

**Relational Query  
Engines**

Operator Selection

**Selection ( $\sigma$ )**

Selectivity

Conjunctive Predicates

Disjunctive Predicates

**Projection ( $\pi$ )**

**Join ( $\bowtie$ )**

Nested Loops Join

Block Nested Loops Join

Index Nested Loops Join

Sort-Merge Join

Hash Join

**Operator Pipelining**

Volcano Iterator Model

# Chapter 8

## Evaluation of Relational Operators

Implementing the Relational Algebra

*Architecture and Implementation of Database Systems*

Winter 2010/11



# Relational Query Engines

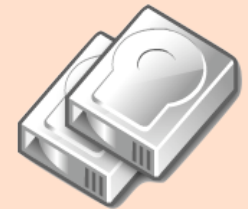
- In many ways, a DBMS's **query engine** compares to virtual machines (e.g., the Java VM):

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 <b>equivalent variants</b> of an operator	Compact instruction set

## Equivalent operator variants

Instead of a single  $\bowtie$  operator, a typical DBMS query engine features equivalent variants  $\bowtie'$ ,  $\bowtie''$ , ....

What would **equivalent** mean in the context of the relational model?



## Relational Query Engines

Operator Selection

### Selection ( $\sigma$ )

Selectivity

Conjunctive Predicates

Disjunctive Predicates

### Projection ( $\pi$ )

### Join ( $\bowtie$ )

Nested Loops Join

Block Nested Loops Join

Index Nested Loops Join

Sort-Merge Join

Hash Join

### Operator Pipelining

Volcano Iterator Model

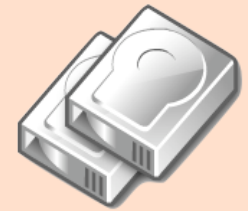
# Operator Variants

- 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).



### Selection ( $\sigma$ )

Selectivity

Conjunctive Predicates

Disjunctive Predicates

### Projection ( $\pi$ )

### Join ( $\bowtie$ )

Nested Loops Join

Block Nested Loops Join

Index Nested Loops Join

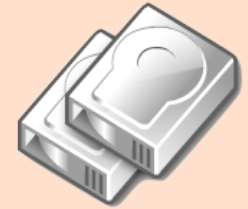
Sort-Merge Join

Hash Join

### Operator Pipelining

Volcano Iterator Model

# Operator Selection



## Relational Query Engines

Operator Selection

### Selection ( $\sigma$ )

Selectivity  
Conjunctive Predicates  
Disjunctive Predicates

### Projection ( $\pi$ )

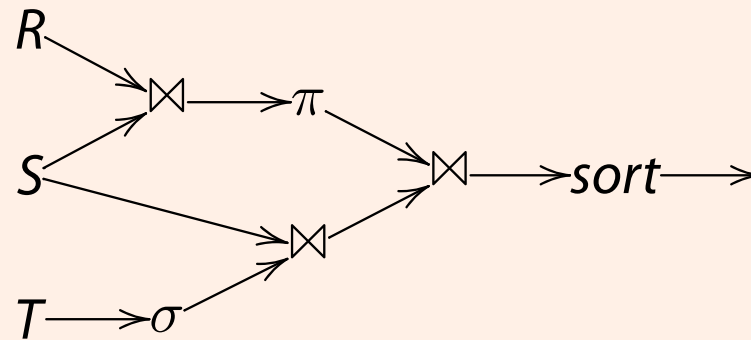
### Join ( $\bowtie$ )

Nested Loops Join  
Block Nested Loops Join  
Index Nested Loops Join  
Sort-Merge Join  
Hash Join

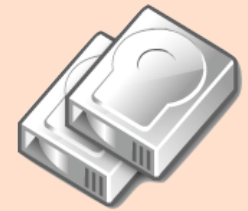
### Operator Pipelining

Volcano Iterator Model

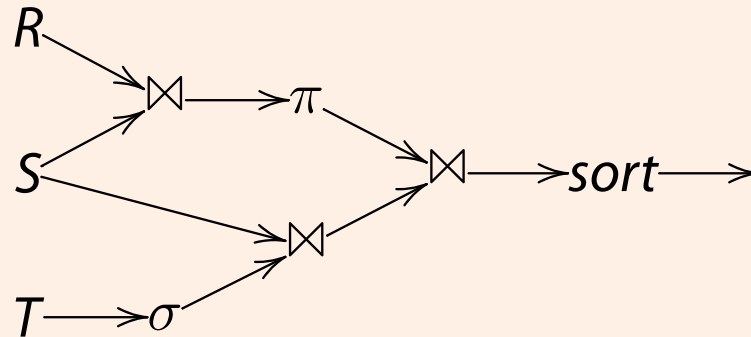
## Initial, logical operator network ("plan")



# Operator Selection



## Initial, logical operator network ("plan")



## Relational Query Engines

### Operator Selection

### Selection ( $\sigma$ )

- Selectivity
- Conjunctive Predicates
- Disjunctive Predicates

### Projection ( $\pi$ )

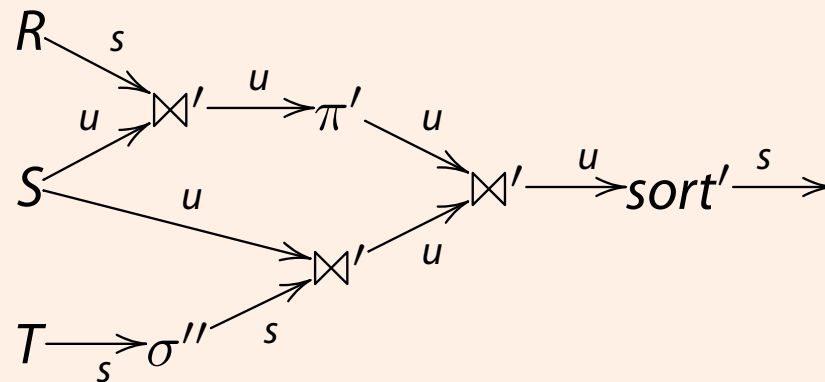
### Join ( $\bowtie$ )

- Nested Loops Join
- Block Nested Loops Join
- Index Nested Loops Join
- Sort-Merge Join
- Hash Join

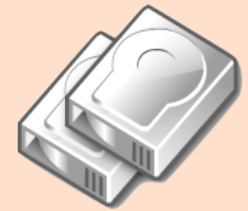
### Operator Pipelining

- Volcano Iterator Model

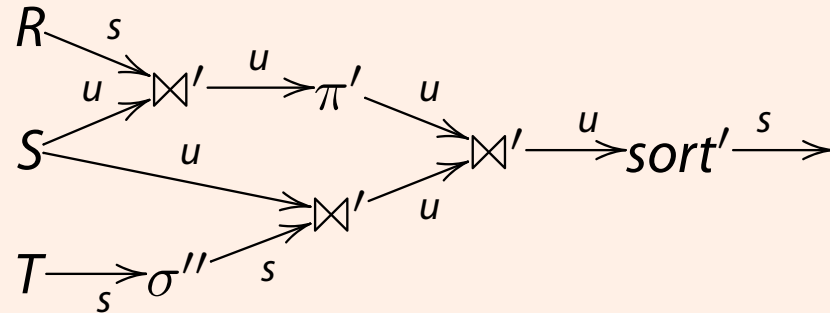
## Physical plan with (un)sortedness annotations (*u/s*)



# Plan Rewriting



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



## Relational Query Engines

Operator Selection

### Selection ( $\sigma$ )

Selectivity

Conjunctive Predicates

Disjunctive Predicates

### Projection ( $\pi$ )

### Join ( $\bowtie$ )

Nested Loops Join

Block Nested Loops Join

Index Nested Loops Join

Sort-Merge Join

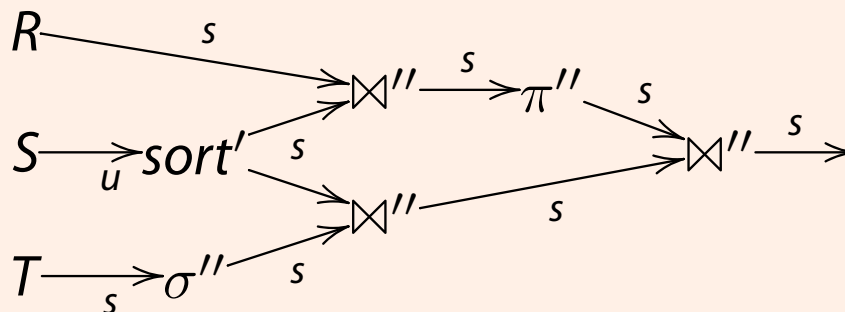
Hash Join

### Operator Pipelining

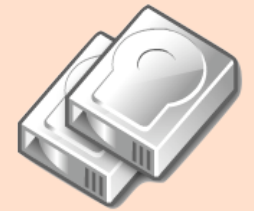
Volcano Iterator Model

- **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 ( $\sigma$ )—No Index, Unsorted Data



## Relational Query Engines

Operator Selection

### Selection ( $\sigma$ )

Selectivity

Conjunctive Predicates

Disjunctive Predicates

### Projection ( $\pi$ )

### Join ( $\bowtie$ )

Nested Loops Join

Block Nested Loops Join

Index Nested Loops Join

Sort-Merge Join

Hash Join

### Operator Pipelining

Volcano Iterator Model

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

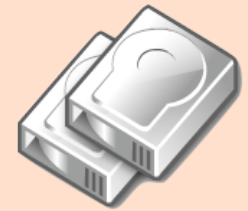
## Selection

```
1 Function:  $\sigma(p, R_{in}, R_{out})$   
2  $out \leftarrow \text{createFile}(R_{out});$   
3  $in \leftarrow \text{openScan}(R_{in});$   
4 while ( $r \leftarrow \text{nextRecord}(in)$ )  $\neq \langle \text{EOF} \rangle$  do  
5   if  $p(r)$  then  
6      $\text{appendRecord}(out, r);$   
7  $\text{closeFile}(out);$ 
```

# Selection ( $\sigma$ )—No Index, Unsorted Data

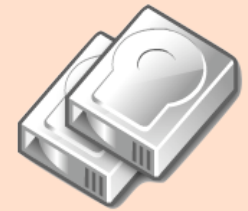
## Remarks:

- Reading the special “record”  $\langle \text{EOF} \rangle$  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 ( $\sigma$ )—No Index, Unsorted Data



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

### Selection ( $\sigma$ )—no index, unsorted data

$\sigma_p (R)$

**input access**<sup>1</sup>

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  $N_R = \lceil |R|/p_R \rceil$ )

<sup>1</sup>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  $p$ :

$$0 \leq sel(p) = \frac{|\sigma_p(R)|}{|R|} \leq 1$$

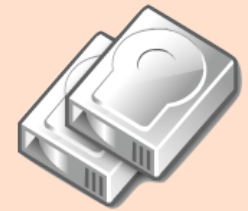
### Selectivity examples

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) selectivities in the operators details of, e.g., its IXSCAN operator.



