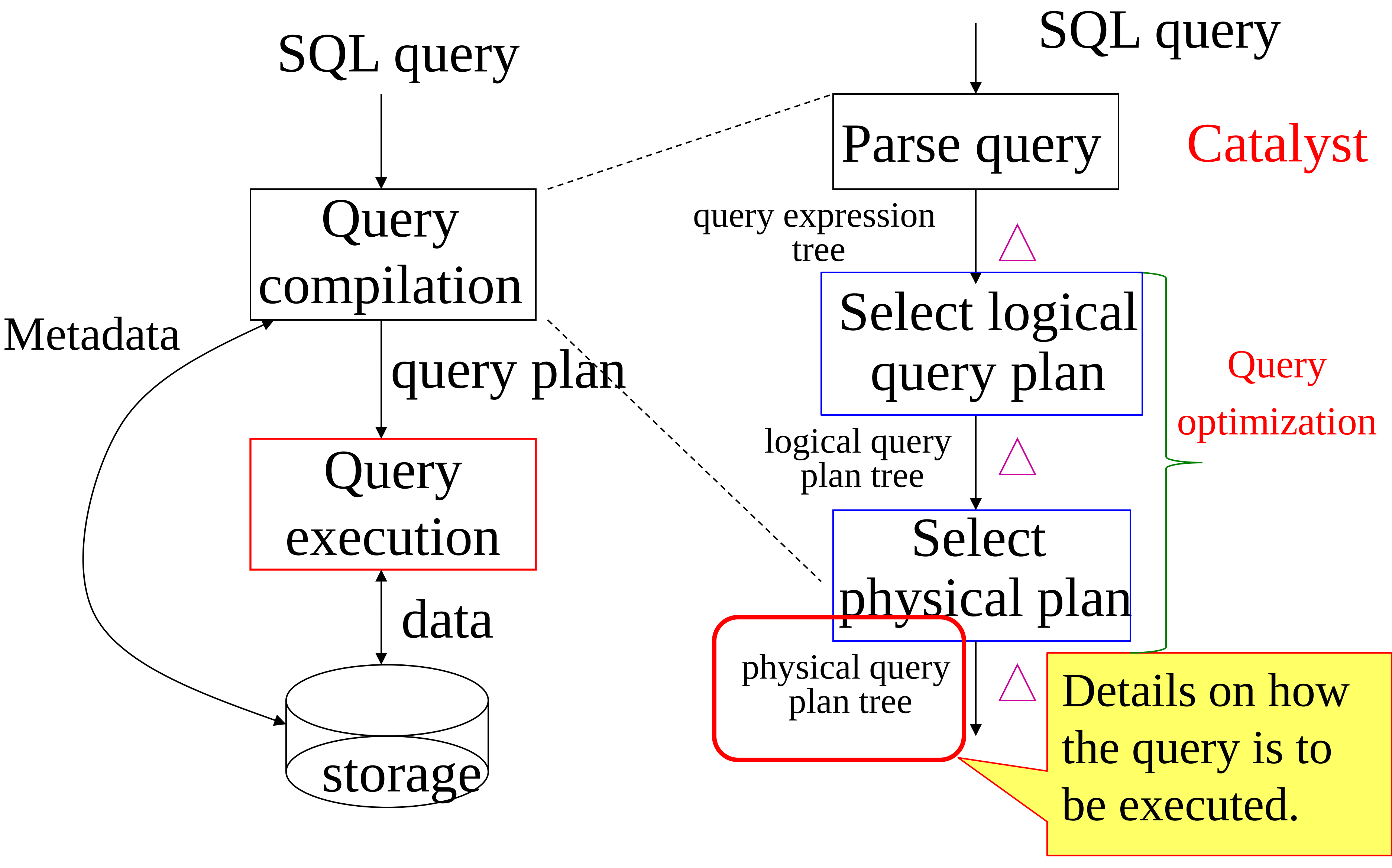
Query Execution

DSCI 551
Wensheng Wu

Components of Query Processor



Converting SQL to Logical Plans

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

$$\Pi_{a1,\ldots,an}(\sigma_C(R_1 \times R_2 \times \ldots \times R_k))$$

Select $b_1, \ldots, b_{m,}$ aggs
From $R_1, \ldots, R_k$
Where C
Group by $b_1, \ldots, b_m$

$$\gamma_{b1, \ldots, bm, aggs}(\sigma_C(R_1 \times R_2 \times \ldots \times R_k))$$

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

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

- 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: 1.x
- 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 index-based
  - 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
```

NLJ (Purchase outer) :

    for p in Purchase:
        for q in Person:
            if (p.buyer = q.name)

NLJ (Person is outer):…

Construct logical plan…

P (hash on buyer) =>
P0 (john), P1
Q (on name) =>
Q0 (john), Q1

join P1 with

hash function

h(x) = x % 2

h(5) = 1

h(10) = 0

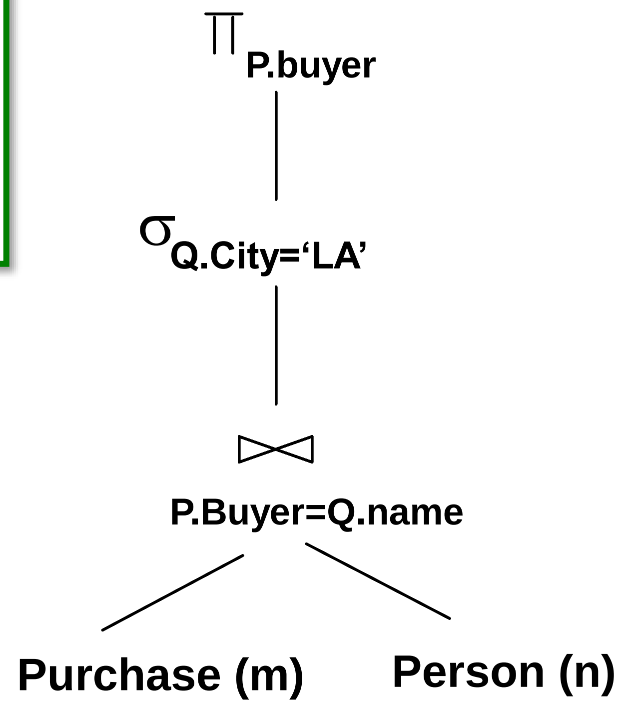
h(john) = (110 + 123 + 92 + 98) % 2

= 1 => P2

h(mary) = 0 => P1

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

Notes

h(John) = (74+111+104+110)
% 2 + 1 = 2

R join S join T

dynamic programming

Hive HiveQL => MapReduce
Spark (Catalyst)

Example

M = 1GB

	Purchase	Person
A:	100MB	200MB
B:	100MB	2GB
C:	2GB	2GB
	R1: 500M	P1: 500M
	R2: 500M	P2: 500M
	R3: 500M	P3: 500M
	R4: 500M	P4: 500M

hash-based:

R1 join P1

R1 join P2

R1 join P3

R1 join P4

R2 join P1

…

Example (cont'd)

M = 1GB

	Purchase	Person
A:	100MB	200MB
B:	100MB	2GB
C:	2GB	2GB
	R1: 500M	P1: 500M
	R2: 500M	P2: 500M
	R3: 500M	P3: 500M
	R4: 500M	P4: 500M

Block-based NLJ algorithm

4* (500 + 2G)

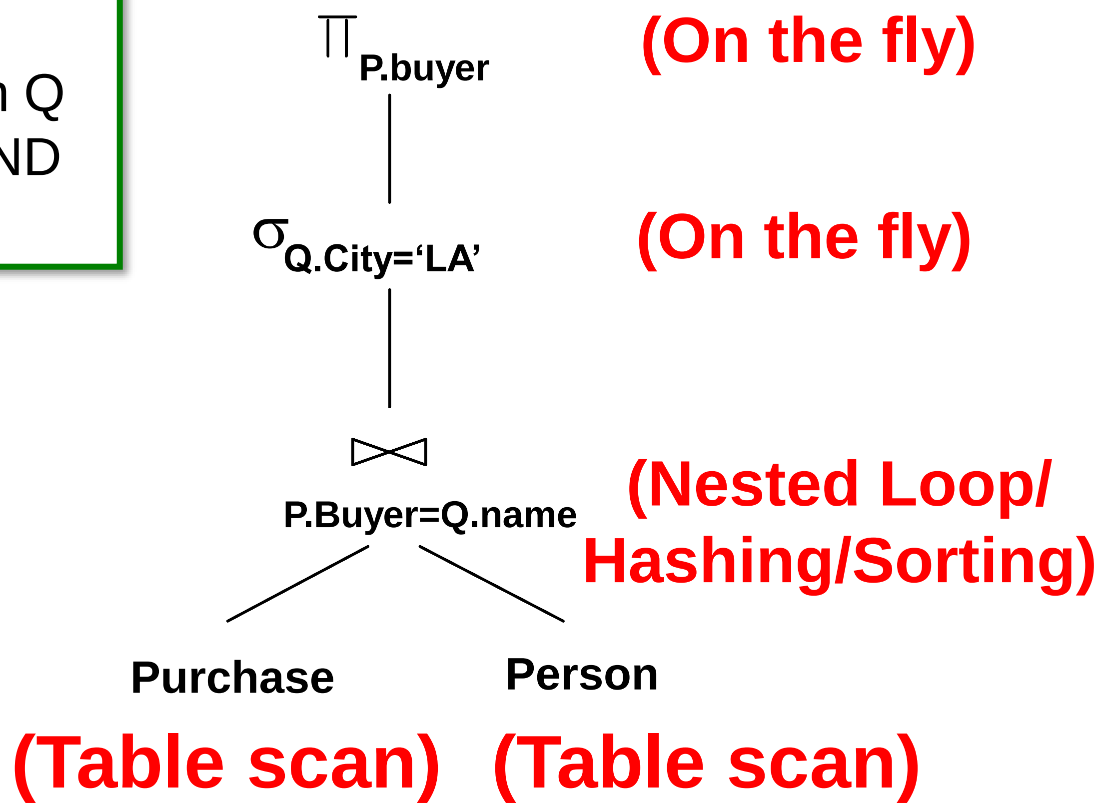
= 4* 500 + 4 * 2G

= 2GB + 4 * 2GB

= 2GB + 2GB/500MB * 2GB

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

Notes

```
class Operator:
   def open():
   def next():
   def close():

class ProjOp(Operator):
    def open()
    def next():
    …
class FilterOp(Operator):
     def open()
class JoinOp(Operator):
    def next():
```

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

Selectivity

- The larger V(R,a), the more selective a is for R

- Employee(ssn, name, age, gender)
  - Which of the above attributes is most/least selective?
