COMP 3311
DATABASE MANAGEMENT SYSTEMS

LECTURE 15
QUERY PROCESSING: INTRODUCTION

QUERY PROCESSING: OUTLINE

Overview of Query Processing

Cost Estimates

- Selection
- Sorting
- Join
- Other Operations

Evaluation of Expressions

- Materialization
- Pipelining

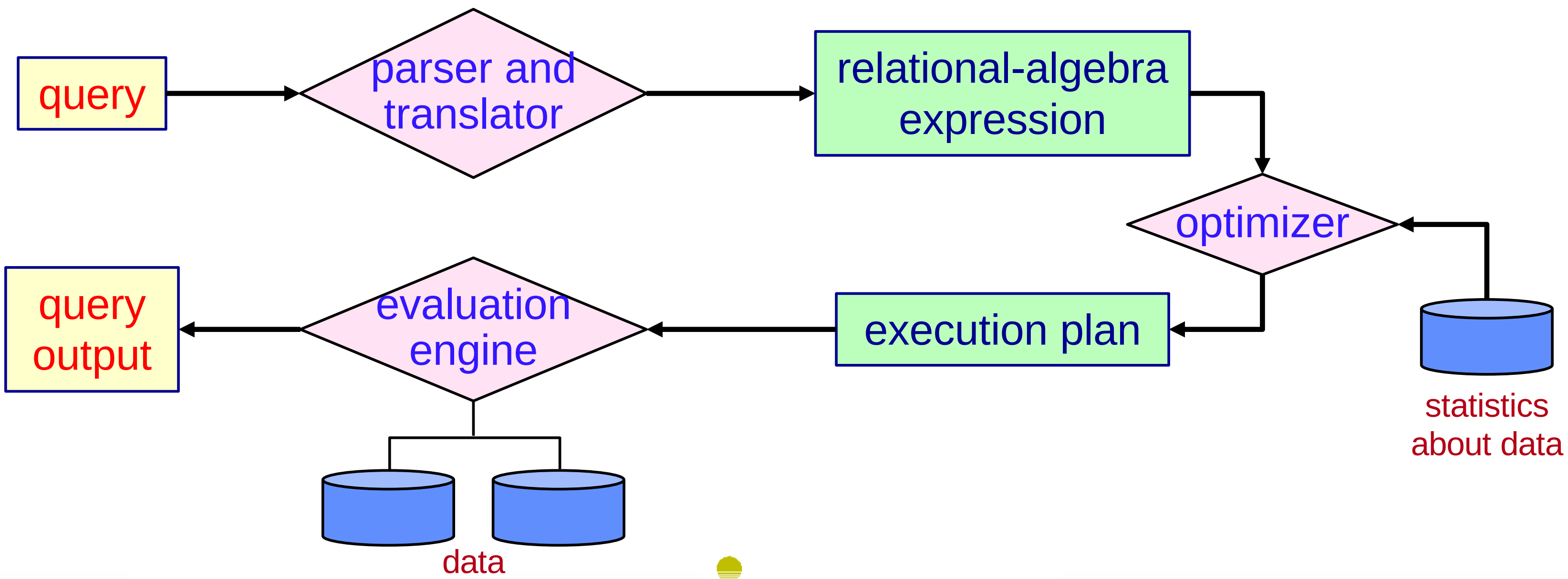
QUERY PROCESSING OVERVIEW

Parsing and Translation

– The query is checked for syntax, relations are verified against the system catalog, the query is translated into a relational-algebra expression and transformed by the optimizer into an execution plan.

Evaluation

– The evaluation engine takes an execution plan, executes that plan, and returns the query output.



QUERY EVALUATION

- A given relational-algebra expression may have many equivalent expressions.

$$\sigma_{balance<2500}(\pi_{balance}(Account)) \equiv \pi_{balance}(\sigma_{balance<2500}(Account))$$

☞ **A relational-algebra expression can be evaluated using one of several different algorithms.**

- A query-execution plan represents a relational-algebra expression as a relational-algebra tree and annotates it with a detailed evaluation strategy for each operation.

