>>

Sorting

^ >>

❖ The Sort Operation

Sorting is explicit in queries only in the **`order by`** clause

```
select * from Students order by name;
```

Sorting is used internally in other operations:

- eliminating duplicate tuples for projection
- ordering files to enhance select efficiency
- implementing various styles of join
- forming tuple groups in **`group by`**

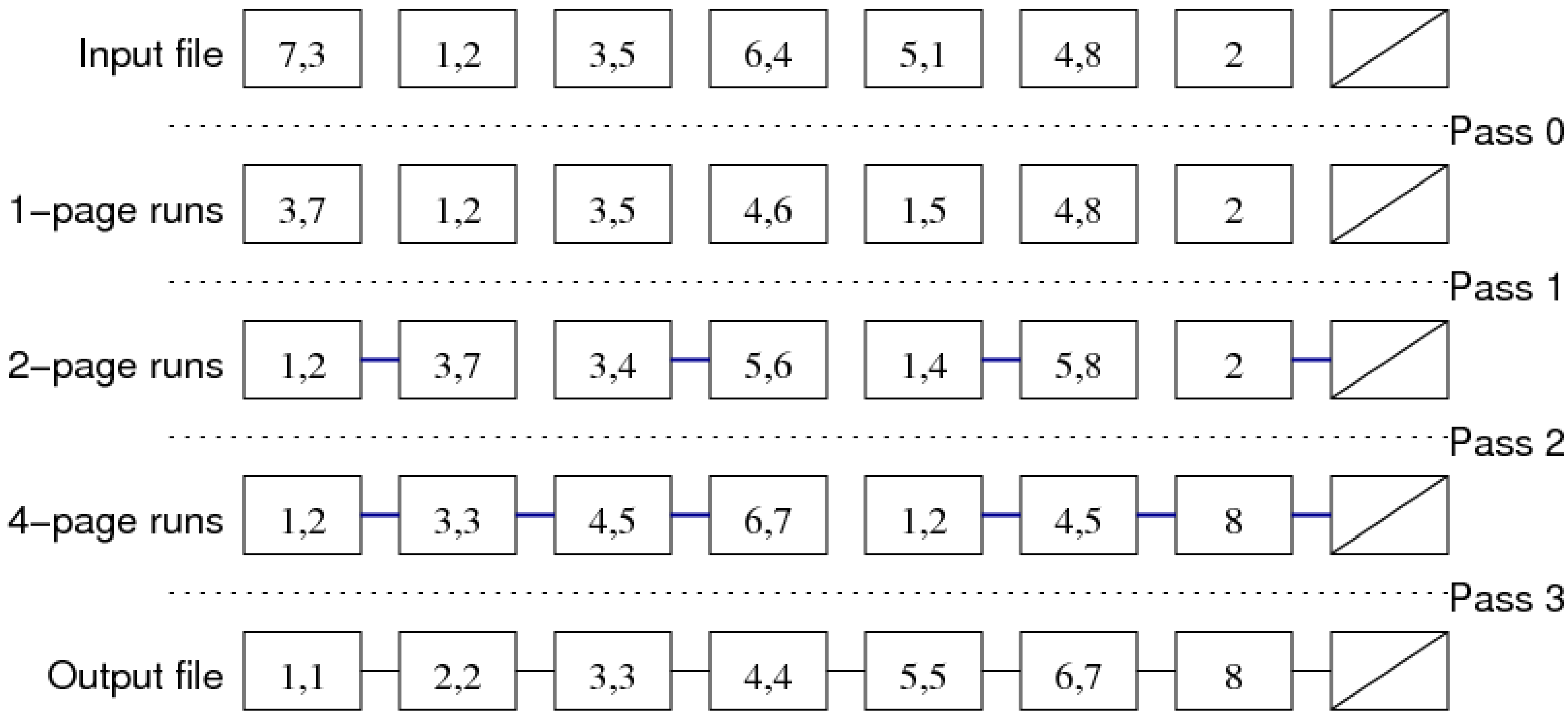
Sort methods such as quicksort are designed for in-memory data.

For large data on disks, need external sorts such as merge sort.

