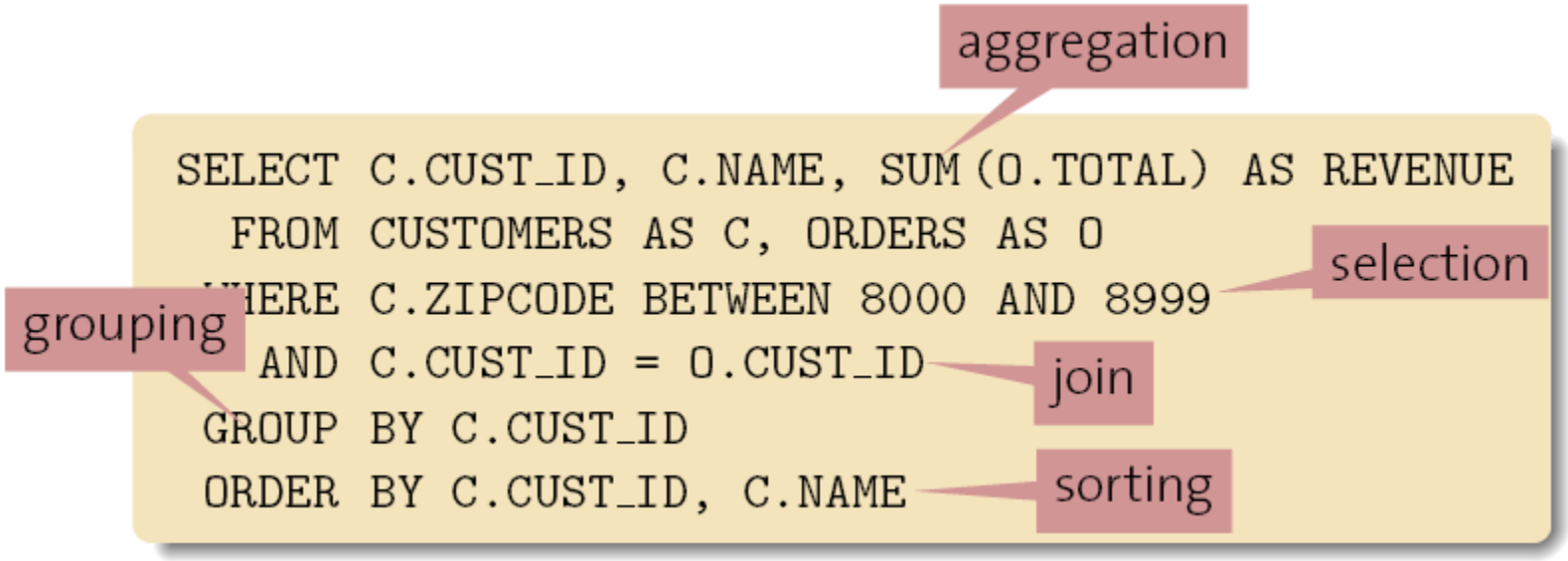
Systems Infrastructure for Data Science

Web Science Group
Uni Freiburg
WS 2012/13

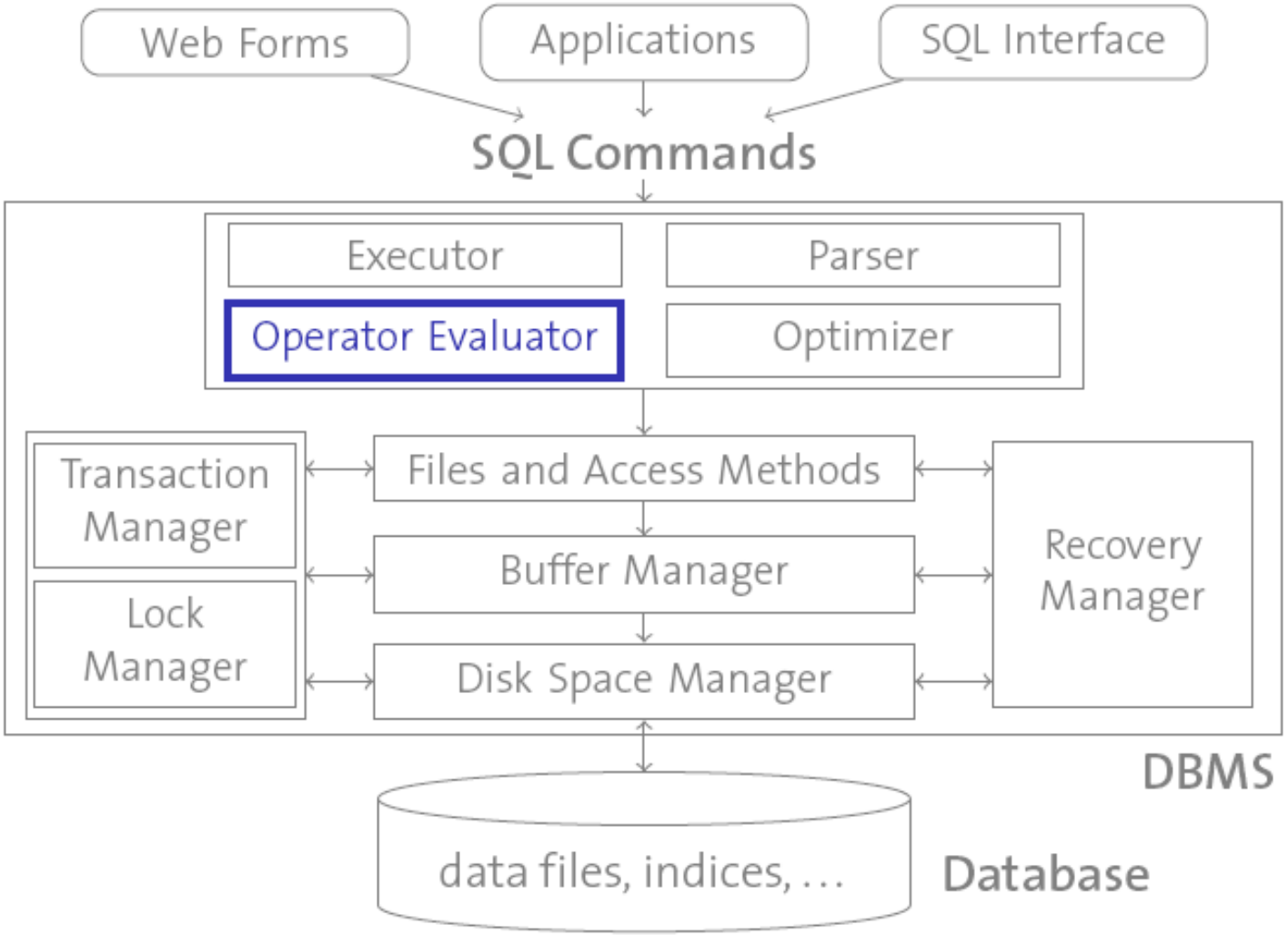
Lecture IV: Query Processing

Query Processing



- A DBMS needs to perform a number of tasks
  - with **limited memory resources**,
  - over **large amounts of data**,
  - yet, **as fast as possible**.

Query Processing



Query Processing: Our Agenda

- Efficient algorithms for implementing the main relational operators
  - Sorting
  - Join
  - Selection
  - Projection
  - Set Operators, Aggregate Operators
- Efficient techniques for executing compositions of operators in a query plan
  - Pipelining

Sorting

- Sorting is a core database operation with numerous applications:
  - An SQL query may explicitly request sorted output:
    - `SELECT A,B,C FROM R ORDER BY A`
  - Bulk-loading a B^+-tree pre-supposes sorted data.
  - Duplicate elimination is particularly easy over sorted input:
    - `SELECT DISTINCT A,B,C FROM R`
  - Some database operators rely on their input files being already sorted (some of which we will see later in this course such as sort-merge join).
- How can we sort a file that exceeds the available main memory size by far (let alone the available buffer manager space)?

