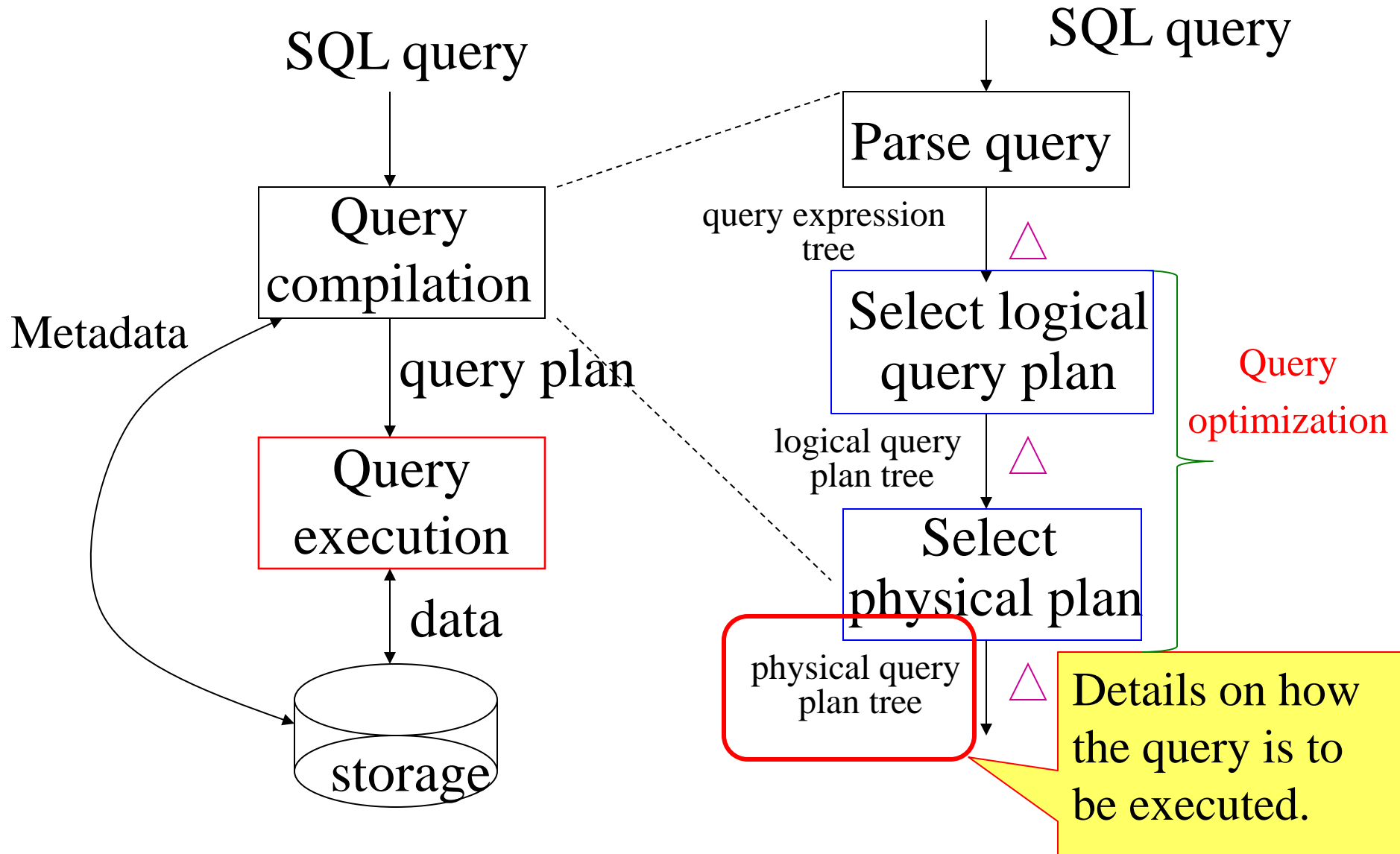


# Query Execution

INF 551

Wensheng Wu

# Components of Query Processor



# Converting SQL to Logical Plans

Select  $a_1, \dots, a_n$   
From  $R_1, \dots, R_k$   
Where  $C$

$$\Pi_{a_1, \dots, a_n}(\sigma_C(R_1 \times R_2 \times \dots \times R_k))$$

Select  $a_1, \dots, a_n$ , aggs  
From  $R_1, \dots, R_k$   
Where  $C$   
Group by  $b_1, \dots, b_m$

$$\Pi_{a_1, \dots, a_n}(\gamma_{b_1, \dots, b_m, \text{aggs}}(\sigma_C(R_1 \times R_2 \times \dots \times R_k)))$$

# Logical Query Optimization

- Apply algebraic laws to turn initial query plan into more efficient one
- Use heuristics
  - E.g., do selections & projection as early as possible

# Example of Algebraic Law

$$\square \sigma_C (R \bowtie S) = \sigma_C (R) \bowtie S$$

- That is, we can push selection down to R if condition C only contains attributes in R

# Physical Query Optimization

- Turn logical query plan into physical ones
  - That is, plan with physical operators
- Pick a physical plan with the lowest cost (I/O's)
  - I.e., cost-based optimization

# Outline

- Logical/physical operators
- Cost model
- One-pass algorithms
- Nested-loop joins
- Two-pass algorithms
  - Sorting-based
  - Hashing-based
- Index-based algorithms

# Logical vs. Physical Operators

- Logical operators
  - what they do
  - e.g., union, selection, projection, join, group-by
- Physical operators
  - how they do it
  - Main methods: scanning, hashing, sorting, and indexing
  - E.g., methods for implementing joins include:
    - nested loop join, sort-merge join, hash join, index join
  - Different methods may have different requirements on the amount of available memory & different costs



# Logical Query Plans

```
SELECT  P.buyer  
FROM    Purchase P, Person Q  
WHERE   P.buyer=Q.name AND  
        Q.city='LA'
```

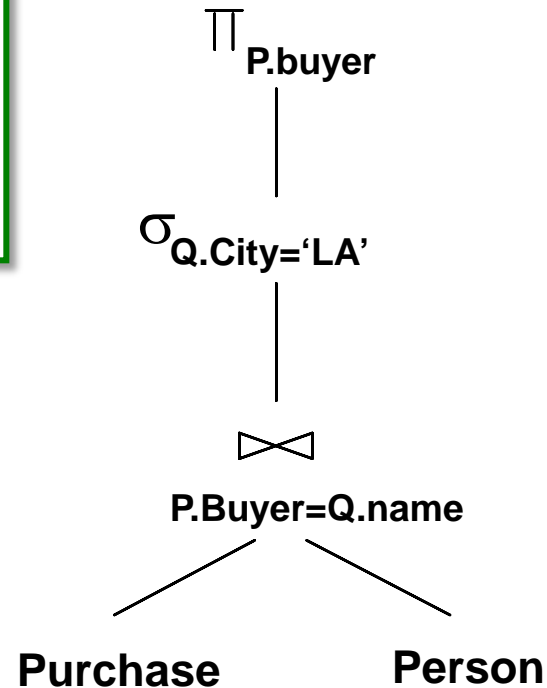
Construct logical  
plan...

