External memory merge-sort

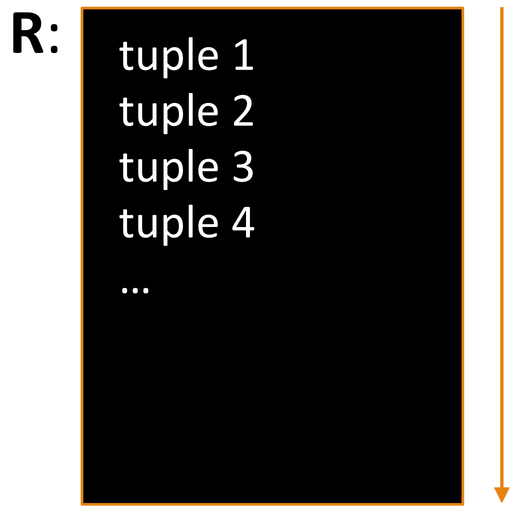
THIS ONE IS NOT REQUIRED FOR THE EXAM!

Overview over this video

This video (which is not required for the exam), we will cover external memory merge sort

Reminder: Computing $\sigma_{condition}(R)$

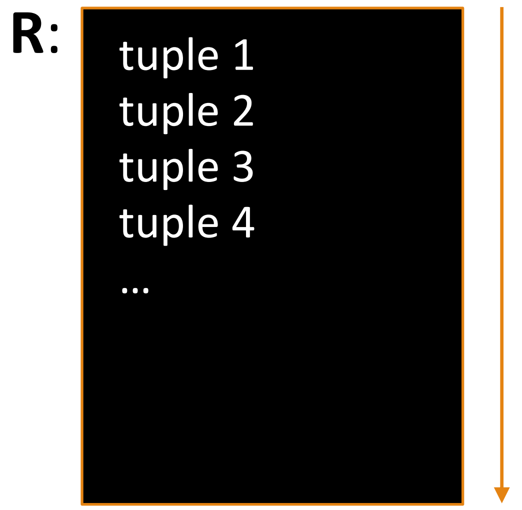
Basic procedure:



```
for each tuple t in R:
    if t satisfies condition:
        output t
```

Reminder: Computing $\sigma_{condition}(R)$

Basic procedure:

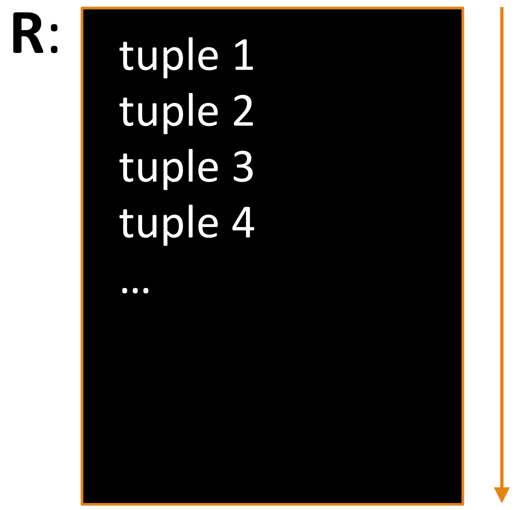


Needs to read the entire relation

```
for each tuple t in R:
    if t satisfies condition:
        output t
```

Reminder: Computing $\sigma_{\text{condition}}(R)$

Basic procedure:



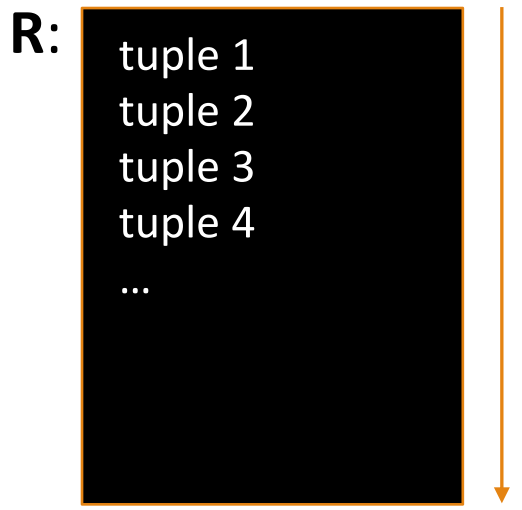
```
for each tuple t in R:
    if t satisfies condition:
        output t
```

Needs to read the entire relation

Can this be done faster?

Reminder: Computing $\sigma_{condition}(R)$

Basic procedure:



```
for each tuple t in R:
    if t satisfies condition:
        output t
```

Needs to read the entire relation

Can this be done faster?

Yes, sometimes!

Example

$\sigma_{\text{programme='G401'}}(\text{Students})$

Students

id	name	programme
…	…	….
1234	Anna	G401
2345	Ben	G701
3456	Chloe	G401
4567	Dave	G401
…	…	…

Example

$\sigma_{\text{programme='G401'}}(\text{Students})$

Students

id	name	programme
…	…	….
1234	Anna	G401
2345	Ben	G701
3456	Chloe	G401
4567	Dave	G401
…	…	…

Selection can be performed faster if we know **where to find the rows for 'G401'**

Example

$\sigma_{\text{programme='G401'}}(\text{Students})$

Students

id	name	programme
…	…	….
1234	Anna	G401
2345	Ben	G701
3456	Chloe	G401
4567	Dave	G401
…	…	…

Selection can be performed faster if we know **where to find the rows for ‘G401’**

Two solutions: **sorting** & **index**

Example

$\sigma_{\text{programme}=\text{'G401'}}(\text{Students})$

Students

id	name	programme
…	…	….
1234	Anna	G401
2345	Ben	G701
3456	Chloe	G401
4567	Dave	G401
…	…	…

Selection can be performed faster if we know **where to find the rows for ‘G401’**

Two solutions: **sorting** & **index**

This video

Faster Joins With Sorting

(from video "Faster Joins With Sorting")

Sort Join Algorithm:

Compute R $\bowtie_{A=B}$ S:

1. Sort **R** on **A**
2. Sort **S** on **B**
3. Merge the sorted **R** and **S**

Running time: $O(|R| \times \log_2 |R|)$

Running time: $O(|S| \times \log_2 |S|)$

Running time: $O(\text{size of output})$

Having a run time depending on the size of output is called output sensitive

Typical running time: $O(|\mathbf{R}|\log_2|\mathbf{R}| + |\mathbf{S}|\log_2|\mathbf{S}|)$

- If not "too many" values in **A** occur multiple times
- E.g., this is the case if **A** is a key

Typically much faster than Nested Loop Join

- Same time in the worst case, because output can have size up to $|\mathbf{R}| \times |\mathbf{S}|$

Faster Joins With Sorting

(from video "Faster Joins With Sorting")

Sort Join Algorithm:

Compute R $\bowtie_{A=B}$ S:

1. Sort **R** on **A**
2. Sort **S** on **B**
3. Merge the sorted **R** and **S**

Running time: $O(|R| \times \log_2 |R|)$

Want to do that faster!

Running time: $O(|S| \times \log_2 |S|)$

Running time: $O(\text{size of output})$

Having a run time depending on the size of output is called output sensitive

Typical running time: $O(|\mathbf{R}|\log_2|\mathbf{R}| + |\mathbf{S}|\log_2|\mathbf{S}|)$

- If not "too many" values in **A** occur multiple times
- E.g., this is the case if **A** is a key

Typically much faster than Nested Loop Join

- Same time in the worst case, because output can have size up to $|\mathbf{R}| \times |\mathbf{S}|$

Outside databases

Many problems are much easier then the data is sorted and in general, a significant fraction of all computing time is spend on sorting

- Was estimated to be around 25% back in the 60s
  - Found some claims that it is still in the range 25%-50%, but it is hard to verify

Outside databases

Many problems are much easier then the data is sorted and in general, a significant fraction of all computing time is spend on sorting

- Was estimated to be around 25% back in the 60s
  - Found some claims that it is still in the range 25%-50%, but it is hard to verify

Some problems were you need sorting are “small” in that they fit in main memory

Outside databases

Many problems are much easier then the data is sorted and in general, a significant fraction of all computing time is spend on sorting

- Was estimated to be around 25% back in the 60s
  - Found some claims that it is still in the range 25%-50%, but it is hard to verify

Some problems were you need sorting are “small” in that they fit in main memory

Some ain’t

Understanding the issue solved

To find one record on hard disk (with very fast "normal" drive of 15,000 rpm):

- 2.00 **milli-seconds** or $2 \cdot 10^6$ **nano-seconds**

Understanding the issue solved

To find one record on hard disk (with very fast “normal” drive of 15,000 rpm):

- 2.00 **milli-seconds** or $2 \cdot 10^6$ **nano-seconds**

To find one record on SSD:

- 0.031 **milli-seconds** or $3.1 \cdot 10^4$ **nano-seconds**

Understanding the issue solved

To find one record on hard disk (with very fast “normal” drive of 15,000 rpm):

- 2.00 **milli-seconds** or $2 \cdot 10^6$ **nano-seconds**

To find one record on SSD:

- 0.031 **milli-seconds** or $3.1 \cdot 10^4$ **nano-seconds**

To find one record in RAM (with relative normal DDR4-2666 RAM):

- 13 **nano-seconds**

Understanding the issue solved

To find one record on hard disk (with very fast "normal" drive of 15,000 rpm):

- 2.00 **milli-seconds** or $2 \cdot 10^6$ **nano-seconds**

SLOW!!!

To find one record on SSD:

- 0.031 **milli-seconds** or $3.1 \cdot 10^4$ **nano-seconds**

slow

To find one record in RAM (with relative normal DDR4-2666 RAM):

- 13 **nano-seconds**

fast

Merge Sort

(Internal memory) merge sort:

```
**Divide** input in two parts $P_1$, $P_2$
**Sort** $P_1$ & $P_2$ **recursively**
**While** $P_1$ or $P_2$ is not empty:
    Add smallest remaining
    element from $P_1$ or $P_2$
    to output
```

Merge Sort

(Internal memory) merge sort:

```
**Divide** input in two parts $P_1$, $P_2$
**Sort** $P_1$ & $P_2$ **recursively**
**While** $P_1$ or $P_2$ is not empty:
    Add smallest remaining
    element from $P_1$ or $P_2$
    to output
```

Merge

External Merge Sort

External merge sort:

```
**Divide** input in **M** parts $P_1, P_2, \ldots, P_M$
**Sort** $P_1, P_2, \ldots, P_M$ **recursively**
**While** not all $P_i$ are empty:
    Add smallest remaining
    element from any part $P_i$ to
    output
```

External Merge Sort

External merge sort:

Merge

```
**Divide** input in **M** parts $P_1, P_2, \ldots, P_M$
**Sort** $P_1, P_2, \ldots, P_M$ **recursively**
**While** not all $P_i$ are empty:
    Add smallest remaining
    element from any part $P_i$ to
    output
```

External Merge Sort

External merge sort:

