CSE462/562: Database Systems (Spring 24)

Lecture 10: Single-table query processing (Part 2)

4/8/2024

University at Buffalo
Department of Computer Science and Engineering
School of Engineering and Applied Sciences

Reminders

- HW4 released today, due on 4/22/2024, 23:59 PM EDT
- Project 4 due next Monday, 4/15/2024, 23:59 PM EDT

Recap on Single-Table QP

A few basic operators

- Selection: σ
- Projection: π (w/ and w/o deduplication)
- Aggregation: γ w/o or w/ group by
- Set operators: $\cup, -, \cap$
- Sorting
- Cartesian product: $\times$ or Join: $\bowtie$

```
SELECT E
FROM R
WHERE P
ORDER BY S
```

$\Rightarrow$ $Sort_S(\pi_E \sigma_P R)$

```
SELECT G, SUM(E)
FROM R
WHERE P
GROUP BY G
HAVING P'
ORDER BY S
```

$\Rightarrow$ $Sort_S\left(\sigma_{P'}\ {}_G\gamma_{SUM(E)} \sigma_P R\right)$

Measuring cost

- We'll start with the simplest single-table queries w/o or w/ aggregations
  - How to translate it into a query plan?
  - How to implement each operator?
  - How to measure the cost of each operator?
- For disk-based systems, we mainly measure the number of I/Os
  - Differences between random I/O and sequential I/O
  - Faster storage -> also need to measure the CPU cost
- A simple cost model
  - t_T: average time to transfer a page of data (data transfer time)
  - t_S: average time to randomly seek data (seek time + rotation delay)
    - For SSD, time overhead for initiating an I/O request
  - Cost = $B \times t_T + S \times t_S$
    - B: number of pages read/written; S: number of random I/O

Typical t_T and T_S

	HDD*	SSD†
t_T (ms)	0.1	0.01
t_S (ms)	4	0.09

Data from DB Concept book (Ch. 15.2).
Assuming 4KB pages.
* typical HDD with 40 MB/s transfer rate, 15000 rpm disk in 2018
† typical SATA SSD that supports 10K IOPS (QD-1), 400 MB/s sequential read rate

Measuring cost

- Other assumptions
  - Ignoring the buffer effect for random pages
    - Do consider the private workspace size M for the operators
  - Omitting the cost of transferring output to the user/disk
    - Common to any equivalent plan

- Notations: for relation R
  - T_R: number of records, N_R: number of pages in its heap file, B_R: (average) number of tuples per page
  - h_I: height of a B-tree index I over the file
  - M: private workspace size in pages
- Running example
  - $t_S = 4\ ms$, $t_T = 0.1\ ms$, *4000-byte page*
  - Student: R(sid: int, name: varchar(19), login: varchar(19), major: char(2), adm_year: int)
    - 50 bytes/tuple, $B_R = 80$, $T_R = 40{,}000$, $N_R = 500$
  - Enrollment: E(<u>sid: int, semester: char(3), cno: int</u>, grade: double)
    - 20 bytes/tuple, $B_E = 200$, $T_E = 200{,}000$, $N_E = 1000$

Aggregation γ without grouping

F is an aggregation function, e.g., $SUM, COUNT, VAR, STDDEV, AVG, MIN, MAX$ or UDA etc.

- $\gamma_{F_1(E_1), F_2(E_2), \ldots, F_k(E_k)} Q$
  - Blocking
  - Only produce one row of output

  - An aggregation can be expressed as three functions: $F = \left(F^{init}, F^{acc}, F^{final}\right)$
    - Initialization $F^{init}: void \rightarrow A$ (where A is some internal state of the aggregation)
    - Accumulation $F^{acc}: (A, T) \rightarrow A$ or $(A, T) \rightarrow void$
    - Finalization $F^{final}: A \rightarrow V$ (where V is the final type of the aggregation)
    - Some systems also have an optional combine function $F^{combine}: (A, A) \rightarrow A$
      - allows parallelizing the aggregation

  - Example: AVG of integers
    - AVG^{init} ( ): create a pair of (s, c) -- s: sum of values, c: number of values
    - $AVG^{acc}\left((s, c), x\right) = (s + x, c + 1)$
    - $AVG^{final}\left((s, c)\right) = 1.0 * s / c$
  - Cost: does not add additional I/O cost

Aggregation γ without grouping

F is an aggregation function, e.g., $SUM, COUNT, VAR, STDDEV, AVG, MIN, MAX$ or UDA etc.

- Example: AVG of integers
  - AVG^{init} ( ): create a pair of (s, c) -- s: sum of values, c: number of values
  - $AVG^{acc}\big((s, c), x\big) = (s + x, c)$
  - $AVG^{final}\big((s, c)\big) = 1.0 \; * s \,/\, c$

  - Consider a column in a table with the following values
    - $5, 4, 1, 3, 2$
    - Steps:
      - AVG^{init} ( ) = (0.0, 0)
      - $AVG^{acc}\big((0.0, 0), 5\big) = (5.0, 1)$
      - $AVG^{acc}\big((5.0, 1), 4\big) = (9.0, 2)$
      - $AVG^{acc}\big((9.0, 2), 1\big) = (10.0, 3)$
      - $AVG^{acc}\big((10.0, 3), 3\big) = (13.0,4)$
      - $AVG^{acc}\big((13.0, 4), 2\big) = (15.0, 5)$
      - $AVG^{final}\big((15.0, 5)\big) = 3.0 = \frac{5+4+1+3+2}{5}$

Aggregation γ with grouping

- $_{G_1,G_2,\ldots,G_n}\gamma_{F_1(E_1),F_2(E_2),\ldots,F_k(E_k)}Q$
  - Blocking
  - One record per group (distinct values in $G_1, G_2, \ldots, G_n$)
    - Let group by columns be $\mathcal{G} = (G_1, G_2, \ldots, G_n)$
  - Solution: sorting or hashing

Aggregation γ with grouping

- ${}_{G_1,G_2,\ldots,G_n}\gamma_{F_1(E_1),F_2(E_2),\ldots,F_k(E_k)}Q$
  - Blocking
  - One record per group (distinct values in $G_1, G_2, \ldots, G_n$)
    - Let group by columns be $\mathcal{G} = (G_1, G_2, \ldots, G_n)$
  - Sort-based solution: sort all tuples in Q on $\mathcal{G}$; for each result t
    1. If t is the first one, $g \leftarrow \pi_{\mathcal{G}} t$ and $a_1 \leftarrow F_1^{init}(\,)$, … $a_k \leftarrow F_k^{init}(\,)$
    2. If t is not the first and $\pi_{\mathcal{G}} t \neq g$, emit $g \circ \left(F_1^{final}(a_1), \ldots F_k^{final}(a_k)\right)$
       - Then, $g \leftarrow \pi_{\mathcal{G}} t$ and $a_1 \leftarrow F_1^{init}(\,)$, … $a_k \leftarrow F_k^{init}(\,)$
    3. In both cases, $a_1 \leftarrow F_1^{acc}(a_1, \pi_{E_1} t\,)$, … $a_k \leftarrow F_k^{acc}(a_k, \pi_{E_k} t\,)$
    4. After the last record is read, emit the last group as $g \circ \left(F_1^{final}(a_1), \ldots F_k^{final}(a_k)\right)$
  - If there are too many groups, use external sorting
    - Optimization opportunities (next lecture)

Aggregation γ with grouping

- Example for sort-based solution:
  - Consider two columns (x, y) with the following values
    - $(1, 1.0), (2, 2.0), (1, 4.0), (2, 6.0)$
    - ${}_x\gamma_{SUM(y)}$
    - Step 1: sort by x
      - $(1, 1.0), (1, 4.0), (2, 2.0), (2, 6.0)$
    - Step 2: scan and calculate the group aggregates
      - Scan (1, 1.0): $g \leftarrow x = 1$, $a_1 \leftarrow 0.0 + 1.0 = 1.0$
      - Scan (1, 4.0): $a_1 \leftarrow a_1 + 4.0 = 5$
      - Scan (2, 2.0):
        - Since $x = 2 \neq g = 1$, emit $(g, a_1) = (1, 5.0)$ as a result
        - $g \leftarrow x = 2$, $a_1 \leftarrow 0.0 + 2.0 = 2.0$
      - Scan (2, 6.0): $a_1 \leftarrow a_1 + 6.0 = 8.0$
    - Step 3: emit the final group: $(g, a_1) = (2, 8.0)$

