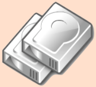
Chapter 8

Evaluation of Relational Operators

Implementing the Relational Algebra

Architecture and Implementation of Database Systems
Summer 2013

Torsten Grust
Wilhelm-Schickard-Institut für Informatik
Universität Tübingen



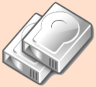
Relational Query Engines

- In many ways, a DBMS's **query engine** compares to virtual machines (*e.g.*, the Java VM):

Relational Query Engine	Virtual Machine (VM)
Operators of the relational algebra	Primitive VM instructions
Operates over streams of rows	Acts on object representations
Operator network (tree/DAG)	Sequential program (with branches, loops)
Several **equivalent variants** of an operator	Compact instruction set

✎ Equivalent operator variants

Instead of a single $\bowtie$ operator, a typical DBMS query engine features equivalent variants $\bowtie', \bowtie'', \dots$.
What would **equivalent** mean in the context of the relational model?



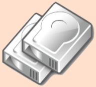
Operator Variants

- Specific operator variants may be tailored to exploit **physical properties** of its input or the current system state:
  1. The **presence or absence of indexes** on the input file(s),
  2. the **sortedness** of the input file(s),
  3. the **size** of the input file(s),
  4. the **available space in the buffer pool**,
  5. the **buffer replacement policy**,
  6. . . .

Physical operators

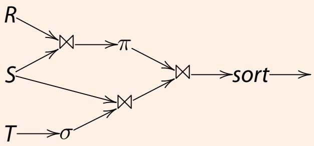
The variants ($\bowtie'$, $\bowtie''$) are thus referred to **physical operators**. They implement the **logical operators** of the relational algebra.

- The **query optimizer** is in charge to perform optimal (or, reasonable) **operator selection** (much like the instruction selection phase in a programming language compiler).



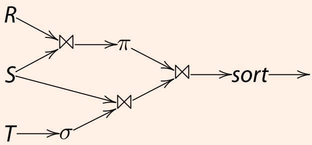
Operator Selection

Initial, logical operator network (“plan”)

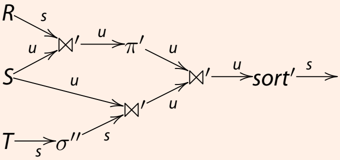


Operator Selection

Initial, logical operator network ("plan")

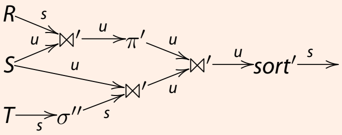


Physical plan with (un)sortedness annotations (*u*/*s*)



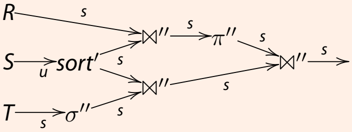
Plan Rewriting

Physical plan with (un)sortedness annotations (*u*/*s*)



- **Rewrite** the plan to exploit that the $\oplus''$ variant of operator $\oplus$ can benefit from/preserve sortedness of its input(s):

Rewritten physical plan (preserve equivalence!)



Selection (σ)—No Index, Unsorted Data

- **Selection** (σ_p) reads an input file R_{in} of records and writes those records satisfying predicate p into the output file:

Selection

```
Function: $\sigma(p, R_{in}, R_{out})$
$out \leftarrow$ createFile($R_{out}$) ;
$in \leftarrow$ openScan($R_{in}$) ;
while ($r \leftarrow$ nextRecord($in$)) $\neq \langle$EOF$\rangle$ do
    if $p(r)$ then
        appendRecord($out, r$) ;
closeFile($out$) ;
```

Selection (σ)—No Index, Unsorted Data

Remarks:

- Reading the special "record" $\langle$`EOF`$\rangle$ from a file via `nextRecord()` indicates that all its record have been retrieved (scanned) already.
- This simple procedure does **not require r_{in} to come with any special physical properties** (the procedure is exclusively defined in terms of heap files).
- In particular, **predicate** p may be **arbitrary**.

Selection (σ)—No Index, Unsorted Data

- We can summarize the characteristics of this implementation of the selection operator as follows:

Selection (σ)—no index, unsorted data

$\sigma_p\ (R)$

input access[1]	file scan (`openScan`) of R
prerequisites	none (p arbitrary, R may be a heap file)
I/O cost	$\underbrace{N_R}_{\text{input cost}} + \underbrace{sel(p) \cdot N_R}_{\text{output cost}}$

- N_R denotes the **number of pages** in file R, $|R|$ denotes the **number of records**
  (if p_R records fit on one page, we have $N_R = \left\lceil {}^{|R|}/_{p_R} \right\rceil$)

---

[1] Also known as **access path** in the literature and text books.

Aside: Selectivity

- $sel(p)$, the **selectivity of predicate** p, is the fraction of records satisfying predicate p:

$$0 \quad \leqslant \quad sel(p) = \frac{|\sigma_p\,(R)|}{|R|} \quad \leqslant \quad 1$$

Selectivity examples

What can you say about the following selectivities?

