CS525: Advanced Database Organization

Notes 6: Query Optimization and Execution

Yousef M. Elmehdwi

Department of Computer Science

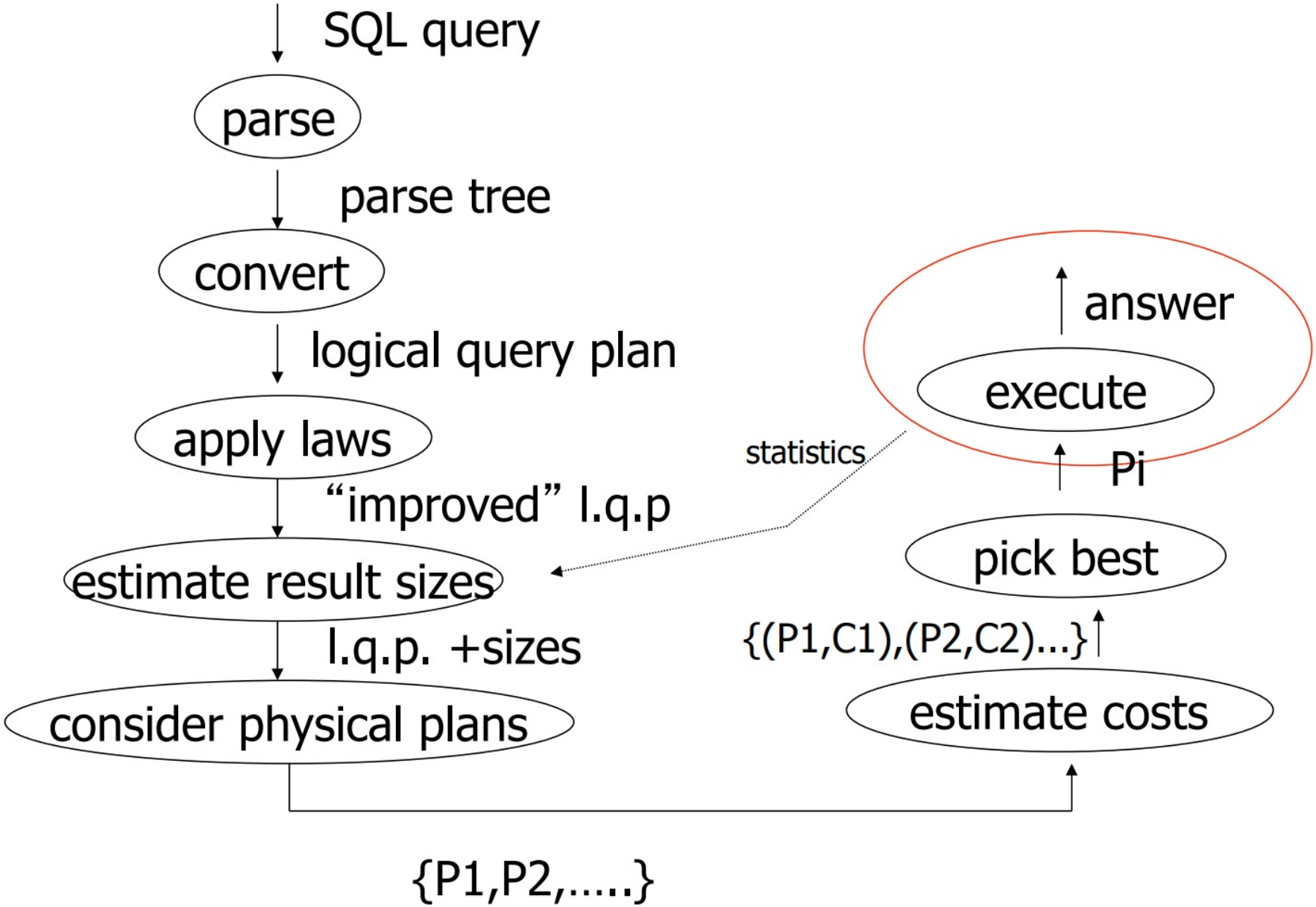
Illinois Institute of Technology

yelmehdwi@iit.edu

October 30, 2018

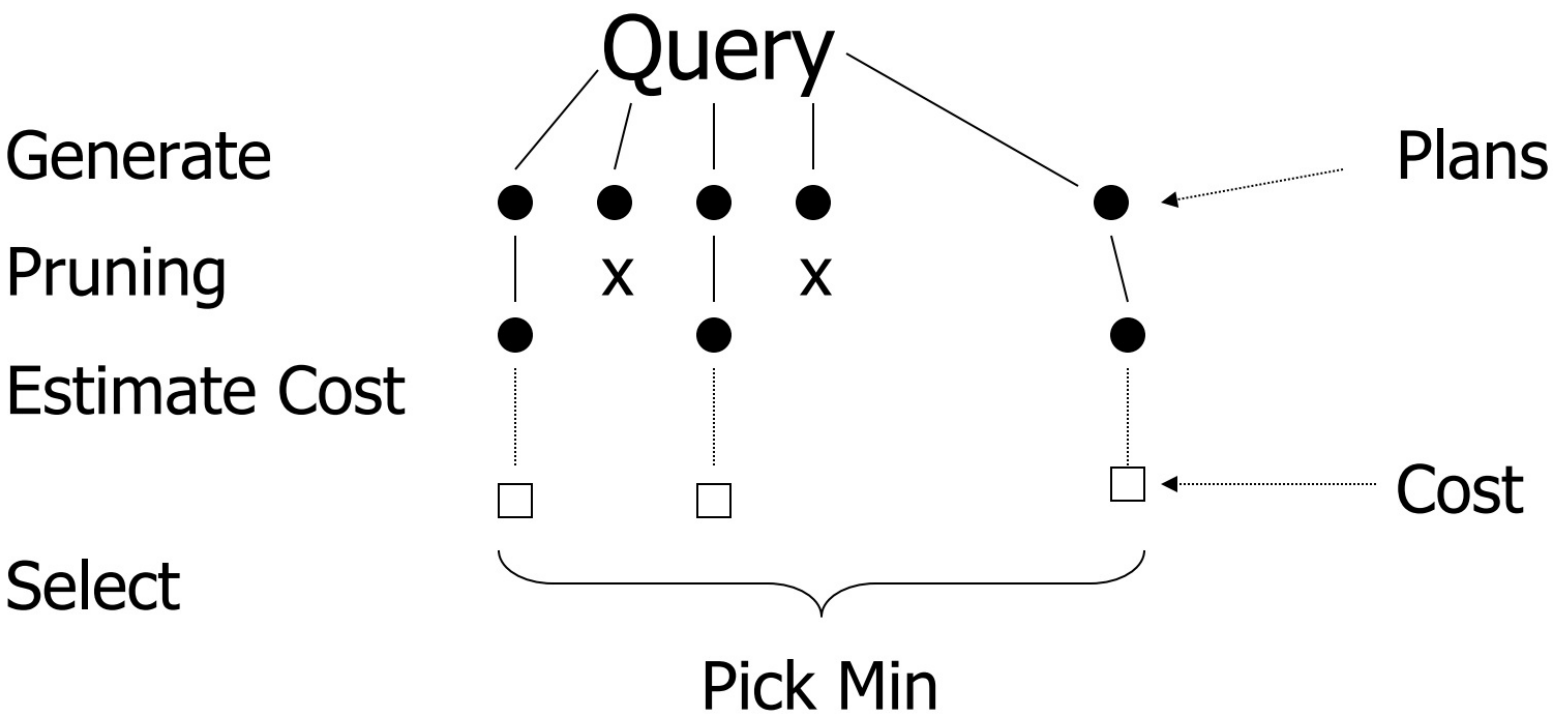
Slides: adapted from a courses taught by Shun Yan Cheung, Emory University, Hector Garcia-Molina, Stanford, & Jennifer Welch, Texas A&M

Query Processing



Query Optimization

- Generating and comparing plans

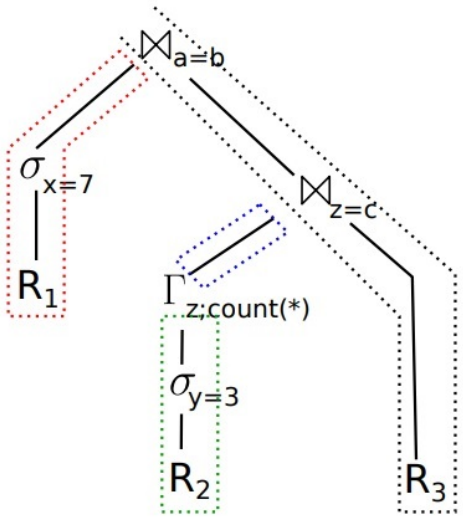


Query Execution

- Query plan is not directly executable.
- need to translate into executable code
- From SQL to executable code[1]

```
select   *
from     R1,R3,
         (select   R2.z,
                   count(*)
          from     R2
          where    R2.y=3
          group by R2.z) R2
where    R1.x=7
and      R1.a=R3.b
and      R2.z=R3.c
```

(a) Example SQL Query



(b) Execution Plan

initialize memory of $\bowtie_{a=b}$, $\bowtie_{c=z}$, and Γ_z
for each tuple t in R_1
  if $t.x = 7$
    materialize t in hash table of $\bowtie_{a=b}$
for each tuple t in R_2
  if $t.y = 3$
    aggregate t in hash table of Γ_z
for each tuple t in Γ_z
  materialize t in hash table of $\bowtie_{z=c}$
for each tuple t_3 in R_3
  for each match t_2 in $\bowtie_{z=c}$ $[t_3.c]$
    for each match t_1 in $\bowtie_{a=b}$ $[t_3.b]$
      output $t_1 \circ t_2 \circ t_3$

(c) Compiled query (pseudo-code)

[1] Compiling Database Queries into Machine Code

Execution Strategies

- Compiled
  - Translate into C/C++/Assembler code
  - Compile, link, and execute code
- Interpreted
  - Generic operator implementations
  - Generic executor
    - Interprets query plan

Virtual Machine Approach

- Implement virtual machine of low-level DBMS operations
- Compile query into machine-code for that machine

Reading: Query Compiler

- How to Architect a Query Compiler, Revisited
- Compiling Database Queries into Machine Code

Query Execution

- Here only
  - how to implement operators
  - what are the costs of implementations
  - how to implement queries
    - Data flow between operators

Execution Plan

- A tree of physical operators that implement a query
- May use indices
- May create temporary relations
- May create indices on the fly
- May use auxiliary operations such as sorting

Physical Operators used in Physical Query Plans

- `Physical query operators`: the programs (algorithms) used to execute a query
- `Physical query plan`: a sequence of `physical query operators` that accomplishes the execution of a query
- Many of the `physical plan operators` have multiple implementations
  - Some implementations of the operator are very efficient but require a large amount of memory
  - While other implementations are less efficient but require a smaller amount of memory
- Amount of buffers will determine which implementation that we can use to process the query

How to estimate costs

- If everything fits into memory
  - Standard computational complexity
- if not
  - Assume fixed memory available for buffering pages
  - Count I/O operations
  - Real systems combine this with CPU estimations

Cost and constraint of a Physical Query Plan

- `Cost of a physical query plan`: the number of disk blocks that are accessed by the execution of the `physical query plan`
- Constraint on a `physical query plan`
  - Memory limitation
    - The operators used in the physical query plan will require a certain minimum buffer allocation requirement
  - The amount of memory available can prohibit the choice of certain (= more efficient) algorithms for an operator
- Query optimization: find the least cost (physical) query plan such that total memory used by the operators $\leq$ total available memory buffers

Assumption in calculating the cost of an operator

- Recall: `Cost of an operator`: the number of disk I/O operations (= # disk blocks) performed by the operator
- Assumption in computing the cost of an operator
  - The output of the operator is left in memory
  - i.e.: The cost of an operation does not include the disk I/O's to write result (to disk).
  - Unless the execution plan needs to write the result to disk (to save memory space)
- Reason:
  - Query plans often use pipelining to execute the operations in the query plans
  - Pipelining passes the output tuples using memory buffers

Estimating disk IOs

- Count # of disk blocks that must be read (or written) to execute query plan
- To estimate costs, we may have additional parameters
  - `B(R)`: # of blocks containing `R` tuples
  - `f(R)`: max # of tuples of `R` per block
  - `M`: # of memory buffers in main memory that is available to an operator
  - `HT(i)`: # levels in index `i`
  - `LB(i)`: # of leaf blocks in index `i`

The basic (relation) access operators and their cost

- There are 2 basic tuple access operators available for the `Physical Query Plan`
  - `Table-Scan(R)`
    - Read tuples from the relation `R` by reading data blocks - one block at a time from disk to memory
  - `Index-Scan(R)`
    - The relation `R` must have an index
    - The index is scanned to find blocks that contain the desired tuples
    - All the blocks containing desired tuples are then read into the memory - one block at a time.

Clustered and unclustered files/relations/indexes

- `Clustered file`: A file that stores records of one relation
- `Unclustered file`: A file that stores records of multiple relation
- `Clustered index`: Index that allows tuples to be read in an order that corresponds to physical order

The cost of the basic scan operators

- Relation R is clustered

Operator	I/O cost	Explanation
Table-Scan	B(R)	Read all blocks, and there are B(R) blocks.
Index-Scan	$\leq$ B(R)	Depends on number of values in the index is scanned

- Relation R is unclustered

Operator	I/O cost	Explanation
Table-Scan	$\sim$ T(R)	Assuming the next tuple is not found in the current block We will read 1 block per tuple
Index-Scan	$\leq$ T(R)	Depends on number of values in the index is scanned

Data flow and how to pass results in a query execution

- A query consists of a hierarchy (= tree) of physical (query plan) operators
- The result of the operators at the lower level in the tree must be passed (up) to some operator at a higher level in the tree
- There are 2 techniques used to pass the result of one operator to another operator
  1. Materialization
  2. Pipelining

1. Materialization

- The result of each operator is produced in its entirety and
- either is written (as a temporal relation) to disk or allowed to take up space in main memory.
- The receiving operator will read the tuples from disk
- Advantage/disadvantages:
  - Simple to implement
  - High I/O costs
  - Low memory size requirement (because we do not use extra memory buffers)
  - Very attractive for small memory computer systems

2. Pipelining

- If the query is composed of several operators results can also be pipelined between operators
  - For example, the result of a join can be directly pipelined into a selection
    - Every record produced by the join is immediately checked for the selection condition(s)
    - Thus, selection is applied on-the-fly
- Within a pipeline, only tuples are passed among operations
  - Each operation has a buffer for storing tuples
- Advantage/disadvantages:
  - Harder to implement: The commonly used technique to implementing pipelining is `Iterator` objects
  - No creation of temporary tables necessary
  - No expensive writing to/ reading from disk
  - High memory requirement (because we need extra memory buffers)
  - Attractive for large memory computer systems

Iterator

- (program) object that provides (at least) the following 3 methods (operations):
- Open: Prepare operator to read inputs. Allocate buffer space for inputs/outputs and passes on parameters (e.g., selection conditions)
- Close: Close operator and clean up. Deallocate all state information
- Next: Return next result tuple. Repetitively call operator specific code and can be used to control progression rates for operators

Pipelining

- Pipelining restricts available operations
- Pipelining usually works well for
  - Selection, projection
  - Index nested loop joins
- Pipelining usually does not work well for
  - Sorting
  - Hash joins and merge joins
- Sometimes, materialization will be more efficient than pipelining
  - Hard to estimate
  - e.g., introducing materializing sorts to allow for merge joins

Categories of the query processing algorithms

- The algorithms in query processing can be broadly categorized as follows
  - One-pass algorithms
  - 2-pass algorithms
  - Multi-pass algorithms

One-pass algorithms

- Read input relations just once
- Use the `lowest` disk I/O operations
- But, also have the highest memory space requirements.
  - Store one relation in memory

2-pass algorithms

- Read input relations 2 times
  - The first time, the operation performs a preparation step (e.g., sort) and write the result to disk
  - The second time, the prepared result is read and the actual operation is performed
- Use a `large` number disk I/O operations
- But, also have a lower memory space requirements
  - They do not store entire relation in memory.
  - Only a portion of the relation is read

Multi-pass algorithms

- Read input relations more than 2 times
- They use the highest disk I/O operations
- But, they also have the lowest memory space requirements

General guideline for picking an algorithm

- If you have sufficient memory space to run a `one-pass algorithm`, do that.
- Otherwise, check if you have sufficient memory space to run a `2-pass algorithm`. If so, do that.
- Otherwise, run a `multi-pass algorithm`
- Note
  - These general guidelines may need to be modified if you can use an index to speed up the access to the tuples/records

Operators Overview

- (External) Sorting
- Joins (Nested Loop, Merge, Hash, ...)
- Aggregation (Sorting, Hash)
- Selection, Projection (Index, Scan)
- Union, Set Difference
- Intersection
- Duplicate Elimination

