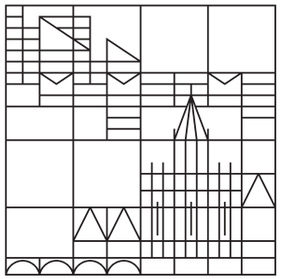
Universität
Konstanz



Database System Architecture and Implementation

Module 7

Evaluating Relational Operators

December 4, 2018

Orientation

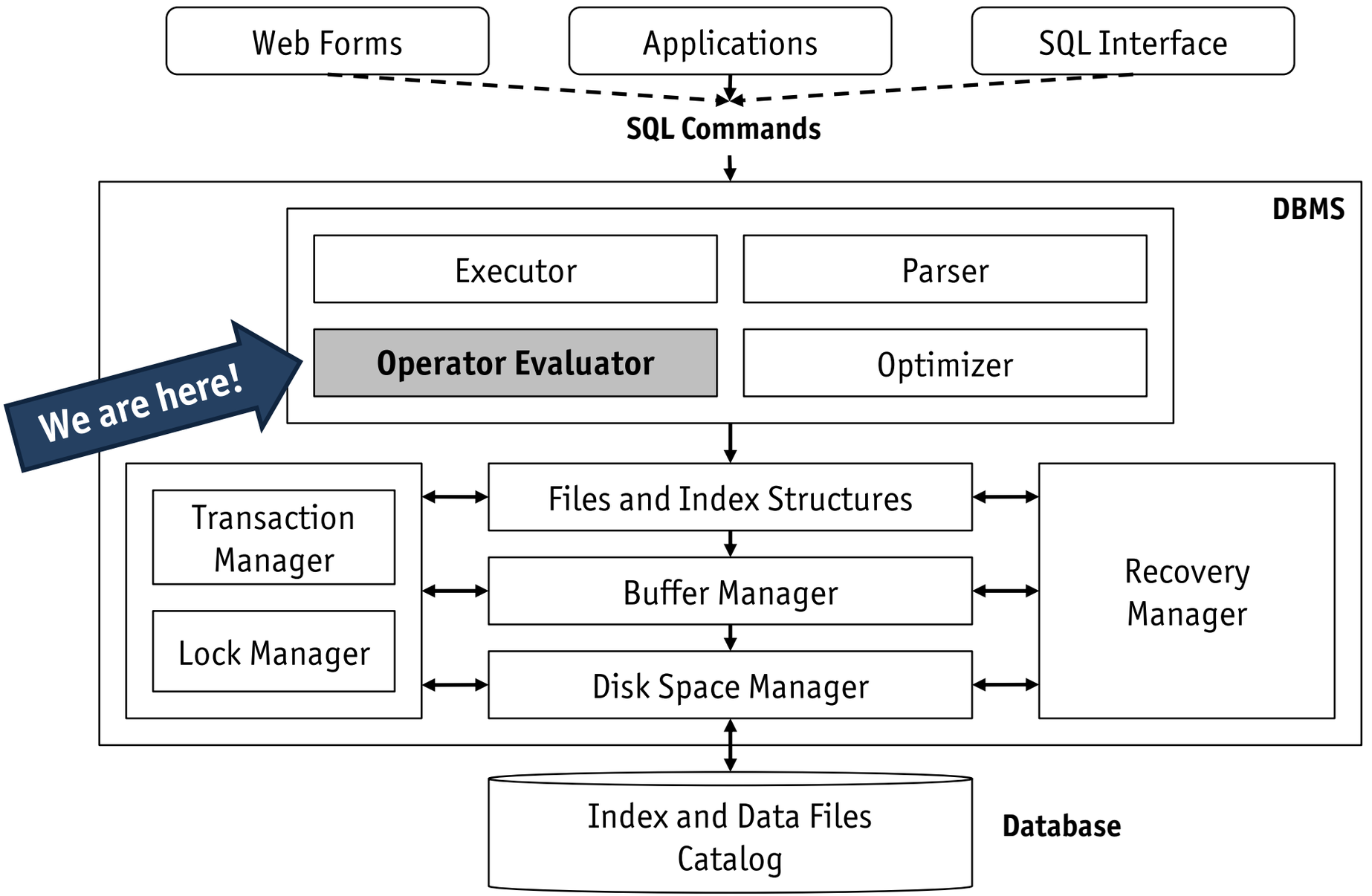


Figure Credit: Raghu Ramakrishnan and Johannes Gehrke: "Database Management Systems", McGraw-Hill, 2003.

