Sapienza
Università di Roma

Department of INGEGNERIA INFORMATICA, AUTOMATICA E GESTIONALE ”ANTONIO RUBERTI”

External Multi-Pass Sorting Visualizer

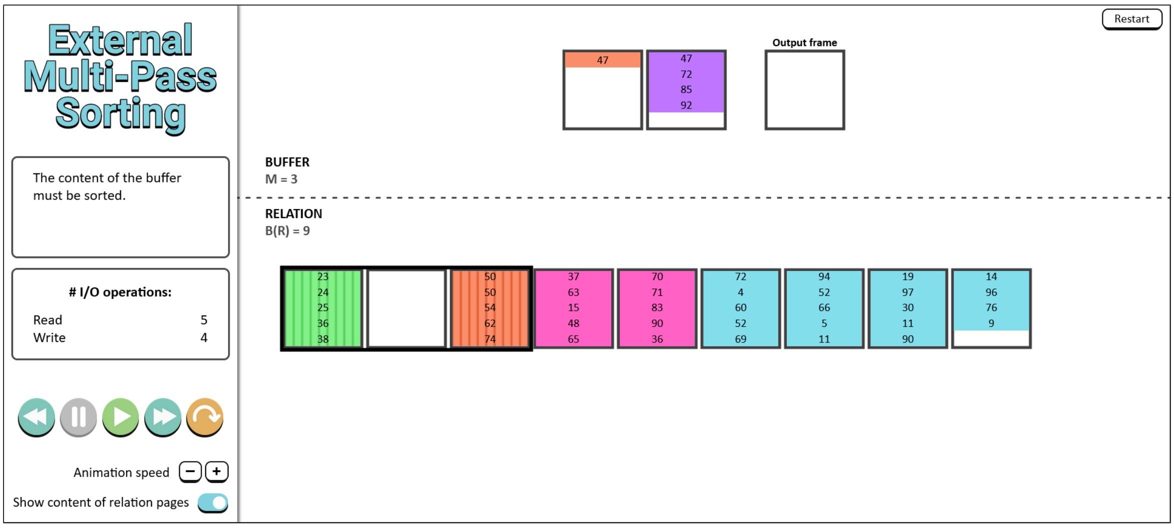
Data Management

Students:
Anna Carini 1771784
Jacopo Fabi 1809860

Academic Year 2023/2024

Contents

1 Introduction



The content of the buffer must be sorted.

I/O operations:

Read 5
Write 4

This project has been realized for the Data Management course of the master's degree in Engineering in Computer Science. It consists of a software for visualizing the External Multi-Pass Sorting algorithm.

The visualizer has been developed in JavaScript and deployed using GitHub Pages, so that it can be easily accessed from the web.

The project has been realized following these steps:

1. Fully understanding the algorithm (”External Multi Pass Sorting Algorithm” section).

2. Deciding which functionalities to offer to the users (”Functionalities” section).

3. Designing the application and creating a mock-up (”Design Choices” section).

4. Creating a flow chart of the application's states (the chart is in the ”States Flow Chart” section, but described in the ”Implementation” section).

5. Implementing the application (”Implementation” section).

2 External Multi Pass Sorting Algorithm

External sorting is a class of algorithms used when the data to be sorted does not fit into main memory. Through External Multi Pass Sorting, data is sorted using both main and external memory, with the data to be sorted residing in secondary memory. Let's define the available main memory as the Buffer, with **M** representing the number of frames in the buffer. Additionally, let's define **B(R)** as the number of pages in secondary memory that need to be sorted.
The algorithm can be divided into two phases:

- Phase 1 - Sorting
  In this phase, the pages in secondary memory are divided into **B(R)/M** chunks, each chunk containing at most **M** pages. All pages in each chunk are read into the buffer simultaneously. The elements within the pages are then sorted within the buffer, and the resulting pages are rewritten to secondary memory.

