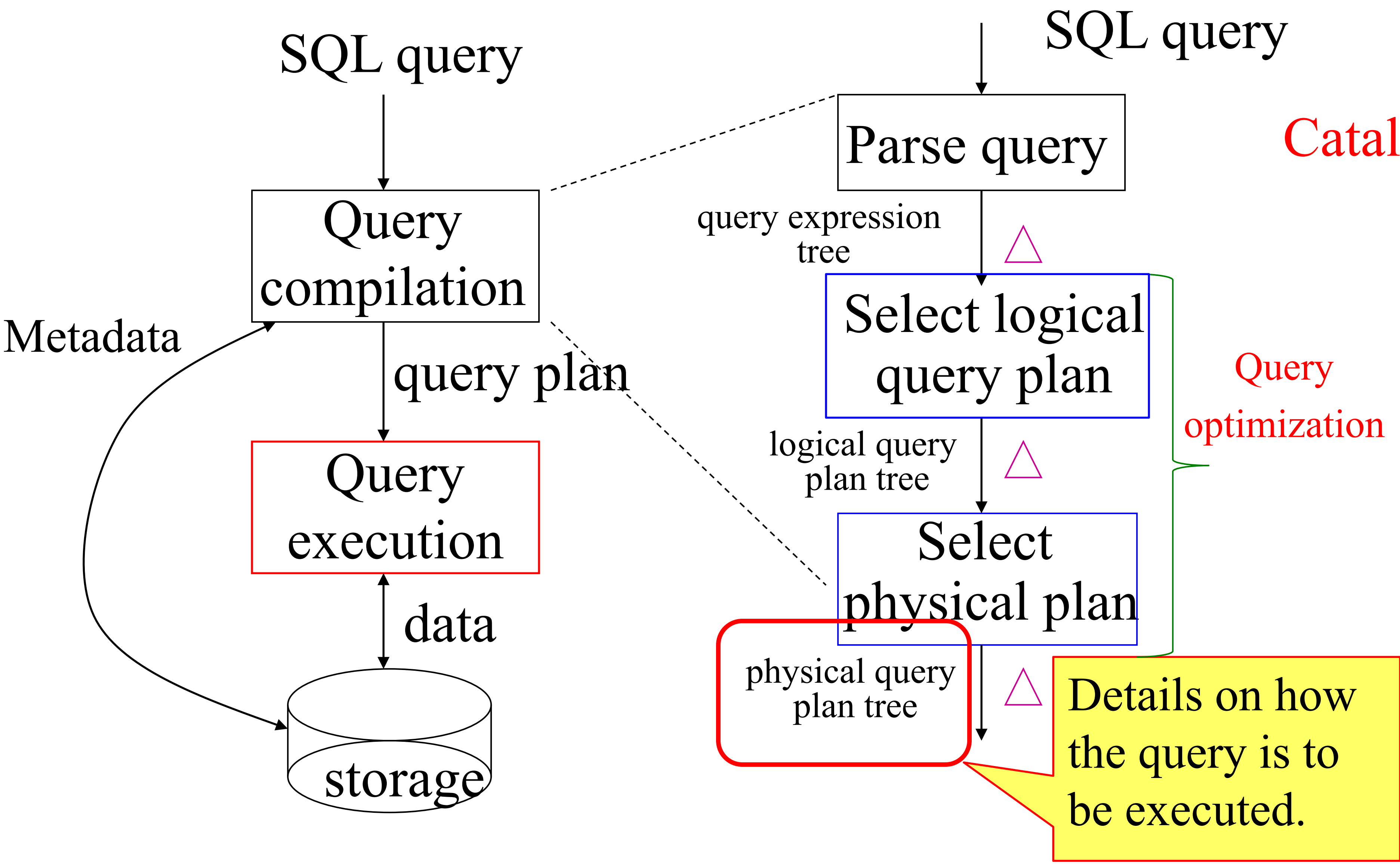
Query Execution

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 NLJ
- 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
```

Construct logical plan…

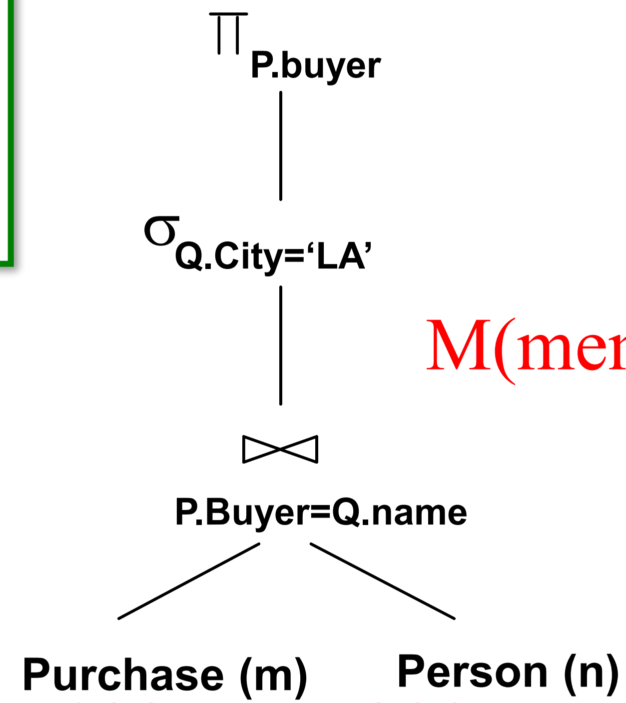
```
NLJ (Purchase outer) :
        for p in Purchase:
            for q in Person:
                if (p.buyer = q.name)
NLJ (Person is outer):…
```

Logical Query Plans

100MB (purchase)

2GB= Part1:
P2: 90
P3: 20

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



M(memory) = 2

n, john

Query Plan:

- Tree with logical operators
- h(buyer)
- h(name)
- h(John) = 0/1

	Purchase (m)	Person (n)
Scenario A:	100MB	200MB
B:	100MB	2GB = P1 (1GB),
C:	2GB	2GB
	R1: 1GB	P1: 1GB

Notes

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

Example (cont'd)

M = 1GB

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

R1 join P1
R1 join P2
R1 join P3
R1 join P4
R2 join P1
…

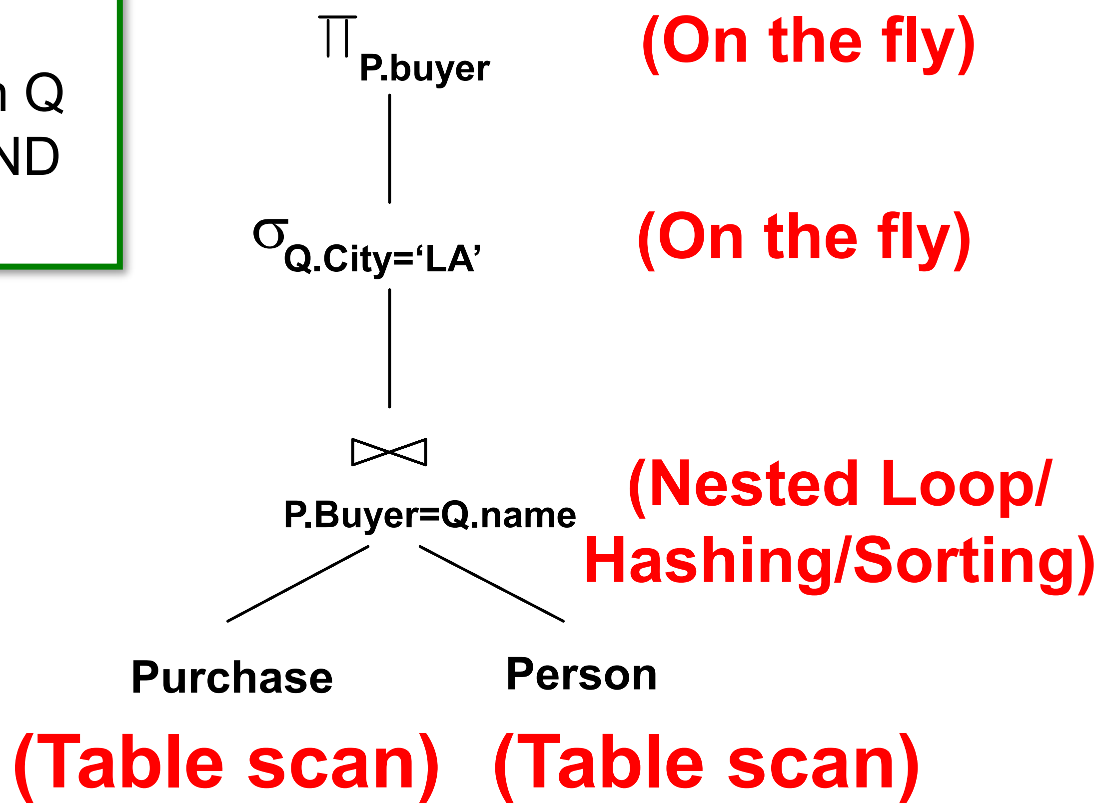
Block-based NLJ algorithm

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

```
class C
  def o
  def n
  def c

class F
   def
   def

class F
    def

class J
```

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

V(Employee, gender) = 2 (assuming binary gender) This means the 'gender' attribute is not very selective as it has only 2 distinct values. V(Employee, ssn) = n, where n is the number of employees. This means the 'ssn' attribute is highly selective as (presumably) every employee has a distinct ssn value.