$\pi_{balance}$
|
$\sigma_{balance<2500}$
|
Account

- To process the above query, the query-execution plan can:
  - use an index on balance to find accounts with balance<2500.

    or
  - perform a complete relation scan and discard accounts with balance≥2500.

Which option to use may depend on several factors.

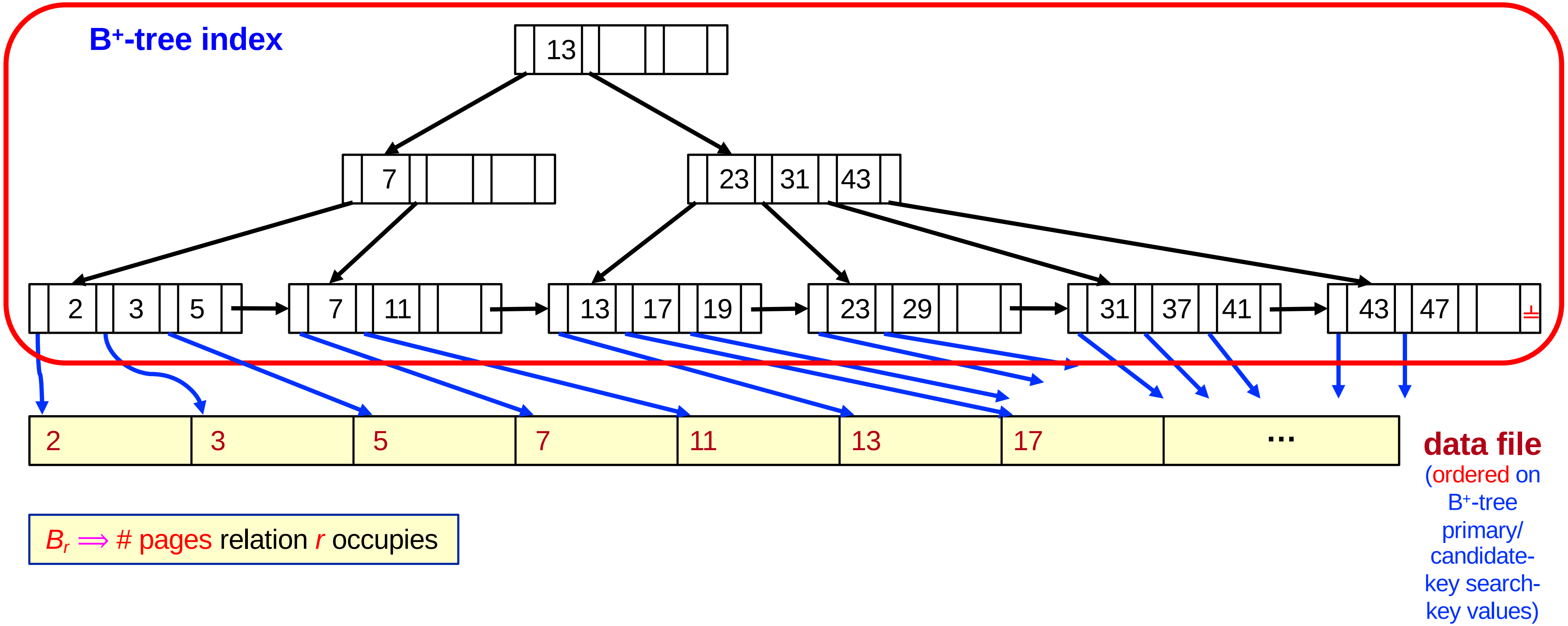
QUERY COST MEASURES

Query optimization **chooses,** ***amongst all equivalent evaluation plans,*** **the one with the (expected) lowest cost.**

- Cost is estimated using information from the database catalog.
  - ☞ The number of records in each relation, the size of the records, whether an index is available, etc.
- The cost to execute a query depends on the size of the buffer in main memory.
  - ☞ We often use worst case estimates, assuming that only the minimum amount of memory needed for the operation is available.
- For simplicity we use *number of page I/Os* as the cost measure.
  - ☞ **We ignore the difference in cost between sequential and random I/O as well as CPU cost.**
    (Real query processors take these factors into consideration.)