① $sel(true)$
② $sel(false)$
③ $sel(A = 0)$

DB2 Estimated selectivities

IBM DB2 reports (estimated) selecitvities in the operators details of, *e.g.*, its `IXSCAN` operator.

Selection (σ)—Matching Predicates with an Index

- A selection on input file R can be sped up considerably if an index has been defined and that **index matches predicate** p.
- The matching process depends on p itself as well as on the index type. If there is no immediate match but p is **compound**, a sub-expression of p may still find a **partial match**. Residual predicate evaluation work may then remain.

Selection (σ)—Matching Predicates with an Index

- A selection on input file *R* can be sped up considerably if an index has been defined and that **index matches predicate** *p*.
- The matching process depends on *p* itself as well as on the index type. If there is no immediate match but *p* is **compound**, a sub-expression of *p* may still find a **partial match**. Residual predicate evaluation work may then remain.

✎ When does a predicate match a sort key?

Assume *R* is tree-indexed on attribute *A* in ascending order. Which of the selections below can benefit from the index on *R*?

① $\sigma_{A=42}\,(R)$

② $\sigma_{A<42}\,(R)$

③ $\sigma_{A>42 \texttt{ AND } A<100}\,(R)$

④ $\sigma_{A>42 \texttt{ OR } A>100}\,(R)$

⑤ $\sigma_{A>42 \texttt{ AND } A<32}\,(R)$

⑥ $\sigma_{A>42 \texttt{ AND } B=10}\,(R)$

⑦ $\sigma_{A>42 \texttt{ OR } B=10}\,(R)$

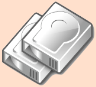
Selection (σ)—B$^+$-tree Index

- A **B$^+$-tree index** on R whose key **matches** the selection predicate p is clearly the superior method to evaluate $\sigma_p\ (R)$:
  - **Descend the B$^+$-tree** to retrieve the first index entry to satisfy p. If the index is **clustered**, access that record on its page in R and continue to scan inside R.
  - If the index is **unclustered** and $sel(p)$ indicates a large number of qualifying records, it pays off to
    ① read the matching index entries $k* = \langle k, rid \rangle$ in the sequence set,
    ② sort those entries on their *rid* field,
    ③ and then access the pages of R in sorted *rid* order.

    Note that lack of clustering is a minor issue if $sel(p)$ is close to 0.

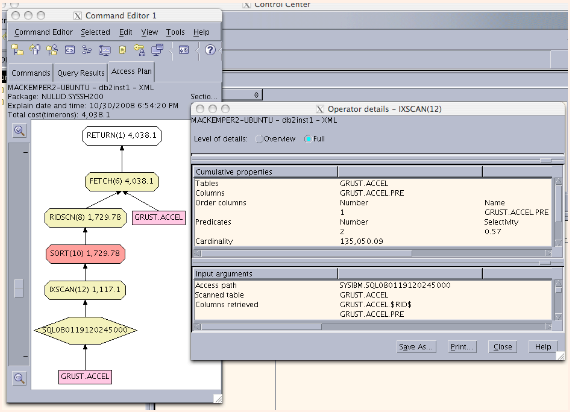
DB2 Accessing unclustered B$^+$-trees

IBM DB2 uses physical operator quadruple `IXSCAN/SORT/RIDSCN/FETCH` to implement the above strategy.



Selection (σ)—B$^+$-tree Index

The `IXSCAN/SORT/RIDSCN/FETCH` quadruple



MACKEMPER2-UBUNTU – db2inst1 – XML
Package: NULLID.SYSSH200
Explain date and time: 10/30/2008 6:54:20 PM
Total cost(timerons): 4,038.1

MACKEMPER2-UBUNTU – db2inst1 – XML

Level of details: ○ Overview ● Full

Cumulative properties		
Tables	GRUST.ACCEL	
Columns	GRUST.ACCEL.PRE	
Order columns	Number	Name
	1	GRUST.ACCEL.PRE
Predicates	Number	Selectivity
	2	0.57
Cardinality	135,050.09	

Input arguments		
Access path	SYSIBM.SQL080119120245000	
Scanned table	GRUST.ACCEL	
Columns retrieved	GRUST.ACCEL.RID	
	GRUST.ACCEL.PRE	

- Note: Selectivity of predicate estimated as 57 % (table `accel` has 235,501 rows).

Selection (σ)—B$^+$-tree Index

Selection (σ)—clustered B$^+$-tree index

$\sigma_p\ (R)$

input access	access of B$^+$-tree on R, then sequence set scan
prerequisites	clustered B$^+$-tree on R with key k, p matches key k
I/O cost	$\underbrace{\approx 3}_{\text{B}^+\text{-tree acc.}} + \underbrace{sel(p) \cdot N_R}_{\text{sorted scan}} + \underbrace{sel(p) \cdot N_R}_{\text{output cost}}$

Selection (σ)—Hash Index, Equality Predicate

- A selection predicate p **matches an hash index** only if p contains a term of the form `A = ` c (c constant, assuming the hash index has been built over column `A`).
- We are directly led to the bucket of qualifying records and pay I/O cost only for the access of this bucket[2]. Note that $sel(p)$ is likely to be close to 0 for equality predicates.