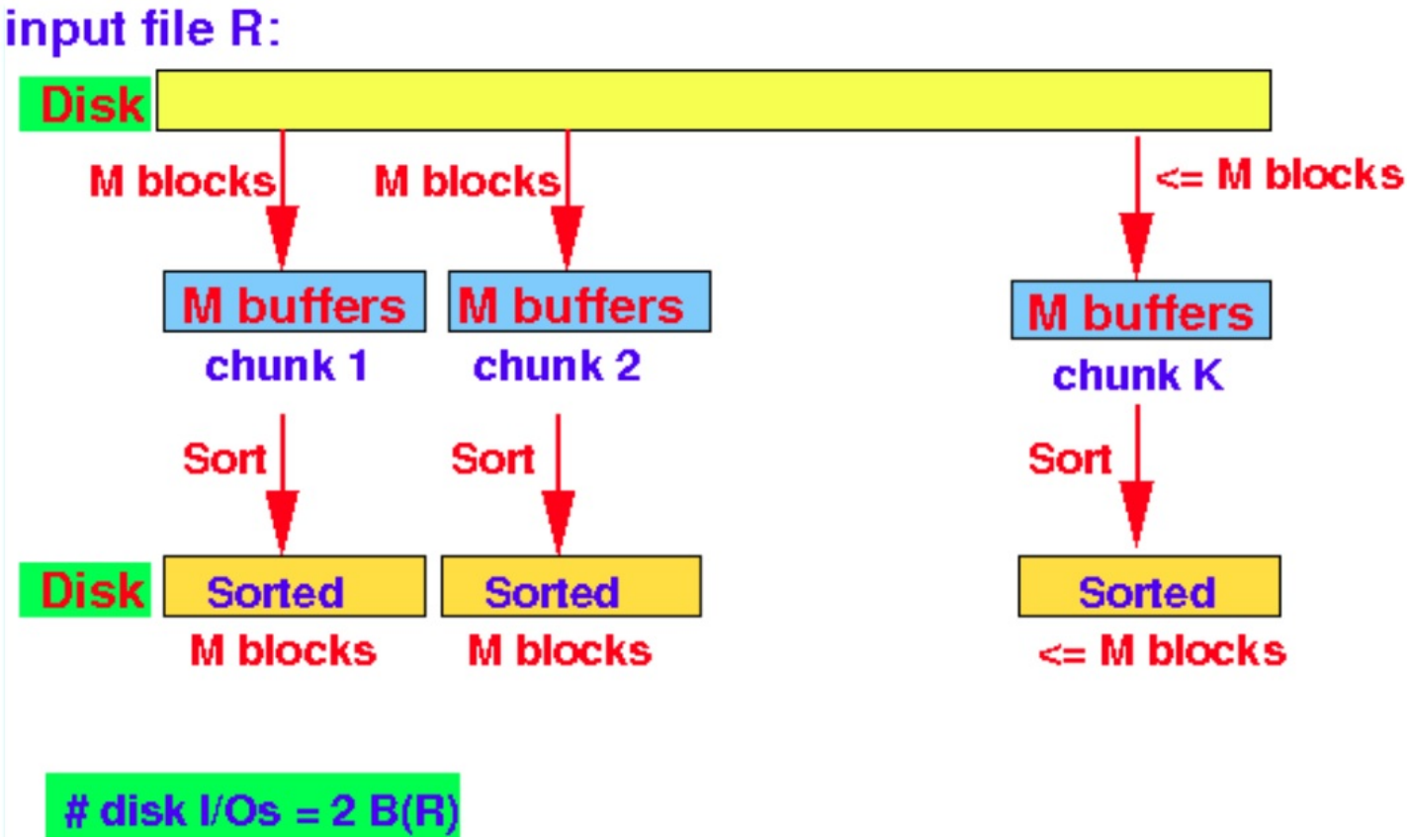
Sorting

- Why do we want/need to sort
  - Query requires sorting (ORDER BY)
  - Operators require sorted input
    - Merge-join
    - Aggregation by sorting
    - Duplicate removal using sorting
- Traditional sort algorithms (e.g., quick sort) requires that the file must fit in the main memory to be sorted
- The `Two-Pass Multiway Merge Sort` (TPMMS) algorithm can be used to sort files that are larger than the main memory

The TPMMS algorithm

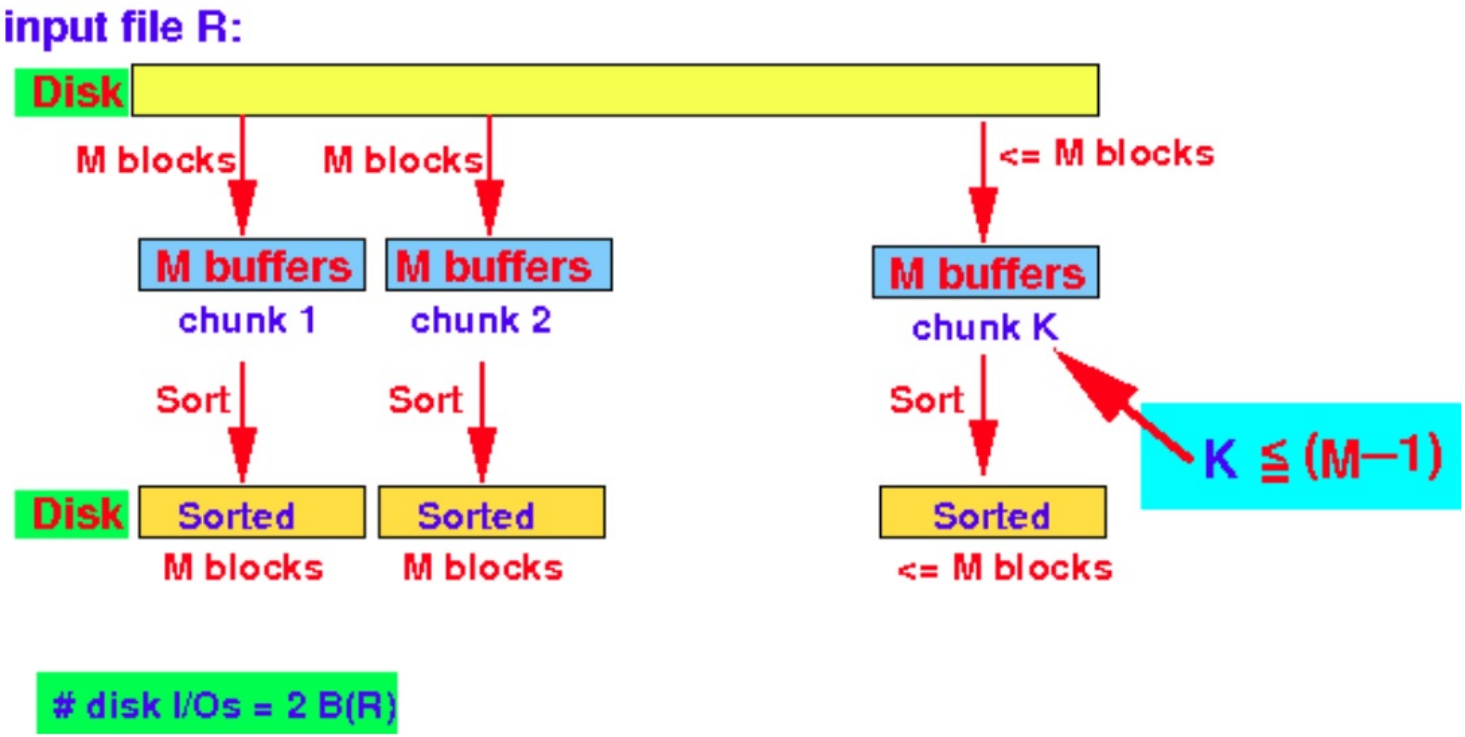
- Suppose
  - There are `M` buffers available for storing the file data
  - `1` buffer can hold `1` data block
- **Pass 1**
  - Divide the input file into chunks of `M` blocks each
  - Sort each chunk individually using the `M` buffers
  - Write the sorted chunks to disk

The TPMMS algorithm: Pass 1



The TPMMS algorithm: Constraint

- The number of chunks (K) $\leq$ M-1
- This constraint is imposed by **Pass 2** of the TPMMS algorithm



The TPMMS algorithm: **Pass 2**

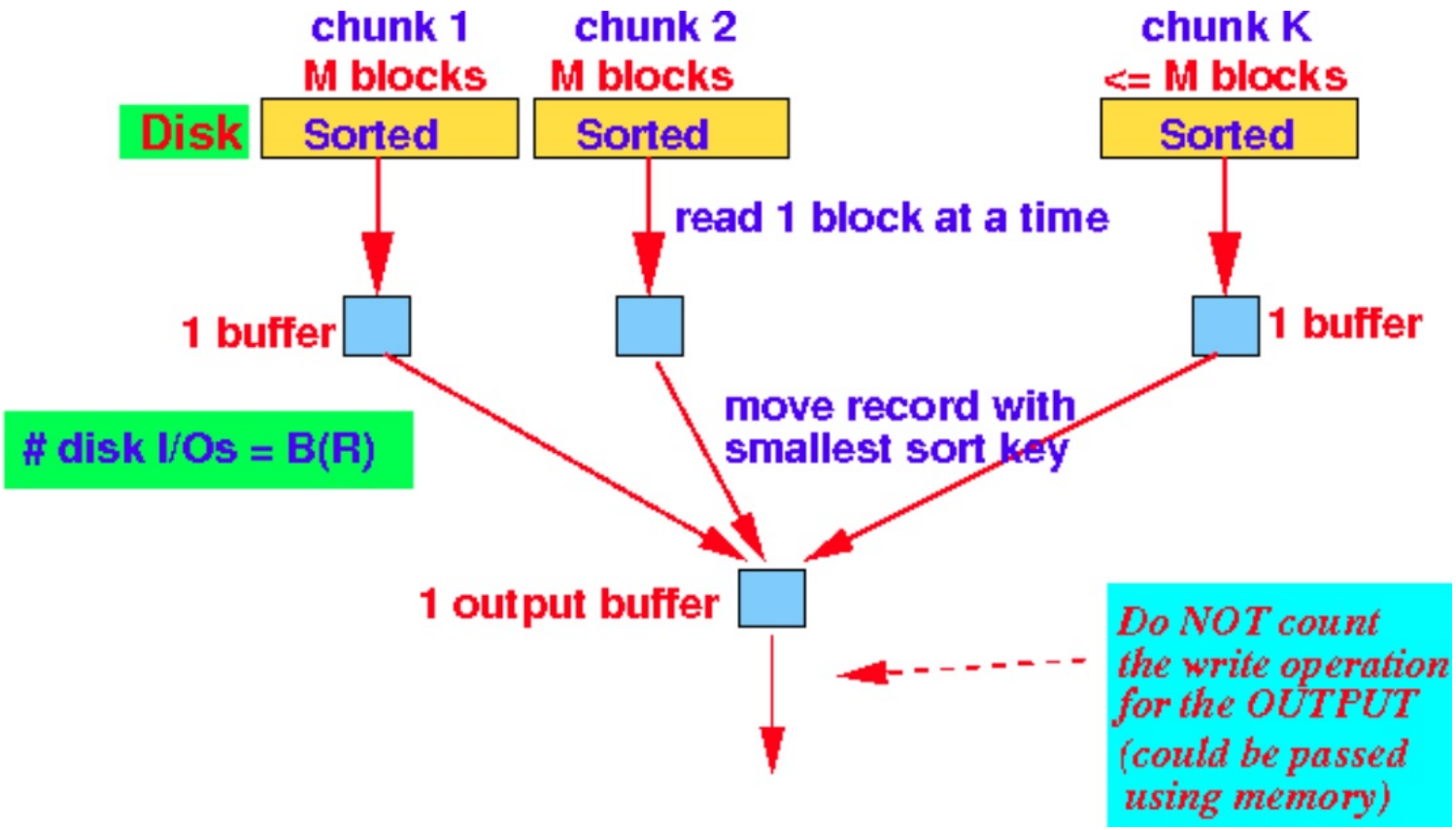
- Divide the `M` buffers into
  - `M - 1 input buffers`
  - `1 output buffer`
- Use the `M - 1 input buffers` to read the `K` sorted chunks (`1` block at a time)

  - Constraint: $K \leq M - 1$
- Use `output buffer` to `merge sort` the `K` sorted chunks together into a sorted file as follows
  - Find the record with the smallest sort key among the `K` buffers
  - Move the record with the smallest sort key to the `output buffer`
    - If the `output buffer` is `full`, then write (= empty) the `output buffer` to disk
  - If some `input buffer` is empty

    - Read the next (sorted) block from the sorted chunk if there is more data

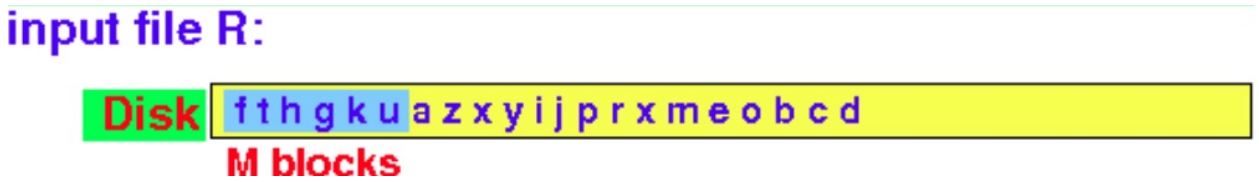
    - If there is no more data in the chunk, then ignore the chunk in the merge operation