- Phase 2 - Merge
  At the beginning of this phase, there are **B(R)/M** sorted chunks (called Runs) on secondary memory. The last frame of the buffer is designated as the output frame. Until the number of Runs exceeds one, a Pass is performed:

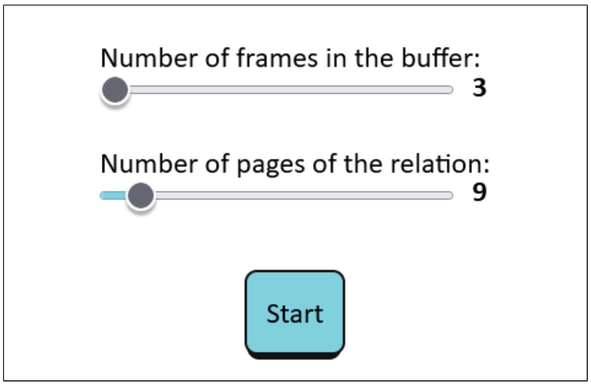
  - Pass
    Load the first pages of **M - 1** runs into the buffer. Sort the elements, writing them into the output buffer and removing them from the corresponding frame. When the output frame is full, its content is written to secondary memory. When a frame is empty, if the corresponding run has another page to read, load it into the buffer; otherwise, leave it empty. All pages written to secondary memory during this phase belong to a new single run. The old runs that have been read are deleted. When all the pages of the **M - 1** runs are sorted and if there is at least one run that is still unread and unsorted, load another **M - 1** runs into the buffer as explained at the beginning of the pass. Repeat this process until all runs have been read.

  If there is only 1 run remaining, it means that all pages have been sorted, and thus the algorithm terminates.

3 Functionalities

In order to illustrate the functioning of the algorithm, we decide to provide the user with the following functionalities.

3.1 Start menu



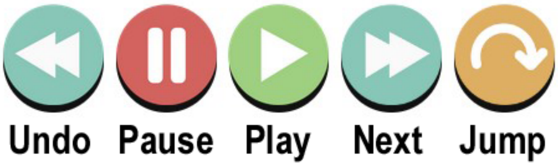
Number of frames in the buffer: 3

Number of pages of the relation: 9

First of all, when the application is started, the user is shown a menu through which they can decide the size of the buffer (how many frames, including the output frame) and the size of the relation (how many pages). By pressing ”Start”, the simulation begins with the selected parameters.

3.2 Controls

After the simulation has started, the user is allowed to do some actions, using either the buttons shown in the picture below, or some keyboard controls.



These are the available actions:

- ”**Next**”: Executes a single step of the algorithm, showing the animation. Via keyboard, it can be done by pressing the **enter key**.
- ”**Jump**”: Executes a single step skipping the animation, useful to move more quickly to different parts of the algorithm. Via keyboard, it can be done by pressing the **right arrow key**.
- ”**Play**”: Starts playing the algorithm automatically, step by step. It can be done through the **space bar**.
- ”**Pause**”: If the algorithm is being played automatically, it pauses it. It can be done through the **space bar**.

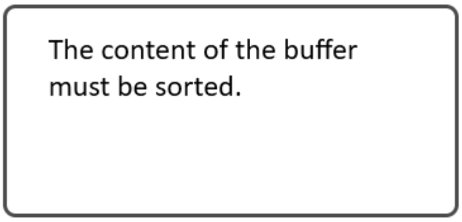
- "**Undo**": Undoes the last step that has been executed, restoring the previous states of the relation and buffer. Via keyboard it can be done by pressing the **left arrow key**.

The user is also able to modify the speed of the animation. This can be done using either some buttons we provided, or by pressing the "plus" (+) key to increase the speed, or the "minus" (-) key to decrease the speed.

Lastly, we added a toggle that allows the user to show or hide the content of the pages of the relation. In this way, in case the relation has a high amount of pages and thus the numbers written inside each page would appear very small, the user can reduce the visual clutter by hiding them. The algorithm can still be understood thanks to the fact that the sub-groups have different colors and that sorted sub-groups have a texture applied on them.

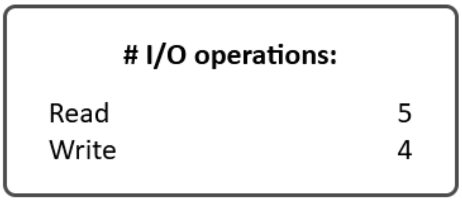
3.3 Information boxes

On the left column of the interface, we added two information boxes.



The content of the buffer must be sorted.

The upper one is a message box that gives information about the current state of the application. Every time a step is executed – and thus the application moves to a different state – the content of this box changes, to explain to the user what is happening and what is the next action that will be executed. The content of this box is hidden when the automatic play is active, because the messages would change automatically and the user wouldn't have enough time to read them.



```
# I/O operations:

Read        5
Write       4
```

The lower box shows the amount of read and write operations that have been done up to the current step of the execution. A read operation consists in copying a page of the relation from secondary storage to the buffer in main memory, while a write operation consists in writing back a page of the buffer to the secondary storage. These are the most expensive operations, so they determine the cost of the algorithm.