Module Overview

- Binary operators
  - join
  - set operations
- Unary operators
  - selection
  - projection
- Extended relational algebra operators
  - grouping
  - aggregation
- Impact of buffering

Running Example

A simple schema

```
CREATE TABLE Sailors (
  sid      INTEGER,
  sname    STRING,
  rating   INTEGER,
  age      REAL,
  PRIMARY KEY (sid)
)
```

```
CREATE Table Reserves (
  sid      INTEGER,
  bid      INTEGER,
  day      DATE,
  rname    STRING,
  PRIMARY KEY (sid, bid, day),
  FOREIGN KEY sid REFERENCES
    Sailors(sid)
)
```

- Assumptions
  - relation **`Sailors`**: 50 bytes/tuple, 80 tuples/page, and 500 pages
  - relation **`Reserves`**: 40 bytes/tuple, 100 tuples/page, and 1000 pages

↳ 4 kB page size, 2 MB **`Sailors`** data, and 4 MB **`Reserves`** data

Operator Variants

- Operator variants can be implemented using specific algorithms that are tailored to exploit **physical properties** of the system
  - **presence or absence of indexes** on the input file(s)
  - **sortedness** of the input file(s)
  - **size** of the input file(s)
  - available **buffer space in the buffer pool** (as with external sorting)
  - **buffer replacement policy**, …

Logical and physical operators

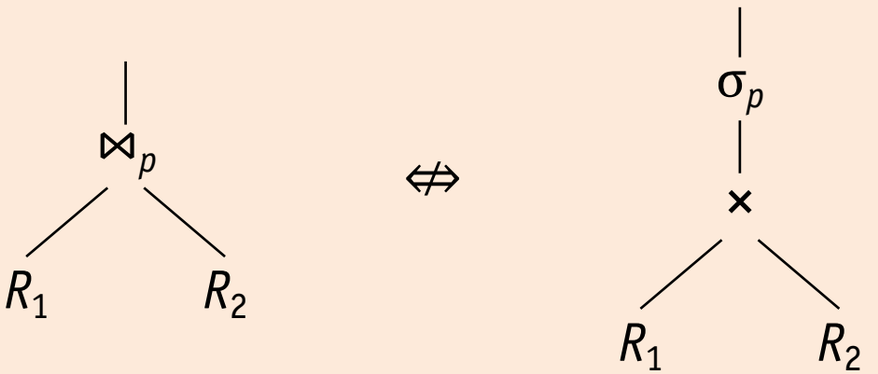
Such a specific operator variant is referred to as a **physical operator**. In general, physical operators implement the **logical operators** of the relational algebra. During query processing, the **query optimizer** replaces logical operators by physical operators based on its knowledge of system internals, statistics, and ongoing book-keeping. It selects the best (or most reasonable) variant for each operator.

Relational Operator Evaluation

- Several **alternative algorithms** can be used to implement each relational operators
  - for most operators no algorithm has universally superior performance
  - performance depends on sizes of involved tables, existing indexes and sort orders, size of buffer pool and buffer replacement policy
- Alternative algorithms for relational operators often use **common implementation techniques**
  - **indexing:** an index is used to only process tuples that satisfy a selection or join condition
  - **iteration:** process all tuples of an input table, one after the other, or scan all index data entries of an index that contains all required attributes (not using the search structure of the index)
  - **partitioning:** decompose an operation into a less expensive collection of operations by partitioning tuples on a sort key (e.g., sorting and hashing)

Join

Join vs. Cartesian product



- Semantics of **join operator** $\bowtie_p$ is equivalent to combination of **cross product** × and **selection** σ_p
- One way to implement $\bowtie_p$ is to use this relational equivalence
  1. enumerate all record pairs in the cross product $R_1 \times R_2$
  2. pick those record pairs that satisfy the predicate p
- More advance algorithms try to avoid the obvious inefficiency in Step 1 (the size of the intermediate result is $|R_1| \cdot |R_2|$)