# Logical Query Plans

```
SELECT  P.buyer
FROM    Purchase P, Person Q
WHERE   P.buyer=Q.name AND
        Q.city='LA'
```

## Query Plan:

- Tree with logical operators

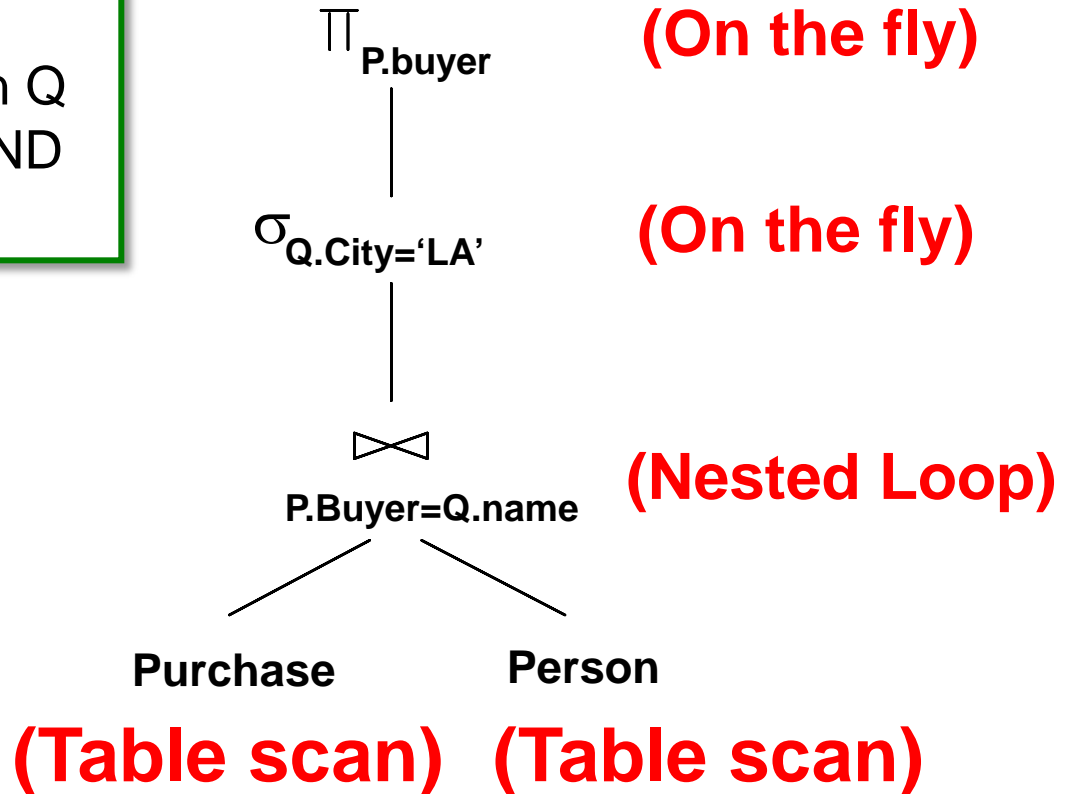


# Physical Query Plans

```
SELECT P.buyer
FROM   Purchase P, Person Q
WHERE  P.buyer=Q.name AND
       Q.city='LA'
```

## Query Plan:

- Logical tree plus
- **Implementation**  
choice at each node



# How do We Combine Operations?

- **The iterator model.** Each operation is implemented by 3 functions:
  - *Open*: sets up the data structures and performs initializations
  - *GetNext*: returns the the next tuple of the result.
  - *Close*: ends the operations. Cleans up the data structures.
- Enables pipelining!
- Contrast with **data-driven materialized model**

# Cost Model

- Cost parameters
  - $M$  = number of blocks/pages that are available in main memory
  - $B(R)$  = number of blocks holding  $R$
  - $T(R)$  = number of tuples in  $R$
  - $V(R,a)$  = number of distinct values of the attribute  $a$  of  $R$
- Estimating the cost of physical operators:
  - Important in query optimization
  - Here we consider I/O cost only
  - We assume operands are relations stored on disk, but operator results will be left in main memory (e.g., pipelined to next operator in query plan)
  - So we don't include the cost of *writing* the result

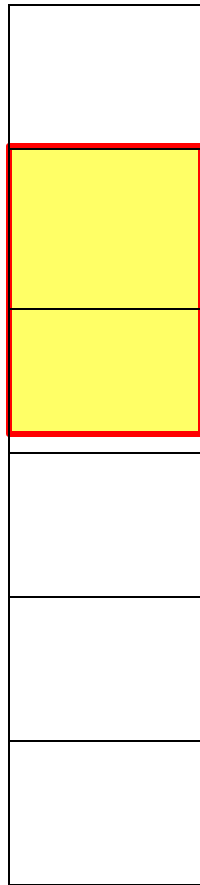
# I/O Cost

- # of blocks read from or written to disk
- Recall that disk reads/writes data in the unit of block

# Scanning Tables

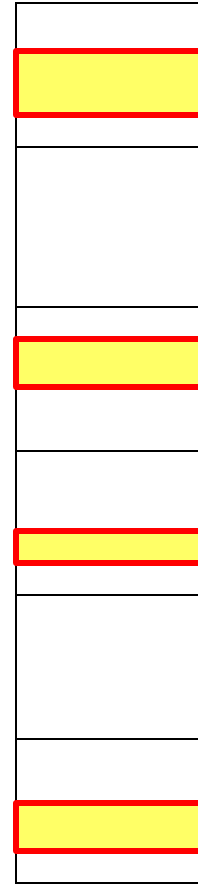
- Reading every row of tables
- The table is *clustered* (i.e., block consists only of records from this table):
  - # of I/O's = # of blocks
- The table is *unclustered* (e.g. its records are placed in blocks with those of other tables)
  - May need one block read for each record

# Scanning Clustered/Unclustered Tables



2 Block Reads  
( $B(R) = 2$ )

Clustered table



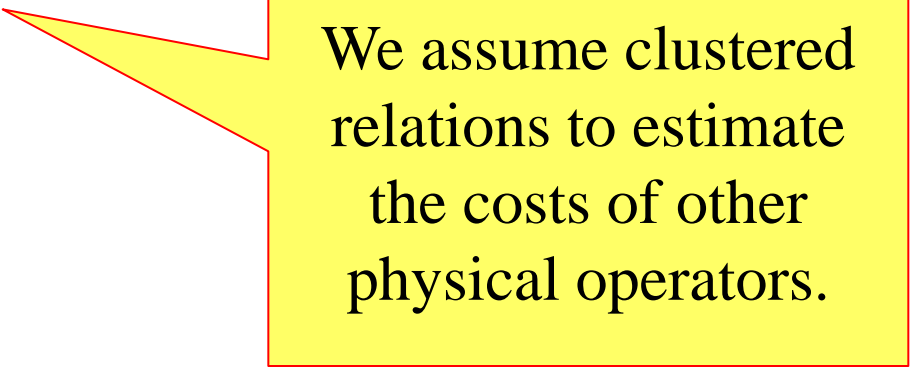
4 Reads  
( $T(R) = 4$ )

Unclustered table



# Cost of the Scan Operator

- Clustered relation:
  - Table scan:  $B(R)$
- Unclustered relation:
  - $T(R)$



We assume clustered relations to estimate the costs of other physical operators.

# Classification of Physical Operators

- One-pass algorithms
  - Read the data only once from disk
  - Usually, require at least one of the input relations fits in main memory
- Nested-Loop Join algorithms
  - Read one relation only once, while the other will be read repeatedly from disk
- Two-pass algorithms
  - First pass: read data from disk, process it, write it to the disk
  - Second pass: read the data for further processing

# Classification of Physical Operators

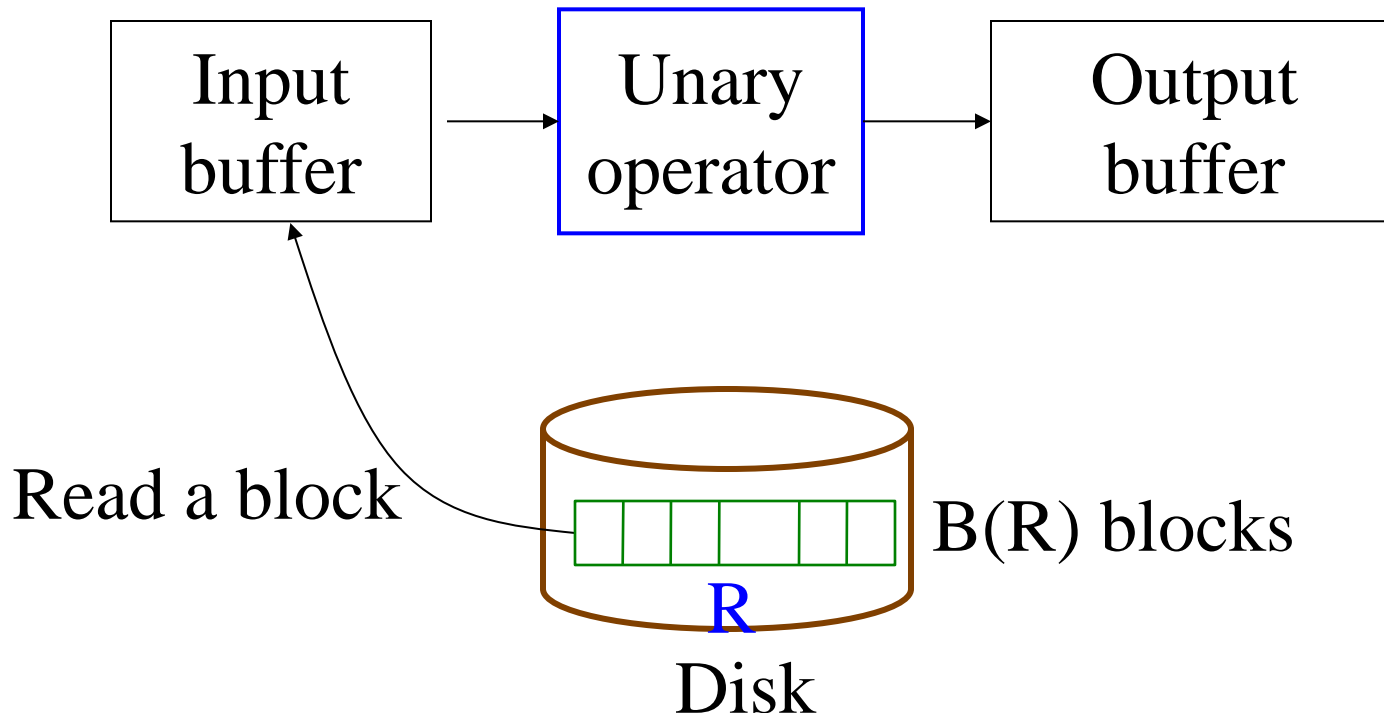
- K-pass algorithms
  - If data are too big or memory is too small, the algorithm may need  $k > 2$  passes over the data