### Relational Query Engines

Operator Selection

### Selection ( $\sigma$ )

Selectivity

Conjunctive Predicates

Disjunctive Predicates

### Projection ( $\pi$ )

### Join ( $\bowtie$ )

Nested Loops Join

Block Nested Loops Join

Index Nested Loops Join

Sort-Merge Join

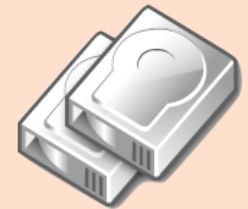
Hash Join

### Operator Pipelining

Volcano Iterator Model

## Selection ( $\sigma$ )—No Index, Sorted Data

- If the input file  $R$  is **sorted** with respect to a sort key  $k$ , we can use **binary search** on  $R$  to find the first record matching predicate  $p$  more quickly. To find more hits, scan the sorted file.
- Obviously, predicate  $p$  must “**match**” the sort key  $k$  in some way. Otherwise we won't benefit from the sortedness of  $R$ .



### Relational Query Engines

Operator Selection

### Selection ( $\sigma$ )

Selectivity

Conjunctive Predicates

Disjunctive Predicates

### Projection ( $\pi$ )

### Join ( $\bowtie$ )

Nested Loops Join

Block Nested Loops Join

Index Nested Loops Join

Sort-Merge Join

Hash Join

### Operator Pipelining

Volcano Iterator Model

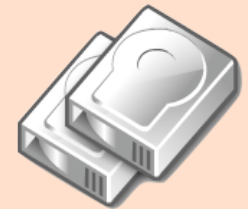
## Selection ( $\sigma$ )—No Index, Sorted Data

- If the input file  $R$  is **sorted** with respect to a sort key  $k$ , we can use **binary search** on  $R$  to find the first record matching predicate  $p$  more quickly. To find more hits, scan the sorted file.
- Obviously, predicate  $p$  must “**match**” the sort key  $k$  in some way. Otherwise we won't benefit from the sortedness of  $R$ .

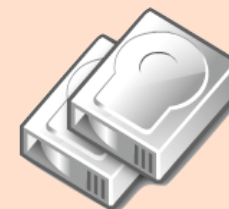
### When does a predicate match a sort key?

Assume  $R$  is sorted on attribute  $A$  in ascending order. Which of the selections below can benefit from the sortedness of  $R$ ?

- 1  $\sigma_{A=42} (R)$
- 2  $\sigma_{A<42} (R)$
- 3  $\sigma_{A>42 \text{ AND } A<100} (R)$
- 4  $\sigma_{A>42 \text{ OR } A>100} (R)$
- 5  $\sigma_{A>42 \text{ AND } A<32} (R)$
- 6  $\sigma_{A>42 \text{ AND } B=10} (R)$
- 7  $\sigma_{A>42 \text{ OR } B=10} (R)$



# Selection ( $\sigma$ )–No Index, Sorted Data



## Relational Query Engines

Operator Selection

## Selection ( $\sigma$ )

Selectivity

Conjunctive Predicates

Disjunctive Predicates

## Projection ( $\pi$ )

## Join ( $\bowtie$ )

Nested Loops Join

Block Nested Loops Join

Index Nested Loops Join

Sort-Merge Join

Hash Join

## Operator Pipelining

Volcano Iterator Model

## Selection ( $\sigma$ )–no index, sorted data

---

$\sigma_p (R)$

**input access**      binary search, then sorted file scan of  $R$

**prerequisites**     $R$  sorted on key  $k$ ,  $p$  matches sort key  $k$

**I/O cost**             $\underbrace{\log_2 N_R}_{\text{binary search}} + \underbrace{sel(p) \cdot N_R}_{\text{sorted scan}} + \underbrace{sel(p) \cdot N_R}_{\text{output cost}}$

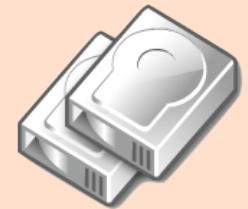
---

## Selection ( $\sigma$ )—B<sup>+</sup>-tree Index

- A **clustered B<sup>+</sup>-tree index** on  $R$  whose key **matches** the selection predicate  $p$  is clearly the superior method to evaluate  $\sigma_p (R)$ :
  - **Descend the B<sup>+</sup>-tree** to retrieve the first index entry to satisfy  $p$ . Then scan the sequence set to find more matching records.
  - 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.

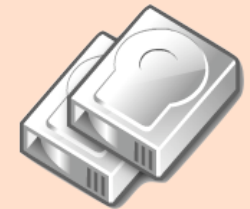
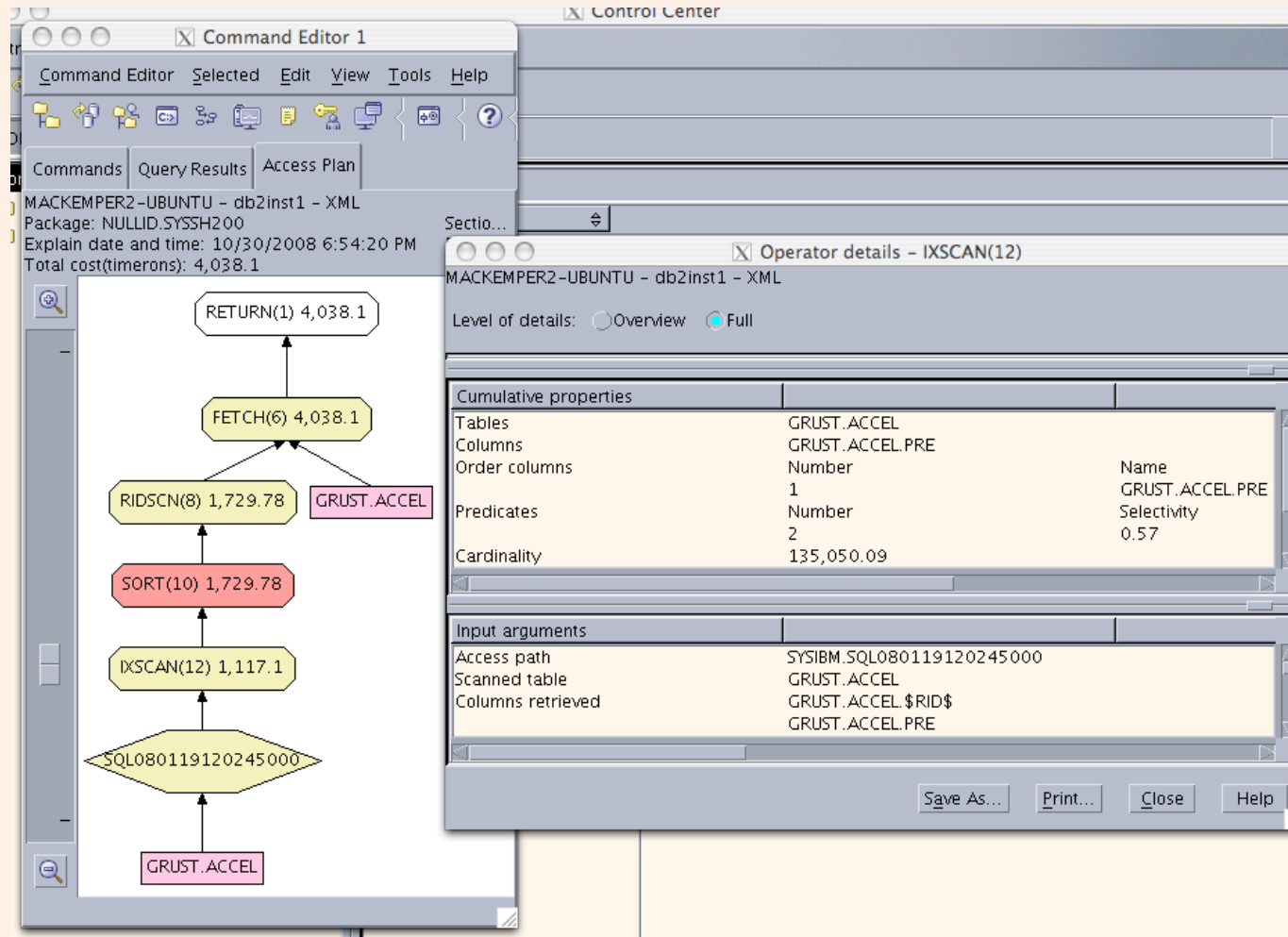
### DB2® Accessing unclustered B<sup>+</sup>-trees

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



# Selection ( $\sigma$ )—B<sup>+</sup>-tree Index

## The IXSCAN/SORT/RIDSCN/FETCH quadruple



### Relational Query Engines

Operator Selection

### Selection ( $\sigma$ )

Selectivity

Conjunctive Predicates

Disjunctive Predicates

### Projection ( $\pi$ )

### Join ( $\bowtie$ )

Nested Loops Join

Block Nested Loops Join

Index Nested Loops Join

Sort-Merge Join

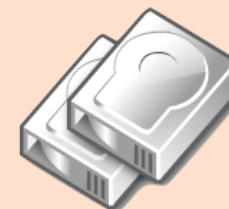
Hash Join

### Operator Pipelining

Volcano Iterator Model

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

# Selection ( $\sigma$ )—B<sup>+</sup>-tree Index



## Relational Query Engines

Operator Selection

## Selection ( $\sigma$ )

Selectivity

Conjunctive Predicates

Disjunctive Predicates

## Projection ( $\pi$ )

## Join ( $\bowtie$ )

Nested Loops Join

Block Nested Loops Join

Index Nested Loops Join

Sort-Merge Join

Hash Join

## Operator Pipelining

Volcano Iterator Model

## Selection ( $\sigma$ )—clustered B<sup>+</sup>-tree index

$\sigma_p(R)$

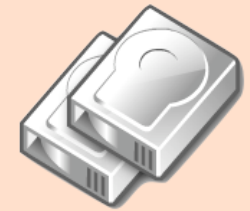
**input access** access of B<sup>+</sup>-tree on  $R$ , then sequence set scan

**prerequisites** clustered B<sup>+</sup>-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 ( $\sigma$ )—Hash Index, Equality Predicate



### Relational Query Engines

Operator Selection

### Selection ( $\sigma$ )

Selectivity

Conjunctive Predicates

Disjunctive Predicates

### Projection ( $\pi$ )

### Join ( $\bowtie$ )

Nested Loops Join

Block Nested Loops Join

Index Nested Loops Join

Sort-Merge Join

Hash Join

### Operator Pipelining

Volcano Iterator Model

- 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<sup>2</sup>. Note that  $sel(p)$  is likely to be close to 0 for equality predicates.

