Module-4

Basic Information:  
External sorting is a method for sorting large datasets that don't fit into main memory. Unlike internal sorting, which loads all data into memory at once, external sorting algorithms only work with small portions of data at a time, residing in external memory like disk drives. This approach minimizes the number of times data is read from and written to the external memory, which is much slower than main memory.

One way to implement external sorting is through algorithms like "external merge sort" and "external distribution sort." In external merge sort, the data is divided into smaller chunks, sorted individually, and then merged together in the correct order. Another approach, called "replacement selection strategy," optimizes the creation of sorted runs for merging by using priority queues.

The "two-way external merge sort" and "k-way external merge sort" are variations of external merge sort. In the two-way approach, runs are merged two at a time, while in the k-way approach, multiple runs are merged simultaneously. These methods minimize the number of times data needs to be read from and written to external memory, improving efficiency.

1. Comparisions:  
   **Two-Way External Merge Sort**:
   * **Description**: This algorithm sorts data stored in secondary storage by dividing it into runs, sorting each run in memory, and then merging the runs.
   * **Memory Usage**: Utilizes a fixed number of frames (M) in memory for sorting and merging runs. Requires at least 3 frames for operation.
   * **Number of Passes (N)**: Determined by the formula ⌈log(M-1)(B)⌉ + 1, where B is the number of pages in the file and M is the number of available frames. Each pass involves reading and writing each page of the file once.
   * **Cost**: The total number of memory accesses is 2×B×N.
   * **Performance**: Straightforward implementation but may require multiple passes, depending on the size of the data and available memory.
2. **K-Way External Merge Sort**:
   * **Description**: A variation of external merge sort where M pages are read at a time instead of one. This reduces the number of passes required for sorting.
   * **Memory Usage**: Utilizes M frames for sorting and merging runs, where M is the number of available frames.
   * **Number of Passes (N)**: Determined by the formula ⌈log(M-1)(⌈B/M⌉)⌉ + 1, where B is the number of pages in the file and M is the number of available frames.
   * **Cost**: The total number of memory accesses is 2×B×N.
   * **Performance**: Reduces the number of passes compared to the two-way algorithm by reading multiple pages at once. More efficient for larger datasets.
3. **Replacement Selection Strategy**:
   * **Description**: This strategy optimizes the creation of runs by utilizing priority queues. It aims to create sorted runs exceeding the number of available frames in memory.
   * **Memory Usage**: Utilizes M-2 frames for creating sorted runs, along with additional frames for priority queues.
   * **Number of Passes (N)**: Depends on the number of runs produced during the initial pass, which can vary.
   * **Cost**: Offers potential performance benefits by creating runs of more than M pages, leading to fewer overall passes compared to K-Way External Merge Sort, especially in the average case scenario.
   * **Performance**: Offers potentially better performance by creating runs of varying sizes, optimizing memory usage, and reducing the number of passes needed for sorting.

In summary, while Two-Way External Merge Sort and K-Way External Merge Sort are more straightforward and predictable in terms of memory usage and number of passes, the Replacement Selection Strategy offers potential improvements in performance by optimizing memory utilization and reducing the number of passes, particularly for datasets where the average case scenario applies. However, it comes with the complexity of managing priority queues and varying run sizes.

**Algorithms used:**

// Pass 0 of the Two-Way External Sorting Algorithm

function TwoWayPassZero():

while (no more pages in R):

load 1 page of relation R in the buffer

sort records of the page in the buffer

write corresponding page in a new run

// Pass 0 of the K-Way External Sorting Algorithm

function KWayPassZero():

while (no more pages in R):

load M pages of relation R in the M frames of the buffer

sort records of the M pages in the buffer

write corresponding pages in a new run

// Pass 0 of the Replacement Selection External Sorting Algorithm

function ReplacementSelectionPassZero():

load first page of relation R in the input frame F1

load next M-2 pages of R in the buffer

set Q1 = queue of records in the M-2 frames starting from F2

set Q2 = empty queue

while (F1 or Q1 or Q2 are not empty):

if (Q1 is empty):

move Q2 into Q1

if (output frame is not empty):

write output frame in secondary storage

create new run

find minimum record r1 in Q1

move r1 in the output frame

if (F1 is empty and R has more pages):

read next page from R

if (F1 is NOT empty):

point to the next record r2 in input frame F1

if (r1 < r2):

move r2 in Q1

else:

move r2 in Q2

if (output frame is full):

write output frame in secondary storage

Code:

Replacement Selection Sort:  
from queue import PriorityQueue

class Relation:

    def \_\_init\_\_(self, pages):

        self.pages = pages

        self.page\_index = 0

    def read\_next\_page(self):

        if self.page\_index < len(self.pages):

            page = self.pages[self.page\_index]

            self.page\_index += 1

            return page

        return None

    def has\_next\_page(self):

        return self.page\_index < len(self.pages)

    def write\_page(self, page):

        print("Writing page to secondary storage:", page)

def replacement\_selection\_pass\_zero(relation\_R, M):

    input\_frame = []

    output\_frame = []

    Q1 = PriorityQueue()

    Q2 = PriorityQueue()

    input\_frame.append(relation\_R.read\_next\_page())

    for \_ in range(M - 2):

        page = relation\_R.read\_next\_page()

        if page:

            input\_frame.append(page)

    for frame in input\_frame[1:]:

        for record in frame:

            Q1.put(record)

    while input\_frame or not Q1.empty() or not Q2.empty():

        if Q1.empty():

            while not Q2.empty():

                Q1.put(Q2.get())

            if output\_frame:

                relation\_R.write\_page(output\_frame)

                output\_frame = []

        if input\_frame:

            r1 = Q1.get()

            output\_frame.append(r1)

            if not input\_frame and relation\_R.has\_next\_page():

                input\_frame.append(relation\_R.read\_next\_page())

            if input\_frame and input\_frame[0]:

                r2 = input\_frame[0].pop(0)

                if r2 < r1:

                    Q1.put(r2)

                else:

                    Q2.put(r2)

            if len(output\_frame) == M:

                relation\_R.write\_page(output\_frame)

                output\_frame = []

# Sample input

pages = [

    [5, 8, 10],

    [3, 6, 9],

    [2, 4, 7]

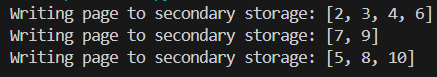
]

relation\_R = Relation(pages)

M = 4

replacement\_selection\_pass\_zero(relation\_R, M)

Output:



2-way merge sort:

from typing import List

class Relation:

    def \_\_init\_\_(self, pages: List[List[int]]):

        self.pages = pages

        self.page\_index = 0

    def read\_next\_page(self):

        if self.page\_index < len(self.pages):

            page = self.pages[self.page\_index]

            self.page\_index += 1

            return page

        return None

    def has\_next\_page(self):

        return self.page\_index < len(self.pages)

    def write\_page(self, page):

        print("Writing page to secondary storage:", page)

def two\_way\_pass\_zero(relation\_R):

    while relation\_R.has\_next\_page():

        page = relation\_R.read\_next\_page()

        sorted\_page = sorted(page)

        relation\_R.write\_page(sorted\_page)

# Modified input

pages = [

    [9, 6, 3],

    [7, 4, 1],

    [5, 2, 0]

]

relation\_R = Relation(pages)

two\_way\_pass\_zero(relation\_R)

Output:

A black background with white text

Description automatically generated

k-way merge sort:

(for 3 frames)

Code:  
class Relation:

    def \_\_init\_\_(self, pages):

        self.pages = pages

        self.page\_index = 0

    def read\_next\_page(self, M):

        start = self.page\_index

        end = min(self.page\_index + M, len(self.pages))

        self.page\_index = end

        return self.pages[start:end]

    def has\_next\_page(self):

        return self.page\_index < len(self.pages)

    def write\_pages(self, pages):

        for page in pages:

            print("Writing page to secondary storage:", " ".join(map(str, page)))

def k\_way\_pass\_zero(relation\_R, M):

    while relation\_R.has\_next\_page():

        pages = relation\_R.read\_next\_page(M)

        sorted\_pages = sorted([record for page in pages for record in page])

        print("Writing page to secondary storage:", " ".join(map(str, sorted\_pages)))

# Sample input

pages = [

    [3, 9, 6],

    [10, 5, 8],

    [4, 7, 2],

    [11, 12, 1]

]

relation\_R = Relation(pages)

# Number of frames in the buffer

M = 3

k\_way\_pass\_zero(relation\_R, M)

Output:  