Selection (σ)—hash index, equality predicate

$\sigma_p\ (R)$

input access	hash table on R
prerequisites	r_{in} hashed on key `A`, p has term `A = ` c
I/O cost	$\underbrace{1}_{\text{hash access}} + \underbrace{sel(p) \cdot N_R}_{\text{output cost}}$

---

[2] Remember that this may include access cost for the pages of an overflow chain hanging off the primary bucket page.

Selection (σ)—Conjunctive Predicates

- Indeed, selection operations with simple predicates like $\sigma_{\mathtt{A}\,\theta\,c}\,(R)$ are a special case only.
- We somehow need to deal with **complex predicates**, built from **simple comparisons** and the **Boolean connectives** `AND` and `OR`.
- Matching a selection predicate with an index can be extended to cover the case where predicate p has a **conjunctive form**:

$$\underbrace{\mathtt{A}_1\,\theta_1\,c_1}_{\text{conjunct}} \quad \mathtt{AND} \quad \mathtt{A}_2\,\theta_2\,c_2 \quad \mathtt{AND} \quad \cdots \quad \mathtt{AND} \quad \mathtt{A}_n\,\theta_n\,c_n \;.$$

  - Here, each **conjunct** is a simple comparison ($\theta_i \in \{=, <, >, <=, >=\}$).

Selection (σ)—Conjunctive Predicates

- Indeed, selection operations with simple predicates like $\sigma_{\mathtt{A}\,\theta\,c}\,(R)$ are a special case only.
- We somehow need to deal with **complex predicates**, built from **simple comparisons** and the **Boolean connectives** `AND` and `OR`.
- Matching a selection predicate with an index can be extended to cover the case where predicate p has a **conjunctive form**:

$$\underbrace{\mathtt{A}_1\,\theta_1\,c_1}_{\text{conjunct}} \quad \mathtt{AND} \quad \mathtt{A}_2\,\theta_2\,c_2 \quad \mathtt{AND} \quad \cdots \quad \mathtt{AND} \quad \mathtt{A}_n\,\theta_n\,c_n \quad .$$

  - Here, each **conjunct** is a simple comparison ($\theta_i \in \{=, <, >, <=, >=\}$).

- An index with a multi-attribute key may match the *entire* complex predicate.

Selection (σ)—Conjunctive Predicates

✎ Matching a multi-attribute hash index

Consider a hash index for the multi-attribute key $k = (\mathtt{A}, \mathtt{B}, \mathtt{C})$, *i.e.*, all three attributes are input to the hash function.

Which conjunctive predicates *p* would **match** this type of index?

Selection (σ)—Conjunctive Predicates

✎ Matching a multi-attribute hash index

Consider a hash index for the multi-attribute key $k = (\mathtt{A}, \mathtt{B}, \mathtt{C})$, *i.e.*, all three attributes are input to the hash function.

Which conjunctive predicates p would **match** this type of index?

Conjunctive predicate match rule for hash indexes

A **conjunctive predicate** p **matches a (multi-attribute) hash index** with key $k = (\mathtt{A}_1, \mathtt{A}_2, \ldots, \mathtt{A}_n)$, if p **covers the key**, *i.e.*,

$$p \quad \equiv \quad \mathtt{A}_1 \;\mathtt{=}\; c_1 \;\;\mathtt{AND}\;\; \mathtt{A}_2 \;\mathtt{=}\; c_2 \;\;\mathtt{AND}\;\; \cdots \;\;\mathtt{AND}\;\; \mathtt{A}_n \;\mathtt{=}\; c_n \;\;\mathtt{AND}\;\; \phi \;\; .$$

The residual conjunct ϕ is not supported by the index itself and has to be **evaluated after index retrieval**.

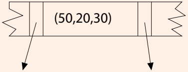
Selection (σ)—Conjunctive Predicates

Matching a multi-attribute B$^+$-tree index

Consider a B$^+$-tree index for the multi-attribute key $k = (\mathtt{A}, \mathtt{B}, \mathtt{C})$, *i.e.*, the B$^+$-tree nodes are searched/inserted in lexicographic order w.r.t. these three attributes:

$$k_1 < k_2 \equiv \begin{array}{l} \mathtt{A}_1 < \mathtt{A}_2 \lor \\ (\mathtt{A}_1 = \mathtt{A}_2 \land \mathtt{B}_1 < \mathtt{B}_2) \lor \\ (\mathtt{A}_1 = \mathtt{A}_2 \land \mathtt{B}_1 = \mathtt{B}_2 \land \mathtt{C}_1 < \mathtt{C}_2) \end{array}$$

Excerpt of an inner B$^+$-tree node (separator):



Which conjunctive predicates p would **match** this type of index?