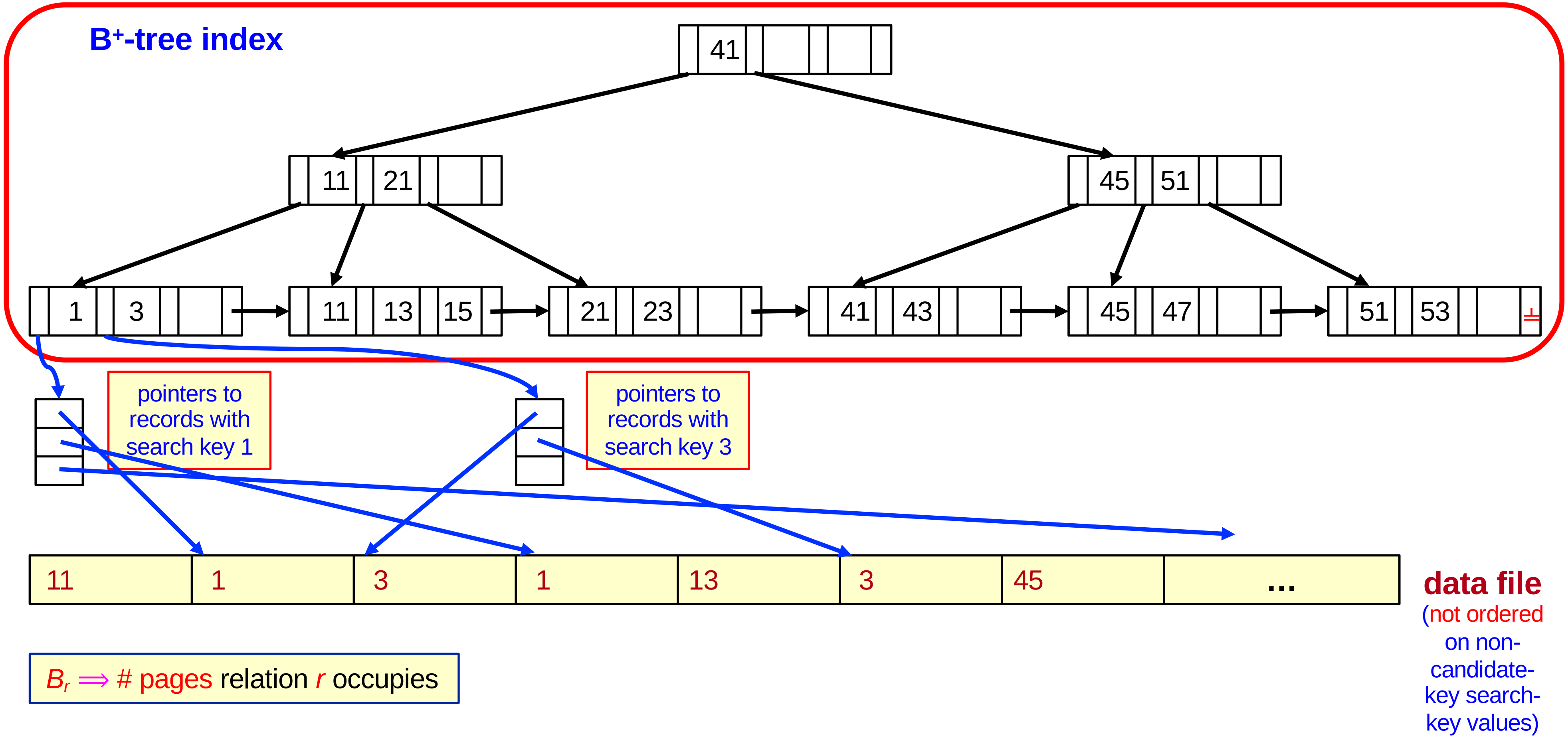
☞ **We normally do not include the cost of writing the final output to disk. Why?**

EXAMPLE B+-TREE INDEX



$B_r \Longrightarrow$ # pages relation r occupies

EXAMPLE B+-TREE INDEX



$B_r \Longrightarrow$ # pages relation r occupies

COST ESTIMATES: SELECTION — FILE SCAN

Algorithm A1: *linear search*

$B_r \Rightarrow$ # pages relation r occupies

Scan each page and check all records to see whether they satisfy the selection condition.