Nested Loops Join

- The **nested loops join** is the straightforward implementation of the cross product (×) and selection (σ) combination
- For obvious reasons, R_1 is referred to as the **outer** (relation), whereas R_2 is called the **inner** (relation)

Nested loops join

```
function ⋈^nl(R1, R2, Rout, p)
  in1 ← openScan(R1);
  out ← createFile(Rout);
  while (r1 ← nextRecord(in1)) ≠ ⟨EOF⟩ do
    in2 ← openScan(R2);
    while (r2 ← nextRecord(in2)) ≠ ⟨EOF⟩ do
      if p(r1, r2) then appendRecord(out, ⟨r1, r2⟩);
  closeFile(out);
end
```

Nested Loops Join

Cost of $R_1 \bowtie_p^{nl} R_2$

access path	file scan (`openScan`) of R_1 and R_2
prerequisites	none (p arbitrary, R_1 and R_2 may be heap files)
I/O cost	$\underbrace{\|\|R_1\|\|}_{\text{outer loop}} + \underbrace{\|R_1\| \cdot \|\|R_2\|\|}_{\text{inner loop}}$

- Remarks
  - algorithm can easily be refined such that one scan of the inner relation is initiated for each **page** of the outer relation (instead of each **record**)
  - in this case, the cost of the inner loop is $\|\|R_1\|\| \cdot \|\|R_2\|\|$

Nested Loops Join

- The **good news** is that $\bowtie^{nl}$ only needs **three pages** of buffer space (two to read R_1 and R_2, one to write the result)
- The **bad news** is its **staggering I/O cost**, since it effectively enumerate all records in the cross product $R_1 \times R_2$

Exercise: execution time of $\bowtie_p^{nl}$

Assume that $\|R_1\| = 1000$ pages and $\|R_2\| = 500$ pages. Both relations store 100 tuples per page. If the access time is 10 ms per page, how long will the join $R_1 \bowtie_p^{nl} R_2$ take?

Block Nested Loops Join

- The **block nested loops join** saves random access cost by reading R_1 and R_2 in blocks of, say b_1 and b_2 pages
  - R_1 is still read once, with only $\lceil \|R_1\| / b_1 \rceil$ disk seeks
  - R_2 is scanned $\lceil \|R_1\| / b_1 \rceil$ times, with $\lceil \|R_1\| / b_1 \rceil \cdot \lceil \|R_2\| / b_2 \rceil$ disk seeks

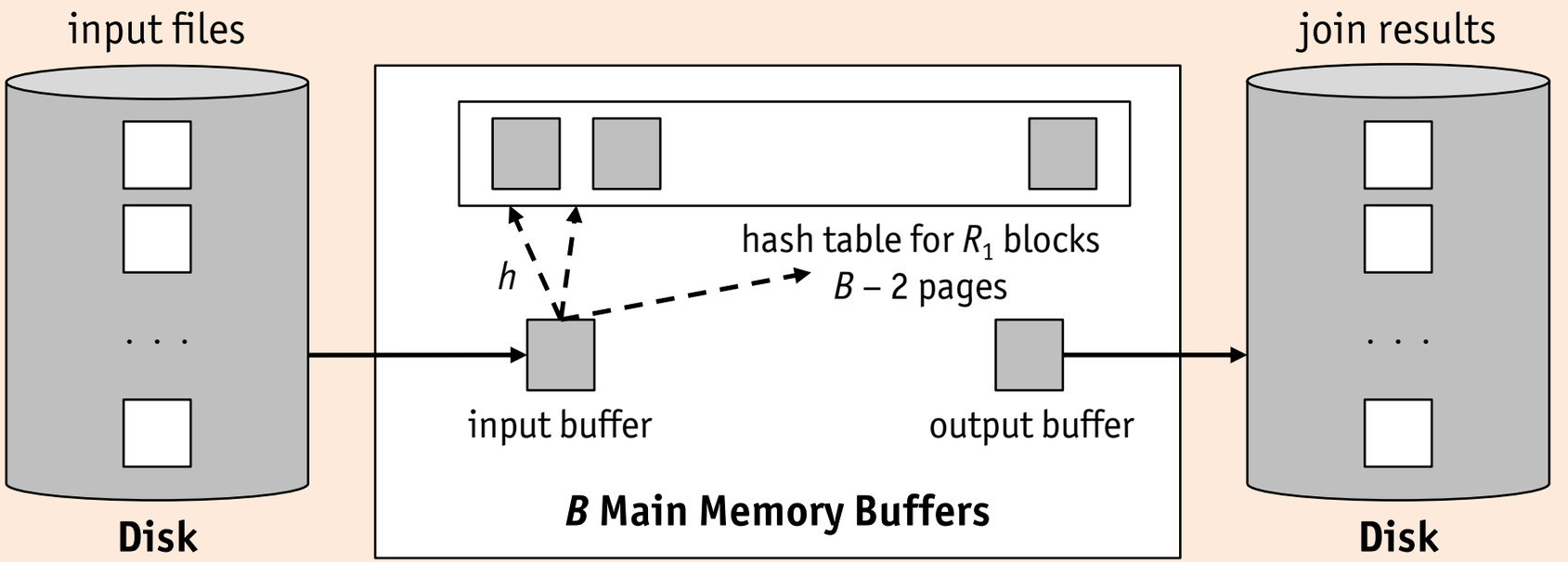
General block nested loops join

```
function ⋈^block-nl (R1, R2, Rout, p)
   in1 ← openScan(R1) ;
   out ← createFile(Rout) ;
   foreach b1-sized block in in1 do
      in2 ← openScan(R2) ;
      foreach b2-sized block in in2 do
         for all matching in-memory tuples r1 ∈ R1 block and r2 ∈ R2 blocks,
         add ⟨r1, r2⟩ to the result out
   closeFile(out) ;
end
```