  - V(Employee, gender) = 2
  - V(Employee, ssn) = n

I/O Cost

- # of blocks read from or written to disk

- Recall that disk reads/writes data in the unit of block

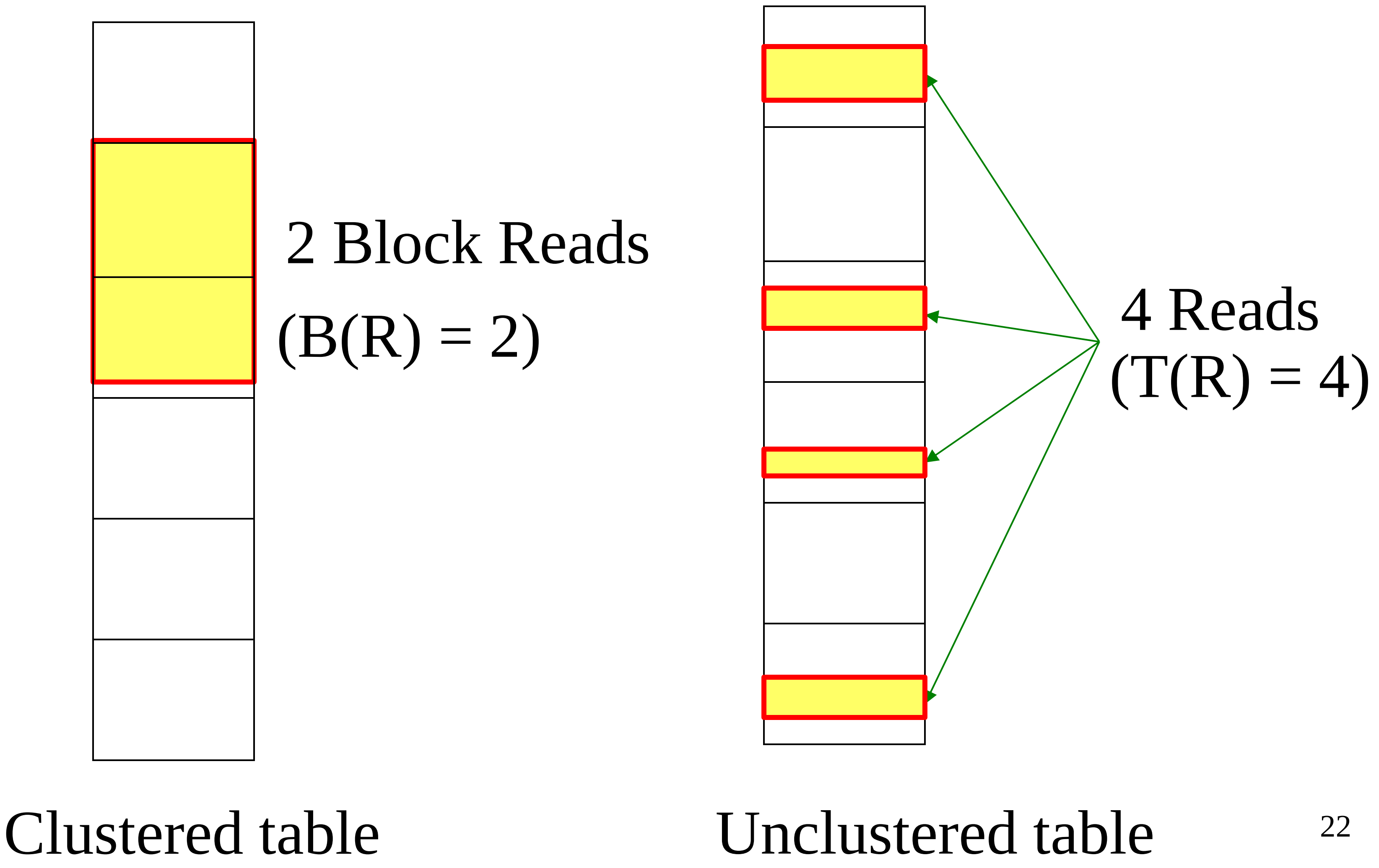
Scanning Tables

- 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/Unclusterted Tables



Cost of the Scan Operator

- Clustered relation:
  - Table scan: B(R)

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

- Unclustered relation:
  - T(R)

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 (1.x)
  - 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

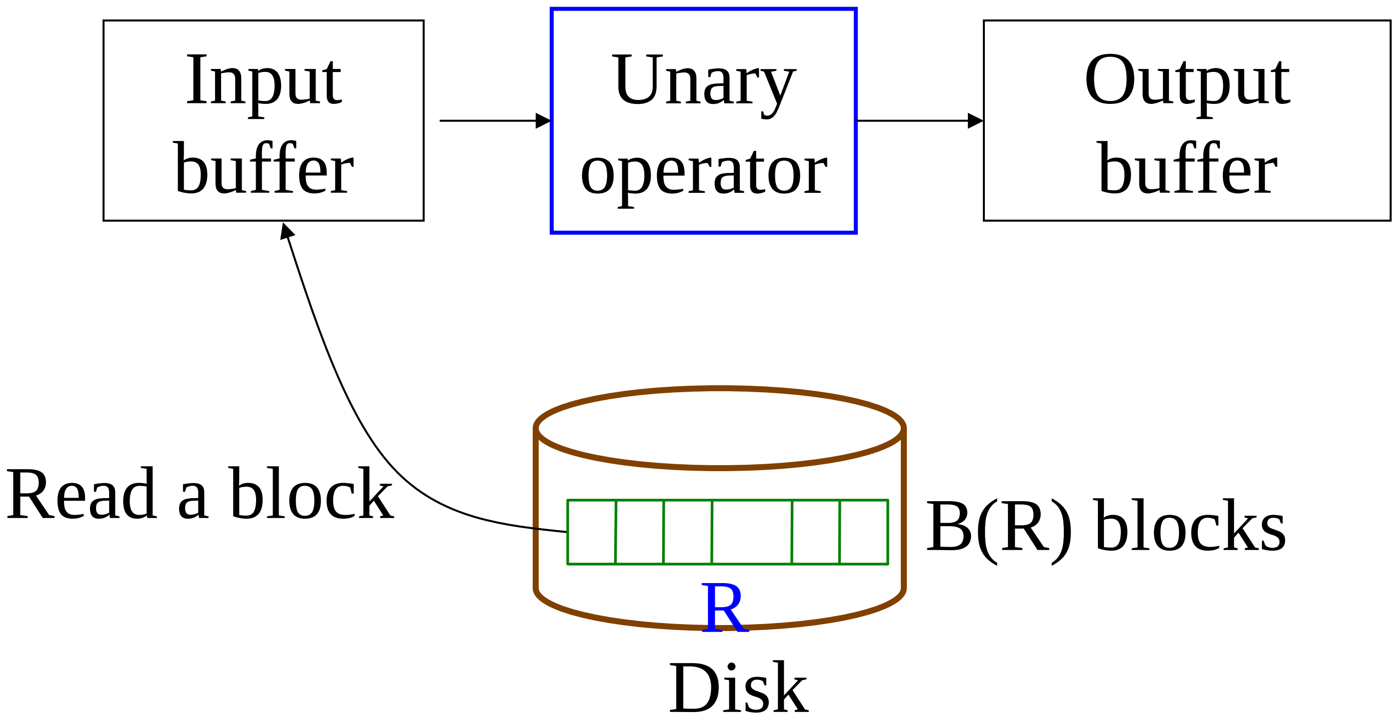
- K-pass algorithms
  - If data are too big or memory is too small, the algorithm may need $k > 2$ passes over the data

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

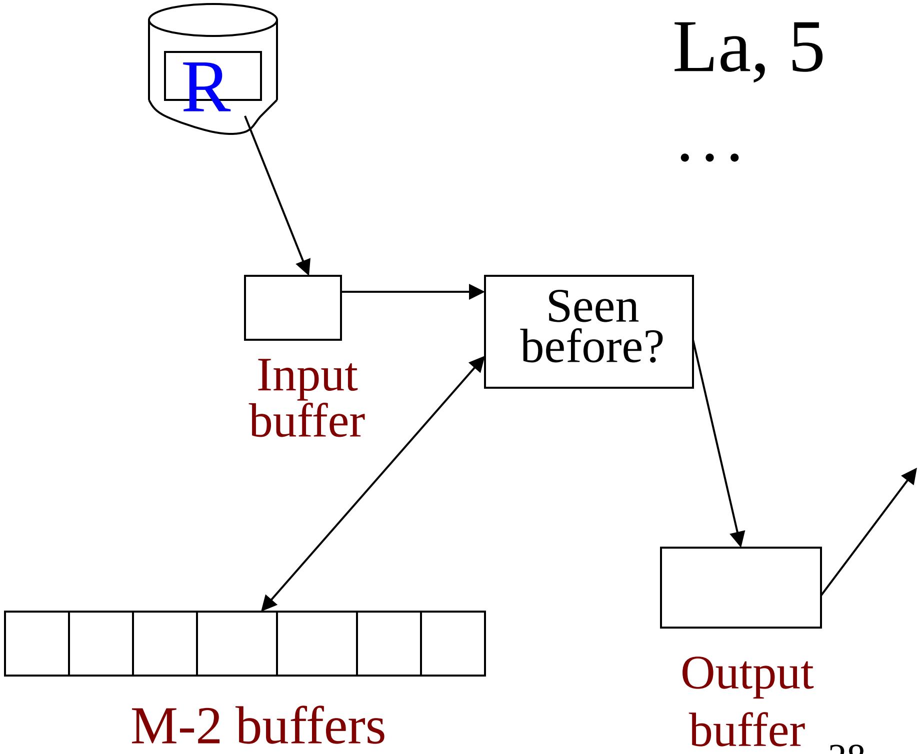
Duplicate elimination δ(R)

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

  B(δ(R)) <= M-2

  or roughly M

La, 2
La, 3
Sfo, 2
Sfo, 4
La, 5
…



One-pass Algorithms

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

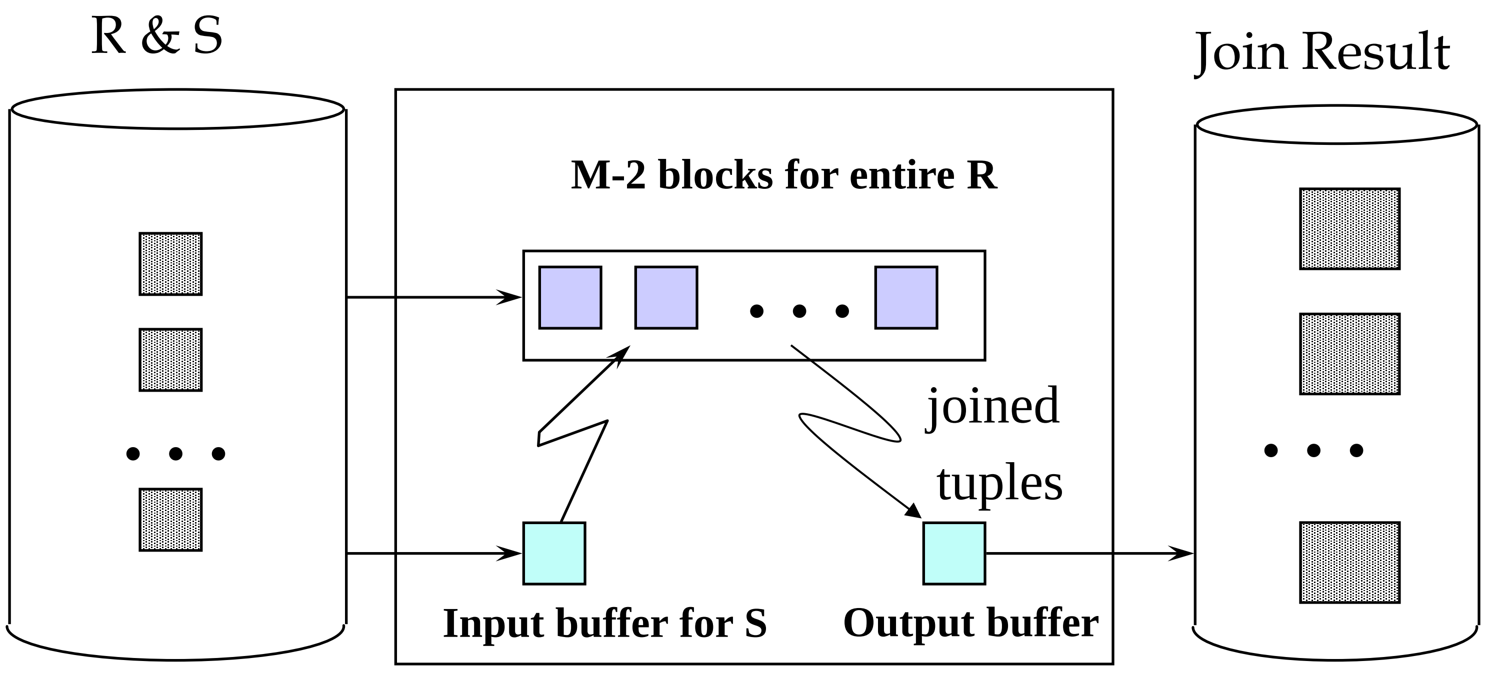
- Need to keep a dictionary in memory
  - Also store the sum(price) for each city
- Cost: B(R)
- Assumption: number of cities and sums fit in memory

One-pass Algorithms

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

- Assumption: min(B(R), B(S)) <= 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) (assume both are clustered)
- E.g. $R \cap S$ (assume set-based, no duplicates)
  - 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

One-pass join algorithm



$M = 102$

$B(R) \le 100$

Nested-loop join
(none of tables fits in memory…)

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

<u>for</u> each (M-2) blocks b_r of R <u>do</u>

    <u>for</u> each block b_s of S <u>do</u>

        <u>for</u> each tuple r in b_r <u>do</u>

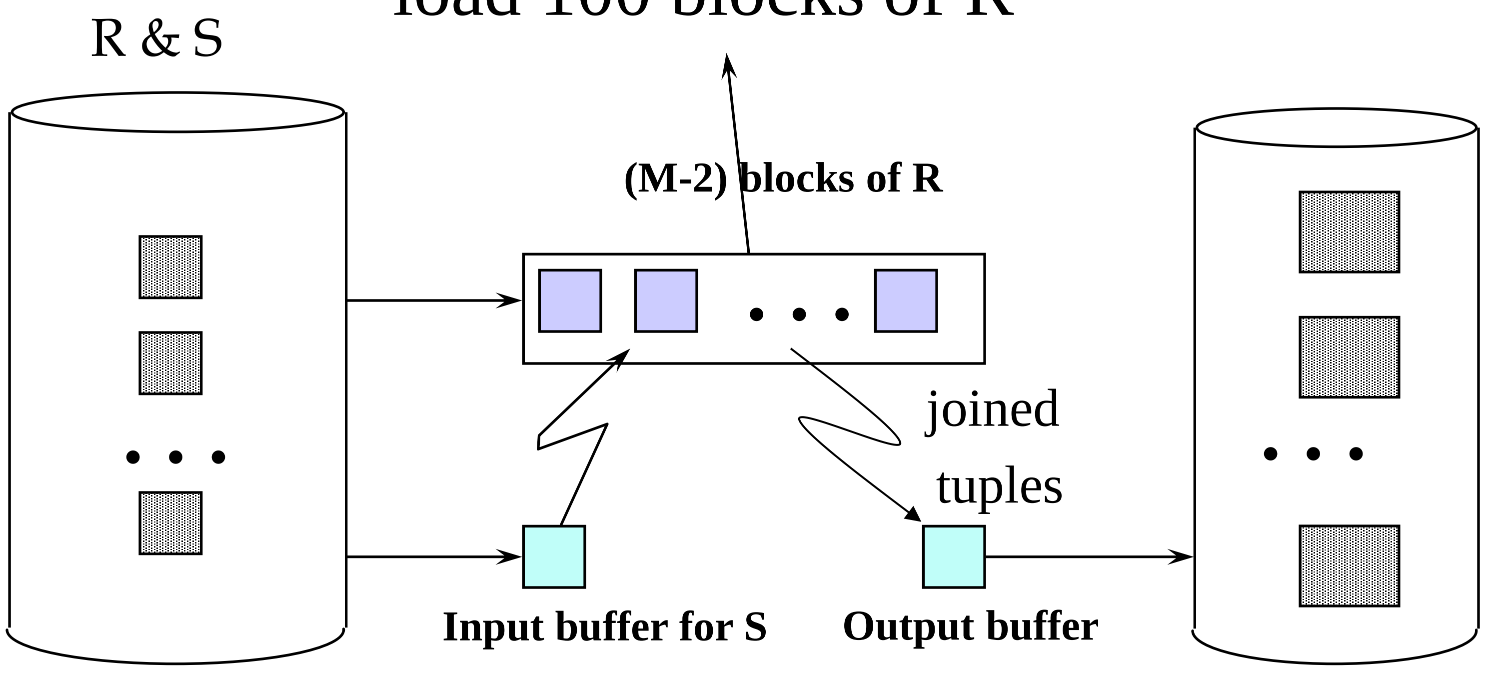
            <u>for</u> each tuple s in b_s <u>do</u>

                <u>if</u> r and s join <u>then</u> output(r,s)

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

Block-based Nested Loop Joins

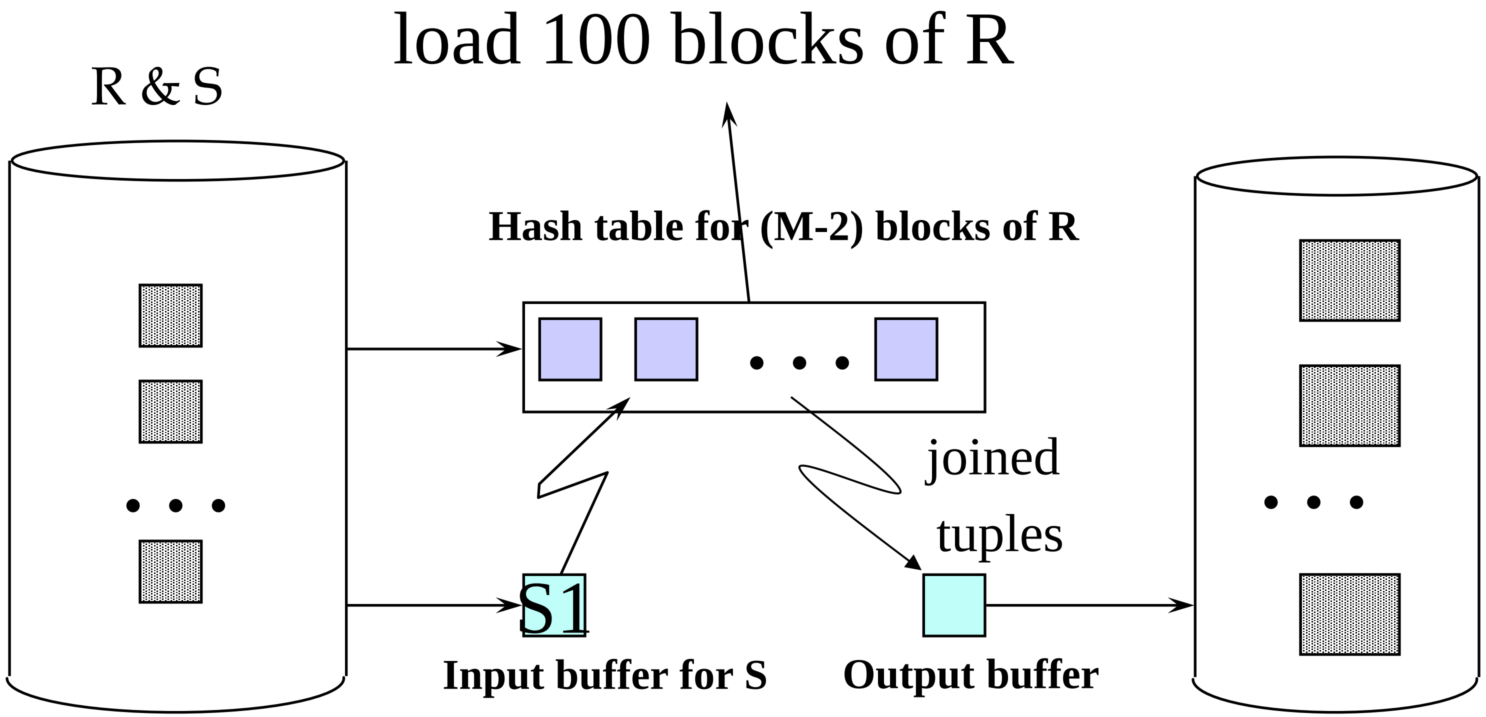