Cost estimate (page I/Os)

- B_r the number of pages that contain records of relation r

or

- $B_r/2$ if equality selection on a candidate key attribute, since we stop when we find the record (as there is at most only one with the specified value).

– Linear search can be applied regardless of
  - selection condition ***or***
  - ordering of records in the file ***or***
  - availability of indexes.

COST ESTIMATES: SELECTION — FILE SCAN (cont'd)

Algorithm A2: *binary search on ordered file*

$B_r \Longrightarrow$ # pages relation r occupies

Applicable if the selection is an equality comparison on the attribute on which the file is ordered *and* the pages are stored contiguously.

Cost estimate (page I/Os)

- $\lceil \log_2(B_r) \rceil$ if equality selection is on a candidate key attribute
- $\lceil \log_2(B_r) \rceil$ for locating the first record satisfying the selection condition
  \+ # pages containing records satisfying the selection condition

COST ESTIMATES: SELECTION — EQUALITY SEARCH (cont'd)

Algorithm A3: *clustering index on candidate key*

Retrieve a single record that satisfies the corresponding equality condition.

Cost estimate (page I/Os)

- For a tree index: $HT_i + 1$ where HT_i is the height of the tree index
- For a hash index: Normally a hash index would not be used in this case as the file would use a hash file organization.

Algorithm A4: *clustering index on non-candidate key*

Retrieve multiple records that satisfy the corresponding equality condition on consecutive pages.

Cost estimate (page I/Os)

- HT_i + # pages containing records satisfying the selection condition

COST ESTIMATES: SELECTION — EQUALITY SEARCH (CONT'D)

Algorithm A5: *non-clustering (secondary) index*

Retrieve a single record if the search key is a candidate key.

Cost estimate (page I/Os)

- For a tree index: $HT_i + 1$ where HT_i is the height of the tree index
- For a hash index: 1 + 1 *or* 1.2 + 1 if there exist overflow buckets

Retrieve multiple records if the search key is not a candidate key.

Cost estimate (page I/Os)

- For a tree index: HT_i + cost of retrieving indirection pointers + # records retrieved (assumes each record requires a page access)
- For a hash index: 1 + cost to retrieve indirection pointers + # records retrieved
  or
  1.2 + cost to retrieve indirection pointers + # records retrieved if there exist overflow buckets

☞ Can be very expensive! **Why?**

COST ESTIMATES: SELECTION — COMPARISONS (CONT'D)

Can implement comparison selections of the form

$\sigma_{A \geq v}(r)$ *or* $\sigma_{A \leq v}(r)$

by using linear file scan, binary search or indexes as follows.

Algorithm A6: *clustering index*

Relation is sorted on A.

$\sigma_{A \geq v}(r)$ use the index to find the first record where $A \geq v$ and scan the relation sequentially from there.

$\sigma_{A \leq v}(r)$ do not use the index; instead scan the relation sequentially until you find the first record where $A > v$.

COST ESTIMATES: SELECTION — COMPARISONS (CONT'D)

Can implement selections of the form

$\sigma_{A \geq v}(r)$ *or* $\sigma_{A \leq v}(r)$

by using linear file scan, binary search or indexes as follows.

Algorithm A7: *non-clustering (secondary) index*

$\sigma_{A \geq v}(r)$ use the index to find first index entry where $A \geq v$ and scan the leaf pages of the index sequentially from there to find pointers to the records.

$\sigma_{A \leq v}(r)$ scan the leaf pages of the index sequentially finding pointers to records, until the first entry where $A > v$.

- In either case, retrieve records that are pointed to.
  - Requires an I/O for each record.
  - Linear file scan may be cheaper if many records need to be retrieved!

