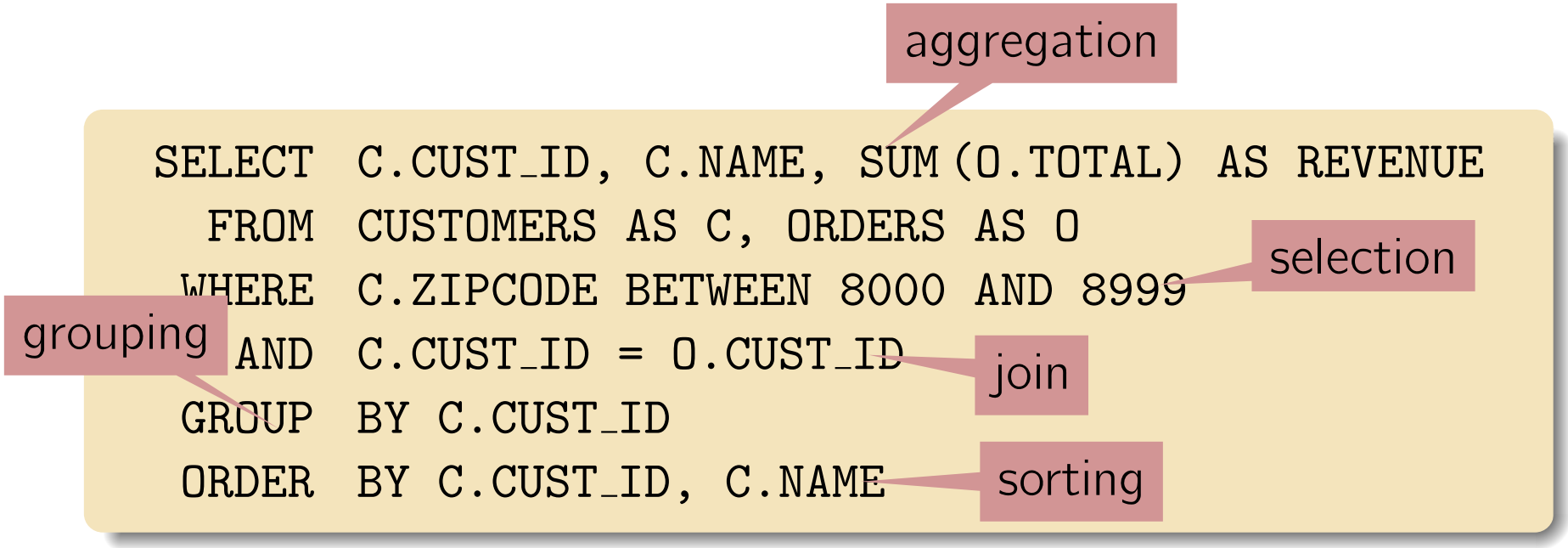
Architecture and Implementation of Database Systems (Summer 2018)