The TPMMS algorithm: **Pass 2**



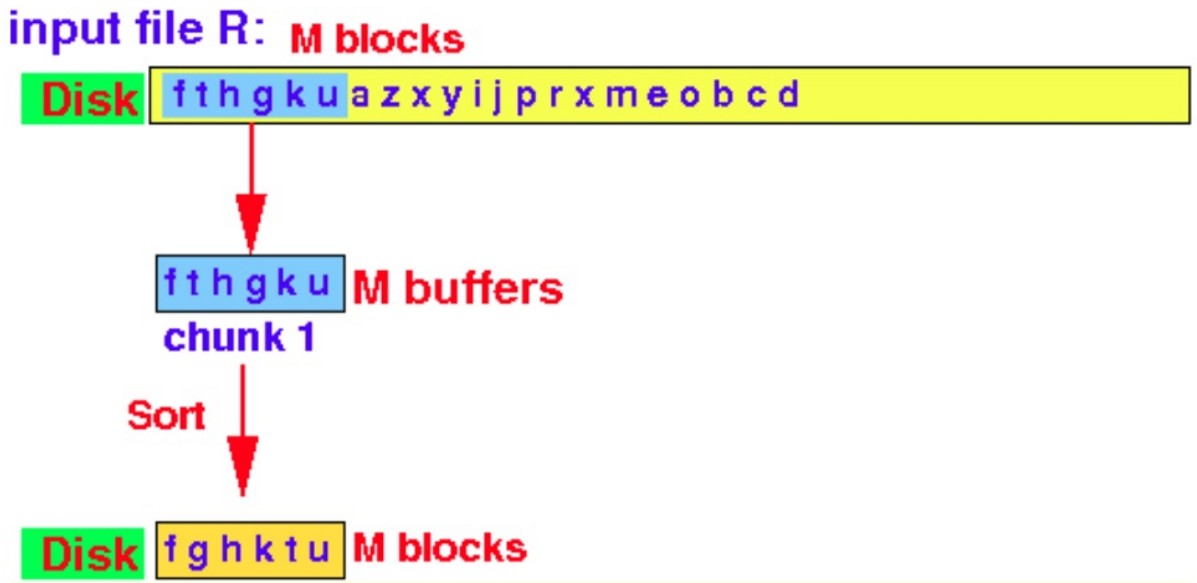
The TPMMS algorithm: Example

- Sort the following input file



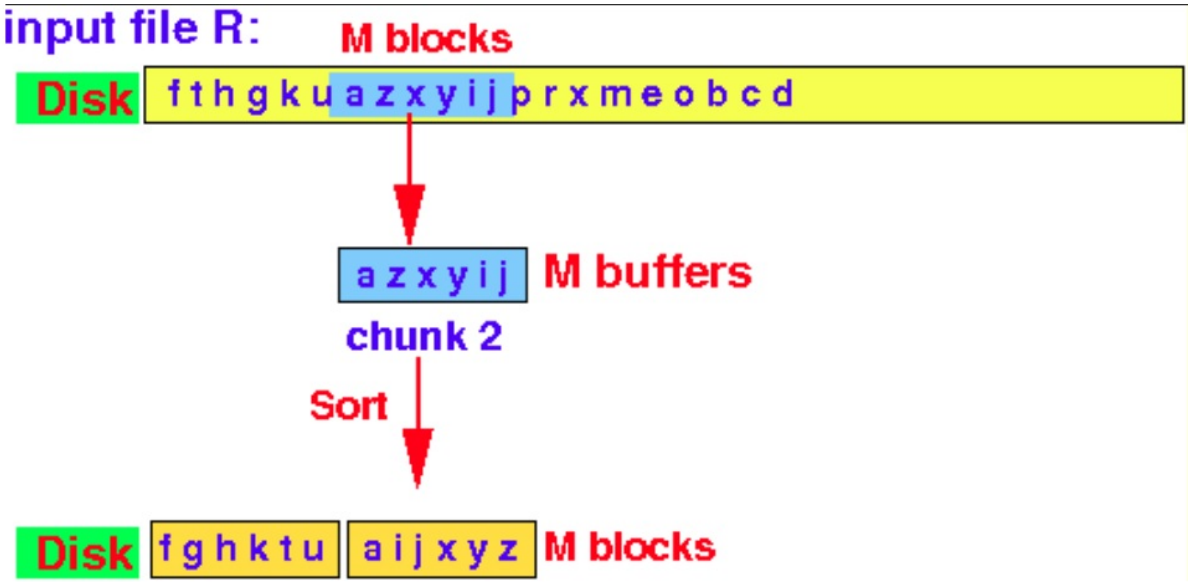
The TPMMS algorithm: Example: Pass 1

- Step 1: sort first chunk of `M` blocks



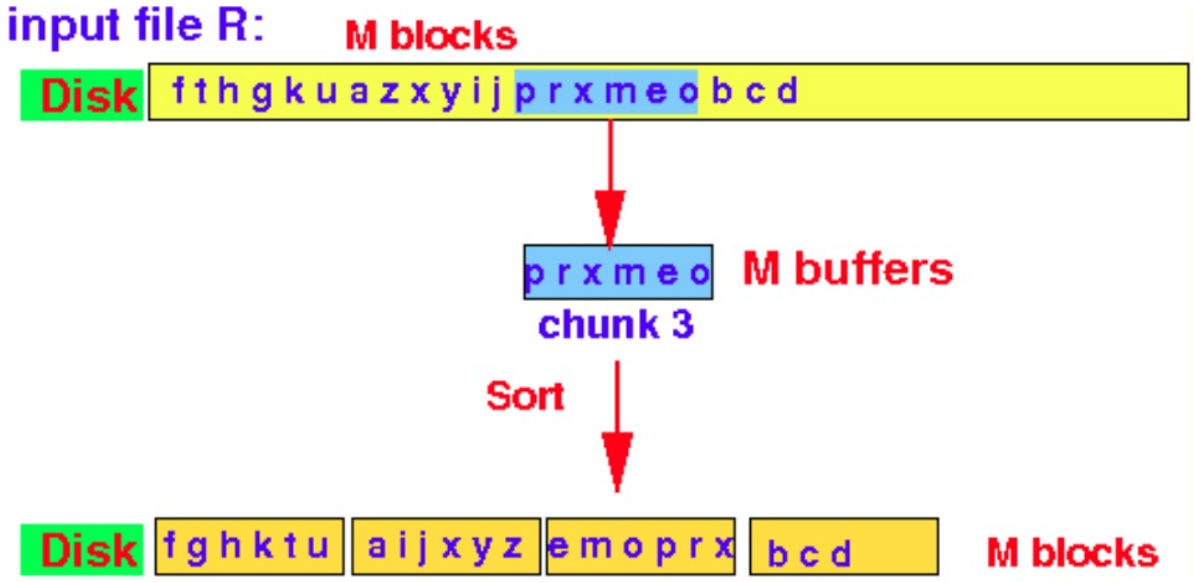
The TPMMS algorithm: Example: Pass 1

- Step 2: sort second chunk of `M` blocks



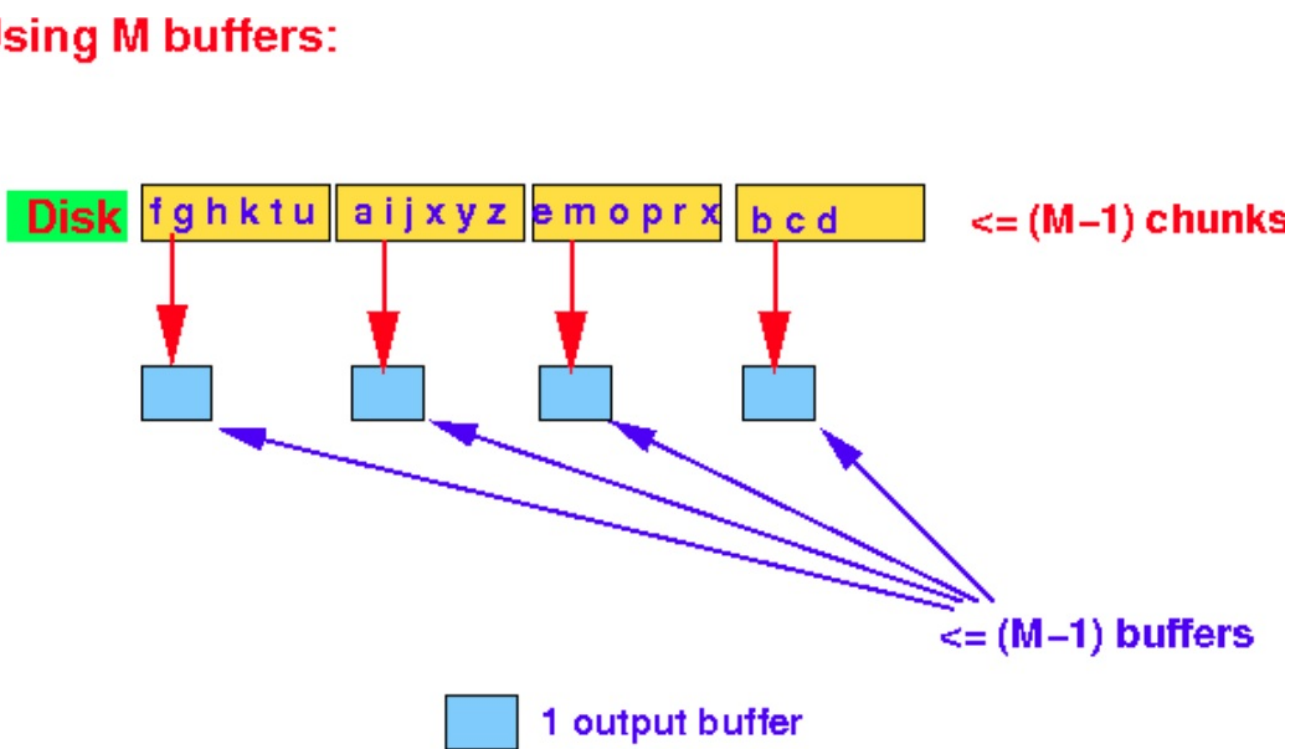
The TPMMS algorithm: Example: Pass 1

- And so on



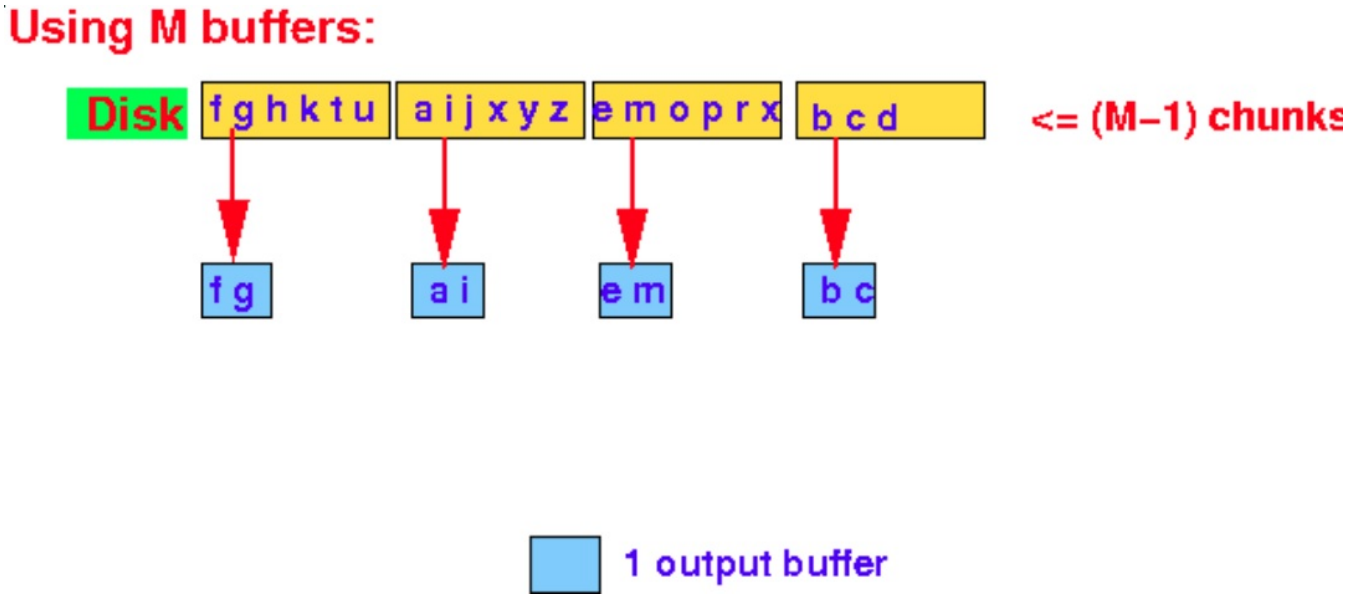
The TPMMS algorithm: Example: Pass 2

- Use `1 buffer` to read each chunk and use `1 buffer` for output



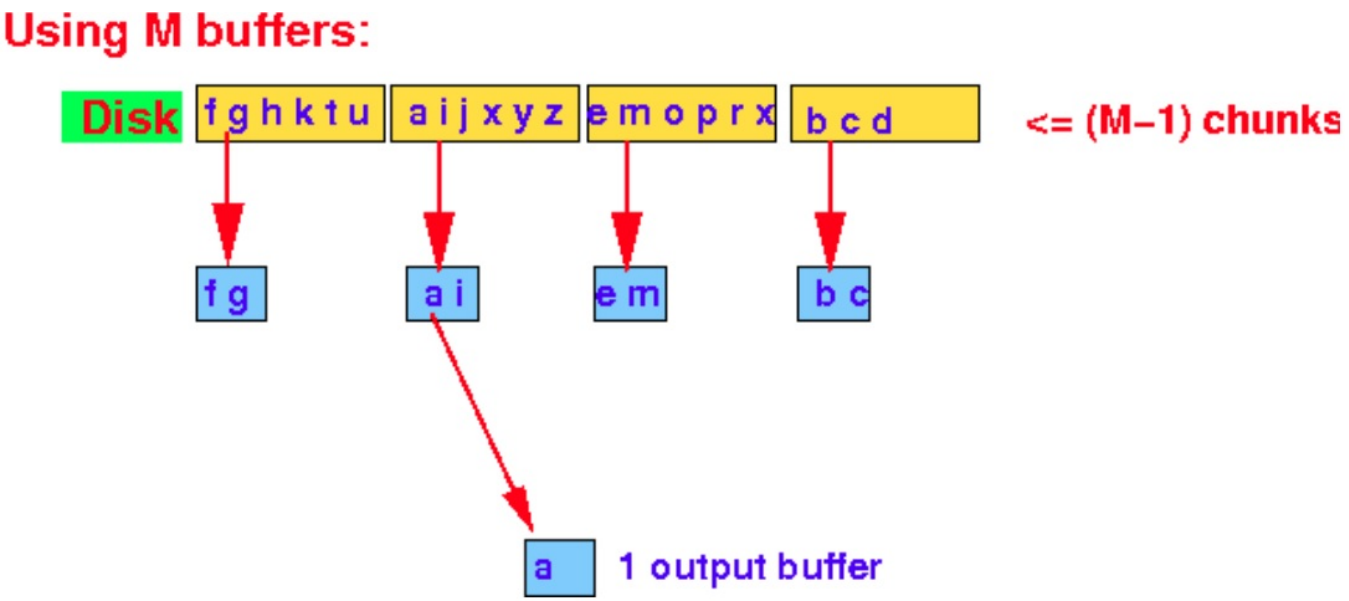
The TPMMS algorithm: Example: Pass 2

- Read each chunk `1 block` at a time



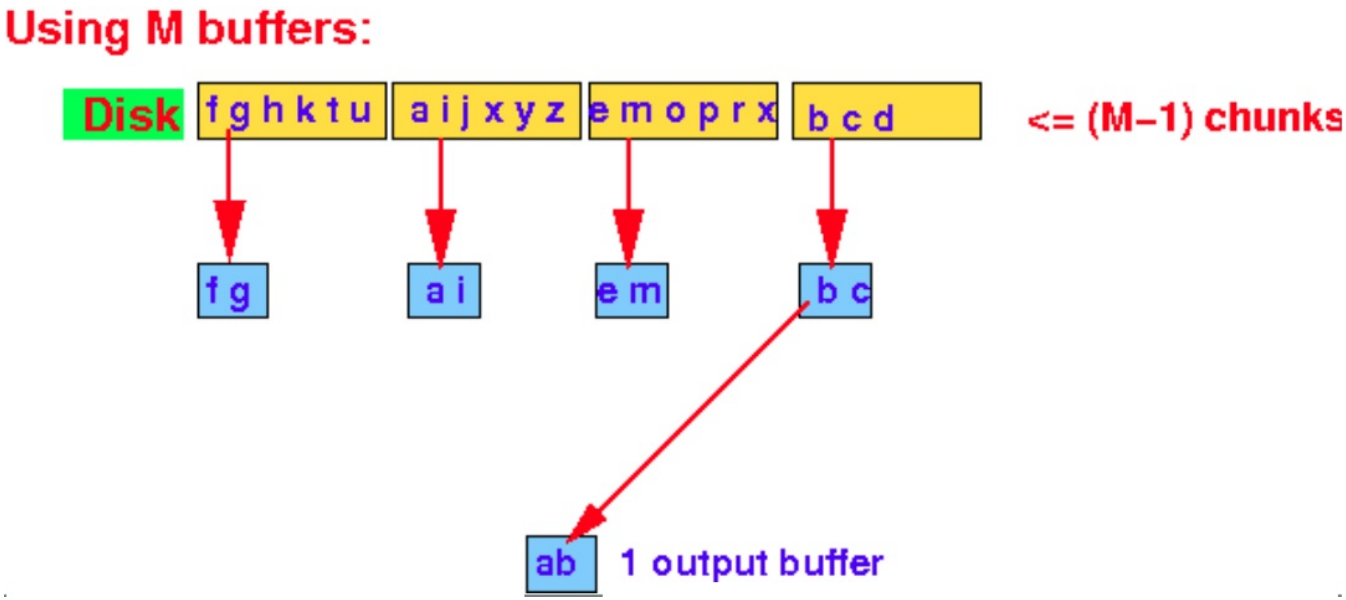
The TPMMS algorithm: Example: Pass 2

- Move the smallest element to the `output buffer`



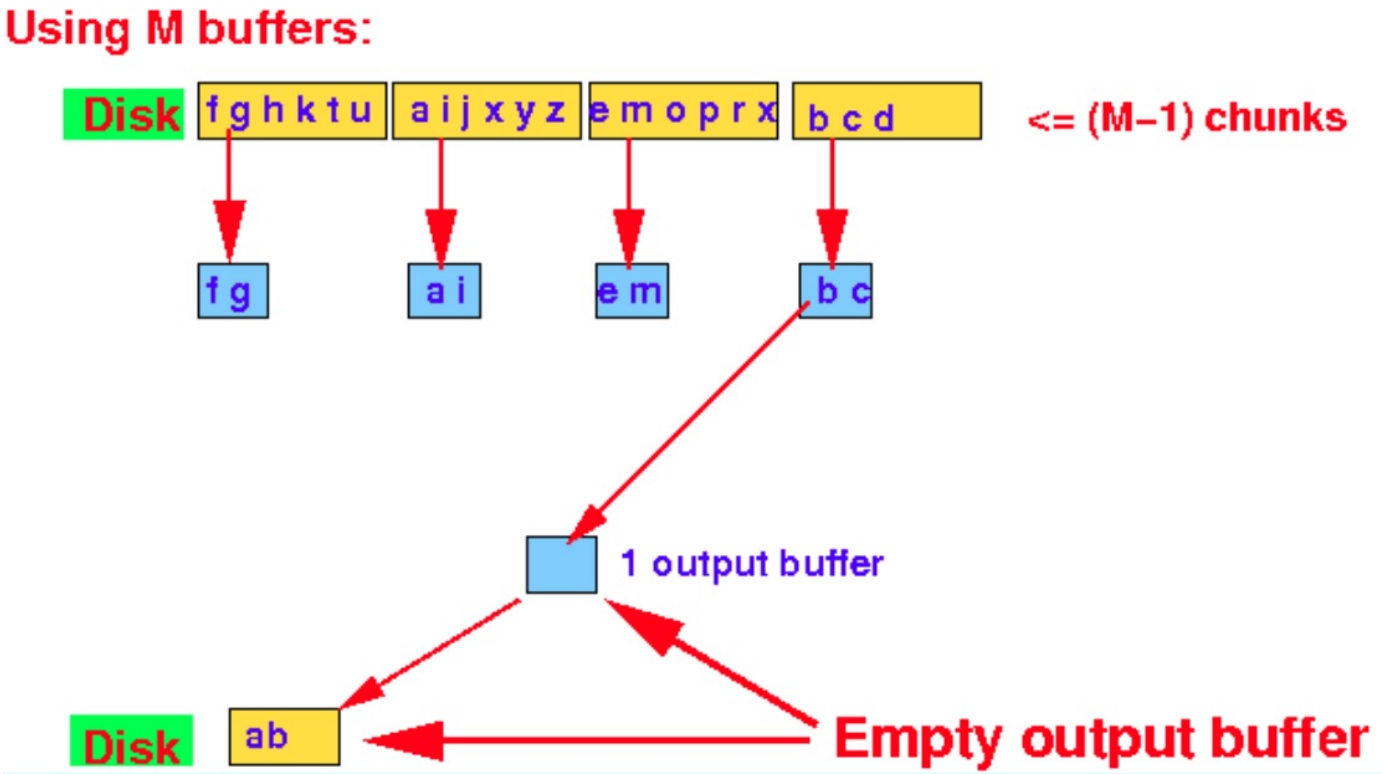
The TPMMS algorithm: Example: Pass 2

- And so on



The TPMMS algorithm: Example: Pass 2

- When `output buffer` is `full`, empty it for re-use



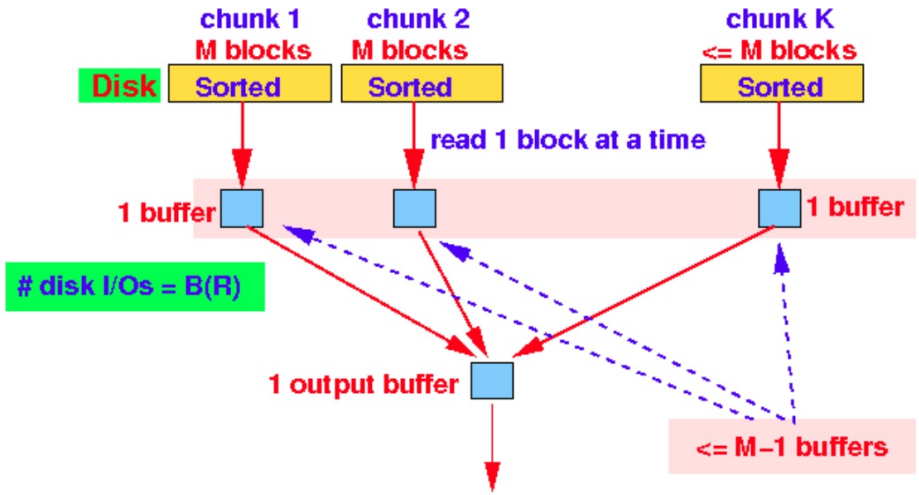
- And so on

File size constraint on the TPMMS algorithm

- The maximum size of a file that can be sorted by the `TPMMS algorithm` is $B(R) \leq M(M - 1)$