Number of disk blocks that fit in RAM

Merge

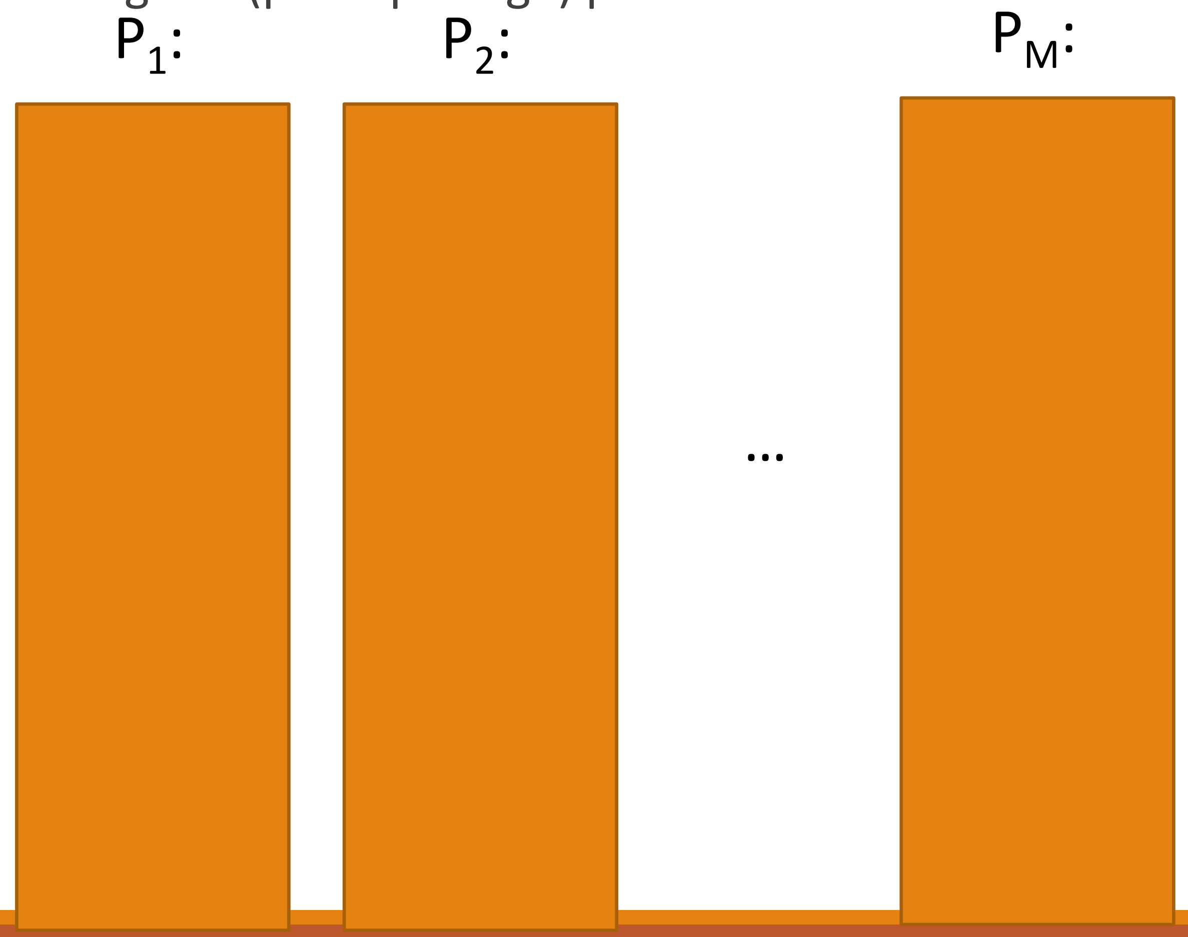
```
**Divide** input in **M** parts $P_1, P_2, \ldots, P_M$
**Sort** $P_1, P_2, \ldots, P_M$ **recursively**
**While** not all $P_i$ are empty:
    Add smallest remaining
    element from any part $P_i$ to
    output
```

Merge in more details

Goal: Must merge M (perhaps large) parts into one

Merge in more details

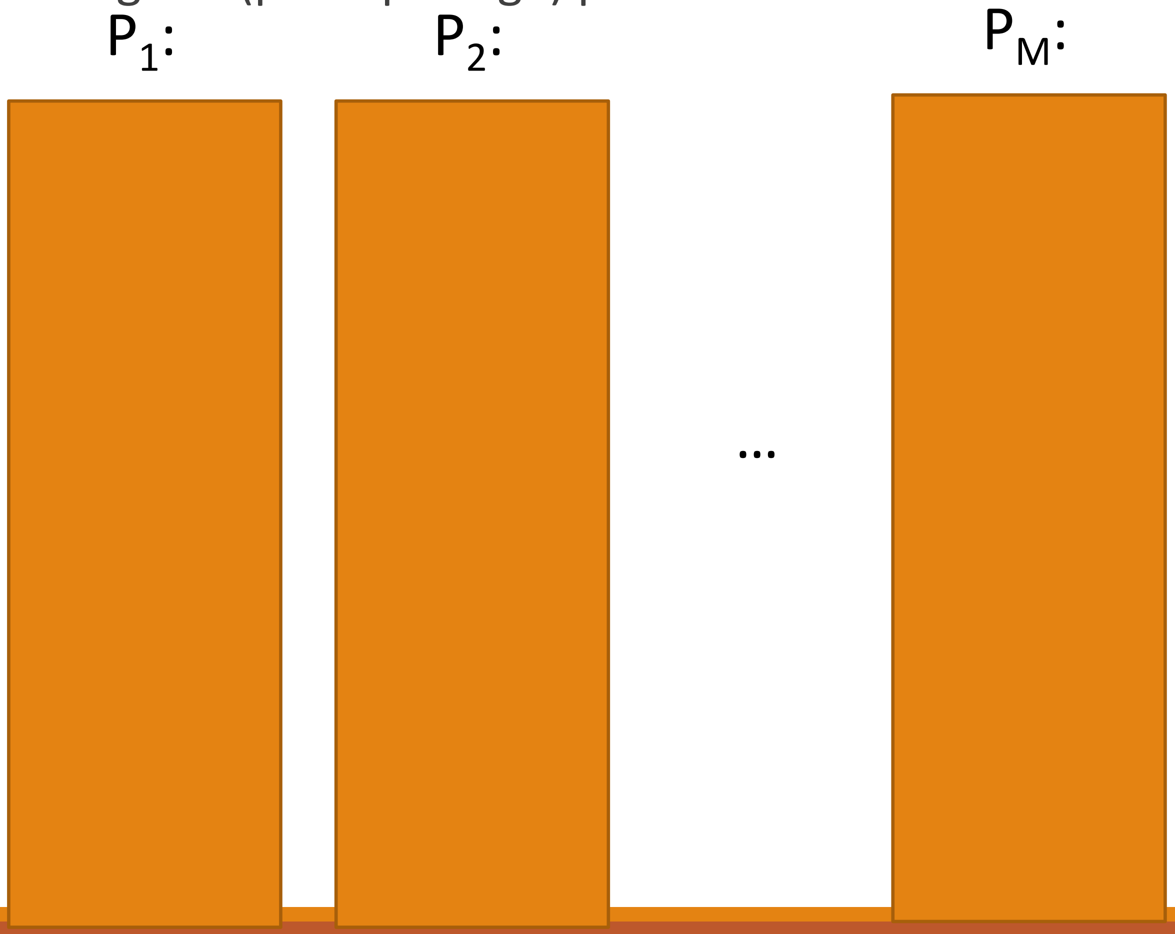
Goal: Must merge M (perhaps large) parts into one



Merge in more details

Goal: Must merge M (perhaps large) parts into one

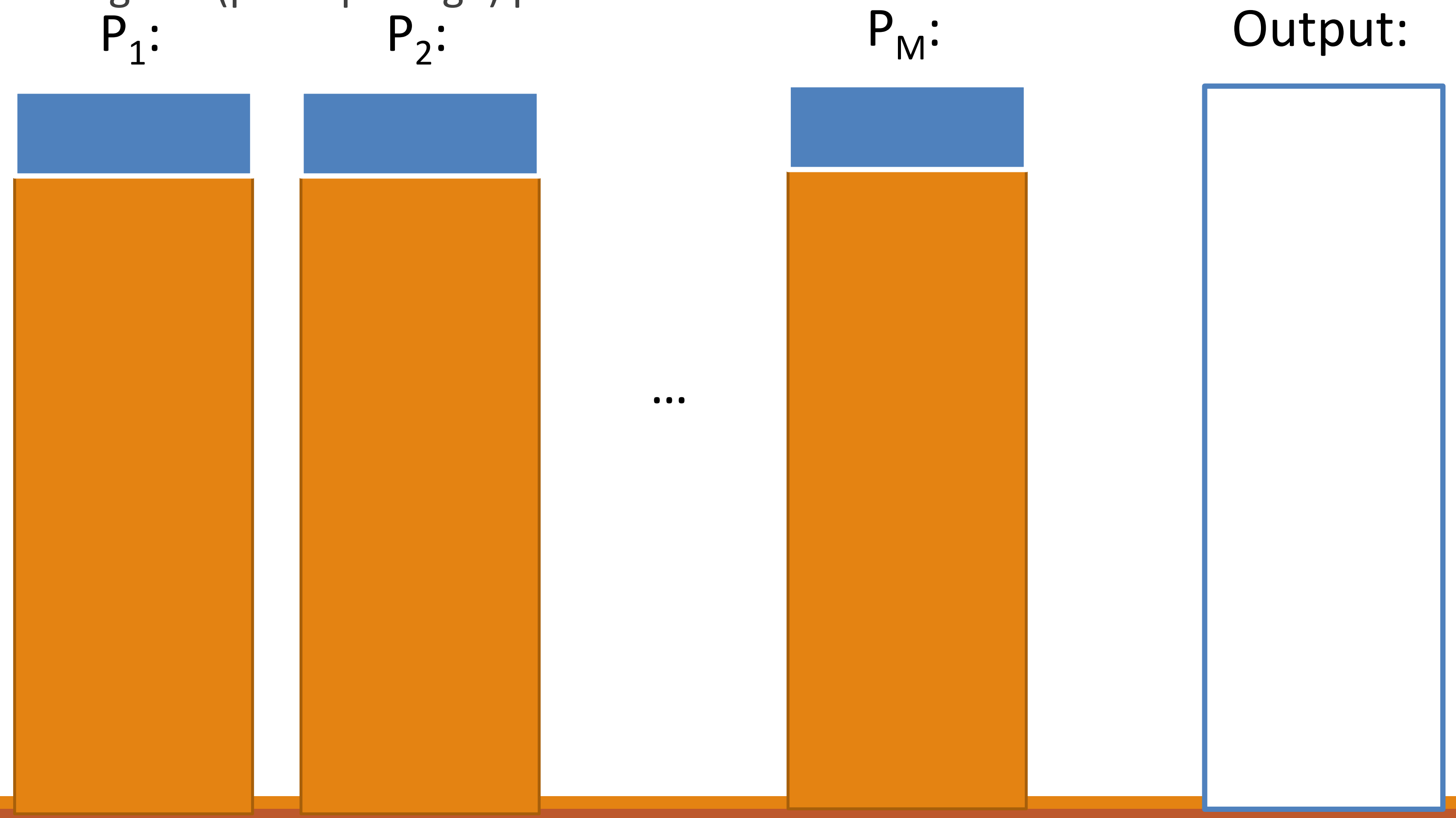
Colors:
Blue in RAM
Orange in disk
White does not exists yet