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

sequentially scanning

好像和indexing里面的聚簇索引和非聚簇索引不太一样

- 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

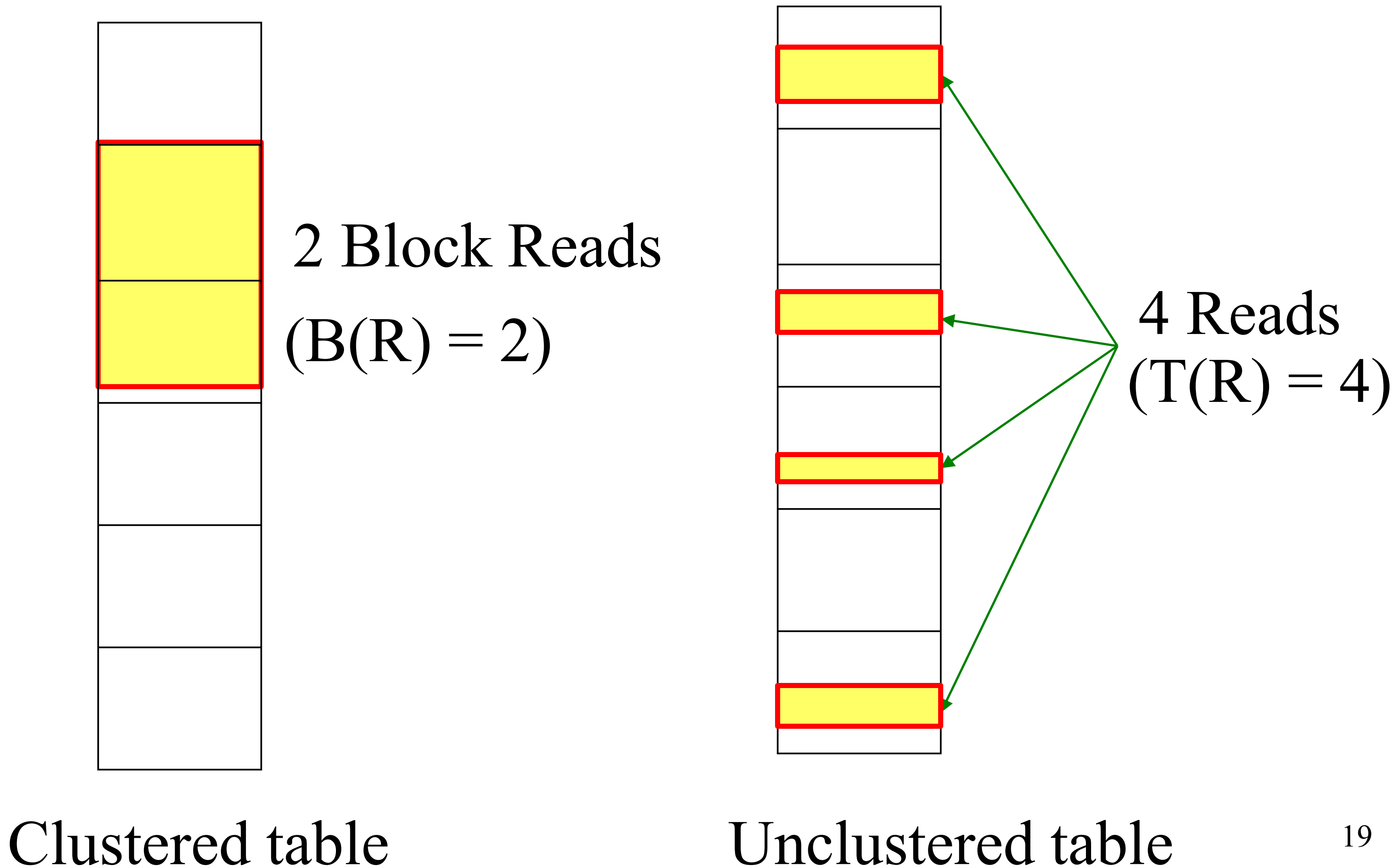
Clustered:

A table is clustered if its records are stored together contiguously on disk blocks.
In this case, each disk block contains records only from this table.
To scan the entire table, the database needs to read as many disk blocks as required to hold the table.
So the I/O cost (number of disk block reads) is simply equal to the number of blocks B(R) that the table occupies on disk.

Unclustered Table:

In this case, the table records are not stored contiguously.
The records are scattered across disk blocks, potentially interleaved with records from other tables.
In the worst case, to read all records of this table, the database may need to read one disk block for every record in the table.
So the I/O cost could be as high as T(R), the number of tuples/records in the table, if the records are completely scattered.

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)

  is an example of one-pass algorithm
  - Read one relation only once, while the other will be read repeatedly from disk
- Two-pass algorithms

  If the operation cannot be completed in a single pass over the data, a two-pass algorithm may be used.
  - First pass: read data from disk, process it, write it to the disk
  - Second pass: read the data for further processing

  Examples: Sort-merge join, hash join

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

i.e. more than 2 passes

Each pass reads the prior pass's output from disk, processes it, and writes it out again.
The high number of passes increases the I/O cost significantly.

总结:

The goal is to complete operations in as few passes as possible, ideally in a single pass if feasible, to minimize the costly disk I/O operations. Two-pass algorithms strike a balance when one-pass is not possible. Multi-pass algorithms are a last resort for handling very large data.

1. The basic nested-loop join algorithm is called "Naive Nested-Loop Join". For each tuple in the outer relation R, it scans the entire inner relation S to find matching tuples.
2. “1.x” indicates there are other nested-loop join variations beyond the basic naive version, such as:

1.1) Block Nested-Loop Join - Where the inner relation is read in blocks of rows instead of one row at a time.

1.2) Batched Nested-Loop Join - Where a batch/set of tuples from the outer relation is joined with the inner relation in each iteration.

1.3) Index Nested-Loop Join - Where an index on the inner relation is used to optimize the nested loop join.

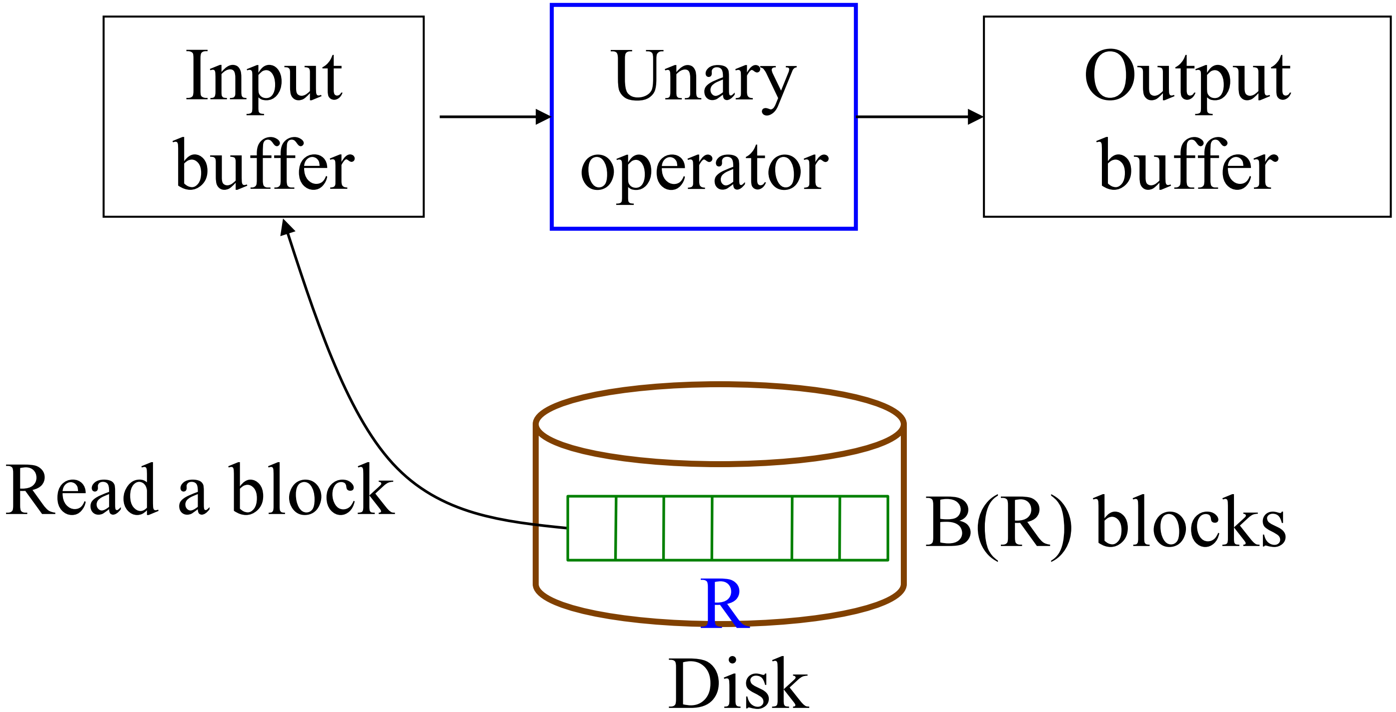
So in summary, "1.x" is used to generically refer to the nested-loop join algorithm and its multiple variations/improvements over the basic naive nested-loop implementation. The nested-loop family has optimized algorithms tailored for different relation sizes, ordering, available indexes etc. But they all follow the basic nested-loop approach of reading one relation only once while repeatedly scanning the other relation.