Selection (σ)—Conjunctive Predicates

Conjunctive predicate match rule for B$^+$-tree indexes

A **conjunctive predicate** p **matches a (multi-attribute) B$^+$-tree index** with key $k = (\mathtt{A}_1, \mathtt{A}_2, \dots, \mathtt{A}_n)$, if p **is a prefix of the key**, *i.e.*,

$$\begin{aligned}
p &\equiv \mathtt{A}_1\ \theta_1\ c_1\ \mathtt{AND}\ \phi \\
p &\equiv \mathtt{A}_1\ \theta_1\ c_1\ \mathtt{AND}\ \mathtt{A}_2\ \theta_2\ c_2\ \mathtt{AND}\ \phi \\
&\ \vdots \\
p &\equiv \mathtt{A}_1\ \theta_1\ c_1\ \mathtt{AND}\ \mathtt{A}_2\ \theta_2\ c_2\ \mathtt{AND} \cdots \mathtt{AND}\ \mathtt{A}_n\ \theta_n\ c_n\ \mathtt{AND}\ \phi
\end{aligned}$$

- **Note**: Whenever a multi-attribute hash index matches a predicate, so does a B$^+$-tree over the same key.

Selection (σ)—Conjunctive Predicates

- If the system finds that a conjunctive predicate does not match a single index, its (smaller) **conjuncts may nevertheless match distinct indexes**.

Example (Partial predicate match)

The conjunctive predicate in $\sigma_{p\ \texttt{AND}\ q}(R)$ does not match an index, but both conjuncts p, q do.

A typical optimizer might thus decide to transform the original query

$$R \longrightarrow \sigma_{p\ \texttt{AND}\ q} \longrightarrow$$

into

$$\begin{tikzcd}
& \sigma'_p \arrow[dr] & & \\
R \arrow[ur] \arrow[dr] & & \cap^{rid} \arrow[r] & {} \\
& \sigma'_q \arrow[ur] & &
\end{tikzcd}$$

Here, $\cap^{rid}$ denotes a **set intersection operator defined by** ***rid*** **equality** (IBM DB2: `IXAND`).

Selection (σ)—Conjunctive Predicates

Selectivity of conjunctive predicates

What can you say about the selectivity of the conjunctive predicate p `AND` q?

$$sel(p \texttt{ AND } q) \quad =$$

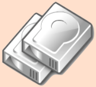
Selection (σ)—Conjunctive Predicates

✎ Selectivity of conjunctive predicates

What can you say about the selectivity of the conjunctive predicate p `AND` q?

$$sel(p\ \texttt{AND}\ q) \quad =$$

Now assume $p \equiv \texttt{AGE <= 16}$ and $q \equiv \texttt{SALARY > 5000}$. Reconsider your proposal above.



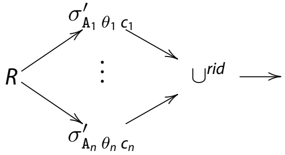
Selection (σ)—Disjunctive Predicates

- Choosing a reasonable execution plan for **disjunctive selection** of the general form

$$\mathtt{A}_1\,\theta_1\,c_1 \quad \mathtt{OR} \quad \mathtt{A}_2\,\theta_2\,c_2 \quad \mathtt{OR} \quad \cdots \quad \mathtt{OR} \quad \mathtt{A}_n\,\theta_n\,c_n$$

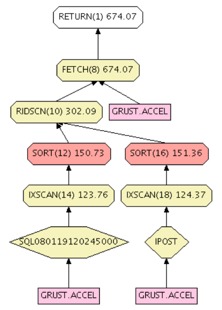
  is much harder:

  - We are forced to **fall back to a naive file scan based evaluation** as soon only a **single term does not match** an index.
  - If **all terms are matched** by indexes, we can exploit a *rid*-**based set union** $\cup^{rid}$ to improve the plan:



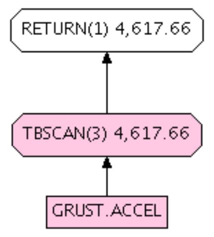
Selection (σ)—Disjunctive Predicates

Selective disjunctive predicate



Note: Multi-input `RIDSCN` operator.

Non-selective disjunctive predicate



Note: Presence of indexes ignored.