load 100 blocks of R



cost(R is outer) = B(R) + B(R)/(M-2) * B(S)

cost(S is outer) = B(S) + B(S)/(M-2) * B(R)

Block-based Nested Loop Joins



R outer: B(R) + B(R)/(M-2) * B(S)

S outer: B(S) + B(S)/(M-2) * B(R)

M-2 >= 1 => M >= 3

Notes

B(R) = 1000 B(S) = 5000, M = 102 (smallest M = 3)

- load 1st 100 blocks of R (R1), join with S (one block a time)
  100 + B(S)
- load 2nd 100 blocks of R, join with S
  100 + B(S)

…

- load 10th 100 blocks, join with S

R is outer:

* Total cost: B(R) + B(R)/(M-2) * B(S)
  = 1000 + 10 * 5000 = 1000 + 50,000

S is outer:

* Total cost: B(S) + B(S)/(M-2) * B(R)

lower cost if smaller relation (R) placed in the outer

notes

- load 1st 100 blocks of R
  - load one block of S for 5000 times
    => making one pass through S
- load 2nd 100 blocks of R
  ⇒ making one pass through S

…

- load 10th 100 blocks of R
  => make one pass through S

cost (R is outer):

- R: one pass
- S: B(R)/(M-2) * B(S)
  - 10 passes through S

⇒ B(R) + B(R)/(M-2) * B(S)

cost (S is outer):

- S: one pass
- R: B(S)/(M-2) * B(R)
  - 50 passes through R

⇒ B(S) + B(S)/(M-2) * B(R)

notes

for every (M-2) blocks of R: // Ri

    for every block of S: // Si

        join rows in Ri with rows in Si

        output

I/O cost:

    cost of reading R: B(R)

    cost of reading S: B(R)/(M-2) * B(S)

total cost:

    B(R) + B(R)B(S)/(M-2)

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) = 1000 blocks, B(S) = 5,000 blocks
  - # of chunks from R = 10, chunk size = 100 blocks

- Cost of R ⋈ S using block-based nested-loop join algorithm
  - If R is outer relation: one pass R; 10 passes through S
    - 1000 blocks + 1000/(102-2) * 5000 = 51,000
  - If S is outer relation: one pass S; 50 passes R
    - + 5000/(102-2) * 1000 = 55,000

NLJ

- select R.a, R.b, S.c
  from R, S
  where R.a = S.a

```
# tuple-based
for r in R:
  for s in S:
      if (r.a == s.a):
          output R.a, R.b, S.c
```

Notes

- M = 102
- B(R) = 1000, B(S) = 5000

- for every (M-2) blocks from R
  - for every block from S
    - join the tuples from these blocks in memory

- Cost of R being outer:
  - reading S: B(R)/(M-2) * B(S) = 10 * 5000 = 50,000