<< ^ >>

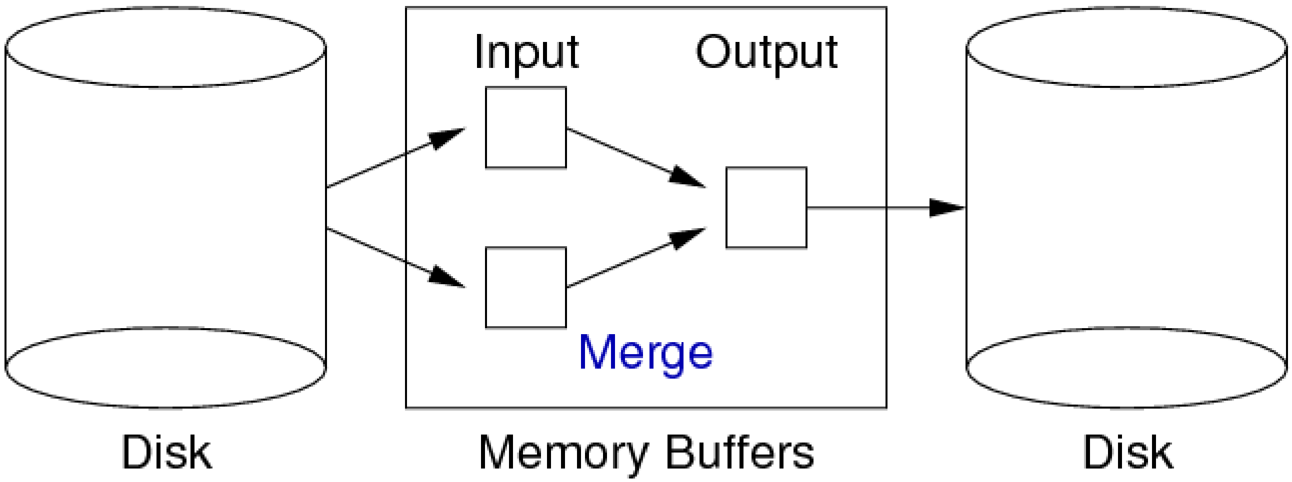
❖ Two-way Merge Sort

Example:



❖ Two-way Merge Sort (cont)

Requires at least three in-memory buffers:



Assumption: cost of Merge operation on two in-memory buffers ≅ *0*.

<< ∧ >>

❖ Comparison for Sorting

Above assumes that we have a function to compare tuples.

Needs to understand ordering on different data types.

Need a function **`tupCompare(r1,r2,f)`** (cf. C's **`strcmp`**)

```
int tupCompare(r1,r2,f)
{
   if (r1.f < r2.f) return -1;
   if (r1.f > r2.f) return 1;
   return 0;
}
```

Assume =, <, > are available for all attribute types.

<< ∧ >>

❖ Comparison for Sorting (cont)

In reality, need to sort on multiple attributes and ASC/DESC, e.g.

```
-- example multi-attribute sort
select * from Students
order by age desc, year_enrolled
```

Sketch of multi-attribute sorting function

```
int tupCompare(r1,r2,criteria)
{
   foreach (f,ord) in criteria {
      if (ord == ASC) {
         if (r1.f < r2.f) return -1;
         if (r1.f > r2.f) return 1;
      }
      else {
         if (r1.f > r2.f) return -1;
         if (r1.f < r2.f) return 1;
      }
```

```
    }
    return 0;
}
```

<< ^ >>

❖ Cost of Two-way Merge Sort

For a file containing b data pages:

- require $ceil(log_2 b)$ passes to sort,
- each pass requires b page reads, b page writes

Gives total cost: $2.b.ceil(log_2 b)$

Example: Relation with $r=10^5$ and $c=50 \Rightarrow b=2000$ pages.