Two-Way Merge Sort

- We start with two-way merge sort, which can sort files of arbitrary size with **only three pages of buffer space**.
- Two-way merge sort sorts a file with $N = 2^k$ pages in multiple **passes**, each of them producing a certain number of sorted sub-files called "**runs**".

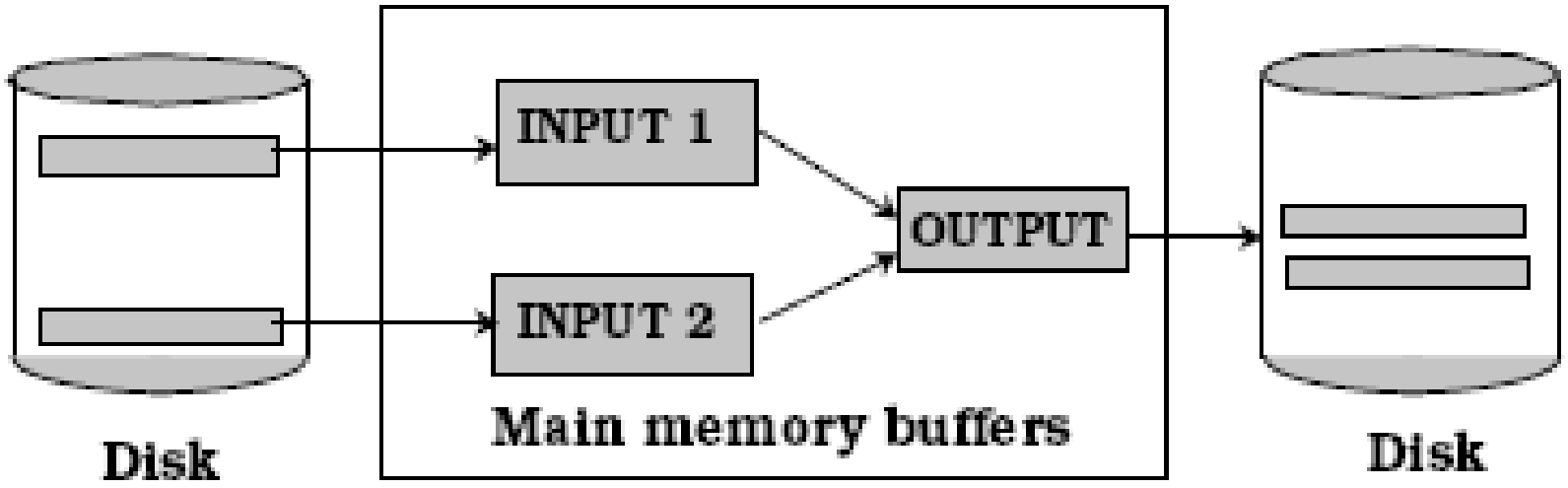
k+1 passes

  - **Pass 0** sorts each of the 2^k input pages individually and in main memory, resulting in 2^k sorted runs.
  - **Pass *n*** merges 2^{k-n} pairs of runs into 2^{k-n} sorted runs.
  - **Pass *k*** leaves only one sorted run left (i.e., the overall sorted result).

- During each pass, we read/write every page in the file. Hence, *(k+1)*N* page reads and *(k+1)*N* page writes are required to sort the file.

Two-Way Merge Sort: Why 3 Buffer Pages?

- Pass 0: Read a page, sort it, write it.
  - Only one buffer page is used.
- Pass 1, 2, …, k: Merge pairs of runs.
  - Three buffer pages are used.



Multiple Passes of Two-Way Merge Sort

Pass 0 (**Input:** $N = 2^k$ unsorted pages; **Output:** 2^k sorted runs)

1. **Read** N pages, **one page at a time**
2. **Sort** records in main memory.
3. **Write** sorted pages to disk (each page results in a **run**).

This pass requires **one page** of buffer space.

Pass 1 (**Input:** $N = 2^k$ sorted runs; **Output:** 2^{k-1} sorted runs)

1. Open two runs r_1 and r_2 from Pass 0 for reading.
2. **Merge** records from r_1 and r_2, reading input page-by-page.
3. **Write** new two-page run to disk (page-by-page).

This pass requires **three pages** of buffer space.

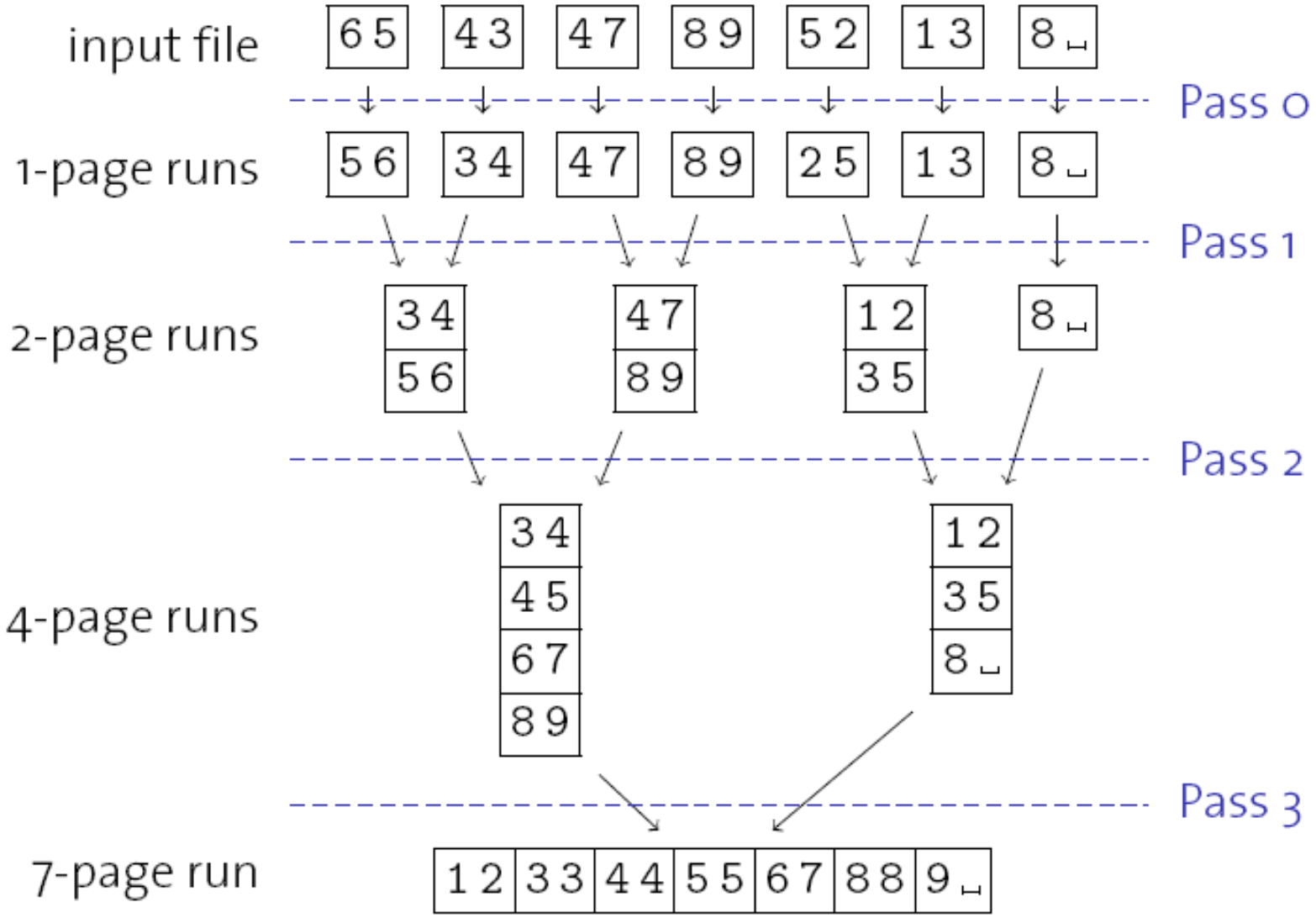
⋮

Pass n (**Input:** 2^{k-n+1} sorted runs; **Output:** 2^{k-n} sorted runs)

1. Open two runs r_1 and r_2 from Pass $n-1$ for reading.
2. **Merge** records from r_1 and r_2, reading input page-by-page.
3. **Write** new 2^n-page run to disk (page-by-page).