# One-pass algorithms

# One-pass Algorithms

Selection  $\sigma(R)$ , projection  $\Pi(R)$

- Both are tuple-at-a-time algorithms
- Cost:  $B(R)$



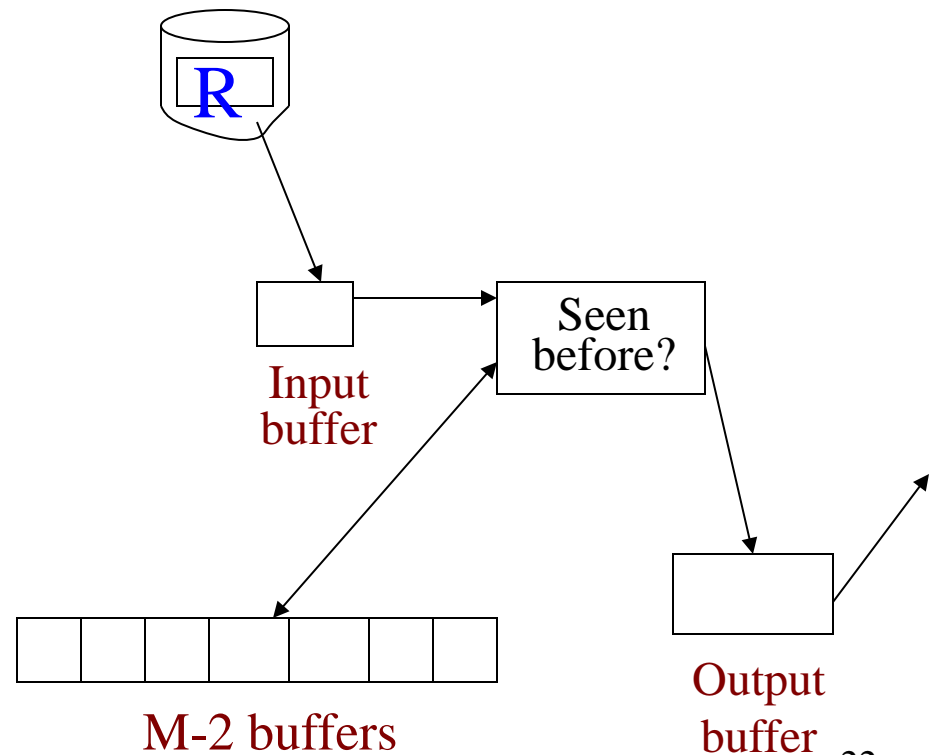
# One-pass Algorithms

## Duplicate elimination $\delta(R)$

- Need to keep a dictionary in memory:
  - balanced search tree
  - hash table
  - Etc.
- Cost:  $B(R)$
- Assumption:

$$B(\delta(R)) \leq M-2$$

or roughly  $M$



# One-pass Algorithms

Grouping:  $\gamma_{\text{city}, \text{sum}(\text{price})} (R)$

- Need to keep a dictionary in memory
  - Also store the  $\text{sum}(\text{price})$  for each city
- Cost:  $B(R)$
- Assumption: number of cities and sums fit in memory

# One-pass Algorithms

Binary operations:  $R \cap S$ ,  $R \cup S$ ,  $R - S$ ,  $R \bowtie S$

- Assumption:  $\min(B(R), B(S)) \leq M$  (or  $M-2$  to be exact)
- Scan a smaller table of  $R$  and  $S$  into main memory, then read the other one, block by block
- Cost:  $B(R) + B(S)$
- Example:  $R \cap S$ 
  - Read  $S$  into  $M-2$  buffers and build a search structure
  - Read each block of  $R$ , and for each tuple  $t$  of  $R$ , see if  $t$  is also in  $S$ .
  - If so, copy  $t$  to the output; if not, ignore  $t$



# Nested-loop join

# Tuple-based Nested Loop Joins

- Join  $R \bowtie S$
- Assume neither relation is clustered

```
for each tuple r in R do  
    for each tuple s in S do  
        if r and s join then output (r,s)
```

- Cost:  $T(R) T(S)$

# Block-based Nested Loop Joins

- Assume both relations are clustered

for each (M-2) blocks  $b_r$  of  $R$  do

for each block  $b_s$  of  $S$  do

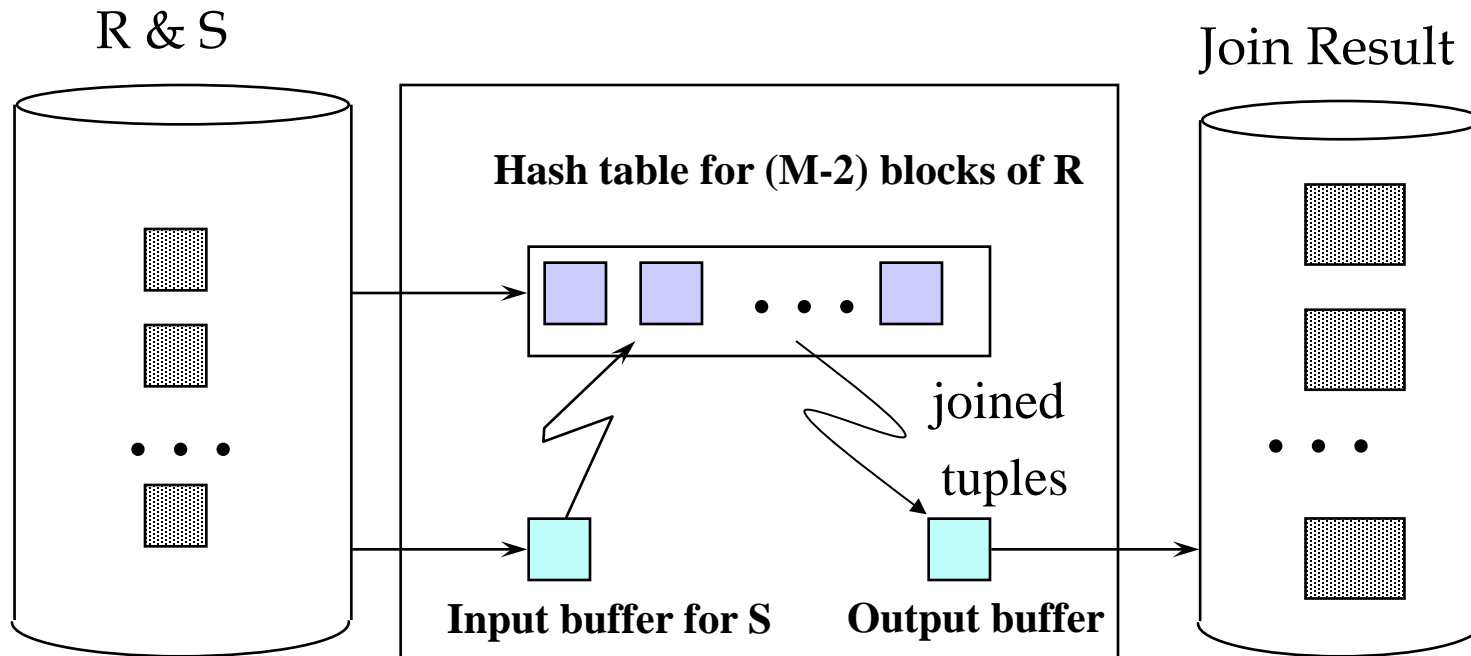
for each tuple  $r$  in  $b_r$  do

for each tuple  $s$  in  $b_s$  do

if  $r$  and  $s$  join then output( $r,s$ )