COST ESTIMATES: SELECTION — COMPARISONS (CONT'D)

Conjunction (AND): $\sigma_{\theta 1 \wedge \theta 2 \wedge \ldots \wedge \theta n}(r)$

Algorithm A8: *using one index*

- Select a θ_i and algorithms A1 through A7 that results in the least cost for $\sigma_{\theta i}(r)$.
- *Check the remaining conditions in the record after retrieving it into memory.*

Algorithm A9: *using a composite index*

- If available, use a composite (multi-attribute) index (a combination of θ_i).
- The type of index determines which of the algorithms A2, A3 or A4 will be used.

COST ESTIMATES: SELECTION — COMPARISONS (CONT'D)

Conjunction (AND): $\sigma_{\theta 1 \wedge \theta 2 \wedge \ldots \wedge \theta n}(r)$

Algorithm A10: *by intersection of record pointers*

If *any attributes* of the individual conditions have indexes with record pointers then

- Use the corresponding index for each condition and take the intersection of all the obtained sets of record pointers.
- Use the record pointers to retrieve the records from the file.
- If some conditions do not have appropriate indexes, then *check the conditions in the records in memory*.

  Note: In some cases only an index scan is required (e.g., count).

Else use linear scan.

Cost estimate

- Using indexes: sum of the costs of the individual index scans
  + the cost of retrieving the records

COST ESTIMATES: SELECTION — COMPARISONS (CONT'D)

Disjunction (OR): $\sigma_{\theta 1 \vee \theta 2 \vee \ldots \vee \theta n}(r)$

Algorithm A11: *by union of record pointers*

If *all attributes* of the individual conditions have indexes with record pointers then

- Use the corresponding index for each condition and take the union of all the obtained sets of record pointers.
- Use the record pointers to fetch the records from the file.

  Note: In some cases only an index scan is required (e.g., count).

Else use linear scan.

Cost estimate

- Using indexes: sum of the costs of the individual index scans
  + the cost of retrieving the records

COST ESTIMATES: SELECTION — COMPARISONS (CONT'D)

Negation (NOT): $\sigma_{\neg\theta}(r)$

– Use linear scan on the file.

– If very few records satisfy $\neg\theta$ *and* an index is applicable for θ then

  - Use the index to find records satisfying the negation and retrieve the records from the file.

Example

Suppose that a B^+-tree index on branchCity is available on relation Branch, and that no other index is available.

$$\sigma_{\neg(branchCity < \text{'Brooklyn'})}(Branch)$$

Use the index to locate the first record whose branchCity field has value "Brooklyn". From this record, follow the pointer chains in the index until the end, retrieving all the records.