File size constraint on the TPMMS algorithm: Reason

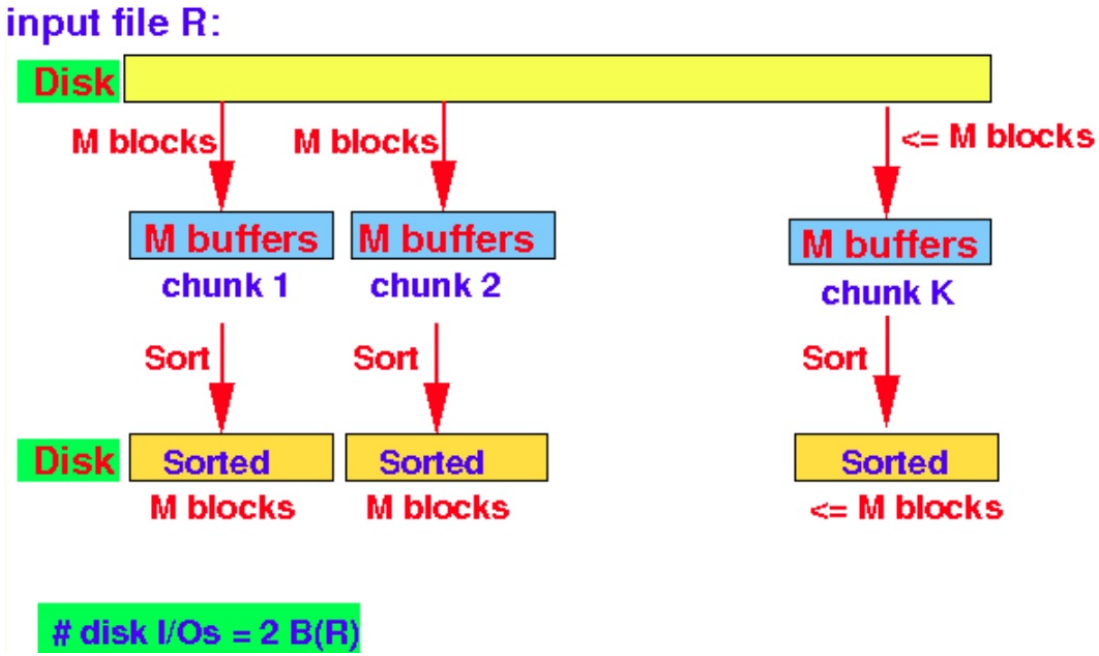
- In **Pass 2**, we can allocate at most `M-1` buffers to read the sorted chunks



- Therefore, the number of chunks that can be generated by **Pass 1** must be $\leq$ `M - 1`

File size constraint on the TPMMS algorithm: Reason

- Each chunk has the size of `M` blocks



- Therefore, the maximum file size is
  - number of chunks in **Pass 1** $\leq$ `M - 1`
  - `1` chunk $=$ `M` blocks
  - $\therefore$ `File size` $\leq$ `(M - 1)` $\times$ `M` blocks

    `B(R)` $\leq$ `M`^{2}. $\Rightarrow$ The memory constraint required to run the TPMMS algorithm is $\sqrt{B(R)} \leq$ `M`

Cost of running TPMMS of a relation R

- cost of TPMMS on relation `R` = 3 $\times$ `B(R)`
- But, we do not include the output (write) cost in the total because we could use pipelining to pass the tuples to the next operator
- If the result is to be written to disk (i.e., materialize), then cost of TPMMS on relation `R` = 4 $\times$ `B(R)`

TPMMS can sort very large files

- Suppose
  - 1 block = 64 K bytes
  - Memory size = 1 G bytes
- M=$\frac{1G}{64K}$ `buffers` = $\frac{1{,}024{,}000{,}000}{64{,}000} = 16,000$ `buffers`
- $\Rightarrow$ Max file size that can be sorted

$$
\begin{aligned}
B(R) &\leq M(M-1) \ \ \textit{blocks} \\
&\leq 16,000(16,000-1) \\
&\leq 255984000 \ \textit{blocks} \quad (1 \ \textit{block} = 64 \ K) \\
&\leq 255984000 \times 64K \\
&\leq 16382976000 \ K \\
&\cong 16 \ \textit{Tera Bytes}
\end{aligned}
$$

When to use 2-Pass algorithms in relational algebra operations

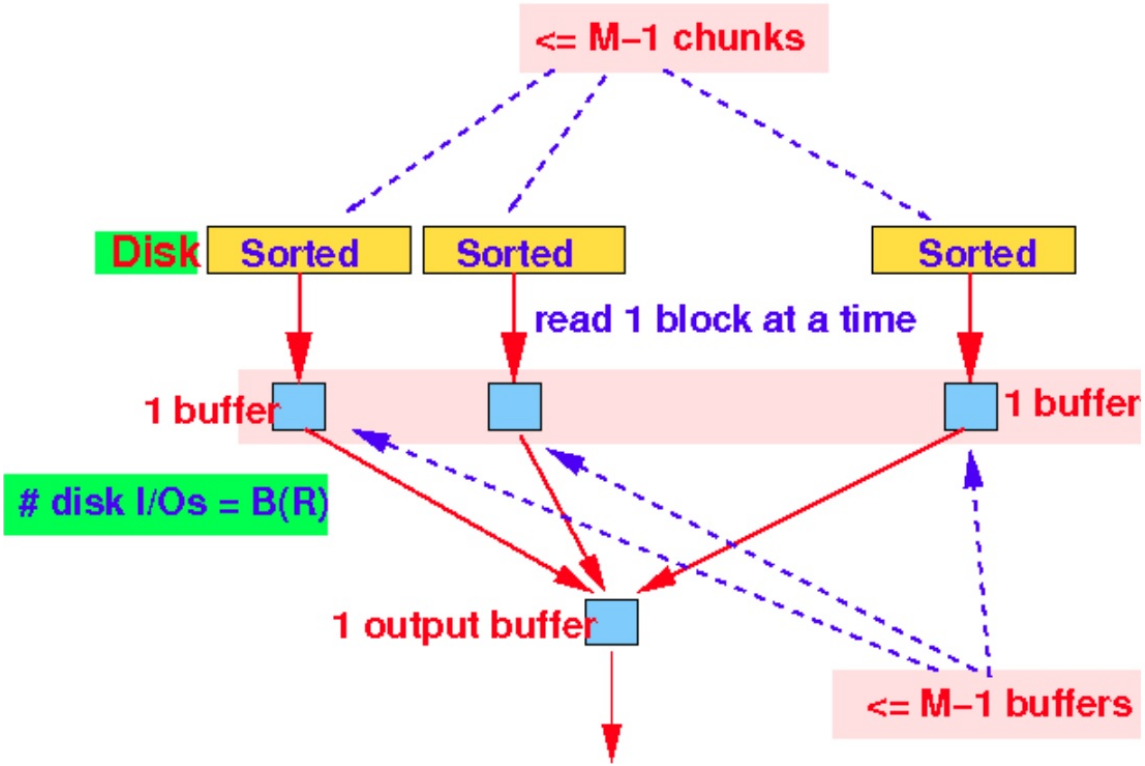
- Unary operator: If `B(R)` $\geq$ available `# buffers (M)` then use a `2-Pass` algorithm (if `B(R)` $\leq$ `M`^{2})
- Binary operator: If `B(R)` $\geq$ available `# buffers (M)` and `B(S)` $\geq$ available `# buffers (M)` then use a `2-Pass` algorithm (if `B(R)+B(S)` $\leq$ `M`^{2})
- Multi-Pass MMS: The `2-Pass` multiway merge sort can be generalized to `multi-Pass` (3 Passes or more)

The multi-pass multiway merge sort algorithm

- Recall the 2-pass multiway sort (TPMMS) algorithm
  - Pass 1
    - Divide the input file into chunks of `M` blocks each
    - Sort each chunk individually using the `M` buffers
    - Write the sorted chunks to disk
    - Requirement: The number of chunks (`K`) $\leq$ `M - 1`
  - Pass 2
    - Divide the M buffers into: `M - 1` input buffers and `1` output buffer
    - Use the `M - 1` input buffers to read the `K` sorted chunks (`1` block at a time)
    - Merge sort the `K` sorted chunks together into a sorted file using `1` output buffer as follows:
      - Find the record with the smallest sort key among the `K` buffers
      - Move the record with the smallest sort key to the output buffer
      - When the output buffer is full, then write the output buffer to disk
      - When some input buffer is empty, then read another block from the sorted chunk if there is more data

An important observation of the TPMMS algorithm

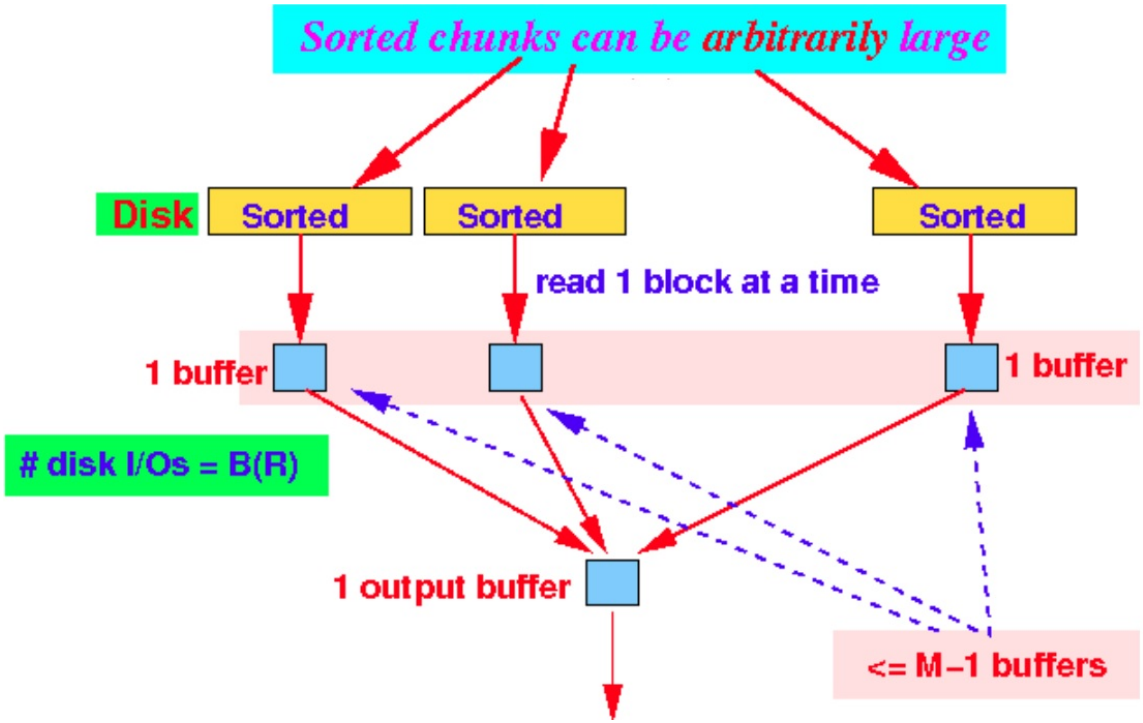
- Consider the Pass 2 in the TPMMS algorithm



- The restriction is `# (sorted) chunks` $\leq$ `M -1` because `# buffers used must be` $\leq$ `M`

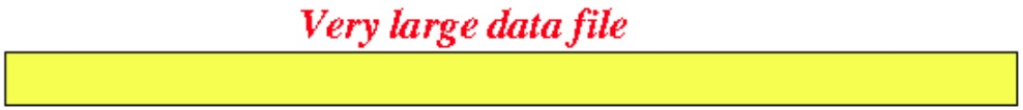
An important observation of the TPMMS algorithm

- There is no restriction on the size of a (sorted) chunk

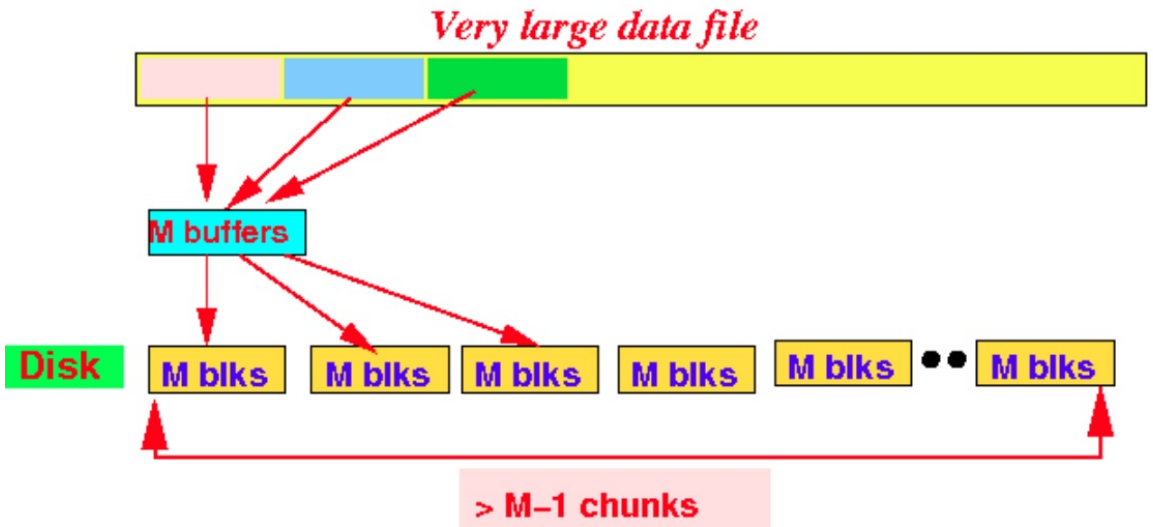


Increasing the size of sorted chunks

- Fact: We can use the `M` buffers to merge sort (any number) $\leq$ `M - 1` sorted chunks into one larger (sorted) chunk
- Suppose we have a very large file:



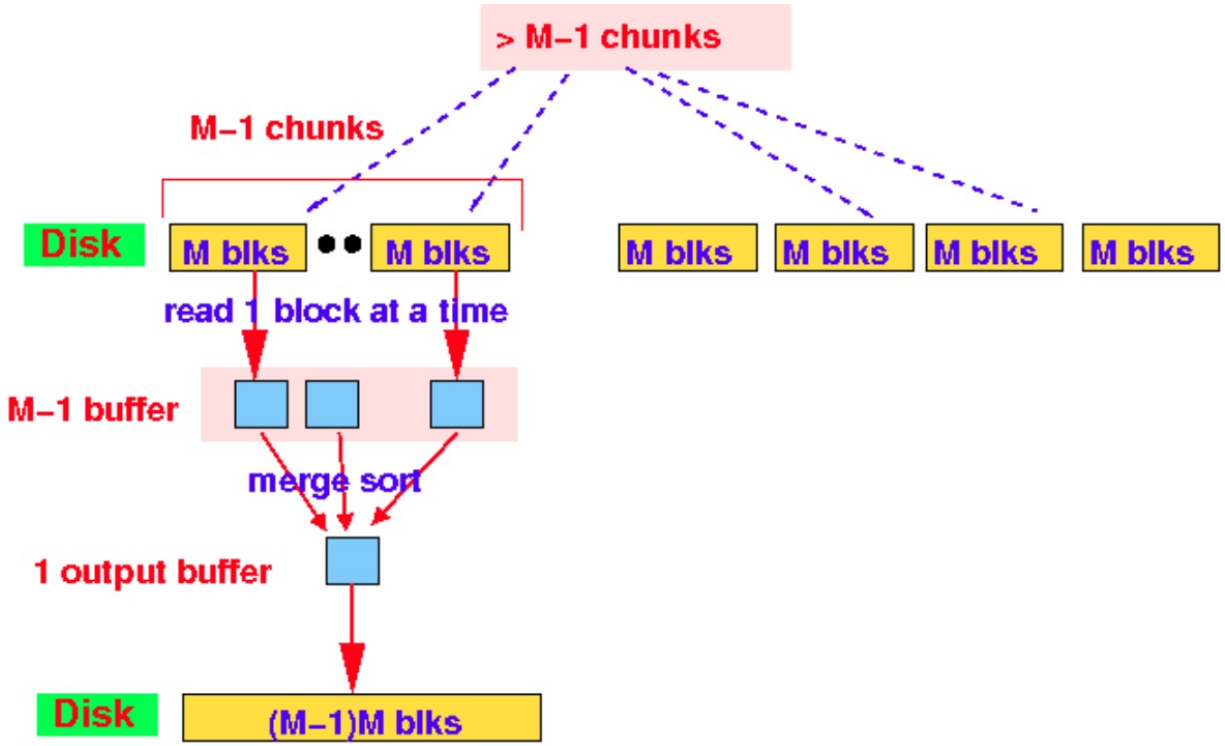
- We first use `M` buffers to sort the file into chunks of `M` blocks