- Assume  $B(R) \leq B(S)$  &  $B(R) > M$

# Block-based Nested Loop Joins



# Block-based Nested Loop Joins

- Cost:
  - Read R once: cost  $B(R)$
  - Outer loop runs  $B(R)/(M-2)$  times, and each time need to read S: costs  $B(R)B(S)/(M-2)$
  - Total cost:  $B(R) + B(R)B(S)/(M-2)$
- Notice: it is better to iterate over the smaller relation first
- $R \bowtie S$ : R=outer relation, S=inner relation
- What is the minimum memory requirement?

# Example

- Suppose  $M = 102$  blocks (i.e., pages),  $B(R) = 1,000$  blocks,  $B(S) = 5,000$  blocks
- Cost of  $R \bowtie S$  using blocked-based nested-loop join algorithm
  - If  $R$  is outer relation
  - If  $S$  is outer relation

# Two-pass algorithms

# Two-pass Algorithms

- If an operation can not be completed in one pass, can we design an algorithm to complete it in two passes?
  - Yes, but with certain restriction on the relation size



# Ideas

- Sorting
  - Sort relation(s) into runs
  - Perform the needed operation while merging the runs
- Hashing
  - Hash relation(s) into buckets
  - Only need to examine a bucket or a pair of bucket at a time

# Duplicate Elimination $\delta(R)$

## Based on Sorting

- Simple idea: sort first, then eliminate duplicates
- Pass 1: sort runs of size  $M$ , write
  - Cost:  $2B(R)$
- Pass 2: merge  $M-1$  runs, but include each tuple only once
  - Cost:  $B(R)$
- Total cost:  $3B(R)$ , Assumption:  $B(R) \leq M^2$ 
  - since  $B/M = \#$  of runs
  - $\#$  of runs has to be  $\leq M-1$  to complete the merging in the second pass
  - So  $B/M \leq M - 1$

# Grouping: $\gamma_{\text{city}, \text{sum}(\text{price})} (R)$ Based on Sorting

- Pass 1: same as before
- Pass 2: same as before, but also compute  $\text{sum}(\text{price})$  for group during the merge phase.
- Total cost:  $3B(R)$
- Assumption:  $B(R) \leq M^2$

# Binary operations: $R \cap S$ , $R \cup S$ , $R - S$

## Based on Sorting

- Idea: sort  $R$ , sort  $S$ , then do the right thing
- A closer look:
  - Step 1: split  $R$  into runs of size  $M$ , then split  $S$  into runs of size  $M$ . Cost:  $2B(R) + 2B(S)$
  - Step 2: **merge  $M-1$  runs from  $R$  and  $S$** ; output a tuple on a case by cases basis
- Total cost:  $3B(R) + 3B(S)$
- Assumption:  $B(R) + B(S) \leq M^2$

# Sort-Merge Join

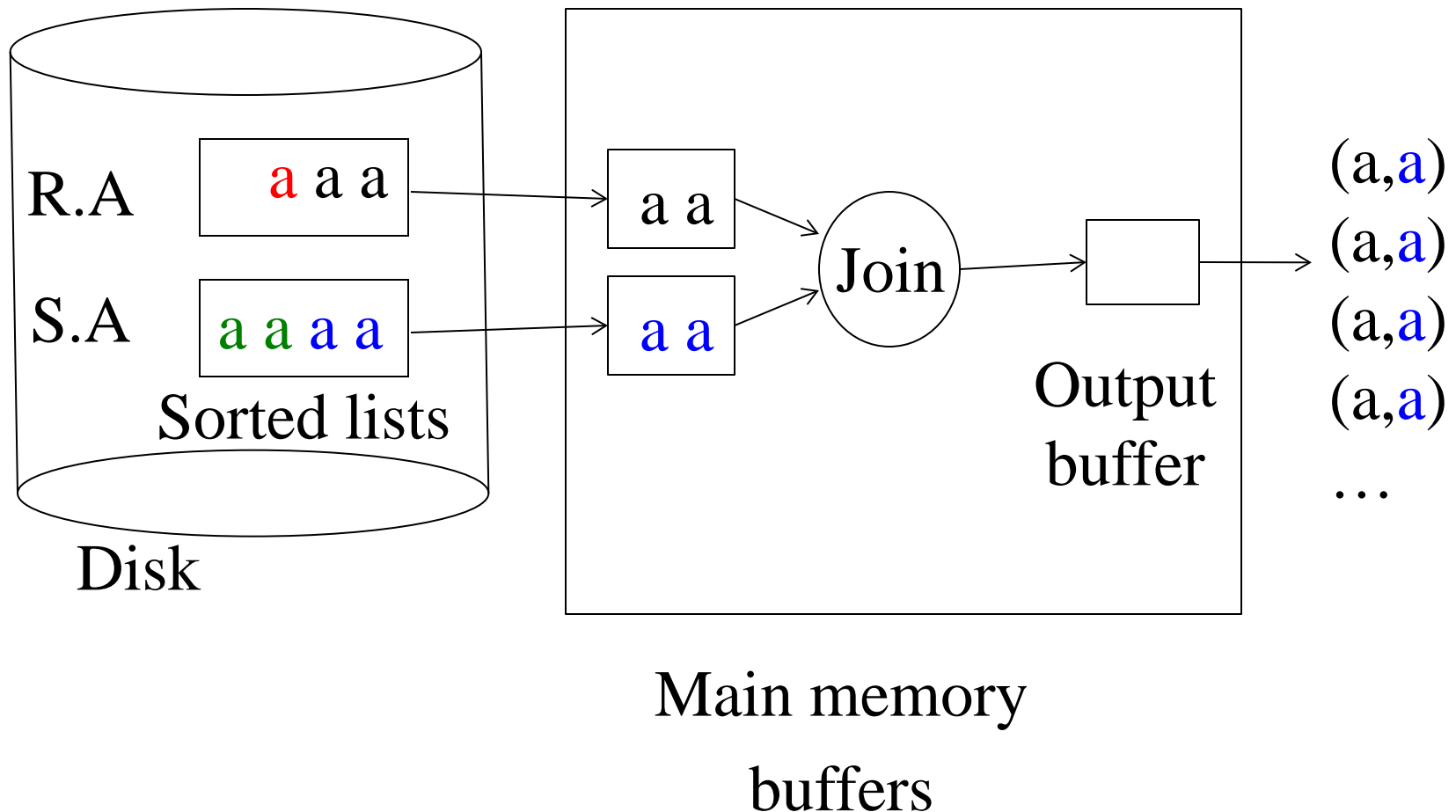
- Assume buffer is enough to hold join tuples for at least one relation
  - Note that buffer also needs to hold a block for each run of the other relation
- Total cost:  $3B(R)+3B(S)$
- Assumption:  $B(R) + B(S) \leq M^2$

# Example

- Suppose  $M = 101$  blocks (i.e., pages),  $B(R) = 1,000$  blocks,  $B(S) = 5,000$  blocks
  - Suppose we use 100 blocks in sorting
- Cost of  $R \bowtie S$  using sort-merge join algorithm
- What if  $B(S) = 50,000$  blocks?