This pass requires **three pages** of buffer space.

Two-Way Merge Sort Example



Two-Way Merge Sort: I/O Behavior

- To sort a file of N pages, we need to read and write N pages during each pass.
- Number of I/O operations per pass:

$$2 \cdot N$$

- Number of passes:

$$\underbrace{1}_{\text{Pass 0}} + \underbrace{\lceil \log_2 N \rceil}_{\text{Passes } 1 \ldots k}$$

- Total number of I/O operations:

$$2 \cdot N \cdot \left(1 + \lceil \log_2 N \rceil\right)$$

General External Merge Sort

- So far, we “voluntarily” used only three pages of buffer space.
- How could we make effective use of a significantly larger buffer pool (of, say, *B* memory frames)?
- There are basically two knobs that we can turn:
  - **Reduce the number of initial runs** by using the full buffer space during the in-memory sort.
  - **Reduce the number of passes** by merging more than 2 runs at a time.

Reducing the Number of Initial Runs

- With *B* frames available in the buffer pool, we can **read *B* pages at a time** during Pass 0 and sort them in memory:

> Pass 0 (**Input:** *N* unsorted pages; **Output:** $\lceil N/B \rceil$ sorted runs)
> 1. **Read** *N* pages, *B* **pages at a time**
> 2. **Sort** records in main memory.
> 3. **Write** sorted pages to disk (resulting in $\lceil N/B \rceil$ **runs**).
>
> This pass uses *B* **pages** of buffer space.

- The number of initial runs determines the **number of passes** we need to make.
  - Total number of I/O operations:

$$2 \cdot N \cdot \underbrace{\left(1 + \left\lceil \log_2 \lceil N/B \rceil \right\rceil\right)}_{\text{number of passes}}$$

Reducing the Number of Passes

- With *B* frames available in the buffer pool, we can **merge *B-1* pages at a time** (leaving one frame as a write buffer).

Pass n (**Input:** $\frac{\lceil N/B \rceil}{(B-1)^{n-1}}$ sorted runs; **Output:** $\frac{\lceil N/B \rceil}{(B-1)^{n}}$ sorted runs)

1. Open $B-1$ runs $r_1 \ldots r_{B-1}$ from Pass $n-1$ for reading.
2. **Merge** records from $r_1 \ldots r_{B-1}$, reading input page-by-page.
3. **Write** new $B \cdot (B-1)^n$-page run to disk (page-by-page).

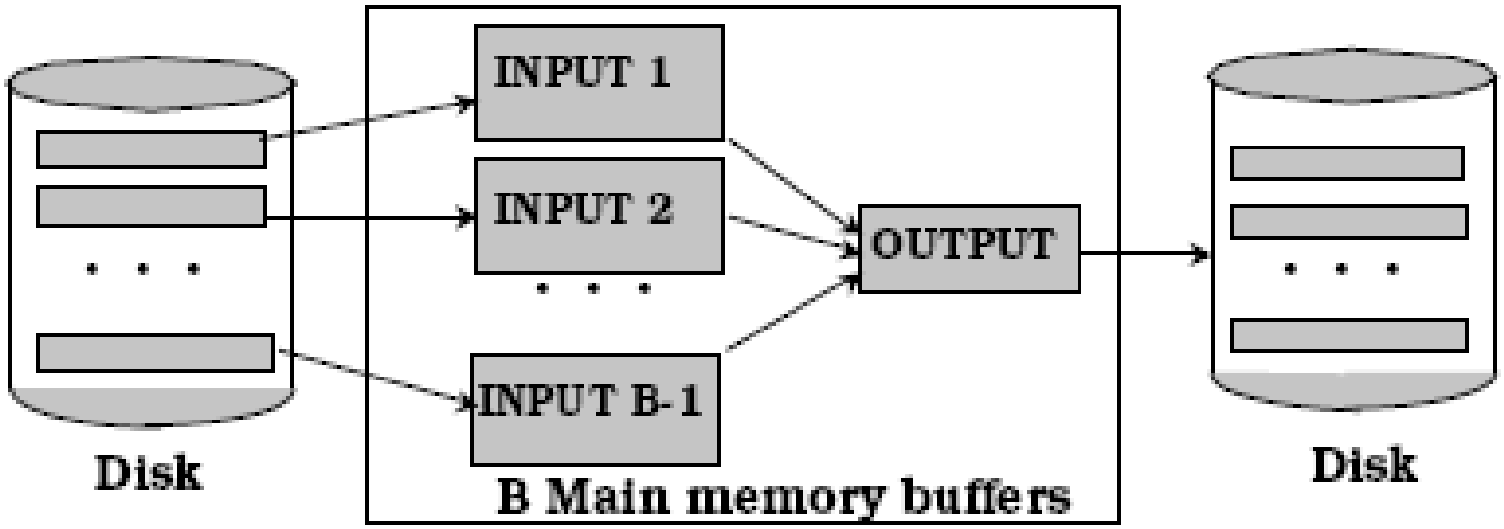
This pass requires B **pages** of buffer space.

- With *B* pages of buffer space, we can do a **(*B-1*)-way merge**.
  - Total number of I/O operations:

$$2 \cdot N \cdot \underbrace{\left(1 + \left\lceil \log_{B-1} \lceil N/B \rceil \right\rceil\right)}_{\text{number of passes}}$$

General ("(B-1)-Way") External Merge Sort: Recap

- To sort a file with *N* pages using *B* buffer pages:
  - Pass 0: Use *B* buffer pages. Produce sorted runs of *B* pages each.
  - Pass 1, 2, …, etc.: Merge *B-1* runs.



External Sorting: I/O Behavior

- Number of I/O operations required for sorting *N* pages with *B* buffer frames:

$$2 \cdot N \cdot \underbrace{\left(1 + \left\lceil \log_{B-1} \left\lceil N/B \right\rceil \right\rceil\right)}_{\text{number of passes}}$$

N	B=3	B=5	B=9	B=17	B=129	B=257
100	7	4	3	2	1	1
1,000	10	5	4	3	2	2
10,000	13	7	5	4	2	2
100,000	17	9	6	5	3	3
1,000,000	20	10	7	5	3	3
10,000,000	23	12	8	6	4	3
100,000,000	26	14	9	7	4	4
1,000,000,000	30	15	10	8	5	4

➢ What is the access pattern of these I/O operations?

