

# Architecture and Implementation of Database Systems

## Query Processing

---

Ralf Möller  
Hamburg University of Technology

# Acknowledgements

- The course is partially based on <http://www.systems.ethz.ch/education/past-courses/hs08/arch-dbms> by Jens Teubner, ETH Zürich

```
SELECT C.CUST_ID, C.NAME, SUM(O.TOTAL) AS REVENUE
FROM CUSTOMERS AS C, ORDERS AS O
WHERE C.ZIPCODE BETWEEN 8000 AND 8999
AND C.CUST_ID = O.CUST_ID
GROUP BY C.CUST_ID
ORDER BY C.CUST_ID, C.NAME
```

aggregation

selection

grouping

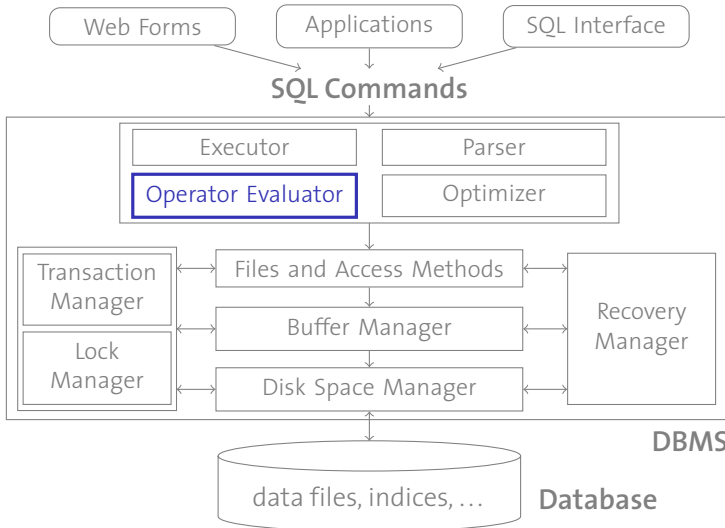
join

sorting

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<sup>+</sup>-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.

⋮

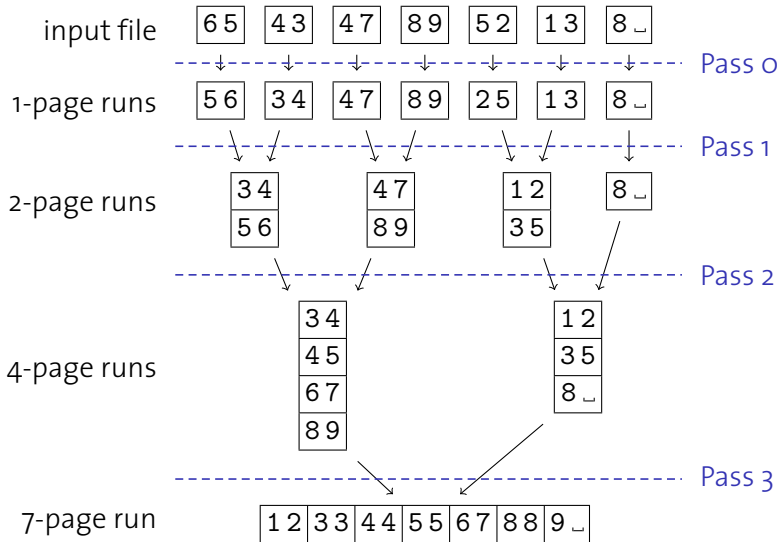
Pass  $n$  (Input:  $2^{k-n+1}$  sorted runs; Output:  $2^{k-n}$  sorted runs)

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.

⋮

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

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

- ▶ Total number of I/O operations:

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



**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 122):

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 124):

→ 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  $r_1 \dots r_{B-1}$  from Pass  $n - 1$  for reading.
2. **Merge** records from  $r_1 \dots r_{B-1}$ , reading input page-by-page.
3. **Write** new  $B \cdot (B - 1)^{n-1}$ -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:  $2 \cdot N \cdot (1 + \lceil \log_{B-1} \lceil N/B \rceil \rceil)$  .



**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

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

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)?