4 Design Choices

4.1 Why a web application?

We decided to make this visualizer a web application because we found it was the easiest way to achieve high **portability**: we exploited *Github Pages* to make the application available on any device that has access to the internet. We tested the application on different browsers to ensure browser compatibility.

4.2 Libraries used

In order to create the necessary graphics, we relied on two libraries: Two.js and Tween.js.

- **Two.js**
  Two.js is a Javascript library that provides a two-dimensional drawing API, specialized for vector graphics. We chose to use it because it provides an easy interface to draw basic shapes like squares and texts, and also functions to translate and scale these shapes, so it was a good fit for our needs.

- **Tween.js**
  Tween.js is a Javascript animation engine. It provides a simple tween function, that can be exploited in a render loop to smoothly interpolate values, and generate an animation. We combined Tween with Two's translation and scaling functions to create the animations we needed.

4.3 Parameters

As described in the "Functionalities" section, the user is given a menu through which they can decide the size of the buffer and the size of the relation.

The minimum size of the buffer is 3: this is indeed the minimum amount of frames that is required in order to run the merge phase of the algorithm. The maximum size of the buffer is 10; this choice has been caused by graphical needs, as we wanted to show the buffer on a single row and having more than 10 frames would have forced us to make the squares smaller, making it difficult to read their content. This limit doesn't subtract value to the application, as the goal of showing the behavior of the algorithm with a varying size of the buffer is still satisfied.

The amount of pages of the relation can instead vary between 1 and 100. In order to fit up to 100 pages, when necessary the squares and their content are automatically scaled down. As mentioned in the ”Functionalities” section, it is possible to hide the content of the pages of the relation, which is useful if the squares are quite small.

Each frame of the buffer and each page of the relation can contain at most 5 values. This choice has, again, been taken for graphical reasons. This value can’t be changed by the users because we didn’t consider it an interesting feature, as the behavior of the application would still be the same.

4.4 Interface design

Before diving into the implementation, we built a **mock-up** to decide what we wanted the interface to look like.

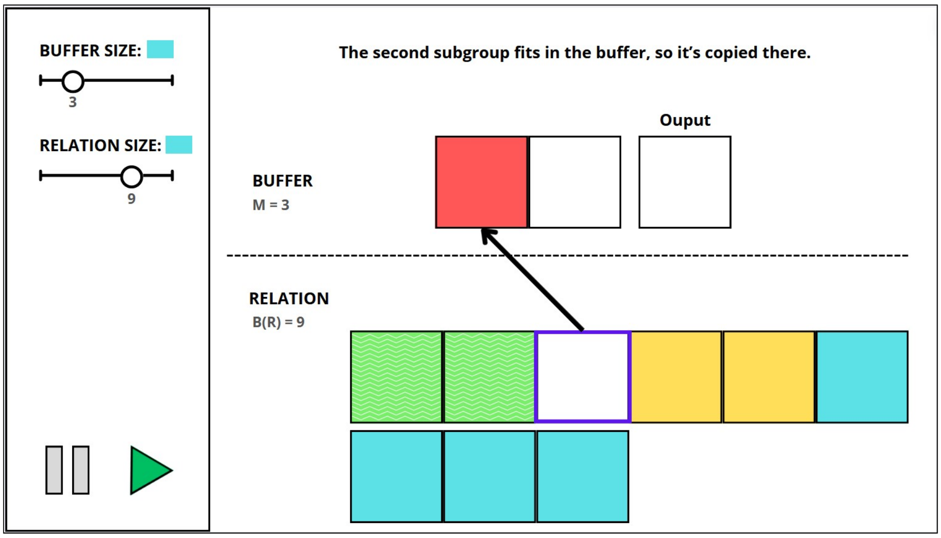

BUFFER SIZE:
3
RELATION SIZE:
9
The second subgroup fits in the buffer, so it’s copied there.
Ouput
BUFFER
M = 3
RELATION
B(R) = 9

Figure 1: The mock-up, realized using Canva

This mock-up has allowed us to explore the best solutions for some critical graphical aspects:

- How to show that the relation is divided into sub-groups: we did it by assigning a different color to each sub-group.
- How to show which group is currently taken into consideration: we highlighted the current group by framing it with a black box (purple in the mock-up).
- How to show which sub-groups have been sorted: we applied a texture on the pages of the sorted sub-groups.