Jens Teubner, DBIS Group
jens.teubner@cs.tu-dortmund.de

Summer 2018

Part V

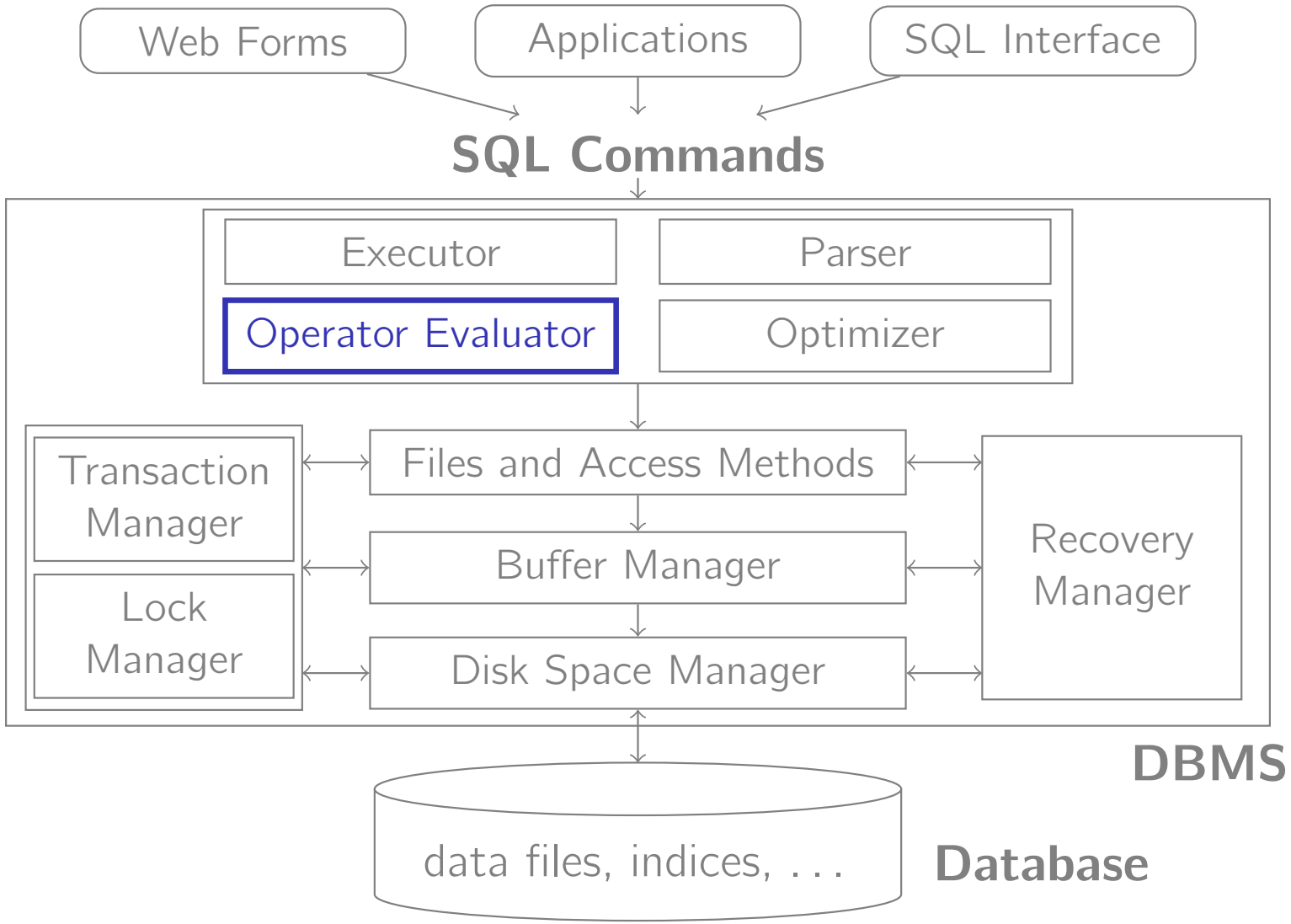
Query Processing



A DBMS needs to do a number of tasks

- with **limited memory resources**,
- over **large amounts of data**,
- yet **as fast as possible**.

Query Processing



Sorting

Sorting is a core database operation with numerous applications:

- A SQL query may explicitly request **sorted output**:

```
SELECT A, B, C FROM R ORDER BY A
```

- **Bulk-loading a B$^+$-tree** presupposes 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 meet later in this course).

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**.

- **Pass 0** sorts each of the 2^k input pages individually and in **main memory**, resulting in 2^k sorted runs.
- **Subsequent passes merge** pairs of runs into larger runs. Pass n produces 2^{k-n} runs.
- **Pass** k leaves only one run left, the sorted overall result.

During each pass, we read every page in the file. Hence, $(k+1) \cdot N$ page reads and $(k+1) \cdot N$ page writes are required to sort the file.

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.

$\vdots$

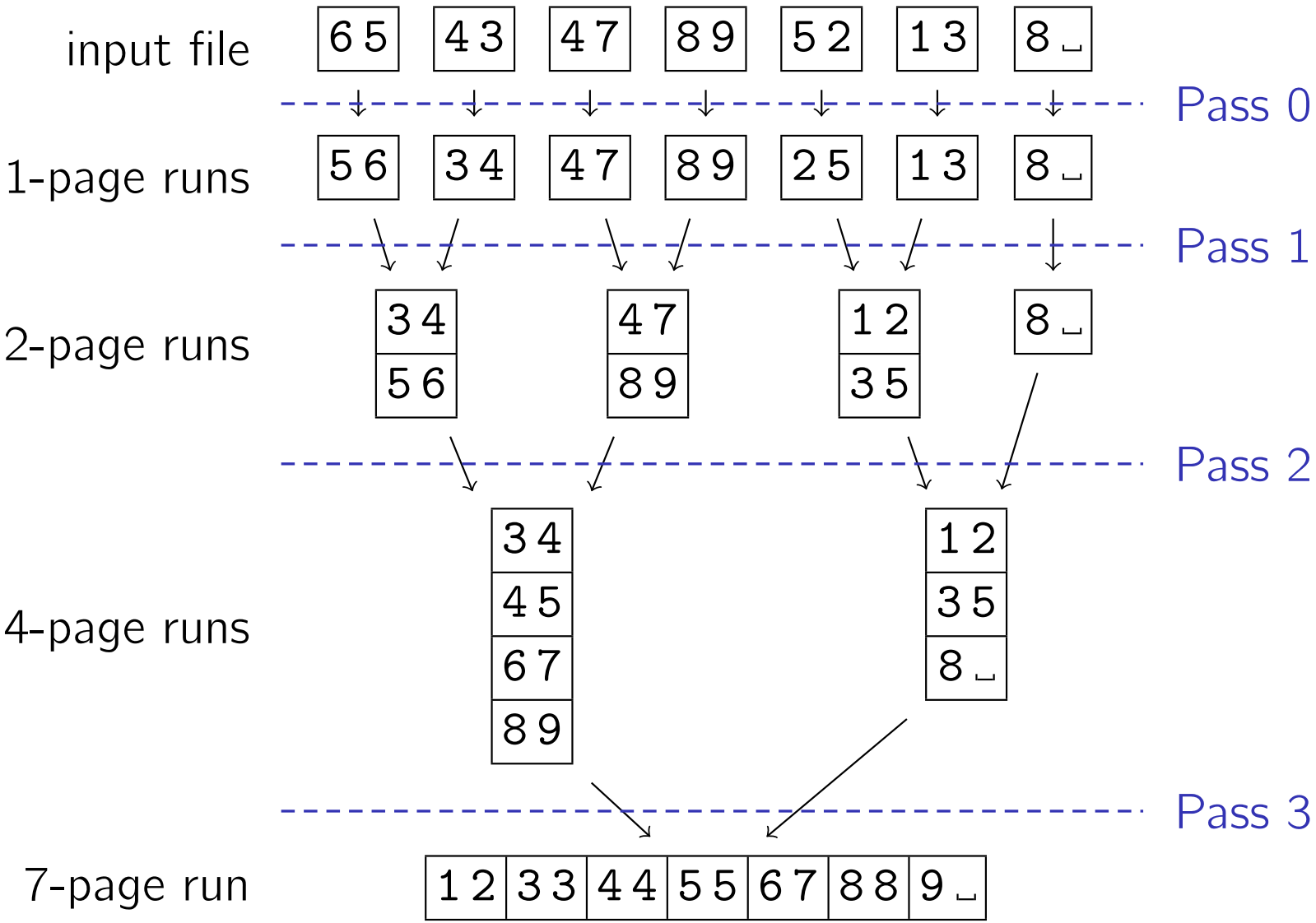
Pass n (**Input:** 2^{k-n+1} sorted runs; **Output:** 2^{k-n} sorted runs)