Result

x	SUM(y)
1	5.0
2	8.0

Aggregation γ with grouping

- ${}_{G_1,G_2,\ldots,G_n}\gamma_{F_1(E_1),F_2(E_2),\ldots,F_k(E_k)}Q$
  - Blocking
  - One record per group (distinct values in $G_1, G_2, \ldots, G_n$)
    - Let group by columns be $\mathcal{G} = (G_1, G_2, \ldots, G_n)$ or $\bigcup_{1\le i\le n} Var(G_i)$
  - Hash-based solution: create a hash table from $\mathcal{G}$ to $(A_1, A_2, \ldots, A_k)$
    - Maintain the hash table using the aggregation functions while reading records from Q
    - After deplete the records in Q, scan the hash table, and
    - emit one row for each distinct value in $\mathcal{G}$ and compute its final value using the finalization functions

  - Again, if there are too many groups, use external hashing
    - Optimization opportunities (next lecture)

Aggregation γ with grouping

- Example for hash-based solution:
  - Consider two columns (x, y) with the following values
    - $(1, 1.0), (2, 2.0), (1, 4.0), (2, 6.0)$
      - assume $h(1) = 2, h(2) = 0$
    - ${}_{x}\gamma_{SUM(y)}$
    - Step 1: create an empty hash table
    - Step 2: scan records and maintain aggregates
      - scan (1, 1.0): $x[h(1)] \leftarrow x = 1$, $a_1[h(1)] \leftarrow 0.0 + y = 1.0$
      - scan (2, 2.0): $x[h(2)] \leftarrow x = 2$, $a_1[h(2)] \leftarrow 0.0 + y = 2.0$

hash table

h(x)	0	1	2	3
x	2		1	
a_1	2.0		1.0	

Aggregation γ with grouping

- Example for hash-based solution:
  - Consider two columns (x, y) with the following values
    - $(1, 1.0), (2, 2.0), (1, 4.0), (2, 6.0)$
      - assume $h(1) = 2, h(2) = 0$
    - ${}_{x}\gamma_{SUM(y)}$
    - Step 1: create an empty hash table
    - Step 2: scan records and maintain aggregates
      - scan (1, 1.0): $x[h(1)] \leftarrow x = 1$, $a_1[h(1)] \leftarrow 0.0 + y = 1.0$
      - scan (2, 2.0): $x[h(2)] \leftarrow x = 2$, $a_1[h(2)] \leftarrow 0.0 + y = 2.0$
      - scan (1, 4.0): $a_1[h(1)] \leftarrow a_1[h(1)] + y = 1.0 + 4.0 = 5.0$
      - scan (2, 6.0): $a_1[h(2)] \leftarrow a_1[h(2)] + y = 2.0 + 6.0 = 8.0$
    - Step 3: scan hash table and emit results

hash table

h(x)	**0**	**1**	**2**	**3**
x	2		1	
a_1	8.0		5.0	

Result

x	SUM(y)
1	5.0
2	8.0

Set operators ∪,∩ , —

- SQL performs deduplication before the set operators by default, unless one specifies ALL
  - e.g., A = {1, 1, 2}, B = {1, 2}
    - `SELECT * FROM A EXCEPT SELECT * FROM B;` -- result is empty
    - `SELECT * FROM A EXCEPT ALL SELECT * FROM B;` -- result is {1} (one row)
  - UNION ALL can be made pipelining: emit everything from LHS and then RHS

  - All the others are similar: using UNION as an example
    - Solution: sorting or hashing
    - sorting: sort A and B separately, merge them together by removing any duplicates
      - Similar to a sort-merge join we will discuss in later lectures
    - hashing: create a hash table over all the attributes, scan A and B
      - Only keep the first occurrence of each distinct value
  - Once again, optimization opportunities exist when the result set(s) of A and/or B do not fit in memory

Sort operator

- Use cases
  - ORDER BY
  - For Sort-Merge Join (next lecture)
  - For bulk loading tree indexes
  - …
- If data fit in memory -- easy
  - quick sort
  - merge sort
  - …

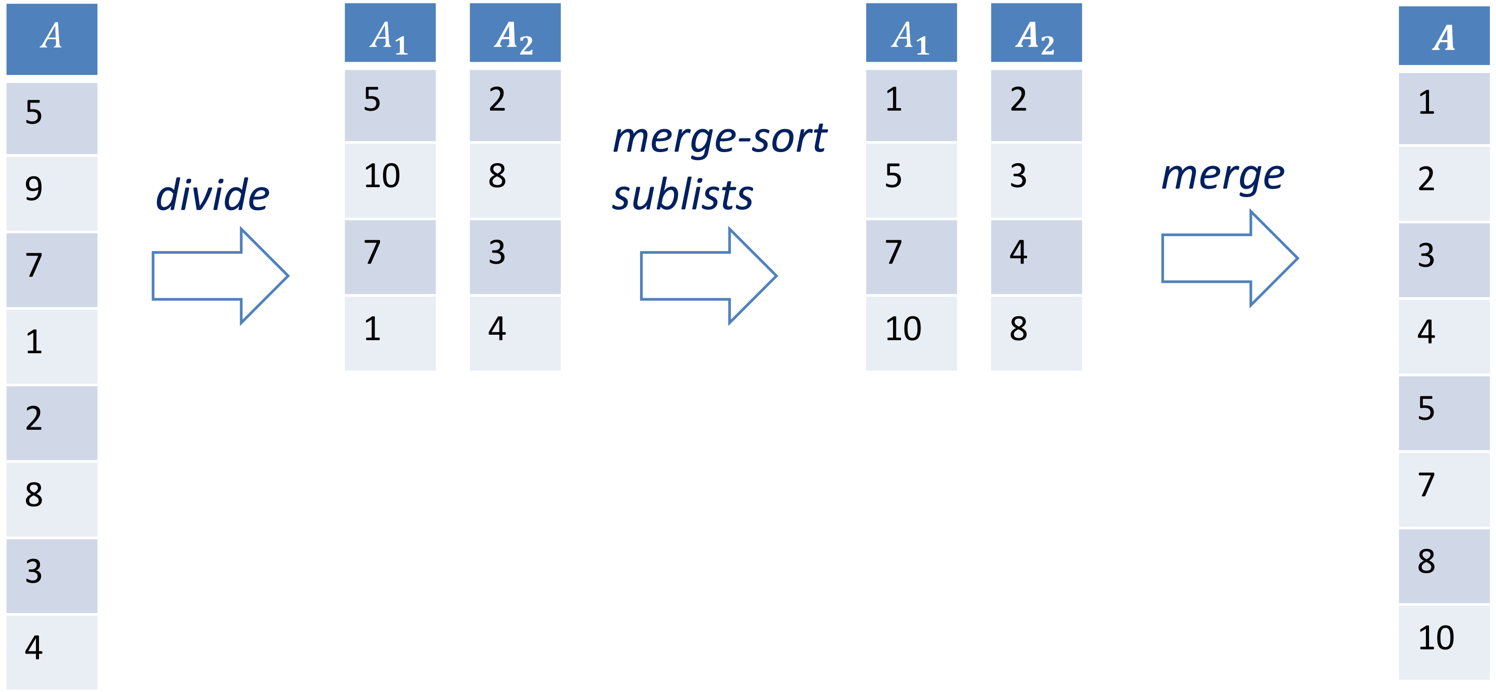
External sorting

- Problem: sort or hashing 1TB of data over 1GB of RAM
  - Why not virtual memory?
    - Swaps involve expensive random I/Os
  - Why not using B-Tree/extendible hashing/linear hashing?
    - Dynamic structures carry additional overhead for maintenance (not needed in QP)
    - Missing optimization opportunities with hybrid approach (see later)

- General wisdom:
  - I/O cost dominates the total cost
  - Design algorithms to reduce the number of I/Os