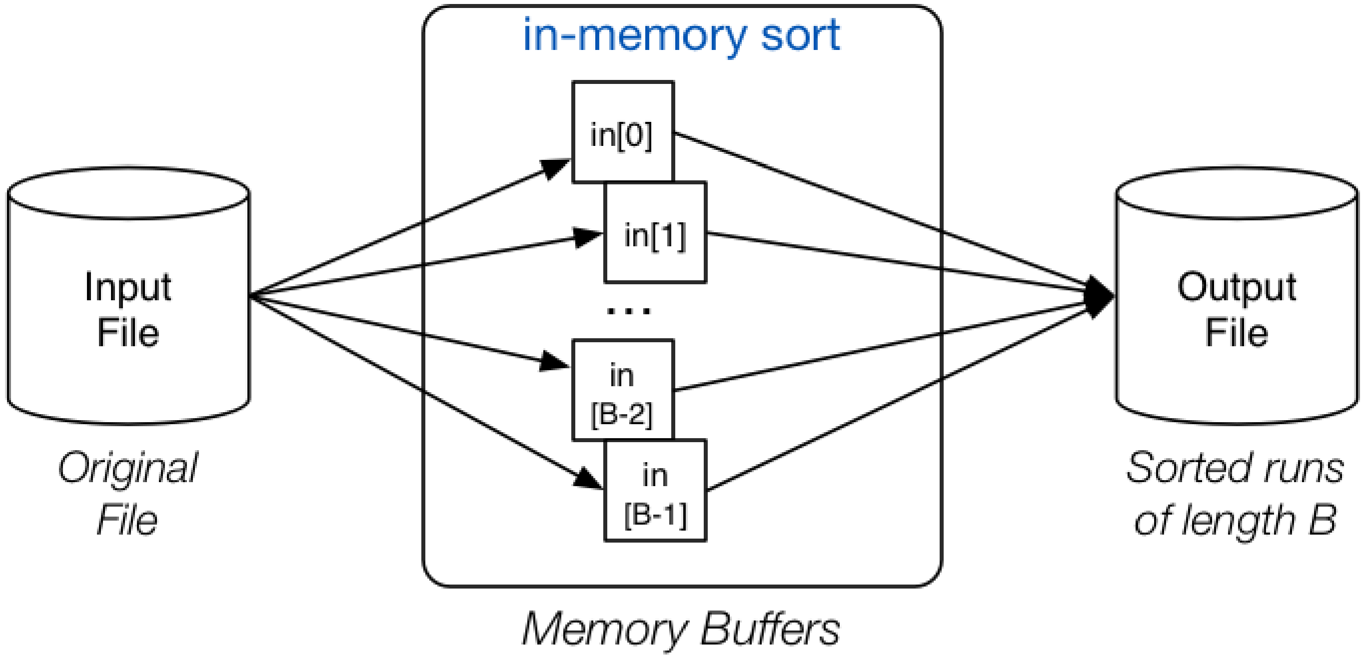
Number of passes for sort: $ceil(log_2 2000) = 11$

Reads/writes entire file 11 times! Can we do better?