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.

$\vdots$

Illustration / 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

  $\rightarrow$ $2 \cdot N$ I/O operations per pass.

- The number of passes is $\underbrace{1}_{\text{Pass } 0} + \underbrace{\lceil \log_2 N \rceil}_{\text{Passes } 1 \ldots k}$.

- Total number of I/O operations:

$$2 \cdot N \cdot (1 + \lceil \log_2 N \rceil) \quad .$$

✎ **How many I/Os does it take to sort an 8 GB file?**

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 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 (↗ slide 156):

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 (↗ slide 158):

→ Total number of I/O operations: $2 \cdot N \cdot (1 + \lceil \log_2 \lceil N/B \rceil \rceil)$.

✎ **How many I/Os does it now take to sort an 8 GB file?**

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**.

$\rightarrow$ Total number of I/O operations: $2 \cdot N \cdot \left(1 + \left\lceil \log_{B-1} \lceil N/B \rceil \right\rceil\right)$.

✎ **How many I/Os does it now take to sort an 8 GB file?**

External Sorting: I/O Behavior

Sorting N pages with B buffer frames requires

$$2 \cdot N \cdot \left(1 + \left\lceil \log_{B-1} \left\lceil N/B \right\rceil \right\rceil\right)$$

I/O operations.

✎ **What is the access pattern of these I/Os?**

Blocked I/O

We could improve the I/O pattern by reading **blocks** of, say, b pages at once during the **merge** phases.

- Allocate b pages for each input (instead of just one).
- **Reduces per-page I/O cost** by a factor of $\approx b$.
- The price we pay is a **decreased fan-in** (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 one merge pass**, even with blocked I/O.