  - reading R: B(R)
  - total = B(R) + B(R)B(S)/(M-2)

Notes

- Cost of R being outer:
  - total = B(R) + B(R)B(S)/(M-2)

- Cost of S being outer:
  - total = B(S) + B(R)B(S)/(M-2)

- smaller relation in the outer
- M >= 3
- larger M, lower cost

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 buckets at a time

Duplicate Elimination $\delta(R)$ Based on Sorting

- Simple idea: sort first, then eliminate duplicates
- Pass1: 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 <= M-1 to complete the merging in the second pass
  - So $B/M \leq M - 1$

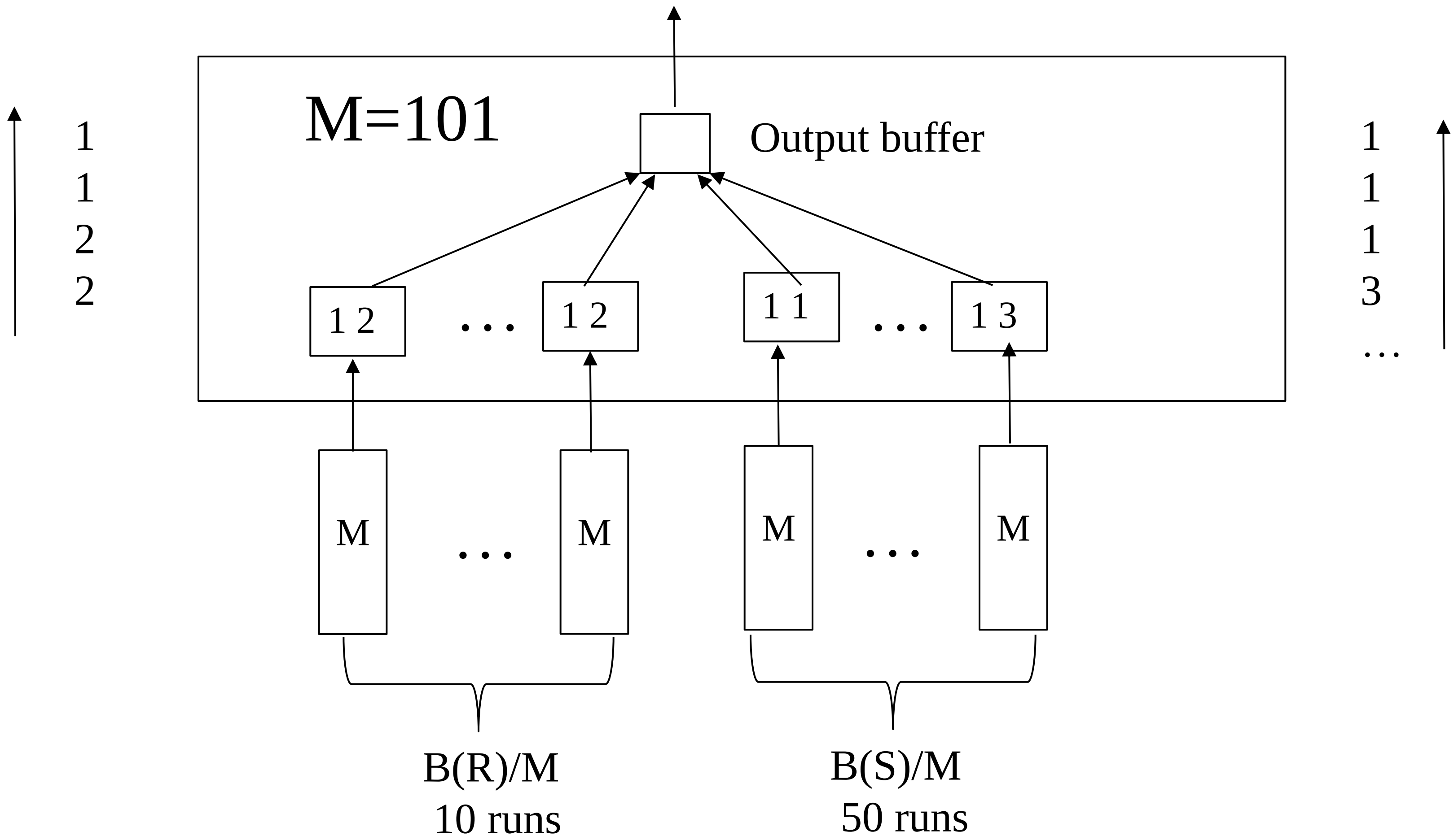
Grouping: $\gamma_{city,\ sum(price)}$ (R) Based on Sorting

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

Binary operations: R ∩ S, R U 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: $B(R)+B(S) \leq M^2$

Merging picture



B(R)/(M-1) + B(S)/(M-1) <= M-1

B(R) + B(S) <= (M-1)(M-1) ~ M^2

Notes

R join S = ?

```
select country.Name, city.Name,….
from country join city on country.Capital = city.ID
```

Notes

R(a)

1
1
2
2

S(a)

1
1
1
3

select count(*)
from R join S on R.a = S.a

= 6

notes (simple-sort)

1. completely sort R:
   - R (1000) => 10 runs => 1 run
   - cost: 4B(R)
2. completely sort S:
   - S (50,000) => 500 runs => 5 runs => 1 run
   - cost: 6B(S)
3. merge R and S (both sorted)
   - cost: B(R) + B(S)

Total cost: 5B(R) +7B(S)

Notes (sort-merge)

- R (1000 blocks) => 10 runs
  - cost: 2 B(R)
- S (50,000) => 500 runs => 5 runs
  - cost: 4 B(S)

- join by merging 10 runs with 5 runs
  - cost: B(R) + B(S)

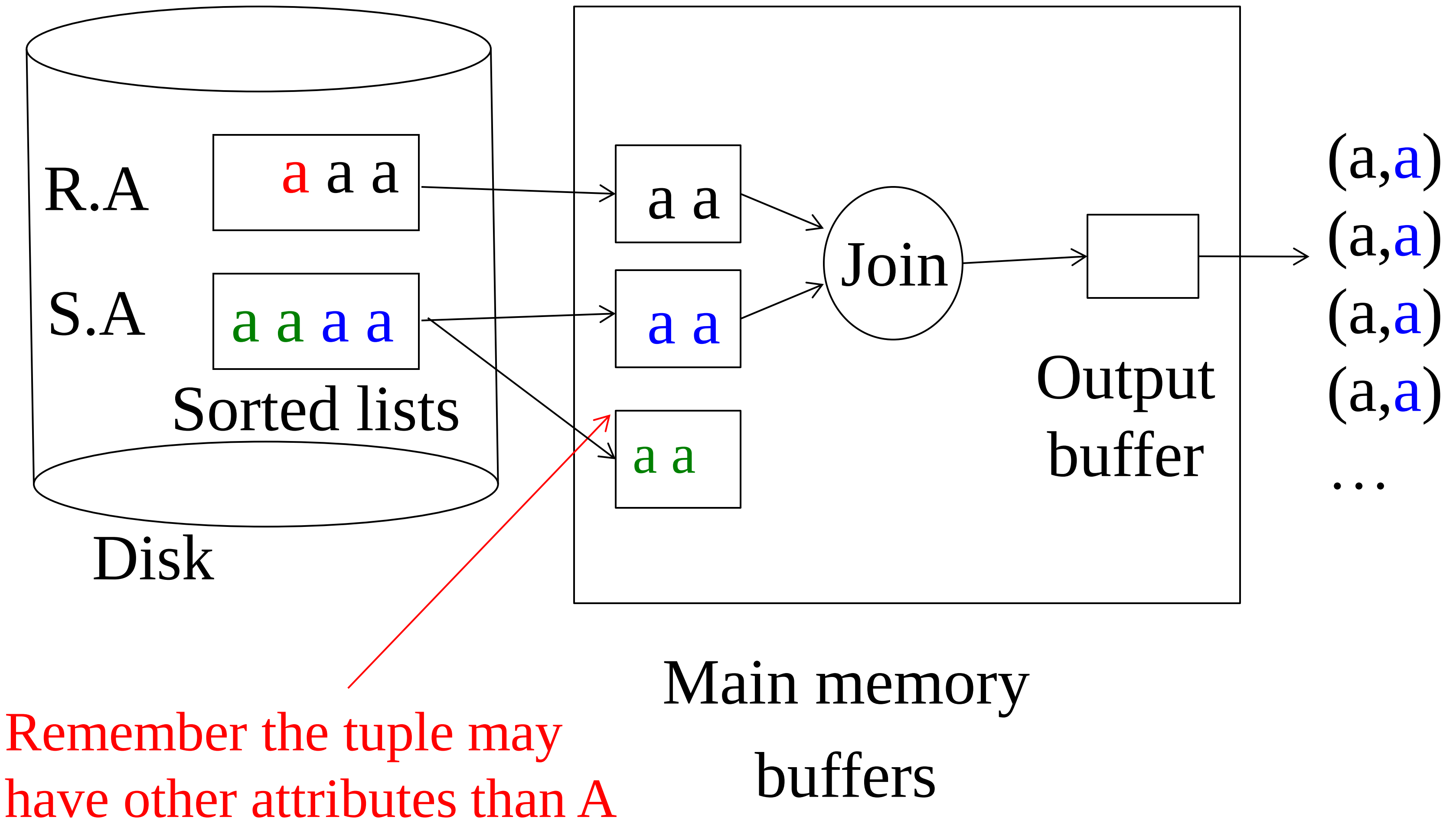
- total: 3B(R) + 5B(S)

Problem with join

- 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

Problem with join

Many tuples may have the same value on the join attribute



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) = 5000 blocks (R.a = S.a)
  - Suppose we use 100 blocks in sorting

- Cost of R $\bowtie$ S using sort-merge join algorithm
  - Pass 1: sort R => 10 runs, 100 blocks/run

    sort S => 50 runs, 100 blocks/run
  - Pass 2 (merge): B(R) + B(S)
  - total cost: 3B(R) + 3B(S) => 3B(R) + 3B(S)
- What if B(S) = 50,000 blocks?