Block Nested Loops Join

- Building a hash table over the R_1 block can speed up the **in-memory join** between the R_1 and R_2 blocks considerably
  - read outer relation R_1 in **blocks of** $B - 2$ **pages**
  - this optimization only works for **equi-joins**

Block nested loops join with hash table



Block Nested Loops Join

Block nested loops join with hash table

```
function ⋈^block-nl (R1, R2, Rout, p)
  in1 ← openScan(R1) ;
  out ← createFile(Rout) ;
  repeat
    B′ ← min(B − 2, #remaining blocks in R1) ;        (do not read beyond ⟨EOF⟩ of R1)
    if B′ > 0 then
      read B′ blocks of R1 into buffer, hash record r ∈ R1 to buffer page h(r.A1) mod B′ ;
      in2 ← openScan(R2) ;
      while (r2 ← nextRecord(in2)) ≠ ⟨EOF⟩ do
        compare record r2 with records r1 stored in buffer page h(r2.A2) mod B′ ;
        if r1.A1 = r2.A2 then appendRecord(out, ⟨r1, r2⟩) ;
  until B′ < B − 2 ;
  closeFile(out) ;
end
```

Block Nested Loops Join

Exercise: execution time of $\bowtie_p^{block\text{-}nl}$

As before, assume that $\|R_1\|$ = 1000 pages and $\|R_2\|$ = 500 pages. If the access time is 10 ms per page, how long will the join $R_1 \bowtie_p^{block\text{-}nl} R_2$ take, using B = 100 buffer pages?

Index Nested Loops Join

- The **index nested loops** join takes advantage of an index on the inner relation (swap *outer* ↔ *inner*, if necessary)
  - index must **match** join condition p
  - index nested loops **avoids** enumeration of the cross product

⌨ Index nested loops join

```
function ⋈^index-nl (R1, R2, Rout, p)
  in1 ← openScan(R1) ;
  out ← createFile(Rout) ;
  while (r1 ← nextRecord(in1) ) ≠ ⟨EOF⟩ do
    probe index on R2 using (key value in) r1 to find matching tuples r2 ∈ R2 ;
    appendRecord(out, ⟨r1, r2⟩) ;
  closeFile(out) ;
end
```

Index Nested Loops Join

- For each tuple in R_1, the index is probed for matching R_2 tuples
- Breakdown of costs
  1. **access** index to find its first matching entry costs
  2. **scan** the index to retrieve **all** n matching rids (I/O cost for this step is typically negligible due to locality in the index)
  3. **fetching** the n matching R_2 tuples
- Note that the cost of an index nested loops joins **depends on the size of the join result**, due to points 2 and 3 above

Exercise: quantifying these costs

You have seen a few cost analyses so far and should be able to quantify these costs for clustered or unclustered hash and B+ tree indexes!