✎ **How long does it take to sort 8 GB (counting only I/O cost)?**

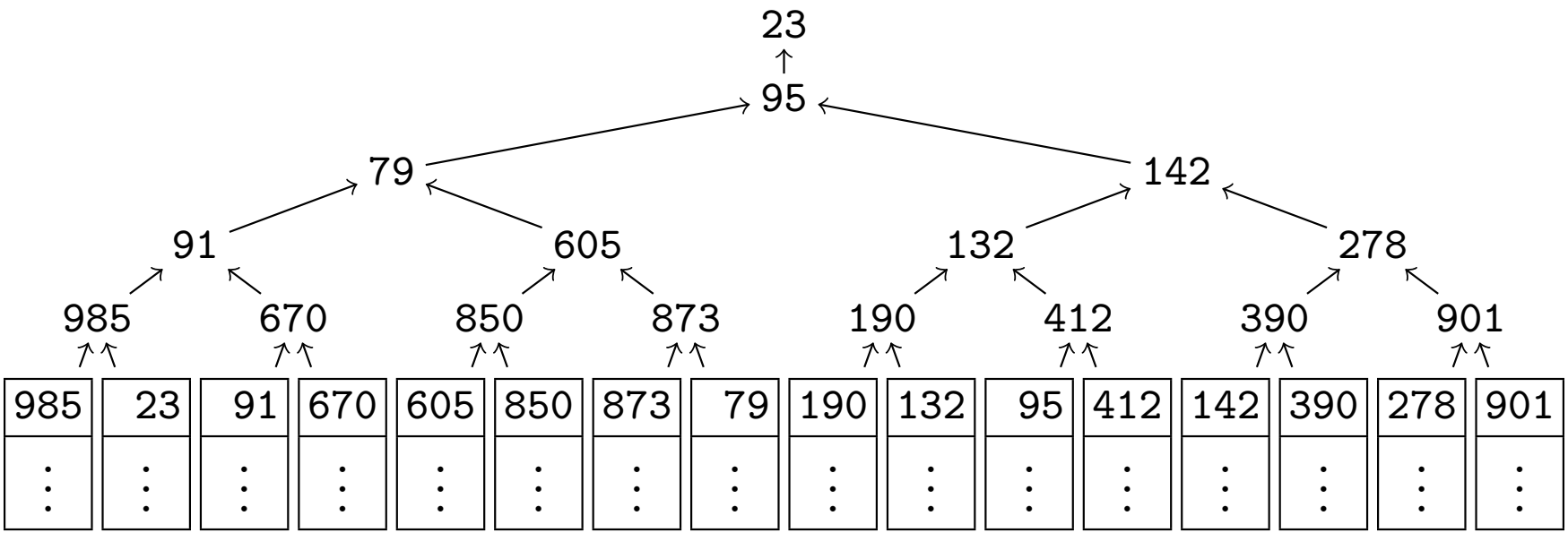
1000 buffer pages, 8 KB each; 10 ms total disk latency

- Without blocked I/O: $\approx 4 \cdot 10^6$ disk seeks (11.6 h) + transfer of $\approx 6 \cdot 10^6$ disk pages (17 min)
- With blocked I/O (32 page blocks): $\approx 6 \cdot 32,768$ disk seeks (33 min) + transfer of $\approx 8 \cdot 10^6$ disk pages (22 min)

Selection Trees

Choosing the next record from $B-1$ (or $B/b-1$) input runs can be quite CPU intensive ($B-2$ comparisons).

- Use a **selection tree** to reduce this cost.
- *E.g.*, "tree of losers" (↗ D. Knuth, TAoCP, vol. 3):

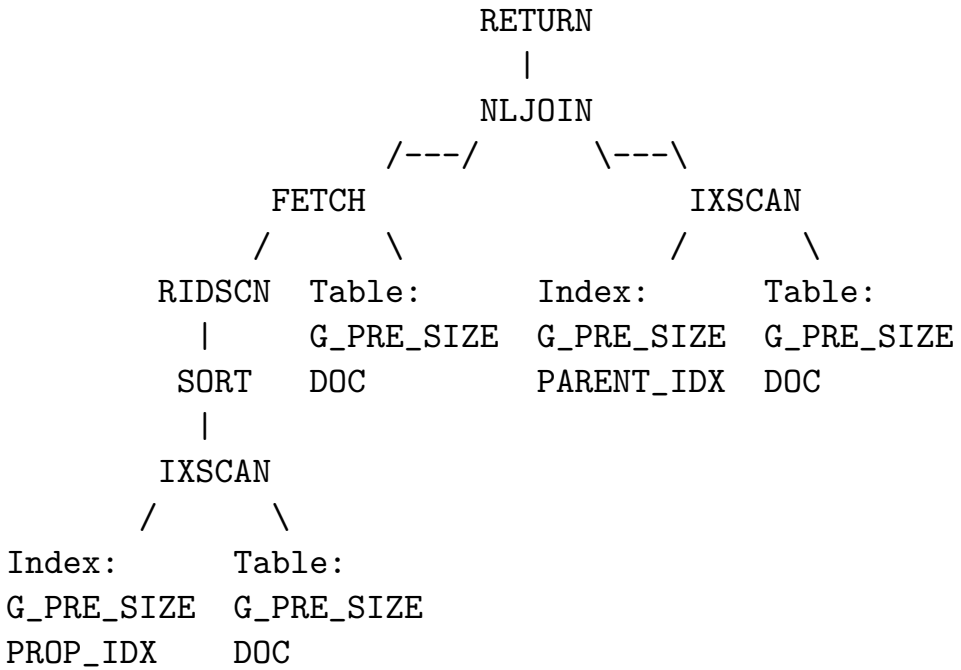


- This cuts the number of comparisons to $\log_2(B-1)$.

External Sorting—Discussion

- External sorting follows the principle of **divide and conquer**.
  - This leads to a number of **independent** tasks.
  - These tasks can 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.

Query Plans



Actual DB2 execution plan.

- 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.

- We'll have a deeper look into some of these operators next.