Selection (σ)—Disjunctive Predicates

✎ Selectivity of disjunctive predicates

What can you say about the selectivity of the disjunctive predicate p `OR` q?

$$sel(p \; \texttt{OR} \; q) \quad =$$

Projection (π)

- **Projection** (π_ℓ) modifies each record in its input file and cuts off any field not listed in the attribute list ℓ:

Relational projection

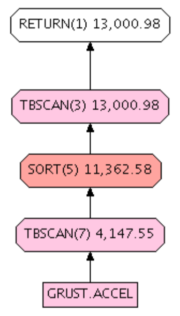
$$\pi_{\texttt{A,B}}\left(\begin{array}{ccc} \texttt{A} & \texttt{B} & \texttt{C} \\ 1 & \texttt{'foo'} & 3 \\ 1 & \texttt{'bar'} & 2 \\ 1 & \texttt{'foo'} & 2 \\ 1 & \texttt{'bar'} & 0 \\ 1 & \texttt{'foo'} & 0 \end{array}\right) \underset{\textcircled{1}}{=} \begin{array}{cc} \texttt{A} & \texttt{B} \\ 1 & \texttt{'foo'} \\ 1 & \texttt{'bar'} \\ 1 & \texttt{'foo'} \\ 1 & \texttt{'bar'} \\ 1 & \texttt{'foo'} \end{array} \underset{\textcircled{2}}{=} \begin{array}{cc} \texttt{A} & \texttt{B} \\ 1 & \texttt{'foo'} \\ 1 & \texttt{'bar'} \end{array}$$

- In general, the size of the resulting file will only be a fraction of the original input file:
  1. any unwanted fields (here: `C`) have been thrown away, and
  2. optionally **duplicates removed** (SQL: `DISTINCT`).

Projection (π)—Duplicate Elimination, Sorting

- **Sorting** is one obvious preparatory step to facilitate duplicate elimination: records with all fields equal will end up adjacent to each other.

DB2 Implementing `DISTINCT`



- One benefit of sort-based projection is that operator π_ℓ will write a sorted output file, *i.e.*,

$$R \xrightarrow{?} \pi_\ell^{sort} \xrightarrow{s}$$

Sort ordering?

What would be the correct ordering θ to apply in the case of duplicate elimination?

Projection (π)—Duplicate Elimination, Hashing

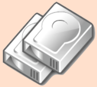
- If the DBMS has a fairly large number of buffer pages (B, say) to spare for the $\pi_\ell\ (R)$ operation, a **hash-based** projection may be an efficient alternative to sorting:

Hash-based projection π_ℓ: partitioning phase

① Allocate all B buffer pages. One page will be the **input buffer**, the remaining $B - 1$ pages will be used as **hash buckets**.

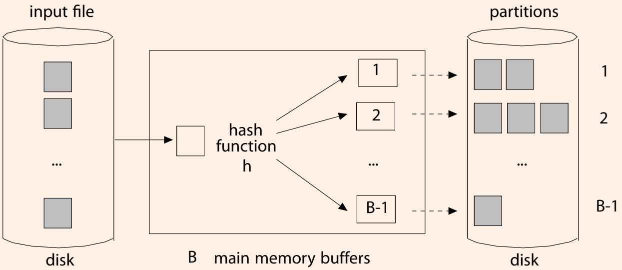
② Read the file R page-by-page, for each record r: cut off fields not listed in ℓ.

③ For each such record, apply hash function $h_1(r) = h(r) \mod (B-1)$—which depends on **all remaining fields of** r—and store r in hash bucket $h_1(r)$. (Write the bucket to disk if full.)



Projection (π)—Hashing

Hash-based projection π_ℓ: partitioning phase



- After partitioning, duplicate elimination becomes an **intra-partition** problem only: two identical records have been mapped to the same partition:

$$h_1(r) = h_1(r') \quad \Leftarrow \quad r = r' \ .$$

Projection (π)—Hashing

Hash-based projection π_ℓ: duplicate elimination phase

① For each partition, read each partition page-by-page (possibly in parallel).

② To each record, apply hash function $h_2 \overset{!}{\neq} h_1$ to all record fields.

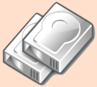
③ Only if two records **collide** w.r.t. h_2, check if $r = r'$. If so, discard r'.

④ After the entire partition has been read in, append all hash buckets to the result file (which will be free of duplicates).

✎ Huge partitions?

Note: Works efficiently only if duplicate elimination phase can be **performed in the buffer** (main memory).

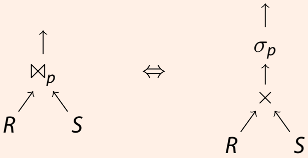
What to do if partition size exceeds buffer size?



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.

Join vs. Cartesian product



One way to implement $\bowtie_p$ is to follow this equivalence:

① Enumerate and concatenate all records in the cross product of r_1 and r_2.