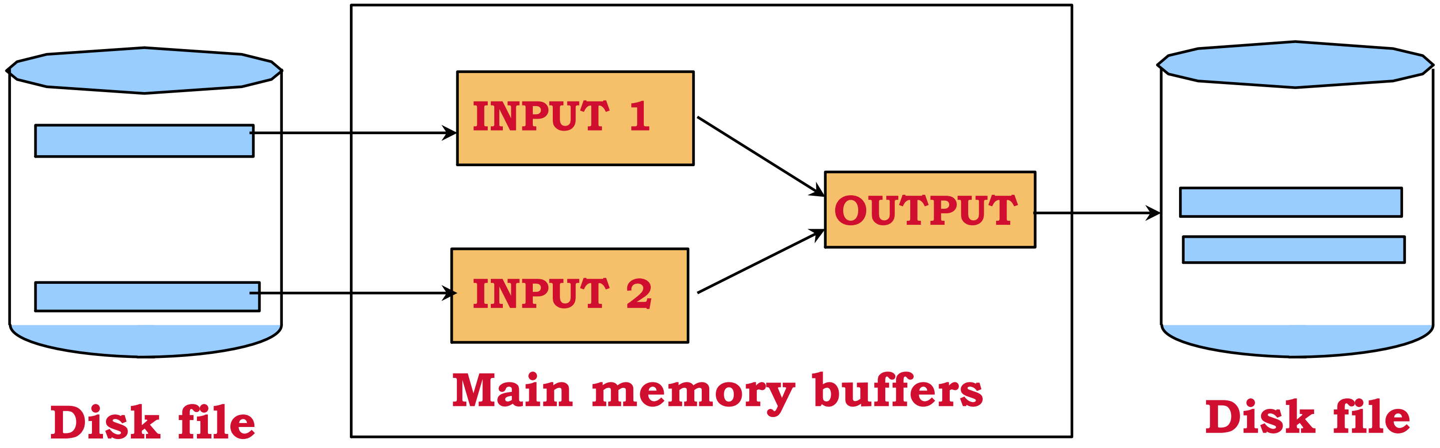
In-memory two-way merge-sort: a starting point

- Recall the two-way merge-sort
  - given a list of items in $A[0..n-1]$
  - recursively divide and conquer the problem
    - divide the list into two halves $A_1\left[0..\left\lfloor\frac{n}{2}\right\rfloor\right], A_2\left[\left\lfloor\frac{n}{2}\right\rfloor+1, n-1\right]$
    - merge-sort A_1 and A_2 individually
    - merge the two sorted list A_1, A_2



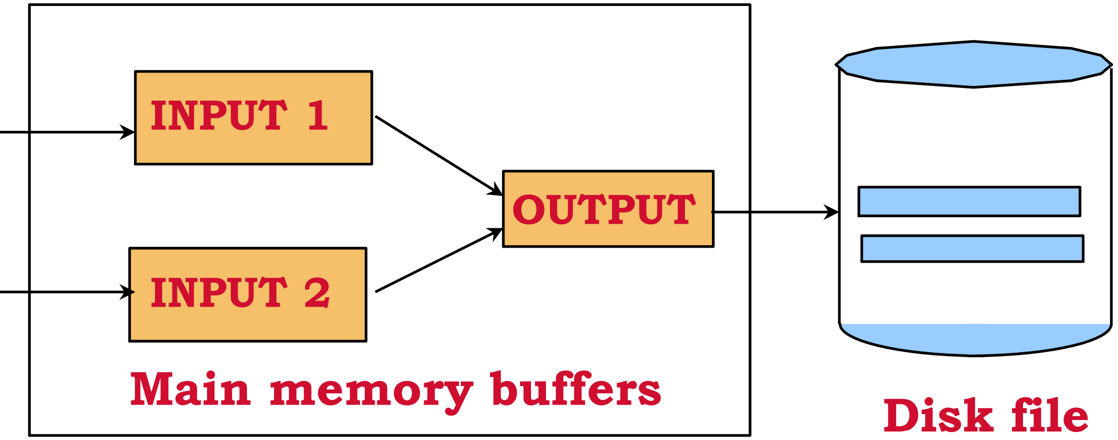
External two-way merge sort

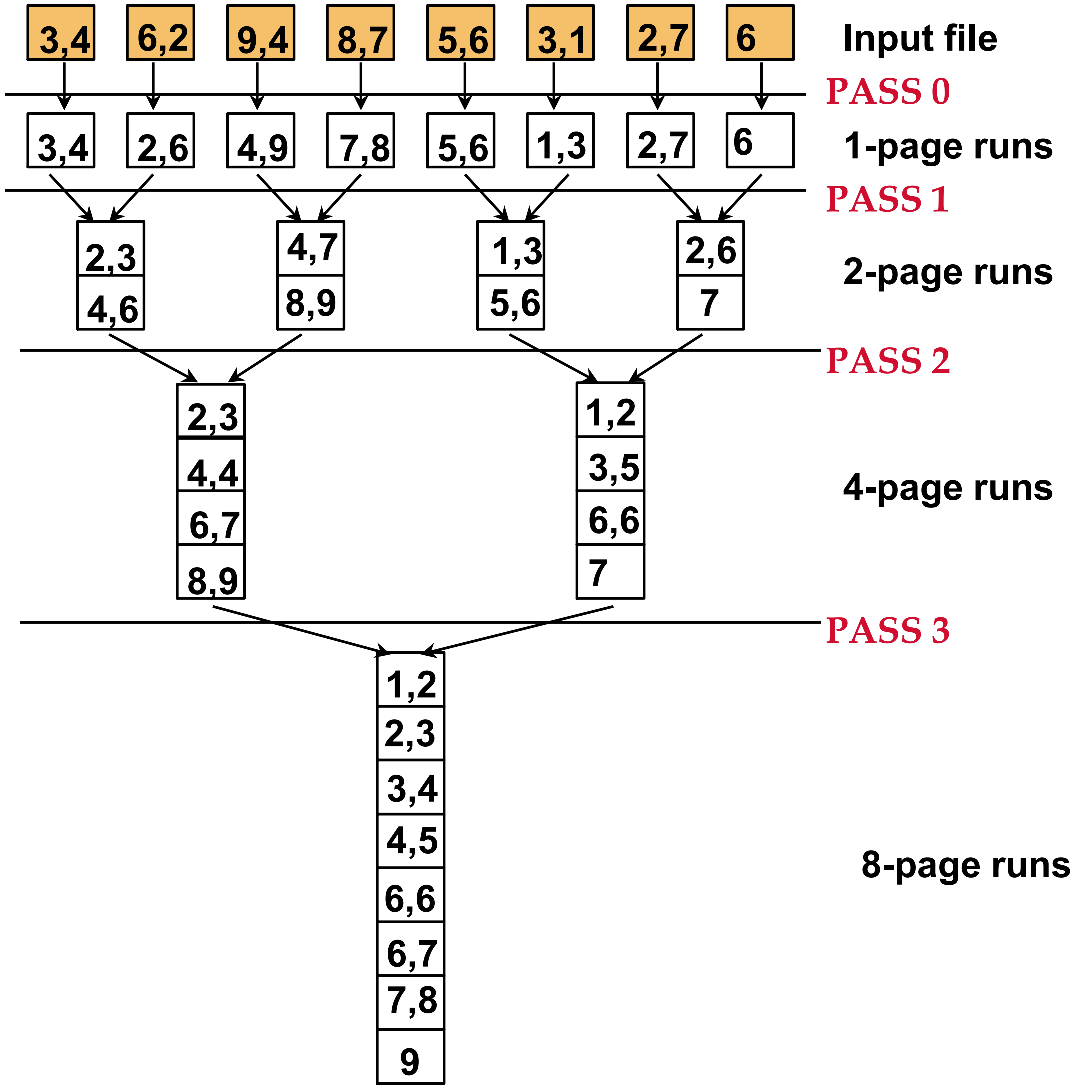
- Needs 3 buffers
- Instead of recursion
  - works bottom up from the input