Merge in more details

Goal: Must merge M (perhaps large) parts into one

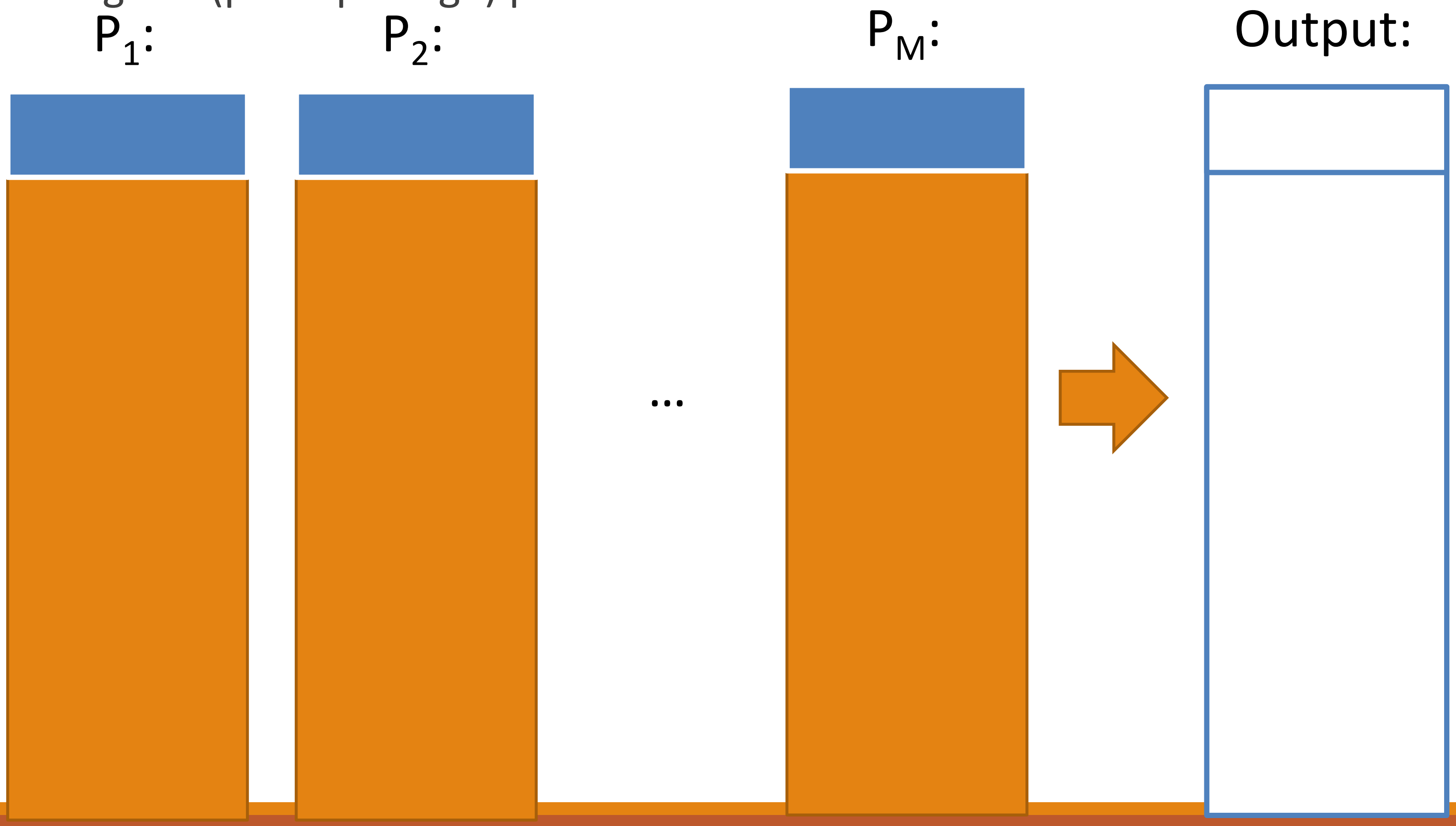
Colors:
Blue in RAM
Orange in disk
White does not exists yet



Merge in more details

Goal: Must merge M (perhaps large) parts into one

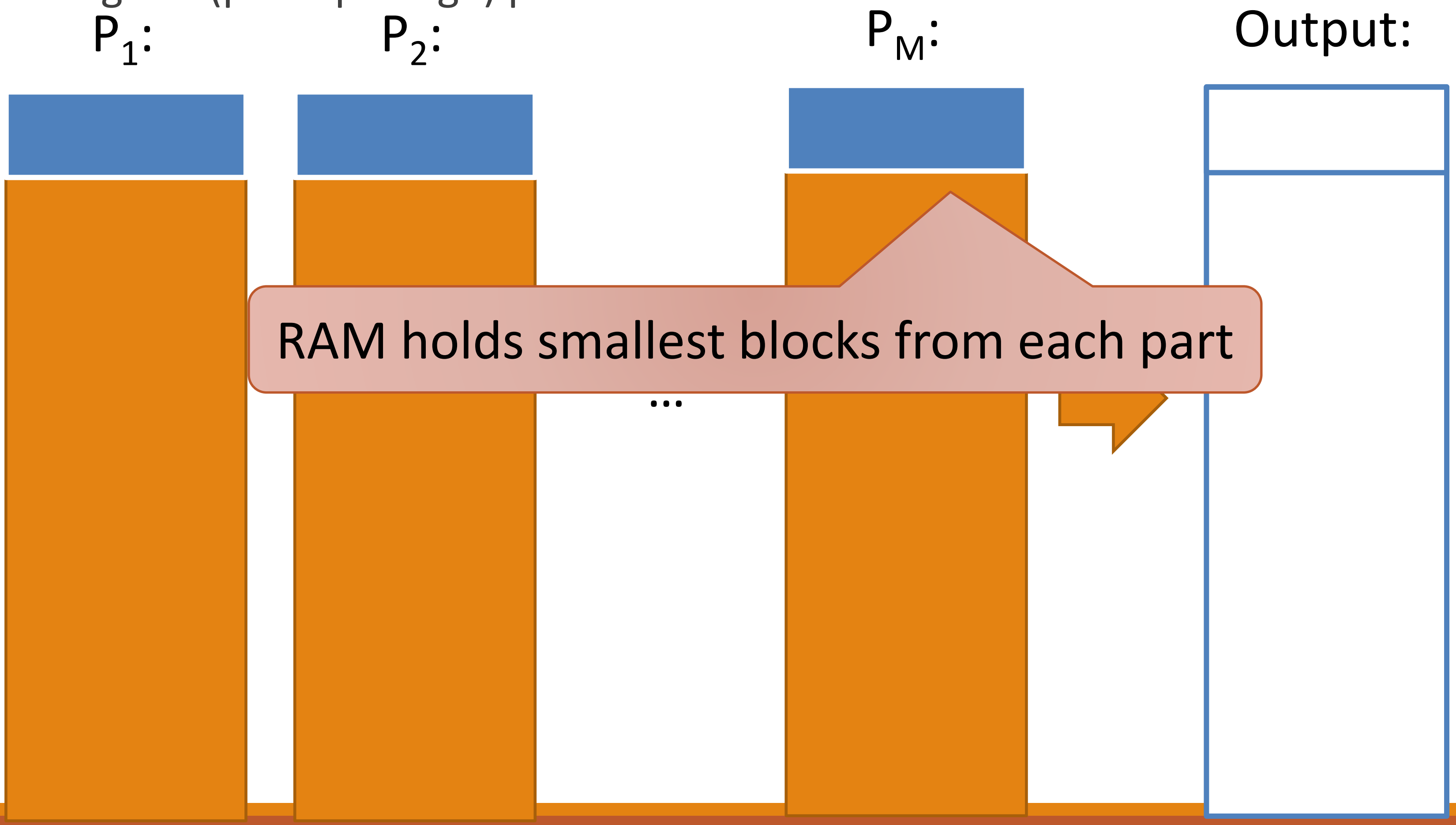
Colors:
Blue in RAM
Orange in disk
White does not exists yet



Merge in more details

Goal: Must merge M (perhaps large) parts into one

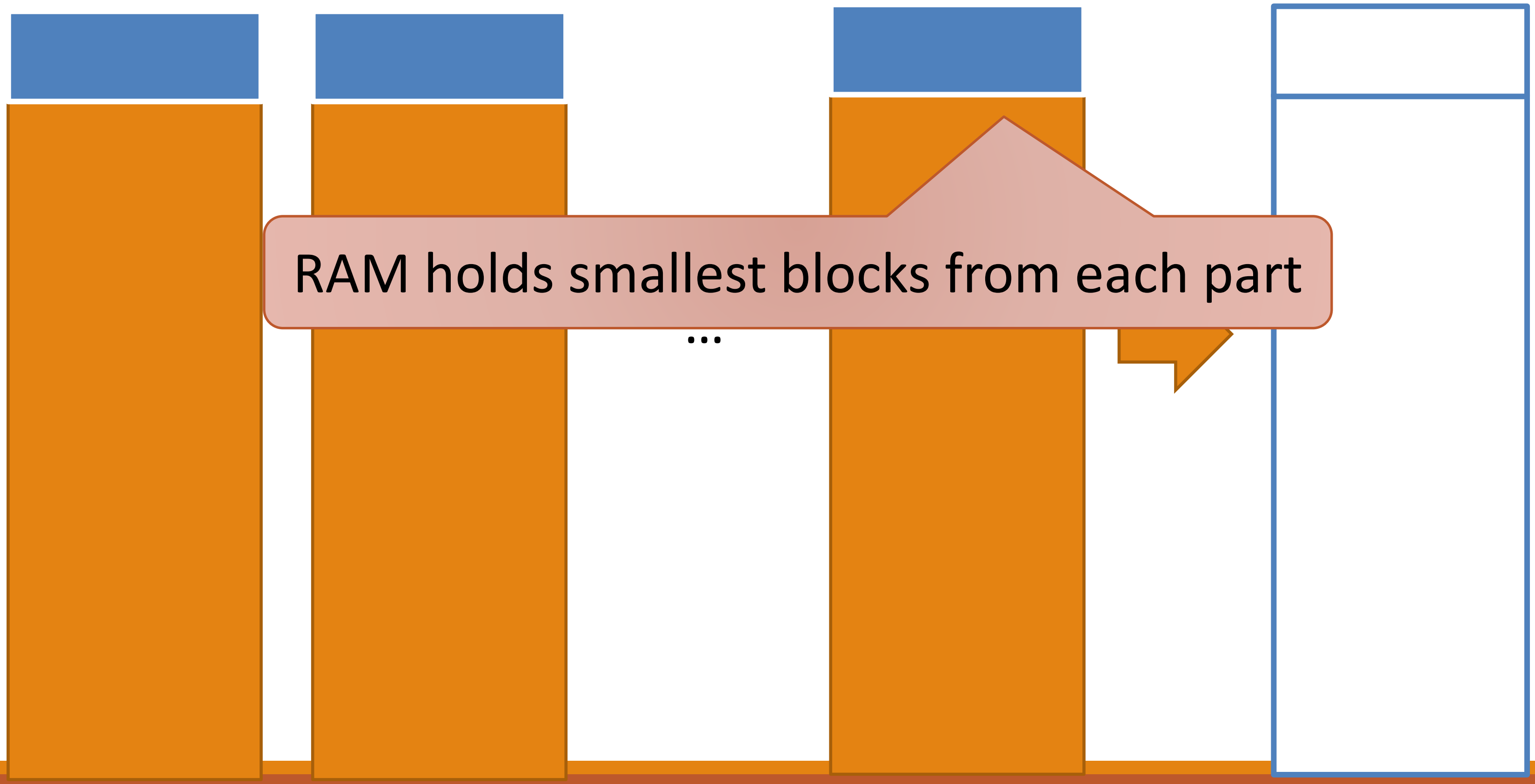
Colors:
Blue in RAM
Orange in disk
White does not exists yet