# Problem

Many tuples may have the same value on the join attribute



# Problem

- A large number of tuples with the same value on the join attribute(s)
- But buffer can not hold all joining tuples (with the same value on join attribute) for at least one relation



# Simple Sort-based Join

- Start by **completely** sorting both R and S on the join attribute (assuming this can be done in 2 passes):
  - Cost:  $4B(R)+4B(S)$  (because need to write to disk)
- Read both relations in sorted order, match tuples
  - Cost:  $B(R)+B(S)$
- Can use as many buffers as possible to load join tuples from one relation (with the same join value), say R
  - Only one buffer is needed for the other relation, say S
- If we still can not fit all join tuples from R
  - Need to use nested loop algorithm, higher cost

# Simple Sort-based Join

- Total cost:  $5B(R)+5B(S)$
- Assumption:  $B(R) \leq M^2$ ,  $B(S) \leq M^2$ , and at least one set of the tuples with a common value for the join attributes fit in  $M$  (or  $M-2$  to be exact)
  - Note that we only need one page buffer for the other relation

# Example

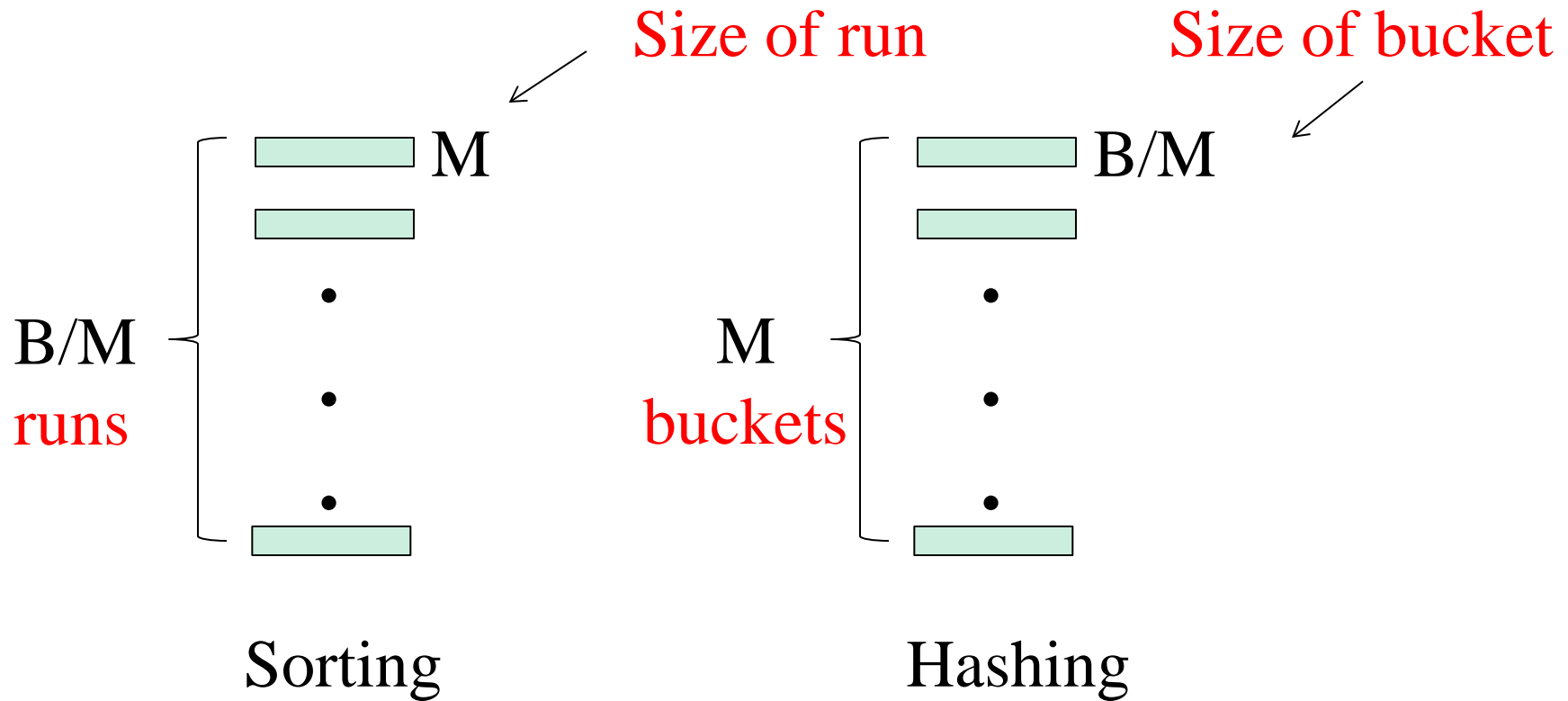
- Suppose  $M = 101$  blocks (i.e., pages),  $B(R) = 1,000$  blocks,  $B(S) = 5,000$  blocks
  - Assume that we use 100 blocks in sorting
- Cost of  $R \bowtie S$  using simple sort-based join algorithm
- What if  $B(S) = 50,000$  blocks?

# Two-Pass Algorithms Based on Hashing

# Hashing-Based Algorithms

- Hash all the tuples of input relations using an appropriate hash key such that:
  - All the tuples that need to be considered together to perform an operation go to the same bucket
- Reduce the size of input relations by a factor of  $M$
- Perform the operation by working on a bucket (or a pair of buckets for binary operations) at a time
  - Apply a one-pass algorithm for the operation

# Sorting vs. Hashing



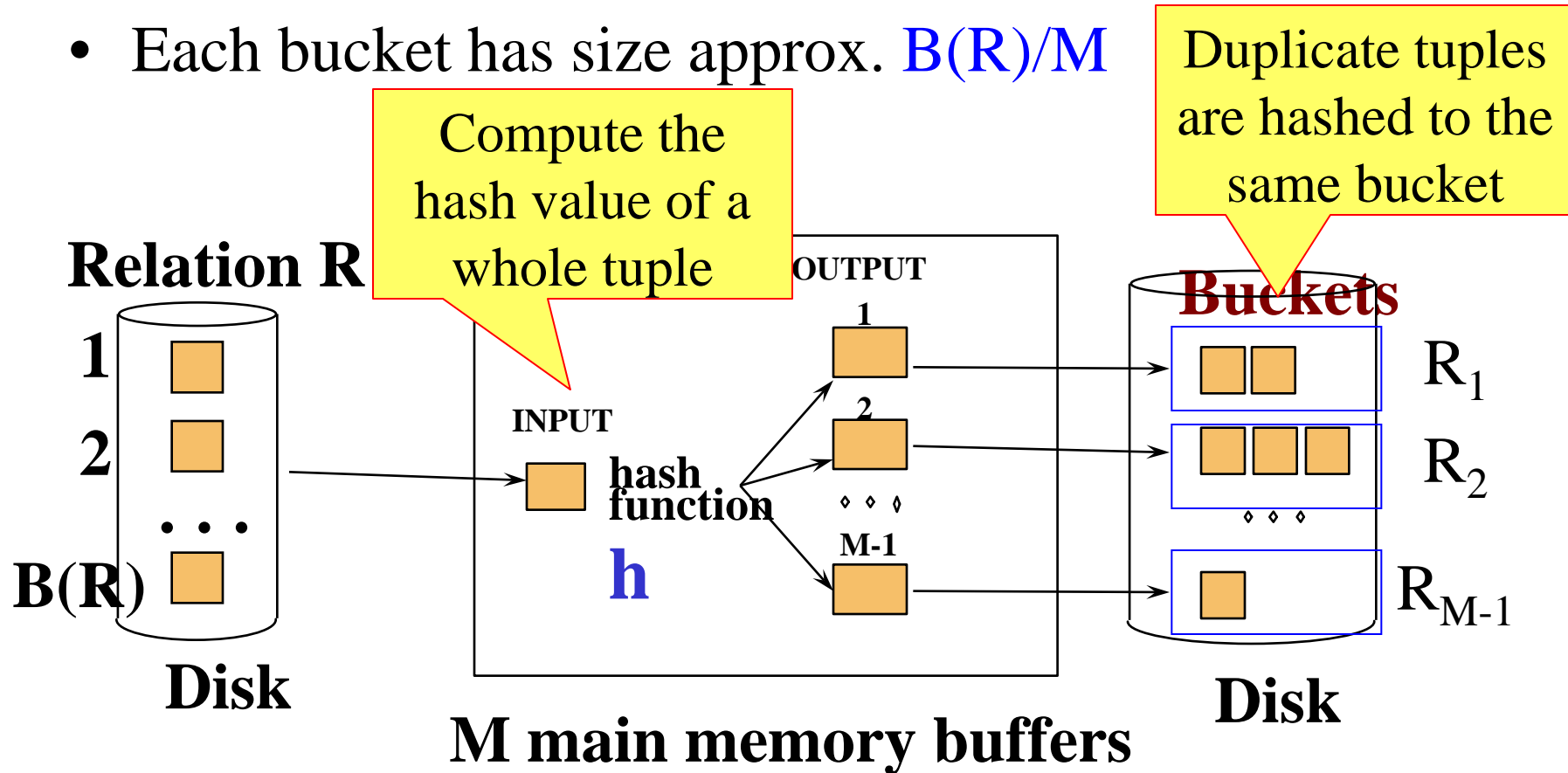
"Partitioning" picture

# Hashing-Based Algorithm for $\delta$

- Recall:  $\delta(R)$  = duplicate elimination
- Step 1. Partition  $R$  into  $(M-1)$  buckets
- Step 2. Apply  $\delta$  to each bucket (must read in main memory)
- Cost:  $3B(R)$
- Assumption:  $B(R) \leq M^2$ 
  - To be more exact:  $B(R)/(M-1) \leq M-2$