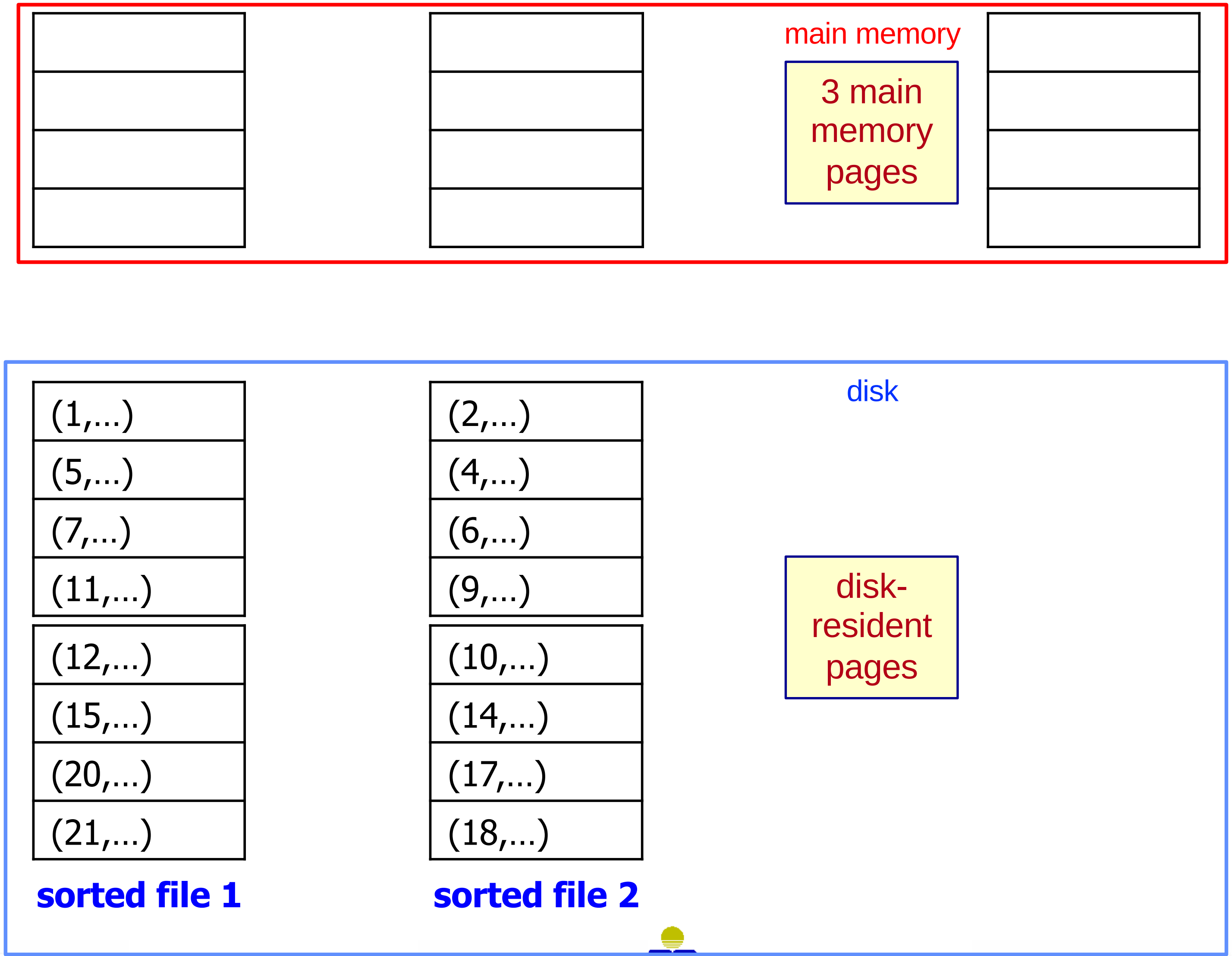
SORTING IN A DBMS

- When does a DBMS need to sort data?
  - To order query results (e.g., increasing by age).
  - To remove duplicate results.
  - For bulk loading a B^+-tree index.
  - Before performing a join operation (e.g., merge-join).
- Often database data is too large to fit into available main memory all at once.