External two-way merge sort

- Needs 3 buffers
- Instead of recursion
  - works bottom-up from the input

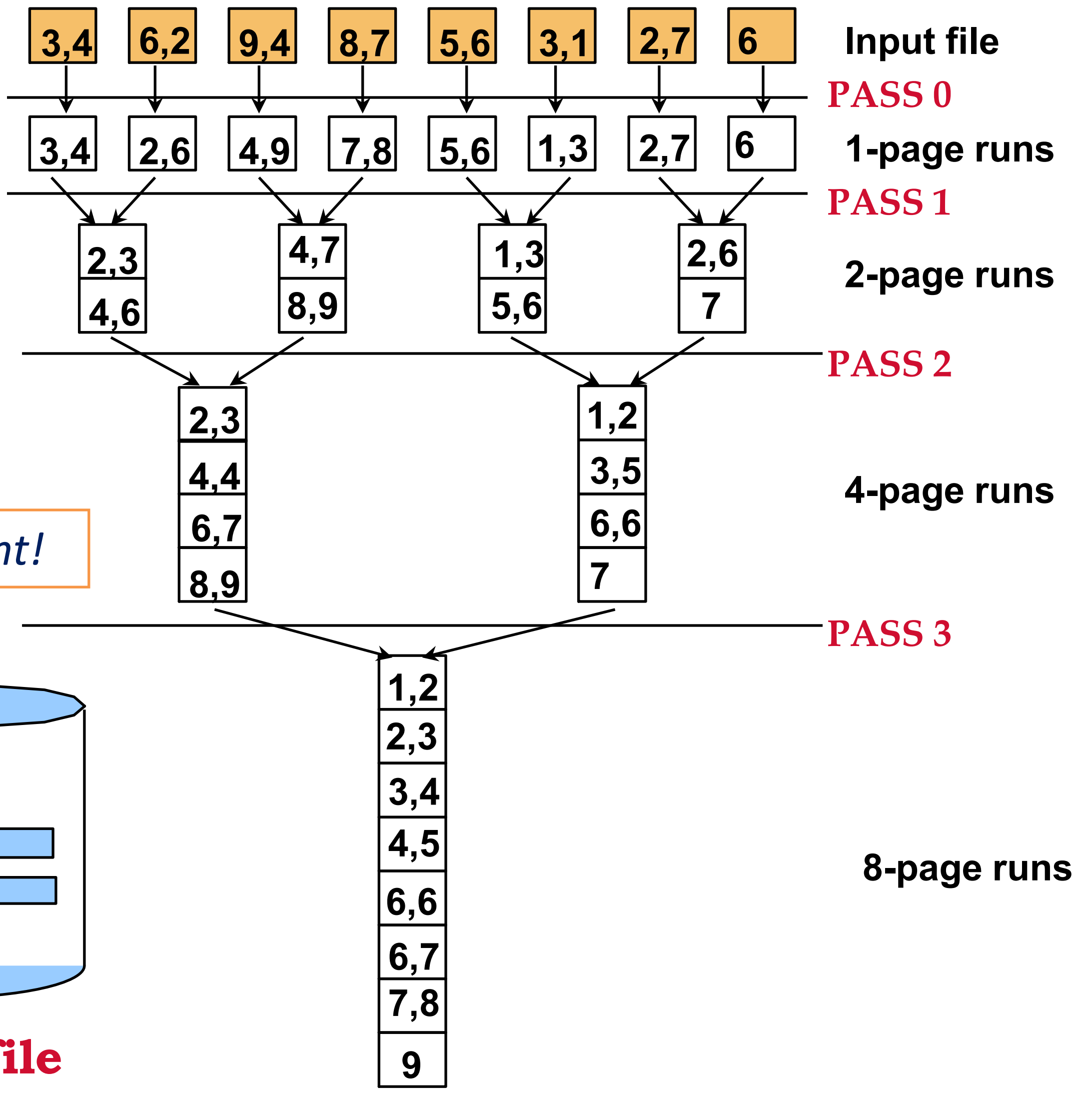


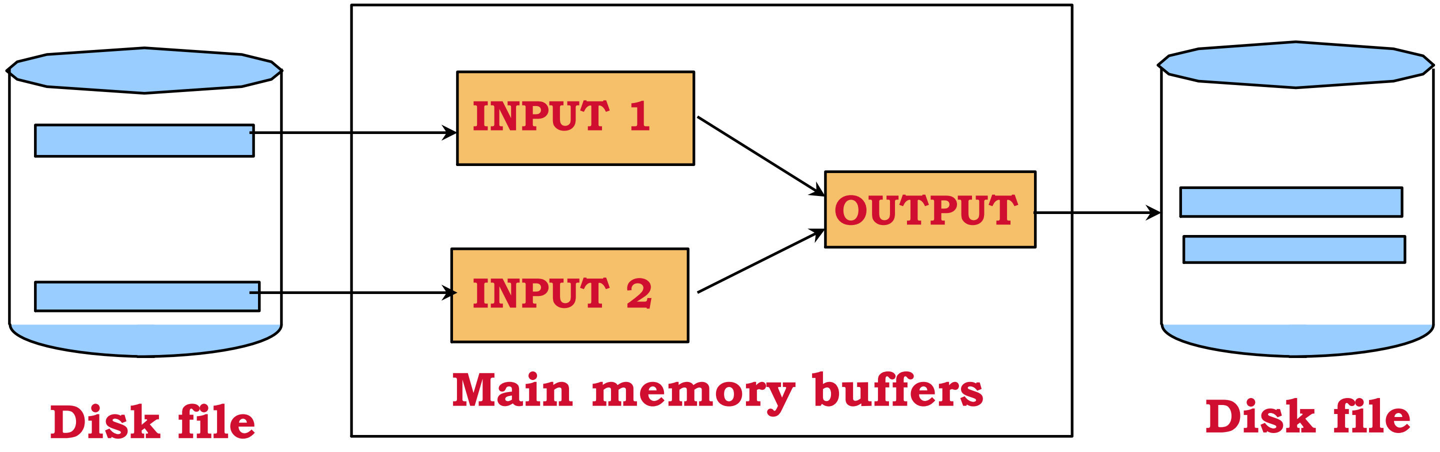


External two-way merge sort

- Input: N pages
- Cost for a pass: reading & writing N pages once
- # of passes: height of the tree = $\lceil \log_2 N \rceil + 1$
- Total cost: $2N(\lceil \log_2 N \rceil + 1)$ I/Os
  - Transfer cost: $2t_T N(\lceil \log_2 N \rceil + 1)$
  - *Seek cost:* $2t_S N(\lceil log_2 N \rceil + 1)$
  - *total* = $2(t_T + t_S)N(\lceil \log_2 N \rceil + 1)$

Not so efficient!

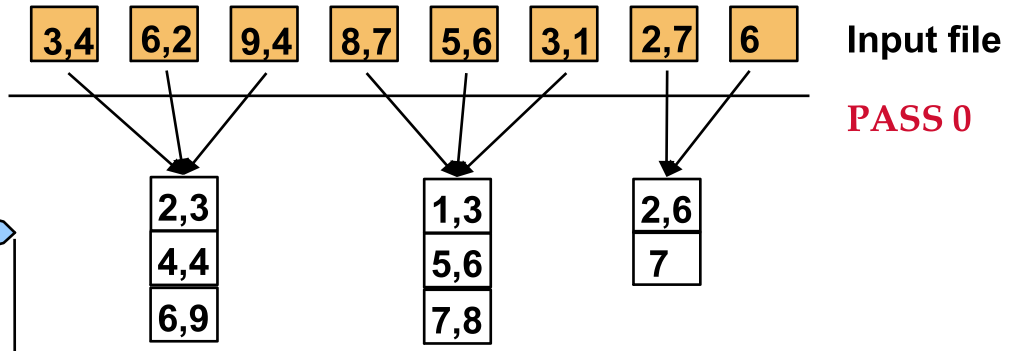


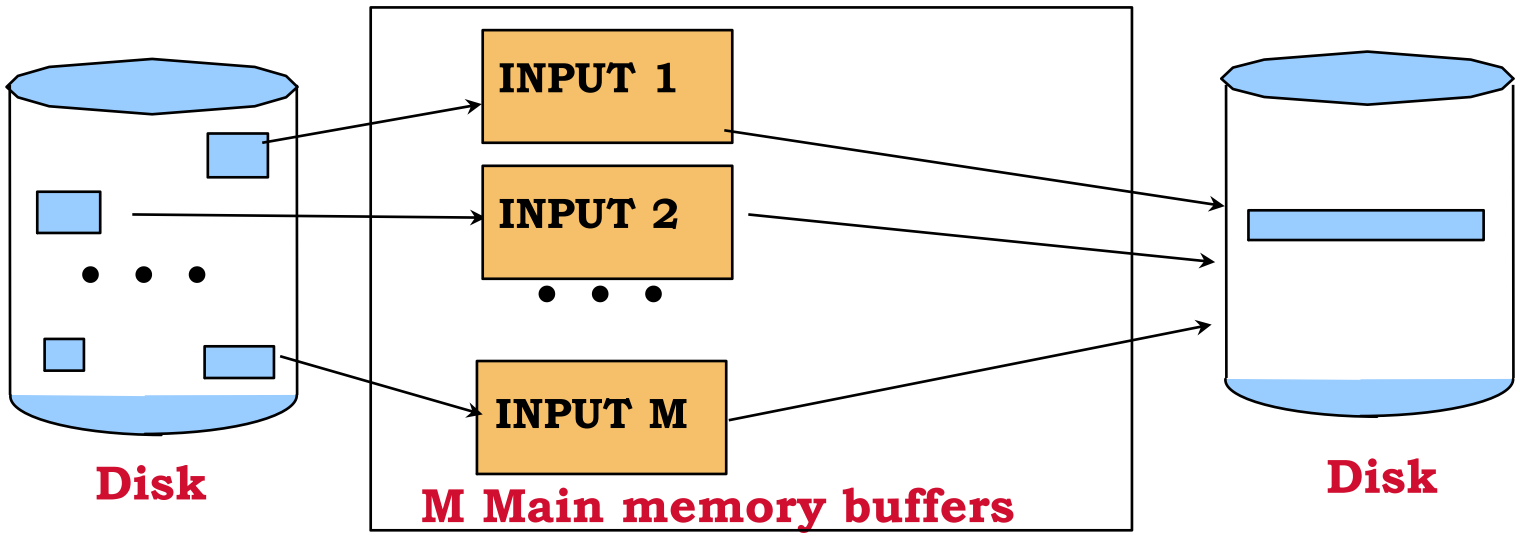


External multi-way merge sort

- How do we fully utilize all the M buffers?
  - Solution: (M-1)-way merge-sort
- Pass 0: internal sort to produce initial runs
  - read every M pages into memory
  - use some internal sorting algorithm (e.g., quick sort)
    - *can produce even larger runs (later)*
  - write all the M pages as a run