Index Nested Loops Join

Cost of $R_1 \bowtie_p^{\text{index-nl}} R_2$

access path	file scan (`openScan`) of R_1, index access to R_2
prerequisites	index on R_2 that matches join predicate p
I/O cost	$\underbrace{\|\|R_1\|\|}_{\text{outer loop}} + \underbrace{\|R_1\| \cdot (\text{cost of } \textbf{one} \text{ index access to } R_2)}_{\text{inner loop}}$

- Remarks
  - $\bowtie_p^{\text{index-nl}}$ is particularly useful if the index is a **clustered index**
  - even with **unclustered indexes** and few matches per outer tuple, $\bowtie_p^{\text{index-nl}}$ outperform simple nested loops
  - overall, the use of $\bowtie_p^{\text{index-nl}}$ pays off if the join is **selective** (picks out a few tuples only from a big table)

Index Nested Loops Join

Exercise: execution time of $\bowtie_p^{\textit{index-nl}}$

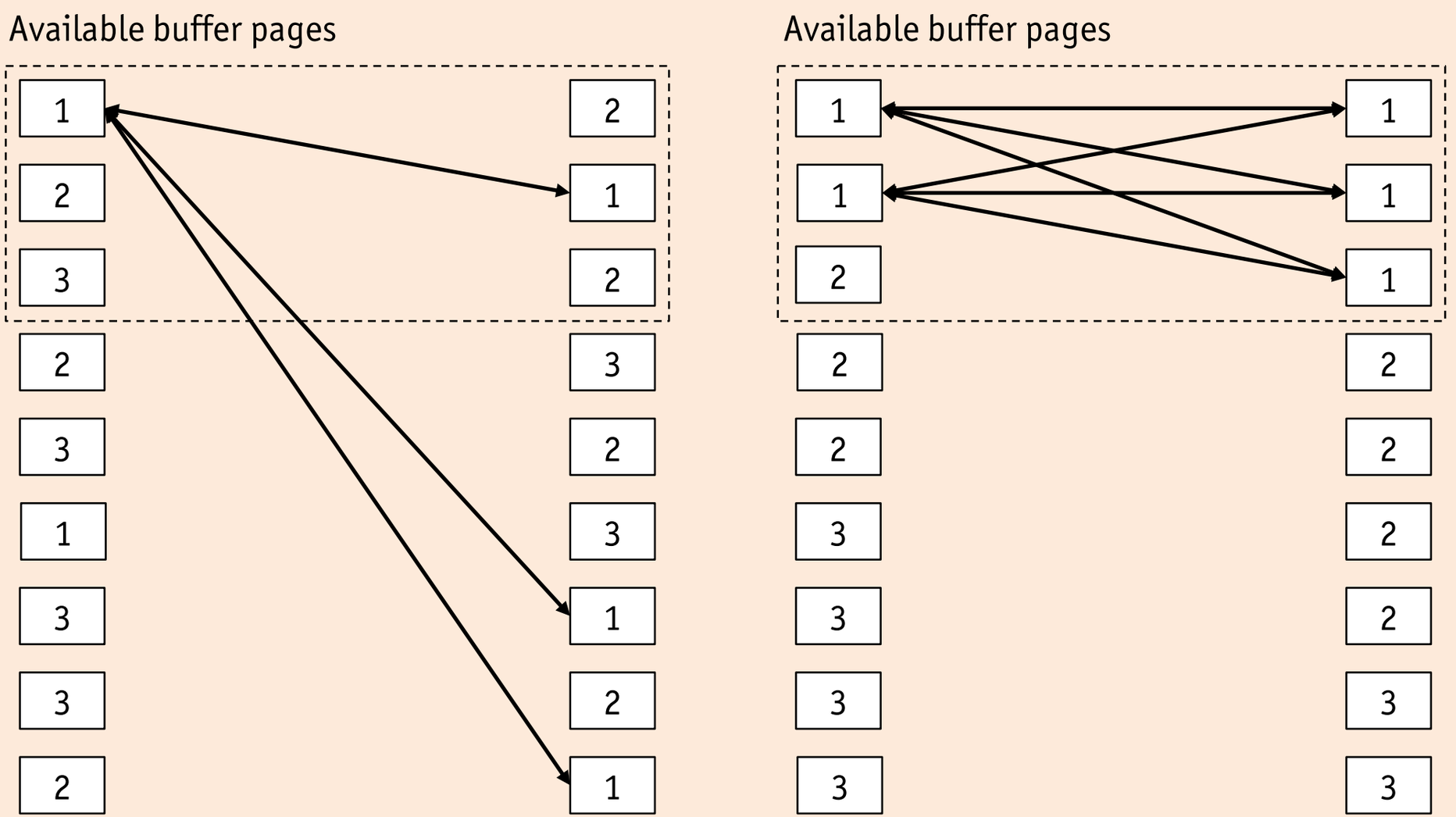
If the access time is 10 ms per page, how long will the join