One-pass algorithms

One-pass Algorithms

Selection σ(R), projection Π(R)

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



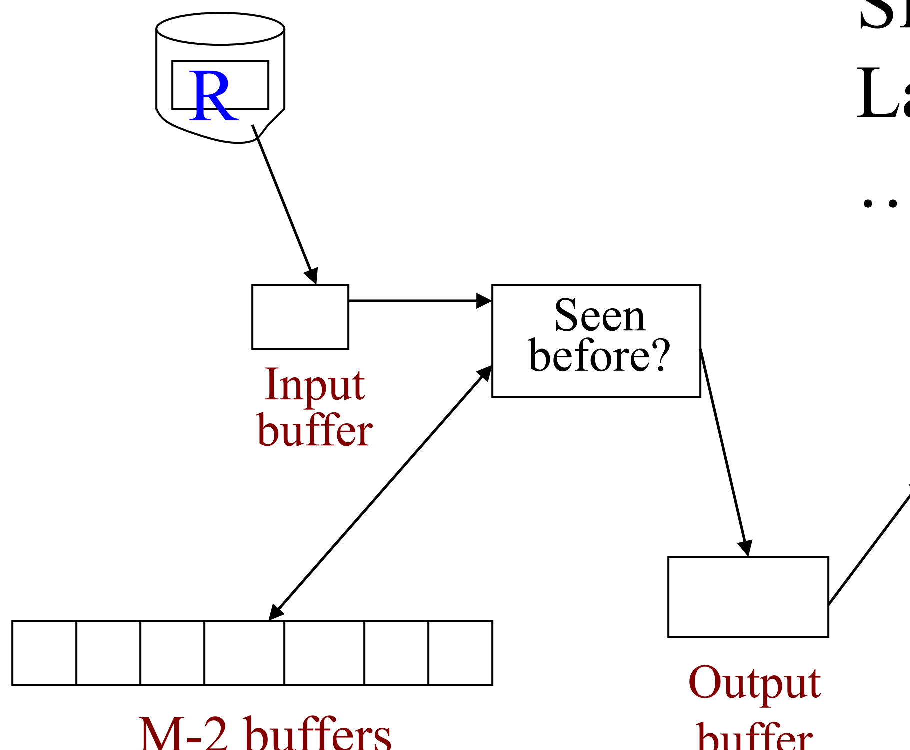
One-pass Algorithms

Duplicate elimination δ(R)

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

  B(δ(R)) <= M-2

  or roughly M



La, 2
La, 3
Sfo,
Sfo,
La, 5
…

Duplicate elimination

requires maintaining an in-memory data structure (dictionary) to keep track of tuples already seen while scanning R.

This dictionary can be implemented using different data structures like:

- A balanced search tree (e.g. Red-Black tree)
- A hash table

The cost of this algorithm is B(R), which is the number of disk blocks occupied by relation R. This assumes scanning R requires reading all its blocks.

There is an assumption that the size of the duplicate-eliminated result d(R) fits in memory.
Specifically:

$B(d(R)) \le M-2$ blocks,
where M is the number of available memory blocks

Or roughly, $B(d(R)) \le M$ blocks

This is because the algorithm needs some memory buffers:

- One for reading input blocks of R
- Some for the in-memory dictionary
- One for writing the output duplicate-free tuples

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

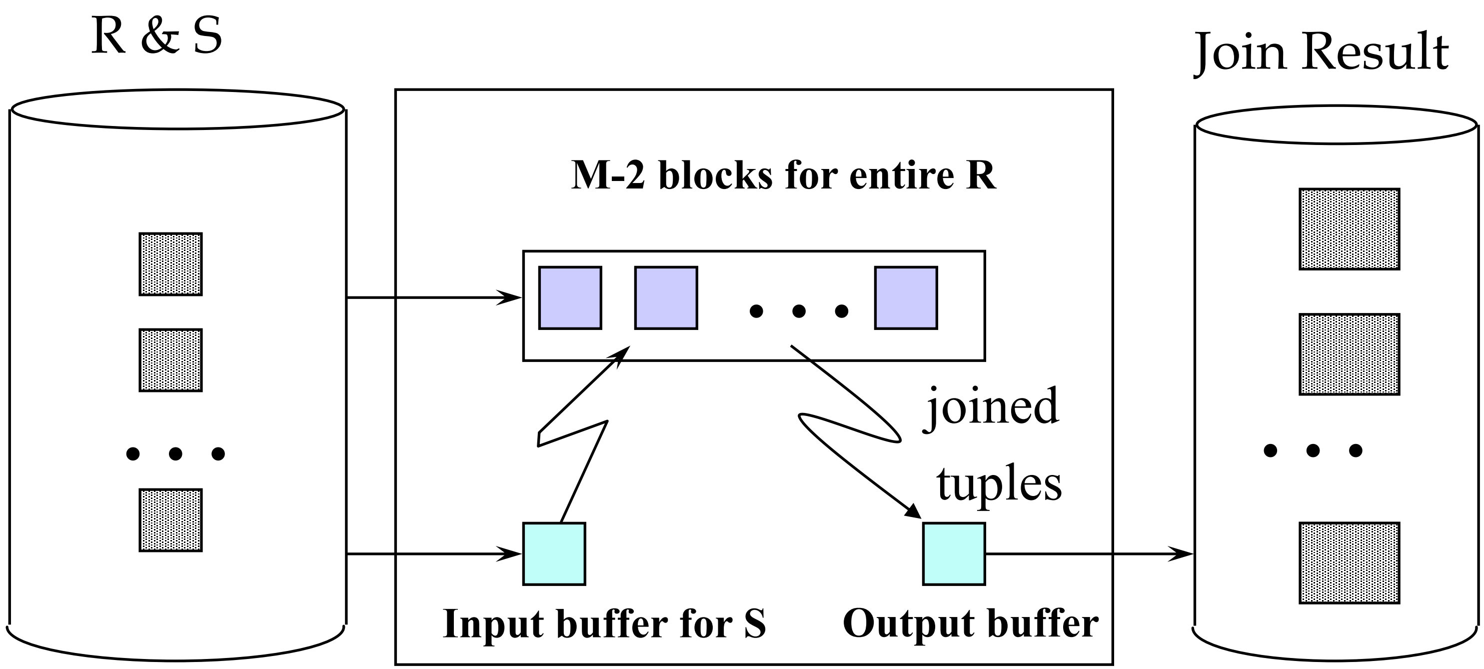
One-pass Algorithms

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

R & S 的示意图



$$M = 102$$
$$B(R) <= 100$$

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)

The key difference between tuple-based NLJ and block-based NLJ is that: the tuple-based scans relations tuple-by-tuple, while the block-based loads a block of the outer relation at once to reduce diskI/O costs if data is clustered.

