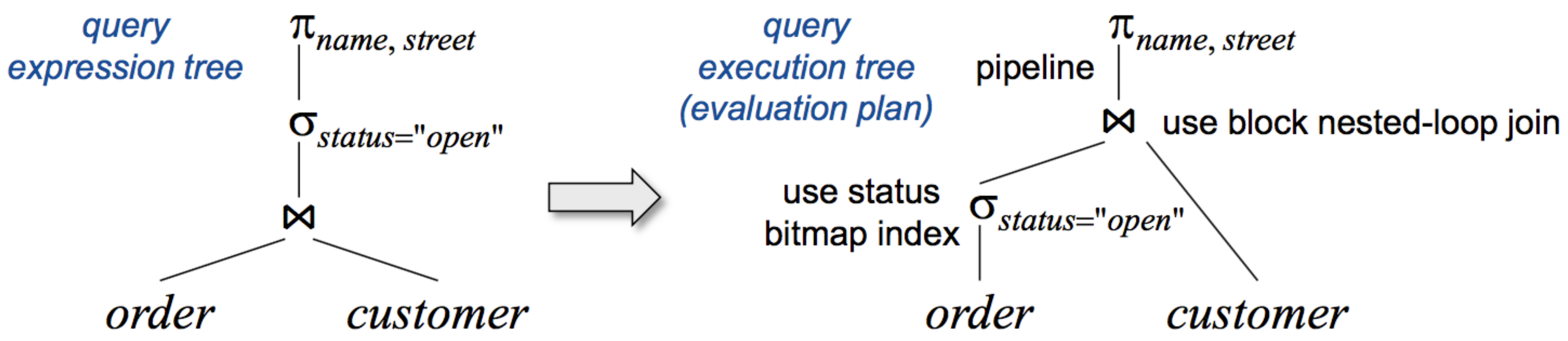
- Project 2 Overview
- Join Operations
- Query Processing

Query Expression and Execution

```sql
SELECT name, street
FROM Customer, Order
WHERE Order.customerID = Customer.customerID AND status = 'open';
```

- Transform the SQL query to the following query plan

$$\pi_{name,\ street}(\sigma_{status="open"}(order \bowtie customer))$$

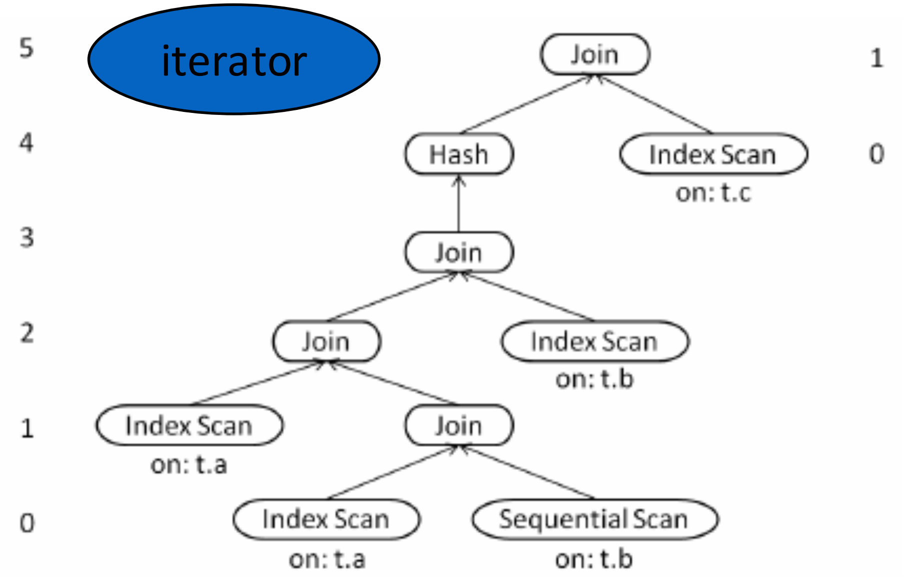


note that we will later see how to optimise the query expression tree

Relational Operator - Iterator

- The relational operators are all subclasses of the class `iterator`:

```
class iterator {
    void init();
    tuple next();
    void close();
    iterator inputs[];
    // additional states
}
```



- Note:
  - Edges in the graph are specified by inputs (max 2, usually)
  - Encapsulation: any iterator can be input to any other!
  - When subclassing, different iterators will keep different kinds of state information