# Two-Pass Duplicate Elimination Based on Hashing

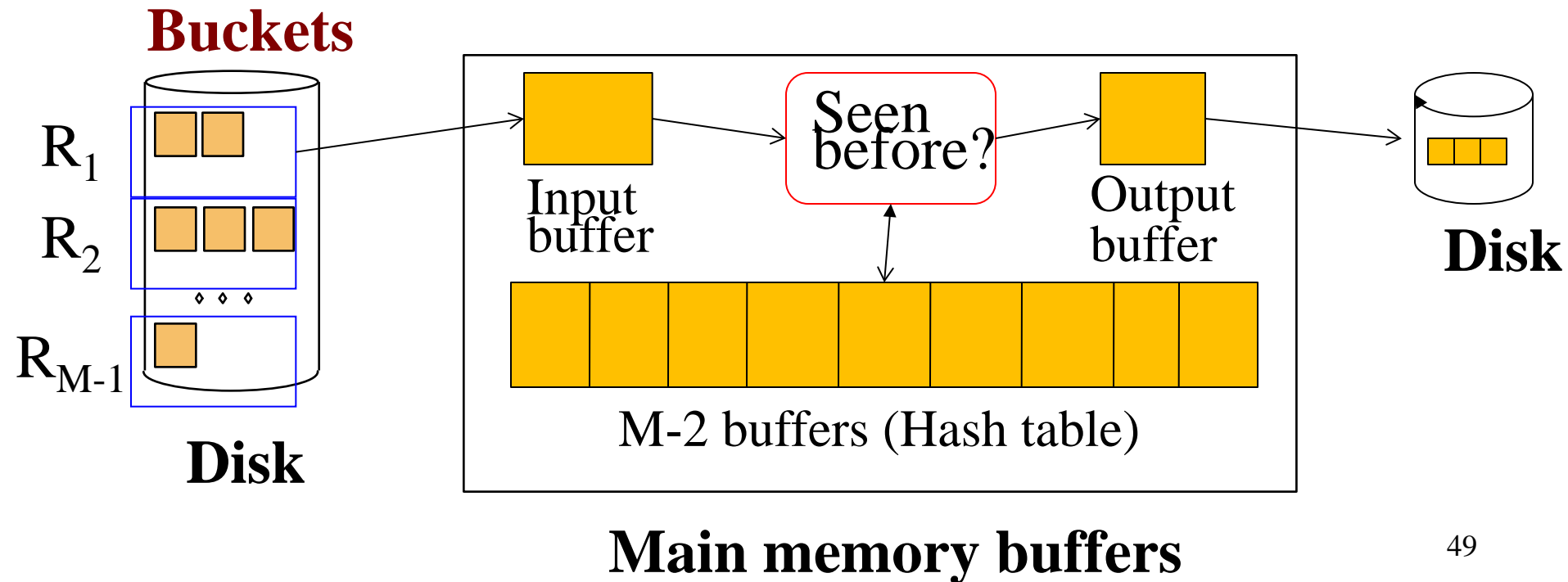
- Idea: partition a relation  $R$  into buckets, on disk
- Each bucket has size approx.  $B(R)/M$





# Two Pass Duplicate Elimination Based on Hashing

- Does each bucket fit in main memory ?
  - Yes if  $B(R)/M \leq M$  (i.e.,  $B(R) \leq M^2$ )
- Apply the one-pass  $\delta$  algorithm for each  $R_i$



# Partitioned Hash Join

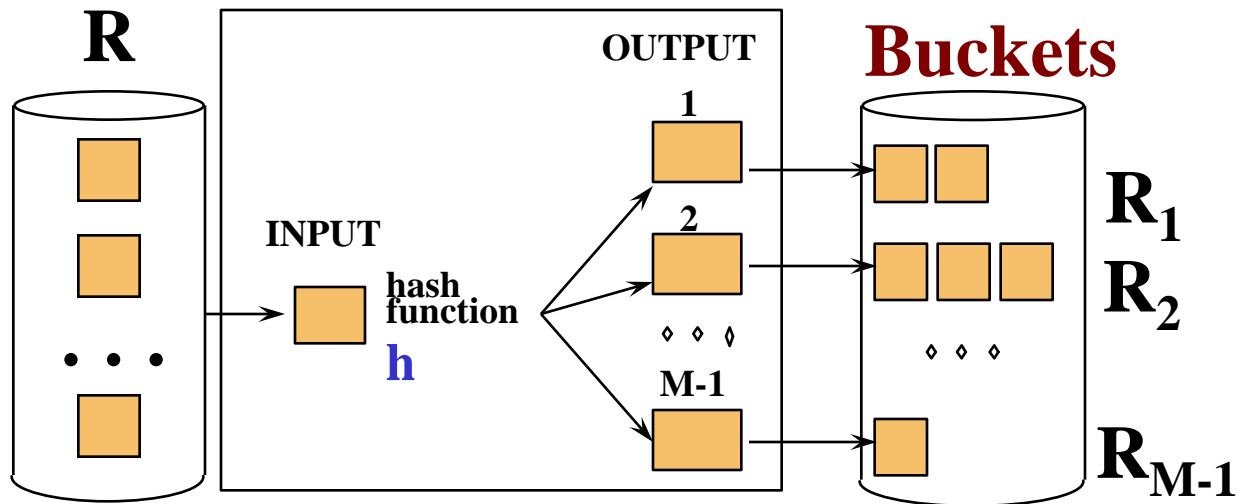
$R \bowtie S$

- Step 1:
  - Hash S into  $M - 1$  buckets
  - send all buckets to disk
- Step 2
  - Hash R into  $M - 1$  buckets
  - Send all buckets to disk
- Step 3
  - Join every pair of **corresponding** buckets

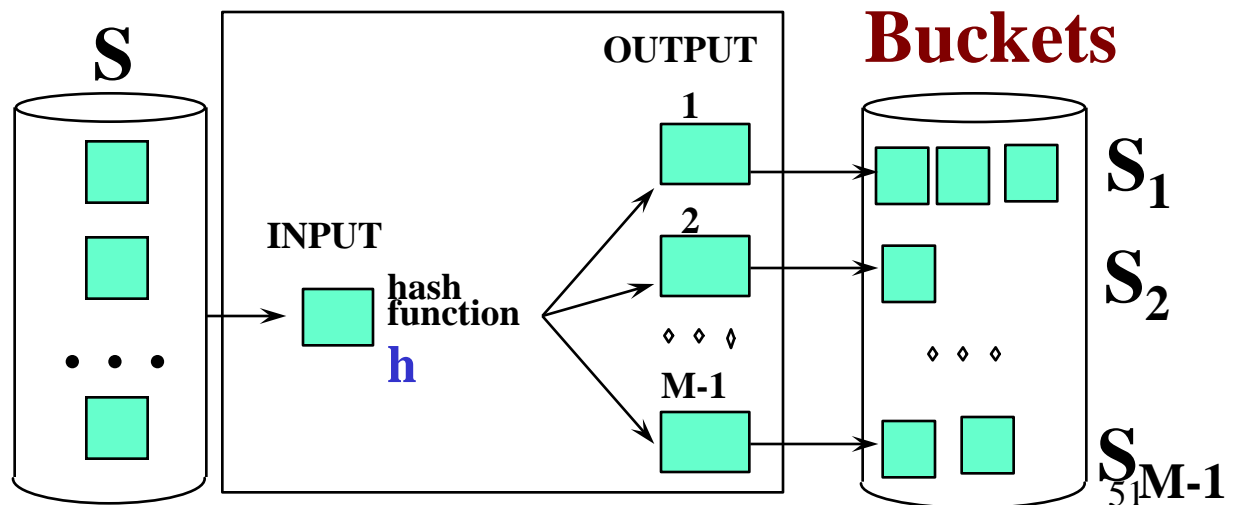
# Partitioned Hash-Join

- Partition tuples in  $R$  and  $S$  using join attributes as key for hash
- Tuples in partition  $R_i$  only match tuples in partition  $S_i$ .