The final application is quite similar to the mock-up, but we added more controls for the execution of the algorithm and we moved the information box to the left, to have more space for the buffer and the relation.

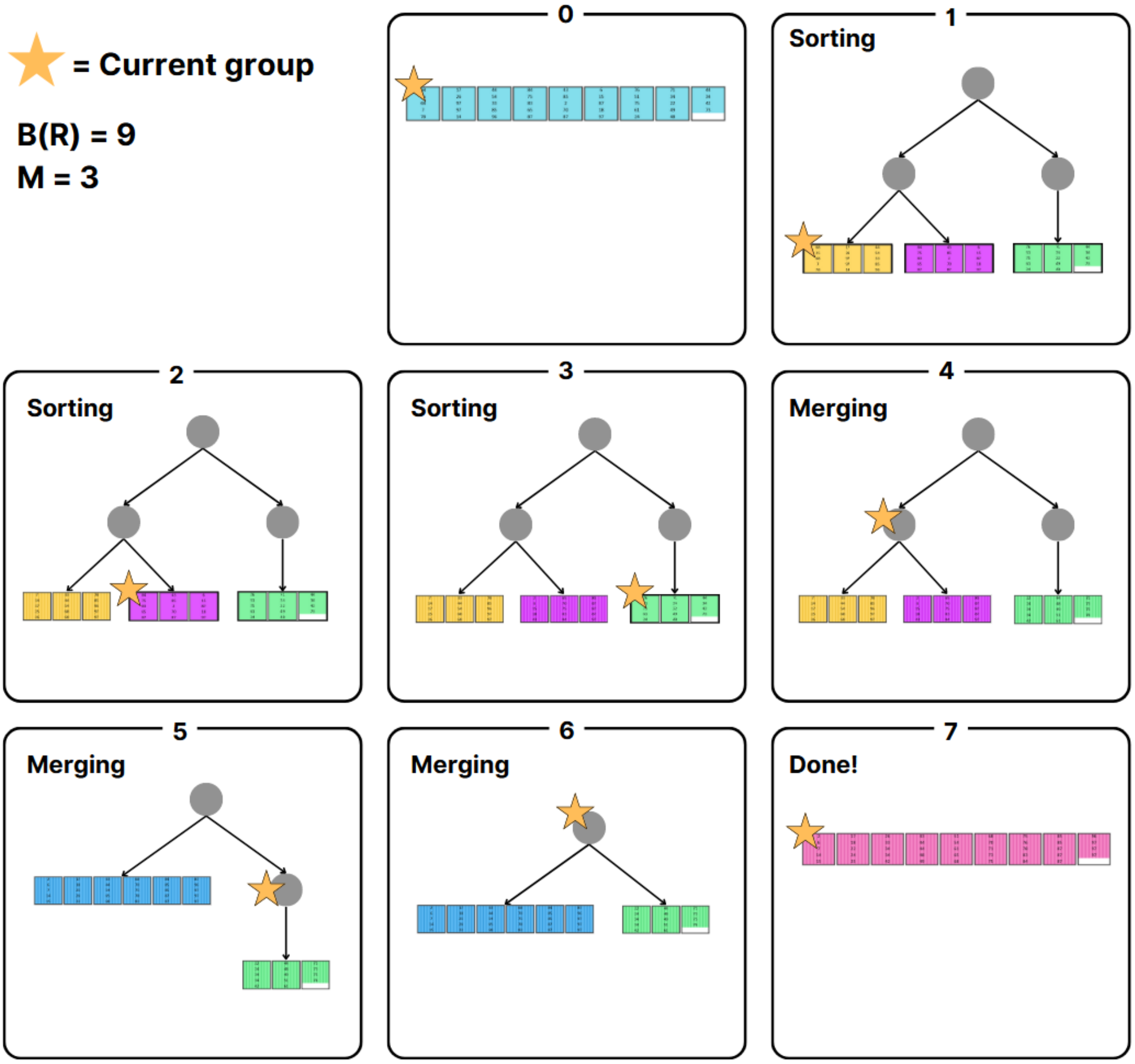
5 Implementation

5.1 Tree structure

The external multi-pass sorting algorithm at first considers the whole relation, but then it splits it into sub-runs that are small enough to fit inside the buffer. Then it merges the sorted runs together, producing bigger sorted runs, until the whole relation is sorted.