- Suppose we get $>$ `(M - 1)` chunks

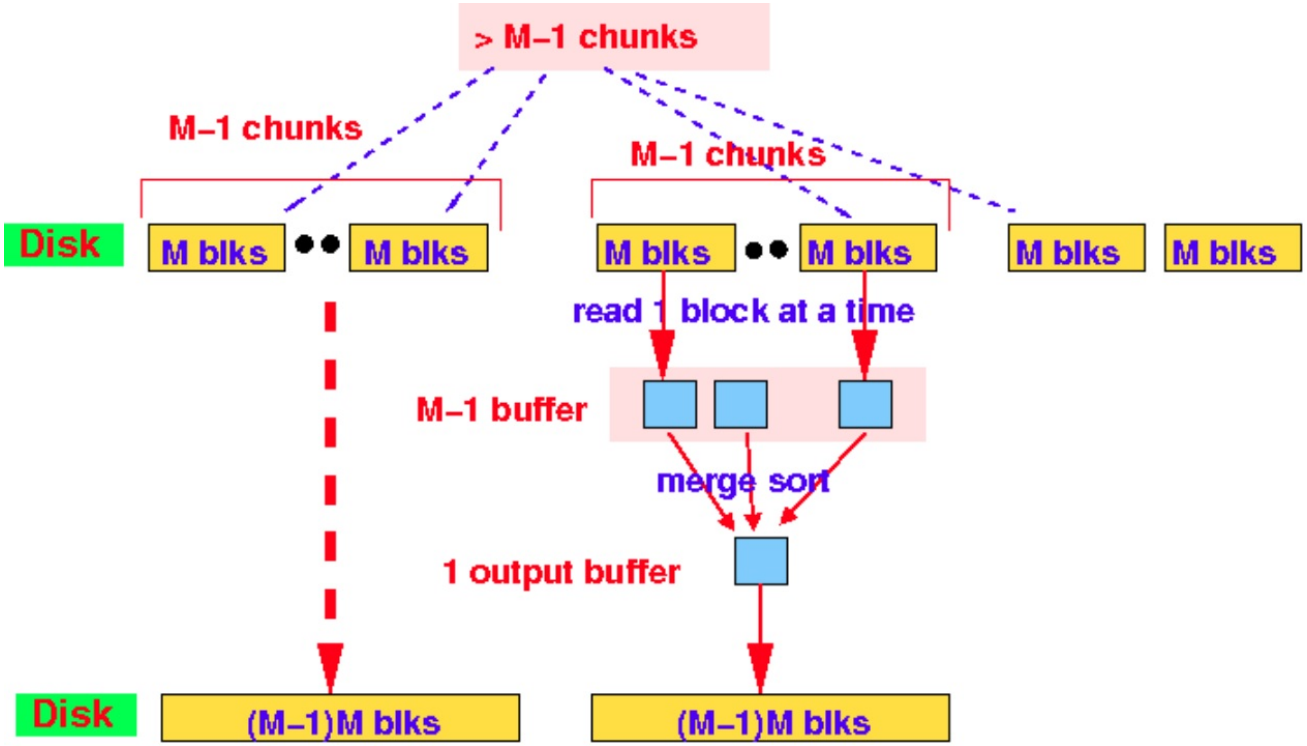
Increasing the size of sorted chunks

- We (re)-use the `M` buffers to merge the first `(M - 1)` chunks into a chunk of size `M(M - 1)` blocks



Increasing the size of sorted chunks

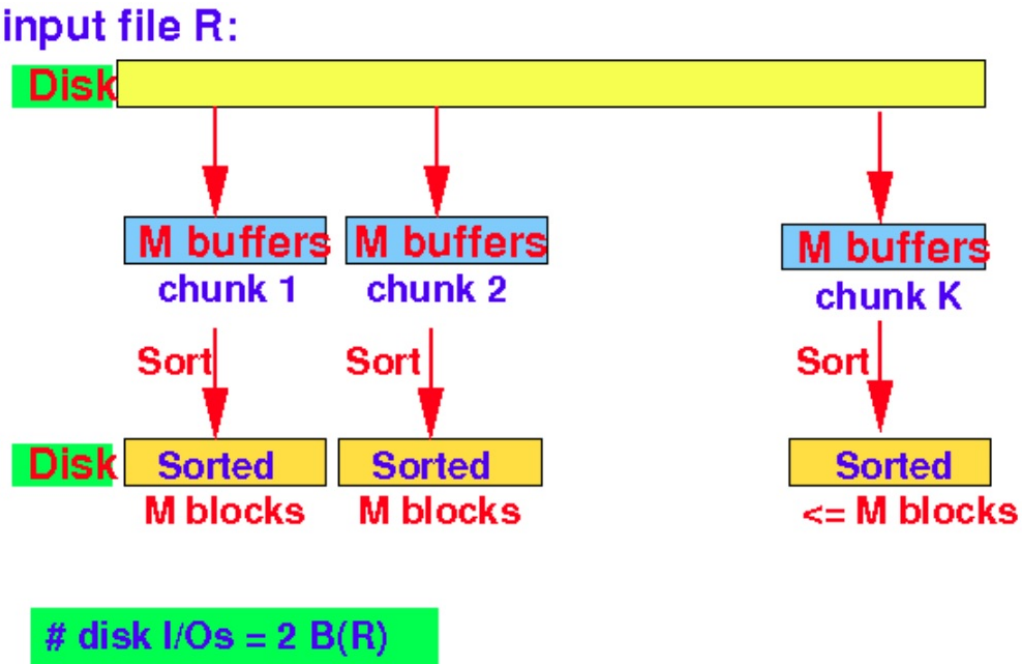
- Then, we (re)-use the `M` buffers to merge the 2^{nd} `(M - 1)` chunks into a chunk of size `M(M - 1)` blocks



- And so on. We will have large sorted chunks

A 3-Pass Multiway Sort Algorithm

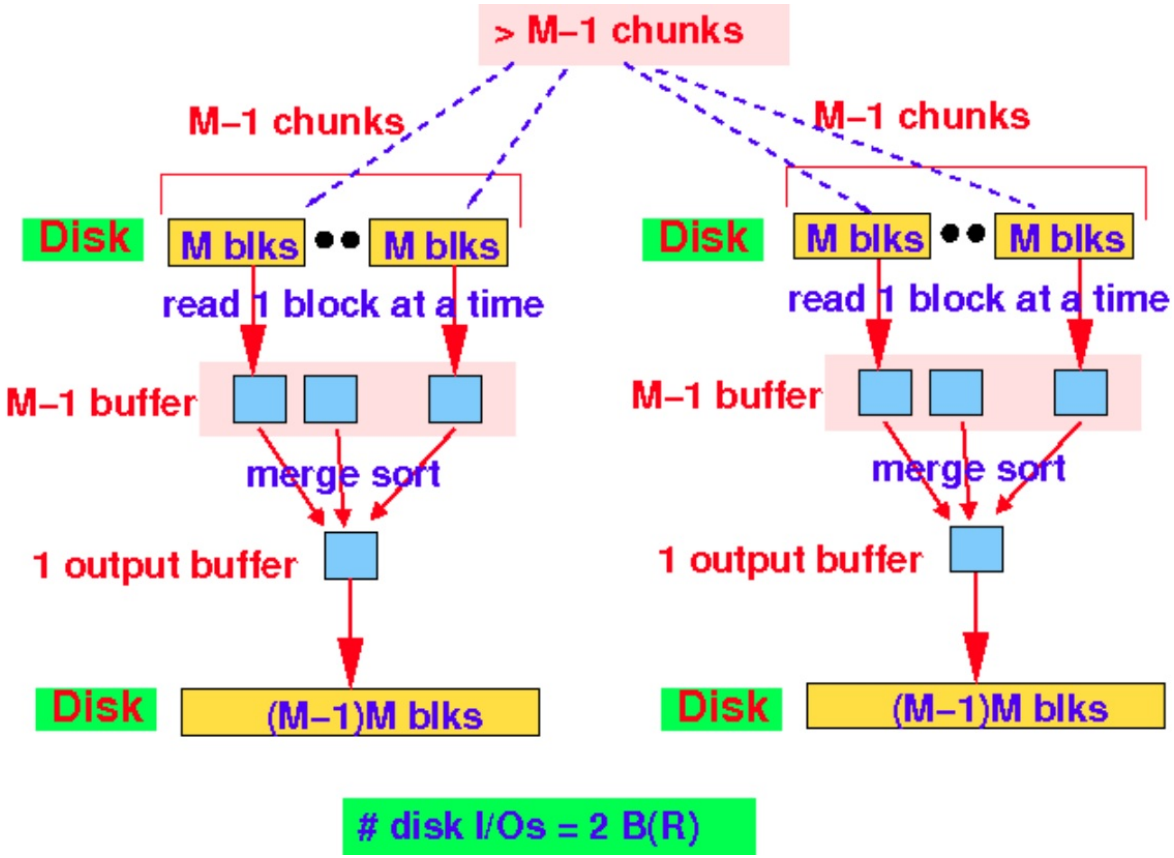
- Pass 1: (same as Phase 1 in TPMMS)
  - Divide the input file into chunks of `M` blocks each
  - Sort each chunk individually using the `M` buffers
  - Write the sorted chunks to disk



- Number of disk I/Os used in step:
  - `B(R)` disk I/Os to read the entire input file plus
  - `B(R)` disk I/Os to write the entire file (= all the sorted chunks)

A 3-Pass Multiway Sort Algorithm

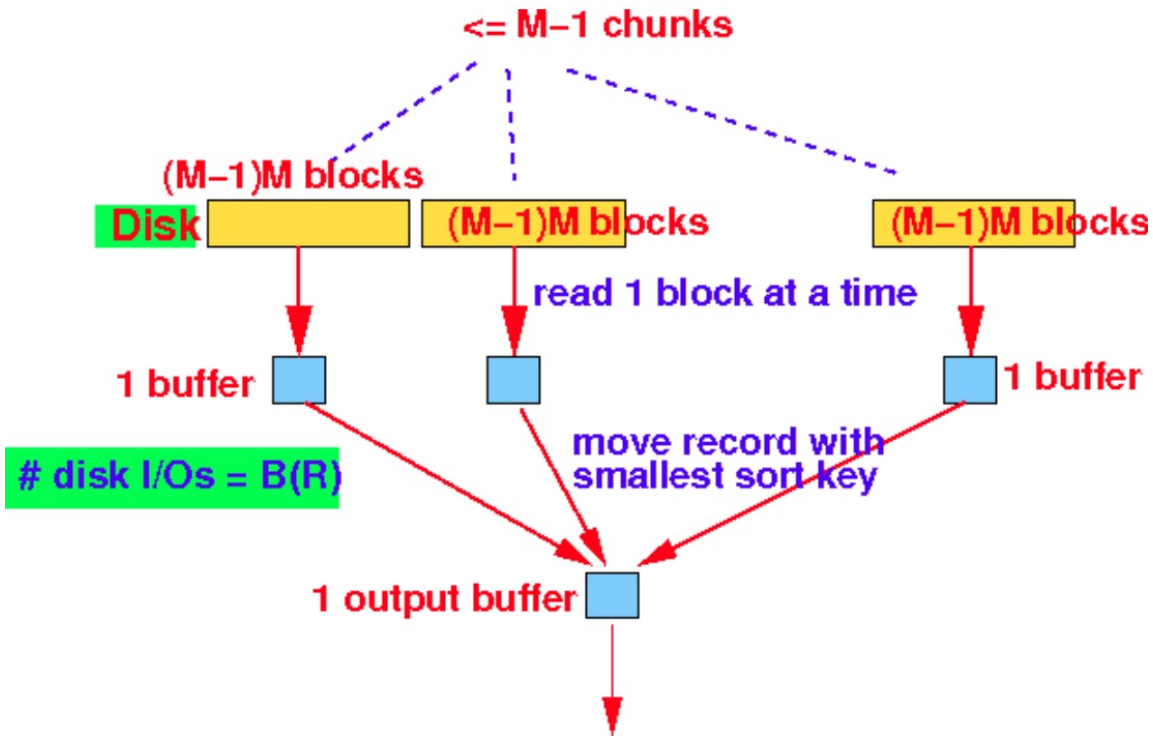
- Pass 2: merge groups of `(M - 1)` sorted chunks into a `M(M - 1)` block sorted "super" chunk and write sorted "super" chunk to disk



- Number of disk I/Os used in step:
  - `B(R)` disk I/Os to read the entire file (= all the chunks in the file) plus
  - `B(R)` disk I/Os to write the entire file (= all the "super" chunks in file)

A 3-Pass Multiway Sort Algorithm

- Pass 3: merge upto `(M - 1)` (much larger) sorted chunks



A 3-Pass Multiway Sort Algorithm

- Maximum file that can be handled by the 3-Pass Multiway Sort:
  - There are no memory restrictions on `Pass 1` and `Pass 2`
  - `Pass 3` must merge: $\leq$ `M - 1` chunks
  - Each chunk in `Pass 3` has size $\leq$ `M` $\times$ `(M - 1)` blocks
  - Maximum file size $\leq$ `M` $\times$ `(M - 1)`^{2} blocks
- Number of disk IOs used
  - `Pass 1:` read R + write sorted chunks = `2` $\times$ `B(R)`
  - `Pass 2:` read sorted chunks + write bigger chunks = `2` $\times$ `B(R)`
  - `Pass 3:` read bigger chunks + sort = `1` $\times$ `B(R)`
  - `Total = 5` $\times$ `B(R)`, (let's not count final output write to disk)

k-Pass Multiway Merge Sort

- `Pass 1`
  - Same as `Pass 1` of TPMMS
  - Cost: `2` $\times$ `B(R)` disk IOs
  - Size of each chunk at end: `M` blocks
- `Pass 2, 3, ..., k-1` (`(k-2) passes`):
  - Use the "increase chunk size" algorithm
  - Cost per pass: `2` $\times$ `B(R)` disk IOs
  - Cost for `k-2` passes: `2(k-2)` $\times$ `B(R)` disk IOs
  - Size increase per pass: `(M - 1)` times
  - Size of chunks at end: `M` $\times$ `(M - 1)`$^{k-2}$ blocks
- `Pass k`
  - Same as `Pass 2` of TPMMS
  - Cost: `B(R)` disk IOs (do not count output)
  - Size of sorted file $\leq$ `M` $\times$ `(M - 1)`$^{k-1}$ blocks

k-pass multiway merge sort

- Total # disk IOs performed by the `k-Pass multiway merge algorithm`
  - # disk IOs = `(2k - 1)` $\times$ `B(R)` (if we don't count final output IO)
    Or:
  - # disk IOs = `2` $\times$ `k` $\times$ `B(R)` (if we include final output IO)
  - `Max file size` $\leq$ `M` $\times$ `(M - 1)`$^{k-1}$ blocks

Operators Overview

- (External) Sorting
- Joins (Nested Loop, Merge, Hash, ...)
- Aggregation (Sorting, Hash)
- Selection, Projection (Index, Scan)
- Union, Set Difference
- Intersection
- Duplicate Elimination

Joins

- Example $\mathtt{R} \bowtie \mathtt{S}$ over common attribute $\mathtt{C}$
- $\mathtt{T(R)}$=10,000
- $\mathtt{T(S)}$=5,000
- $\mathtt{S(R)}$=$\mathtt{S(S)}=\frac{1}{10}$ blocks (each block 10 tuples)
- Memory available = 101 blocks
- Metric: # of IOs (ignoring writing of result)

Caution

- This may not be the best way to compare
  - ignoring CPU costs
  - ignoring timing
  - ignoring double buffering requirements

Options

- Transformations: $\mathtt{R} \bowtie_C \mathtt{S}$, $\mathtt{S} \bowtie_C \mathtt{R}$
- Joint algorithms
  - Iteration (nested loops)
  - Merge join
  - Join with index
  - Hash join
- Factors that affect performance
  1. Tuples of relation stored physically together?
  2. Relations sorted by join attribute
  3. Indexes exist?

Nested-Loop Joins

- We now consider algorithms for the join operator
- The simplest one is the nested-loop join, a one-and-a-half pass algorithm.
- One table is read once, the other one multiple times.
- It is not necessary that one relation fits in main memory
- Perform the join through two nested loops over the two input relations.

Tuple-based Nested-loop Join

Algorithm 1: Tuple-based Nested-loop Join

```
1 for each tuple r ∈ R do
2     for each tuple s ∈ S do
3         if (r.C=s.C) then
4             output (r,s)
5         end
6     end
7 end
```

- Advantage:
  - Outer relation `R`, inner relation `S`.
  - Requires only 2 input buffers. 1 buffer to read tuples for relation `R` and 1 buffer to read tuples for relation `S`)
- Applicable to
  - Any join condition `C`
  - Cross-product

Example 1(a): Tuple-based Nested Loop Join R ⋈ S

- Relations not clustered
- Recall:
  - T(R)=10,000
  - T(S)=5,000
  - Memory available = 101 blocks
- R as the outer relation
- Cost for each R tuple r: [Read tuple r + Read relation S] = 1+5,000
- Total IO cost =10,000 × (1+5,000) = 5,010,000 IOs
- Can we do better?

Block-based nested loop join algorithm

- Can do much better by organizing access to both relations by blocks.
  - Use as much buffer space as possible `(M - 1)` to store tuples of the outer relation.
- Use `(M - 1)` buffers to read and index data blocks from the smaller relation `S`
- Use `1` buffer to read data blocks from the larger relation `R` and compute the Join result

Algorithm 2: Block-based Nested-loop Join

```
1  for each M-1 blocks of S do
2      Organize these tuples into a search structure (e.g., hash table)
3      for each block b of R do
4          Read b into main memory
5          for each tuple t ∈ block b do
6              - Find the tuples s1,s2,... of S (in the search structure) that
                 join with t
7              - Output (t,s1),(t,s2),...
8          end
9      end
10 end
```

Example 1(a): Block-based Nested Loop Join R ⋈ S

- Relations not contiguous
- Recall:
  - T(R)=10,000
  - T(S)=5,000
  - S(R)=S(S)=$\frac{1}{10}$ blocks
  - Memory available = 101 blocks
- R as the outer relation
- 100 buffers for R, 1 buffer for S
- cost for each R chunk:

  read chunk: 1,000 IOs

  read S: 5,000 IOs
- 10 R chunks
- Total I/O cost is $10 \times 6{,}000 = 60{,}000$ IOs

Can we do better?

- Reverse join order $\mathtt{S} \bowtie \mathtt{R}$
- Cost when using smaller relation $\mathtt{S}$ in the outer loop
- 100 buffers for $\mathtt{S}$, 1 buffer for $\mathtt{R}$
- cost for each $\mathtt{S}$ chunk:

  read chunk: 1,000 IOs

  read $\mathtt{S}$: 10,000 IOs
- 5 $\mathtt{S}$ chunks
- Total I/O cost is $5 \times 11{,}000 = 55{,}000$ IOs
- In general, there is a slight advantage to using the smaller relation in the outer loop.