Blocked I/O

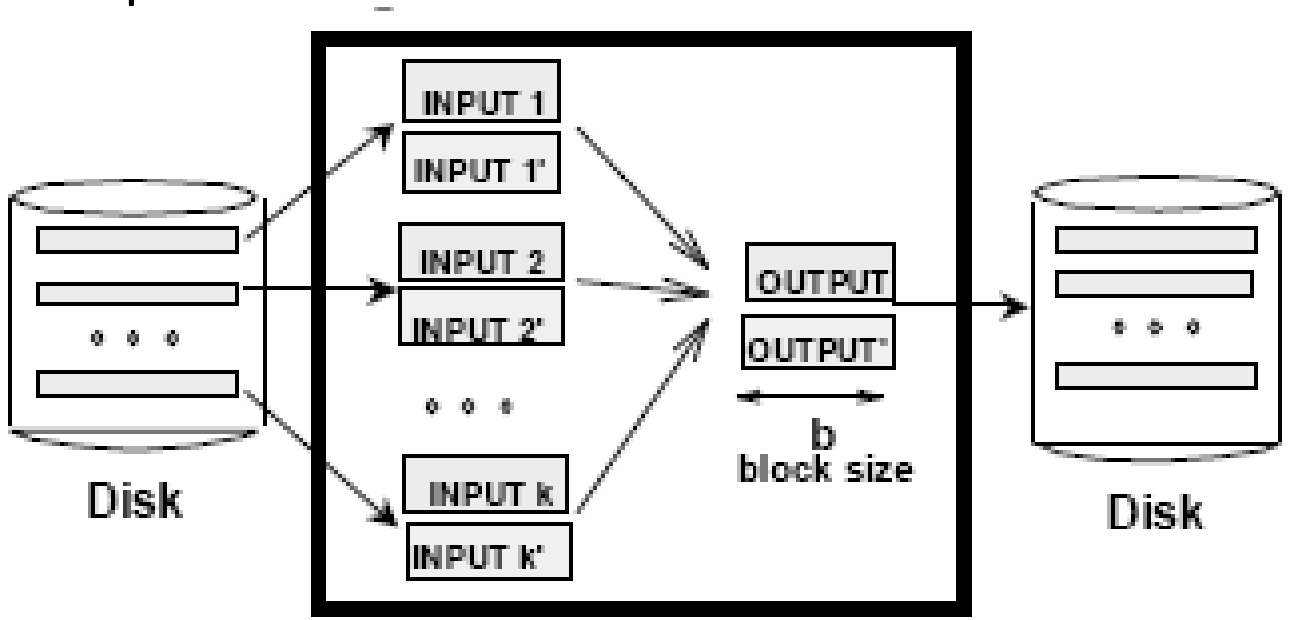
- We could improve the I/O pattern by reading blocks of, say, *b* pages sequentially at once during the merge phases.
  - Allocate *b* pages for each input (instead of just 1). In other words, make each buffer (input/output) be a block of *b* pages.
  - This reduces per-page I/O cost by a factor of ~ *b*.
  - The price we pay is a decreased fan-in during merges (resulting in an increased number of passes and more I/O operations).
  - In practice, main memory sizes are typically large enough to sort files with just 1 merge pass (even with blocked I/O).

External Sorting: Discussion

- External sorting follows the principle of **divide and conquer**.
  - This leads to a number of **independent** tasks.
  - These tasks could be executed **in parallel** (think of multi-processor machines or distributed databases).
- External sorting makes sorting very efficient. In most practical cases, **two passes** suffice to sort even huge files.
- There are a number of tweaks to tune sorting even further:
  - **Replacement sort:** Re-load new pages while writing out initial runs in Pass 0, thus increasing the initial run length.
  - **Double buffering:** Interleave page loading and input processing in order to hide disk latency.

Double Buffering

- To reduce wait time for I/O request to complete, we can prefetch into a “shadow block”.
  - Potentially, more passes; in practice, most files still sorted in 2-3 passes.

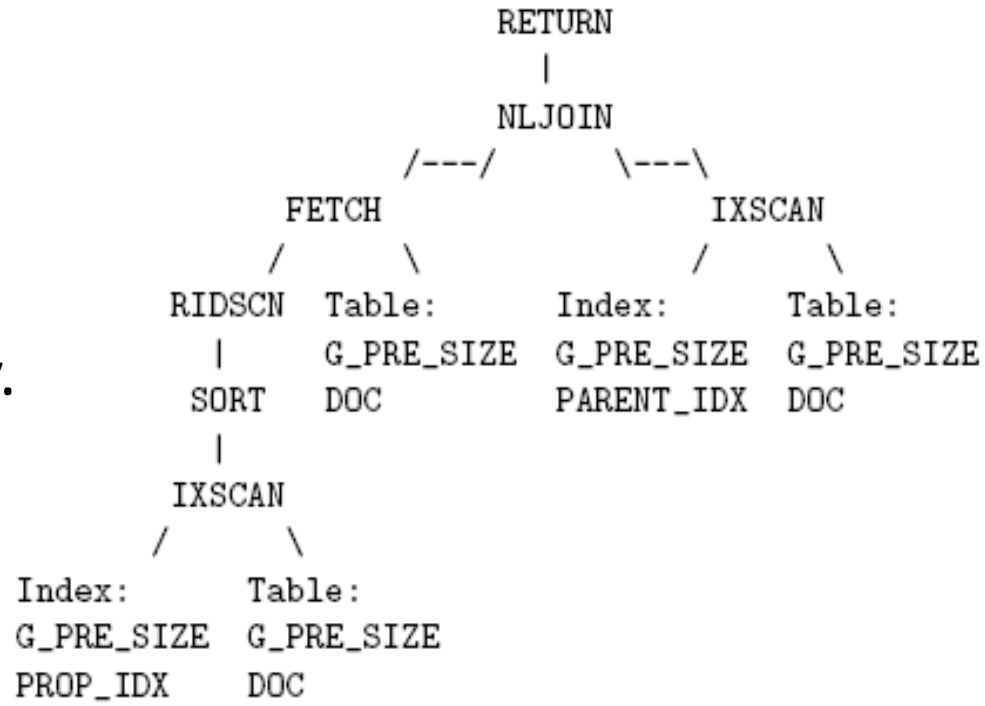


B main memory buffers, k-way merge

Query Plans

- External sorting is one instance of a (physical) **database operator**.
- Operators can be assembled into a query **execution plan**.
- Each plan operator performs one sub-task of a given query. Together, the operators of a plan evaluate the full query.

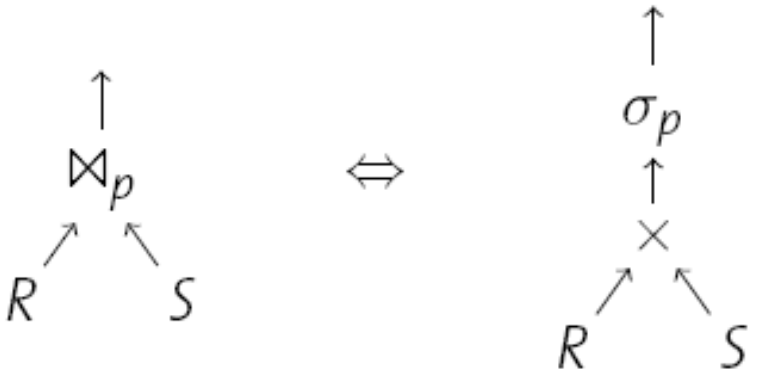
- An example IBM DB2 query execution plan:



➢ We'll have a deeper look into **join operators** next.

The Join Operator

- The join operator $\bowtie_p$ is actually a short-hand for a combination of cross product x and selection σ_p.



- One way to implement $\bowtie_p$ is to follow this equivalence:
  1. Enumerate all records in the cross product of *R* and *S*.
  2. Then pick those that satisfy *p*.
- More **advanced algorithms** try to avoid the obvious inefficiency in Step 1 (the size of the intermediate result is $|R|*|S|$).

Nested Loops Join

- The nested loops join is the straight forward implementation of the x-σ combination:

```
1 Function: nljoin (R, S, p)
2 foreach record r ∈ R do
3     foreach record s ∈ S do
4         if ⟨r, s⟩ satisfies p then
5             append ⟨r, s⟩ to result
```

- Let N_R and N_S the number of pages in R and S; let p_R and p_S be the number of records per page in R and S. The total number of disk reads is then:

$$N_R + \underbrace{p_R \cdot N_R}_{\#\text{ tuples in } R} \cdot N_S$$

Nested Loops Join: I/O Behavior

- The good news about `nljoin()` is that it needs only **three pages of buffer space** (two to read *R* and *S*, one to write the result).
- The bad news is its **enormous I/O cost**:
  - Assuming $p_R = p_S = 100$, $N_R = 1000$, $N_S = 500$, we need to read 1000 + (100*1000*500) disk pages.
  - With an access time of 10 ms for each page, this join would take 140 hours!
  - Switching the role of *R* and *S* to make *S* (the smaller one) the outer relation does not bring any significant advantage (disk pages = 500 + (100*500*1000)).