$$\textit{Reserves} \bowtie_{\textit{sid}=\textit{sid}}^{\textit{index-nl}} \textit{Sailors}$$

take, assuming there is a hash-based index on *Reserves*.*sid* that uses index entries of alternative ❷?

Partitioning for Joins

Illustration



Sort-Merge Join

- The **sort-merge join** uses sorting to partition both inputs
  - sort both relations on the join attribute
  - look for qualifying tuples by merging the two relations
- Sort-merge join is typically used for equi-joins only
- Merge phase similar to merge passes of external sort
  - simply consider input relations as two runs that have to be merged
  - slight adaptation required to deal duplicates on both sides

Multiple matches per tuple (disrupts sequential access)

A	B
Chuck	1
Sarah	2
Casey	2
Morgan	2
Ellie	4

$\bowtie_{B=C}$

C	D
1	true
2	false
2	true
3	false

Sort-Merge Join

Sort-merge join

```
function ⋈^sort-merge (R1, R2, Rout, R1.A = R2.B)
  if R1 not sorted on A then sort it;        if R2 not sorted on B then sort it;
  out ← createFile(Rout);
  in1 ← openScan(R1);                        in2 ← openScan(R2);
  r1 ← nextRecord(in1);                      r2 ← nextRecord(in2);
  while r1 ≠ ⟨EOF⟩ ∧ r2 ≠ ⟨EOF⟩ do
    while r1.A < r2.B do r1 ← nextRecord(in1);
    while r1.A > r2.B do r2 ← nextRecord(in2);
    r'2 ← r2;                                  (remember current position in R2)
    while r1.A = r'2.B do                      (all R1 tuples with the same A value)
      r2 ← r'2;                                (rewind r2 to r'2)
      while r1.A = r2.B do                     (all R2 tuples with the same B value)
        appendRecord(out, ⟨r1, r2⟩);
        r2 ← nextRecord(in2);
      r1 ← nextRecord(in1);
  closeFile(out);
end
```

Sort-Merge Join

Exercise: best case and worst case of $\bowtie^{\textit{sort-merge}}$

What is the **best case** and what is the **worst case scenario** for $R_1 \bowtie_{A=B}^{\textit{sort-merge}} R_2$?

↳ **best case**

↳ **worst case**

Sort-Merge Join

Cost of $R_1 \bowtie_{A=B}^{\text{sort-merge}} R_2$

access path	sorted file scan of R_1 and R_2
prerequisites	p equality predicate $R_1.A = R_2.B$
I/O cost	cost of sorting R_1 and/or R_2, if not sorted already, plus
	best case: $\|\|R_1\|\| + \|\|R_2\|\|$
	worst case: $\|\|R_1\|\| \cdot \|\|R_2\|\|$

- Remarks
  - best case of sort-merge join is optimal
  - sort-merge join operator can be integrated into the final sort pass of the external sort operator
  - blocked I/O, double buffering, and replacement sort can also be applied to speed up the sorting and merging of the input relations
  - output of sort-merge join is also sorted on join attribute

Sort-Merge Join

Exercise: execution time of $\bowtie_p^{\textit{sort-merge}}$

If the access time is 10 ms per page, how long will the join

$$\textit{Reserves} \bowtie_{\textit{sid}=\textit{sid}}^{\textit{sort-merge}} \textit{Sailors}$$

take in the best and the worst case, assuming both relations are sorted on *sid*?