Example 1(b): Block-based Nested Loop Join R ⋈ S

- Performance is dramatically improved when input relations are clustered (read by block).
- With clustered relations, for each S chunk:

  read chunk: 100 IOs

  read R: 1,000 IOs
- 5 S chunks
- Total I/O cost is $5 \times 1{,}100 = 5{,}500$ IOs

Two-Pass Algorithms Based on Sorting: Sort-merge Join

- If the input relations are sorted, the efficiency of duplicate elimination, set-theoretic operations and join can be greatly improved.
- In the following, we present a simple `sort-merge join` algorithm.
- It is called `merge-join`, if step/phase (1) can be skipped, since the input relations `R` and `S` are already sorted.

Two-Pass Algorithms Based on Sorting: Sort-merge Join

- Phase 1: perform a complete `TPMMS` on both relations and materialize the result on disk
  - Relation `R`: Sort relation `R` using the `TPMMS` algorithm
  - Relation `S`: Sort relation `S` using the `TPMMS` algorithm
- Phase 2: join the sorted relations:
  - Use `1` buffer to read relation `R`. The smallest join value may occupy more than `1` buffer
  - Use `1` buffer to read relation `S`. The smallest join value may occupy more than `1` buffer
  - If necessary (= when smallest join value may occupy more than `1` buffer):
    - Use the remaining `M - 2` buffers to store tuples in `R` and/or `S` that contain all smallest joining attribute values

Two-Pass Algorithms Based on Sorting: Sort-merge Join

Algorithm 3: Sort-merge join

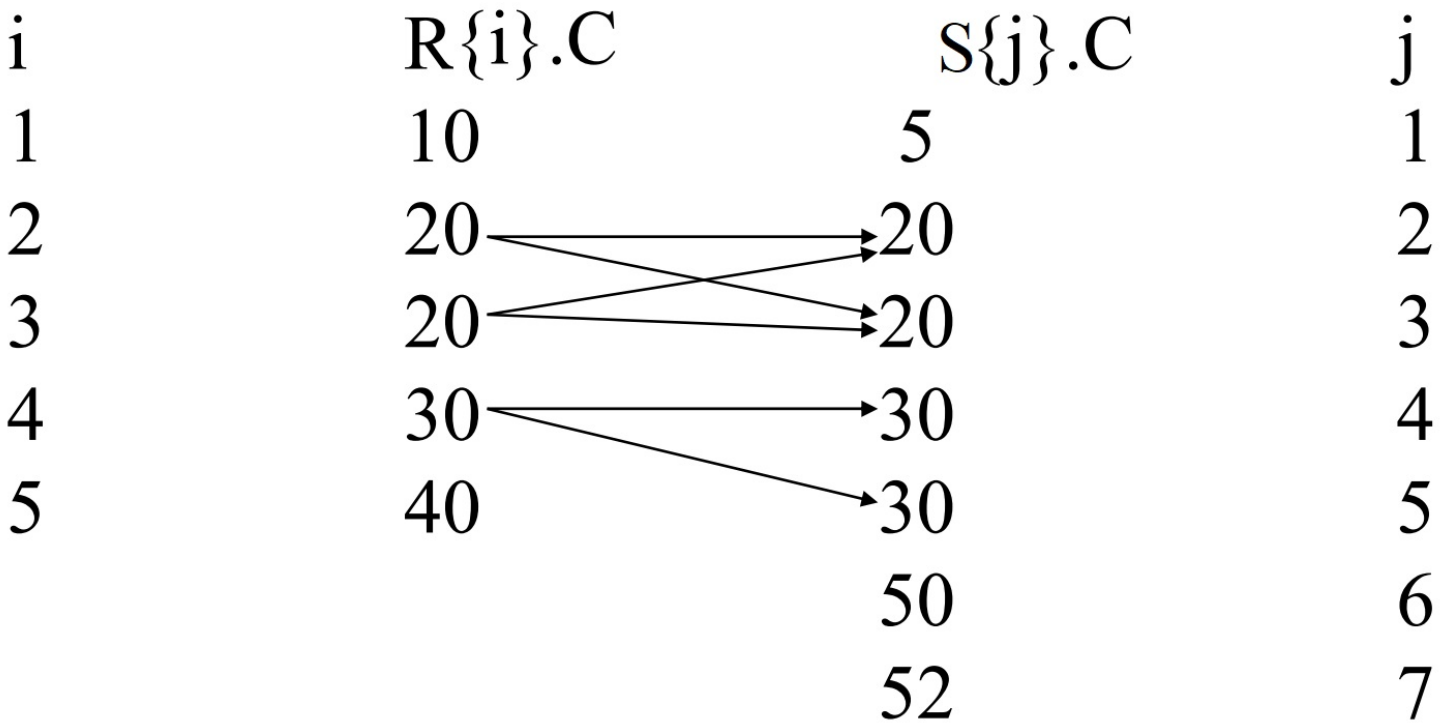
```
1  if R and S not sorted then
2  |  sort them
3  end
4  i ← 1; j ← 1;
5  while ((i ≤ T(R) ∧ j ≤ T(S)) do
6  |  if (R{i}.C = S{j}.C) then
7  |  |  outputTuples
8  |  else if (R{i}.C > S{j}.C) then
9  |  |  j ← j+1
10 |  else
11 |  |  i ← i+1
12 |  end
13 end
```

Algorithm 4: Output-Tuples

```
1 while (R{i}.C = S{j}.C) ∧ i≤ T(R))
   do
2 |  k← j;
3 |  while (R{i}.C = S{k}.C) ∧ k≤
  |   T(S)) do
4 |  |  output pair R{i}, S{k}
5 |  |  k← k + 1
6 |  end
7 |  i ← i+1
8 end
```

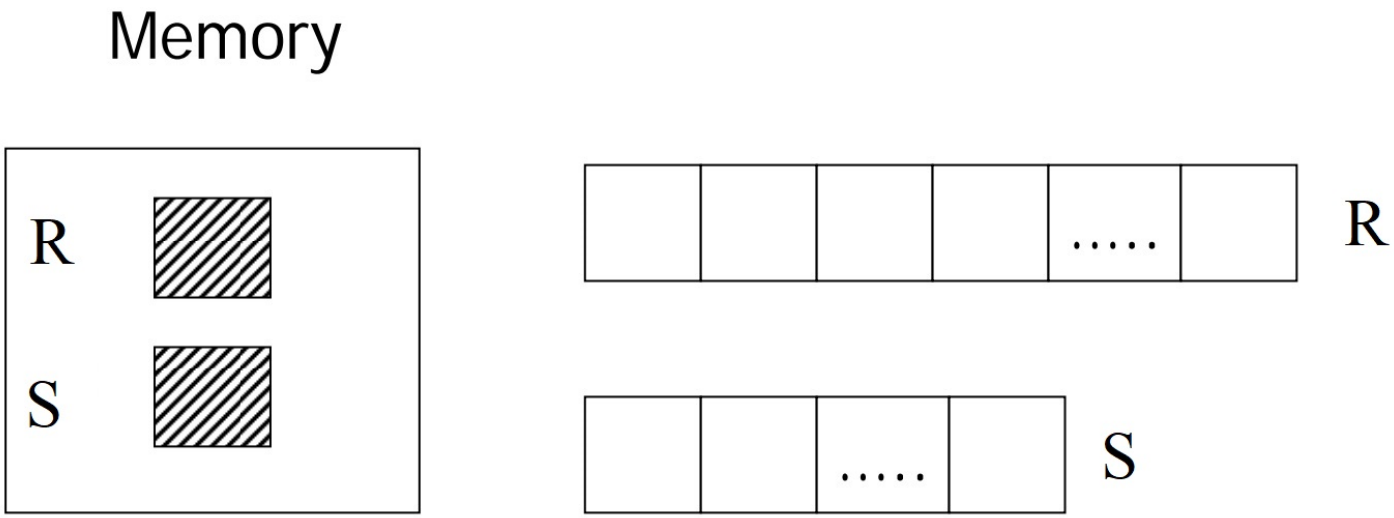
- Procedure `outputTuples` produces all pairs of tuples from `R` and `S` with `R{i}.C = S{j}.C`
- In the worst case, need to match each pairs of tuples from `R` and `S` (nested-loop join).

Sort-merge Join: Example



Example 1(c) Merge Join

- Both `R`, `S` ordered by `C`; relations contiguous



- `Total cost: Read R cost + read S cost`$= 1{,}000 + 500 =$ 1,500 IOs

Example 1(d) Merge Join

- R, S not ordered but contiguous
- Do Two-Phase Multiway Merge-Sort (TPMMS).
  - IO cost is $4 \times B(R)$, if sorting is used as a first step of sort-join and the results must be written to the disk.
  - If relation R is too big, apply the idea recursively.
    - Divide R into chunks of size M(M-1), use TPMMS to sort each one, and take resulting sorted lists as input for a third (merge) phase.
    - This leads to Multi-Phase Multiway Merge Sort.

Example 1(d) Merge Join

- R, S not ordered but contiguous
- Sort cost: each tuple is read, written, read, written
- Join cost: each tuple is read
- Sort cost R: $4 \times 1{,}000 = 4{,}000$
- Sort cost S: $4 \times 500 = 2{,}000$
- `Total cost = sort cost + join cost` $=6{,}000+1{,}500=7{,}500$ IOs
- `Total cost = 5(B(R) + B(S))`

Note

- `Nested loop join` (best version discussed above) needs only 5,500 IOs, i.e. outperforms `sort-join`.
- However, the situation changes for the following scenario:
  - `R` = 10,000 blocks
  - `S` = 5,000 blocks
  - Both `R`, `S` clustered, not ordered
  - `Nested-loops join`: $\frac{5{,}000}{100} \times (100 + 10{,}000) = 50 \times 10{,}100 = 505{,}000$ IOs
  - `Merge join`: $5 \times (10{,}000 + 5{,}000) = 75{,}000$ IOs
  - `Sort-join` clearly outperforms `nested-loop join`

Two-Pass Algorithms Based on Sorting: Sort-merge Join

- Simple sort-join costs $5\big(\texttt{B(R)} + \texttt{B(S)}\big)$ IOs.
- It requires $\texttt{M} \geq \sqrt{B(R)}$ and $\texttt{M} \geq \sqrt{B(S)}$
- It assumes that tuples with the same join attribute value fit in $\texttt{M}$ blocks.

Can we improve on merge join?

- Hint: do we really need the fully sorted files?
- If we do not have to worry about large numbers of tuples with the same join attribute value, then we can combine the second phase of the sort with the actual join (merge).
- We can save the writing to disk in the sort step and the reading in the merge step.

Advanced Sort-merge Join

- This algorithm is an `advanced sort-merge join`.
- Repeatedly find the least `C`-value `c` among the tuples in all input buffers.
- Instead of writing a sorted output buffer to disk, and reading it again later, identify all the tuples of both relations that have `C`=`c`.
- Cost is only $3\big($`B(R) + B(S)`$\big)$ IOs.
- Since we have to simultaneously sort both input tables and keep them in memory, the memory requirements are getting larger:

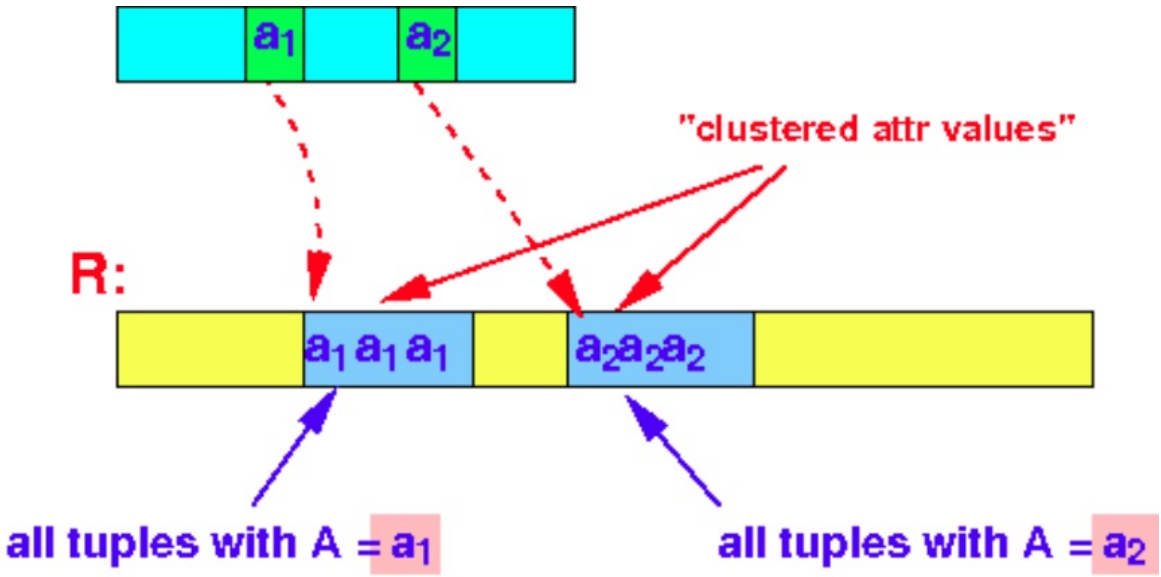
$$\texttt{M} \geq \sqrt{B(R) + B(S)}$$

Index-based algorithms

- Index-based algorithms are especially useful for the selection operator, but also for the join operator.
- We distinguish clustering and non-clustering indexes.
- A clustering index is an index where all tuples with a given search key value appear on (roughly) as few blocks as possible.
- One relation can have only one clustering index, but multiple non-clustering indexes.

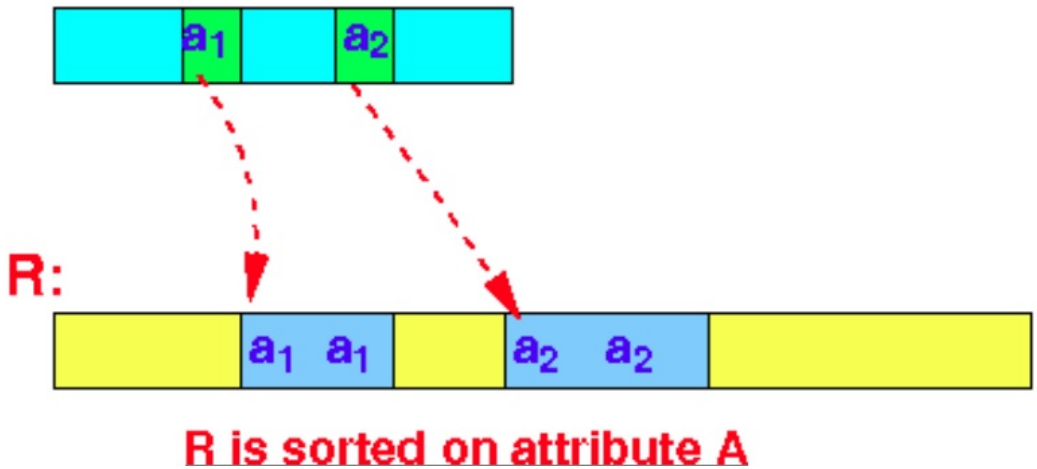
Index-based algorithms

- Clustering index
  - An index on attribute(s) `A` on a file is a clustering index when all tuples with attribute value `A = a` are stored sequentially (= consecutively) in the data file



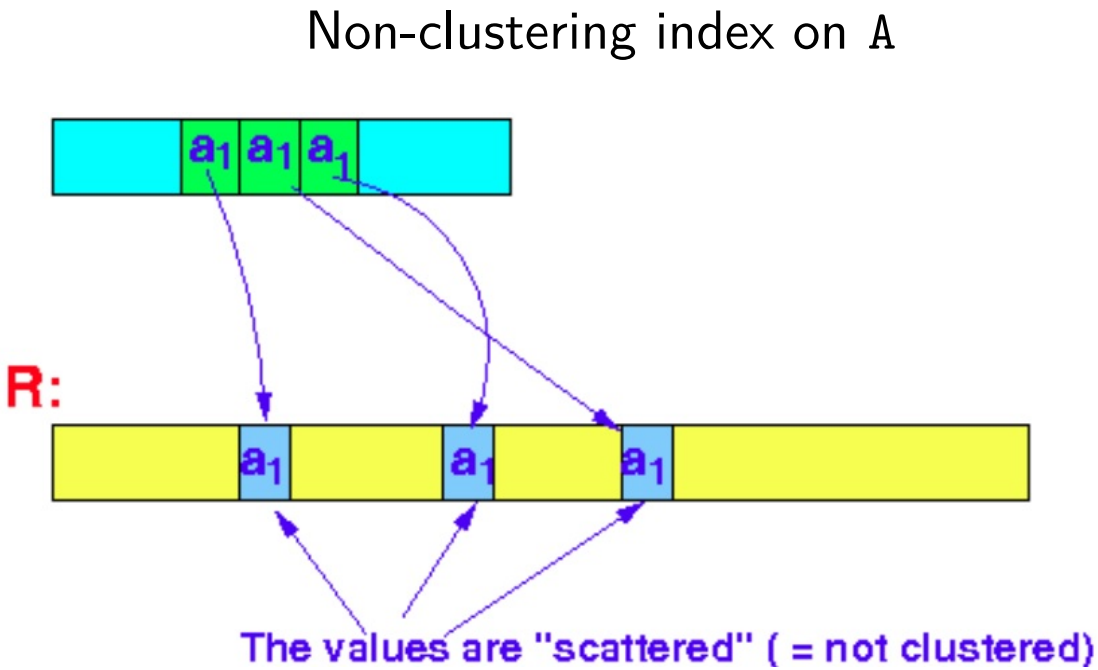
Index-based algorithms

- Common example of a clustering index when the relation R is sorted on the attribute(s) A