  - 3B(R) + 5B(S)

Notes

- $M = 101$, $B(R) = 1000$, $B(S) = 5000$
  - $R.a = S.a$

- sort R =>10 runs, cost = 2*B(R)
- sort S => 50 runs, cost = 2*B(S)

- merge 10 runs from R with 50 runs from S
  - cost = B(R) + B(S)
- cost:
  - 3* B(R) + 3* B(S)

Notes

- M = 101, B(R) = 1000, B(S) = 50, 000
  - R.a = S.a

- sort R =>10 runs, cost = 2*B(R)
- sort S => 500 runs => 5 runs, cost = 4*B(S)

- merge 10 runs from R with 5 runs from S
  - cost = B(R) + B(S)
- cost:
  - 3* B(R) + 5* B(S)

Example (continued)

- Suppose M = 101 blocks (i.e., pages), B(R) = 1,000 blocks, B(S) = 50, 000 blocks (R.a = S.a)
  - Suppose we use 100 blocks in sorting

- Cost of R $\bowtie$ S using sort-merge join algorithm
  - Pass 1: sort R => 10 runs, 100 blocks/run

    sort S => 500 runs, 100 blocks/run

    (merge S) => 5 runs, 10,000 blocks/run
  - Pass 2 (merge): B(R) + B(S)
  - total cost: 3B(R) + 5B(S)
  - 3B(R) + 5B(S)

Notes

- Suppose M = 101 blocks (i.e., pages), B(R) = 1,000 blocks, B(S) = 5000 blocks (R.a = S.a)
  - Suppose we use 100 blocks in sorting

- Sort R => 10 runs, 100 blocks/run: 2B(R)
- Sort S => 50 runs, 100 blocks/run: 2B(S)
- merge 10 runs (from R) and 50 runs (from S)
  - cost: B(R) + B(S)

- Total cost: 3B(R) + 3B(S)

Notes

- M = 101 (using only 100 pages for sorting)
- R.a = S.a
- B(R) = 1000, B(S) = 5000

- Sorting:
  – Sort R into => 10 runs, cost = 2* B(R) = 2000
  – Sort S into => 50 runs, cost = 2* B(S) = 10,000
- Merging (can do 100-way):
  – only need to merge, cost = B(R) + B(S)
- Cost: 3* B(R) + 3*B(S)

Notes

- M = 101 (using only 100 pages for sorting)
- B(R) = 1000, B(S) = 50, 000
- Sorting:
  - Sort R into => 10 runs, cost = 2* B(R) = 2000
  - Sort S into => 500 runs, cost = 2* B(S)

    merge => 5 runs, cost = 2 * B(S)
- Merging (can do 100-way):
  - only need to merge, cost = B(R) + B(S)
- Cost: 3* B(R) + 5*B(S)

Notes

- Suppose M = 101 blocks (i.e., pages), B(R) = 1,000 blocks, B(S) = 50, 000 blocks (R.a = S.a)
  - Suppose we use 100 blocks in sorting
- Sort R => 10 runs, 100 blocks/run: 2B(R)
- Sort S
  - sort into 500 runs, 100 blocks/run: 2B(S)
  - merge 500 runs => 5 runs: 2B(S)
- merge 10 runs (from R) and 50 runs (from S)
  - cost: B(R) + B(S)
- Total cost: 3B(R) + 3B(S)

Notes

- Suppose M = 101 blocks (i.e., pages), B(R) = 1,000 blocks, B(S) = 5000 blocks (R.a = S.a)
  - Suppose we use 100 blocks in sorting
- Steps:
  - sort R => 10 runs, 100 blocks/run
    - cost = B(R) + B(R) = 2B(R) = 2000 blocks
  - sort S => 50 runs, 100 blocks/run
    - cost = 2 * B(S) = 10,000
  - merge runs from R and S
    - cost = B(R) + B(S)
  - Total: 3B(R) + 3B(S) = 3 * 6000 = 18,000 (blocks)

Example

- Suppose M = 101 blocks (i.e., pages), B(R) = 1,000 blocks, B(S) = 50,000 blocks (R.a = S.a)