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:

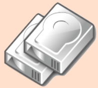
Nested loops join

```
Function: nljoin ($R$, $S$, $p$)
/* outer relation $R$                                  */
foreach record $r \in R$ do
    /* inner relation $S$                              */
    foreach record $s \in S$ do
        /* $\langle r, s \rangle$ denotes record concatenation   */
        if $\langle r, s \rangle$ satisfies $p$ then
            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** then is

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



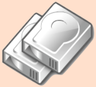
Nested Loops Join: I/O Behavior

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.



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.

Block nested loops join

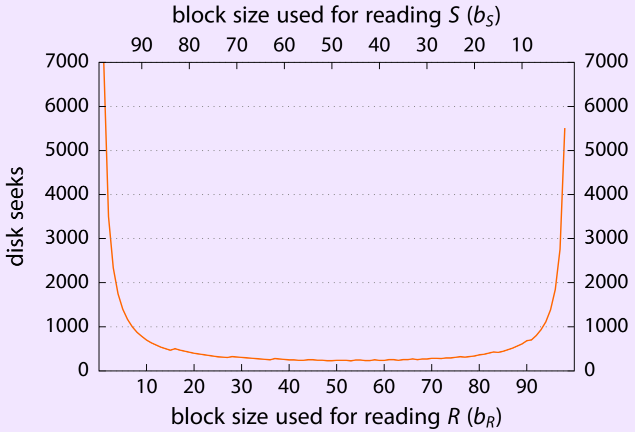
```
Function: block_nljoin ($R$, $S$, $p$)
foreach $b_R$-sized block in $R$ do
    foreach $b_S$-sized block in $S$ do
        /* performed in the buffer                */
        find matches in current $R$- and $S$-blocks and append them
        to the result ;
```

- 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

Example: Buffer pool with $B = 100$ frames, $N_R = 1000$, $N_S = 500$:

Example (Choosing b_r and b_S)



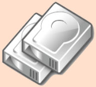
In-Memory Join Performance

- 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.

Block nested loops join: build hash table from outer row block

```
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* ↔ *inner* if necessary):

Index nested loops join

```
Function: index_nljoin ($R, S, p$)
foreach record $r \in R$ do
    scan $S$-index using (key value in) $r$ and concatenate $r$ with all matching tuples $s$ ;
    append $\langle r, s \rangle$ to result ;
```

- The index must **match** the join condition *p*.
  - Hash indices, *e.g.*, only support equality predicates.
  - Remember the discussion about composite keys in B^+-trees.
- Such predicates are also called **sargable** (*sarg*: search argument ↗ Selinger *et al.*, *SIGMOD 1979*)

Index Nested Loop Join: 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 (locality in the index).

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.[3]
- 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 will not help here (no locality in accesses to hash table).
- A typical value for N_{idx} is 1.2 I/Os ($>$ 1 due to overflow pages).

Overall, the use of an index (over, *e.g.*, a block nested loops join) pays off if the join is **selective** (picks out only few tuples from a big table).

---

[3] 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. Both tables are **read once, in parallel**.
- In contrast to sorting, however, we need to be careful whenever a tuple has **multiple** matches in the other relation:

Multiple matches per tuple (disrupts sequential access)

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

$\underset{B=C}{\bowtie}$

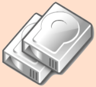
C	D
1	false
2	true
2	false
3	true

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

Merge Join: Algorithm

Merge join algorithm

```
Function: merge_join($R, S, \alpha = \beta$)  // $\alpha, \beta$: join col.s in $R, S$
$r \leftarrow$ position of first tuple in $R$;  // $r$, $s$, $s'$: cursors over $R$, $S$, $S$
$s \leftarrow$ position of first tuple in $S$;
while $r \neq \langle \text{EOF} \rangle$ and $s \neq \langle \text{EOF} \rangle$ do  // $\langle \text{EOF} \rangle$: end of file marker
    while $r.\alpha < s.\beta$ do
        advance $r$;
    while $r.\alpha > s.\beta$ do
        advance $s$;
    $s' \leftarrow s$;                          // Remember current position in $S$
    while $r.\alpha = s'.\beta$ do              // All $R$-tuples with same $\alpha$ value
        $s \leftarrow s'$;                                      // Rewind $s$ to $s'$
        while $r.\alpha = s.\beta$ do           // All $S$-tuples with same $\beta$ value
            append $\langle r, s \rangle$ to result;
            advance $s$;
        advance $r$;
```



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.

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

Merge Join: I/O Behavior

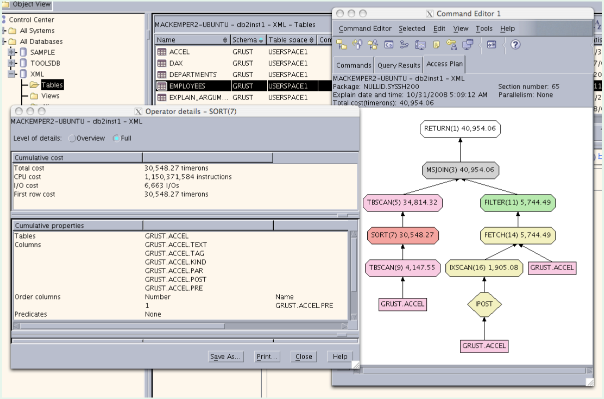
- If both inputs are already sorted **and** there are no exceptionally long sequences of identical key values, the I/O cost of a merge join is $N_R + N_S$ (which is optimal).
- By using **blocked I/O**, these I/O operations can be done almost entirely as **sequential** reads.
- Sometimes, it pays off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

✎ **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.