<< ∧ >>

❖ n-Way Merge Sort

Initial pass uses: *B* total buffers

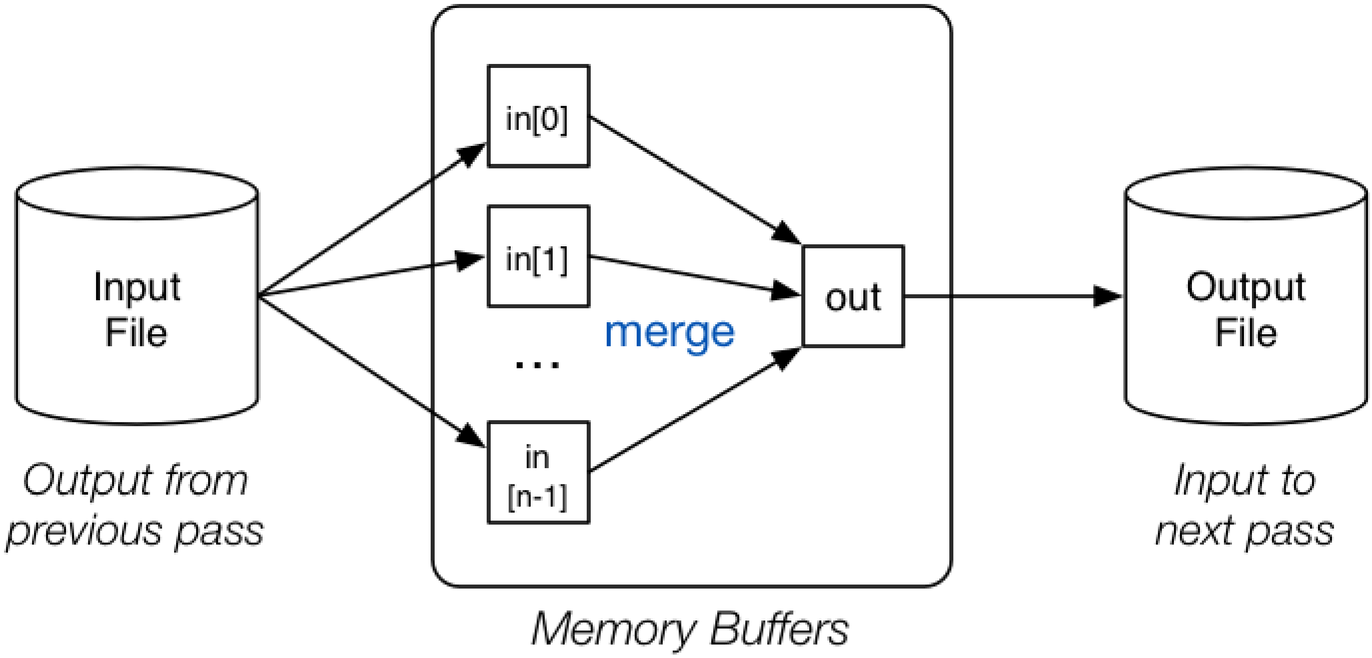


Reads *B* pages at a time, sorts in memory, writes out in order

<< ^ >>

❖ n-Way Merge Sort (cont)

Merge passes use: *n = B-1* input buffers, *1* output buffer



<< ^ >>

❖ n-Way Merge Sort (cont)

Method:

```
// Produce B-page-long runs
for each group of B pages in Rel {
    read B pages into memory buffers
    sort group in memory
    write B pages out to Temp
}
// Merge runs until everything sorted
numberOfRuns = ceil(b/B)
while (numberOfRuns > 1) {
    // n-way merge, where n=B-1
    for each group of n runs in Temp {
        merge into a single run via input buffers
        write run to newTemp via output buffer
    }
    numberOfRuns = ceil(numberOfRuns/n)
    Temp = newTemp // swap input/output files
}
```

<< ^ >>

❖ Cost of n-Way Merge Sort

Consider file where *b = 4096*, *B = 16* total buffers:

- pass 0 produces 256 × 16-page sorted runs
- pass 1
  - performs 15-way merge of groups of 16-page sorted runs
  - produces 18 × 240-page sorted runs (17 full runs, 1 short run)
- pass 2
  - performs 15-way merge of groups of 240-page sorted runs
  - produces 2 × 3600-page sorted runs (1 full run, 1 short run)
- pass 3
  - performs 15-way merge of groups of 3600-page sorted runs
  - produces 1 × 4096-page sorted runs

(cf. two-way merge sort which needs 11 passes)

<< ^ >>

❖ Cost of n-Way Merge Sort (cont)

Generalising from previous example ...

For b data pages and B buffers

- first pass: read/writes b pages, gives $b_0 = ceil(b/B)$ runs
- then need $ceil(log_n b_0)$ passes until sorted, where $n = B\text{-}1$
- each pass reads and writes b pages (i.e. $2.b$ page accesses)

$Cost = 2.b.(1 + ceil(log_n b_0))$, where b_0 and n are defined above

❖ Sorting in PostgreSQL

Sort uses a merge-sort (from Knuth) similar to above:

- backend/utils/sort/tuplesort.c
- include/utils/sortsupport.h

Tuples are mapped to **`SortTuple`** structs for sorting:

- containing pointer to tuple and sort key
- no need to reference actual Tuples during sort
- unless multiple attributes used in sort

If all data fits into memory, sort using **`qsort()`**.

If memory fills while reading, form "runs" and do disk-based sort.

<< ∧ >>

❖ Sorting in PostgreSQL (cont)

Disk-based sort has phases:

- divide input into sorted runs using HeapSort
- merge using *N* buffers, one output buffer
- *N* = as many buffers as **`workMem`** allows

Described in terms of "tapes" ("tape" ≅ sorted run)

Implementation of "tapes": backend/utils/sort/logtape.c

<< ∧

❖ Sorting in PostgreSQL (cont)

Sorting comparison operators are obtained via catalog (in ***`Type.o`***):

```
// gets pointer to function via pg_operator
struct Tuplesortstate { ... SortTupleComparator ... };

// returns negative, zero, positive
ApplySortComparator(Datum datum1, bool isnull1,
                    Datum datum2, bool isnull2,
                    SortSupport sort_helper);
```

Flags in **`SortSupport`** indicate: ascending/descending, nulls-first/last.

`ApplySortComparator()` is PostgreSQL's version of **`tupCompare()`**

Produced: 1 Mar 2021