Merge in more details

Goal: Must merge M (perhaps large) parts into one

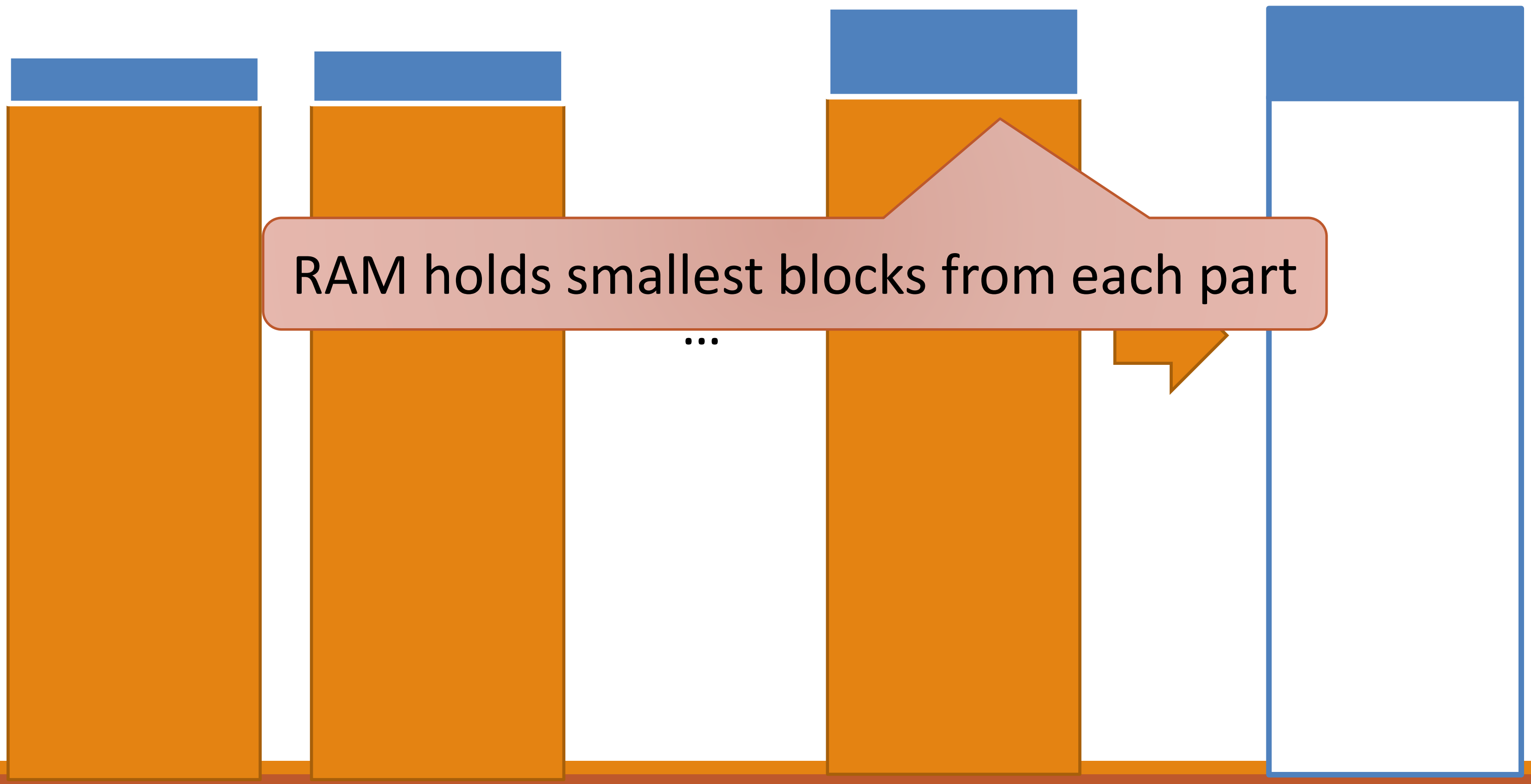
Colors:
Blue in RAM
Orange in disk
White does not exists yet



Merge in more details

Goal: Must merge M (perhaps large) parts into one

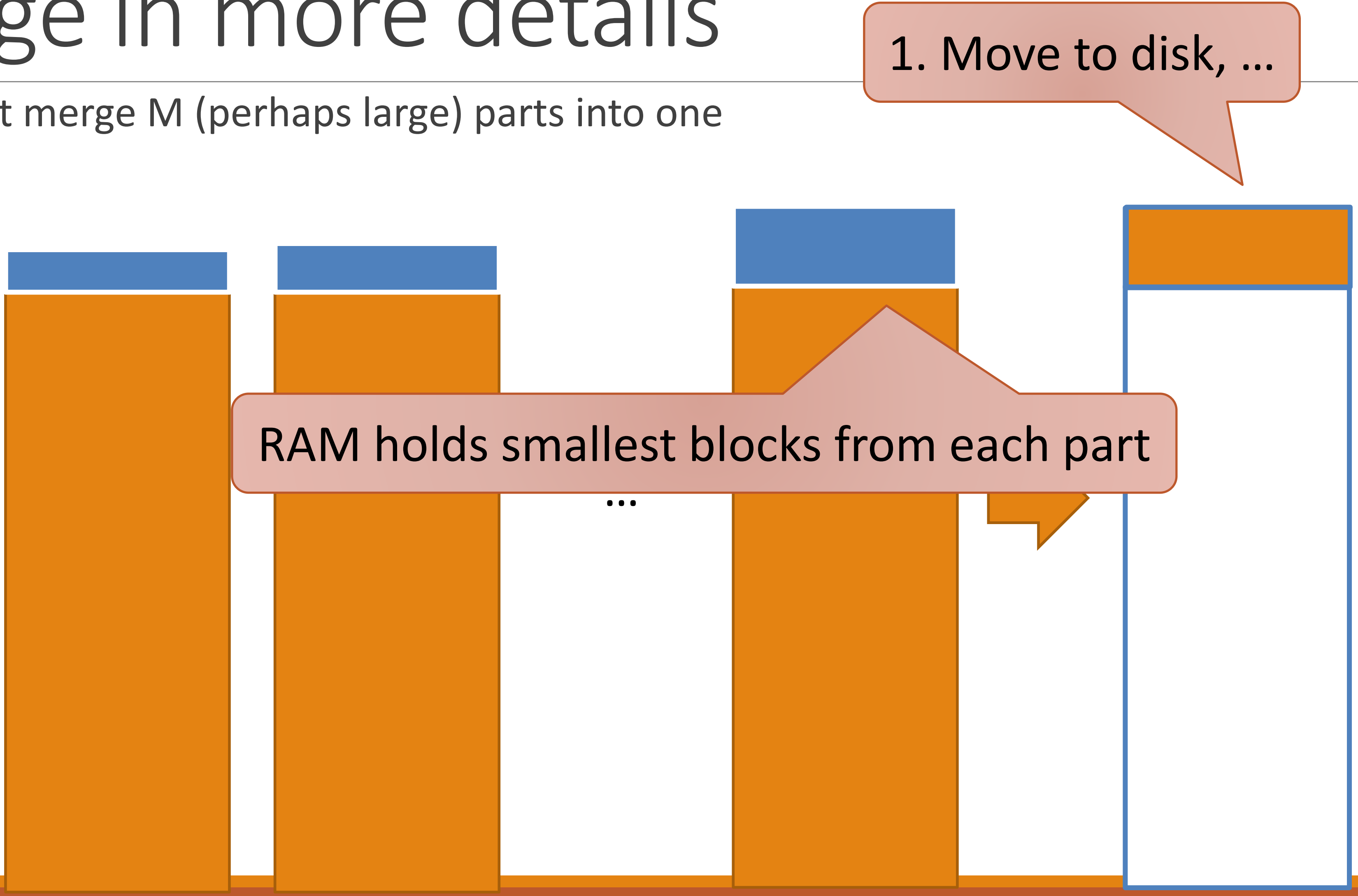
Colors:
Blue in RAM
Orange in disk
White does not exists yet



Merge in more details

Goal: Must merge M (perhaps large) parts into one

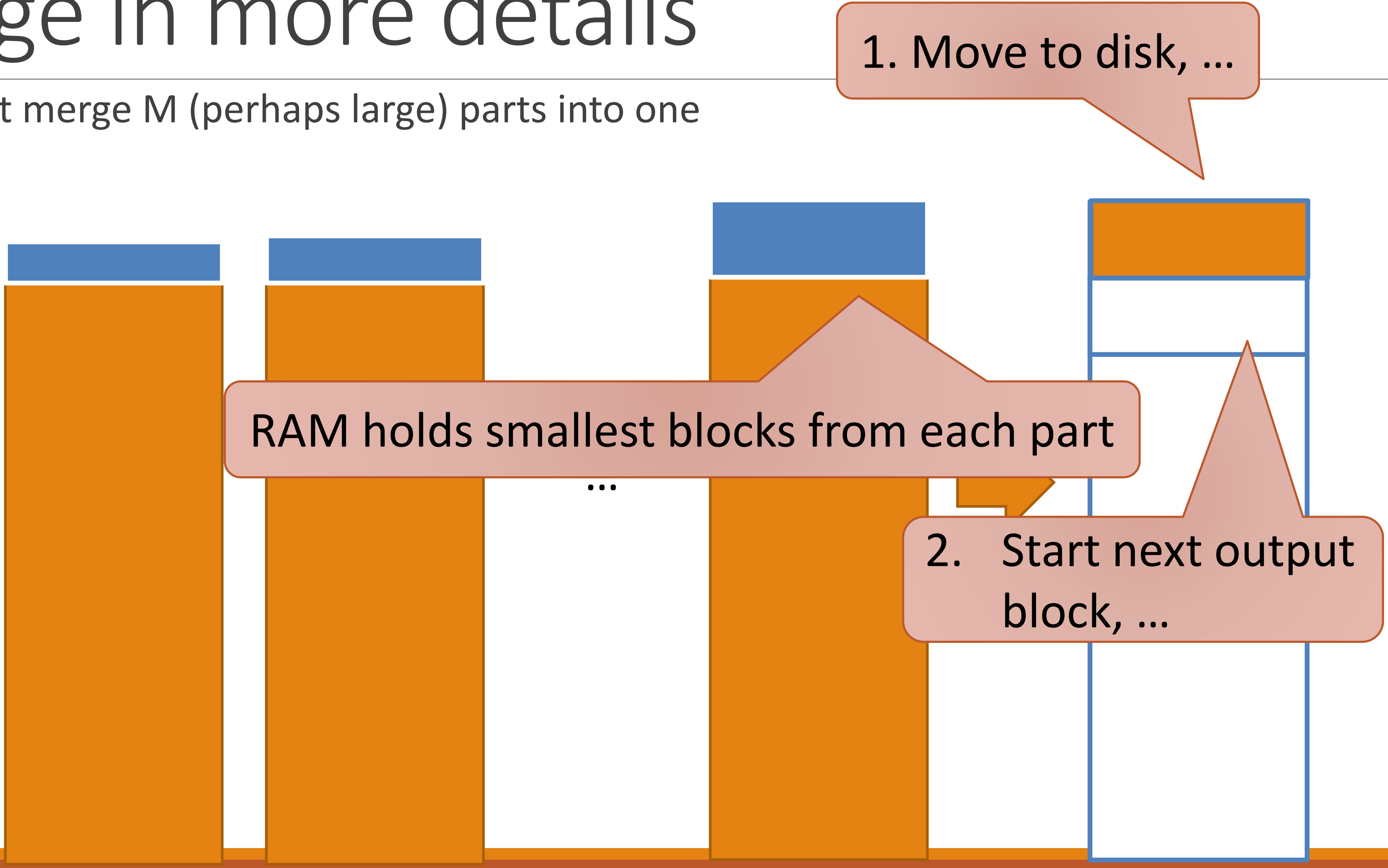
Colors:
Blue in RAM
Orange in disk
White does not exists yet