Block Nested Loops Join

- Again we can save random access cost by reading *R* and *S* in blocks of, say, b_R and b_S pages.

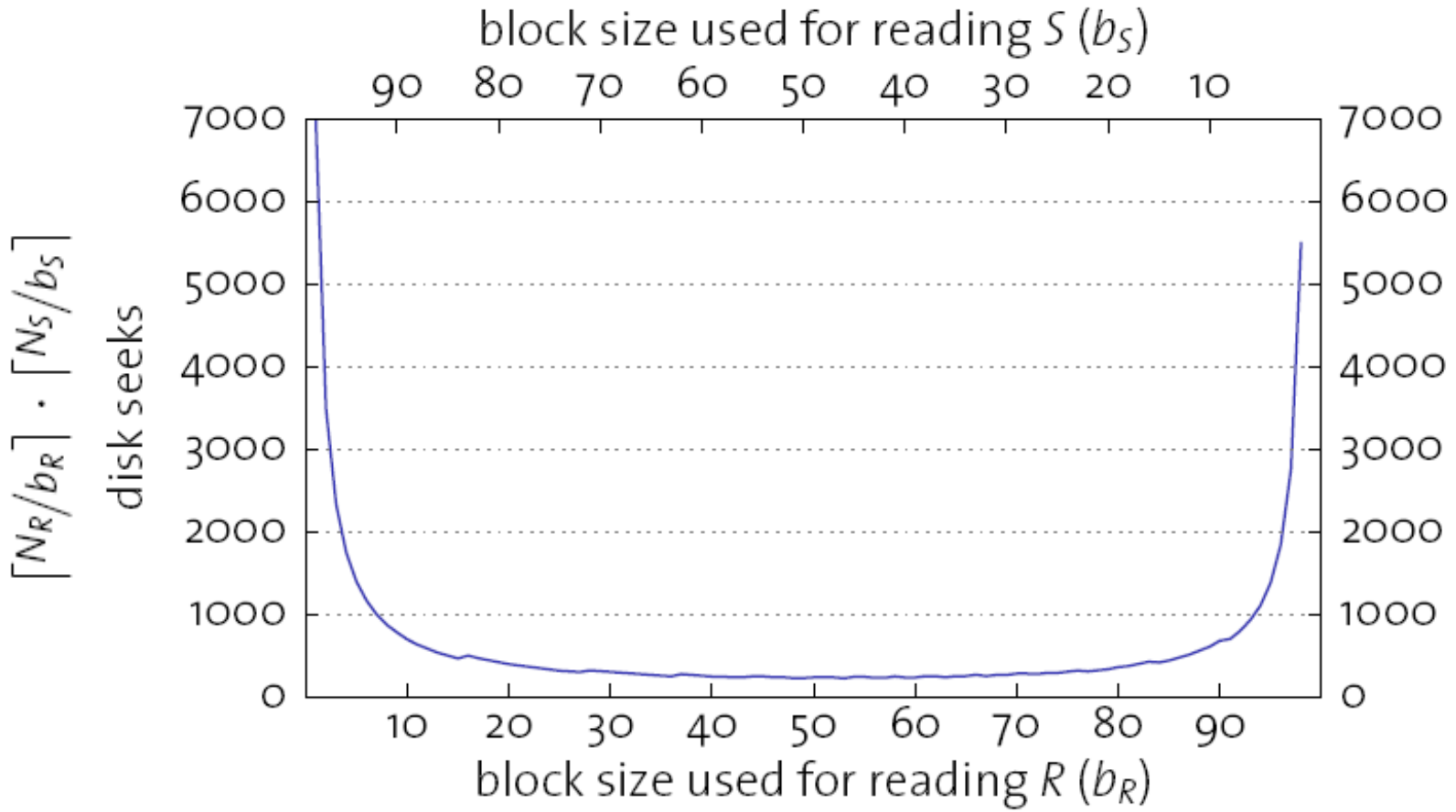
```
1 Function: block_nljoin (R, S, p)
2 foreach b_R-sized block in R do
3     foreach b_S-sized block in S do
4         find matches in current R- and S-blocks and
          append them to the result ;
```

- R is still read once, but now with only $\lceil N_R/b_R \rceil$ disk seeks.
- S is scanned only $\lceil N_R/b_R \rceil$ times now, and we need to perform $\lceil N_R/b_R \rceil \cdot \lceil N_S/b_S \rceil$ disk seeks to do this.

Choosing b_R and b_S

- E.g., buffer pool with $B = 100$ frames, $N_R = 1000$, $N_S = 500$

$b_R + b_S \sim 100$



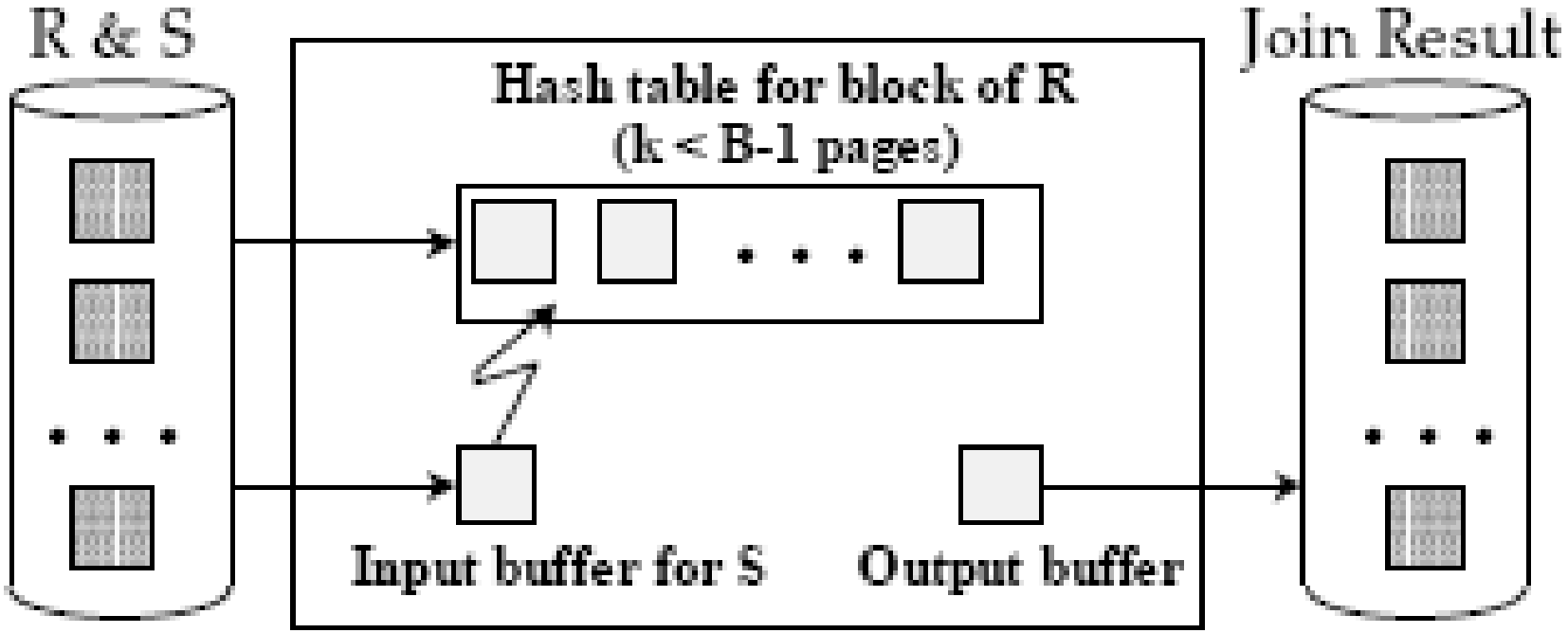
In-Memory Join Performance

- Line 4 in `block_nljoin(R,S,p)` implies an in-memory join between the *R*- and *S*-blocks currently in memory.
- Building a **hash table** over the *R*-block can speed up this join considerably.

```
1 Function: block_nljoin' (R, S, p)
2 foreach b_R-sized block in R do
3     build an in-memory hash table H for the current R-block ;
4     foreach b_S-sized block in S do
5         foreach record s in current S-block do
6             probe H and append matching ⟨r, s⟩ tuples to result ;
```

- Note that this optimization only helps **equi-joins.**

Using a Hash Table in Block Nested Loops Join



Index Nested Loops Join

- The index nested loops join takes advantage of an index on the inner relation (swap outer and inner if necessary):

```
1 Function: index_nljoin (R, S, p)
2 foreach record r ∈ R do
3     probe index using r and append all matching
      tuples to result ;
```

- The index must be compatible with the join condition *p*.
  - Hash indexes, e.g., only support equality predicates.