☞ **External sorting is often required.**

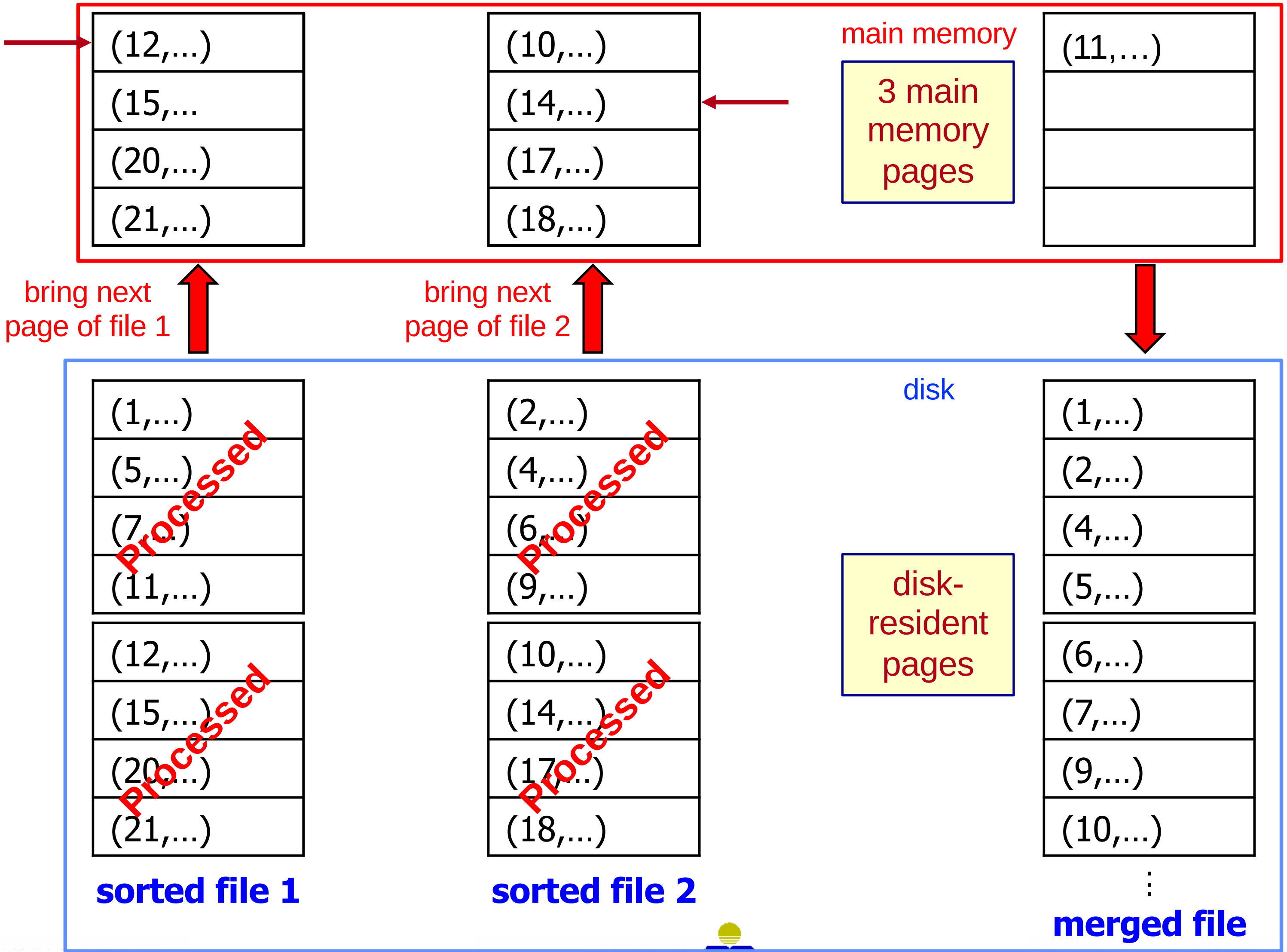
EXTERNAL SORTING (DISK-RESIDENT FILES)

Merging 2 sorted files, 2 pages each with 3 pages of main memory



EXTERNAL SORTING (DISK-RESIDENT FILES)
(CONT'D)

Merging 2 sorted files, 2 pages each with 3 pages of main memory



EXTERNAL SORTING (DISK-RESIDENT FILES) (CONT'D)

Continuing the previous example:

Question: We assumed that each file is already sorted. If a file is not sorted, how do we sort it (using only the 3 buffer pages)?

Question: The previous example assumes two separate files. How do we apply this idea to sort a single file?

Question: Can we do better if we have *M* > 3 main memory pages?

EXTERNAL SORT-MERGE ALGORITHM

Let M denote the memory size (in pages).

Create sorted runs

Let i be 0 initially.

Repeat the following **until** the end of the relation:

(a) Read M pages of the relation into memory.

(b) Sort the in-memory pages.

(c) Write sorted data to run R_i; increment i.

Let the final value of i be N.

Merge the runs (N-way merge)

(We assume (for now) that $N < M$.)

Use N pages of memory to buffer input runs, and 1 page to buffer output.

Read the first page of each run into its buffer page.

repeat

  Select the first record (in sort order) among all buffer pages.

  Write the record to the output buffer; if the output buffer is full write it to disk.

  Delete the record from its input buffer page.

  If the buffer page becomes empty **then**

    Read the next page (if any) of the run into the buffer.

until all input buffer pages are empty.

EXTERNAL SORT-MERGE ALGORITHM (cont'd)

- Let M denote the memory size (in pages), i the number of runs (i.e., the number of files to be merged).
- If $i \geq M$, several merge *passes* are required.

  In each pass, contiguous groups of M - 1 runs are merged.
- A pass reduces the number of runs by a factor of M -1 and creates runs longer by the same factor.