Merge in more details

Goal: Must merge M (perhaps large) parts into one

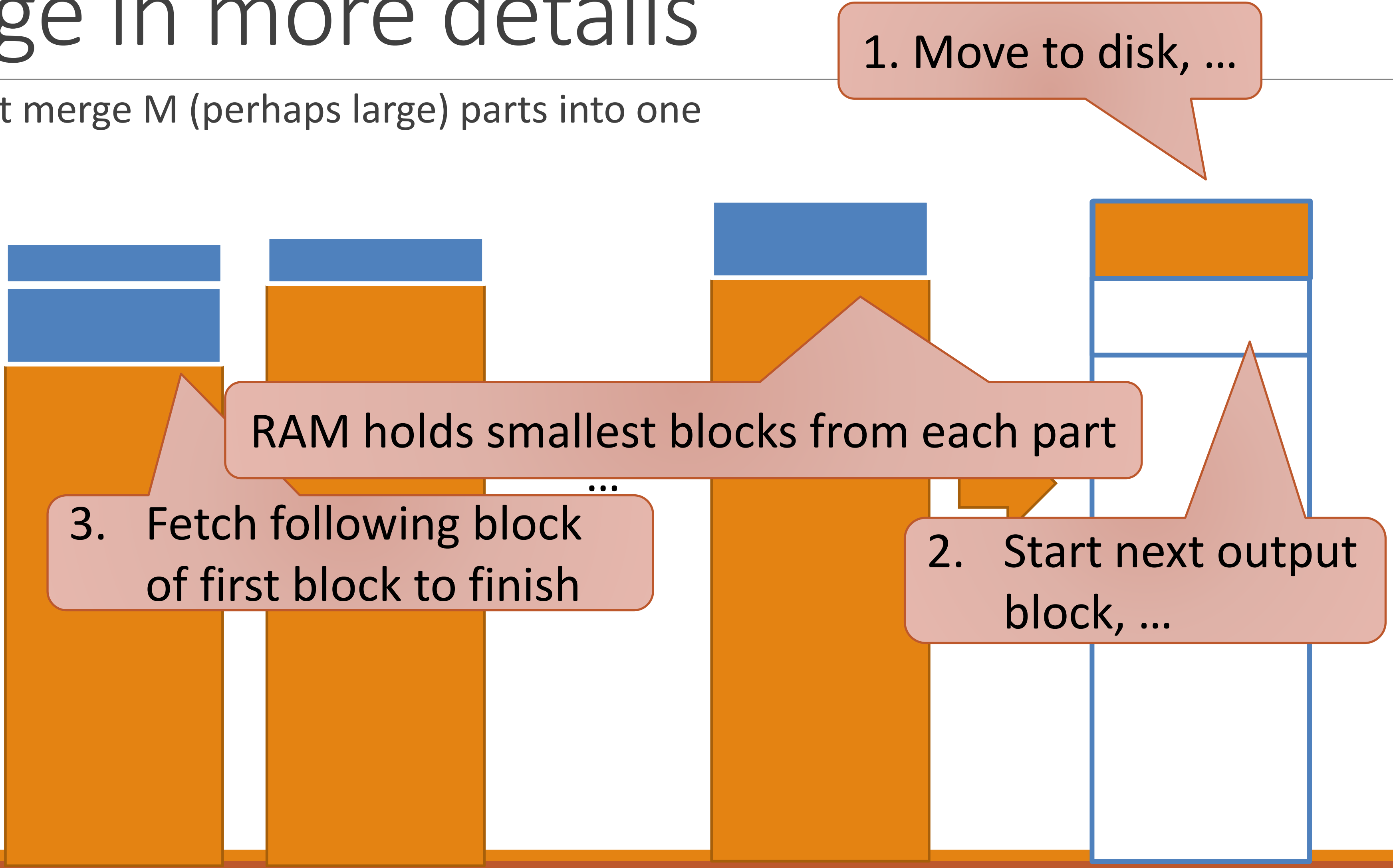
Colors:
Blue in RAM
Orange in disk
White does not exists yet


1. Move to disk, …
RAM holds smallest blocks from each part
…
2. Start next output block, …

Merge in more details

Goal: Must merge M (perhaps large) parts into one

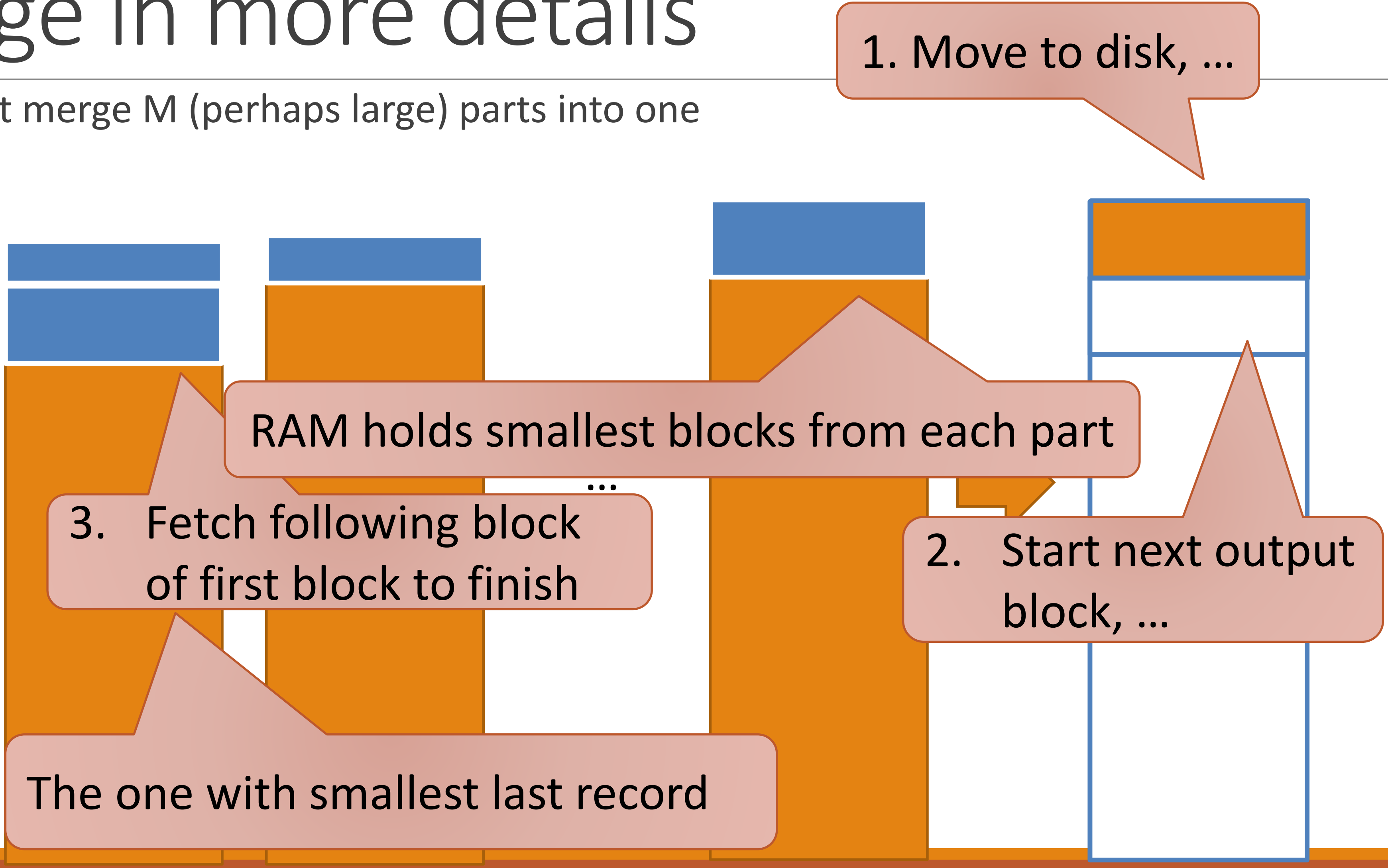
Colors:
Blue in RAM
Orange in disk
White does not exists yet


1. Move to disk, …
RAM holds smallest blocks from each part
…
3. Fetch following block of first block to finish
2. Start next output block, …