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)

cost (R is outer): 1+

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

$\Rightarrow$B(R) + B(R)/(M-

ls:

Beers:

er = b.name):

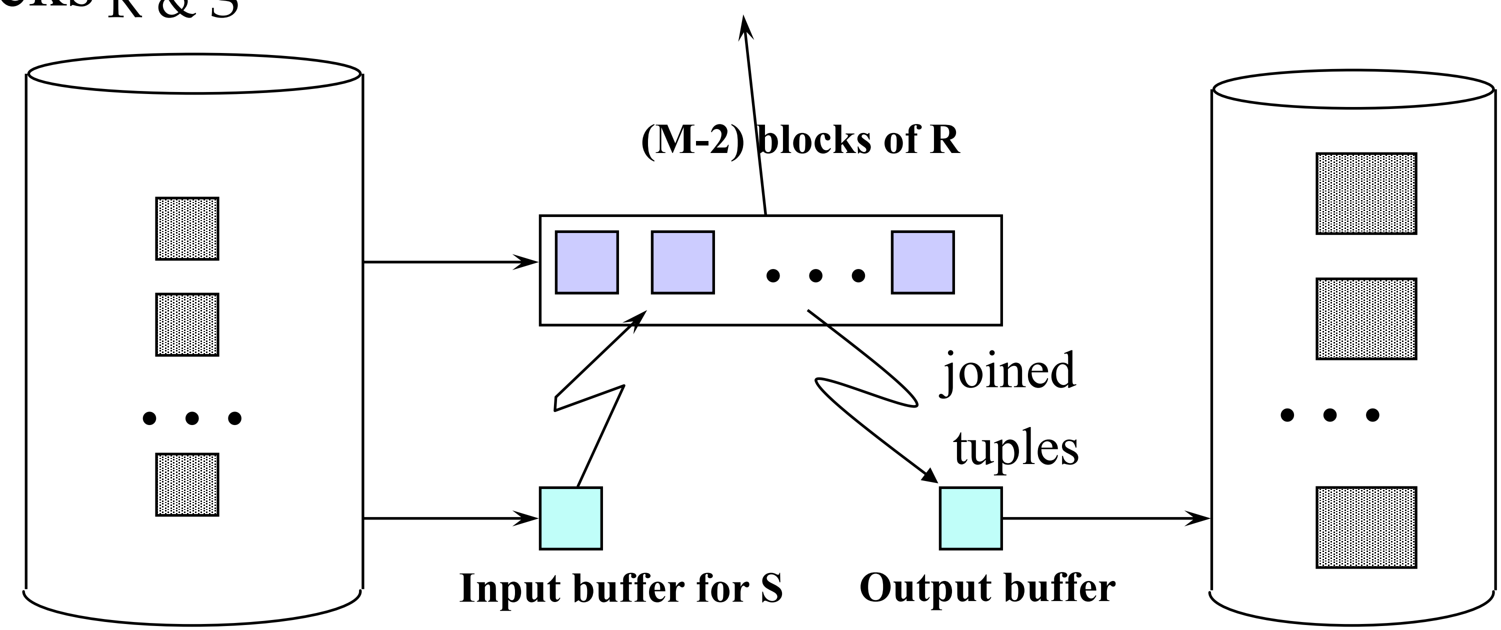
- Assume $B(R) \leq B(S)$ & $B(R) > M$

Block-based Nested Loop Joins

M = 102

load 100 blocks of R

blocks R & S



)0 blocks

2

100个block的内存中的 I/O 次数

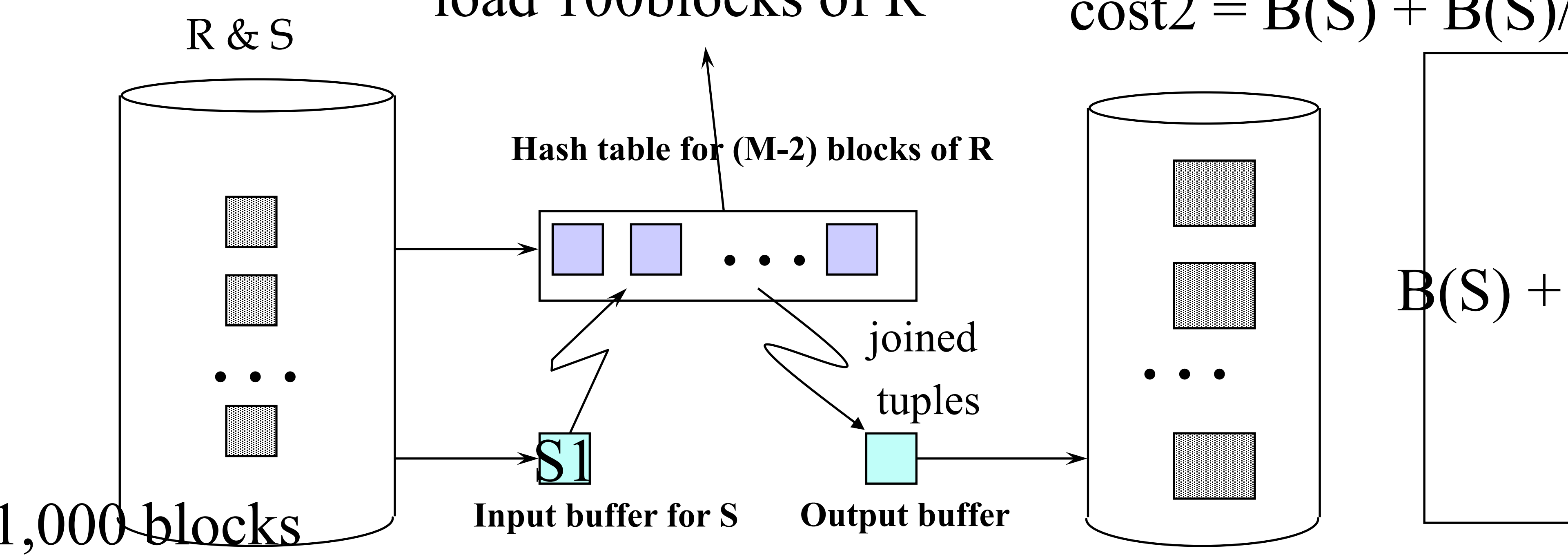
cost(R is outer) = B(R) + B(R)/(M-2) * B(S)

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

Block-based Nested Loop Joins

cost1 = B(R) + B(R)

cost2 = B(S) + B(S)/



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

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

…

- load 10^{th} 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)

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? M

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 blocked-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

所以R应该做outer loop的那个。就是小一点的 做outer loop

sele
for
f
B(R
1.x
B(S
1.x

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) since it requires reading and writing the entire relation
- Pass 2: merge M-1 runs, but include each tuple only once
  - Cost: B(R) since we need to read all the runs

- 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 $B/M \leq M - 1$

There is an assumption that $B(R) \leq M^2$ (or more precisely $B(R)/M \leq M-1$)

* This is because in Pass 2, we need all the sorted runs to fit in memory for merging
* If $B(R)/M > M-1$, then there would be more than M-1 runs, which cannot be merged in a single pass

Grouping: $\gamma_{\text{city, sum(price)}}$ (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: $B(R) \leq M^2$

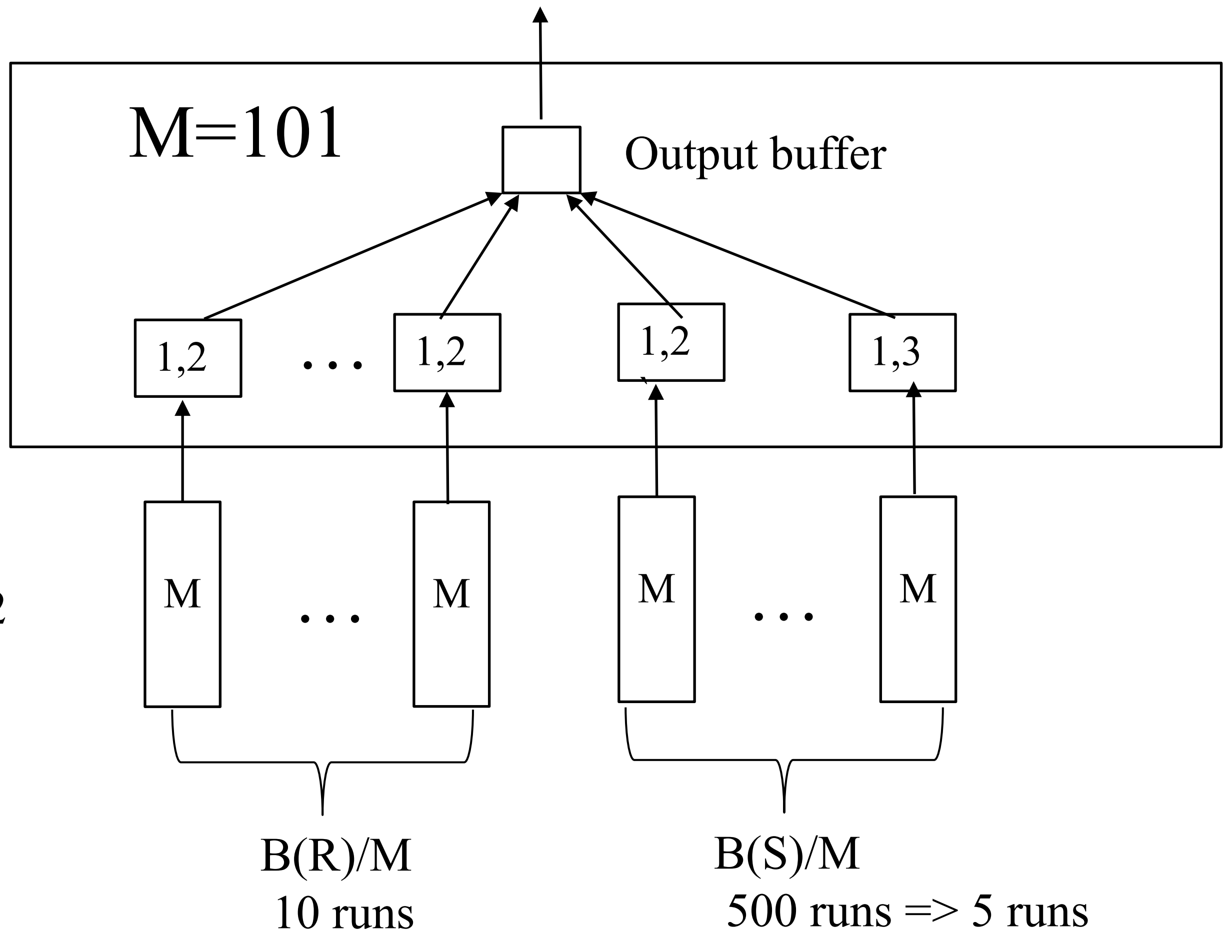
Binary operations: $R \cap S$, $R \cup S$, $R - S$ Based on Sorting

- 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: $B(R)+B(S) \leq M^2$

Merging picture

S on R.a=S.a

1
1
2
2



1
1
1
3
…

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

<= 100^2 = 10,000

R join S = ?