- **best case**

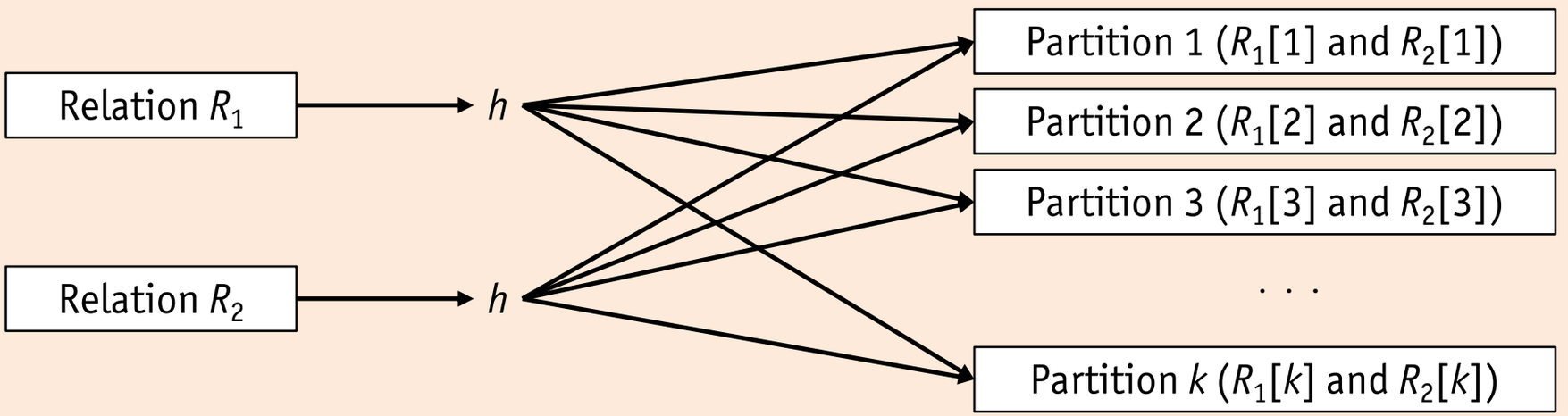
- **worst case**

Hash Joins

- Like the sort-merge join algorithm, join algorithms in the **family of hash joins** work in two phases
  - **partitioning** (or **building**) **phase** uses hashing to divide R_1 and R_2 into k partitions $R_1[k]$ and $R_2[k]$
  - **probing** (or **matching**) **phase** only compares tuples in partition $R_1[i]$ to tuples in the corresponding partition $R_2[i]$
- Algorithm follows **divide and conquer** principle
  - instead of one big disk-based join, do many small in-memory joins
  - observe that $R_1[\mathrm{i}] \bowtie R_2[\mathrm{j}] = \emptyset$ for all $\mathrm{i} \neq \mathrm{j}$
- Examples
  - Grace hash join (named after GRACE database management system)
  - hybrid hash join
  - … there are many more!