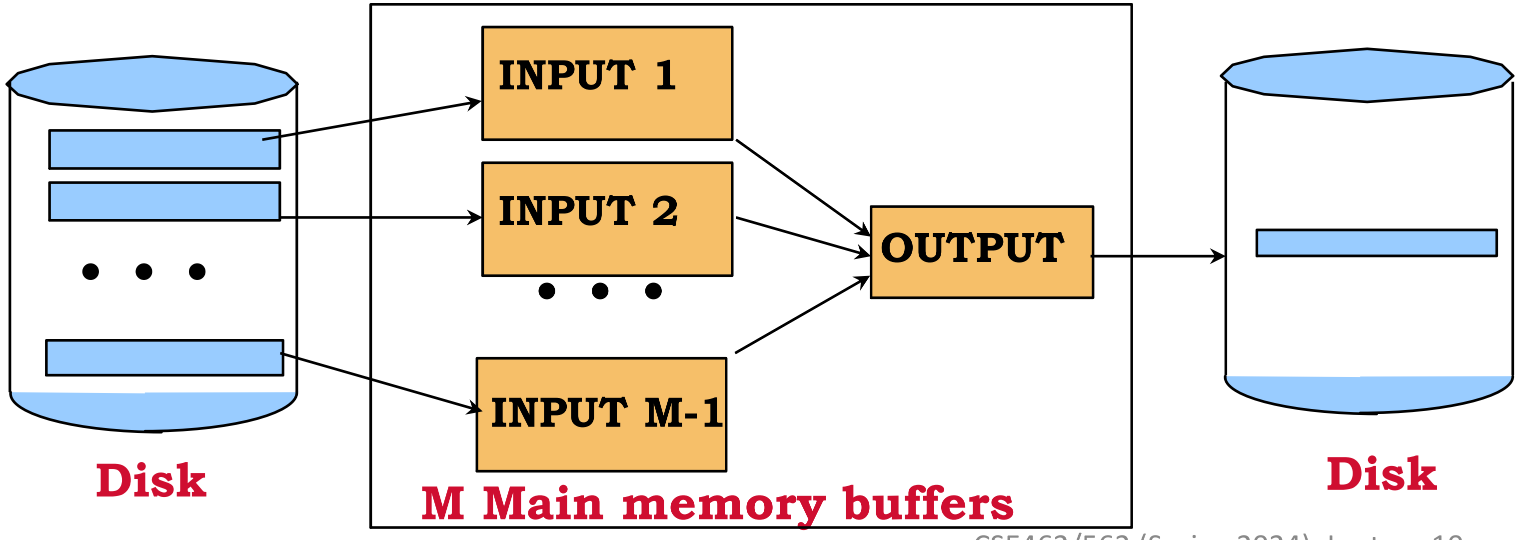
N pages in input
$\lceil \frac{N}{M} \rceil$ runs after pass 0
Cost:
$2N$ pages read/written +
$2\left\lceil \frac{N}{M} \right\rceil$ seeks
i.e. $2Nt_T + 2\left\lceil \frac{N}{M} \right\rceil t_S$

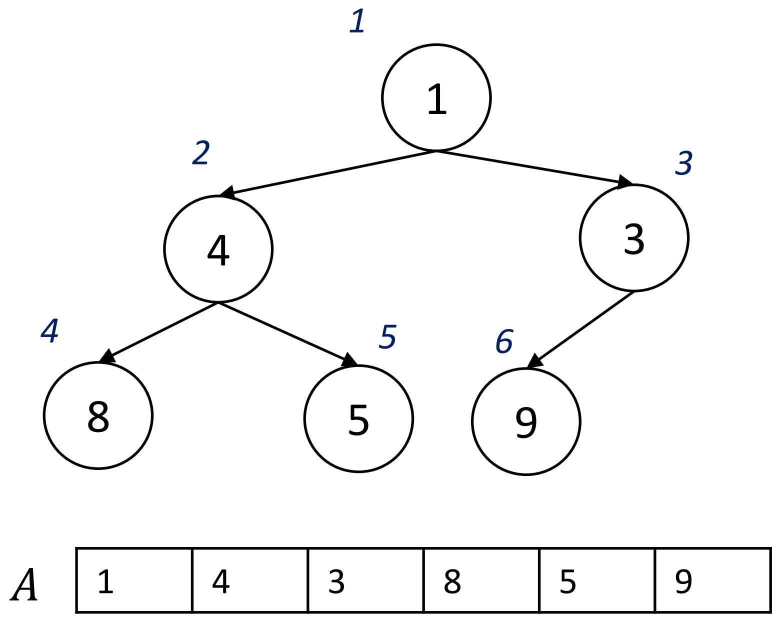




General multi-way merge sort

- Pass 1, 2, …: merge as many runs as possible from previous pass into a sorted run
  - maintain *a min-heap/max-heap (aka priority queue)*
    - supports $O(logM)$ time insertion of any item and deletion of the smallest/largest item
    - a complete binary tree where parent is smaller/larger than both children
    - how to implement
      - numbering nodes level by level sequentially from 1, store in an array $A[1..n]$
        - (how to translate 1-based index to 0-based in C/C++?)
      - parent of $A[i]$ is $A[i/2]$, left child of $A[i]$ is $A[i*2]$, right child of $A[i]$ is $A[i*2+1]$
      - push-down or push-up to maintain the variant

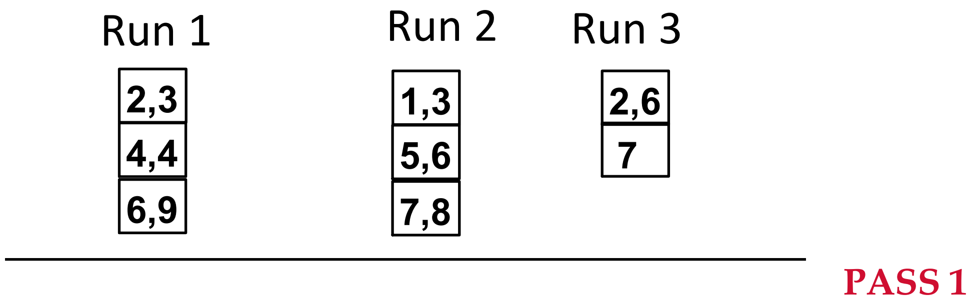




General multi-way merge sort

- Pass 1, 2, …: merge as many runs as possible from previous pass into a sorted run
  - maintain *a min-heap*
  - load one page from each of the $M - 1$ runs
    - and maintain pointers of next page to read
  - for each loaded page
    - insert the first key into the min-heap
    - maintain next slot ids for each page
  - Repeatedly remove the smallest item from the min heap
    - and replace it with the next key in its run
    - write out the output page once it's full

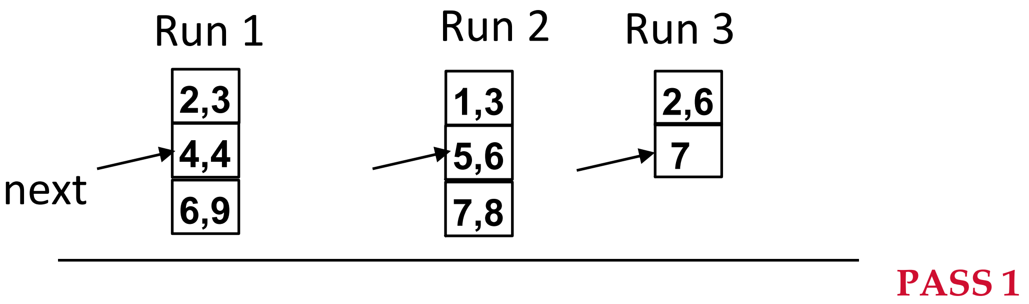
For illustration, let's now assume $M = 4$ instead of 3 from now on.