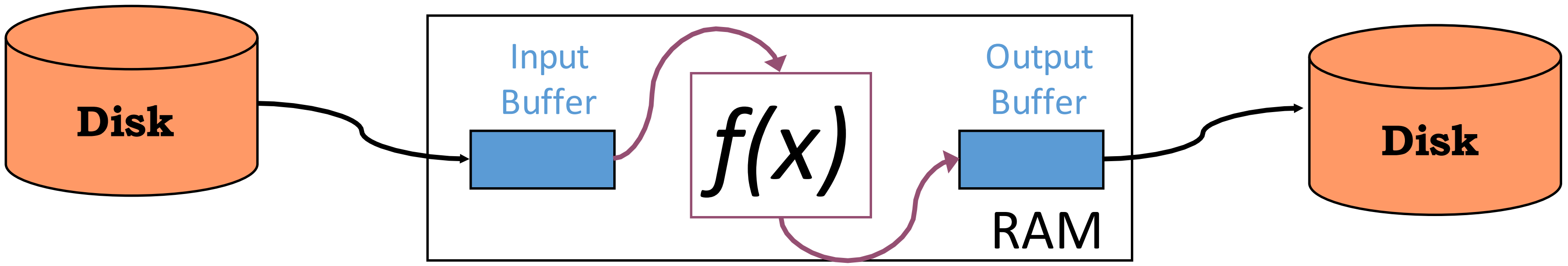
Example: Projection Scan

```
class Scan extends iterator {
   void init();
   tuple next();
   void close();
   List<Attribute> proj_list;
   iterator inputs[1];
}
```

- `init()`:
  - Set up internal state
  - call init() on child – often a file open
- `next()`:
  - call next() on child until qualifying tuple found or EOF
  - keep only those fields in “proj_list”
  - return tuple (or EOF -- “End of File” -- if no tuples remain)
- `close()`:
  - call close() on child
  - clean up internal state

Streaming through RAM

- How to transform input tuples (on Disk) to output tuples (on Disk)?
- Simple case: “Map”. (assume many records per disk page)
  - Goal: Compute *f(x)* for each record, write out the result
  - Challenge: minimize RAM, call read/write rarely
- Approach
  - Read a chunk from INPUT to an *Input Buffer*
  - Write *f(x)* for each item to an *Output Buffer*
  - When Input Buffer is consumed, read another chunk
  - When Output Buffer fills, write it to OUTPUT
- Reads and Writes are *not coordinated* (i.e., not in lockstep)
  - E.g., if f() is Compress(), you read many chunks per write.
  - E.g., if f() is DeCompress(), you write many chunks per read.