Because of this "splitting and merging" behavior of the algorithm, the best way to represent the relation is through a **tree structure**.

So in our application the relation is a tree, and we defined an important variable that we called "**current group**", that in any moment corresponds to a specific node of this tree. Graphically, it's the group that is highlighted with the black frame.



The picture above shows an execution of the algorithm, showing how the tree structure starts from a single node, then it expands, and then gradually contracts until there is again one node.

Initially (state 0) the current group is the root, which corresponds to the whole relation. The relation doesn't fit inside the buffer, so the tree structure is created (state 1). The tree is created using these criteria:

- For the **sorting** phase to work, all **the leaves must have at most M pages**.
- For the **merging** phase to work, **the fan-out** of the nodes in all the layers except for the leaf layer **must be at most M-1**.

Now the current group is the first sub-run, which can be sorted because it fits inside the buffer (state 2). The second and third sub-run are also individually sorted (states 3 and 4). Then the current group becomes the parent node of the first two sub-runs, and the children are merged together (state 5). The algorithm proceeds in this way until the tree has again only one node (state 7).

5.2 Application states

To show the execution of the algorithm in a way that could be understood by the users, we needed a way to pause the execution between one step and the next, and a way to resume the execution from where it was left off.

So we decided to define some **states**, which represent points in which the execution of the algorithm can be interrupted.

The application states are:

- **Start**: It's the initial state, before the execution of the algorithm begins.
- **RunToSort**: The current run must be sorted.
- **RunInBuffer**: The current run has been written in the buffer and must be sorted.
- **BufferSorted**: The content of the buffer has been sorted, and must be written back to secondary storage.
- **RunSorted**: The current run is sorted and has been written back in the relation.
- **RunsToMerge**: The current run is composed of sorted sub-runs that must be merged together.

- **RunsMerged**: The current run is composed of sorted sub-runs that have been merged together.
- **ChildrenInBuffer**: One page of each sub-run ("child") of the current group has been loaded in the buffer. Their content must be written, in the correct order, inside the output frame.
- **OneEmptyFrameInBuffer**: While doing the merging, a frame of the buffer has become empty. A new page of the corresponding sub-run must be loaded, otherwise the execution can't proceed correctly.
- **OutputFrameFullMerging**: While doing the merging, the output frame has become full. It must be written back to secondary storage and emptied.
- **Finish**: The execution has ended, the whole relation is sorted.

Then, we had to define how and under which conditions the application should pass from one state to another.

We did this by defining a **flow chart**, that can be seen in the "States Flow Chart" section.

In the chart, the green squares with rounded corners represent the states of the application (plus "Start" and "Finish" which are in red). The purple rhombuses represent checks that the application does to know which state it should transition to. And finally, the yellow squares explain the side-effects associated to the transitions.

Thus, at any given time the application is in one of the listed states. Whenever a user presses the "Next" button, the application does the necessary checks and actions specified by the flow chart and then moves to a new state, showing an animation.