B^+-Trees for Simple Range Queries

Consider (again, see slide 60) the following query:

```
SELECT  *
  FROM  CUSTOMERS
 WHERE  ZIPCODE BETWEEN 8800 AND 8999
```

Possible execution strategy (using a B^+-tree index):

- **Locate** first record where `ZIPCODE` ≥ 8800.
- Then **scan** B^+-tree leaves until `ZIPCODE` ≤ 8999.
- If index is **not clustered**, **fetch** corresponding tuple for each entry.

B^+-Trees for Simple Range Queries

✎ **Execution cost** of this evaluation strategy?

What would be the cost of answering the query **without** an index?

Execution Cost For Index Scans

Non-clustered index: every record fetch causes a new I/O request.

Example:

- `CUSTOMERS` table with **1,000,000 tuples**
- **50 records per data page** (*i.e.*, 20,000 pages for `CUSTOMERS`)
- filter selectivity of **5 %**

Thus:

→ 50,000 tuples match filter predicate

→ **50,000 I/O requests** to fetch tuples (2.5 times the entire table!)
(Plus 5 % of all index leaves, which should be few hundred pages.)

Execution Cost For Index Scans

The situation is **even worse**:

- All tuple fetches are **"random I/O"**.
  - → $50{,}000 \times 15\,\text{ms} = 750\,\text{seconds}$![12]

To compare:

- A **full table scan** would require only **20,000 I/Os**, which can be read using **sequential I/O**.
  - → $15\,\text{ms} + \frac{20{,}000 \times 8\,\text{kB}}{100\,\text{MB/s}} = 1.6\,\text{seconds}$

→ A full table scan **can** be substantially faster than an index scan with tuple fetch.

Predictable performance typically more important than actual/average/... runtime.

[12] Good server drives may have access times $\ll 15\,\text{ms}$.

Better Evaluation Strategies

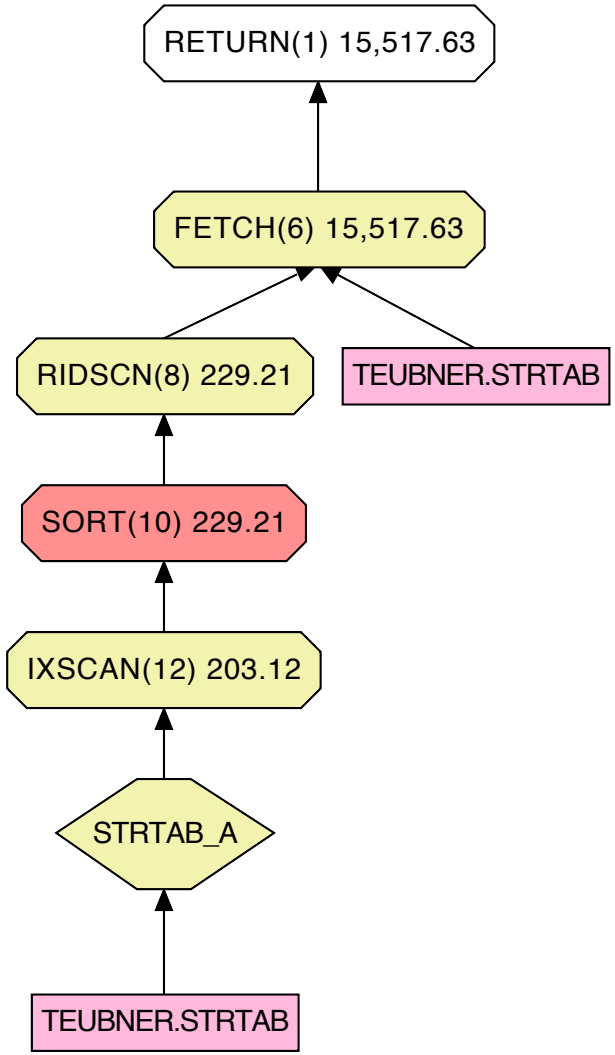
The "randomness" of tuple fetches can be avoided as follows:

1. **Scan index** to retrieve all matching RIDs.
2. **Sort** those RIDs to match the physical order on disk.
3. **Fetch** tuples from disk in disk order.

Consequences:

- Read each data page **at most once**, and only read **necessary pages**.

DB2 chooses this strategy for low to medium selectivities (see plan on the right).



✎ **What does this mean with regard to "cost vs. selectivity"?**

Prefetching

A sorted RID list allows aggressive **prefetching**.

```
List Prefetch Preparation
| Access Table Name = TEUBNER.STRTAB ID = 2,5
| | #Columns = 4
| | Skip Inserted Rows
| | Avoid Locking Committed Data
| | Currently Committed for Cursor Stability
| | RID List Fetch Scan
| | Fetch Using Prefetched List
| | | Prefetch:  14706 Pages
| | Lock Intents
| | | Table:  Intent Share
| | | Row :  Next Key Share
| | Sargable Predicate(s)
| | | #Predicates = 2
| | | Return Data to Application
| | | | #Columns = 4
Return Data Completion
```

Index Re-Scans

If data records are kept in a **clustered index**, that index will be **re-scanned** very frequently.