Merge Join: IBM DB2 Plan

DB2 Merge join (left input: sort, right input: sorted index scan)



MACKEMPER2-UBUNTU - db2inst1 - XML - Tables

Name	Schema	Table space
ACCEL	GRUST	USERSPACE1
DAX	GRUST	USERSPACE1
DEPARTMENTS	GRUST	USERSPACE1
EMPLOYEES	GRUST	USERSPACE1
EXPLAIN_ARGUM...	GRUST	USERSPACE1

MACKEMPER2-UBUNTU - db2inst1 - XML
Package: NULLID.SYSSH200 Section number: 65
Explain date and time: 10/31/2008 5:09:12 AM Parallelism: None
Total cost(timerons): 40,954.06

Operator details - SORT(7)

MACKEMPER2-UBUNTU - db2inst1 - XML

Level of details: ○ Overview ● Full

Cumulative cost	
Total cost	30,548.27 timerons
CPU cost	1,150,371,584 instructions
I/O cost	6,663 I/Os
First row cost	30,548.27 timerons

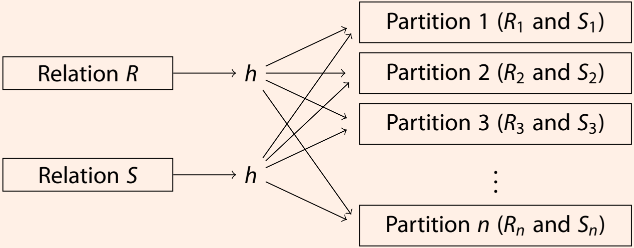
Cumulative properties		
Tables	GRUST.ACCEL	
Columns	GRUST.ACCEL.TEXT	
	GRUST.ACCEL.TAG	
	GRUST.ACCEL.KIND	
	GRUST.ACCEL.PAR	
	GRUST.ACCEL.POST	
	GRUST.ACCEL.PRE	
Order columns	Number	Name
	1	GRUST.ACCEL.PRE
Predicates	None	

- **Note**: The `FILTER(11)` implements the join predicate of the `MSJOIN(3)`.

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).

Hash partitioning for both inputs



Relation R → h

Relation S → h

Partition 1 (R_1 and S_1)

Partition 2 (R_2 and S_2)

Partition 3 (R_3 and S_3)

$\vdots$

Partition n (R_n and S_n)

- 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 (again: works for equality predicates only).
- 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 34).

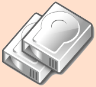
✎ Intra-partition join via hash table

Use a **different** hash function $h' \neq h$ for the intra-partition join. **Why?**

Hash Join Algorithm

Hash join

```
Function: hash_join ($R, S, \alpha = \beta$)
/* Partitioning phase                                   */
foreach record $r \in R$ do
    append $r$ to partition $R_{h(r.\alpha)}$
foreach record $s \in S$ do
    append $s$ to partition $S_{h(s.\beta)}$
/* Intra-partition join phase                           */
foreach partition $i \in 1, \dots, n$ do
    build hash table $H$ for $R_i$, using hash function $h'$;
    foreach block $b \in S_i$ do
        foreach record $s \in b$ do
            probe $H$ via $h'(s.\beta)$ and append matching tuples to
            result ;
```



Hash Join—Buffer Requirements

- We 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 (recursive partitioning).

Hash Join—Buffer Requirements

- We 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 $\approx$?

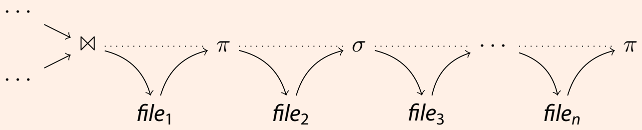
- We can write out at most $B - 1$ partitions in one pass; each of them should be at most $B - 1$ pages in size.
- Hashing does not guarantee an even distribution. Since the actual size of each partition varies, R must actually be smaller than $(B - 1)^2$.

- Larger input tables require **multiple passes** for partitioning (recursive partitioning).

Orchestrating Operator Evaluation

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

File-based operator input and output



- Obviously, using **secondary storage as the communication channel** causes **a lot of disk 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**.

Unix: Temporary Result Files

- Architecting the query processor in this fashion bears much resemblance with using the Unix shell like this:

File-based Unix command sequencing

```
# report "large" XML files below current working dir
$ find . -size +500k      > tmp1
$ xargs file              < tmp1 > tmp2
$ grep -i XML             < tmp2 > tmp3
$ cut -d: -f1             < tmp3
⟨output generated here⟩

# remove temporary files
$ rm -f tmp[0-9]*
```

Pipelined Evaluation

- Alternatively, each operator could pass its result **directly** on to the next operator (without persisting it to disk first).