  E.g., If M=11, and there are 90 runs, one pass reduces the number of runs to $\lceil 90/10 \rceil = 9$, each 10 times the size of the initial runs.
- Repeated passes are performed until all runs are merged into one.

Number of passes (sort & merge)

> $1 + \lceil \log_{M-1}(B_r/M) \rceil$ passes where B_r is the file size in pages

I/O cost (sort & merge)

> $\underbrace{2 * B_r}_{\text{sort}} + \underbrace{2 * B_r * \lceil \log_{M-1}(B_r/M) \rceil}_{\text{merge}} = 2 * B_r * (1 + \lceil \log_{M-1}(B_r/M) \rceil)$ pages

EXTERNAL SORT-MERGE ALGORITHM (CONT'D)

- Example assuming the number of pages B_r=12 with 1 record per page and M=3 buffer pages.

- **Number of passes**

  $1+\lceil \log_{M-1}(B_r/M) \rceil$ = 3

- **Number of disk I/Os** (read and write):

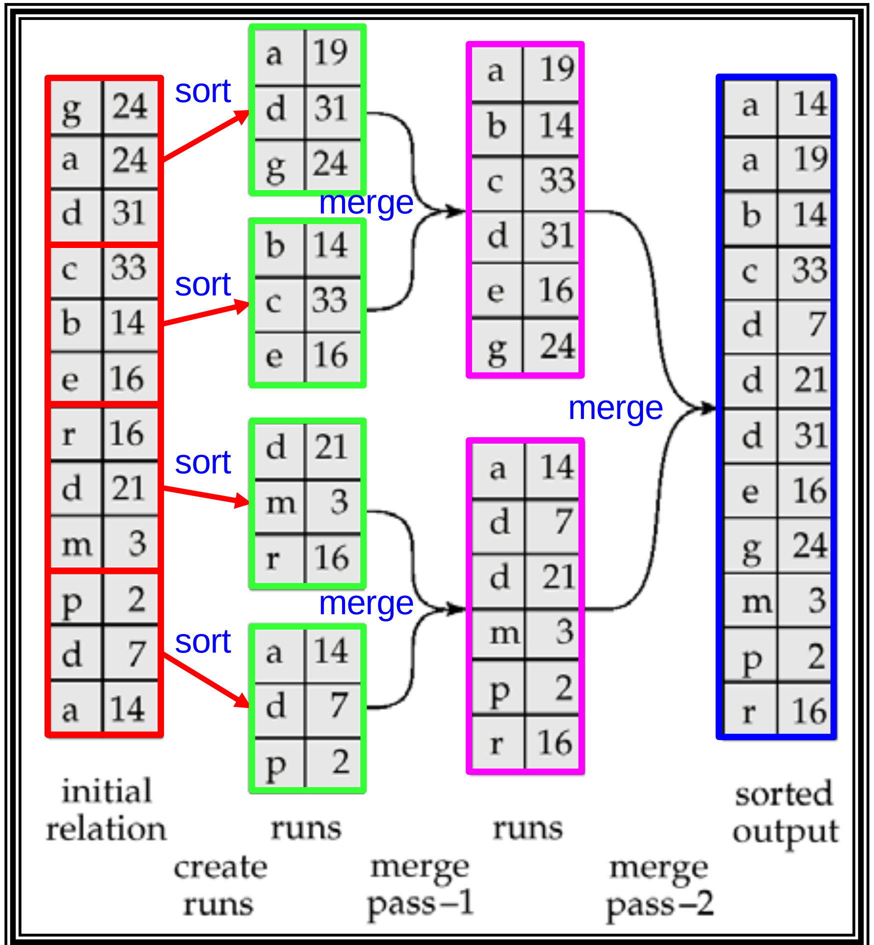
  Pass 0 (create sorted runs): 12 x 2 = 24

  Pass 1: 12 x 2 = 24

  Pass 2: 12 x 2 =24

  Total I/O cost = 72 pages

  ☞ $2^* B_r^*(1+\lceil \log_{M-1}(B_r/M) \rceil)$ = $2^*12^*(1+\lceil \log_2(4) \rceil)$ = 72



Need to read and write all pages on each pass.