## Relation

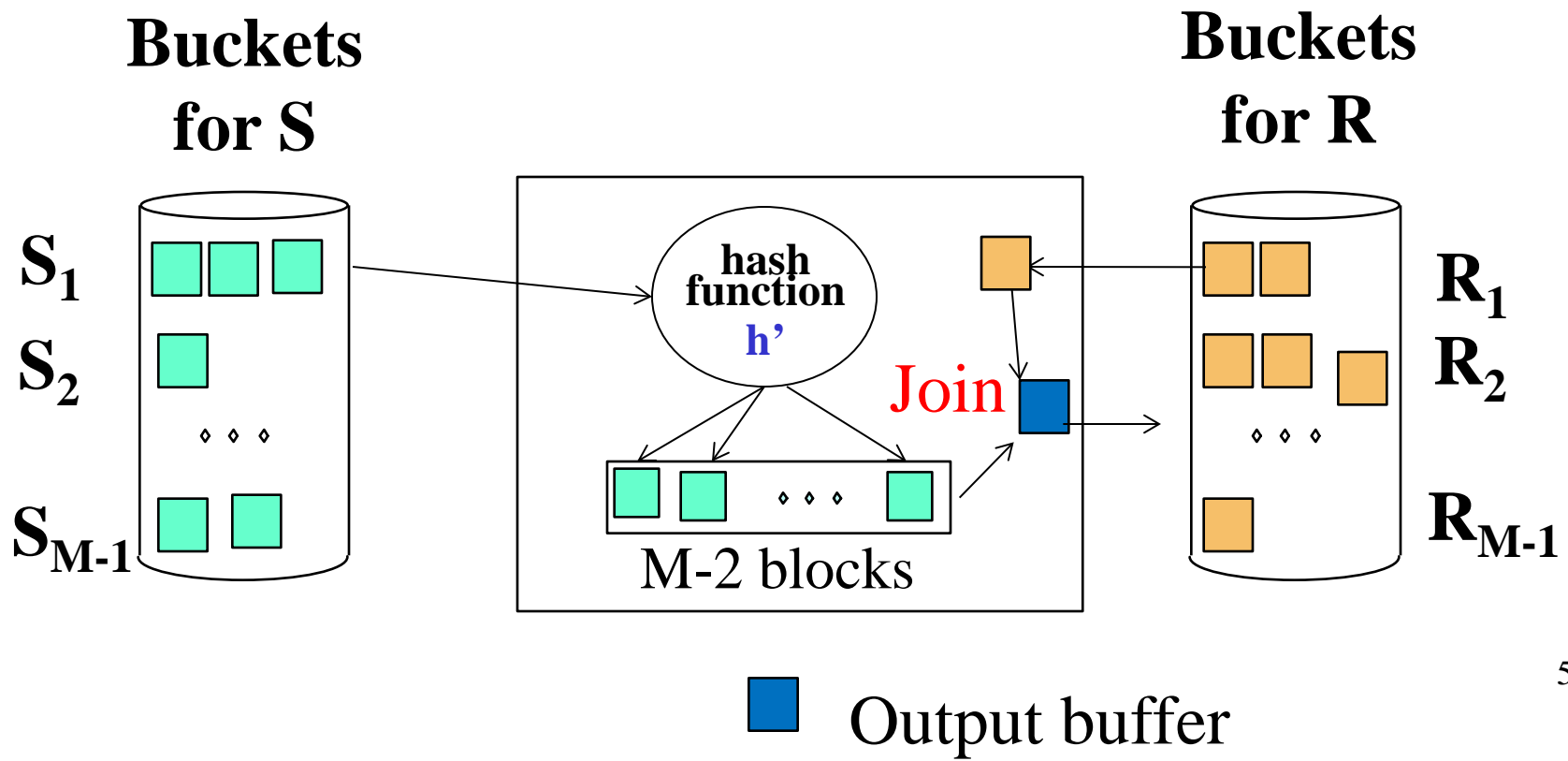


## Relation



# Partitioned Hash-Join: Second Pass

- Read in a partition of  $S_i$ , hash it using another hash function  $h'$
- Load matching partition  $R_i$ , one block at a time, output joining tuples.



# Partitioned Hash Join

- Cost:  $3B(R) + 3B(S)$
- Assumption:  $\min(B(R), B(S)) \leq M^2$ 
  - Or to be more exact:  $\min(B(R), B(S))/(M-1) \leq M-2$

# Example

- Suppose  $M = 101$  blocks (i.e., pages),  $B(R) = 1,000$  blocks,  $B(S) = 5,000$  blocks
- Cost of  $R \bowtie S$  using partitioned hash join algorithm
- What if  $B(S) = 50,000$  blocks?

# Sort-based vs. Hash-based Algorithms

- Hash-based algorithms for binary operations have a size requirement only on the smaller of two input relations
- Sort-based algorithms sometimes allow us to produce a result in sorted order and take advantage of that sort later
- Hash-based algorithm depends on the buckets being of equal size, which may not be true if data are skewed

# Index-Based Algorithms

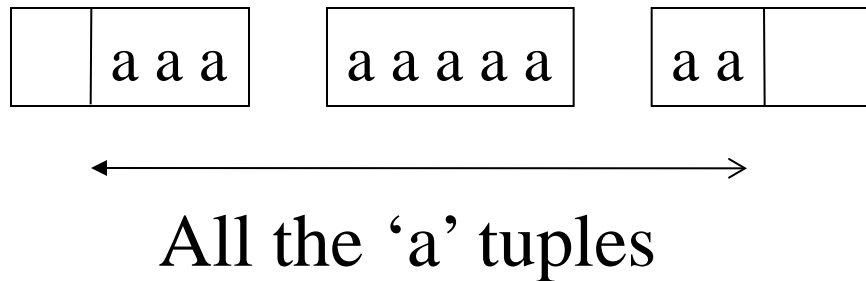


# Index-based Algorithms

- The existence of an index on one or more attributes of a relation makes available some algorithms that would not be feasible without the index
- Useful for selection operations
- Also, algorithms for join and other binary operations use indexes to good advantage

# Clustered indexes

- In a clustered index, all tuples with the same value of the search key appear on roughly as the number of blocks as can hold them
  - That is, they are clustered together



# Index Based Selection

- Selection on equality:  $\sigma_{a=v}(R)$
- Clustered index on attribute  $a$ : cost  $B(R)/V(R,a)$
- Unclustered index on  $a$ : cost  $T(R)/V(R,a)$