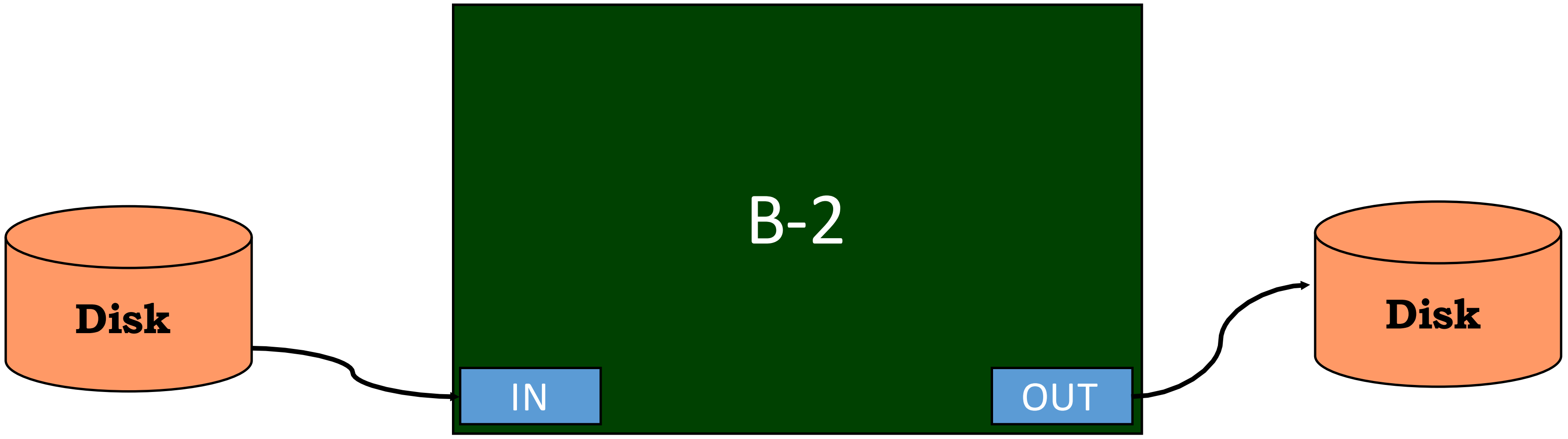


Rendezvous

- Streaming: one chunk at a time. Easy.
- But some algorithms need certain items to be co-resident in memory
  - not guaranteed to appear in the same input chunk
- *Time-space Rendezvous*
  - in the same place (RAM) at the same time
- Examples
  - External Merge Sort, Hashing

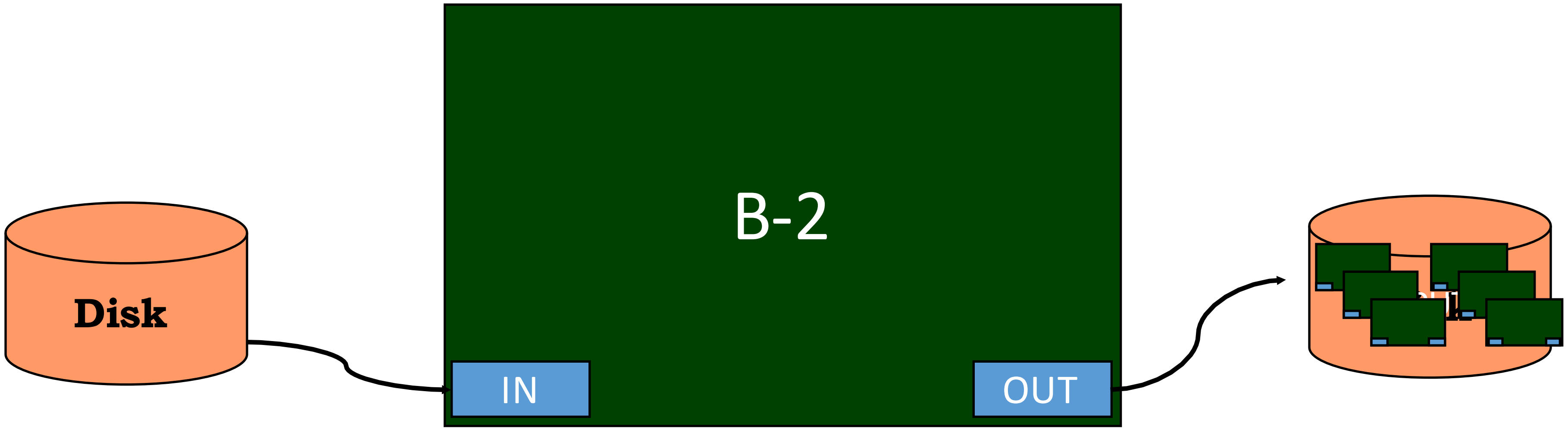
Divide and Conquer

- *Out-of-core (external memory)* algorithms orchestrate rendezvous.
- Typical RAM Allocation:
  - Assume B pages worth of RAM available
  - Use 1 page of RAM to read into
  - Use 1 page of RAM to write into
  - B-2 pages of RAM as workspace