I/O Behavior

- For each record in *R*, we use the index to find matching *S*-tuples. While searching for matching *S*-tuples, we incur the following I/O costs for each tuple in *R*:
  1. Access the index to find its first matching entry: N_{idx} I/Os.
  2. Scan the index to retrieve all *n* matching *rids*. The I/O cost for this is typically negligible.
  3. Fetch the *n* matching *S*-tuples from their data pages.
     - For an unclustered index, this requires *n* I/Os.
     - For a clustered index, this only requires $\lceil n/p_S \rceil$ I/Os.
- Note that (due to 2 and 3), the cost of an index nested loops join becomes dependent on the size of the join result.

Index Access Cost

- If the index is a **B$^+$-tree index**:
  - A single index access requires the inspection of *h* pages (*h*: B$^+$-tree height).
  - If we repeatedly probe the index, however, most of these are cached by the buffer manager.
  - The effective value for N_{idx} is around 1–3 I/Os.
- If the index is a **hash index**:
  - Caching doesn't help us here (no locality in accesses to hash table).
  - A typical value for N_{idx} is 1-2 I/Os (due to overflow pages).
- Overall, the use of an index (over, e.g., a block nested loops join) pays off if the join picks out only few tuples from a big table.

Sort-Merge Join

- Join computation becomes particularly simple if both inputs are sorted with respect to the join attribute(s).
  - The merge join essentially merges both input tables, much like we did for sorting.
  - In contrast to sorting, however, we need to be careful whenever a tuple has multiple matches in the other relation:

A	B
"foo"	1
"foo"	2
"bar"	2
"baz"	2
"baf"	4

$\bowtie_{B=C}$

C	D
1	false
2	true
2	false
3	true

- Merge join is typically used for **equi-joins only**.

Merge Join

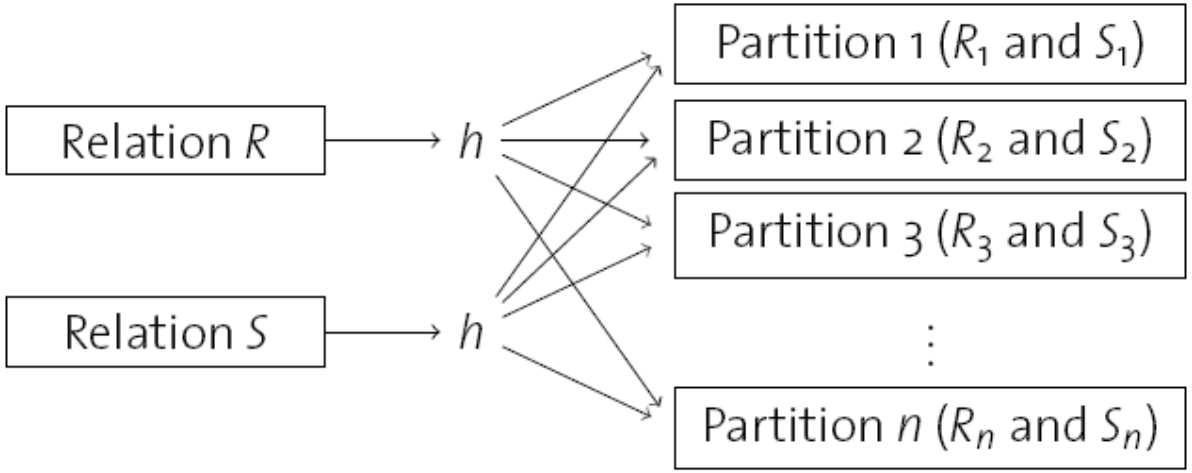
```
1  Function: merge_join (R, S, α = β)    // α, β: join columns in R, S
2  r ← position of first tuple in R ;    // r, s, s′: cursors over R, S, S
3  s ← position of first tuple in S ;
4  while r ≠ eof and s ≠ eof do          // eof: end of file marker
5      while r.α < s.β do
6          advance r ;
7      while r.α > s.β do
8          advance s ;
9      s′ ← s ;                          // Remember current position in S
10     while r.α = s′.β do               // All R-tuples with same α value
11         s ← s′ ;                      // Rewind s to s′
12         while r.α = s.β do            // All S-tuples with same β value
13             append ⟨r, s⟩ to result ;
14             advance s ;
15         advance r ;
```

Sort-Merge Join: I/O Behavior

- If both inputs are already sorted and there are no exceptionally long sequences of identical key values, the I/O cost of a merge join is $N_R + N_S$ (which is optimal).
- By using blocked I/O, these I/O operations can be done almost entirely as sequential reads.
- Sometimes, it pays off to explicitly sort a (unsorted) relation first, then apply merge join. This is particularly the case if a sorted output is beneficial later in the execution plan.
- The final sort pass can also be combined with merge join, avoiding one round-trip to disk and back.

Hash Join

- Sorting effectively brought related tuples into spatial proximity, which we exploited in the merge join algorithm.
- We can achieve a similar effect with hashing, too.
- Partition R and S into partitions $R_1, \ldots, R_n$ and $S_1, \ldots, S_n$ using the same hash function (applied to the join attributes).



- Observe that: $R_i \bowtie S_j = \varnothing$ for all $i \neq j$

Hash Join

- By partitioning the data, we reduced the problem of joining to smaller sub-relations R_i and S_i.
- Matching tuples are guaranteed to end up together in the same partition.
- We only need to compute $R_i \bowtie S_i$ (for all *i*).
- By choosing *n* properly (i.e., the hash function *h*), partitions become small enough to implement the $R_i \bowtie S_i$ as in-memory joins.
- The in-memory join is typically accelerated using a hash table, too. We already did this for the block nested loops join.

Hash Join Algorithm

```
1  Function: hash_join (R, S, α = β)
2  foreach record r ∈ R do
3      append r to partition R_{h(r.α)}
4  foreach record s ∈ S do
5      append s to partition S_{h(s.β)}
6  foreach partition i ∈ 1, ..., n do
7      build hash table H for R_i, using hash function h';
8      foreach block in S_i do
9          foreach record s in current S_i-block do
10             probe H and append matching tuples to result ;
```

Hash Join: Buffer Requirements

- We've assumed that we can create the necessary *n* partitions **in one pass** (note that: given *B* buffer pages, we want the number of pages for partition $R_i = N_{R_i} < (B-1)$ (1 for input, 1 for output, and the rest for R_i)).
- This works out if *R* consists of at most ~ $(B-1)^2$ pages.
  - We can write out at most *B-1* runs in one pass; each of them should be at most *B-1* pages in size.
  - Hashing doesn't guarantee us an even distribution. Since the actual size of each run varies, R must actually be smaller than $(B-1)^2$.
- Larger input tables require multiple passes for partitioning.