  – Suppose we use 100 blocks in sorting
- Cost of R $\bowtie$ S using sort-merge join algorithm
  – Pass 1: sort R => 10 runs, 100 blocks/run
  sort S => 500 runs, 100 blocks/run
  extra step: merging 500 runs from S => 5 runs
  – Pass 2 (merge): B(R) + B(S)
  – total cost: 3B(R) + 3B(S) => 3B(R) + 5B(S)
- What if B(S) = 50,000 blocks?
  – 3B(R) + 5B(S)

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 we need to write result 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: $B(R) <= M^2$, $B(S) <= M^2$, 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 for sorting pass

- Cost of R $\bowtie$ S using simple sort-based join algorithm
  - Sort R (completely) => R': 4B(R) = 4000
  - Sort S => S': 4B(S) = 20,000
  - Join by merging R' with S': B(R) + B(S) (loading)
- What if B(S) = 50,000 blocks?
  - 500 runs => 5 runs => 1 run

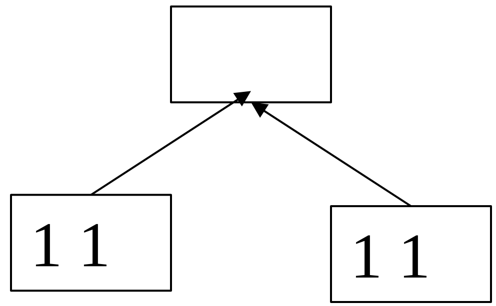
Notes

- M = 101 (but 100 for sorting)
- B(R) = 1000 blocks

- completely sort R => R':
  - pass 0: load 100 blocks of R at a time => 10 runs
  - pass 1: merge 10 runs into a single run
  - cost: 2 * 2 * 1000 = 4000 or 4B(R)
- completely sort S => S':
  - cost: 4B(S)

Notes

- join R' with S', each having a single run



R'	S'
1 1	1 1
2 2	2 3
3 3	4 5
…	…

(1, 1)
(1, 1)
(1, 1)
(1, 1)
(2, 2)
(2, 2)
(3, 3)
(3, 3)
…

Notes

- M = 101 (but 100 for sorting)
- B(R) = 1000 blocks

- completely sort R => R':
  - cost: 2 * 2 * 1000 = 4000 or 4B(R)
- completely sort S => S':
  - pass 0: 50,000 blocks => 500 runs
  - merge 1: 500 runs => 5 runs
  - merge 2: runs => 1 run
  - cost: 3* 2B(S) = 6B(S)

Notes

Sorting R (completely):

B(R) + B(R) // 10 runs

B(R) + B(R) // 1 run

= 4B(R)

Sorting S:

= 4B(S)

Merging R and S:

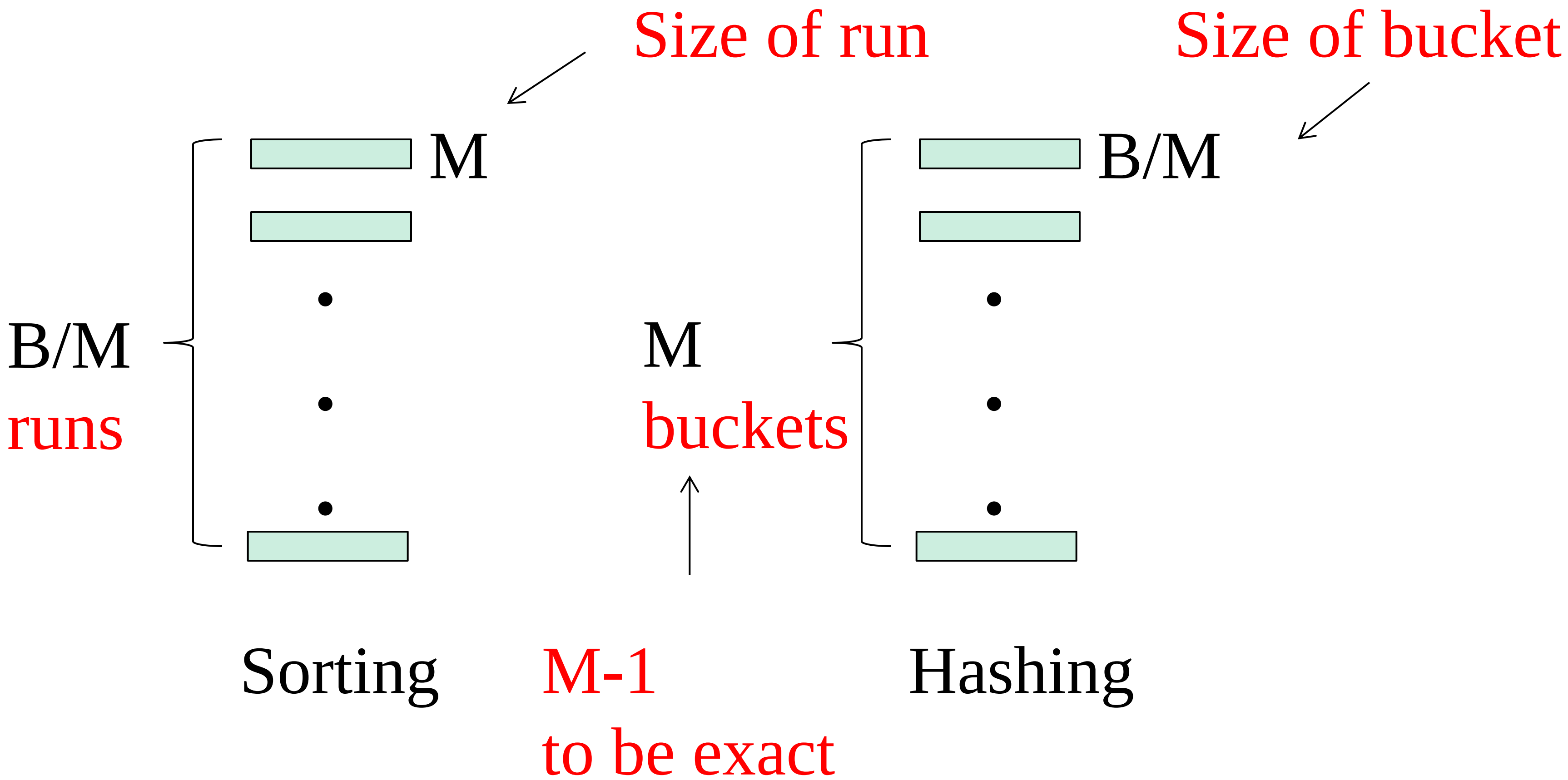
B(R) + B(S)

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

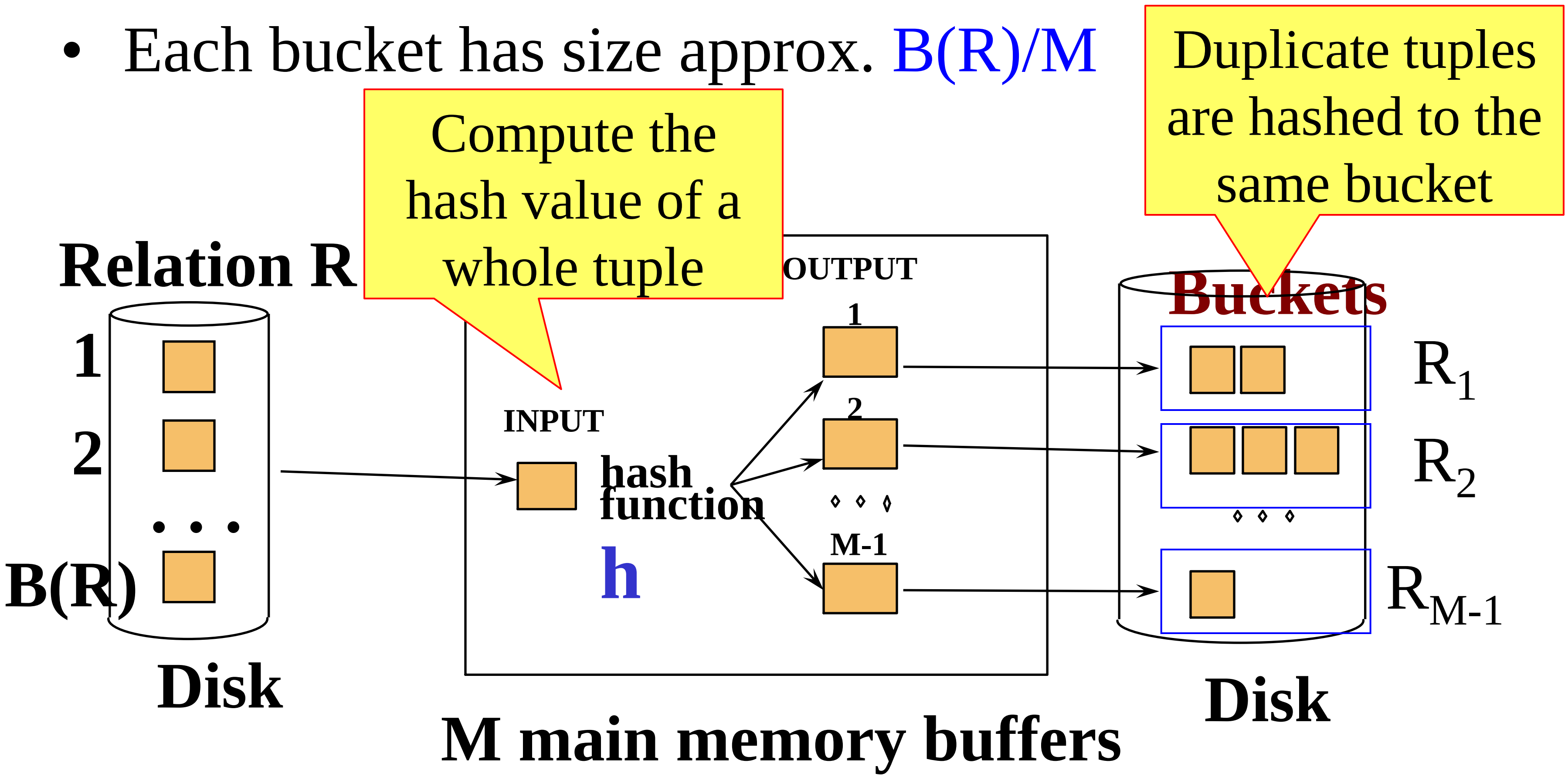
Hashing-Based Algorithm for δ

- Recall: $\delta(R)$ = duplicate elimination
- Step 1. Partition R into (M-1) buckets
- Step 2. Apply δ to each bucket (must read it into main memory)

- Cost: 3B(R)
- Assumption: $B(R) \le M^2$
  - To be more exact: $B(R)/(M-1) \le M-2$

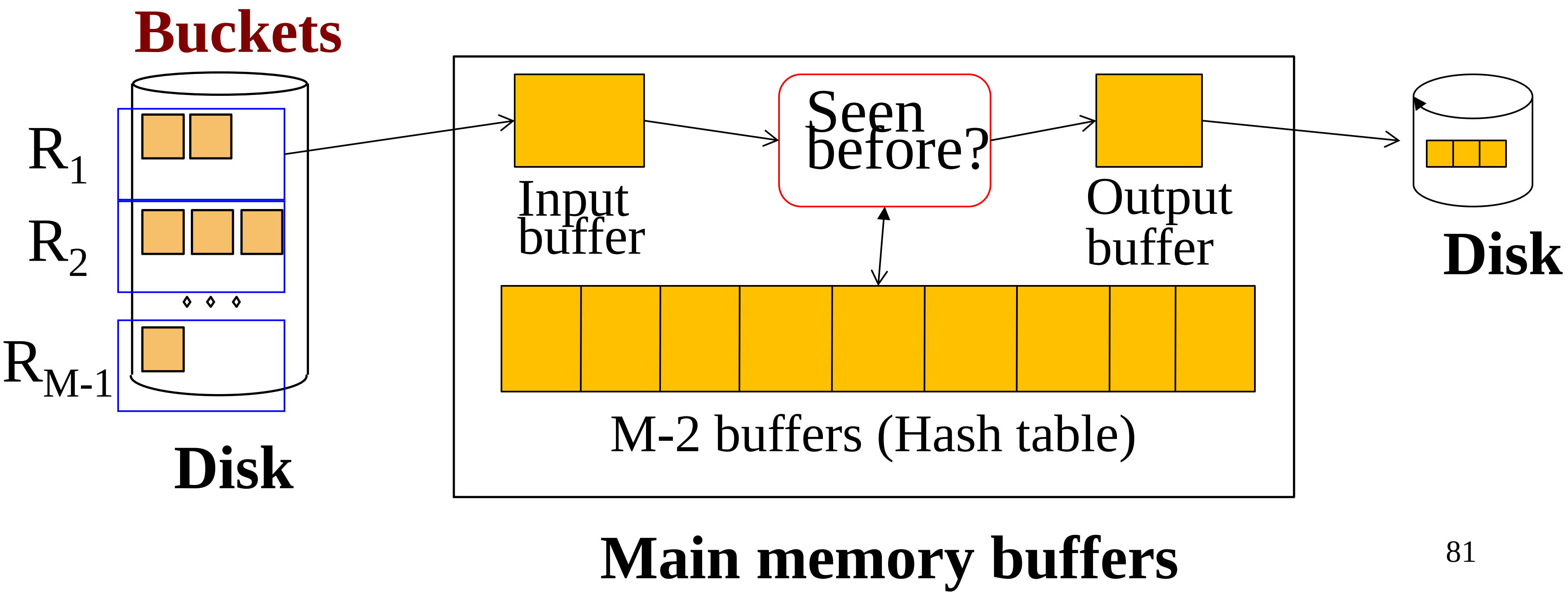
Two-Pass Duplicate Elimination Based on Hashing

- 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 $B(R)/(M-1) <= M-2$ (i.e., approx. $B(R) <= M^2$)
- Apply the one-pass δ algorithm for each R_i



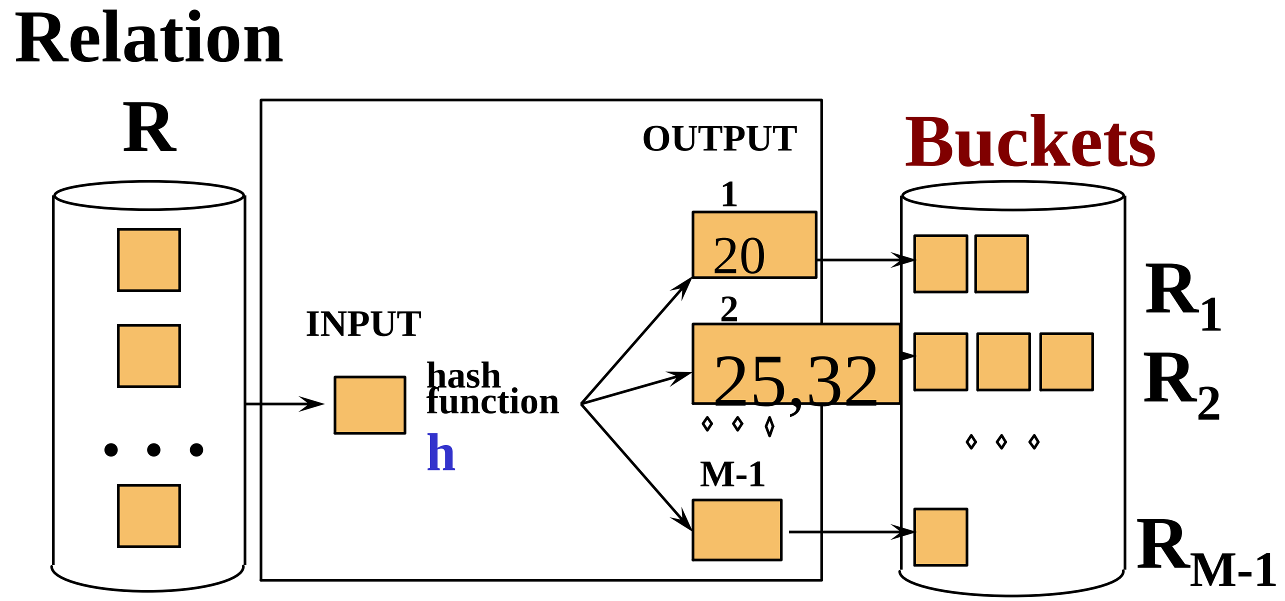
Partitioned Hash Join

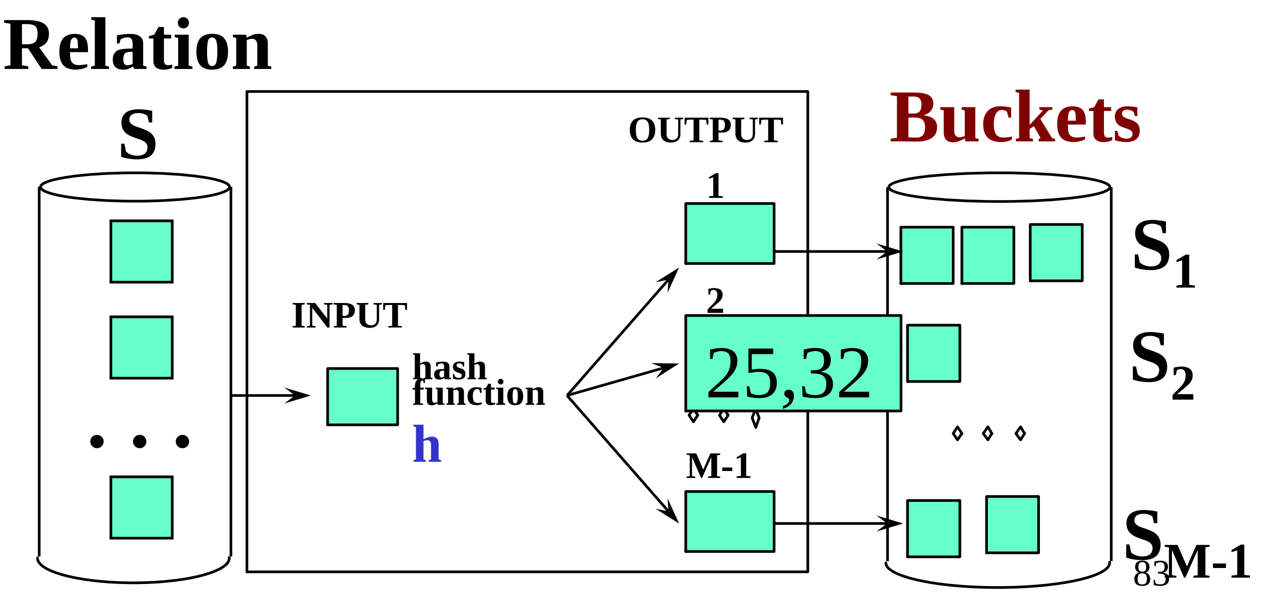
$R \bowtie S$

- Step 1.a:
  - Hash S into M – 1 buckets
  - send all buckets to disk
- Step 1.b
  - Hash R into M – 1 buckets
  - Send all buckets to disk
- Step 2
  - 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.
- R.age = S.age
- h(r.age) = h(25) = 2
- h(s.age) = h(25) = ?