Index-based algorithms

- Non-clustering index
  - an index that is not clustering



Join algorithm for a clustering index

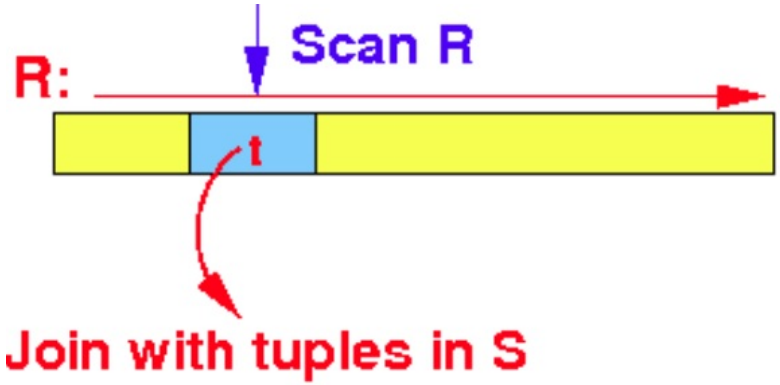
- Assume `S.C` index

Algorithm 5: Join using index S.C

```
1 Read a block of R in b
2 for each tuple t_r ∈ b do
3 |   Use t_r.C to lookup in index S.C
4 |   // You get a list of record addresses
5 |   for each record address s do
6 |   |   read tuple at s
7 |   |   output t_r, t_s pair // Join result
8 |   end
9 end
```

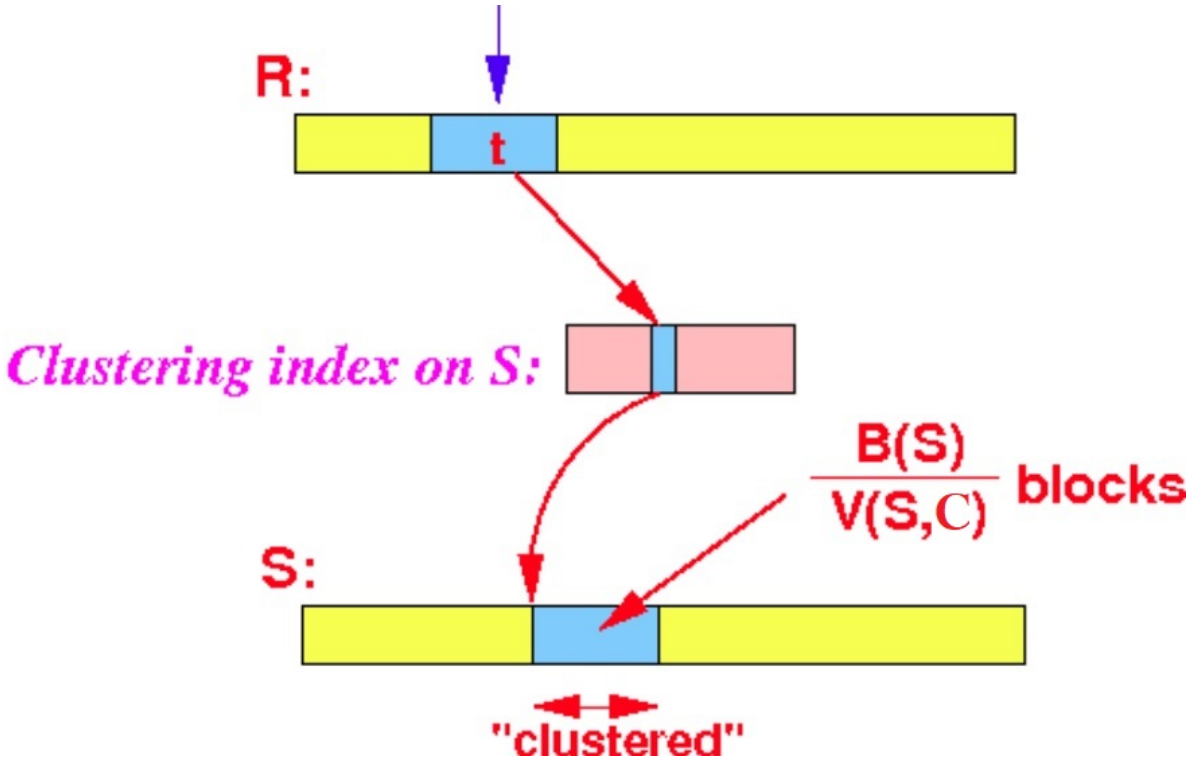
Join algorithm for a clustering index: Performance

- The join algorithm will scan the relation R once



Join algorithm for a clustering index: Performance

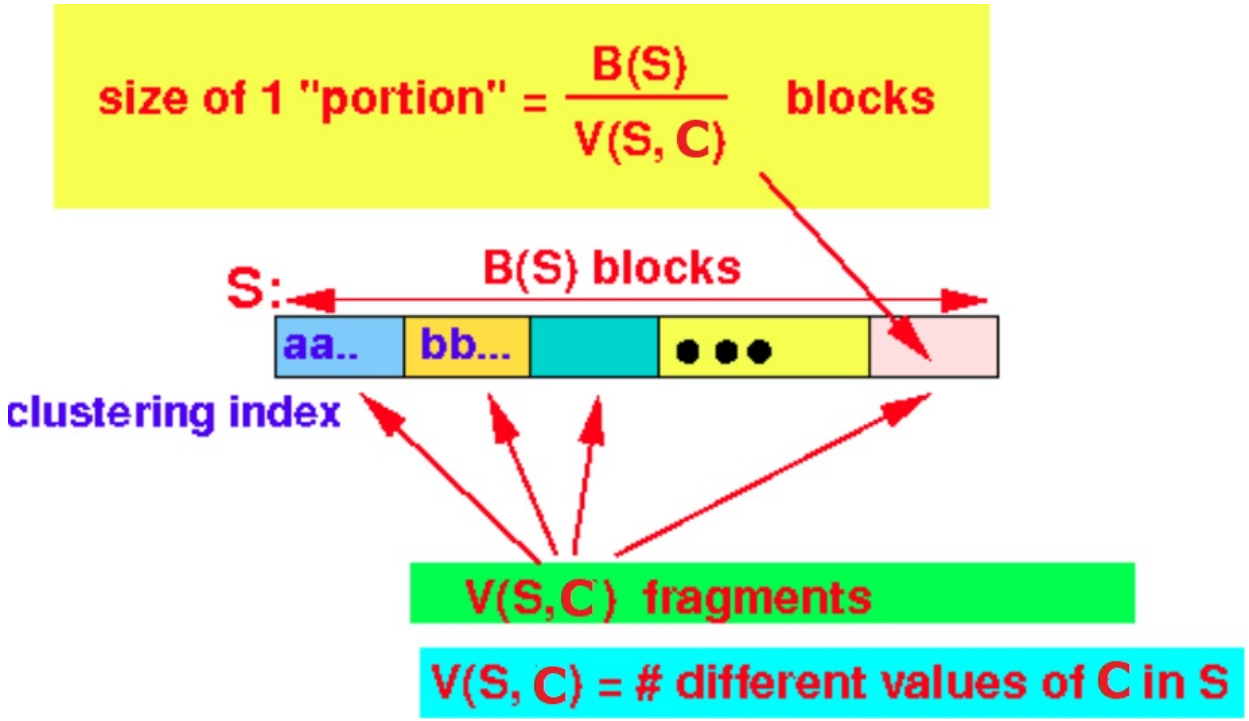
- For each tuple `t` $\in$ `R`, we read the following portion in relation `S`



  - For each tuple `t` $\in$ `R`, we read (= access) this portion in `S`
    - `portion of S read per tuple in R =` $\frac{B(S)}{V(S,C)}$ blocks

Join algorithm for a clustering index: Performance

- because

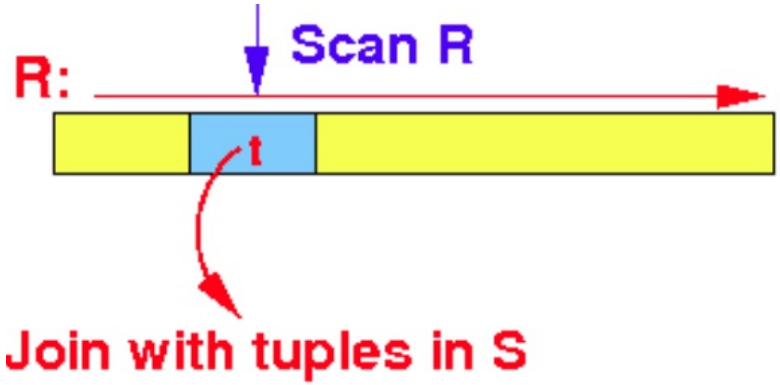


Join algorithm for a clustering index: Performance

- Therefore: # disk IO used in join algorithm with an clustering index
  - `# disk IOs = Scan R once + # tuples in R` $\times$ `# blocks of S read per tuple of R`
  - if relation `R` is clustered:
    - `# disk IOs = B(R) + T(R)` $\times \frac{B(S)}{V(S,C)}$
    - Approximate: `# disk IOs` $\cong$ `T(R)` $\times \frac{B(S)}{V(S,C)}$
  - if relation `R` is not-clustered:
    - `# disk IOs = T(R) + T(R)` $\times \frac{B(S)}{V(S,C)}$
    - Approximate: `# disk IOs` $\cong$ `T(R)` $\times \frac{B(S)}{V(S,C)}$

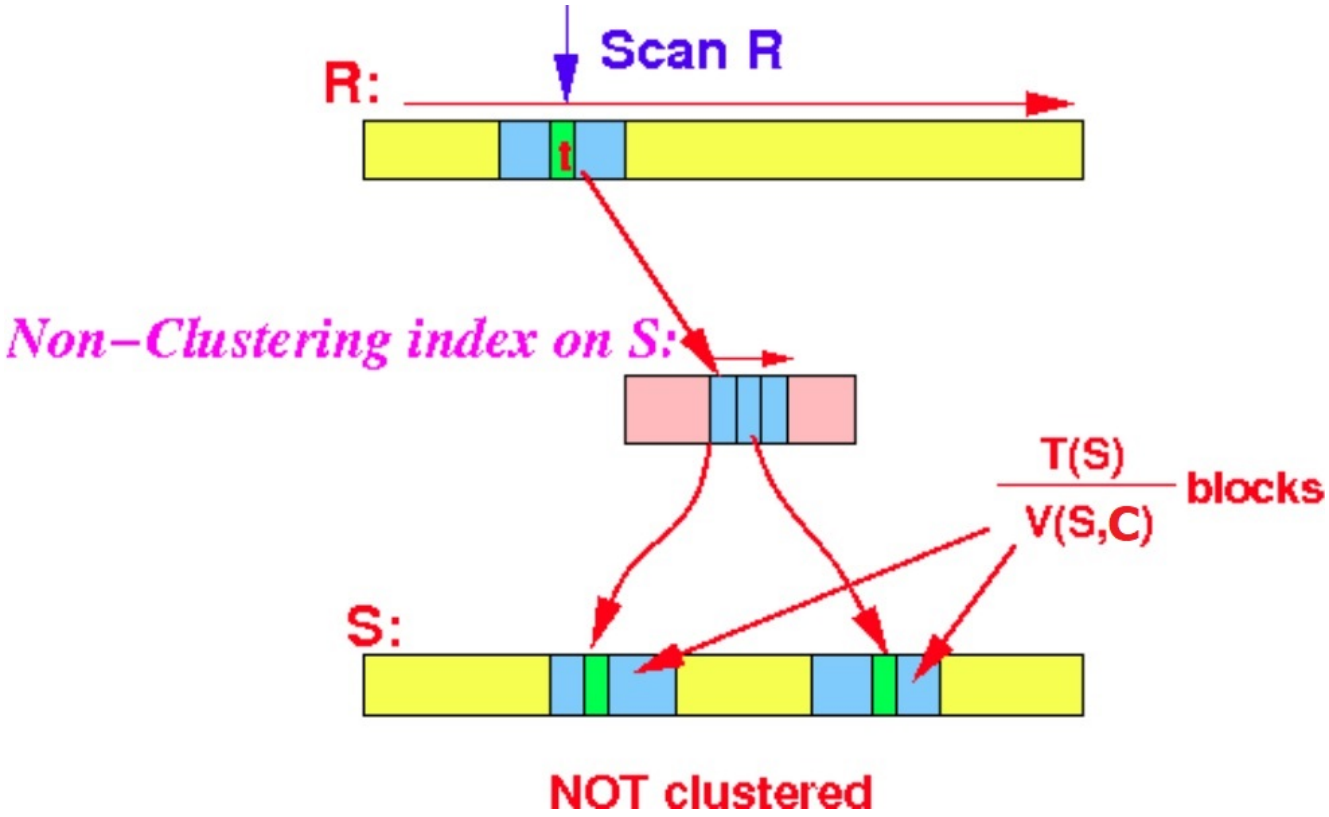
Join algorithm for non-clustering index: Performance

- The join algorithm will scan the relation R once



Join algorithm for non-clustering index: Performance

- For each tuple `t` $\in$ `R`, we read the following portion in relation `S`



  - For each tuple `t` $\in$ `R`, we read (= access) this portion in `S`
    - `portion of S read per tuple in R` = $\frac{T(S)}{V(S,C)}$ blocks
    - We assumed `1` tuple of `S` in each block read

Join algorithm for non-clustering index: Performance

- # disk IO used in join algorithm with non-clustering index
  - `# disk IOs = Scan R once + # tuples in R` $\times$ `# blocks of S read per tuple of R`
  - if relation `R` is clustered:
    - `# disk IOs = B(R) + T(R)` $\times \frac{T(S)}{V(S,C)}$
    - Approximate: `# disk IOs` $\cong$ `T(R)` $\times \frac{T(S)}{V(S,C)}$
  - if relation `R` is not-clustered:
    - `# disk IOs = T(R) + T(R)` $\times \frac{T(S)}{V(S,C)}$
    - Approximate: `# disk IOs` $\cong$ `T(R)` $\times \frac{T(S)}{V(S,C)}$

When to use the Index-Join algorithm

- One of the relations in the join is very small and
- The other (large) relation has an index on the join attribute(s)

Example 1(e) Index Join

- Assume `R.C` non-clustering index exists, 2 levels
- Assume `S` contiguous, unordered
- Assume `R.C` index fits in memory
- `Cost: # disk IOs = B(S) + T(S)` $\times \frac{T(R)}{V(R,C)}$

Example 1(e) Index Join

- Cost
  - reads of `S`: 500 IOs
  - for each `S` tuple:
    - probe index: no IO
    - if match, read `R` tuple: 1 IO

Example 1(e) Index Join

- What is expected number of matching tuples?
  a. say `R.C` is key, `S.C` is foreign key then expect 1 match
  b. say `V(R,C)` $= 5000$, `T(R)` $= 10{,}000$ with uniform distribution assumption expect $\frac{T(R)}{V(R,C)} = \frac{10{,}000}{5{,}000} = 2$ matching tuples
- Total cost of index join
  a. `Total cost = B(S) + T(S)` $\times$ `1 = 500 + 5000` $\times$ `1 = 5,500` IO
  b. `Total cost = B(S) + T(S)` $\times \frac{T(R)}{V(R,C)}$ `= 500 + 5000` $\times$ `2 = 10,500` IO
- Will any of these change if we have a clustering index?
- What if index does not fit in memory?

Hash Join: One-pass Algorithm for R ⋈ S

- Assumption
  - The relation S is the smaller relation
  - Building a search structure using S will minimize the memory requirement
- Phase 1
  - Use 1 buffer and scan the SMALLER relation first.
  - Build a search structure H on the SMALLER relation to help speed up finding common elements.
- Phase 2
  - Output only those tuples in R that have join attributes equal to some tuple in S
  - We use the search structure H to implement the test t(join attrs) $\in$ H efficiently
  - For H, we can use hash table or some binary search tree

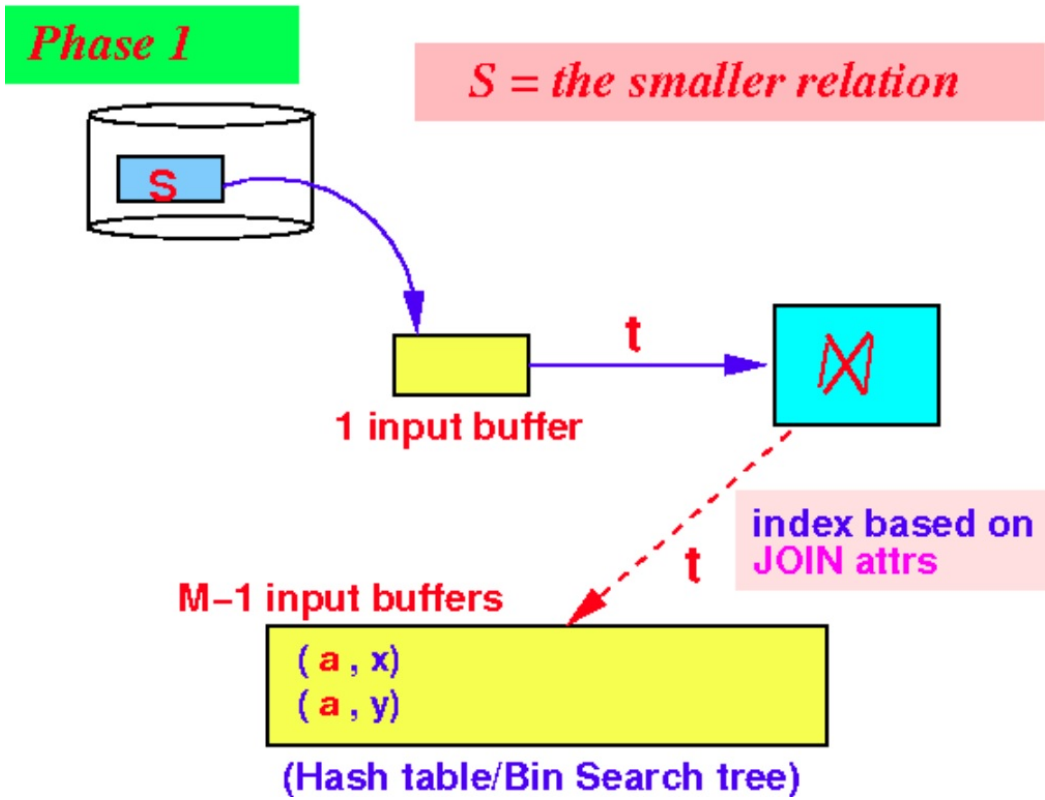
Hash Join: One-pass Algorithm for R ⋈ S

Algorithm 6: One-pass Hash ⋈

```
1  while (S has more data blocks) do
2  |   Read 1 data block in buffer b
3  |   for (each tuple t ∈ b) do
4  |   |   Insert t in H // Build search structure (hash table or search tree)
5  |   end
6  end
7  while (R has more data blocks) do
8  |   Read 1 data block in buffer b
9  |   for (each tuple t ∈ b) do
10 |   |   if (t(join attrs) ∈ H) then
11 |   |   |   Let s = the tuple in H with t(join attrs)
12 |   |   |   Output t, s // successful join
13 |   |   end
14 |   end
15 end
```