We here ignore the cost of reading index blocks

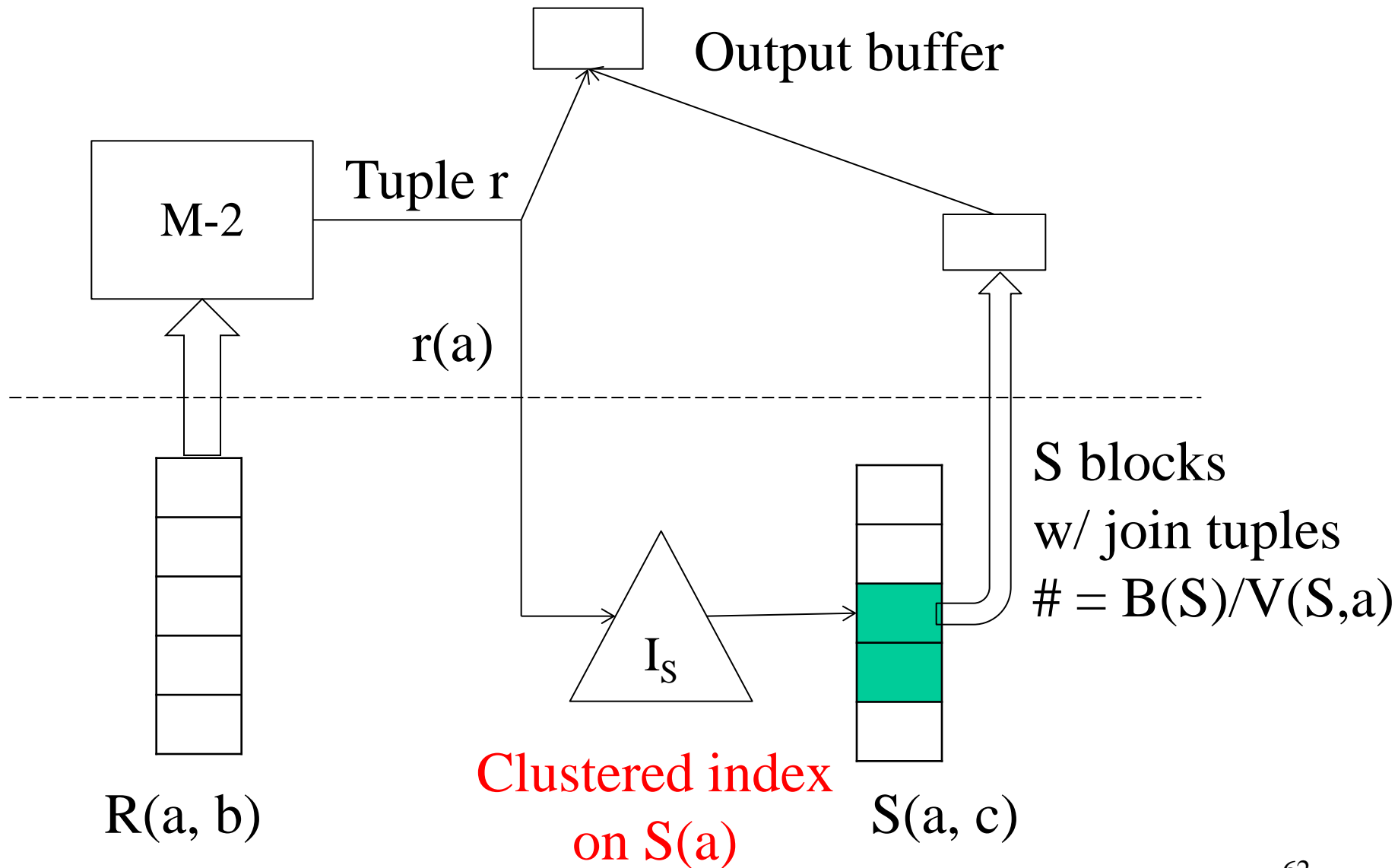
# Index Based Selection

- Example:  $B(R) = 2000$ ,  $T(R) = 100,000$ ,  $V(R, a) = 20$ , compute the cost of  $\sigma_{a=v}(R)$
- Cost of using table scan:
  - If  $R$  is clustered:  $B(R) = 2000$  I/Os
  - If  $R$  is unclustered:  $T(R) = 100,000$  I/Os
- Cost of index-based selection:
  - If index is clustered:  $B(R)/V(R,a) = 100$
  - If index is unclustered:  $T(R)/V(R,a) = 5000$

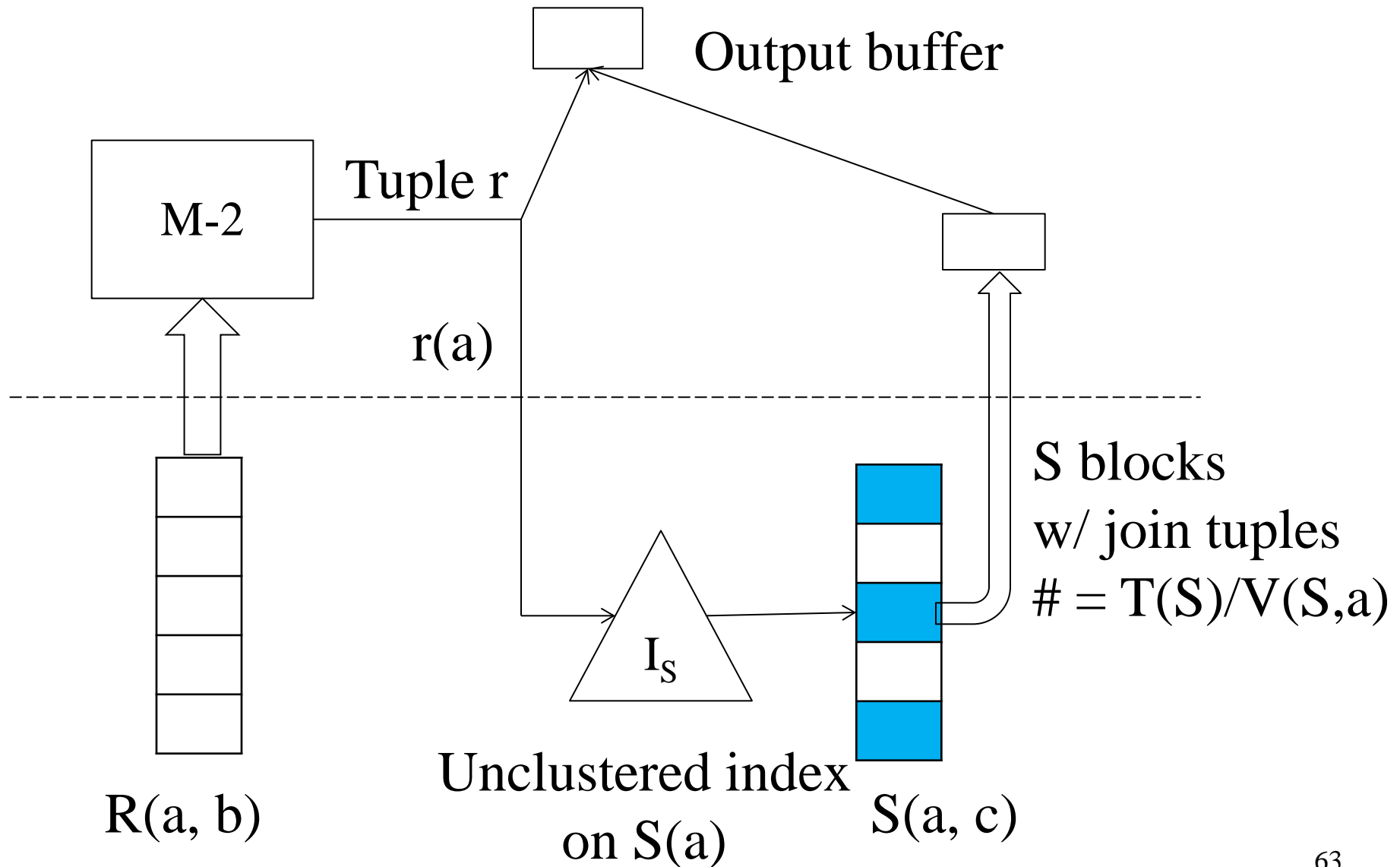
# Index-Based Join

- $R \bowtie S$
- Assume  $S$  has an index on the join attribute
- Iterate over  $R$ , for each tuple, fetch corresponding tuple(s) from  $S$
- Assume  $R$  is clustered. Cost:
  - If index is clustered:  $B(R) + T(R)B(S)/V(S,a)$
  - If index is unclustered:  $B(R) + T(R)T(S)/V(S,a)$

# Index-Based Join: Clustered Index



# Index-Based Join: Unclustered Index



# Example

- Suppose  $M = 102$  blocks (i.e., pages)
- $R(a, b) \bowtie S(a, b)$
- $S$  has an index on attribute "a" and  $V(S, a) = 100$
- $B(R) = 1,000$  blocks,  $B(S) = 5,000$  blocks
- $T(R) = 10,000$  tuples,  $T(S) = 50,000$  tuples
- Cost of  $R \bowtie S$  using index-based join algorithm
  - Index on  $S(a)$  is clustered
  - Index on  $S(a)$  is unclustered



# Index-Based Join: Two Indexes

- Assume both R and S have a clustered index (e.g., B+-tree) on the join attribute
- Then can perform a sort-merge join where sorting is already done (for free)
- Cost:  $B(R) + B(S)$