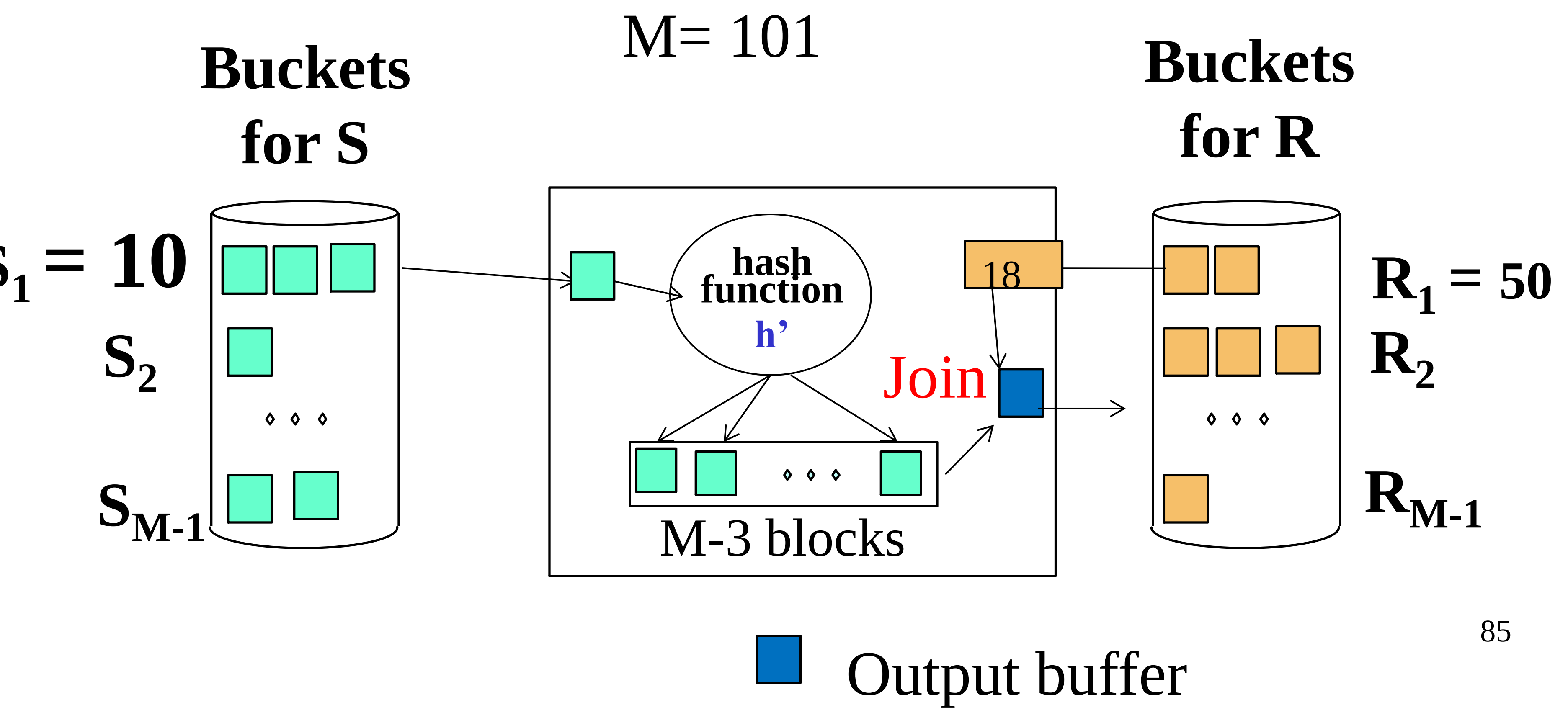
notes

- $h(25) = 1$
- $h(32) = 0$

- $h(a) = a \% 2$
- $h'(a)$
  - $(2+5)\%2 = 1$
  - $(3+2)\%2 = 1$

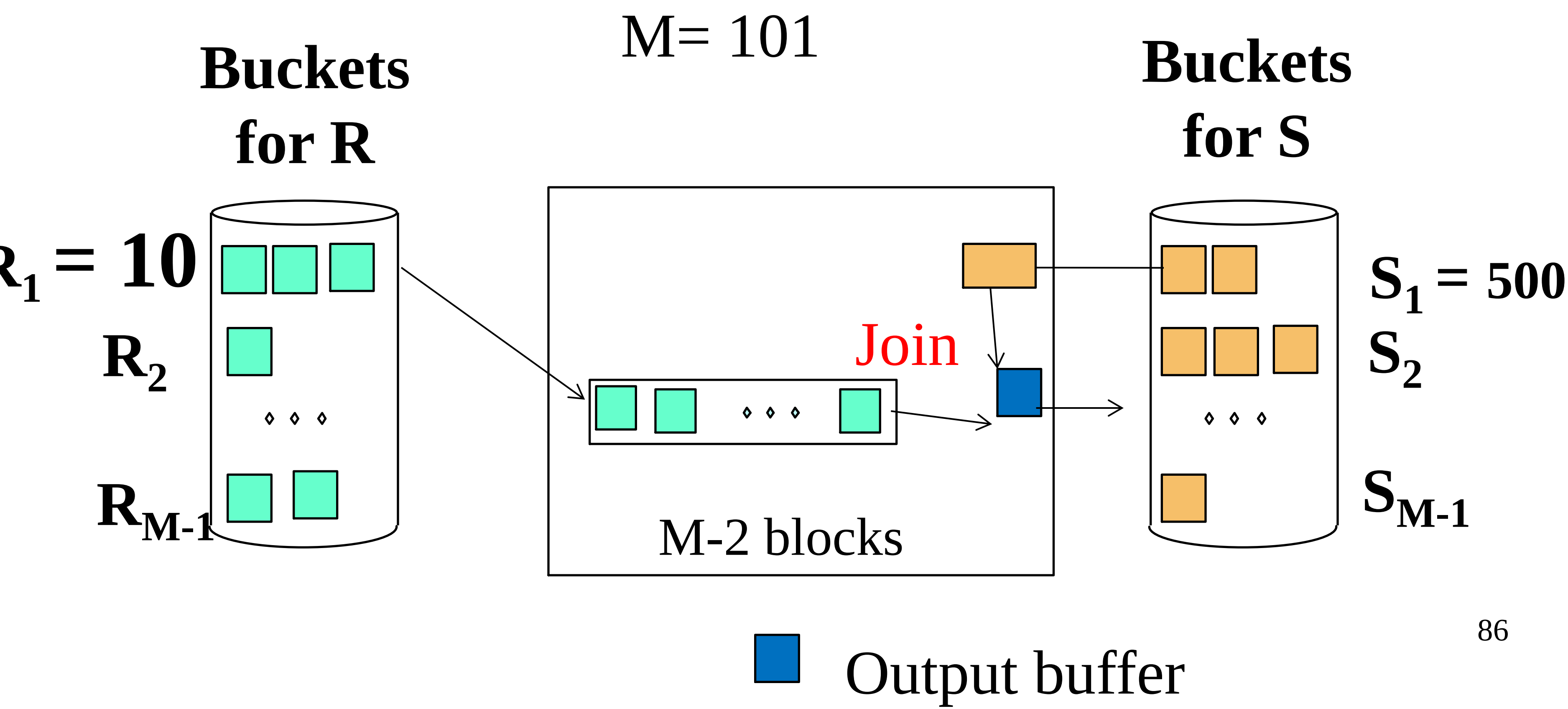
Partitioned Hash-Join: Second Pass

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



Partitioned Hash-Join: Second Pass

- Read in a partition of S, say S_i, hash it using another hash function h'
- Load the 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)) <= M^2
  - Or to be more exact: min(B(R), B(S))/(M-1) <= M-3
  - Or min(B(R), B(S))/(M-1) <= M-2 (if we do not use hash table to speed up the lookup)

Hashing notes

- Suppose M = 101 blocks (i.e., pages), B(R) = 1,000 blocks, B(S) = 5000 blocks (R.a = S.a)

- hash R into 100 buckets, 10 blocks/bucket: 2B(R)
  - R1, R2, …, R100
- hash S into 100 buckets, 50 blocks/bucket: 2B(S)
  - S1, S2, …, S100

- Join R1 with S1, R2 with S2, …, R100 with S100
  - cost: B(R) + B(S)

- total cost: 3B(R) + 3B(S)

Notes

- M = 101, B(R) = 1,000, B(S) = 5000 blocks
- partition R into 100 buckets, 10 blocks/bucket, R1, …, R100
  - cost = 2 * B(R)
- partition S into 100 buckets, 50 blocks/bucket, S1, …, S100
  - cost = 2 * B(S)
- join R1 with S1, R2 with S2, …, R100 with S100
  - R1: 10 blocks, S1: 50 blocks
  - cost: B(R) + B(S)

what if

- M = 101, B(R) = 1,000, B(S) = 50, 000 blocks

- partition
  - R => R1, …, R100, 10 blocks/bucket, cost: 2B(R)
  - S => S1, …, S100, 500 blocks/bucket, cost: 2B(S)

R1: B(R)/(M-1)

S1: B(S)/(M-1)

- join
  - how to join R1 with S1
    - load R1 entirely into memory (10 blocks used)
    - load S1 90 blocks at a time, S11, S12, …, S16
  - cost: B(R) + B(S)

min(R1, S1)<=(M-2)

Notes

- R1: B(R)/(M-1)
- S1: B(S)/(M-1)

- min(R1, S1)<=(M-2)

- min (B(R), B(S)) <= (M-1) (M-2) ~ M^2
  - 10,000

- compared to
  - B(R) + B(S) <= M^2

what if

- M = 101, B(R) = 20,000, B(S) = 50, 000 blocks
- partition
  - R => R1, …, R100, 200 blocks/bucket, cost: 2B(R)
  - S => S1, …, S100, 500 blocks/bucket, cost: 2B(S)
- join
  - how to join R1 (200) with S1 (500)
    - partition R1 => R11,…, R1,100 (2 blocks/bucket), cost: 2B(R)
    - partition S1 => S11,…, S1,100 (5 blocks/bucket), cost: 2B(S)
    - join R11 with S11, …, cost: B(R) + B(S)
  - cost: 5B(R) + 5B(S)

Notes

- M = 101, B(R) = 1000, B(S) = 5000, R.a = S.a

- partition R into 100 buckets, 10 blocks/bucket
  - R1, …, R100
- partition S into 100 buckets, 50 blocks/bucket
  - S1, …, S100

- join R1 with S1, …, R100 with S100
  - R1: 10 blocks, S1: 50 blocks

Notes

- M = 101, B(R) = 1000, B(S) = 50,000, R.a = S.a

- partition R into 100 buckets, 10 blocks/bucket
  - R1, …, R100
- partition S into 100 buckets, 500 blocks/bucket
  - S1, …, S100

- join R1 with S1, …, R100 with S100
  - R1: 10 blocks, S1: 500 blocks
  - load R1 entirely into memory, load S1 90 blocks…

summary

- (R = 1000)

- sort-merge join

  => 10 runs, 100 blocks/run

- partitioned-hash join

  => 100 buckets, 10 blocks/bucket

Notes

- M = 101, B(R) = 20,000, B(S) = 50,000, R.a = S.a
- partition R (h) => 100 buckets, 200 blocks/bucket
  - R1, …, R100
- partition S (h) => 100 buckets, 500 blocks/bucket
  - S1, …, S100
- join R1 with S1, …, R100 with S100
  - R1: 200 blocks, S1: 500 blocks
  - R1 (h') => R11, R12, R13, …, R1,100
  - S1 (h') => S11, ….

Hashing notes

- Suppose M = 101 blocks (i.e., pages), B(R) = 20,000 blocks, B(S) = 50,000 blocks (R.a = S.a)
- Step 1 (hash function = h)
  - hash R using h into 100 buckets, 200 blocks/bucket: 2B(R)
    - R1, R2, …, R100 (cost: 2B(R))
  - hash S using h into 100 buckets, 500 blocks/buck: 2B(S)
    - S1, S2, …, S100 (cost: 2B(S))
- Step 2 (hash function = h') question: h' =? h
  - Join R1 (200 blocks) with S1 (500 blocks)
    - step 1: partitioning (R1, R2, …, R100) cost: 2B(R)
    - step 2: partitioning (…

Hashing notes

- Suppose M = 101 blocks (i.e., pages), B(R) = 20,000 blocks, B(S) = 50,000 blocks (R.a = S.a)

- hash R into 100 buckets, 200 blocks/bucket
  – R1, R2, …, R100
- hash S into 100 buckets, 500 blocks/buck: 2B(S)
  – S1, S2, …, S100

* Join R1 (200 blocks) with S1 (500 blocks)
  – hash R1 into R11, R12, …, R1-100
  – hash S1 into …

Notes

- Suppose M = 101 blocks (i.e., pages), B(R) = 1,000 blocks, B(S) = 50, 000 blocks (R.a = S.a)

Example

- Suppose M = 101 blocks (i.e., pages), B(R) = 1,000 blocks, B(S) = 50,000 blocks (R.a = S.a)

- Cost of R $\bowtie$ S using partitioned hash join algorithm
  – Pass 1: hash R into 100 buckets, 10 blocks/bucket (R1)
  hash S into 100 buckets, 500 blocks/bucket (S1)
  cost: 2B(R) + 2B(S)
  - Pass 2: join Ri with Si
  cost: B(R) + B(S)
- What if B(S) = 50,000 blocks?