## Selection ( $\sigma$ )—hash index, equality predicate

---

$\sigma_p (R)$

**input access**

hash table on  $R$

**prerequisites**

$r_{in}$  hashed on key  $A$ ,  $p$  has term  $A = c$

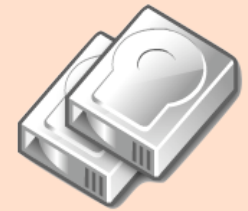
**I/O cost**

$\underbrace{1}_{\text{hash access}} + \underbrace{sel(p) \cdot N_R}_{\text{output cost}}$

---

<sup>2</sup>Remember that this may include access cost for the pages of an overflow chain hanging off the primary bucket page.

# Selection ( $\sigma$ )—Conjunctive Predicates



## Relational Query Engines

Operator Selection

## Selection ( $\sigma$ )

Selectivity

Conjunctive Predicates

Disjunctive Predicates

## Projection ( $\pi$ )

## Join ( $\bowtie$ )

Nested Loops Join

Block Nested Loops Join

Index Nested Loops Join

Sort-Merge Join

Hash Join

## Operator Pipelining

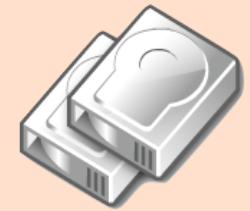
Volcano Iterator Model

- Indeed, selection operations with simple predicates like  $\sigma_{A \theta C}(R)$  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**:

$$\underbrace{A_1 \theta_1 C_1}_{\text{conjunct}} \text{ AND } A_2 \theta_2 C_2 \text{ AND } \dots \text{ AND } A_n \theta_n C_n .$$

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

## Selection ( $\sigma$ )—Conjunctive Predicates



### Relational Query Engines

Operator Selection

### Selection ( $\sigma$ )

Selectivity

Conjunctive Predicates

Disjunctive Predicates

### Projection ( $\pi$ )

### Join ( $\bowtie$ )

Nested Loops Join

Block Nested Loops Join

Index Nested Loops Join

Sort-Merge Join

Hash Join

### Operator Pipelining

Volcano Iterator Model

- Indeed, selection operations with simple predicates like  $\sigma_{A \theta C} (R)$  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**:

$$\underbrace{A_1 \theta_1 C_1}_{\text{conjunct}} \quad \text{AND} \quad A_2 \theta_2 C_2 \quad \text{AND} \quad \dots \quad \text{AND} \quad A_n \theta_n C_n .$$

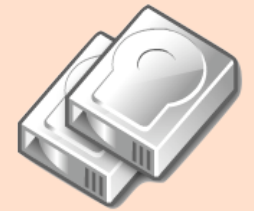
- Here, each **conjunct** is a simple comparison ( $\theta_i \in \{=, <, >, <=, >=\}$ ).
- An index with a multi-attribute key may match the *entire* complex predicate.

# Selection ( $\sigma$ )—Conjunctive Predicates

## Matching a multi-attribute hash index

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

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



### Relational Query Engines

Operator Selection

### Selection ( $\sigma$ )

Selectivity

Conjunctive Predicates

Disjunctive Predicates

### Projection ( $\pi$ )

### Join ( $\bowtie$ )

Nested Loops Join

Block Nested Loops Join

Index Nested Loops Join

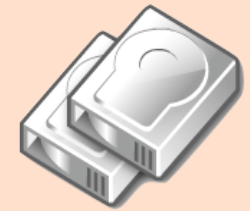
Sort-Merge Join

Hash Join

### Operator Pipelining

Volcano Iterator Model

# Selection ( $\sigma$ )—Conjunctive Predicates



## Matching a multi-attribute hash index

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

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

## Relational Query Engines

Operator Selection

## Selection ( $\sigma$ )

Selectivity

Conjunctive Predicates

Disjunctive Predicates

## Projection ( $\pi$ )

## Join ( $\bowtie$ )

Nested Loops Join

Block Nested Loops Join

Index Nested Loops Join

Sort-Merge Join

Hash Join

## Operator Pipelining

Volcano Iterator Model

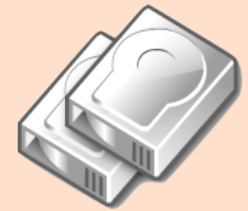
## Conjunctive predicate match rule for hash indexes

A **conjunctive predicate**  $p$  matches a (multi-attribute) hash index with key  $k = (A_1, A_2, \dots, A_n)$ , if  $p$  **covers the key**, *i.e.*,

$$p \equiv A_1 = c_1 \text{ AND } A_2 = c_2 \text{ AND } \dots \text{ AND } A_n = c_n \text{ AND } \phi .$$

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

# Selection ( $\sigma$ )—Conjunctive Predicates

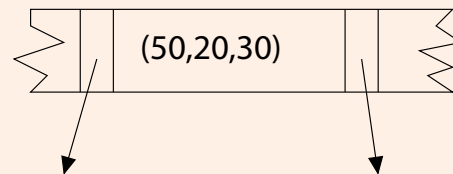


## Matching a multi-attribute B<sup>+</sup>-tree index

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

$$\begin{aligned} k_1 < k_2 &\equiv A_1 < A_2 \vee \\ &\quad (A_1 = A_2 \wedge B_1 < B_2) \vee \\ &\quad (A_1 = A_2 \wedge B_1 = B_2 \wedge C_1 < C_2) \end{aligned}$$

Excerpt of an inner B<sup>+</sup>-tree node (separator):



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

## Relational Query Engines

Operator Selection

## Selection ( $\sigma$ )

Selectivity

Conjunctive Predicates

Disjunctive Predicates

## Projection ( $\pi$ )

## Join ( $\bowtie$ )

Nested Loops Join

Block Nested Loops Join

Index Nested Loops Join

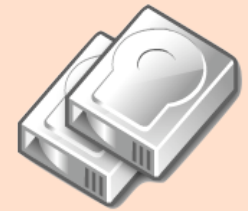
Sort-Merge Join

Hash Join

## Operator Pipelining

Volcano Iterator Model

# Selection ( $\sigma$ )—Conjunctive Predicates



## Conjunctive predicate match rule for B<sup>+</sup>-tree indexes

A **conjunctive predicate**  $p$  matches a (multi-attribute) B<sup>+</sup>-tree **index** with key  $k = (A_1, A_2, \dots, A_n)$ , if  $p$  is a **prefix of the key**, i.e.,

$$p \equiv A_1 \theta_1 c_1 \text{ AND } \phi$$

$$p \equiv A_1 \theta_1 c_1 \text{ AND } A_2 \theta_2 c_2 \text{ AND } \phi$$

$$\vdots$$

$$p \equiv A_1 \theta_1 c_1 \text{ AND } A_2 \theta_2 c_2 \text{ AND } \dots \text{ AND } A_n \theta_n c_n \text{ AND } \phi$$

### Relational Query Engines

Operator Selection

### Selection ( $\sigma$ )

Selectivity

Conjunctive Predicates

Disjunctive Predicates

### Projection ( $\pi$ )

### Join ( $\bowtie$ )

Nested Loops Join

Block Nested Loops Join

Index Nested Loops Join

Sort-Merge Join

Hash Join

### Operator Pipelining

Volcano Iterator Model

- **Note:** Whenever a multi-attribute hash index matches a predicate, so does a B<sup>+</sup>-tree over the same key.

## Selection ( $\sigma$ )—Conjunctive Predicates

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

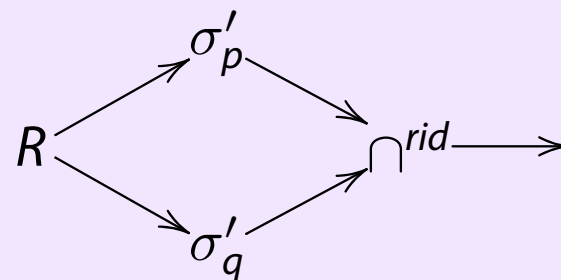
### Example (Partial predicate match)

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

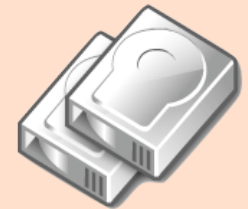
A typical optimizer might thus decide to transform the original query

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

into

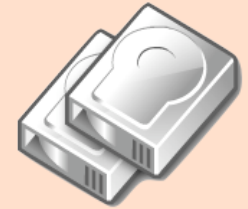


Here,  $\cap^{rid}$  denotes a **set intersection operator defined by *rid* equality** (IBM DB2: IXAND).