⇒ Do not 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 provides a similar mechanism to communicate between processes ("operators"):

Pipeline-based Unix command sequencing

```
$ find . -size +500k | xargs file | grep -i XML | cut -d: -f1
⟨output generated here⟩
```

Execution of this pipe is driven by the **rightmost** operator—all operators act in **parallel**:

- 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

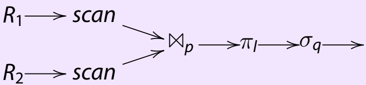
`open ()`	**Initialize** the operator's internal state.
`next ()`	Produce and return the **next result tuple** or ⟨`EOF`⟩.
`close ()`	**Clean up** all allocated resources (typically after all tuples have been processed).

- All **state** is kept inside each operator instance:
  - Operators are required to produce a tuple via `next ()`, then **pause**, and later **resume** on a subsequent `next ()` call.

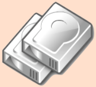
↗ Goetz Graefe. Volcano—An Extensibel and Parallel Query Evaluation System. *Trans. Knowl. Data Eng.* vol. 6, no. 1, February 1994.

Volcano-Style Pipelined Evaluation

Example (Pipelined query plan)



- Given a query plan like the one shown above, query evaluation is driven by the query processor like this (just like in the Unix shell):
  1. The whole plan is initially reset by calling `open ()` on the **root operator**, *i.e.*, σ_q`.open ()`.
  2. The `open ()` call is **forwarded** through the plan by the operators themselves (see σ`.open ()` on slide 53).
  3. Control returns to the query processor.
  4. The root is requested to produce its next result record, *i.e.*, the call σ_q`.next ()` is made.
  5. Operators forward the `next ()` request as needed. **As soon as the next result record is produced, control returns** to the query processor again.



Volcano-Style Pipelined Evaluation

- In a nutshell, the query processor uses the following routine to evaluate a query plan:

Query plan evaluation driver

```
Function: eval ($q$)
$q$.open ();
$r \leftarrow q$.next ();
while $r \neq \langle$EOF$\rangle$ do
    /* deliver record $r$ (print, ship to DB client)   */
    emit ($r$);
    $r \leftarrow q$.next ();
/* resource deallocation now   */
$q$.close ();
```

Volcano-Style Selection (σ_p)

- Input operator (sub-plan root) *R*, predicate *p*:

Volcano-style interface for $\sigma_p(R)$

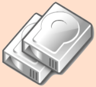
```
Function: open ()
$R$.open () ;
```

```
Function: close ()
$R$.close () ;
```

```
Function: next ()
while (($r \leftarrow R$.next ()) $\neq$ ⟨EOF⟩) do
    if $p(r)$ then
        return $r$ ;
return ⟨EOF⟩;
```

Volcano-Style Nested Loops Join ($\bowtie_p$)

✎ A Volcano-style implementation of nested loops join $R \bowtie_p S$?



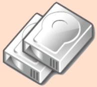
Volcano-Style Nested Loops Join ($\bowtie_p$)

✎ A Volcano-style implementation of nested loops join $R \bowtie_p S$?

```
Function: open ()
$R$.open () ;
$S$.open () ;
$r \leftarrow R$.next () ;
```

```
Function: close ()
$R$.close () ;
$S$.close () ;
```

```
Function: next ()
while ($r \neq \langle$EOF$\rangle$) do
    while (($s \leftarrow S$.next ()) $\neq \langle$EOF$\rangle$) do
        if $p(r, s)$ then
            /* emit concatenated result        */
            return $\langle r, s \rangle$ ;
    /* reset inner join input               */
    $S$.close () ;
    $S$.open () ;
    $r \leftarrow R$.next () ;
return $\langle$EOF$\rangle$;
```



Pipelining (Real DBMS Product)

Volcano-style pipelined selection operator (C code)

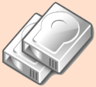
```
/* eFLTR -- apply filter predicate pred to stream
   Filter the in-bound stream, only stream elements that fulfill
   e->pred contribute to the result.  No index support. */

eRC eOp_FLTR(eOp *ip) {
     eObj_FLTR *e = (eObj_FLTR *)eObj(ip);

     while (eIntp(e->in) != eEOS) {
          eIntp(e->pred);
          if (eT_as_bool(eVal(e->pred))) {
               eVal(ip) = eVal(e->in);
               return eOK;
          }
     }
     return eEOS;
}

eRC eOp_FLTR_RST(eOp *ip) {
     eObj_FLTR *e = (eObj_FLTR *)eObj(ip);

     eReset(e->in);
     eReset(e->pred);
     return eOK;
}
```

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 operators do not permit pipelined evaluation?

- 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.



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 operators do not permit pipelined evaluation?**

- (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.