Hash Join: One-pass Algorithm for R ⋈ S

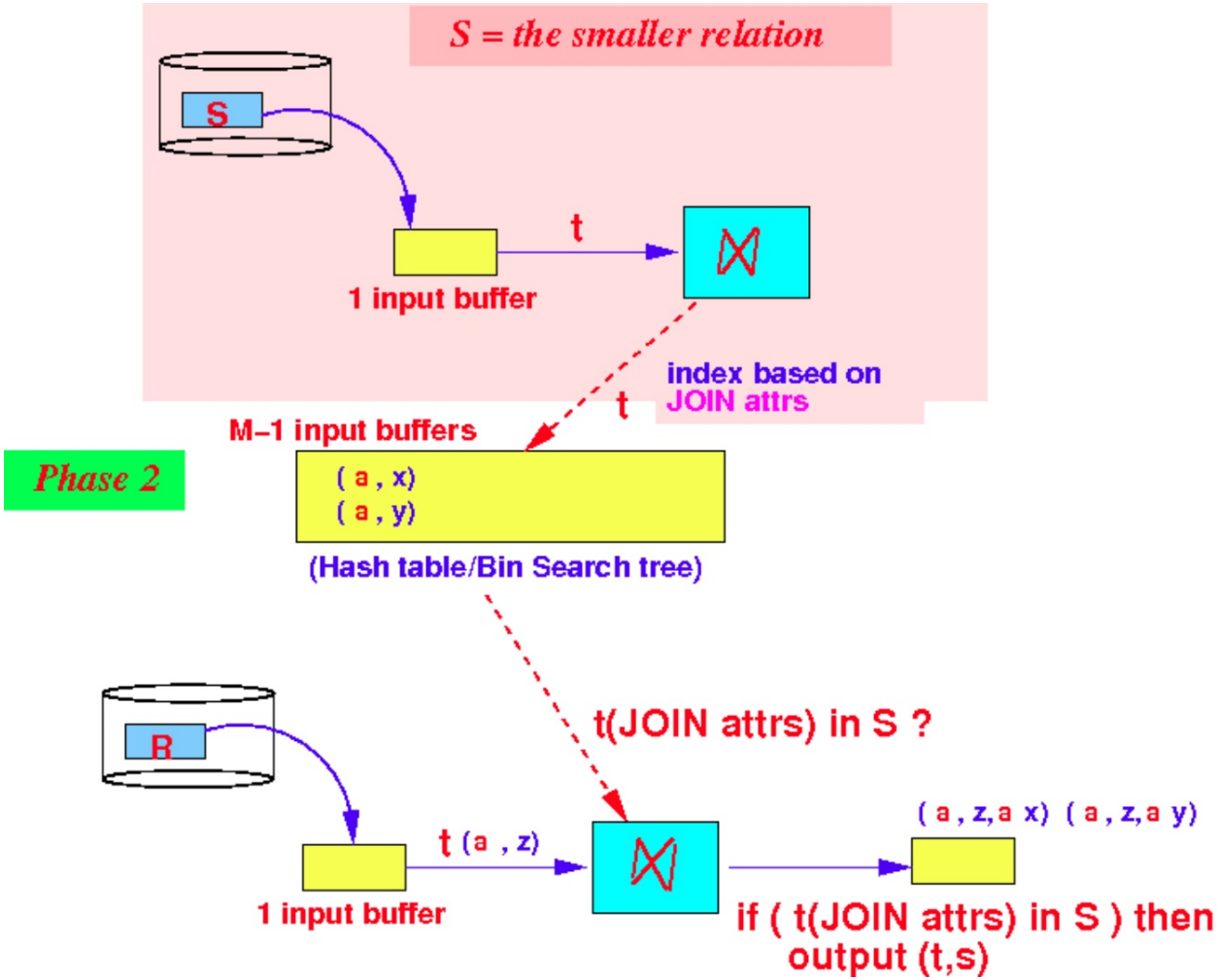
- Buffer utilization when there are M buffers available
  - Phase 1: partition the M buffers as follows



  - Use 1 buffer for input from S
  - Use M−1 buffers for the search structure

Buffer utilization

- Phase 2: partition the M buffers as follows



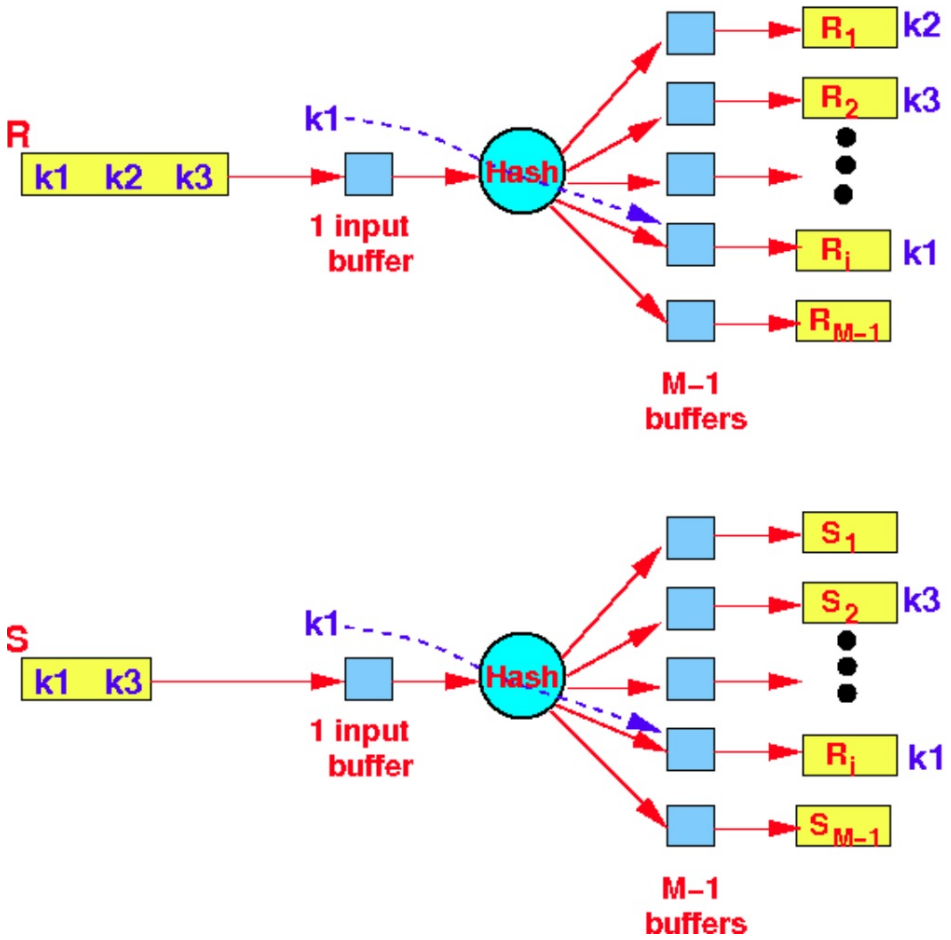
- Use 1 buffer for input from R
- Still using M−1 buffers for the search structure in Phase 2

Hash Join: One-pass Algorithm for R ⋈ S: Cost

- # disk I/O used
  - `B(R) + B(S)`, if the relations `R` and `S` are clustered
- Memory requirement
  - `M` $\geq$ `B(S) + 1 buffer`
  - Relation `S` is the smaller relation in the intersection

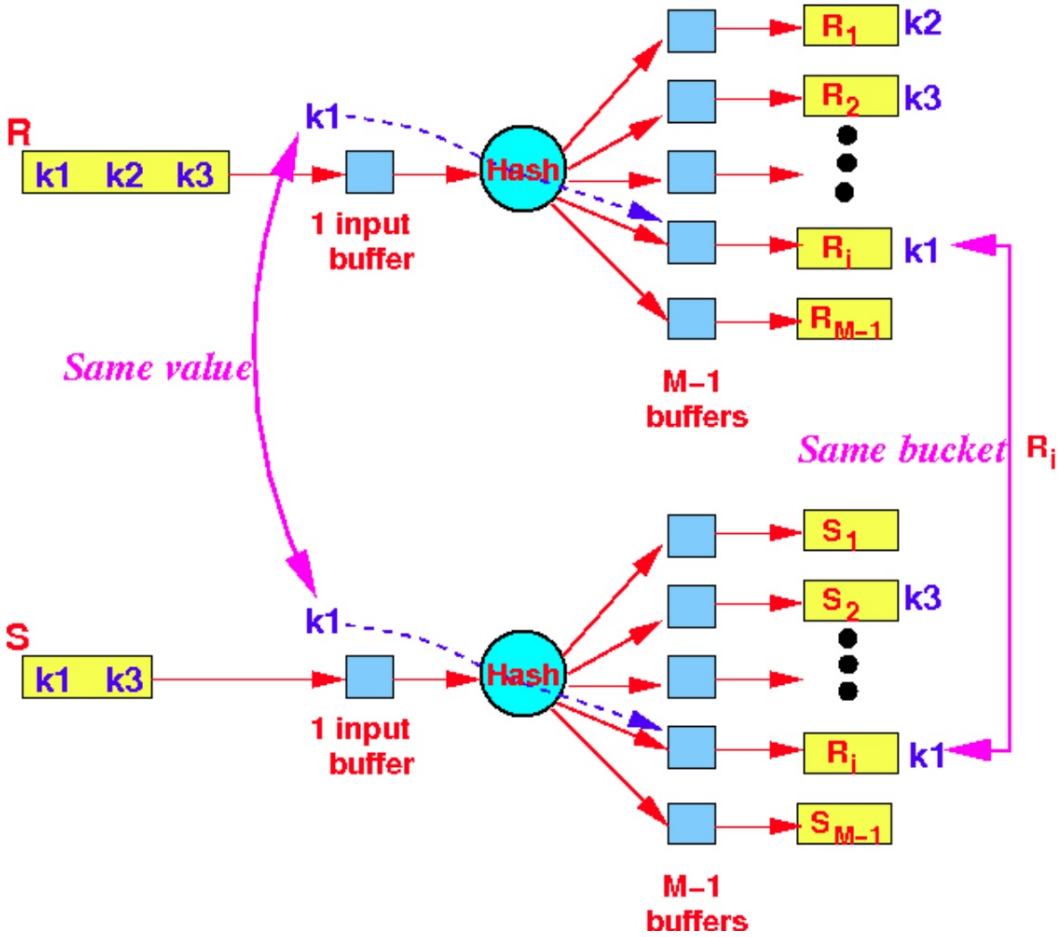
Two-pass algorithm for Join based on hashing

- Phase 1: hash both relations R and S using the same hash function



Two-pass algorithm for Join based on hashing

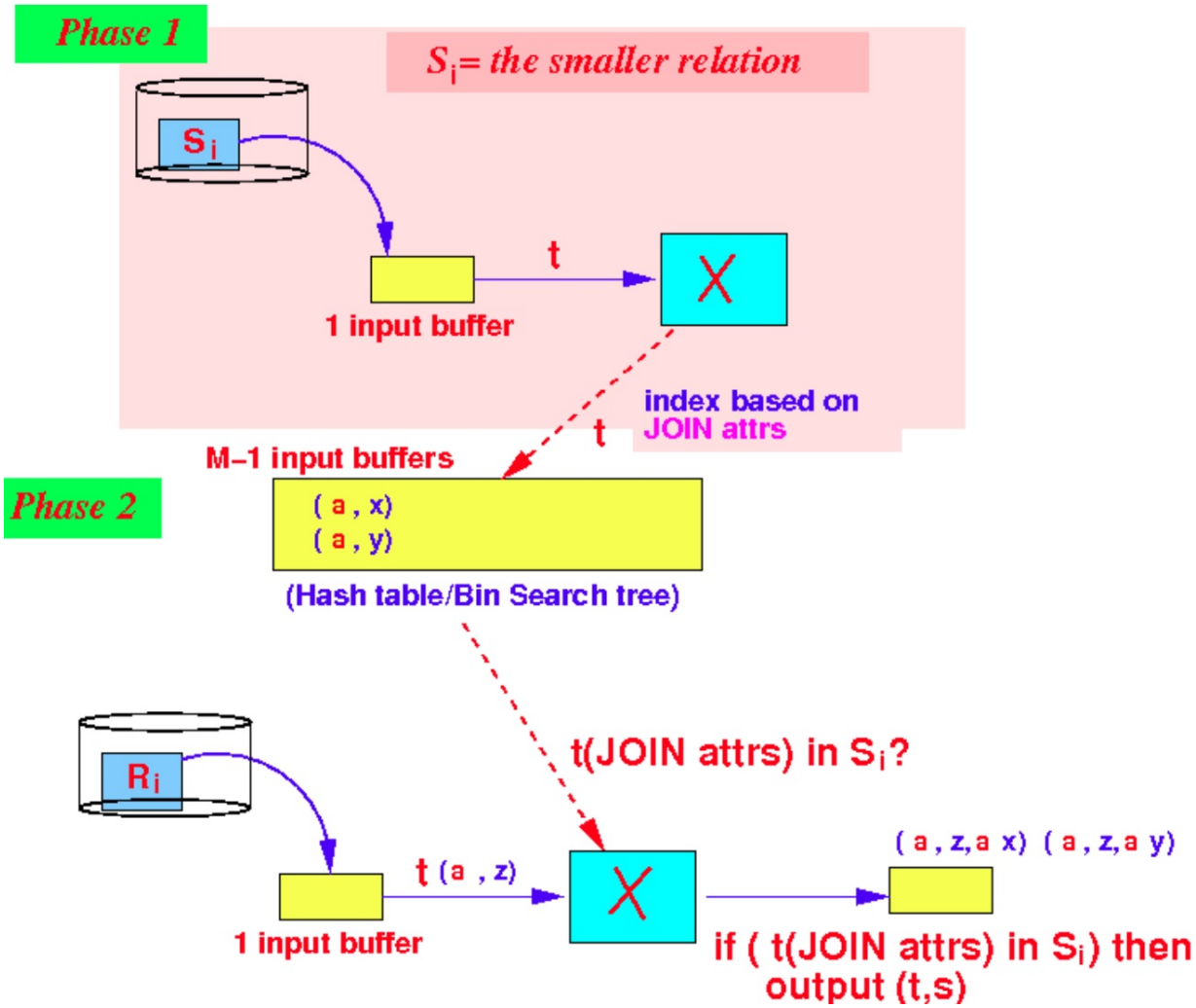
- Notice that: Same attribute value in R and S will be hashed into the same bucket



- Therefore, we can process the (equality) joining tuples in the $\bowtie$ operation by examining the corresponding pair of buckets alone

Two-pass algorithm for Join based on hashing

- Phase 2: process each pair of sub-relation R_i and S_i using the `one-pass` $\bowtie$ `algorithm`



Two-pass algorithm for Join based on hashing: Cost

- Cost incurred during Phase 1: (Hashing relations `R` and `S`)
  - Read `R`: `B(R)`
  - Write `M-1` buckets (total): `B(R)`
  - Read `S`: `B(S)`
  - Write `M-1` buckets (total): `B(S)`
- Cost incurred during Phase 2: (run `one-pass algorithm`)
  - `B(R) + B(S)`
- Total cost of the `R` $\bowtie$ `S` operation

$$\texttt{Cost of (R} \bowtie \texttt{R) = 3} \times \texttt{B(R) + 3} \times \texttt{B(S)}$$

Two-pass algorithm for Join based on hashing: Memory constraint

- The `one-pass algorithm` used in Phase 2 has this memory constraint
  - All tuples of the smaller sub-relation S_i must fit in the search structure using `M - 1` buffers
  - $\therefore \min(B(R_i), B(S_i)) \leq M-1$ blocks
  - The best case scenario of the hashing is
    - Every sub-relation has the same size

      $B(R_i) = \frac{1}{M-1} \times B(R)$

      $B(S_i) = \frac{1}{M-1} \times B(S)$

    $\Rightarrow (M - 1)^2 \geq \min(B(R), B(S))$

    $\iff M - 1 \geq \sqrt{\min(B(R), B(S))}$
  - Memory constraint:

    $$M \geq \sqrt{\min(B(R), B(S))} + 1$$

Example 1(f) Hash Join

- R, S contiguous (un-ordered)
- Total cost $= 3 \times (1000+500) = 4500$

EXPLAIN Statements

- How to know which plan/access paths the database chose for evaluating a query?
  - Use the so called EXPLAIN-statement
  - EXPLAIN analyzes the execution of a given SQL query and stores it's results into explain tables within the DB
    - Shows operation execution order
    - Collects metrics for each basic operation

Summary

- `Nested-loop` ok for "small" relations (relative to memory size)
- `Hash-join` usually is best for `equi-join` (join condition is equal), where relations not sorted and no indexes exist
- `Sort-merge` join is good for `non-equi-join` e.g., `R.C` $>$ `S.C`
- If relations already sorted, use `merge join`
- If index exists, `index-join` can be efficient (depends on expected result size)
- Watch: **27:21**: Postgres Open 2016 - Identifying Slow Queries and Fixing Them!