# Selection ( $\sigma$ )—Conjunctive Predicates



## Selectivity of conjunctive predicates

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

$$sel(p \text{ AND } q) =$$

### Relational Query Engines

Operator Selection

### Selection ( $\sigma$ )

Selectivity

Conjunctive Predicates

Disjunctive Predicates

### Projection ( $\pi$ )

### Join ( $\bowtie$ )

Nested Loops Join

Block Nested Loops Join

Index Nested Loops Join

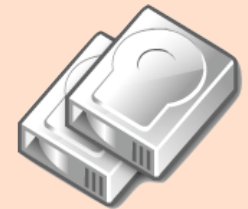
Sort-Merge Join

Hash Join

### Operator Pipelining

Volcano Iterator Model

# Selection ( $\sigma$ )—Conjunctive Predicates



## Selectivity of conjunctive predicates

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

$$sel(p \text{ AND } q) =$$

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

### Relational Query Engines

Operator Selection

### Selection ( $\sigma$ )

Selectivity

Conjunctive Predicates

Disjunctive Predicates

### Projection ( $\pi$ )

### Join ( $\bowtie$ )

Nested Loops Join

Block Nested Loops Join

Index Nested Loops Join

Sort-Merge Join

Hash Join

### Operator Pipelining

Volcano Iterator Model

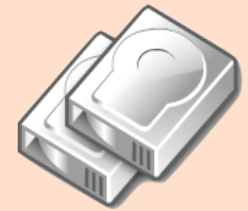
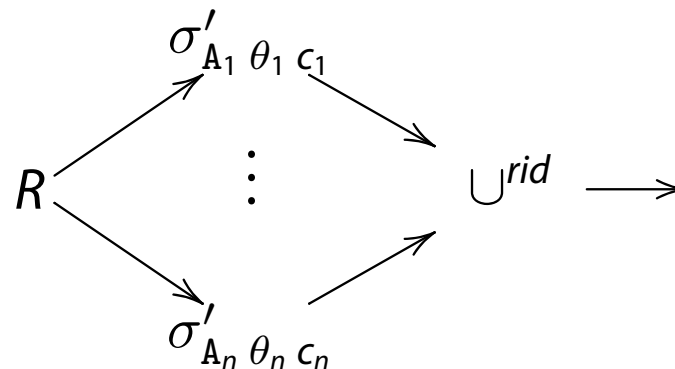
# Selection ( $\sigma$ )—Disjunctive Predicates

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

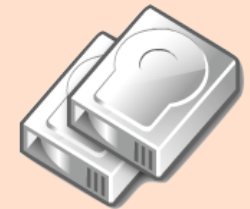
$$A_1 \theta_1 C_1 \text{ OR } A_2 \theta_2 C_2 \text{ OR } \dots \text{ OR } A_n \theta_n C_n$$

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:

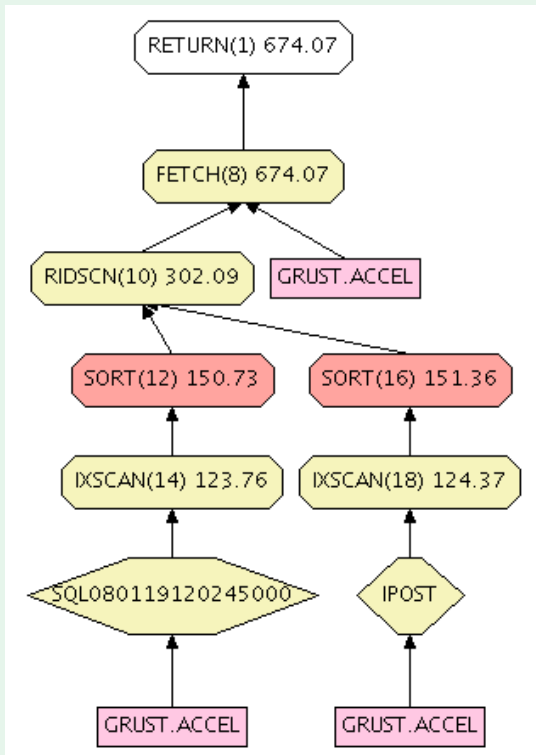


# Selection ( $\sigma$ )—Disjunctive Predicates

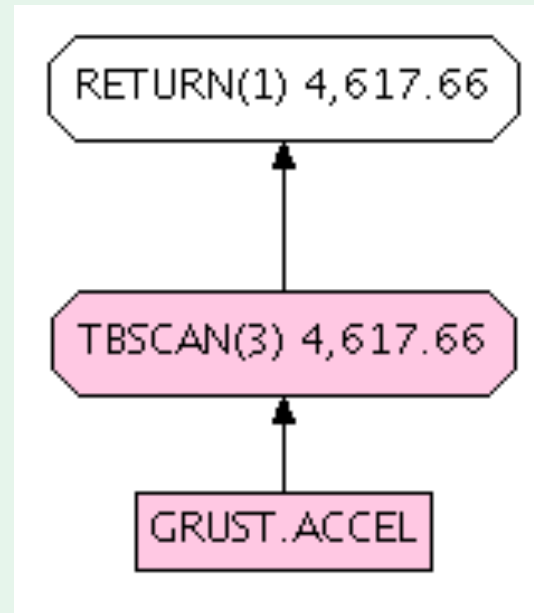


## DB2® Selective disjunctive predicate

## DB2® Non-selective disjunctive predicate



**Note:** Multi-input RIDSCN operator.



**Note:** Presence of indexes ignored.

### Relational Query Engines

Operator Selection

### Selection ( $\sigma$ )

Selectivity

Conjunctive Predicates

Disjunctive Predicates

### Projection ( $\pi$ )

### Join ( $\bowtie$ )

Nested Loops Join

Block Nested Loops Join

Index Nested Loops Join

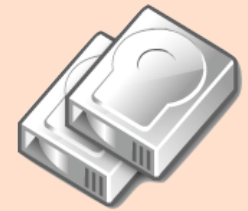
Sort-Merge Join

Hash Join

### Operator Pipelining

Volcano Iterator Model

# Selection ( $\sigma$ )—Disjunctive Predicates



## Relational Query Engines

Operator Selection

## Selection ( $\sigma$ )

Selectivity

Conjunctive Predicates

Disjunctive Predicates

## Projection ( $\pi$ )

## Join ( $\bowtie$ )

Nested Loops Join

Block Nested Loops Join

Index Nested Loops Join

Sort-Merge Join

Hash Join

## Operator Pipelining

Volcano Iterator Model

## Selectivity of disjunctive predicates

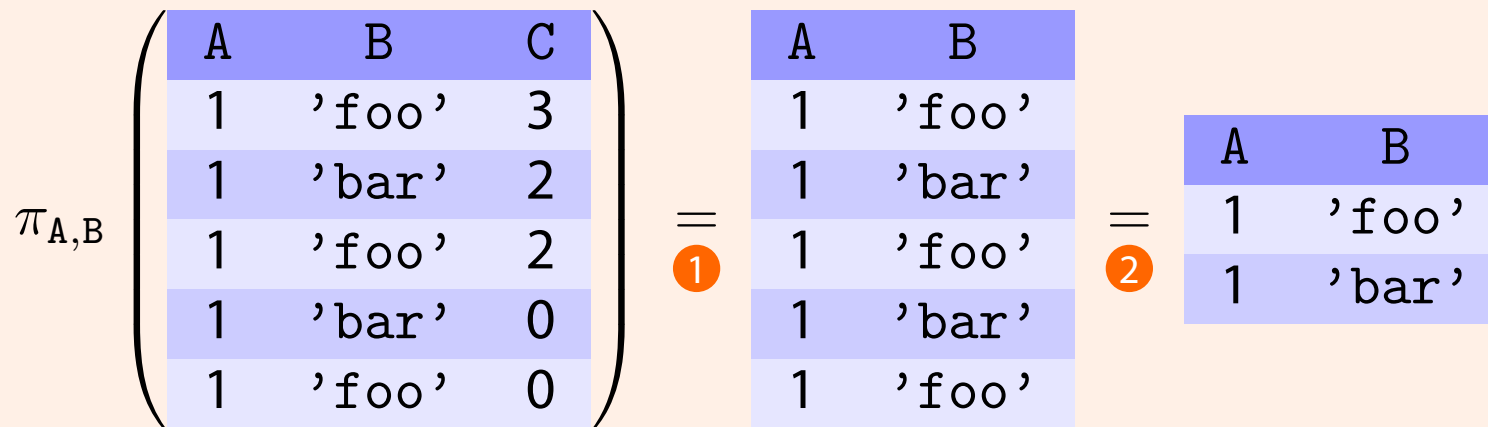
What can you say about the selectivity of the disjunctive predicate  $p \text{ OR } q$ ?

$$sel(p \text{ OR } q) =$$

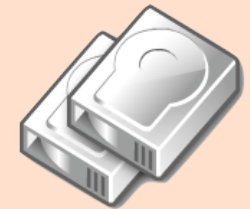
# Projection ( $\pi$ )

- **Projection** ( $\pi_\ell$ ) modifies each record in its input file and cuts off any field not listed in the attribute list  $\ell$ :

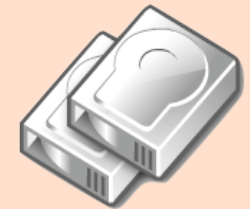
## Relational projection



- 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 ( $\pi$ )—Duplicate Elimination, Sorting



## Relational Query Engines

Operator Selection

## Selection ( $\sigma$ )

Selectivity

Conjunctive Predicates

Disjunctive Predicates

## Projection ( $\pi$ )

## Join ( $\bowtie$ )

Nested Loops Join

Block Nested Loops Join

Index Nested Loops Join

Sort-Merge Join

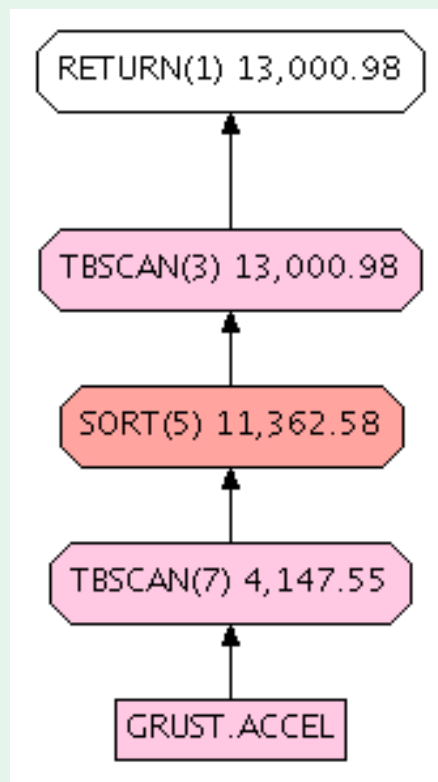
Hash Join

## Operator Pipelining

Volcano Iterator Model

- **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



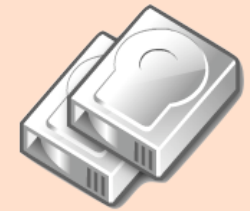
- One benefit of sort-based projection is that operator  $\pi_{\ell}$  will write a sorted output file, *i.e.*,

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

## Sort ordering?

What would be the correct ordering  $\theta$  to apply in the case of duplicate elimination?