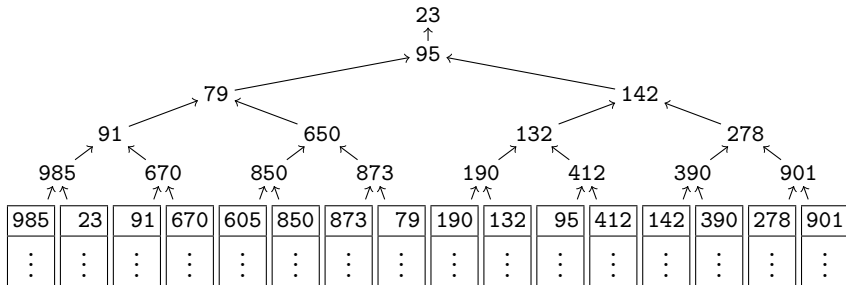
1000 buffer pages, 8 KB each; 10 ms total disk latency

- ▶ 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):

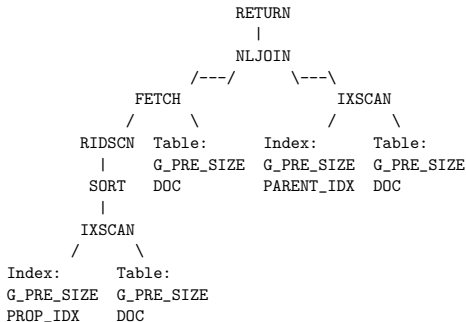


- ▶ This cuts the number of comparisons to  $\log_2 (B - 1)$ .

# 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



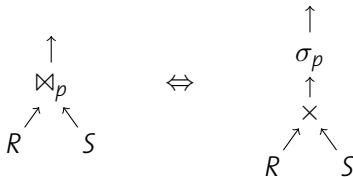
Actual DB2 execution plan.

- ▶ 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 **join operators** next.

# The Join Operator $\bowtie$

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



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  $\sigma$ - $\times$  combination:

```
1 Function: nljoin ( $R, S, p$ )  
2 foreach record  $r \in R$  do  
3   foreach record  $s \in S$  do  
4     if  $\langle r, s \rangle$  satisfies  $p$  then  
5       append  $\langle r, s \rangle$  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

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

## Nested Loops Join: I/O Behavior

The **good news** about `n1join()` 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  $1000 + (5 \cdot 10^7)$  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.

## Block Nested Loops Join

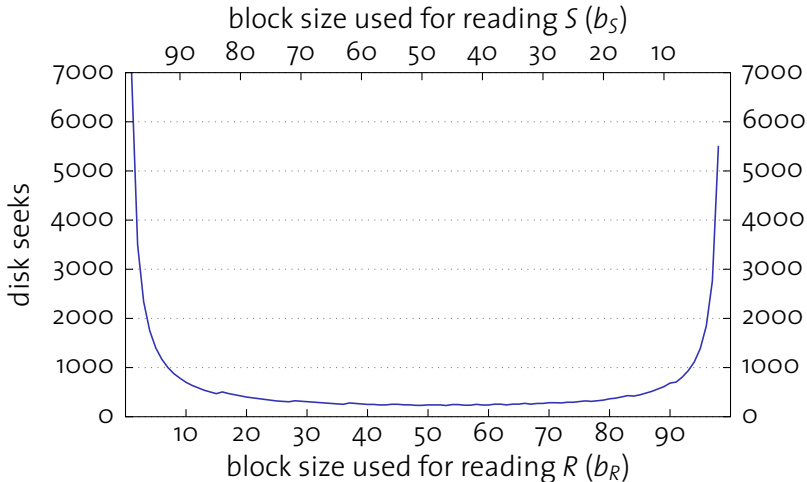
Again we can save random access cost by reading  $R$  and  $S$  in **blocks** of, say,  $b_R$  and  $b_S$  pages.

```
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  $\langle r, s \rangle$  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  $r \in R$  **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 ( $\nearrow$  slide 73).
  - ▶ Such predicates are also called **sargable** (SARG: search argument  $\nearrow$  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<sup>+</sup>-tree index**:

- ▶ A **single** index access requires the inspection of  $h$  pages.<sup>9</sup>
- ▶ 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.

---

<sup>9</sup> $h$ : B<sup>+</sup>-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		C	D
"foo"	1	$\bowtie$ $B=C$	1	false
"foo"	2		2	true
"bar"	2		2	false
"baz"	2		3	true
"baf"	4			