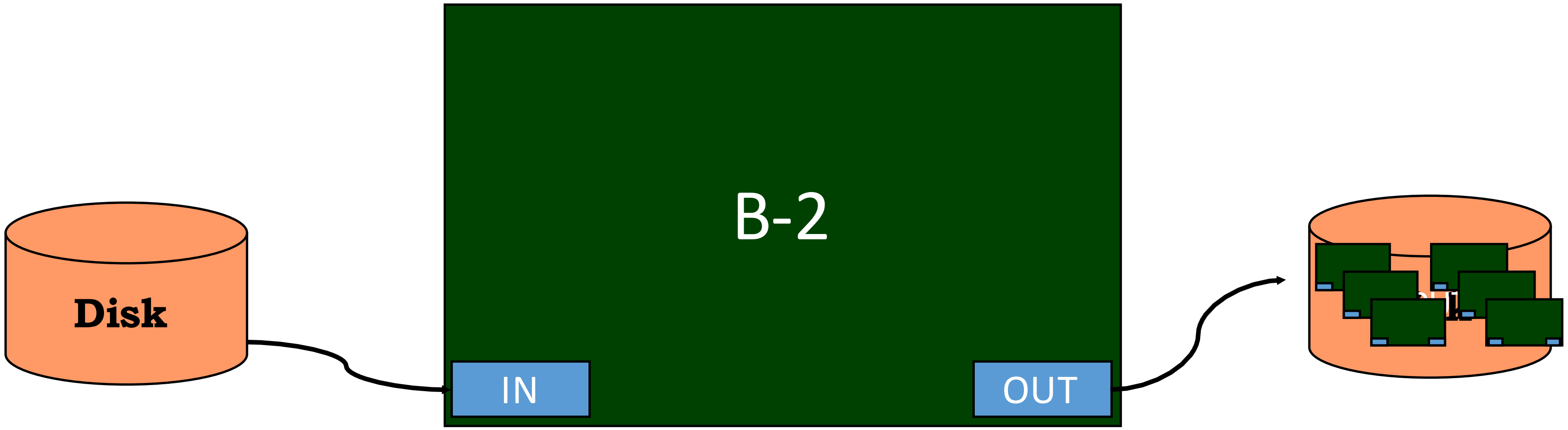
Divide and Conquer

- Phase 1
  - “streamwise” *divide* into N/(B-2) megachunks
  - output (write) to disk one megachunk at a time



Divide and Conquer

- Phase 2
  - Now megachunks will be the input
  - process each *megachunk* individually.



Example: Hashing

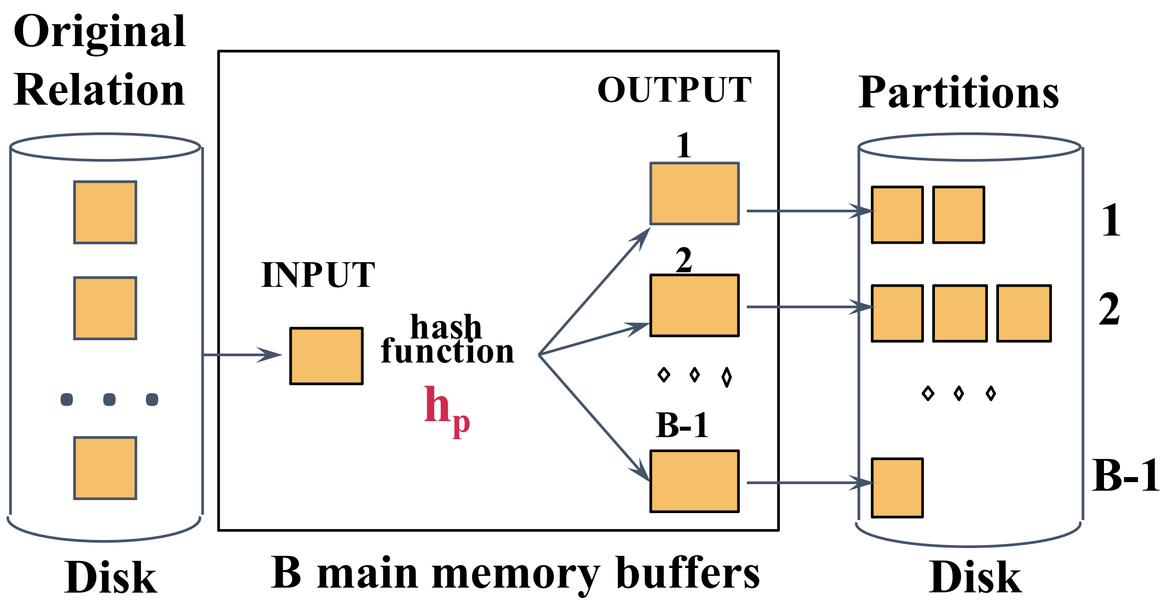
- Idea:
  - Many times we don’t require order
  - E.g. removing duplicates
  - E.g. forming groups
- Often just need to *rendezvous* matches
- Hashing does this
  - And may be cheaper than sorting! (Hmmm…!)
  - But how to do it out-of-core (externally)??