Merge in more details

Goal: Must merge M (perhaps large) parts into one

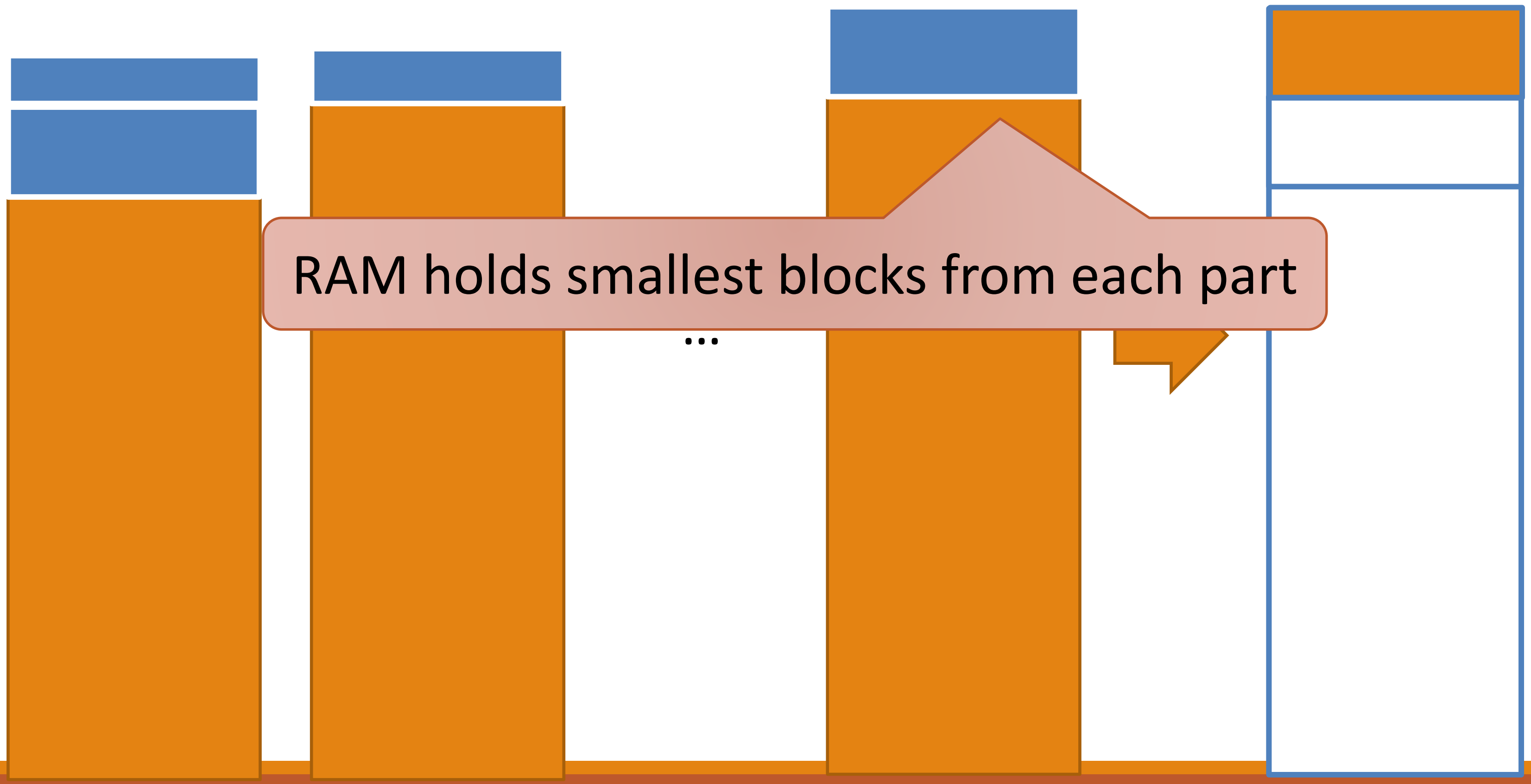
Colors:
Blue in RAM
Orange in disk
White does not exists yet


1. Move to disk, …
RAM holds smallest blocks from each part
…
3. Fetch following block of first block to finish
2. Start next output block, …
The one with smallest last record

Merge in more details

Goal: Must merge M (perhaps large) parts into one

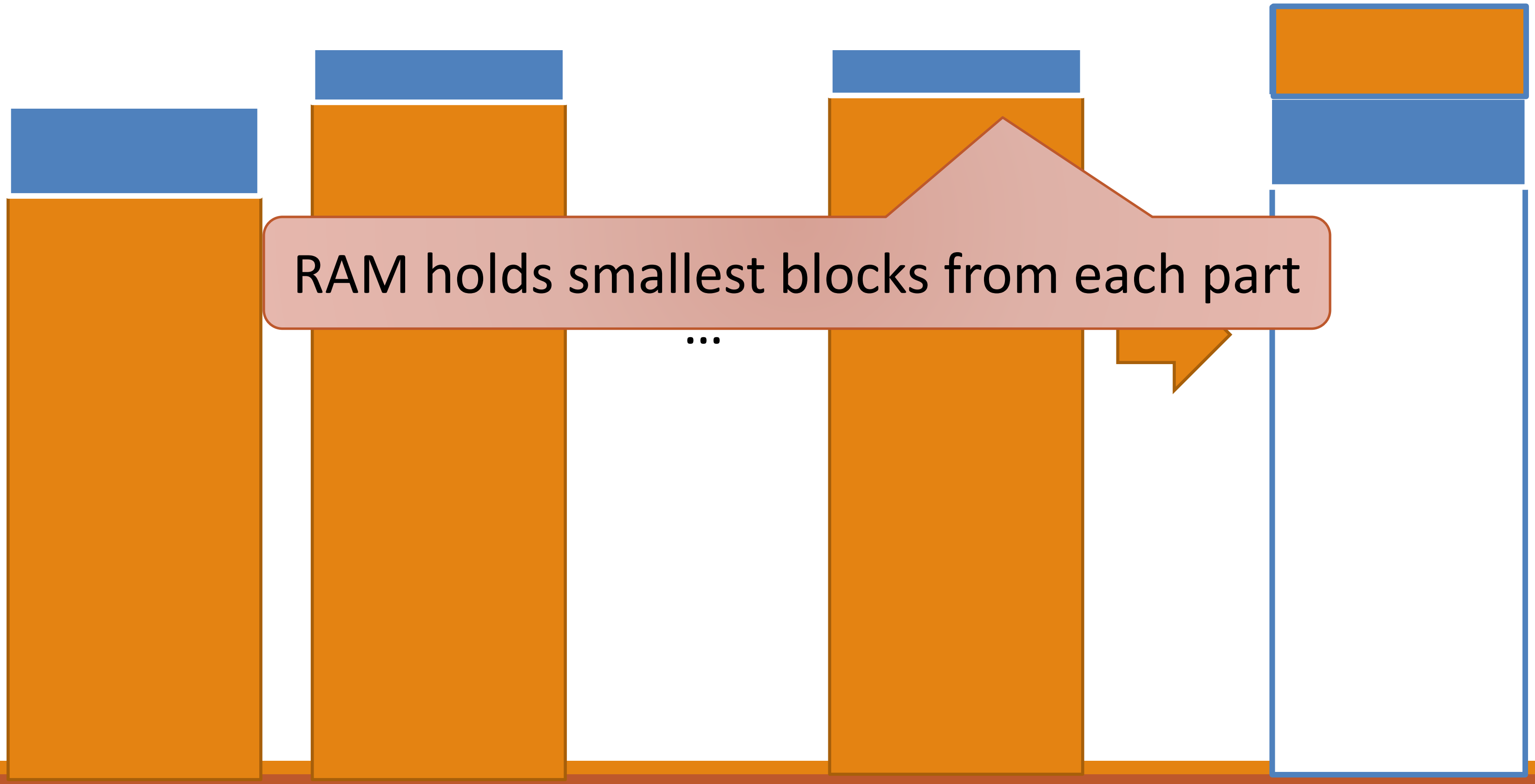
Colors:
Blue in RAM
Orange in disk
White does not exists yet



Merge in more details

Goal: Must merge M (perhaps large) parts into one

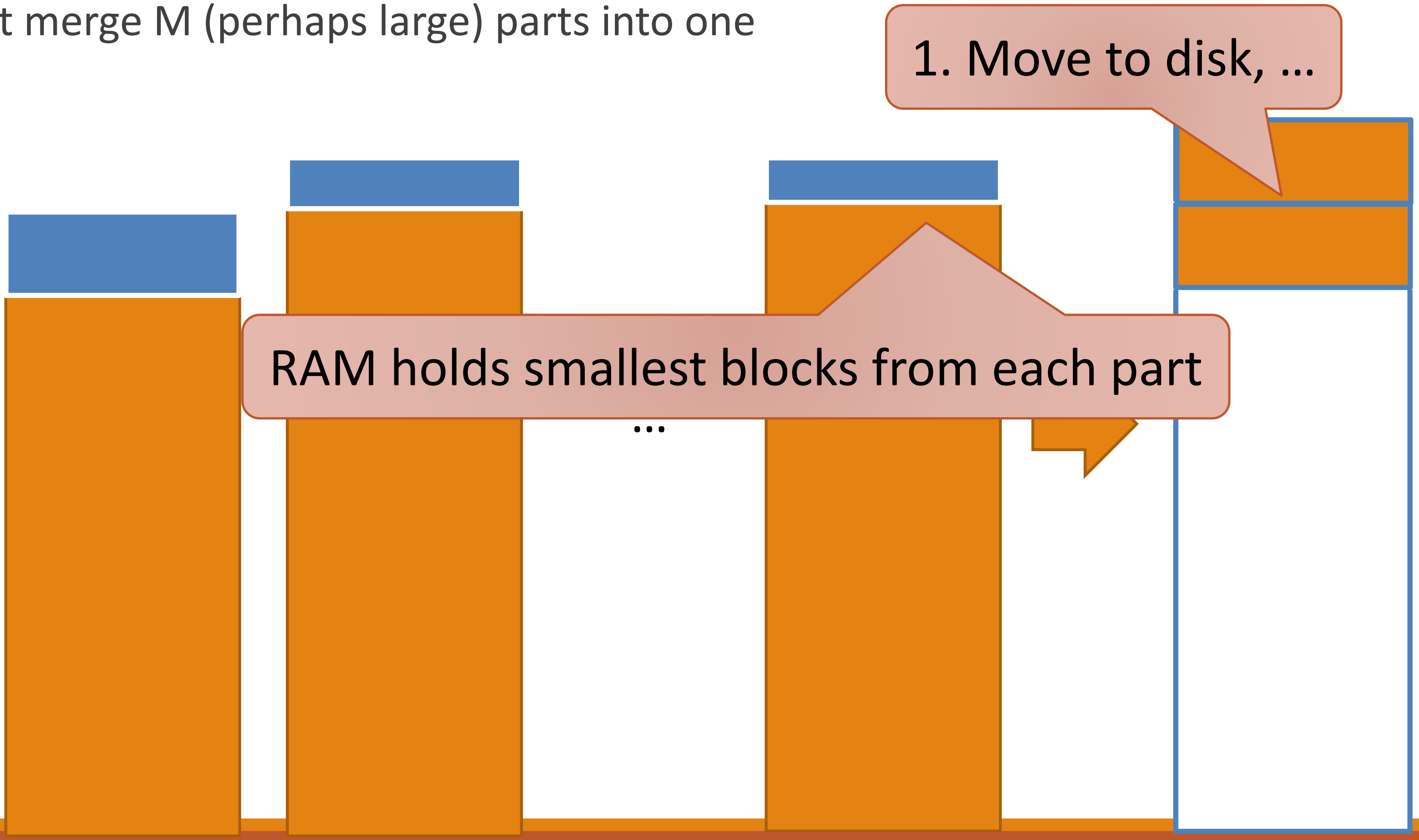
Colors:
Blue in RAM
Orange in disk
White does not exists yet