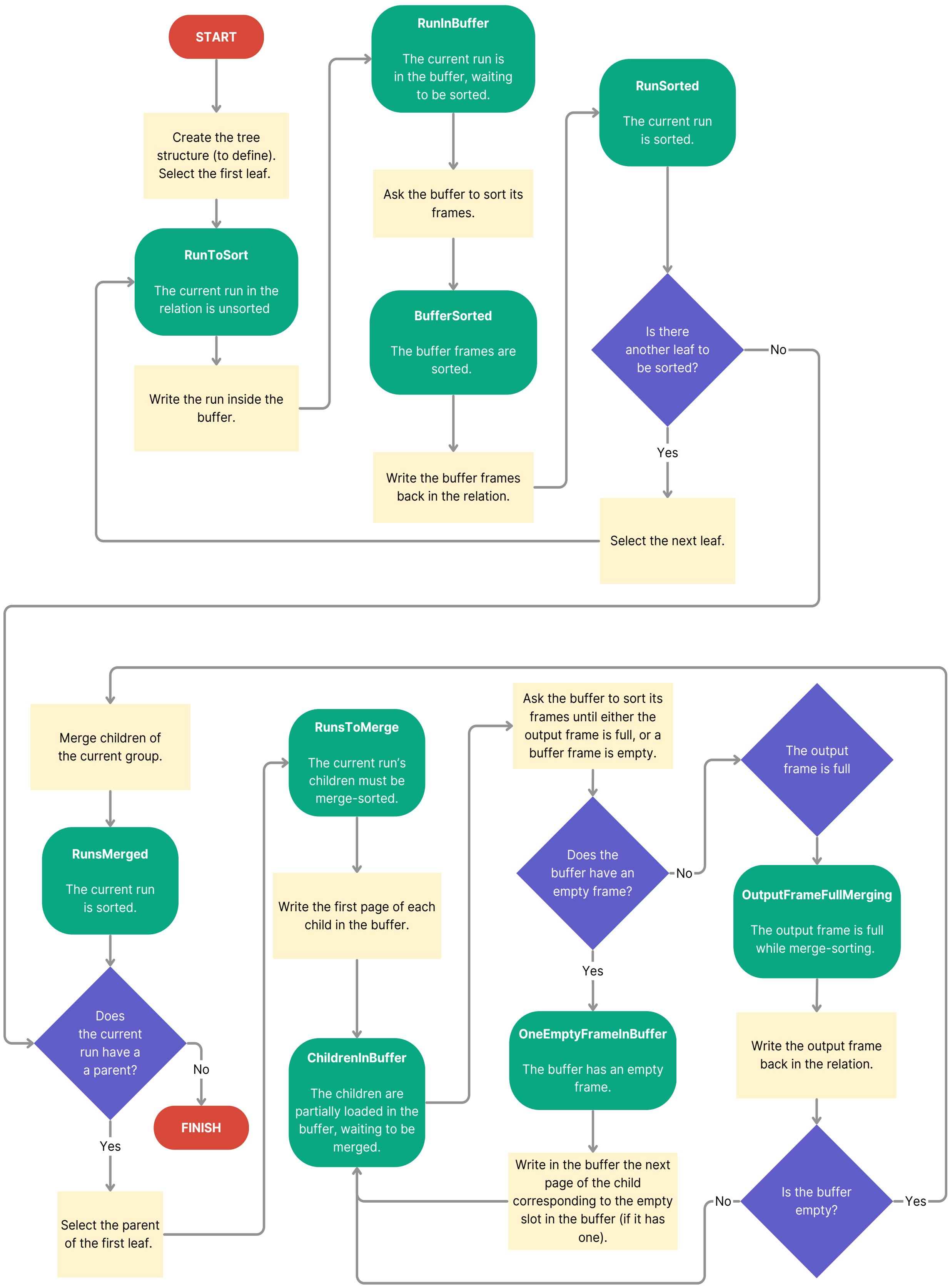
5.3 Undo operation

Undo is managed using an array. Before executing each state change, a function is pushed onto this array. When an undo is performed, the last function is obtained from the array using pop, and executing this function allows the system to return to the previous state.

This function is different for each state change: in fact, it is not necessary to save the entire state to restore it. It is sufficient to define a function that does the exact opposite of what was done to change the state, which most of the time requires only a few data as input.

So the most difficult part was writing these functions that perform the opposite of the state change functions. However, this approach allowed for memory savings due to the smaller amount of data that needs to be saved to restore previous states.

6 States Flow Chart


START
Create the tree structure (to define). Select the first leaf.
RunToSort
The current run in the relation is unsorted
Write the run inside the buffer.
RunInBuffer
The current run is in the buffer, waiting to be sorted.
Ask the buffer to sort its frames.
BufferSorted
The buffer frames are sorted.
Write the buffer frames back in the relation.
RunSorted
The current run is sorted.
Is there another leaf to be sorted?
No
Yes
Select the next leaf.
Merge children of the current group.
RunsMerged
The current run is sorted.
Does the current run have a parent?
No
FINISH
Yes
Select the parent of the first leaf.
RunsToMerge
The current run's children must be merge-sorted.
Write the first page of each child in the buffer.
ChildrenInBuffer
The children are partially loaded in the buffer, waiting to be merged.
Ask the buffer to sort its frames until either the output frame is full, or a buffer frame is empty.
Does the buffer have an empty frame?
No
Yes
OneEmptyFrameInBuffer
The buffer has an empty frame.
Write in the buffer the next page of the child corresponding to the empty slot in the buffer (if it has one).
The output frame is full
OutputFrameFullMerging
The output frame is full while merge-sorting.
Write the output frame back in the relation.
Is the buffer empty?
No
Yes