# Projection ( $\pi$ )—Duplicate Elimination, Hashing



## Relational Query Engines

Operator Selection

## Selection ( $\sigma$ )

Selectivity

Conjunctive Predicates

Disjunctive Predicates

## Projection ( $\pi$ )

## Join ( $\bowtie$ )

Nested Loops Join

Block Nested Loops Join

Index Nested Loops Join

Sort-Merge Join

Hash Join

## Operator Pipelining

Volcano Iterator Model

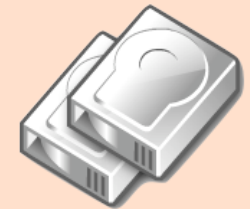
- If the DBMS has a fairly large number of buffer pages ( $B$ , say) to spare for the  $\pi_{\ell}(R)$  operation, a **hash-based** projection may be an efficient alternative to sorting:

## Hash-based projection $\pi_{\ell}$ : partitioning phase

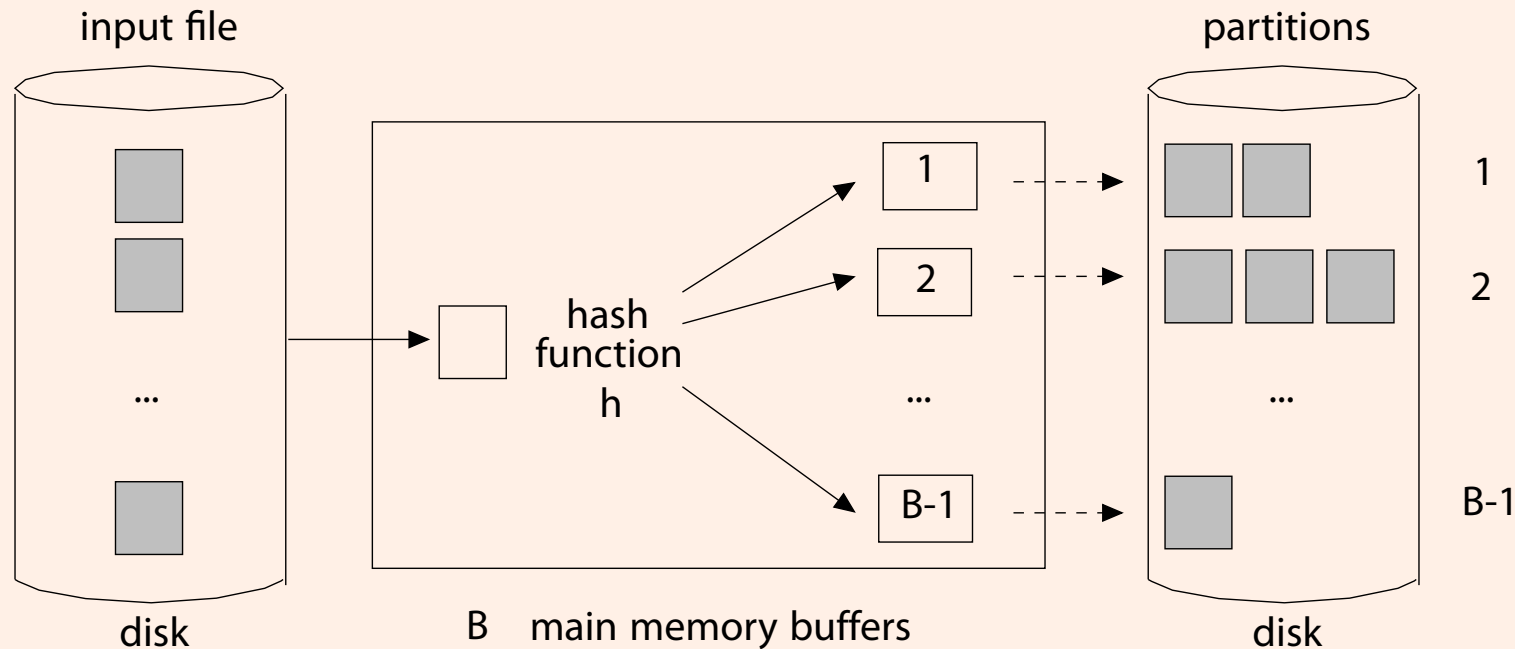
- 1 Allocate all  $B$  buffer pages. One page will be the **input buffer**, the remaining  $B - 1$  pages will be used as **hash buckets**.
- 2 Read the file  $R$  page-by-page, for each record  $r$ : cut off fields not listed in  $\ell$ .
- 3 For each such record, apply hash function  $h_1(r) = h(r) \bmod (B - 1)$ —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 ( $\pi$ )—Hashing



## Hash-based projection $\pi_\ell$ : partitioning phase



### Relational Query Engines

Operator Selection

### Selection ( $\sigma$ )

Selectivity

Conjunctive Predicates

Disjunctive Predicates

### Projection ( $\pi$ )

### Join ( $\bowtie$ )

Nested Loops Join

Block Nested Loops Join

Index Nested Loops Join

Sort-Merge Join

Hash Join

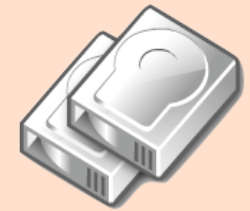
### Operator Pipelining

Volcano Iterator Model

- 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 ( $\pi$ )—Hashing



## Hash-based projection $\pi_h$ : duplicate elimination phase

- 1 For each partition, read each partition page-by-page (possibly in parallel).
- 2 To each record, apply hash function  $h_2 \neq h_1$  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).

### Relational Query Engines

Operator Selection

### Selection ( $\sigma$ )

Selectivity

Conjunctive Predicates

Disjunctive Predicates

### Projection ( $\pi$ )

### Join ( $\bowtie$ )

Nested Loops Join

Block Nested Loops Join

Index Nested Loops Join

Sort-Merge Join

Hash Join

### Operator Pipelining

Volcano Iterator Model

## Huge partitions?

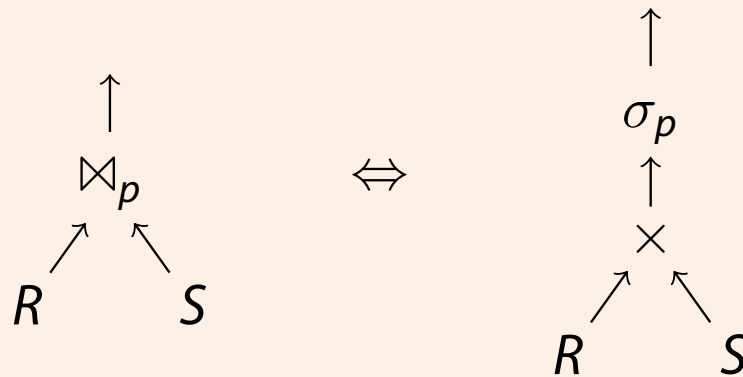
**Note:** Works efficiently only if duplicate elimination phase can be **performed in the buffer** (main memory).

**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**  $\sigma_p$ .

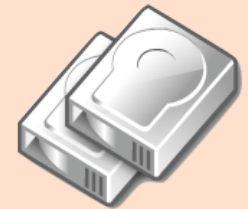
## Join vs. Cartesian product



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  $|R| \cdot |S|$ ).



### Relational Query Engines

Operator Selection

### Selection ( $\sigma$ )

Selectivity

Conjunctive Predicates

Disjunctive Predicates

### Projection ( $\pi$ )

### Join ( $\bowtie$ )

Nested Loops Join

Block Nested Loops Join

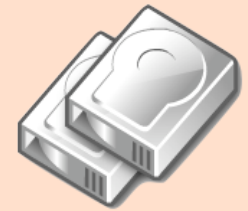
Index Nested Loops Join

Sort-Merge Join

Hash Join

### Operator Pipelining

Volcano Iterator Model

Relational Query  
Engines

Operator Selection

Selection ( $\sigma$ )

Selectivity

Conjunctive Predicates

Disjunctive Predicates

Projection ( $\pi$ )Join ( $\bowtie$ )

Nested Loops Join

Block Nested Loops Join

Index Nested Loops Join

Sort-Merge Join

Hash Join

## Operator Pipelining

Volcano Iterator Model

## Nested Loops Join

The **nested loops join** is the straightforward implementation of the  $\sigma \times$  combination:

### Nested loops join

```

1 Function: nljoin( $R, S, p$ )
   /* outer relation  $R$  */
2 foreach record  $r \in R$  do
   /* inner relation  $S$  */
3   foreach record  $s \in S$  do
4     /*  $\langle r, s \rangle$  denotes record concatenation */
5     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

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

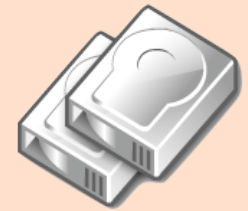
# Nested Loops Join: I/O Behavior

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.



## Relational Query Engines

Operator Selection

### Selection ( $\sigma$ )

Selectivity

Conjunctive Predicates

Disjunctive Predicates

### Projection ( $\pi$ )

### Join ( $\bowtie$ )

Nested Loops Join

Block Nested Loops Join

Index Nested Loops Join

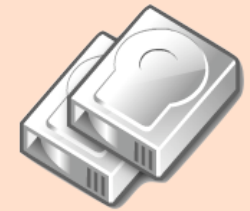
Sort-Merge Join

Hash Join

### Operator Pipelining

Volcano Iterator Model

# Block Nested Loops Join



## Relational Query Engines

Operator Selection

## Selection ( $\sigma$ )

Selectivity

Conjunctive Predicates

Disjunctive Predicates

## Projection ( $\pi$ )

## Join ( $\bowtie$ )

Nested Loops Join

Block Nested Loops Join

Index Nested Loops Join

Sort-Merge Join

Hash Join

## Operator Pipelining

Volcano Iterator Model

- 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

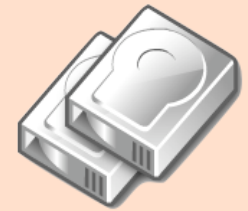
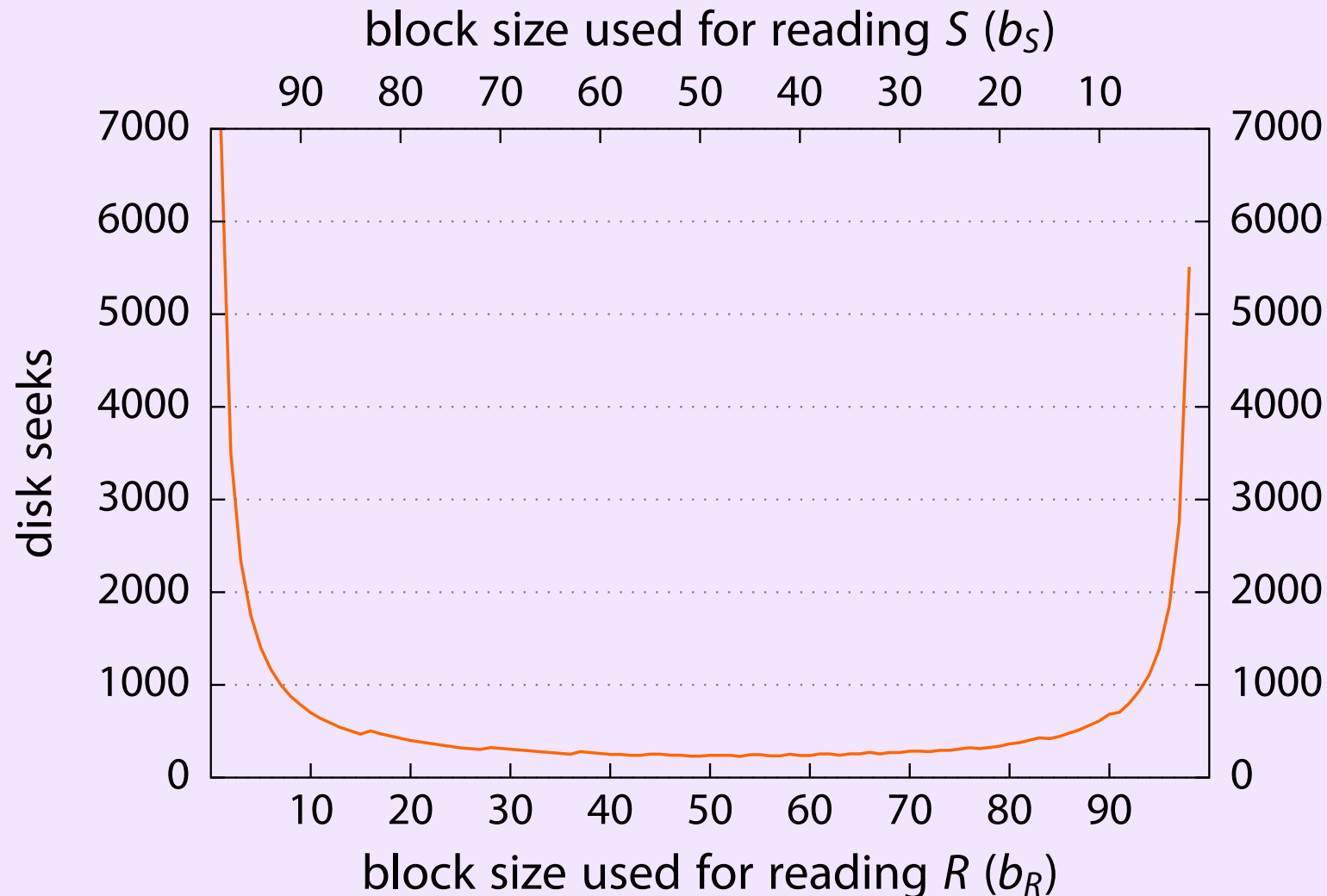
```
1 Function: block_nljoin ( $R, S, p$ )  
2 foreach  $b_R$ -sized block in  $R$  do  
3   foreach  $b_S$ -sized block in  $S$  do  
4     /* performed in the buffer */  
     find matches in current  $R$ - and  $S$ -blocks and append them  
     to the result ;
```

- $R$  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 $b_s$ )



### Relational Query Engines

Operator Selection

### Selection ( $\sigma$ )

Selectivity

Conjunctive Predicates

Disjunctive Predicates

### Projection ( $\pi$ )

### Join ( $\bowtie$ )

Nested Loops Join

Block Nested Loops Join

Index Nested Loops Join

Sort-Merge Join

Hash Join

### Operator Pipelining

Volcano Iterator Model

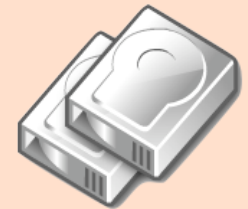
# In-Memory Join Performance

- 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**.



### Relational Query Engines

Operator Selection

### Selection ( $\sigma$ )

Selectivity

Conjunctive Predicates

Disjunctive Predicates

### Projection ( $\pi$ )

### Join ( $\bowtie$ )

Nested Loops Join

Block Nested Loops Join

Index Nested Loops Join

Sort-Merge Join

Hash Join

### Operator Pipelining

Volcano Iterator Model



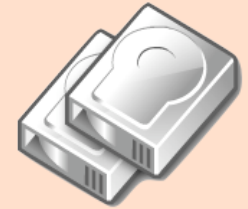
# Index Nested Loops Join

The **index nested loops join** takes advantage of an index on the **inner** relation (swap *outer*  $\leftrightarrow$  *inner* if necessary):

## Index nested loops join

```
1 Function: index_nljoin( $R, S, p$ )  
2 foreach record  $r \in R$  do  
3   scan  $S$ -index using (key value in)  $r$  and concatenate  $r$  with all  
4   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  $\nearrow$  Selinger *et al.*, *SIGMOD* 1979)



## Relational Query Engines

Operator Selection

## Selection ( $\sigma$ )

Selectivity

Conjunctive Predicates

Disjunctive Predicates

## Projection ( $\pi$ )

## Join ( $\bowtie$ )

Nested Loops Join

Block Nested Loops Join

Index Nested Loops Join

Sort-Merge Join

Hash Join

## Operator Pipelining

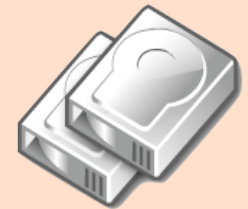
Volcano Iterator Model

# 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:  $N_{\text{idx}}$  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**.



## Relational Query Engines

Operator Selection

## Selection ( $\sigma$ )

Selectivity

Conjunctive Predicates

Disjunctive Predicates

## Projection ( $\pi$ )

## Join ( $\bowtie$ )

Nested Loops Join

Block Nested Loops Join

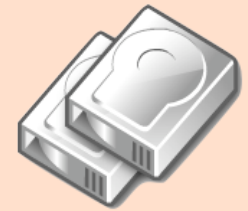
Index Nested Loops Join

Sort-Merge Join

Hash Join

## Operator Pipelining

Volcano Iterator Model



## Relational Query Engines

Operator Selection

### Selection ( $\sigma$ )

Selectivity

Conjunctive Predicates

Disjunctive Predicates

### Projection ( $\pi$ )

### Join ( $\bowtie$ )

Nested Loops Join

Block Nested Loops Join

Index Nested Loops Join

Sort-Merge Join

Hash Join

### Operator Pipelining

Volcano Iterator Model

If the index is a **B<sup>+</sup>-tree index**:

- A **single** index access requires the inspection of  $h$  pages.<sup>3</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 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).

---

<sup>3</sup> $h$ : B<sup>+</sup>-tree height

## Sort-Merge Join

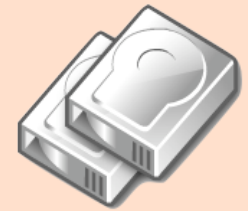
Join computation becomes particularly simple if both inputs are **sorted** with respect to the join attribute(s).

- The **merge join** essentially **merges** both input tables, much like we did for sorting. 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		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**.



### Relational Query Engines

Operator Selection

### Selection ( $\sigma$ )

Selectivity

Conjunctive Predicates

Disjunctive Predicates

### Projection ( $\pi$ )

### Join ( $\bowtie$ )

Nested Loops Join

Block Nested Loops Join

Index Nested Loops Join

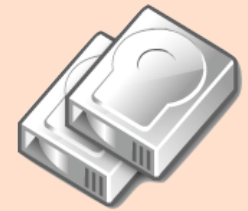
Sort-Merge Join

Hash Join

### Operator Pipelining

Volcano Iterator Model

# Merge Join: Algorithm



## Merge join algorithm

```
1 Function: merge_join( $R, S, \alpha = \beta$ ) //  $\alpha, \beta$ : join cols 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  $r \neq \langle \text{EOF} \rangle$  and  $s \neq \langle \text{EOF} \rangle$  do //  $\langle \text{EOF} \rangle$ : end of file marker
5     while  $r.\alpha < s.\beta$  do
6         | advance  $r$ ;
7     while  $r.\alpha > s.\beta$  do
8         | 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            | append  $\langle r, s \rangle$  to result;
14            | advance  $s$ ;
15        | advance  $r$ ;
```