Merge in more details

Goal: Must merge M (perhaps large) parts into one

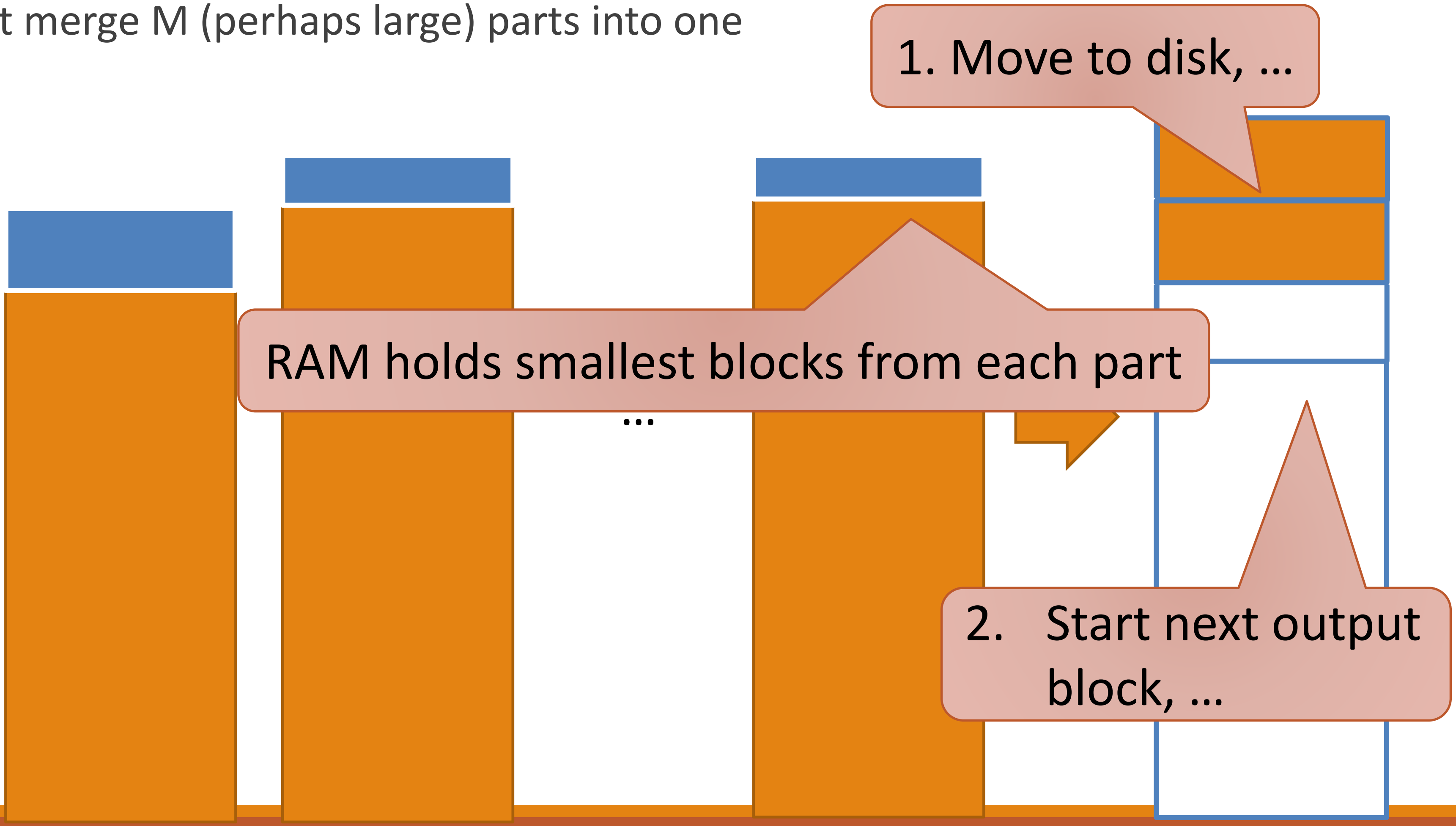
Colors:
Blue in RAM
Orange in disk
White does not exists yet



Merge in more details

Goal: Must merge M (perhaps large) parts into one

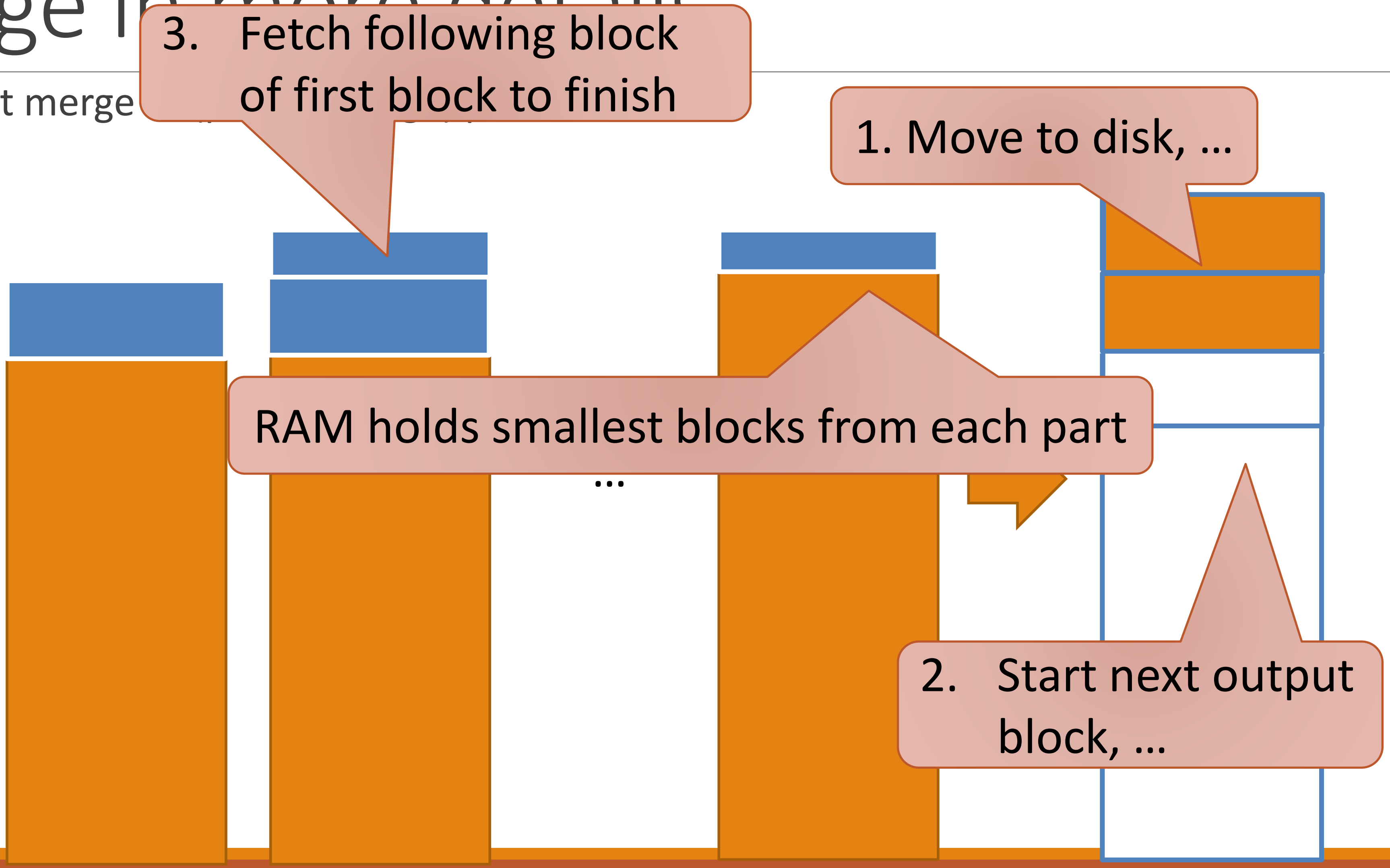
Colors:
Blue in RAM
Orange in disk
White does not exists yet


1. Move to disk, …
RAM holds smallest blocks from each part
…
2. Start next output block, …

Merge in more details

Goal: Must merge

Colors:
Blue in RAM
Orange in disk
White does not exists yet

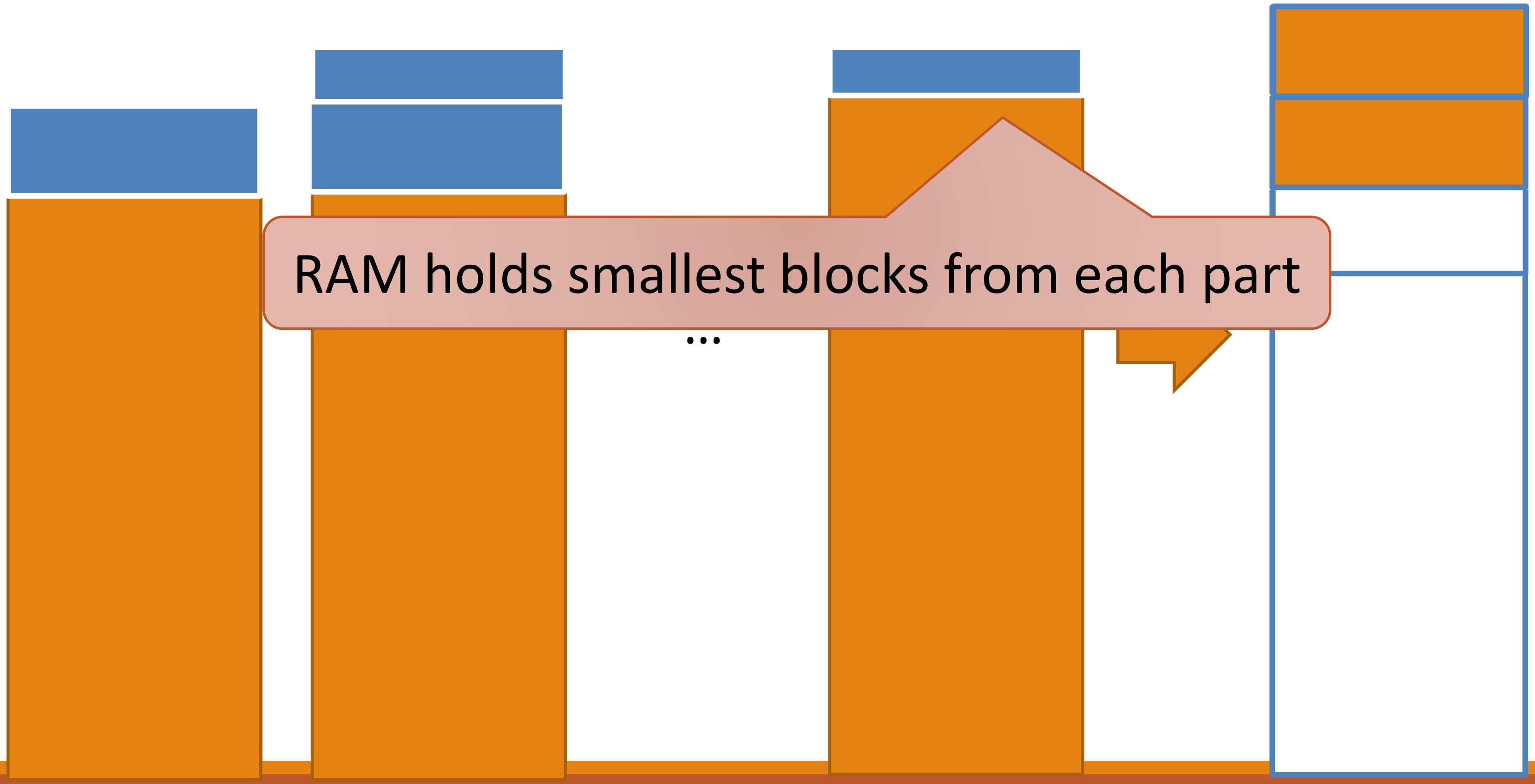


3. Fetch following block of first block to finish
1. Move to disk, …
RAM holds smallest blocks from each part
…
2. Start next output block, …

Merge in more details

Goal: Must merge M (perhaps large) parts into one

Colors:
Blue in RAM
Orange in disk
White does not exists yet



Analysis of External Merge Sort

External merge sort:

Merge

```
**Divide** input in **M** parts $P_1, P_2, \ldots, P_M$
**Sort** $P_1, P_2, \ldots, P_M$ **recursively**
**While** not all $P_i$ are empty:
    Add smallest remaining
    element from any part $P_i$ to
    output
```