→ This is a frequent pattern that also occurs for other reasons.

If keys to be searched are **sorted**, such repeated index scans become particularly efficient.

- Keep full **root-to-leaf path** of index in memory.
  (That's only a few pages, since B^+-trees are not deep.)
- When re-scanning, **start from deepest level** possible.
  - → Nagivate back up, as long as new search key is larger than maximum key of current node.
  - → Can use **fence keys** to decide.

B$^+$-Trees for Simple Range Queries

Some queries **don't actually need** the tuple fetch:

```
SELECT  ZIPCODE
  FROM  CUSTOMERS
 WHERE  ZIPCODE BETWEEN 8800 AND 8999
```

- The index already contains everything needed to answer that query.
- This allows for **index-only** retrieval of `ZIPCODE` values.
- More queries are eligible to index-only retrieval than one might intuitively think.
  - aggregates, existence queries, joins, etc.

More Index-Only Queries

To permit more index-only processing for more queries, add columns to the index, even when they are not part of the key.

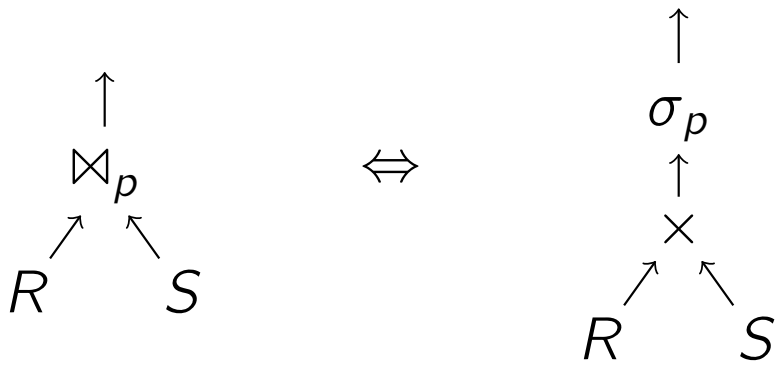
IBM DB2:

```
CREATE INDEX IndexName
    ON TableName ( col_1 , col_2 , ... , col_n )
INCLUDE ( col_1 , col_2 , ... , col_m )
```

(The `INCLUDE` clause is allowed in DB2 only if the index is declared as `UNIQUE`.)

The Join Operator $\bowtie$

The **join operator** $\bowtie_p$ is actually a short-hand for a combination of **cross product** $\times$ 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 $|R| \cdot |S|$).

Nested Loops Join

The **nested loops join** is the straightforward implementation of the σ–$\times$ 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

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

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 $1000 + \left(5 \cdot 10^7\right)$ 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.

Note that reading data page-by-page (even tuple-by-tuple) means that **every** I/O suffers the disk latency penalty, even though we process both relations in sequential order. An exception is the situation when S fully fits into the buffer pool ✎.