Hash Join vs. Sort-Merge Join

- Provided sufficient buffer space, hash join and sort-merge join both require $3*(N_R + N_S)$ I/Os.
  - For **reading** and **writing** each relation once during the partition/sort phase + **reading** each relation once during join/merge).

- The cost for hash join could considerably increase if partitions aren't uniformly sized, whereas sort-merge join is not sensitive to skew.

The Selection Operator (σ)

- Example:

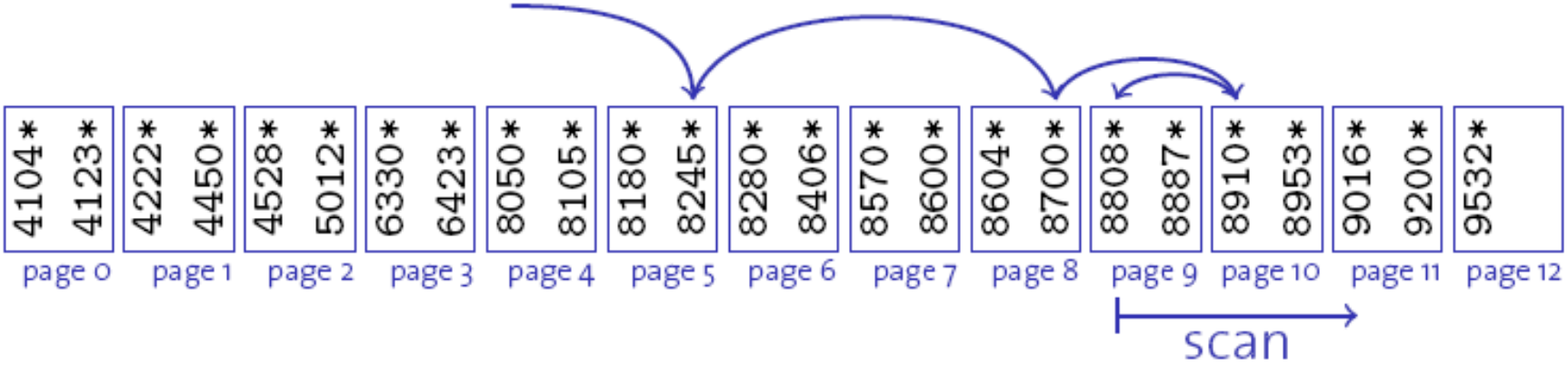
  SELECT *

  FROM Customer

  WHERE Zipcode ≥ 8800
- File scan is the basic approach:
  - Scan the entire Customer relation, checking each tuple's Zipcode and adding the tuple to the result if its Zipcode is ≥ 8800.
  - I/O cost ~ number of pages in Customer => Expensive!
- This approach always works (i.e., doesn't make any assumptions about file organization or index availability).
- We can improve this approach by exploiting the information in the selection condition (e.g., inequality predicate on Zipcode) and
  - leveraging file organization properties (e.g., physical sort order on Zipcode)
  - using a suitable index if available (e.g., B^+-tree on Zipcode)

Using Physical Sort Order for Selections

- Given a selection of the form $\sigma_{R.attr\ op\ value}(R)$, if R is physically sorted on R.attr, then instead of a full file scan, we can do a binary search on R.attr followed by a scan.
- Remember the "Sorted File" example of the Indexing lecture:



Using an Index for Selections

- Given a selection of the form $\sigma_{R.attr\ op\ value}(R)$:
  1. if op is equality, then we can either use a B^+-tree or a hash index on R.attr.
  2. if op is not equality, then we use a B^+-tree index on R.attr.
- Let's consider the second case. Index is used as follows:
  - Search the index to find the first qualifying entry.
  - Then scan the leaf pages to retrieve all qualifying R tuples.
- The first step takes only 2-3 I/Os. The cost of the second step depends on:
  - the number of qualifying R tuples
  - whether the index is clustered.

Using a Clustered Index for Selections

- Clustering affects the retrieval cost greatly.
- Example:
  - Consider “Zipcode ≥ 8800” on Customer.
  - Assume that there are 100,000 customer tuples in total occupying 1000 pages, and the selection should return 10% of all Customer tuples.
  - With a clustered B^+-tree index: 100 + 2-3 I/Os
  - With an unclustered B^+-tree index: up to 10,000 I/Os!

Using an Unclustered Index for Selections

- A possible refinement to improve the I/O cost of unclustered indexes:
  - Find the qualifying index data entries.
  - Sort the rid's of the data entries by their page-id components.
  - Fetch rid's in that order.

- The above approach ensures that when we bring in a page, all qualifying tuples on this page are retrieved one after the other, i.e., we look at each data page only once (though the number of such pages is likely to be higher than the case for using a clustered index).

General Selection Conditions

- In general, a selection condition is a Boolean combination of terms with:
  - conjunctions (e.g., R.attr1 op value1 AND R.attr2 op value2)
  - disjunctions (e.g., R.attr1 op value1 OR R.attr2 op value2)

- There are two approaches to evaluating selections with general conjunctive conditions :
  - based on the most selective access path
  - by intersecting rid sets

Using Most Selective Access Paths

- **Most selective access path** is an index (or file scan) that we estimate will require the fewest page I/Os.
- Find the most selective access path, retrieve tuples using it, and apply any remaining terms that don't match the index.
- Terms that match this index reduce the number of tuples retrieved; other terms are used to discard some of the retrieved tuples, but do not affect the number of tuples/ pages fetched.
- Example: "Zipcode ≥ 8800 AND Cust_ID=5555".
  - A B^+-tree index on Zipcode can be used; then "Cust_ID=5555" must be checked for each retrieved tuple.
  - Alternatively, a hash index on Cust_ID could be used; "Zipcode ≥ 8800" must then be checked.

Intersecting rid Sets

- If we have two or more matching indexes that store rid's at the leaves:
  - Get sets of rids of data records using each matching index.
  - Then intersect these sets of rids.
  - Retrieve the records and apply any remaining terms.
- Example: "Zipcode ≥ 8800 AND Cust_ID=5555".
  - If we have a B^+-tree index on Zipcode and a hash index on Cust_ID, we can retrieve rids of records satisfying "Zipcode ≥ 8800" using the first, rids of records satisfying "Cust_ID=5555" using the second, then intersect and retrieve the records.
- Refinement: Sort rid's by page-id's before the retrieval.

Selections with Disjunctions

- Example: “Zipcode ≥ 8800 OR Cust_ID=5555”.
- If one of the terms in the disjunction is a **file scan**, then the most selective access path has to be a file scan.
- If every term has a matching index, we can retrieve rid’s of candidate tuples using the indexes, take the **union** of the rid’s, and retrieve the records.
  - Refinement: Sort rid’s by page-id’s before the retrieval.

The Projection Operator

- Example query:

  SELECT DISTINCT R.sid, R.bid

  FROM Reserves R

- To implement projection, we have to do the following:
  1. Remove unwanted attributes.
  2. Eliminate any duplicates.
- Step 2 is the difficult part. Typically, systems try to avoid 2 whenever possible. In SQL, duplicate elimination has to be asked for explicitly.
- There are two basic algorithms for projection:
  - sorting-based
  - hashing-based

Sorting-based Projection

- Given an input relation R of size N pages, the conceptual sorting-based projection algorithm is as follows:
  - Scan R and generate a set of tuples with only the desired attributes (I/O cost = O(N)).
  - Sort this set by the combination of all attributes (I/O cost = O(NlogN)).
  - Scan the sorted set, comparing adjacent tuples and eliminating any duplicates (I/O cost = O(N)).
- We can get a better algorithm by adapting the external sort algorithm to do projection with duplicate elimination.