notes (simple-sort)

具体原理见p50

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)

The steps are:

1. Completely sort relation R:

- R has 1000 blocks
- Initially, it gets split into 10 sorted runs of size M each (assuming M=100 memory buffers)
- Then these 10 runs are further merged into a single fully sorted run of R

Cost of this step is 4B(R) = 4 * 1000 = 4000 I/Os

2. Completely sort relation S:

- S has 50,000 blocks
- Initially, it gets split into 500 sorted runs of size M each
- Then these 500 runs are merged into 5 runs
- Finally, the 5 runs are merged into 1 fully sorted run of S

    Cost of this step is 6B(S) = 6 * 50,000 = 300,000 I/Os

3. Merge the fully sorted R and S:

- Now that both R and S are sorted, they can be merged together

    Cost of this final merge step is B(R) + B(S) = 1000 + 50,000 = 51,000 I/Os

Total cost = Cost of sorting R + Cost of sorting S + Cost of final merge
= 4B(R) + 6B(S) + B(R) + B(S)
= 5B(R) + 7B(S)
= 51000 + 750,000
= 355,000 I/Os

So the complete sort-based algorithm has a very high cost of 355,000 I/O operations for the given relations sizes. This illustrates why simpler sort-based algorithms are not preferred for larger relations.

前提：M = 100 - 1

Notes (sort-merge)

具体原理见p48

- 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)

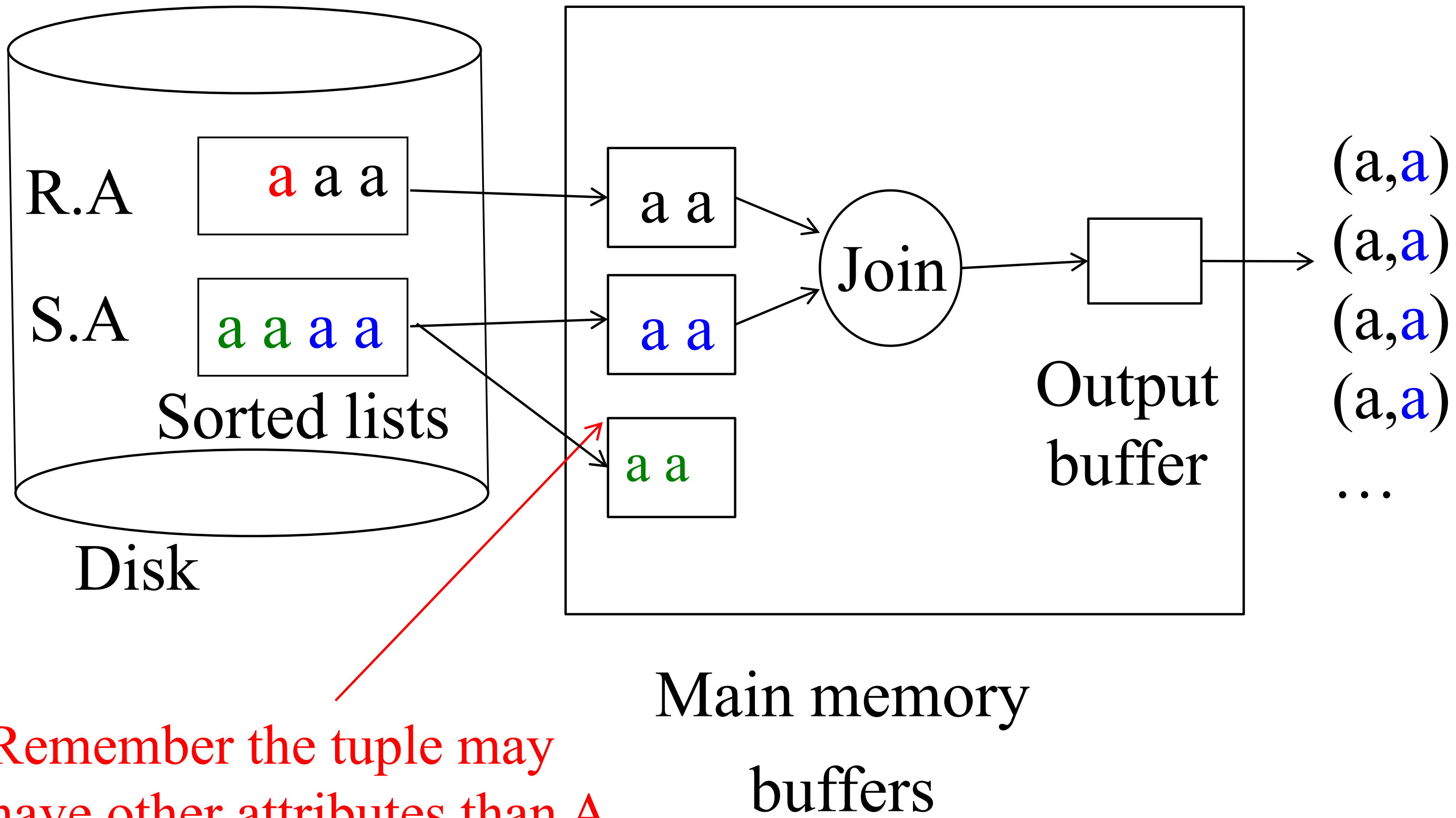
Problem with join

- 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



Remember the tuple may have other attributes than A

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: $B(R) + B(S) \leq M^2$