Analysis of External Merge Sort

External merge sort:

```
**Divide** input in **M** parts $P_1, P_2, \ldots, P_M$
**Sort** $P_1, P_2, \ldots, P_M$ **recursively**
**While** not all $P_i$ are empty:
    Add smallest remaining
    element from any part $P_i$ to
    utput
```

Merge

On each level of recursion we scan through all blocks once = O(N/B) disk operations, where B is block size

Analy ort

We split in M buckets in each level of recursion until reminder is below MB (where it fits in RAM and can be sorted directly): $\log_M(N/(MB))$ levels

External merge sort:

Merge

```
**Divide** inp  in **M** parts $P_1, P_2, \ldots, P_M$
**Sort** $P_1, P_2, \ldots, P_M$ **recursively**
**While** not all $P_i$ are empty:
    Add smallest remaining
    element from any part $P_i$ to
    utput
```

On each level of recursion we scan through all blocks once = $O(N/B)$ disk operations, where B is block size

Analy ort

We split in M buckets in each level of recursion until reminder is below MB (where it fits in RAM and can be sorted directly): $\log_M(N/(MB))$ levels

External merge sort:

```
**Divide** inp  n **M** parts $P_1, P_2, …, P_M$
**Sort** $P_1, P_2, …, P_M$ **recursively**
**While** not all $P_i$ are empty:
    Add smallest remaining
    element from any part $P_i$ to
    utput
```

Total disk operations:
$O(N/B \log_M(N/(MB)))$

On each level of recursion we scan through all blocks once = O(N/B) disk operations, where B is block size

Comparison

In practice, since hard disks are slow:

- D)
- D is time for a disk operation

Quicksort (or other internal memory sorting algorithms) uses $O(N/B \log_2(N) \times D)$ time!

Common values for D, B and M:

- D was around $3.1 \cdot 10^4$ **nano-seconds** on SSDs and $2 \cdot 10^6$ **nano-seconds** on fast normal hard disks
- B is typically around 512-4096 bytes
- M is the size of your RAM divided by B, so say 8-32 GB RAM / 512-4096 bytes: $2.0 \cdot 10^6$ to $6.5 \cdot 10^7$

Comparison

In practice, since hard disks are slow:

- Running time is basically time spend on disk operations
- D)
- D is time for a disk operation

Quicksort (or other internal memory sorting algorithms) uses $O(N/B \log_2(N) \times D)$ time!

Common values for D, B and M:

- D was around $3.1 \cdot 10^4$ **nano-seconds** on SSDs and $2 \cdot 10^6$ **nano-seconds** on fast normal hard disks
- B is typically around 512-4096 bytes
- M is the size of your RAM divided by B, so say 8-32 GB RAM / 512-4096 bytes: $2.0 \cdot 10^6$ to $6.5 \cdot 10^7$

Comparison

D)

In practice, since hard disks are slow:

- Running time is basically time spend on disk operations
- Thus, time is $O(N/B \log_M(N/(MB)) \times D)$
- D is time for a disk operation

Quicksort (or other internal memory sorting algorithms) uses $O(N/B \log_2(N) \times D)$ time!

Common values for D, B and M:

- D was around $3.1 \cdot 10^4$ **nano-seconds** on SSDs and $2 \cdot 10^6$ **nano-seconds** on fast normal hard disks
- B is typically around 512-4096 bytes
- M is the size of your RAM divided by B, so say 8-32 GB RAM / 512-4096 bytes: $2.0 \cdot 10^6$ to $6.5 \cdot 10^7$

Comparison

D)

In practice, since hard disks are slow:

- Running time is basically time spend on disk operations
- D is time for a disk operation
- D is time for a disk operation

Quicksort (or other internal memory sorting algorithms) uses $O(N/B \log_2(N) \times D)$ time!

Common values for D, B and M:

- D was around $3.1 \cdot 10^4$ **nano-seconds** on SSDs and $2 \cdot 10^6$ **nano-seconds** on fast normal hard disks
- B is typically around 512-4096 bytes
- M is the size of your RAM divided by B, so say 8-32 GB RAM / 512-4096 bytes: $2.0 \cdot 10^6$ to $6.5 \cdot 10^7$

Comparison

D) time!

D)

In practice, since hard disks are slow:

- Running time is basically time spend on disk operations
- D is time for a disk operation

Quicksort (or other internal memory sorting algorithms) uses $O(N/B \log_2(N) \times D)$ time!

Quicksort (or other internal memory sorting algorithms) uses $O(N/B \log_2(N) \times D)$ time!

Common values for D, B and M:

- D was around $3.1 \cdot 10^4$ **nano-seconds** on SSDs and $2 \cdot 10^6$ **nano-seconds** on fast normal hard disks
- B is typically around 512-4096 bytes
- M is the size of your RAM divided by B, so say 8-32 GB RAM / 512-4096 bytes: $2.0 \cdot 10^6$ to $6.5 \cdot 10^7$

Comparison

D) time!

D)

In practice, since hard disks are slow:

- Running time is basically time spend on disk operations
- D is time for a disk operation

Quicksort (or other internal memory sorting algorithms) uses $O(N/B \log_2(N) \times D)$ time!

Common values for D, B and M:

Common values for D, B and M:

- D was around $3.1 \cdot 10^4$ **nano-seconds** on SSDs and $2 \cdot 10^6$ **nano-seconds** on fast normal hard disks
- B is typically around 512-4096 bytes
- M is the size of your RAM divided by B, so say 8-32 GB RAM / 512-4096 bytes: $2.0 \cdot 10^6$ to $6.5 \cdot 10^7$

Comparison

D) time!

D)

In practice, since hard disks are slow:

- Running time is basically time spend on disk operations
- D is time for a disk operation

Quicksort (or other internal memory sorting algorithms) uses $O(N/B \log_2(N) \times D)$ time!

Common values for D, B and M:

- D was around $3.1 \cdot 10^4$ **nano-seconds** on SSDs and $2 \cdot 10^6$ **nano-seconds** on fast normal hard disks

Common values for D, B and M:

- D was around $3.1 \cdot 10^4$ **nano-seconds** on SSDs and $2 \cdot 10^6$ **nano-seconds** on fast normal hard disks
- B is typically around 512-4096 bytes
- M is the size of your RAM divided by B, so say 8-32 GB RAM / 512-4096 bytes: $2.0 \cdot 10^6$ to $6.5 \cdot 10^7$

Comparison

D) time!

D)

In practice, since hard disks are slow:

- Running time is basically time spend on disk operations
- D is time for a disk operation

Quicksort (or other internal memory sorting algorithms) uses $O(N/B \log_2(N) \times D)$ time!

Common values for D, B and M:

- D was around $3.1 \cdot 10^4$ **nano-seconds** on SSDs and $2 \cdot 10^6$ **nano-seconds** on fast normal hard disks
- B is typically around 512-4096 bytes

- D was around $3.1 \cdot 10^4$ **nano-seconds** on SSDs and $2 \cdot 10^6$ **nano-seconds** on fast normal hard disks
- B is typically around 512-4096 bytes

- M is the size of your RAM divided by B, so say 8-32 GB RAM / 512-4096 bytes: $2.0 \cdot 10^6$ to $6.5 \cdot 10^7$

Comparison

D) time!

D)

In practice, since hard disks are slow:

- Running time is basically time spend on disk operations
- D is time for a disk operation

Quicksort (or other internal memory sorting algorithms) uses $O(N/B \log_2(N) \times D)$ time!

Common values for D, B and M:

- D was around $3.1 \cdot 10^4$ **nano-seconds** on SSDs and $2 \cdot 10^6$ **nano-seconds** on fast normal hard disks
- B is typically around 512-4096 bytes
- M is the size of your RAM divided by B, so say 8-32 GB RAM / 512-4096 bytes: $2.0 \cdot 10^6$ to $6.5 \cdot 10^7$
- B is typically around 512-4096 bytes
- M is the size of your RAM divided by B, so say 8-32 GB RAM / 512-4096 bytes: $2.0 \cdot 10^6$ to $6.5 \cdot 10^7$

Numeric example

Want to sort N=4 TB in 4 MB of RAM with B=4096

Numeric example

Want to sort N=4 TB in 4 MB of RAM with B=4096

External Merge Sort:

Numeric example

Want to sort N=4 TB in 4 MB of RAM with B=4096

External Merge Sort:

- M = 4 MB/4096 = 1024

Numeric example

Want to sort N=4 TB in 4 MB of RAM with B=4096

External Merge Sort:

- M = 4 MB/4096 = 1024
- $\log_M(N/(M\ B))$ is then ≈ $\log_{1024}(1024^2) = 2$

Numeric example

Want to sort N=4 TB in 4 MB of RAM with B=4096

External Merge Sort:

- M = 4 MB/4096 = 1024
- $\log_M(N/(M\ B))$ is then $\approx \log_{1024}(1024^2) = 2$

We are thus using 2 recursive calls, for 3 scans in total

Numeric example

Want to sort N=4 TB in 4 MB of RAM with B=4096

If we used a normal amount of RAM, we would be done in 1 recursive call…

External Merge Sort:

- M = 4 MB/4096 = 1024
- $\log_M(N/(M\ B))$ is then $\approx\ \log_{1024}(1024^2) = 2$

We are thus using 2 recursive calls, for 3 scans in total

Numeric example

Want to sort N=4 TB in 4 MB of RAM with B=4096

If we used a normal amount of RAM, we would be done in 1 recursive call…

External Merge Sort:

- M = 4 MB/4096 = 1024
- $\log_M(N/(M\ B))$ is then ≈ $\log_{1024}(1024^2) = 2$

We are thus using 2 recursive calls, for 3 scans in total

(Internal) Quick Sort:

Numeric example

Want to sort N=4 TB in 4 MB of RAM with B=4096

If we used a normal amount of RAM, we would be done in 1 recursive call…

External Merge Sort:

- M = 4 MB/4096 = 1024
- $\log_M(N/(M\ B))$ is then ≈ $\log_{1024}(1024^2) = 2$

We are thus using 2 recursive calls, for 3 scans in total

(Internal) Quick Sort:

- $\log_2(N)=42$

Numeric example

Want to sort N=4 TB in 4 MB of RAM with B=4096

If we used a normal amount of RAM, we would be done in 1 recursive call…

External Merge Sort:

- M = 4 MB/4096 = 1024
- $\log_M(N/(M\ B))$ is then ≈ $\log_{1024}(1024^2) = 2$

We are thus using 2 recursive calls, for 3 scans in total

(Internal) Quick Sort:

- $\log_2(N)=42$

We are thus using 42 recursive calls, for 43 scans in total

Numeric example

Want to sort N=4 TB in 4 MB of RAM with B=4096

If we used a normal amount of RAM, we would be done in 1 recursive call…

External Merge Sort:

- M = 4 MB/4096 = 1024
- $\log_M(N/(M\ B))$ is then ≈ $\log_{1024}(1024^2) = 2$

We are thus using 2 recursive calls, for 3 scans in total

(Internal) Quick Sort:

- $\log_2(N)=42$

We are thus using 42 recursive calls, for 43 scans in total

Assuming we are lucky about pivot elements

Summary

External memory sorting is very much faster on inputs that are much larger than main memory