Sorting-based Projection

- Two modifications on the original external sort algorithm:
  - Modify Pass 0 to eliminate unwanted fields. Thus, runs of about 2B (as in external sort with replacement sort refinement) pages are produced, but tuples in these runs are smaller than the original input tuples.
  - Modify merging passes to eliminate duplicates. Thus, the number of result tuples is smaller than that of the input.

- Cost: In Pass 0, read original relation (size N), write out the same number of smaller tuples. In merging passes, fewer tuples are written out in each pass.

Hashing-based Projection

- Consider an input relation R of size N pages and a memory buffer of B pages. The algorithm is similar to hash-join.

- There are two phases in the algorithm:
  1. **Partitioning:** Read R using 1 input buffer. For each tuple, discard unwanted fields, apply hash function h1 to choose one of B-1 output buffers. Result is B-1 partitions (of tuples with no unwanted fields). 2 tuples from different partitions are guaranteed to be distinct.
  2. **Duplicate elimination:** For each partition, read it and build an in-memory hash table, using hash function h2 (<> h1) on all fields, while discarding duplicates.
     - If partition does not fit in memory, can apply hash-based projection algorithm recursively to this partition.

- Cost: For partitioning, read R, write out each tuple, but with fewer fields. This is read in the next phase. (overall cost: O(N)).

Discussion of Projection

- Sort-based approach is the standard.
  - It handles skew (i.e., non-uniform distribution of hash values) better.
  - Result is sorted.
- If an index on the relation contains all wanted attributes in its search key, can do index-only scan.
  - Apply projection techniques to data entries (much smaller!)
- If an ordered (i.e., tree) index contains all wanted attributes as prefix of search key, can do even better:
  - Retrieve data entries in order (index-only scan), discard unwanted fields, compare adjacent tuples to check for duplicates.

Set Operators

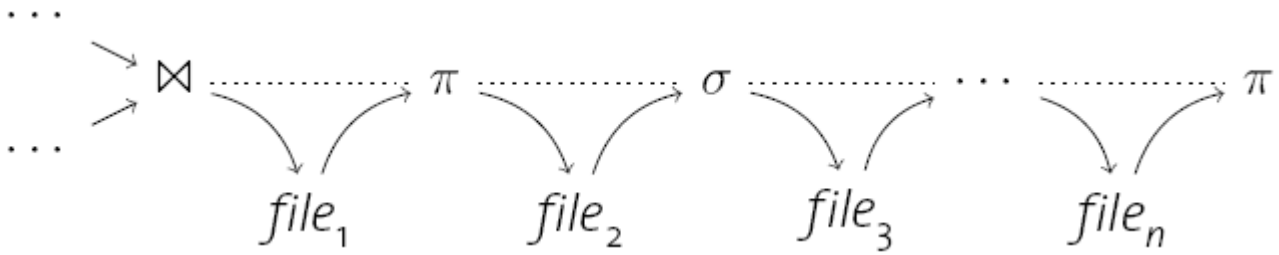
- R∩S, R x S, R U S, R – S
- Intersection and Cross-product are special cases of Join (i.e., equality on all fields, no join condition, respectively).
- Union (and Difference) build on duplicate elimination.
  - Sorting-based approach:
    - Sort both relations on combination of all attributes.
    - Scan sorted relations and merge them.
    - (Difference: During merge, write only R tuples to result that are not in S.)
  - Hashing-based approach:
    - Partition R and S using hash function h1.
    - For each S-partition, build an in-memory hash table (using h2), scan the corresponding R-partition, and add tuples to table while discarding duplicates.
    - (Difference: Write only R tuples that are not in the hash table.)

Aggregate Operators

- AVG, MIN, MAX, SUM, COUNT
- Without GROUP BY:
  - In general, scan the whole relation, maintaining running information (e.g., <total, count> for AVG).
  - Index can be used if the search key includes all attributes in SELECT or WHERE clauses.
- With GROUP BY:
  - Sort on group-by attributes, then scan relation and compute aggregate for each group. (Can improve upon this by combining sorting and aggregate computation.)
  - Similar approach based on hashing on group-by attributes.
  - Given tree index whose search key includes all attributes in SELECT, WHERE and GROUP BY clauses, can do index-only scan; if group-by attributes form prefix of search key, can retrieve data entries/tuples in group-by order.

Orchestrating Operator Evaluation

- So far we have assumed that all database operators consume and produce files (i.e., on-disk items):



- Obviously, this causes a lot of I/O.
- In addition, we suffer from long response times:
  - An operator cannot start computing its result before all its input files are fully generated (i.e., “materialized”).
  - Effectively, all operators are executed in sequence.

Pipelined Evaluation

- Alternatively, each operator could pass its result directly on to the next operator (without persisting it to disk first).
- Don’t wait until entire file is created, but propagate output immediately.
- Start computing results as early as possible, i.e., as soon as enough input data is available to start producing output.
- This idea is referred to as pipelining.
- The granularity in which data is passed may influence performance:
  - Smaller chunks reduce the response time of the system.
  - Larger chunks may improve the effectiveness of (instruction) caches.
  - Actual systems typically operate tuple at a time.

Unix: Pipelines of Processes

- Unix uses a similar mechanism to communicate between processes ("operators"):

```
find . -size +1000k | xargs file \
  | grep -i XML | cut -d: -f1
```

- Execution of this pipe is driven by the rightmost operand:
  - To produce a line of output, `cut` only needs to see the next line of its input: `grep` is requested to produce this input.
  - To produce a line of output, `grep` needs to request as many input lines from the `xargs` process until it receives a line containing the string `"XML"`.
  - ...
  - Each line produced by the `find` process is passed through the pipe until it reaches the `cut` process and eventually is echoed to the terminal.

The Volcano Iterator Model

- The calling interface used in database execution run times is very similar to the one used in Unix process pipelines.
- In databases, this interface is referred to as **open-next-close interface** or **Volcano iterator model**.
- Each operator implements the functions
  - **open ()** : initialize the operator's internal states.
  - **next ()** : produce and return the next result tuple.
  - **close ()** : clean up all allocated resources (typically after all tuples have been processed).
- All state is kept inside each operator.

➢ Goetz Graefe, "Volcano - An Extensible and Parallel Query Evaluation System, IEEE TKDE, 6:1, 1994.

Example: Selection

- Input operator *R*, predicate *p*.

```
1 Function: open ()
2 R.open () ;
```

```
1 Function: close ()
2 R.close () ;
```

```
1 Function: next ()
2 while ((r ← R.next ()) ≠ eof) do
3     if p(r) then
4         return r ;
5 return eof ;
```

Example: Nested Loops Join

```
1 Function: open ()
2 R.open () ;
3 S.open () ;
4 r ← R.next () ;
```

```
1 Function: close ()
2 R.close () ;
3 S.close () ;
```

```
1 Function: next ()
2 while (r ≠ eof) do
3     while ((s ← S.next ()) ≠ eof) do
4         if p(r, s) then
5             return ⟨r, s⟩ ;
6     S.close () ;
7     S.open () ;
8     r ← R.next () ;
9 return eof ;
```

Blocking Operators

- Pipelining reduces memory requirements and response time since each chunk of input is propagated to the output immediately.
- Some operators cannot be implemented in such a way.
  - (external) sorting
  - hash join
  - grouping and duplicate elimination over unsorted input
- Such operators are said to be blocking.
- Blocking operators consume their entire input before they can produce any output.
  - The data is typically buffered (i.e., “materialized”) on disk.

Summary of Techniques

- **Divide and Conquer**
  - Many database algorithms derive their power from chopping a large input problem into smaller, manageable pieces, e.g.,
    - run generation and merging in external sorting,
    - partitioning according to a hash function (hash join).
- **Blocked I/O**
  - Reading and writing chunks of pages at a time can significantly reduce the degree of random disk access.
    - This “trick” was applicable to most operators we saw.
- **Pipelined Processing**
  - The Volcano iterator model can save memory and reduce response time by avoiding the full materialization of intermediate results if possible.