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: 2B(R)+2B(S)+B(R)+B(S)=3B(R)+3B(S)

如果不满足上面条件，就是merge的时候不能一个pass搞定，那就跟外部排序merge一样接着merge M-1 runs，每次merge增加2B(S or R)。
e.g. 见p45

前提：M = 100 - 1

Notes (sort-merge)

具体原理见p48

- 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)

Example

同 p 45

- 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)

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 in sorting

- Cost of R $\bowtie$ S using simple sort-based join algorithm
  - Sort R (completely): 4B(R) = 4000
  - Sort S: 4B(S) = 20,000
  - Join by merging R' with S': B(R) + B(S)

- 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)

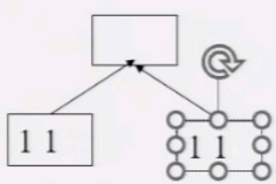
Notes

- 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: 6B(S) = 3 * 2B(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

Sorting R (completely):
B(R) + B(R) // 10 runs
B(R) + B(R) // 1 run
= 4B(R)

Sorting S:
= 4B(S)

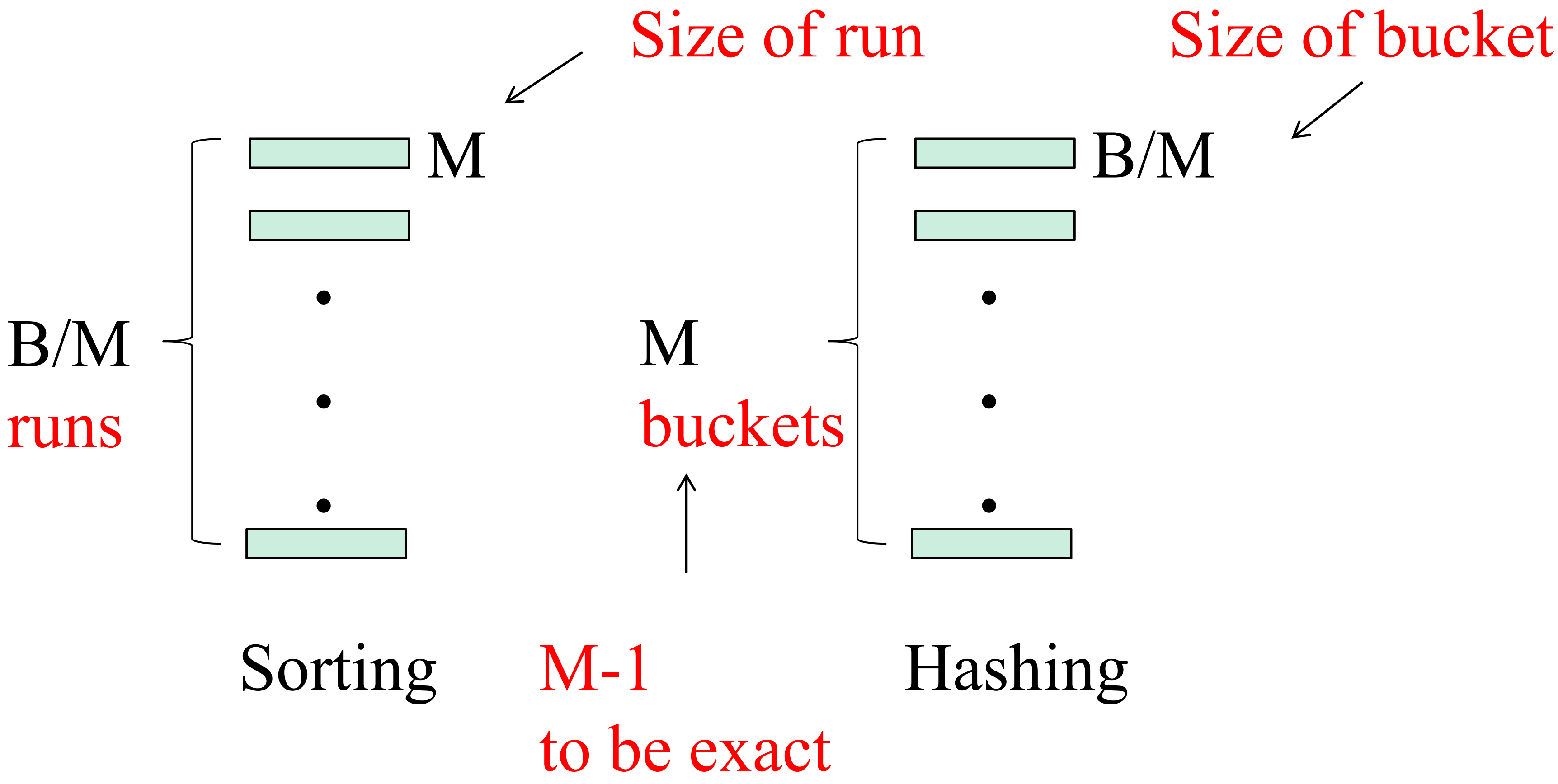
Merging R and S:
B(R) + B(S)

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

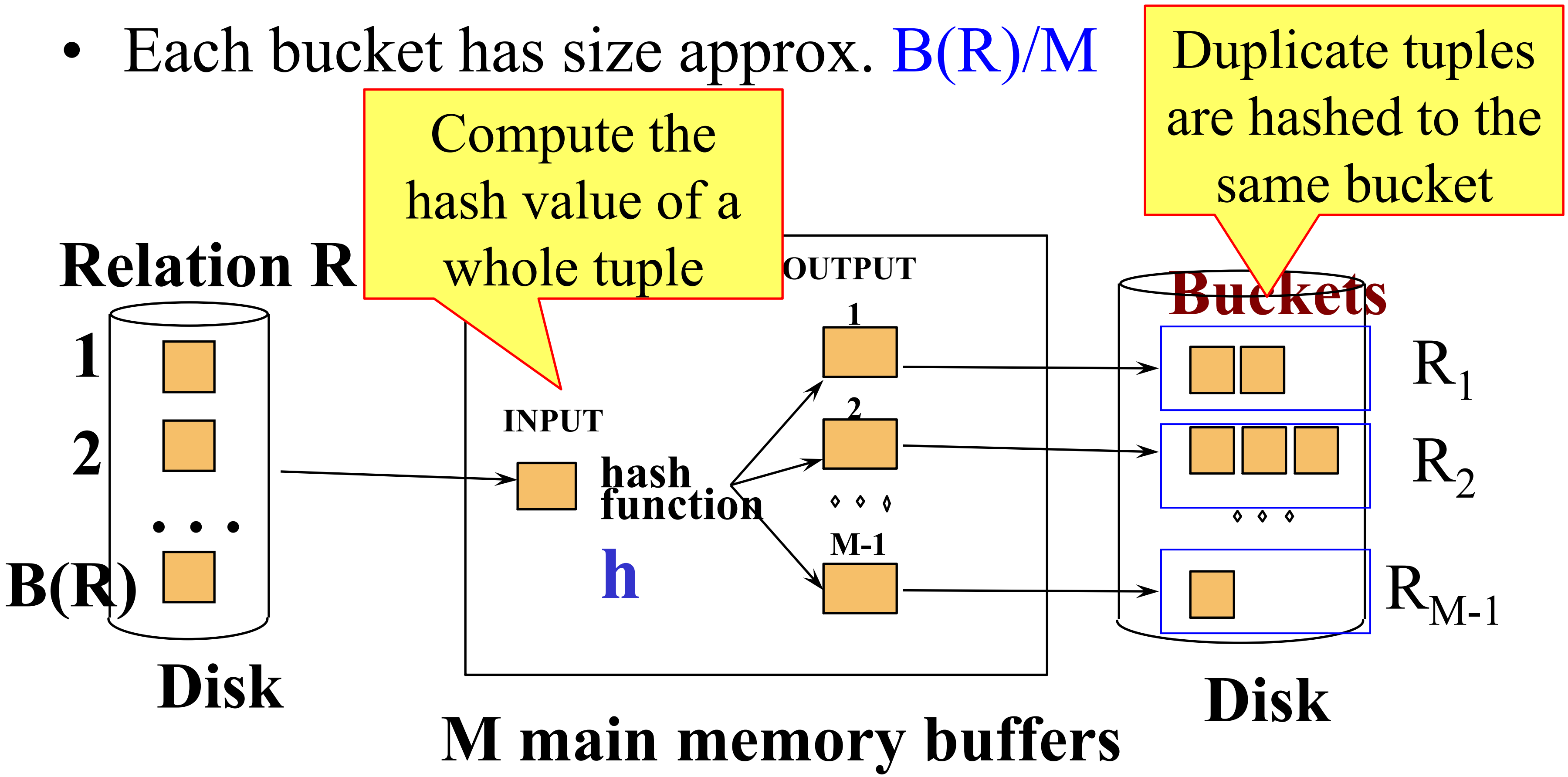
Hashing-Based Algorithm for δ

- 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: $B(R) <= M^2$
  - To be more exact: $B(R)/(M-1) <= M-2$

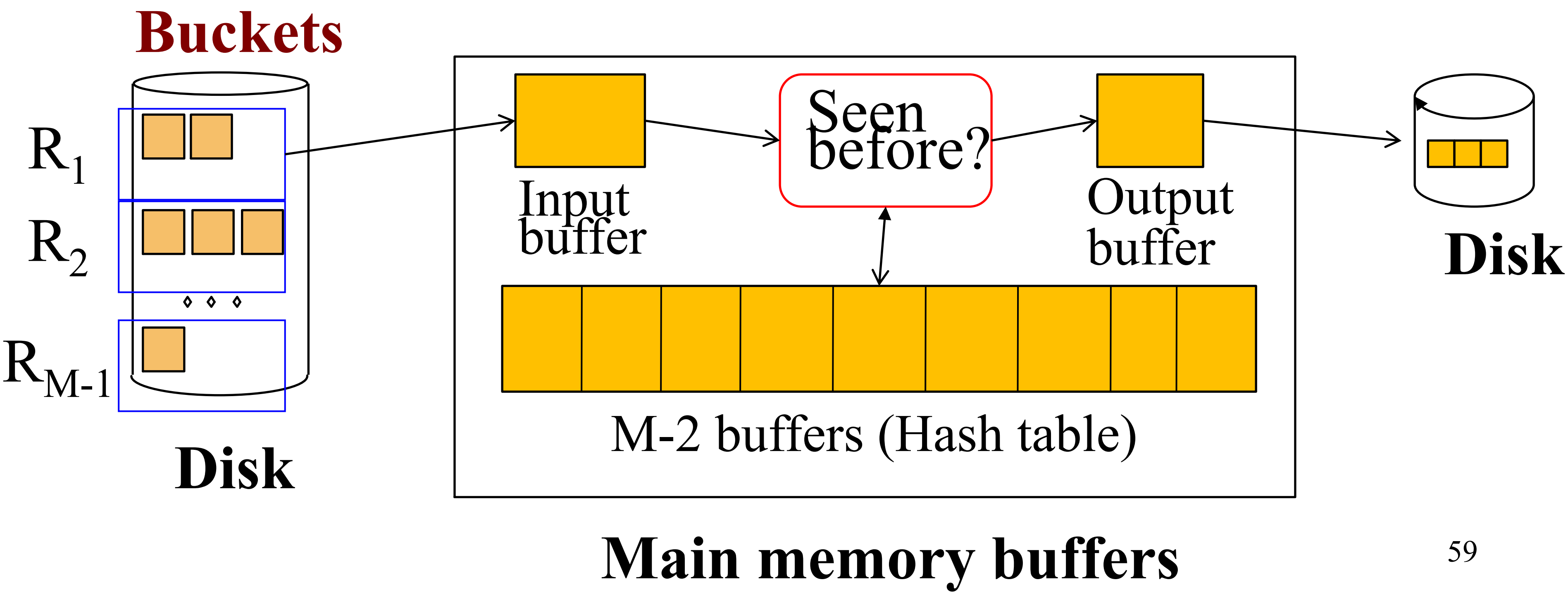
Two-Pass Duplicate Elimination Based on Hashing

- 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



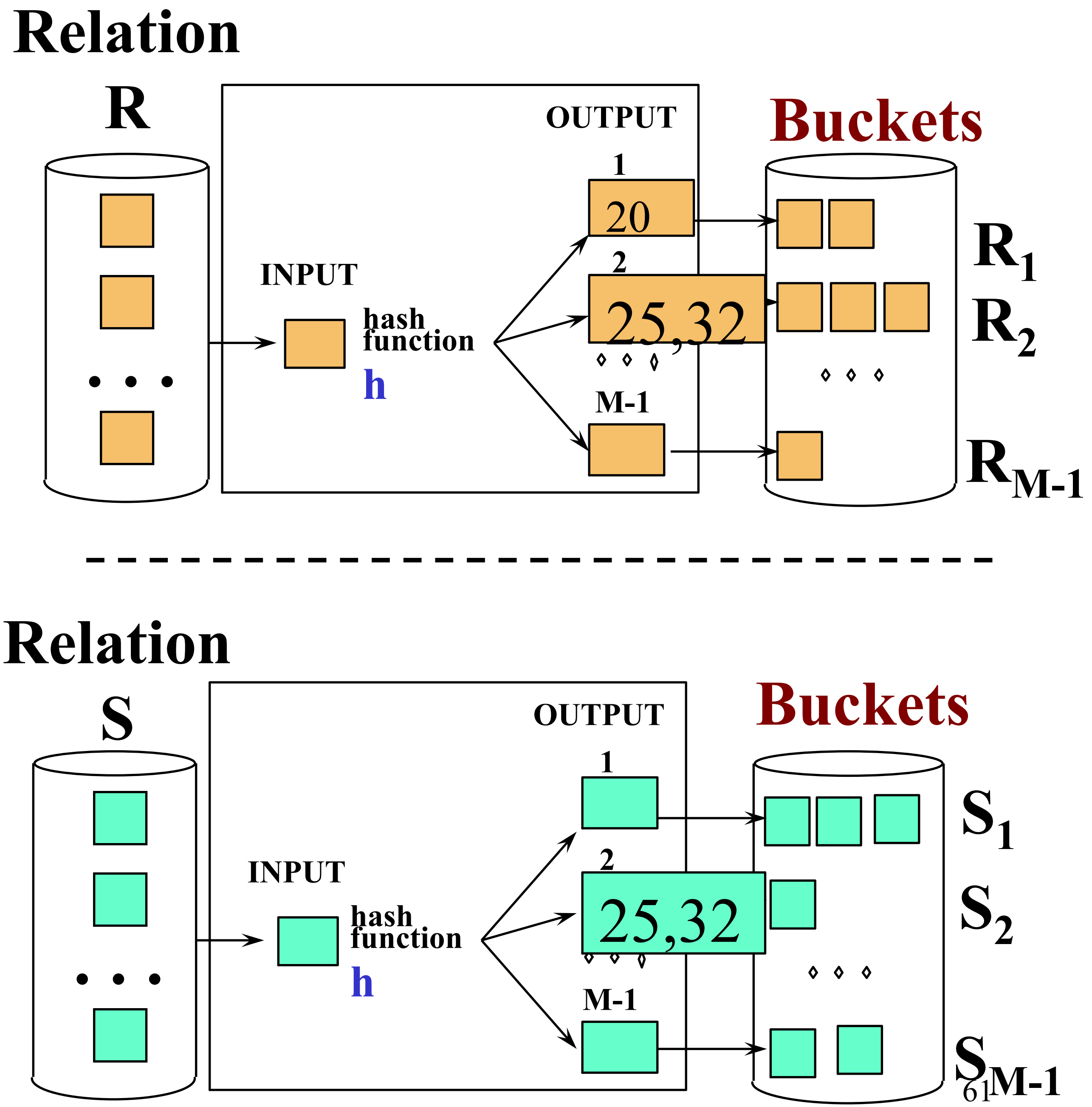
Partitioned Hash Join

$R \bowtie S$

- Step 1:
  - 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.