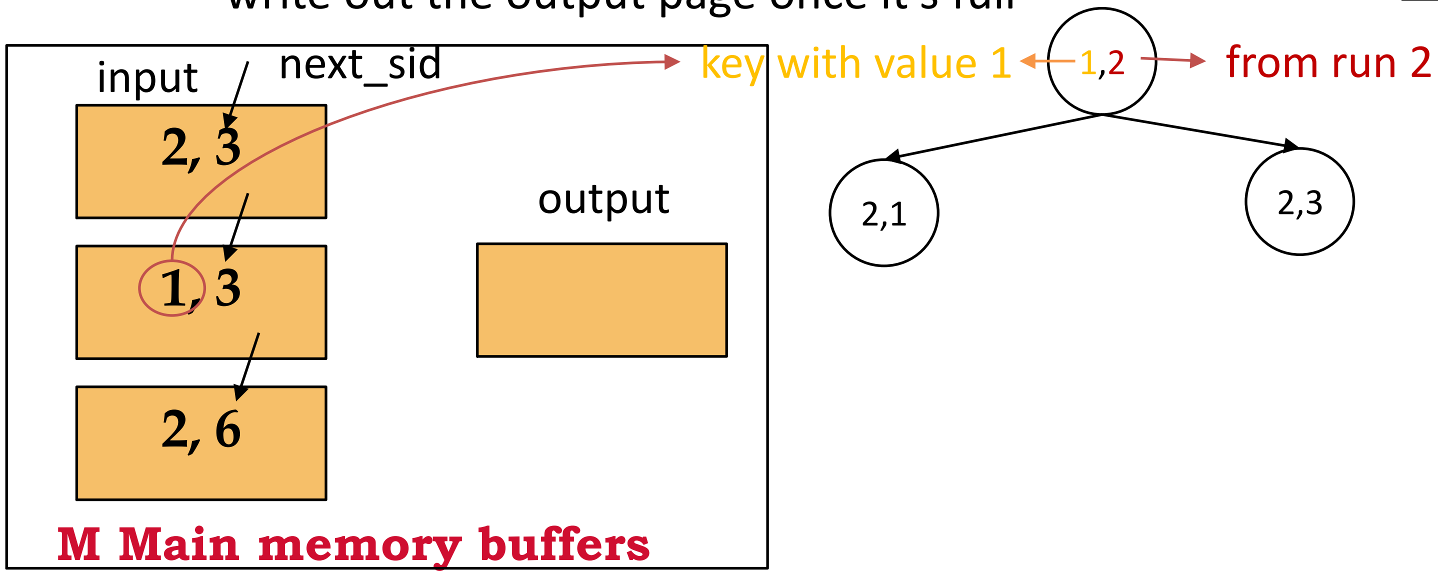


General multi-way merge sort

- Pass 1, 2, …: merge as many runs as possible from previous pass into a sorted run
  - maintain *a min-heap*
  - load one page from each of the $M-1$ runs
    - and maintain pointers of next page to read
  - for each loaded page
    - insert the first key into the min-heap
    - maintain next slot ids for each page
  - Repeatedly remove the smallest item from the min heap
    - and replace it with the next key in its run
    - write out the output page once it's full

For illustration, let's now assume $M = 4$ instead of 3 from now on.

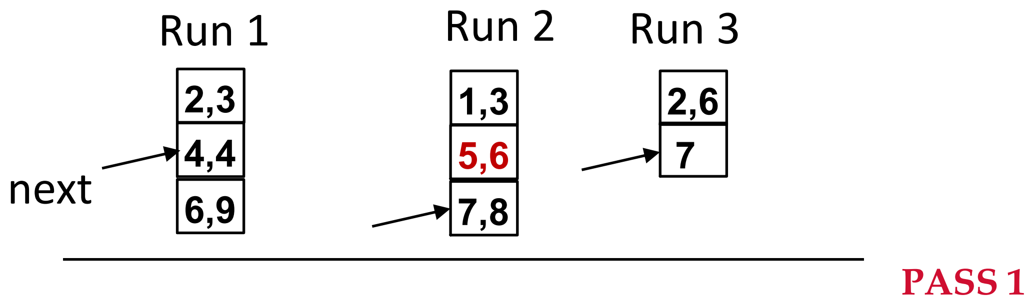


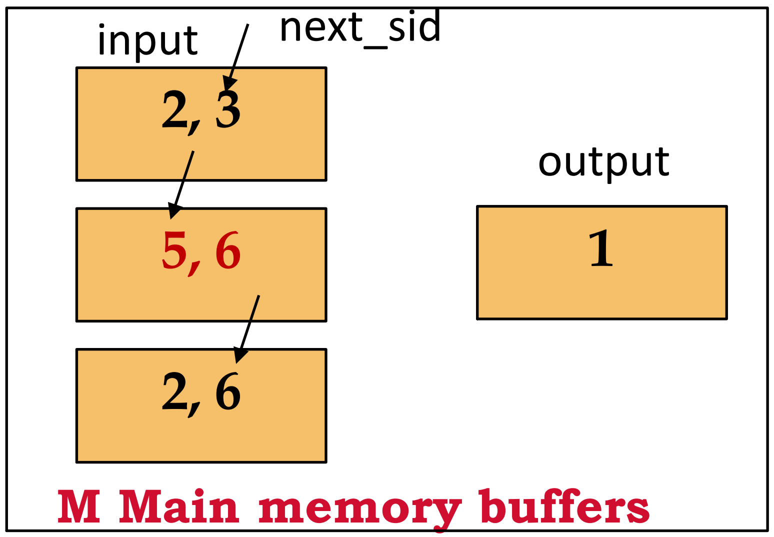


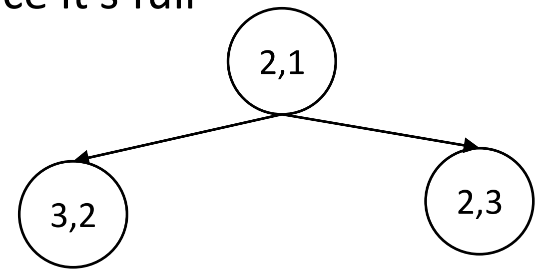
General multi-way merge sort

- Pass 1, 2, …: merge as many runs as possible from previous pass into a sorted run
  - maintain *a min-heap*
  - load one page from each of the $M-1$ runs
    - and maintain pointers of next page to read
  - for each loaded page
    - insert the first key into the min-heap
    - maintain next slot ids for each page
  - Repeatedly remove the smallest item from the min heap
    - and replace it with the next key in its run
    - write out the output page once it's full

For illustration, let's now assume M = 4 instead of 3 from now on.