### Relational Query Engines

Operator Selection

### Selection ( $\sigma$ )

Selectivity

Conjunctive Predicates

Disjunctive Predicates

### Projection ( $\pi$ )

### Join ( $\bowtie$ )

Nested Loops Join

Block Nested Loops Join

Index Nested Loops Join

Sort-Merge Join

Hash Join

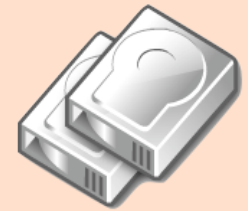
### Operator Pipelining

Volcano Iterator Model

# 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?**



## Relational Query Engines

Operator Selection

## Selection ( $\sigma$ )

Selectivity

Conjunctive Predicates

Disjunctive Predicates

## Projection ( $\pi$ )

## Join ( $\bowtie$ )

Nested Loops Join

Block Nested Loops Join

Index Nested Loops Join

Sort-Merge Join

Hash Join

## Operator Pipelining

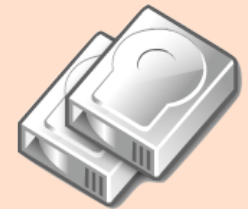
Volcano Iterator Model

# 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.



### Relational Query Engines

Operator Selection

### Selection ( $\sigma$ )

Selectivity

Conjunctive Predicates

Disjunctive Predicates

### Projection ( $\pi$ )

### Join ( $\bowtie$ )

Nested Loops Join

Block Nested Loops Join

Index Nested Loops Join

Sort-Merge Join

Hash Join

### Operator Pipelining

Volcano Iterator Model

# Merge Join: IBM DB2 Plan

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

The screenshot displays the IBM DB2 Command Editor interface. The main window shows a query plan for a Merge Join. The plan consists of the following operators:

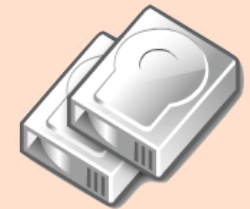
- RETURN(1)** (Total cost: 40,954.06)
- MSJOIN(3)** (Total cost: 40,954.06)
- TBSCAN(5)** (Total cost: 34,814.32)
- IXSCAN(16)** (Total cost: 1,905.08)
- IPOST** (Total cost: 4,147.55)
- SORT(7)** (Total cost: 30,548.27)
- FILTER(11)** (Total cost: 5,744.49)
- FETCH(14)** (Total cost: 5,744.49)
- GRUST.ACCEL** (Table)

The **Operator details - SORT(7)** window is open, showing the following information:

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: 1 Name: GRUST.ACCEL.PRE
Predicates	None



### Relational Query Engines

Operator Selection

### Selection ( $\sigma$ )

Selectivity

Conjunctive Predicates

Disjunctive Predicates

### Projection ( $\pi$ )

### Join ( $\bowtie$ )

Nested Loops Join

Block Nested Loops Join

Index Nested Loops Join

Sort-Merge Join

Hash Join

### Operator Pipelining

Volcano Iterator Model

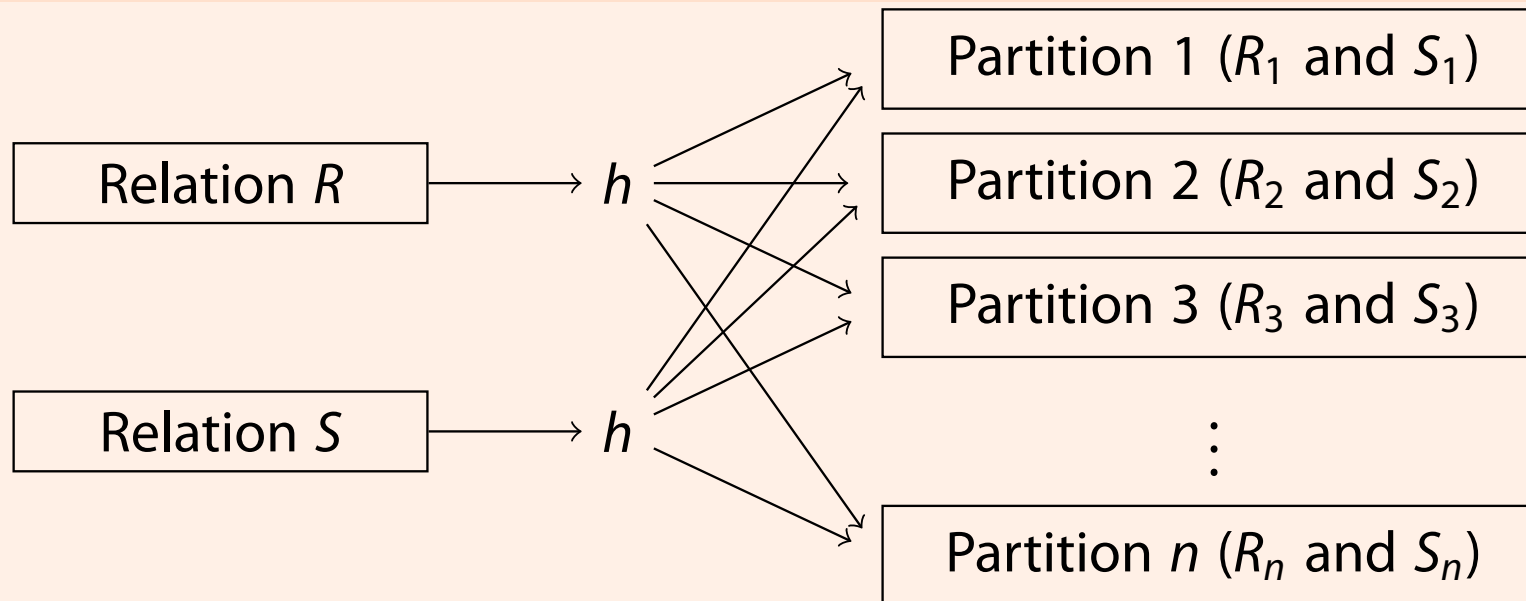
- **Note:** The **FILTER(11)** implements the join predicate of the **MSJOIN(3)**.



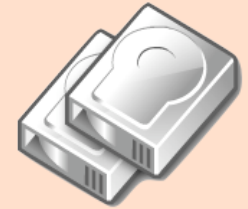
# Hash Join

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

## Hash partitioning for both inputs



- Observe that  $R_i \bowtie S_j = \emptyset$  for all  $i \neq j$ .



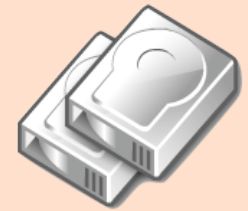
# Hash Join

- 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 (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 35).

## Intra-partition join via hash table

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

**Why?**



### Relational Query Engines

Operator Selection

### Selection ( $\sigma$ )

Selectivity

Conjunctive Predicates

Disjunctive Predicates

### Projection ( $\pi$ )

### Join ( $\bowtie$ )

Nested Loops Join

Block Nested Loops Join

Index Nested Loops Join

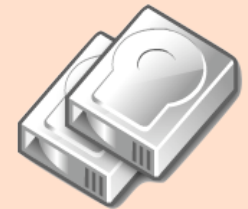
Sort-Merge Join

Hash Join

### Operator Pipelining

Volcano Iterator Model

# Hash Join Algorithm



## Hash join

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

### Relational Query Engines

Operator Selection

### Selection ( $\sigma$ )

Selectivity

Conjunctive Predicates

Disjunctive Predicates

### Projection ( $\pi$ )

### Join ( $\bowtie$ )

Nested Loops Join

Block Nested Loops Join

Index Nested Loops Join

Sort-Merge Join

Hash Join

### Operator Pipelining

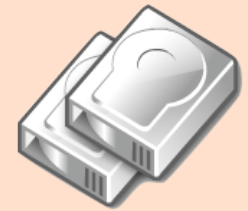
Volcano Iterator Model

# 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).



## Relational Query Engines

Operator Selection

### Selection ( $\sigma$ )

Selectivity

Conjunctive Predicates

Disjunctive Predicates

### Projection ( $\pi$ )

### Join ( $\bowtie$ )

Nested Loops Join

Block Nested Loops Join

Index Nested Loops Join

Sort-Merge Join

Hash Join

### Operator Pipelining

Volcano Iterator Model

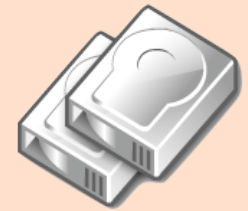
# Hash Join—Buffer Requirements

- 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).



### Relational Query Engines

Operator Selection

### Selection ( $\sigma$ )

Selectivity

Conjunctive Predicates

Disjunctive Predicates

### Projection ( $\pi$ )

### Join ( $\bowtie$ )

Nested Loops Join

Block Nested Loops Join

Index Nested Loops Join

Sort-Merge Join

Hash Join

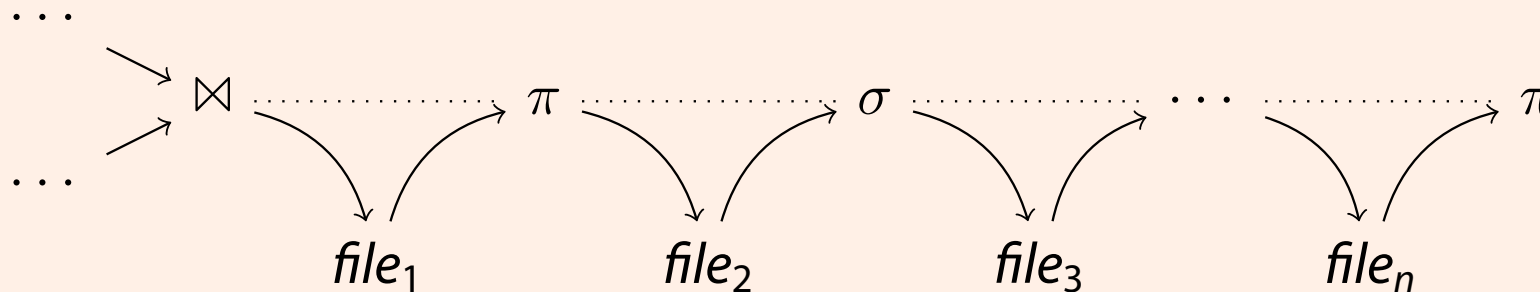
### Operator Pipelining

Volcano Iterator Model

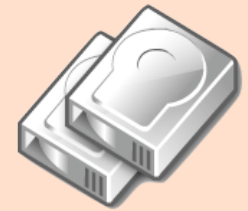
# Orchestrating Operator Evaluation

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

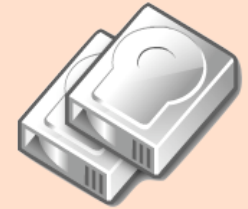
## File-based operator input and output



- 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



## Relational Query Engines

Operator Selection

## Selection ( $\sigma$ )

Selectivity

Conjunctive Predicates

Disjunctive Predicates

## Projection ( $\pi$ )

## Join ( $\bowtie$ )

Nested Loops Join

Block Nested Loops Join

Index Nested Loops Join

Sort-Merge Join

Hash Join

## Operator Pipelining

Volcano Iterator Model

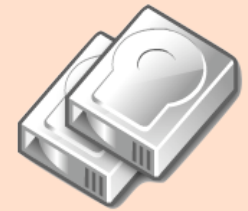
- Architecting the query processor in this fashion bears much resemblance with using the Unix shell like this:

## File-based Unix command sequencing

```
1 # report "large" XML files below current working dir
2 $ find . -size +500k      > tmp1
3 $ xargs file              < tmp1 > tmp2
4 $ grep -i XML             < tmp2 > tmp3
5 $ cut -d: -f1             < tmp3
6 <output generated here>
7
8 # remove temporary files
9 $ 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**.



## Relational Query Engines

Operator Selection

### Selection ( $\sigma$ )

Selectivity

Conjunctive Predicates

Disjunctive Predicates

### Projection ( $\pi$ )

### Join ( $\bowtie$ )

Nested Loops Join

Block Nested Loops Join

Index Nested Loops Join

Sort-Merge Join

Hash Join

## Operator Pipelining

Volcano Iterator Model



# Unix: Pipelines of Processes

Unix provides a similar mechanism to communicate between processes (“operators”):

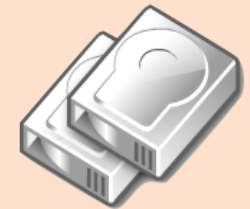
## Pipeline-based Unix command sequencing

```
1 $ find . -size +500k | xargs file | grep -i XML | cut -d: -f1
2 <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.



### Relational Query Engines

Operator Selection

### Selection ( $\sigma$ )

Selectivity

Conjunctive Predicates

Disjunctive Predicates

### Projection ( $\pi$ )

### Join ( $\bowtie$ )

Nested Loops Join

Block Nested Loops Join

Index Nested Loops Join

Sort-Merge Join

Hash Join

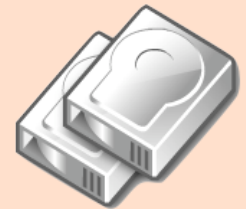
### Operator Pipelining

Volcano Iterator Model

# 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  $\langle \text{EOF} \rangle$ .
  - 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 Extensible and Parallel Query Evaluation System. *Trans. Knowl. Data Eng.* vol. 6, no. 1, February 1994.



## Relational Query Engines

Operator Selection

## Selection ( $\sigma$ )

Selectivity

Conjunctive Predicates

Disjunctive Predicates

## Projection ( $\pi$ )

## Join ( $\bowtie$ )

Nested Loops Join

Block Nested Loops Join

Index Nested Loops Join

Sort-Merge Join

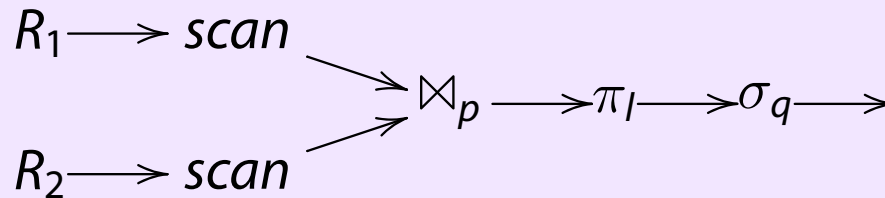
Hash Join

## Operator Pipelining

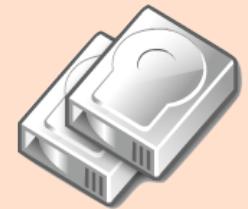
Volcano Iterator Model

# Volcano-Style Pipelined Evaluation

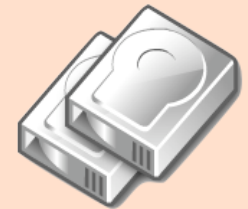
## Example (Pipelined query plan)



- 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):
  - The whole plan is initially reset by calling `open ()` on the **root operator**, *i.e.*, `σq.open ()`.
  - The `open ()` call is **forwarded** through the plan by the operators themselves (see `σ.open ()` on slide 54).
  - Control returns to the query processor.
  - The root is requested to produce its next result record, *i.e.*, the call `σq.next ()` is made.
  - 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



## Relational Query Engines

Operator Selection

### Selection ( $\sigma$ )

Selectivity

Conjunctive Predicates

Disjunctive Predicates

### Projection ( $\pi$ )

### Join ( $\bowtie$ )

Nested Loops Join

Block Nested Loops Join

Index Nested Loops Join

Sort-Merge Join

Hash Join

### Operator Pipelining

Volcano Iterator Model

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

## Query plan evaluation driver

```
1 Function: eval (q)
2 q.open ();
3 r ← q.next ();
4 while r ≠ ⟨EOF⟩ do
5     | /* deliver record r (print, ship to DB client) */
6     | emit (r);
7     | r ← q.next ();
8     /* resource deallocation now */
9 q.close ();
```

# Volcano-Style Selection ( $\sigma_p$ )

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

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

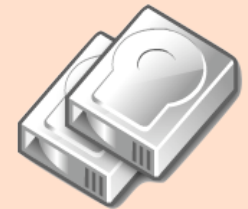
```
1 Function: open ()
2  $R.open () ;$ 


---


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


---


1 Function: next ()
2 while  $((r \leftarrow R.next ()) \neq \langle EOF \rangle)$  do
3   if  $p(r)$  then
4     return  $r ;$ 
5 return  $\langle EOF \rangle ;$ 
```



### Relational Query Engines

Operator Selection

### Selection ( $\sigma$ )

Selectivity

Conjunctive Predicates

Disjunctive Predicates

### Projection ( $\pi$ )

### Join ( $\bowtie$ )

Nested Loops Join

Block Nested Loops Join

Index Nested Loops Join

Sort-Merge Join

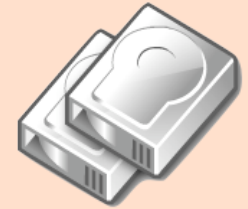
Hash Join

### Operator Pipelining

Volcano Iterator Model

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

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



## Relational Query Engines

Operator Selection

## Selection ( $\sigma$ )

Selectivity

Conjunctive Predicates

Disjunctive Predicates

## Projection ( $\pi$ )

## Join ( $\bowtie$ )

Nested Loops Join

Block Nested Loops Join

Index Nested Loops Join

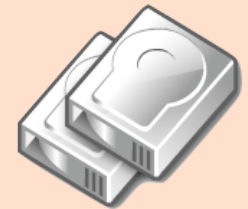
Sort-Merge Join

Hash Join

## Operator Pipelining

Volcano Iterator Model

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



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

```
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 ( $r \neq \langle EOF \rangle$ ) do
3   while ( $(s \leftarrow S.next ()) \neq \langle EOF \rangle$ ) do
4     if  $p(r, s)$  then
5       return  $\langle r, s \rangle$  ;
6     /* reset inner join input */
7     S.close ();
8     S.open ();
9      $r \leftarrow R.next ()$  ;
9 return  $\langle EOF \rangle$  ;
```

### Relational Query Engines

Operator Selection

### Selection ( $\sigma$ )

Selectivity

Conjunctive Predicates

Disjunctive Predicates

### Projection ( $\pi$ )

### Join ( $\bowtie$ )

Nested Loops Join

Block Nested Loops Join

Index Nested Loops Join

Sort-Merge Join

Hash Join

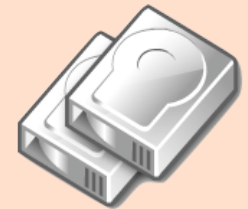
### Operator Pipelining

Volcano Iterator Model

# Pipelining (Real DBMS Product)

## Volcano-style pipelined selection operator (C code)

```
1 /* eFLTR -- apply filter predicate pred to stream
2    Filter the in-bound stream, only stream elements that fulfill
3    e->pred contribute to the result.  No index support. */
4
5 eRC eOp_FLTR(eOp *ip) {
6     eObj_FLTR *e = (eObj_FLTR *)eObj(ip);
7
8     while (eIntp(e->in) != eEOS) {
9         eIntp(e->pred);
10        if (eT_as_bool(eVal(e->pred))) {
11            eVal(ip) = eVal(e->in);
12            return eOK;
13        }
14    }
15    return eEOS;
16 }
17
18 eRC eOp_FLTR_RST(eOp *ip) {
19     eObj_FLTR *e = (eObj_FLTR *)eObj(ip);
20
21     eReset(e->in);
22     eReset(e->pred);
23     return eOK;
24 }
```



### Relational Query Engines

Operator Selection

### Selection ( $\sigma$ )

Selectivity

Conjunctive Predicates

Disjunctive Predicates

### Projection ( $\pi$ )

### Join ( $\bowtie$ )

Nested Loops Join

Block Nested Loops Join

Index Nested Loops Join

Sort-Merge Join

Hash Join

### Operator Pipelining

Volcano Iterator Model

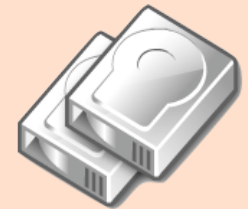


# Blocking Operators

- 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.



### Relational Query Engines

Operator Selection

### Selection ( $\sigma$ )

Selectivity

Conjunctive Predicates

Disjunctive Predicates

### Projection ( $\pi$ )

### Join ( $\bowtie$ )

Nested Loops Join

Block Nested Loops Join

Index Nested Loops Join

Sort-Merge Join

Hash Join

### Operator Pipelining

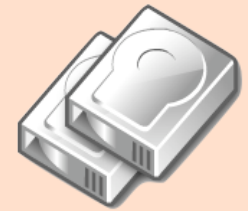
Volcano Iterator Model

# 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.



### Relational Query Engines

Operator Selection

### Selection ( $\sigma$ )

Selectivity

Conjunctive Predicates

Disjunctive Predicates

### Projection ( $\pi$ )

### Join ( $\bowtie$ )

Nested Loops Join

Block Nested Loops Join

Index Nested Loops Join

Sort-Merge Join

Hash Join

### Operator Pipelining

Volcano Iterator Model