- R.age = S.age
- h(r.age) = h(25) = 2
- h(s.age) = h(25) = ?



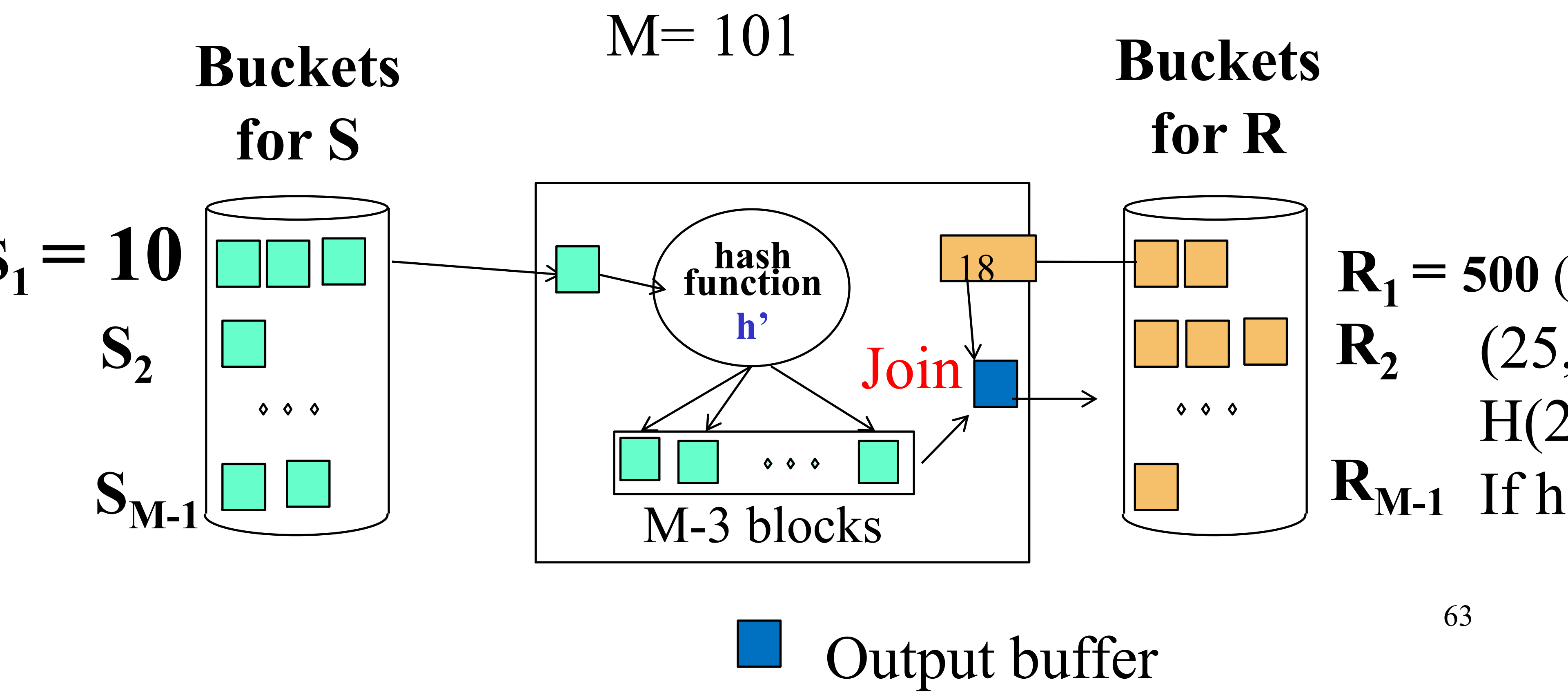
notes

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

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

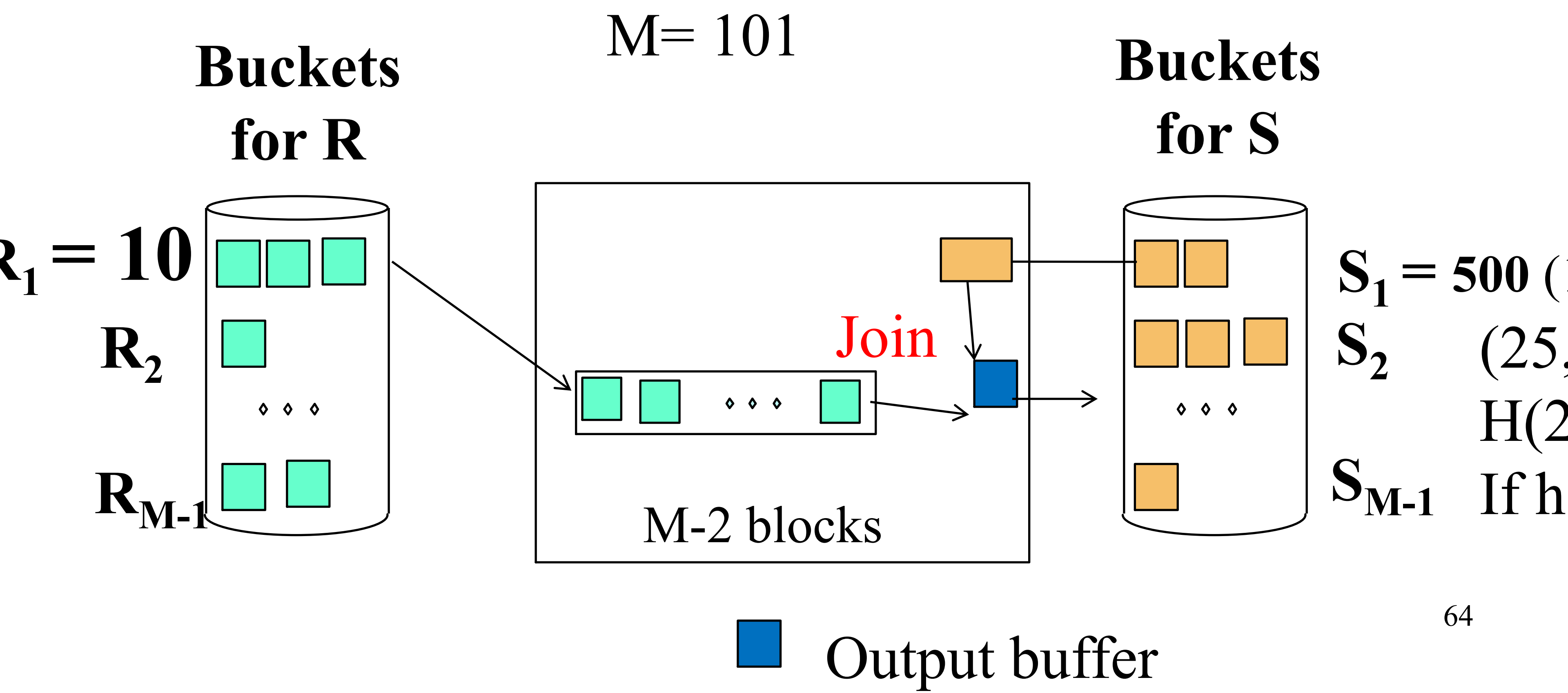
Partitioned Hash-Join: Second Pass

- 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)

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 S1,10
  - Pass 2: join Ri with Si

    cost: B(R) + B(S)

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: …

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

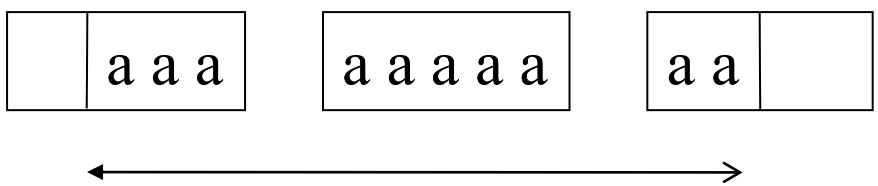
Index-Based Algorithms

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