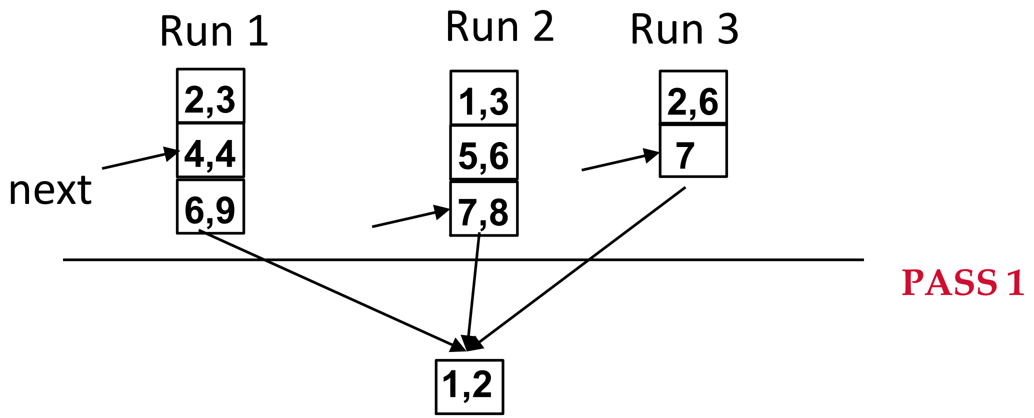


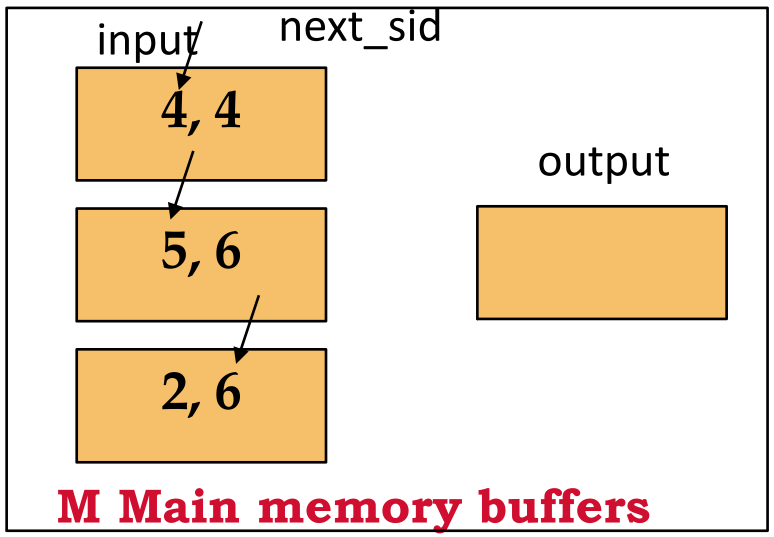


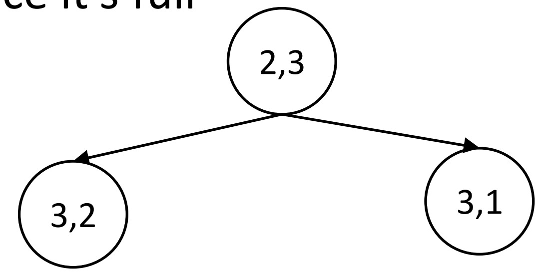
General multi-way merge sort

- Pass 1, 2, …: merge as many runs as possible from previous pass into a sorted run
  - maintain *a min-heap*
  - load one page from each of the $M - 1$ runs
    - and maintain pointers of next page to read
  - for each loaded page
    - insert the first key into the min-heap
    - maintain next slot ids for each page
  - Repeatedly remove the smallest item from the min heap
    - and replace it with the next key in its run
    - write out the output page once it's full

For illustration, let's now assume M = 4 instead of 3 from now on.



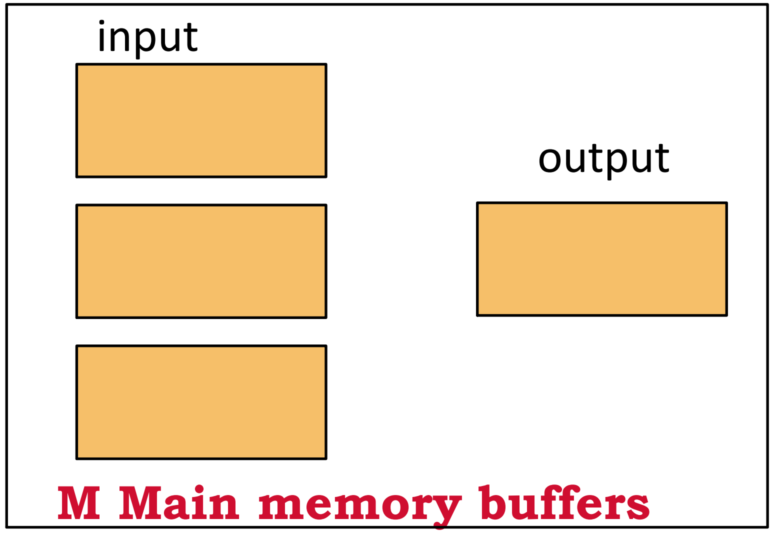




General multi-way merge sort

- Pass 1, 2, …: merge as many runs as possible from previous pass into a sorted run
  - maintain *a min-heap*
  - load one page from each of the $M-1$ runs
    - and maintain pointers of next page to read
  - for each loaded page
    - insert the first key into the min-heap
    - maintain next slot ids for each page
  - Repeatedly remove the smallest item from the min heap
    - and replace it with the next key in its run
    - write out the output page once it's full

For illustration, let's now assume M = 4 instead of 3 from now on.



N pages to read/write per pass

$\left\lceil log_{M-1} \left\lceil \frac{N}{M} \right\rceil \right\rceil$ merge passes

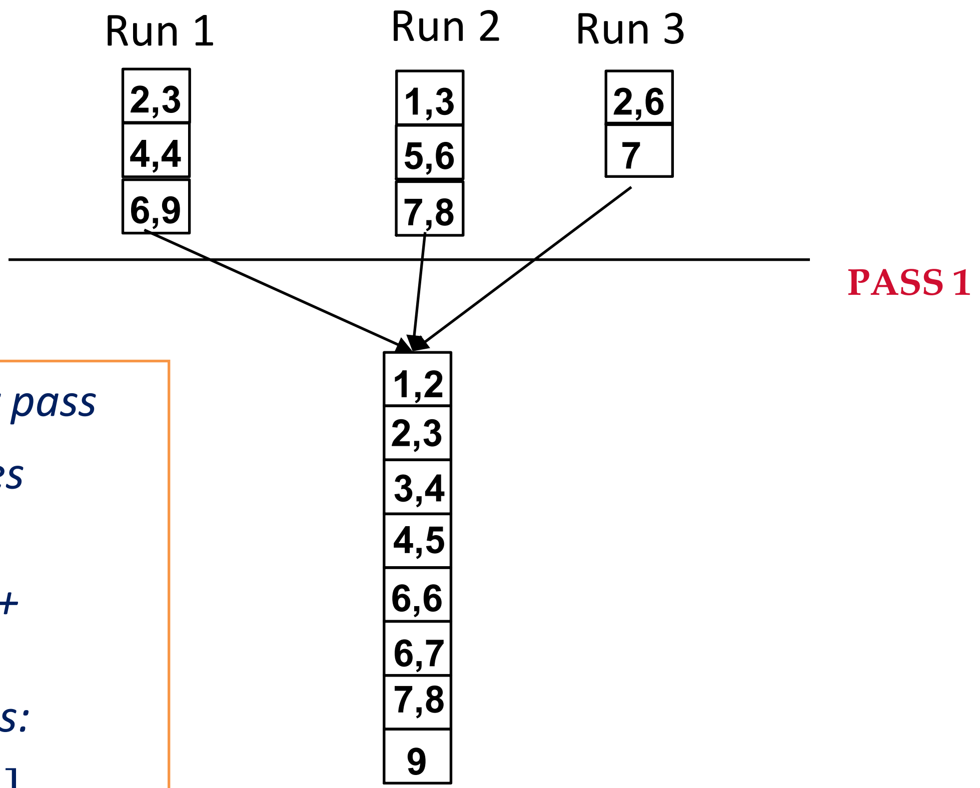
Cost per merge pass:

$2N$ pages read/written +

$2N$ seeks

Total cost for merge passes:

$2(t_T + t_S)N\lceil \log_{M-1} \lceil \frac{N}{M} \rceil \rceil$



Cost analysis

- Cost analysis:
  - Pass 0: $2Nt_T + 2\left\lceil\frac{N}{M}\right\rceil t_S$
  - Pass 1, 2, … combined: $2(t_T + t_S)N\lceil\log_{M-1}\lceil\frac{N}{M}\rceil\rceil$
  - Total = $2t_T N\left(\left\lceil log_{M-1}\left\lceil\frac{N}{M}\right\rceil\right\rceil + 1\right) + 2t_S\left(\left\lceil\frac{N}{M}\right\rceil + N\lceil log_{M-1}\lceil\frac{N}{M}\rceil\rceil\right)$

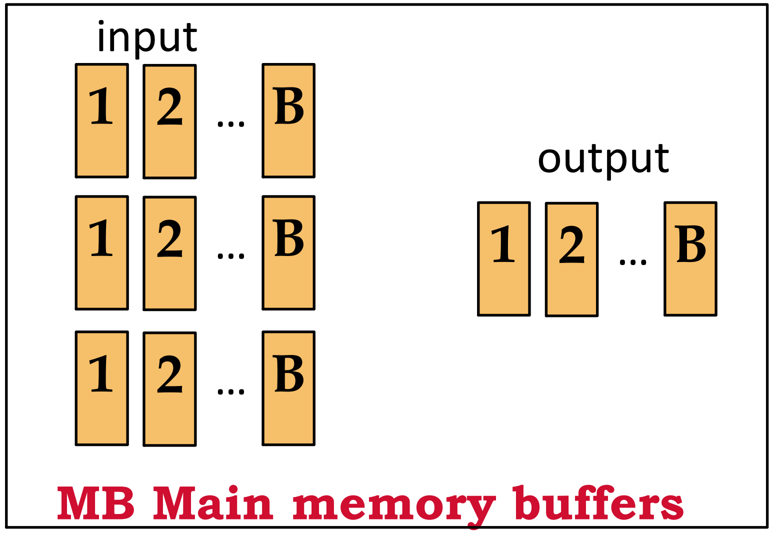
> - *gain of utilizing all available buffers*
> - *importance of a high fan-in during merging*

N	M=3	=5	=9	=17	=129	=257
100	7	4	3	2	1	1
1,000	10	5	4	3	2	2
10,000	13	7	5	4	2	2
100,000	17	9	6	5	3	3
1,000,000	20	10	7	5	3	3
10,000,000	23	12	8	6	4	3
100,000,000	26	14	9	7	4	4
1,000,000,000	30	15	10	8	5	4

  - Can we do it better?