Block Nested Loops Join

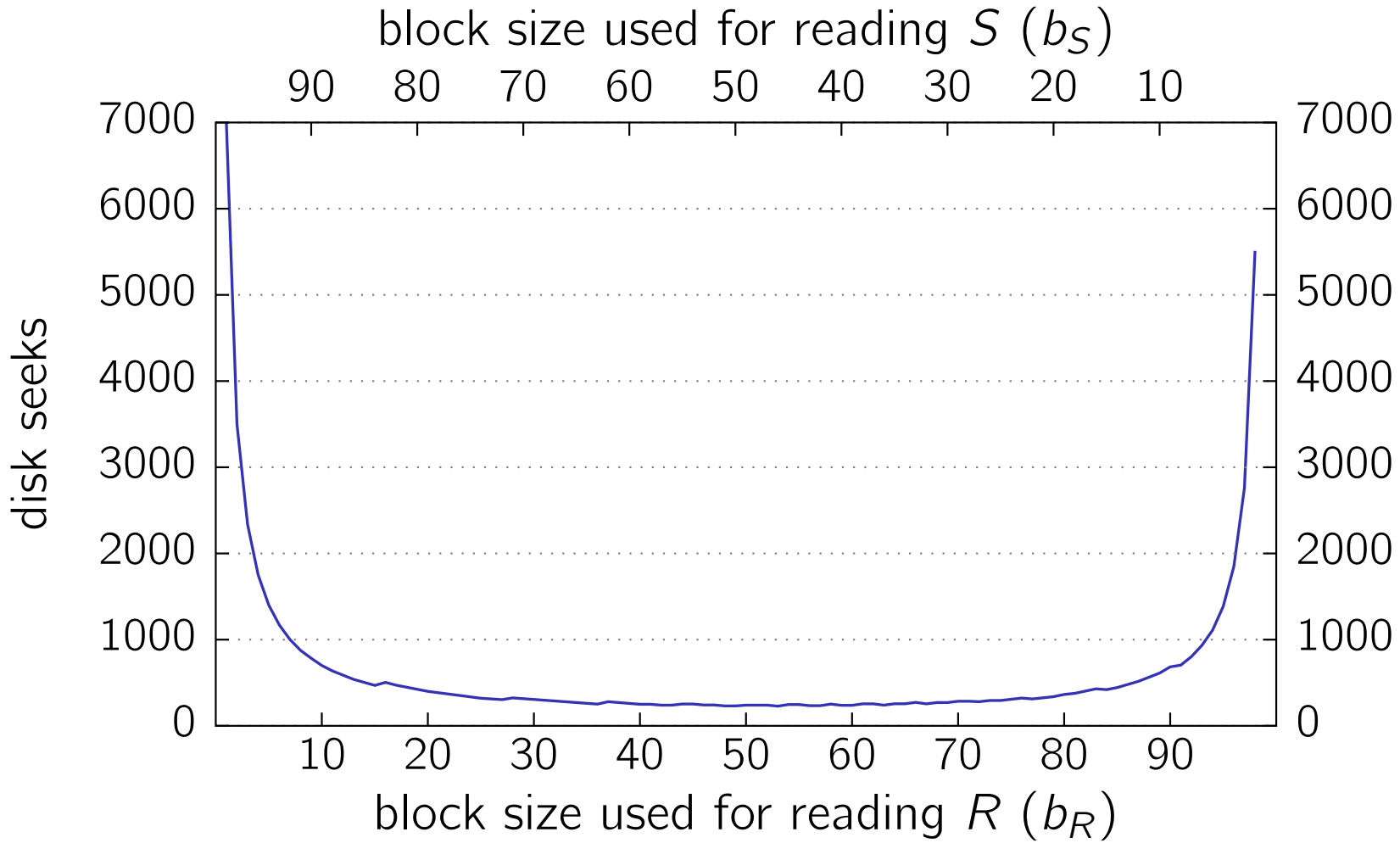
Again we can save random access cost by reading R and S in **blocks** of, say, b_R and b_S pages.

```
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, $N_R = 1000$, $N_S = 500$:



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**.

Index Nested Loops Join

The **index nested loops join** takes advantage of an index on the **inner** relation (swap *outer* $\leftrightarrow$ *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 indices, *e.g.*, only support equality predicates.
  - Remember the discussion about composite keys in B$^+$-trees (↗ slide 107).
- Such predicates are also called **sargable** (SARG: search argument ↗ Selinger *et al.*, *SIGMOD 1979*)

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.[13]
- 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.

[13] h: B$^+$-tree height

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.
- 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**.