Example

- Suppose M = 101 blocks (i.e., pages), B(R) = 10,000 blocks, B(S) = 50,000 blocks (R.a = S.a)
- Cost of R $\bowtie$ S using partitioned hash join algorithm
  - Pass 1: hash R into 100 buckets, 100 blocks/bucket (R1)

    hash S into 100 buckets, 500 blocks/bucket (S1)

    cost: 2B(R) + 2B(S)

    -extra: hash (R1) => 100 buckets, 1 block/bucket (R11)

    hash(S1) => 100 buckets, 5 blocks/bucket (S11)

    join R11 with S11, R12 with S12, … R1,100 with S1,100
  - Pass 2: join Ri with Si

    cost: B(R) + B(S)
- What if B(S) = 50,000 blocks?

notes

- size of Ri = B(R)/(M-1)
- size of Si = B(S)/(M-1)

- min[B(R)/(M-1), B(S)/M-1)] <= M – 2
- min[B(R), B(S)] <= (M-1) (M-2) ~ M^2
  - min(1000, 50000) <= 10,000

- recall sorting formula
  - B(R) + B(S) <= M^2
    - 1000 + 50,000 <= 10,000

Example

- Suppose M = 101 blocks (i.e., pages), B(R) = 20,000 blocks, B(S) = 50,000 blocks
- Cost of $R \bowtie S$ using partitioned hash join algorithm
  – Pass 1: hash R into 100 buckets, 200 blocks/bucket (Ri)

  hash S into 100 buckets, 500 blocks/bucket (Si)

  cost: 2B(R) + 2B(S)

  -- pass 2: join R1 (200 blocks) with S1 (500 blocks)

  join R2 with S2, …

  – Pass 3: ….

  cost: …
- 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

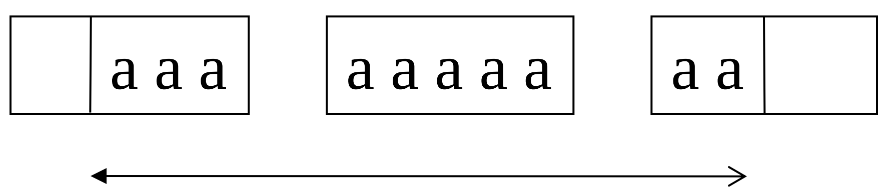
Index-Based Algorithms

Index-based Algorithms

- The existence of an index on one ore 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



All the ‘a’ tuples

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

Compare this

- 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

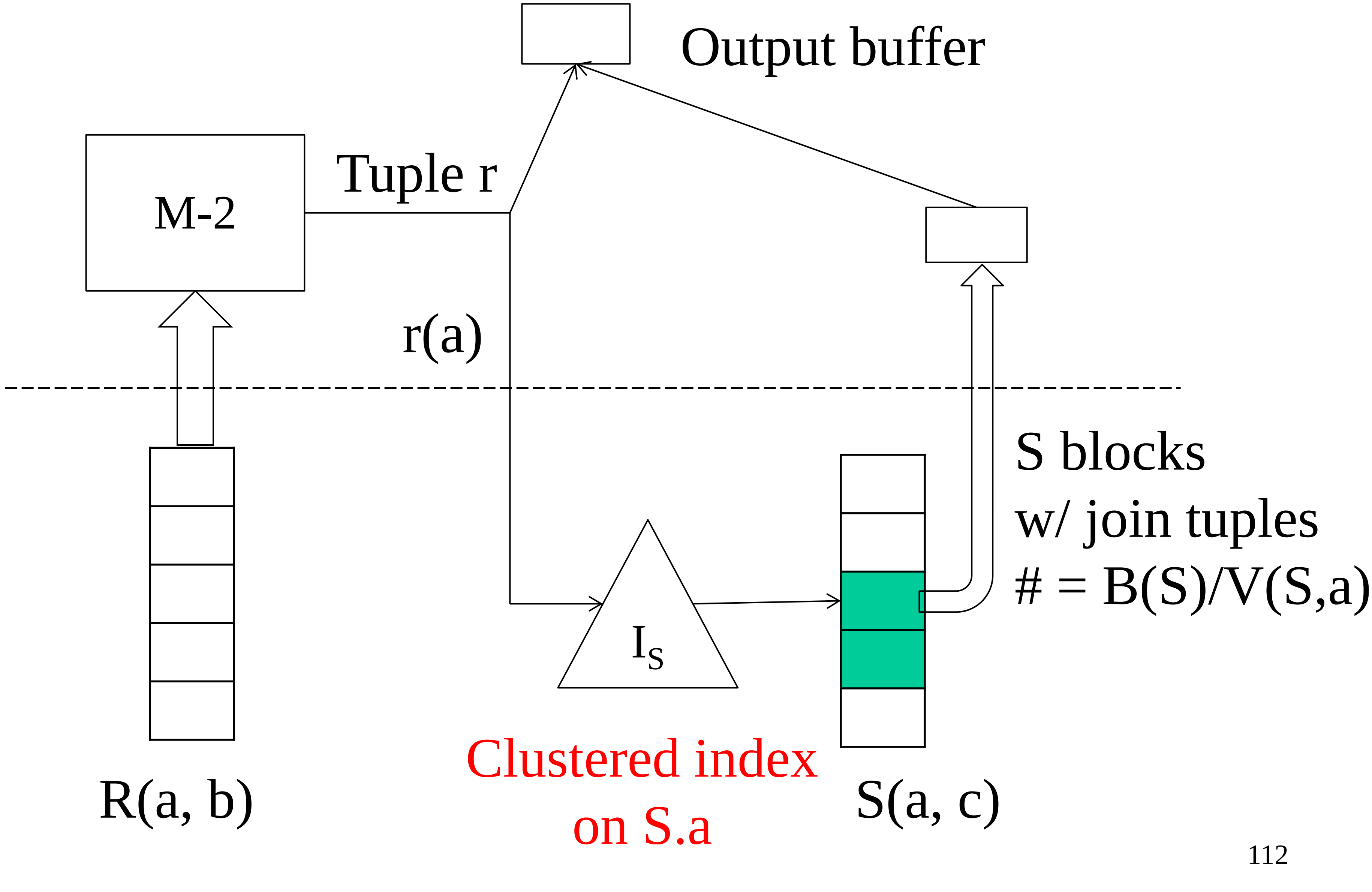
- $R \bowtie S$
- Assume S has an index on the join attribute
- Iterate over R, for each tuple, fetch corresponding tuple(s) from S
- Assume R is clustered. Cost:
  - If index is clustered: B(R) + T(R)B(S)/V(S,a)
  - If index is unclustered: B(R) + T(R)T(S)/V(S,a)
- Compare this to NLJ (both R & S clustered)
  - B(R) + B(R)/(M-2) * B(S)

Indexed-Based Join vs. NLJ

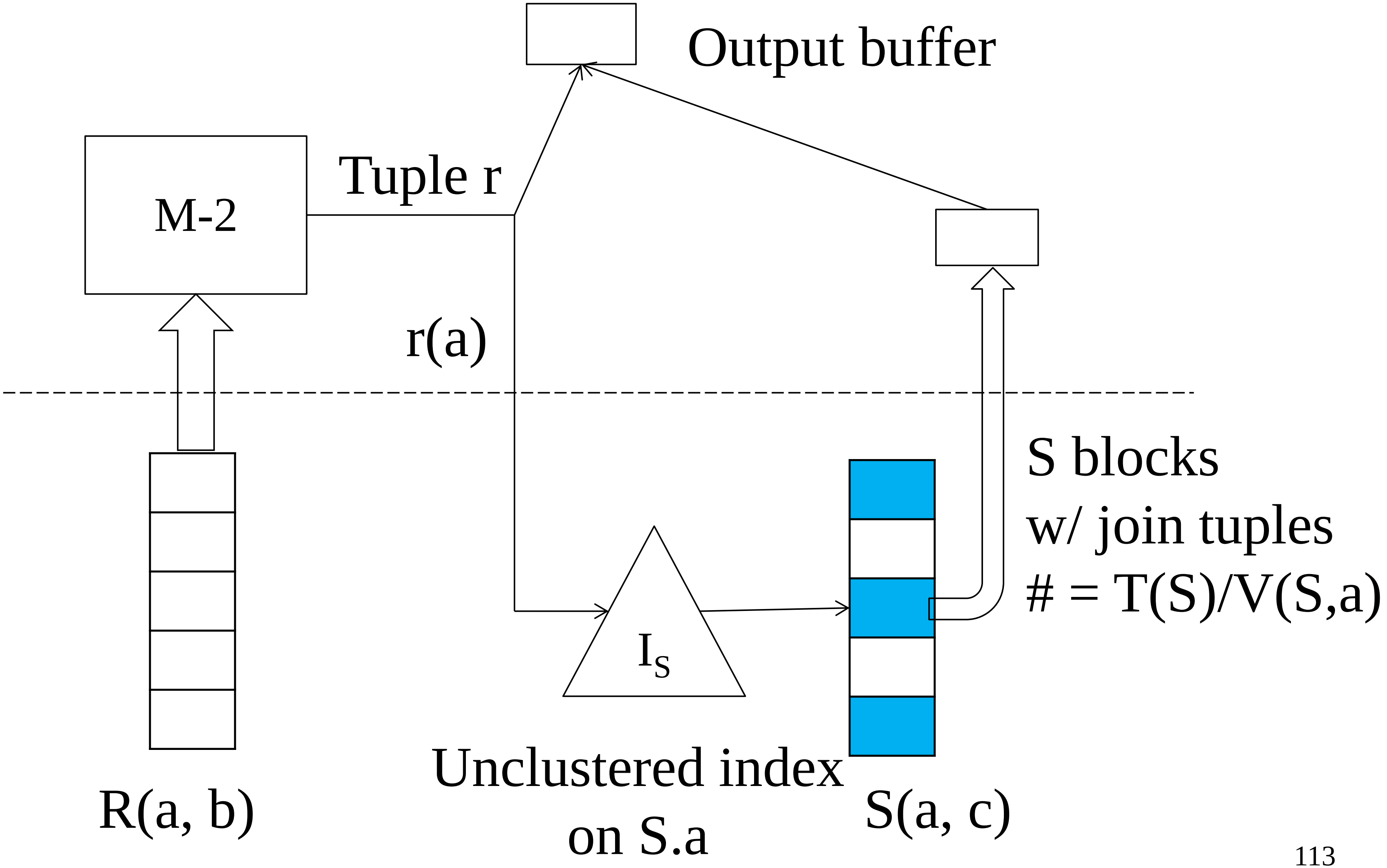
- Index-based (R clustered, clustered index S.a)
  - B(R) + T(R)B(S)/V(S,a)
- NLJ (R & S clustered)
  - B(R) + B(R)/(M-2) * B(S)

- Index-Based wins if:
  - T(R)/V(S,a) < B(R)/(M-2), or
  - V(S,a) > (M-2) * T(R)/B(R)

Index-Based Join: Clustered Index



Index-Based Join: Unclustered Index



Example

- Suppose M = 102 blocks (i.e., pages)
- $R(a, b) \bowtie S(a, c)$
- 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)