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

```
1 Function: merge_join ( $R, S, \alpha = \beta$ ) //  $\alpha, \beta$ : join columns in  $R, S$ 
2  $r \leftarrow$  position of first tuple in  $R$ ; //  $r, s, s'$ : cursors over  $R, S, S$ 
3  $s \leftarrow$  position of first tuple in  $S$ ;
4 while  $r \neq \text{eof}$  and  $s \neq \text{eof}$  do // eof: end of file marker
5     while  $r.\alpha < s.\beta$  do
6          $\lfloor$  advance  $r$ ;
7     while  $r.\alpha > s.\beta$  do
8          $\lfloor$  advance  $s$ ;
9      $s' \leftarrow s$ ; // Remember current position in  $S$ 
10    while  $r.\alpha = s'.\beta$  do // All  $R$ -tuples with same  $\alpha$  value
11         $s \leftarrow s'$ ; // Rewind  $s$  to  $s'$ 
12        while  $r.\alpha = s.\beta$  do // All  $S$ -tuples with same  $\beta$  value
13             $\lfloor$  append  $\langle r, s \rangle$  to result;
14             $\lfloor$  advance  $s$ ;
15         $\lfloor$  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.

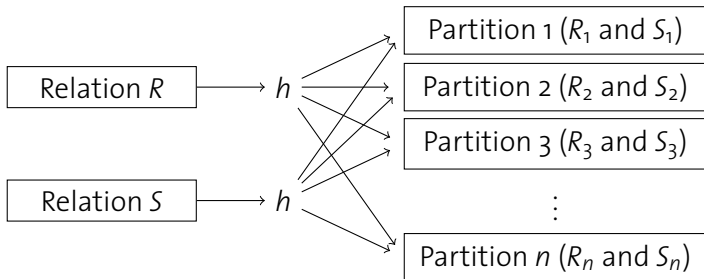


### What is the worst-case behavior of merge join?

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  $R_1, \dots, R_n$  and  $S_1, \dots, S_n$  using the **same** hash function (applied to the join attributes).



- ▶ Observe that  $R_i \bowtie S_j = \emptyset$  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 138).



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



**Why?**

# Hash Join Algorithm

- 1 **Function:** `hash_join` ( $R, S, \alpha = \beta$ )
- 2 **foreach** record  $r \in R$  **do**
- 3     └ append  $r$  to partition  $R_{h(r.\alpha)}$
- 4 **foreach** record  $s \in S$  **do**
- 5     └ append  $s$  to partition  $S_{h(s.\beta)}$
- 6 **foreach** partition  $i \in 1, \dots, n$  **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 $\approx$ ?

- ▶ 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 ( $B \gtrsim \sqrt{N}$ ), hash join and sort-merge join **both** require  $3(N_R + N_S)$  I/Os.<sup>10</sup>

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

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

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

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

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

---

<sup>10</sup>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.
  - Could use a **hash join-like** algorithm or **sorting** to implement duplicate elimination or grouping.
- ▶ See **exercises** for further details.

# Other Database Operators

## Projection $\pi$

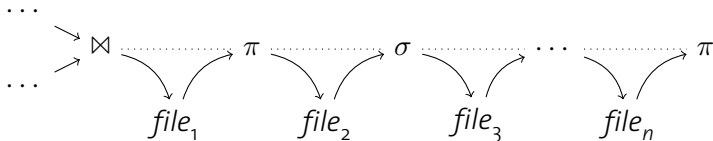
- ▶ Text book-style processing of  $\pi$  implies
  - (a) discarding unwanted fields and
  - (b) eliminating duplicates.
- ▶ Implementing (a) amounts to a straightforward **file scan**. We have mentioned implementations for (b) a moment ago.
- ▶ Typically, systems try to **avoid** (b) whenever possible.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

## Selection $\sigma$

- ▶ A straightforward approach to  $\sigma$  is a **file scan**.
- ▶ Alternatively, we can exploit **sortedness** or take advantage of an available **index**.

# 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 Extensible and Parallel Query Evaluation System. *Trans. Knowl. Data Eng.* vol. 6, no. 1, February 1994.

## Example: Selection ( $\sigma$ )

- ▶ Input operator  $R$ , predicate  $p$ .

```
1 Function: open ()
```

```
2  $R$ .open ();
```

---

```
1 Function: close ()
```

```
2  $R$ .close ();
```

---

```
1 Function: next ()
```

```
2 while (( $r \leftarrow R$ .next ())  $\neq$  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  $r \leftarrow R.next () ;$

1 **Function:** close ()

2  $R.close () ;$

3  $S.close () ;$

---

1 **Function:** next ()

2 **while** ( $r \neq eof$ ) **do**

3     **while** ( $(s \leftarrow S.next ()) \neq eof$ ) **do**

4         **if**  $p(r,s)$  **then**

5             **return**  $\langle r,s \rangle ;$

6      $S.close () ;$

7      $S.open () ;$

8      $s \leftarrow S.next () ;$

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?

- ▶ (external) sorting (this is also true for Unix `sort`)
  - ▶ 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 (“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.