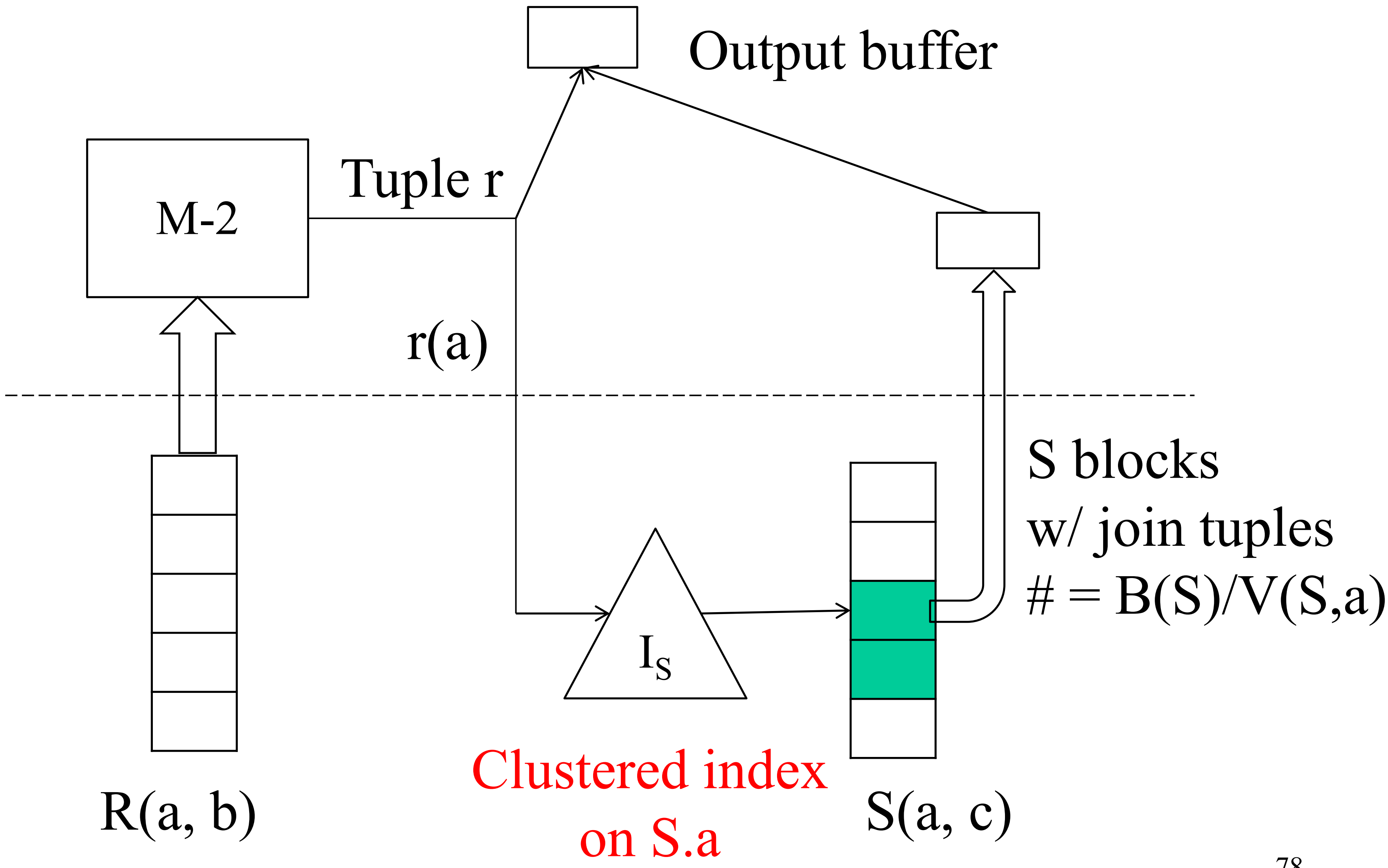
- $R \bowtie S$
- Assume S has an index on the join attribute
- Iterate over R, for each tuple, fetch corresponding tuple(s) from S
- Assume R is clustered. Cost:
  - If index is clustered: $B(R) + T(R)B(S)/V(S,a)$
  - If index is unclustered: $B(R) + T(R)T(S)/V(S,a)$
- Compare this to NLJ (both R &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

Indexed-Based Join vs. NLJ

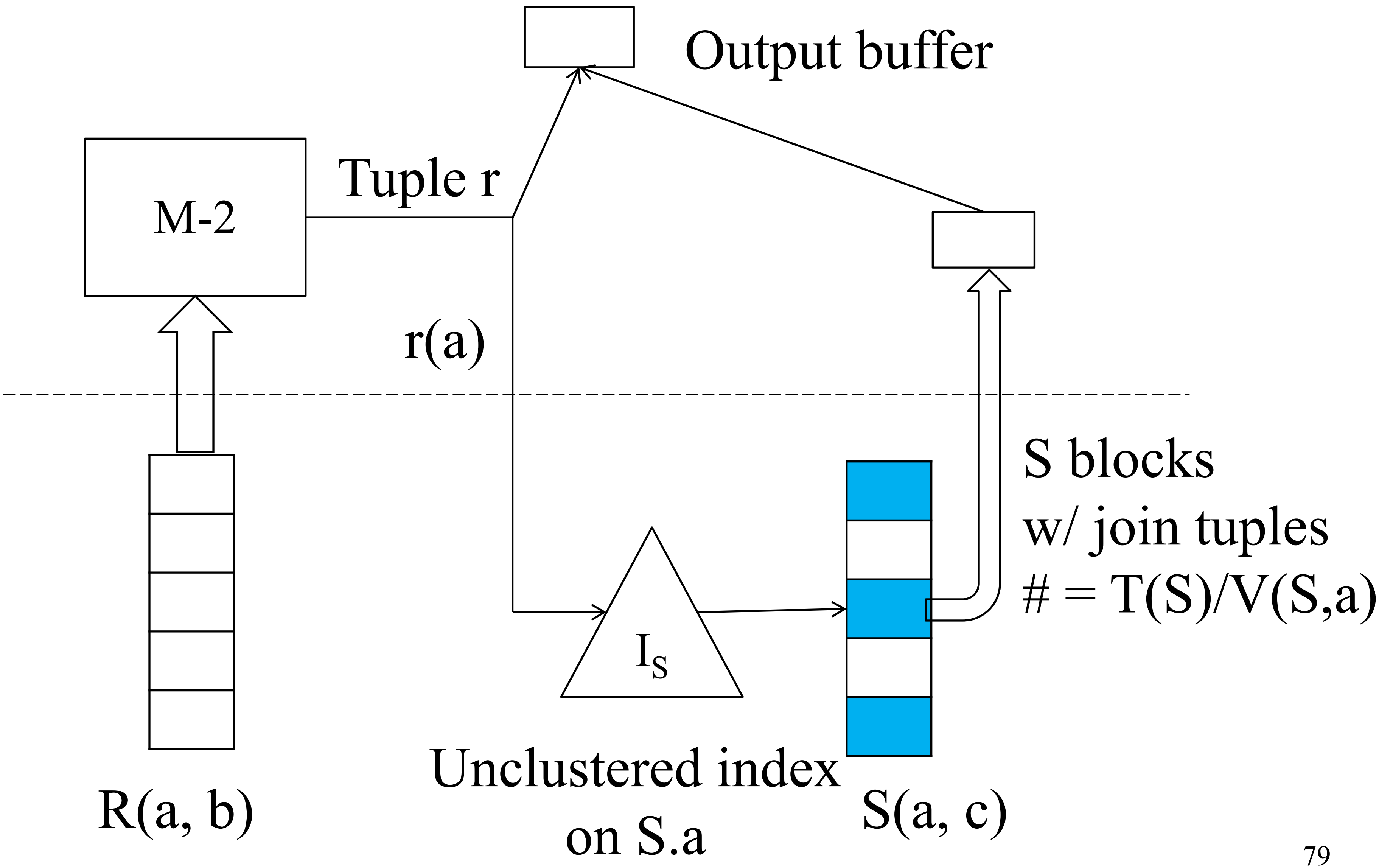
- Index-based (R clustered, clustered index S.a)
  - $B(R) + T(R)B(S)/V(S,a)$
- NLJ (R & S clustered)
  - $B(R) + B(R)/(M-2) * B(S)$

- 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)