Divide & Conquer

- Streaming Partition (divide):
  Use a hash function h_p to stream records to disk-based partitions
  - All matches happens in the same partition.
  - *Streaming* algorithm to create partitions on disk:
    - "Spill" partitions to disk via output buffers
- ReHash (conquer):
  Read partitions into memory-based hash table one at a time, using hash function h_r
  - Then go through each bucket of this hash table to achieve rendezvous in RAM
- Note: Two different hash functions
  - h_p is coarser-grained than h_r

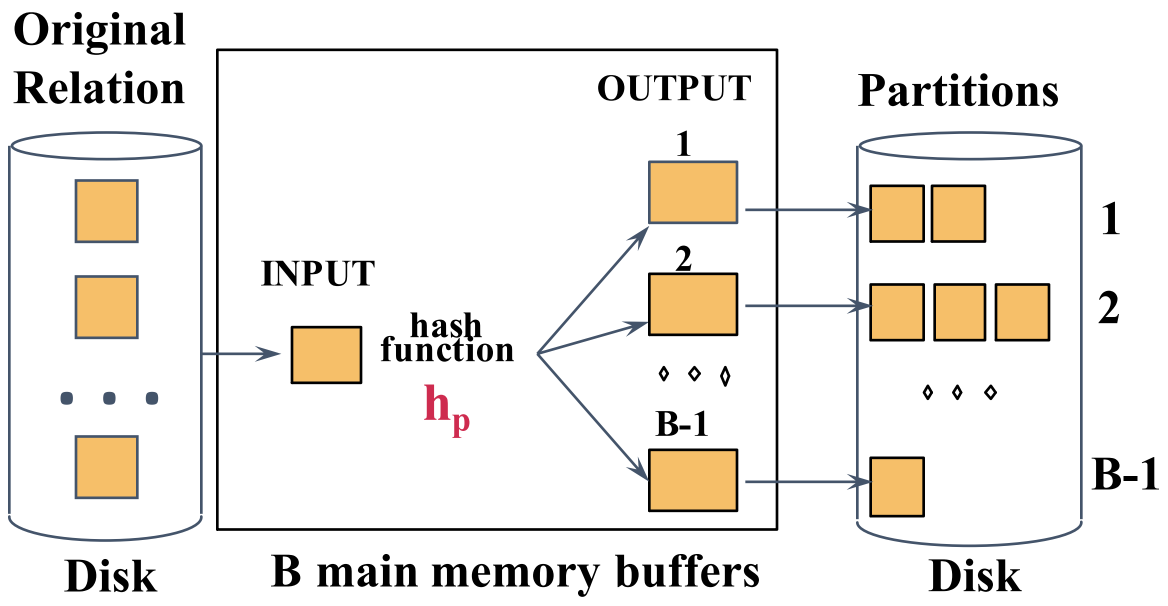
Two Phases

- Partition:

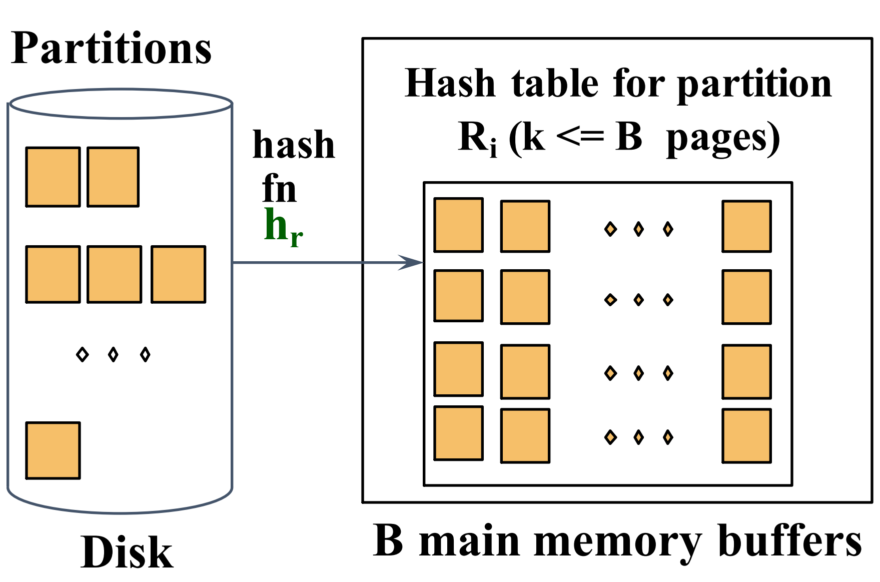


Two Phases

- Partition:



- Rehash:



Hashing Summary

- Hashing:
  - Pass 1 uses coarse-grained hash function to produce B-1 partitions
  - Pass 2 loads each partition into memory, rehashes them, and writes them out to disk (or probes them for hash join).
  - Any partition of size > B must be repartitioned (repeat step 1)
  - In 2 passes, sort B(B-1) pages of data