Batching I/Os for merge sort

- Refinement 1
  - reducing random I/Os by reading/writing B pages per run during merge
  - using $(M-1)$-way merge sort
    - memory usage increases to MB pages
    - number of pages transferred do not change
    - but the number of random seeks per merge pass reduced to approximately $2\lceil\frac{N}{B}\rceil$
  - total cost reduced to $2t_T N\left(\left\lceil \log_{M-1}\left\lceil\frac{N}{MB}\right\rceil\right\rceil + 1\right) + 2t_S\left(\left\lceil\frac{N}{MB}\right\rceil + \lceil\frac{N}{B}\rceil\lceil \log_{M-1}\lceil\frac{N}{MB}\rceil\rceil\right)$

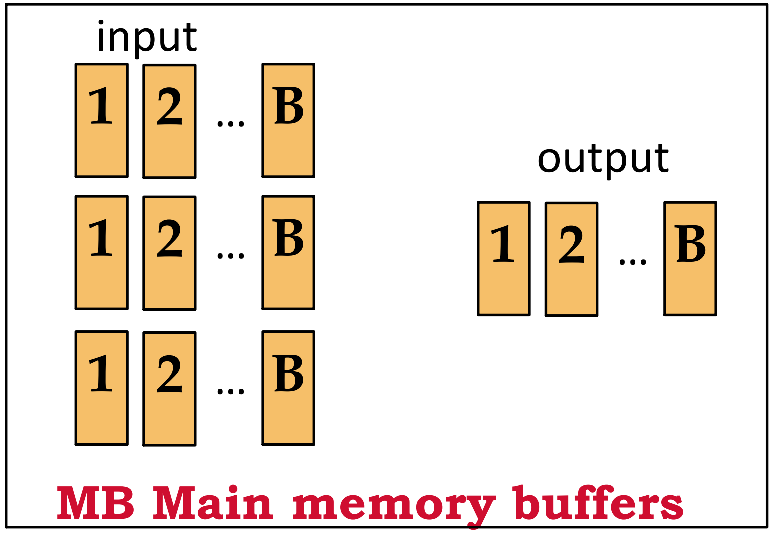


Exercise: what if we only have M pages instead of MB pages and still read/write pages in B-page batches?

$$2t_T N\left(\left\lceil \log_{\lfloor\frac{M}{B}\rfloor-1}\left\lceil\frac{N}{M}\right\rceil\right\rceil + 1\right) + 2t_S\left(\left\lceil\frac{N}{M}\right\rceil + \lceil\frac{N}{B}\rceil\lceil \log_{\lfloor\frac{M}{B}\rfloor-1}\lceil\frac{N}{M}\rceil\rceil\right)$$

Pipelining output

- Refinement 2
  - in most cases, do not need to write the final file
    - pipelining to the next operator
    - or output to user
  - Hence, no need to count the write of the final pass
  - total cost reduced to $t_T N \left(2 \left\lceil log_{\lfloor \frac{M}{B} \rfloor - 1} \left\lceil \frac{N}{M} \right\rceil \right\rceil + 1 \right) + t_S \left(2 \left\lceil \frac{N}{M} \right\rceil + \lceil \frac{N}{B} \rceil (2 \lceil log_{\lfloor \frac{M}{B} \rfloor - 1} \lceil \frac{N}{M} \rceil \rceil - 1) \right)$



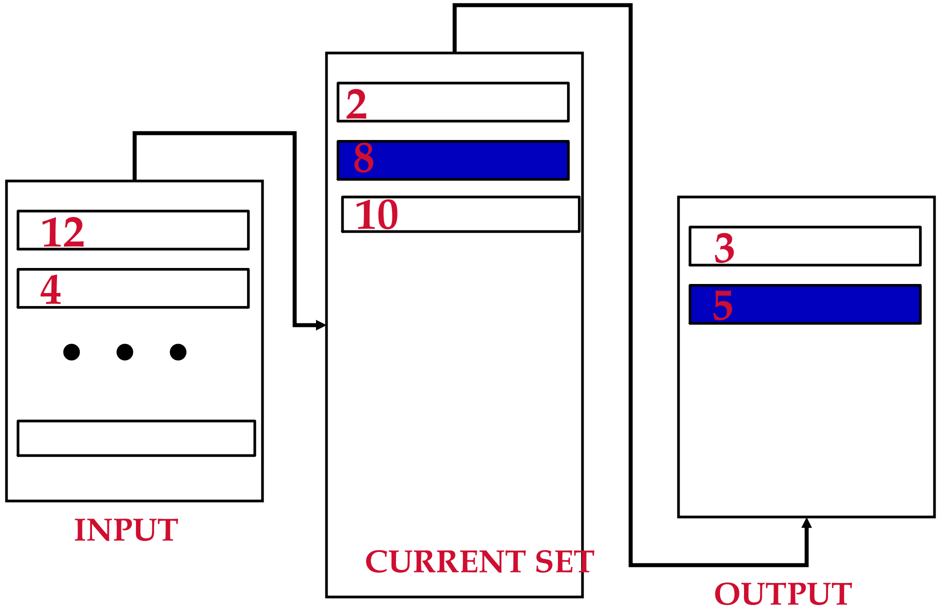
Tournament sort

- Refinement 3
  - producing initial runs as large as possible in pass 0
  - Alternative to quick-sort: “tournament sort” (a.k.a. “heapsort”, “replacement selection”)
- Keep two heaps in memory, H1 and H2, reserve an input buffer page and an output buffer page

```
read M-2 pages of records, inserting into H1;
while (records left) {
    m = H1.removemin();  put m in output buffer;
    if (H1 is empty)
        swap H1 and H2 (pointer swap only!); start new output run;
    else
        read in a new record r (use 1 buffer for input pages);
        if (r < m)  H2.insert(r);
        else        H1.insert(r);
}
H1.output();  start new run;  H2.output();
```

Tournament sort

- Tournament sort explained:



- 1 input, 1 output, M - 2 for current and next set (min heaps)
- Main idea: ensure the *smallest* key in the current set (H1) is *greater* than any key that has been written to this output run.
  - If it can't be satisfied, write to the next set (H2), which goes into the next run.
- Memory usage of the min-heaps combined never exceeds the M-2 pages

Tournament sort

- Fact: average length of a run is *2(M-2)*
- Total cost reduced to on average

$$t_T N\left(2\left\lceil log_{\lfloor\frac{M}{B}\rfloor-1}\left\lceil\frac{N}{2M-4}\right\rceil\right\rceil+1\right)+t_S\left(2\left\lceil\frac{N}{2M-4}\right\rceil+\lceil\frac{N}{B}\rceil(2\lceil log_{\lfloor\frac{M}{B}\rfloor-1}\lceil\frac{N}{2M-4}\rceil\rceil-1)\right)$$

- Worst-Case:
  - What is min length of a run?
  - How does this arise?
- Best-Case:
  - What is max length of a run?
  - How does this arise?
- Quicksort is faster, but … longer runs often means fewer passes!

Using B+ Trees for Sorting

- Scenario: Table to be sorted has B+ tree index on sorting column(s).
- Idea: Can retrieve records in order by traversing leaf pages.
- *Is this a good idea?*
- Cases to consider:
  - B+ tree is clustered — ***Good idea since it's already available!***
  - B+ tree is not clustered — ***Could be a very bad idea! (Random I/O) unless all columns are included in the key***

Certain basic operator implementation w/ sorting

- Some basic operators can be implemented on top of sorting
  - Can use pipelining over the sort results
- Examples
  - deduplication (projection in standard RA)
    - maintain the last key
    - for each output from the sort
      - emit it if it is different from the last key
      - otherwise, discard it
  - aggregation
    - maintain the aggregation state
    - for each output from the sort
      - emit the finalized aggregation value if it is different from the last key (unless this is the first)
      - otherwise, accumulate it to the state
  - exercise: work out the details of $\cup, \cap, -$
- No additional I/O due to pipelining
  - can support rewinding (why?)

This lecture

- Summary:
  - Aggregation and set operators
  - External sorting (multi-way merge-sort)
- Next lecture
  - Join algorithms

- Reminders:
  - HW4 released today, due on 4/22/2024, 23:59 PM EDT
  - Project 4 due next Monday, 4/15/2024, 23:59 PM EDT