```
1  Function: merge_join(R, S, α = β)    // α, β: join cols 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 ;
```

I/O Behavior

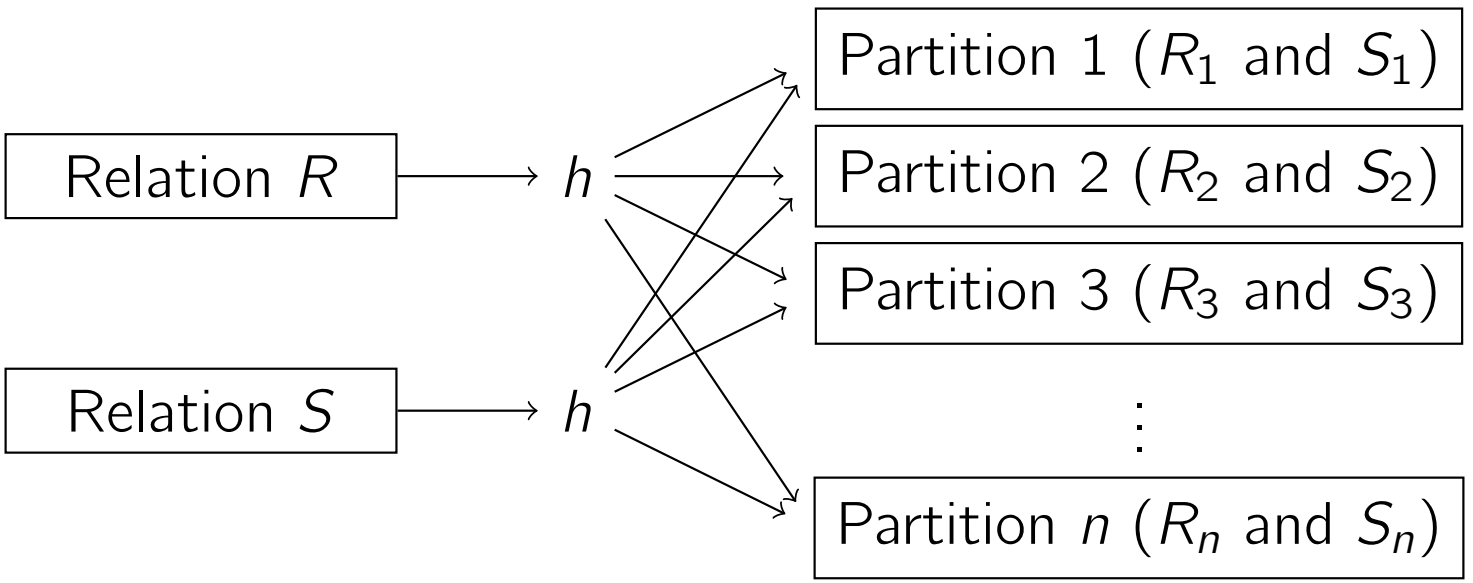
- If both inputs are already sorted **and** there are no exceptionally long sequences of identical key values, the I/O cost of a merge join is $N_R + N_S$ (which is optimal).
- By using **blocked I/O**, these I/O operations can be done almost entirely as **sequential** reads.
- Sometimes, it pays off 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.

What is the worst-case behavior of merge join?

If both join attributes are constants and carry the same value (*i.e.*, the result is the Cartesian product), merge join degenerates into a nested loops join.

Hash Join

- Sorting effectively brought related tuples into **spacial 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 (↗ slide 181).

Use a **different** hash function h' for the in-memory join.

✎ **Why?**

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 we want $N_{R_i} < (B-1)$).
- This works out if R consists of **at most** $\approx (B-1)^2$ pages.

Why $(B-1)^2$? Why $\approx$?

- Larger input tables require **multiple passes** for partitioning.

Hash Join vs. Sort-Merge Join

Provided sufficient buffer space ($B \gtrsim \sqrt{N}$), hash join and sort-merge join **both** require $3\,(N_R + N_S)$ I/Os.[14]

- For **sort-merge join**, **both** relations need to be smaller than $B(B-1)$ (assuming we need to sort before the join), *i.e.*,

$$N_R < B(B-1) \text{ and } N_S < B(B-1) \quad .$$

- In case of **hash join**, only the **inner relation** needs to be partitioned into $(B-1)$-sized chunks, *i.e.*,

$$\min(N_R, N_S) \lesssim (B-1)^2 \quad .$$

- The cost for hash join could considerably increase if partitions aren't uniformly sized.

[14] Read/write both relations to partition/sort; read both relations to join.

Implementing Grouping and Duplicate Elimination

- Challenge is to find **identical tuples** in a file.
- This task has obvious similarities to a **self-join** based on all of the file's columns.
  - $\rightarrow$ Could use a **hash join-like** algorithm or **sorting** to implement duplicate elimination or grouping.
- See **exercises** for further details.

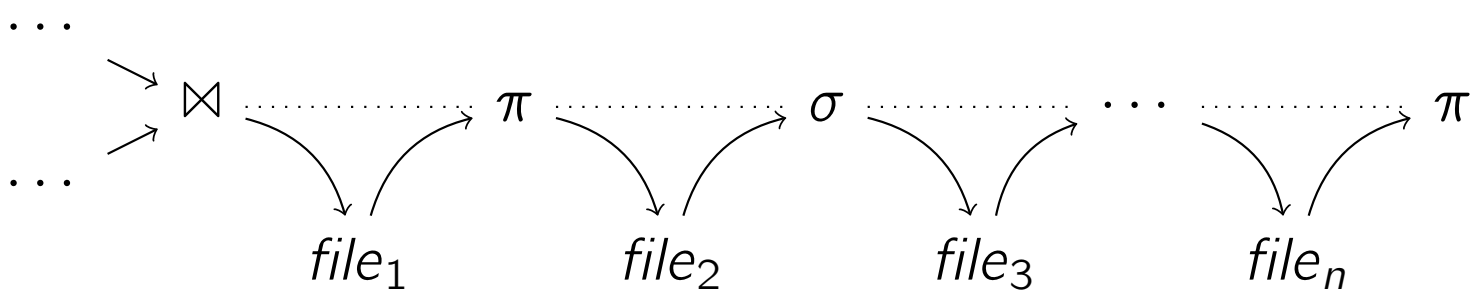
Other Database Operators

Projection π

- Text book-style processing of π implies
  (a) discarding unwanted fields and
  (b) eliminating duplicates.
- Implementing (a) amounts to a straightforward **file scan**. We have mentioned implementations for (b) a moment ago.
- Typically, systems try to **avoid** (b) whenever possible. In SQL, duplicate elimination has to be asked for explicitly.

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 (**"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 runtimes 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 Extensibel and Parallel Query Evaluation System. *Trans. Knowl. Data Eng.* vol. 6, no. 1, February 1994.

Example: Selection (σ)

- 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 ;
```

How would you implement a Volcano-style 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.

Which ones?

- Such operators are said to be **blocking**.
- Blocking operators consume their entire input before they can produce any output.
  - The data is typically buffered ("materialized") on disk.

Techniques We Saw In This Chapter

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.