Grace Hash Join

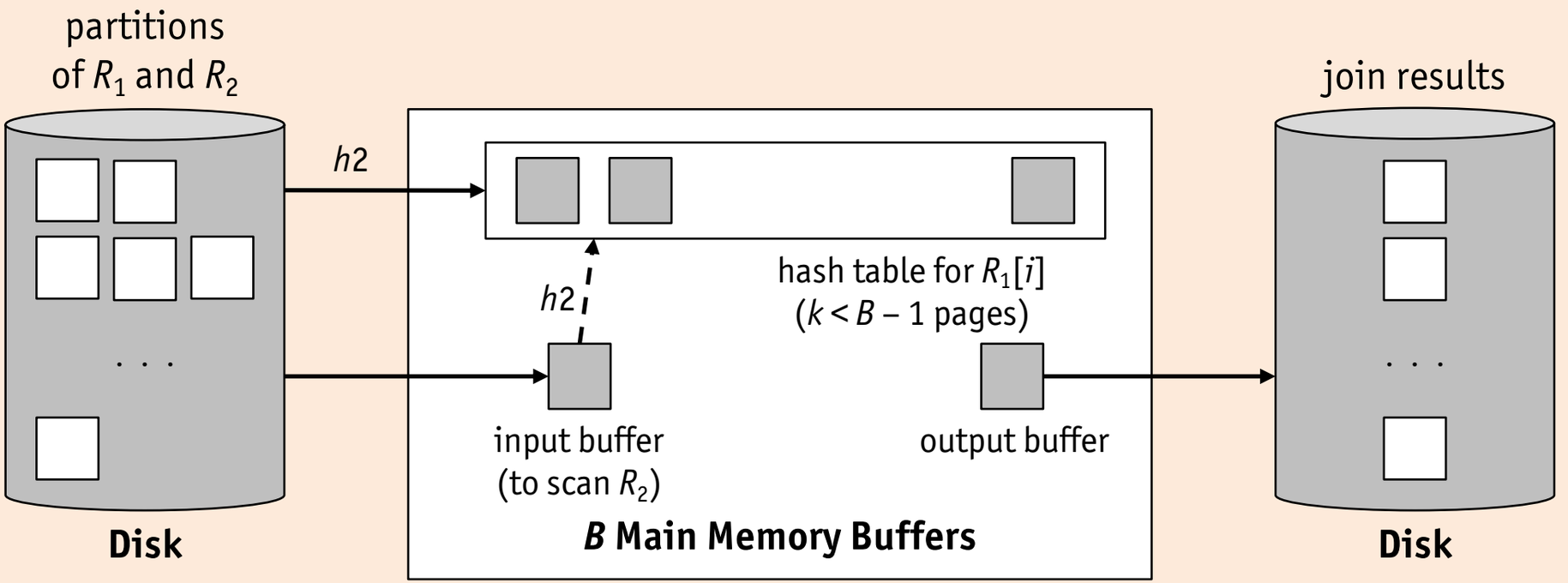
Partitioning phase of hash join



- Partitioning phase scans each input relation in turn, applies hash function *h*, and writes out *k* hash buckets
- Matching tuples are guaranteed to end up together in same partition (works for **equality predicates** only)

Grace Hash Join

Probing phase of a hash join



- Probing phase scans each of the k buckets one, and computes an intra-partition **in-memory join** (hopefully)
- As with the block nested loops join, the in-memory join accelerated using a hash table with hash function *h*2

Grace Hash Join

Grace hash join

function $\bowtie^{\textit{hash-join}}$ **(**R_1, R_2, R_{out}, $R_1.A = R_2.B$**)**
  $in_1 \leftarrow$ **openScan (**R_1**) ;** *(partitioning phase)*
  while ($r_1 \leftarrow$ **nextRecord (**in_1**)**) $\neq$ ⟨**EOF**⟩ **do**
    add r_1 to partition $R_1[h(r_1.A)]$; *(flushed as page fills)*
  $in_2 \leftarrow$ **openScan (**R_2**) ;**
  while ($r_2 \leftarrow$ **nextRecord (**in_2**)**) $\neq$ ⟨**EOF**⟩ **do**
    add r_2 to partition $R_2[h(r_2.B)]$; *(flushed as page fills)*
  $out \leftarrow$ **createFile (**R_{out}**) ;**
  foreach $i \in 1, \ldots, k$ **do** *(probing phase)*
    foreach tuple $r_1 \in R_1[i]$ **do** *(build in-memory hash table for $R_1[i]$, using h2)*
      insert r_1 into hash table H, using $h2(r_1.A)$;
    foreach tuple $r_2 \in R_2[i]$ **do** *(scan $R_2[i]$ and probe for matching $R_1[i]$ tuples)*
      probe H using $h_2(r2.B)$ and append matching tuples $\langle r_1, r_2 \rangle$ to out ;
    clear H to prepare for next partition ;
  closeFile (out**) ;**
end

Grace Hash Join

Cost of $R_1 \bowtie_{A=B}^{\text{hash-join}} R_2$

access path	file scan (`openScan`) of R_1 and R_2
prerequisites	equi-join, i.e., p equality predicate $R_1.A = R_2.B$
I/O cost	$\underbrace{\underbrace{\|\|R_1\|\| + \|\|R_2\|\|}_{\text{read}} + \underbrace{\|\|R_1\|\| + \|\|R_2\|\|}_{\text{write}}}_{\text{partitioning phase}} + \underbrace{\|\|R_1\|\| + \|\|R_2\|\|}_{\text{probing phase}} = 3 \cdot (\|\|R_1\|\| + \|\|R_2\|\|)$

- Remarks
  - the partitioning phase reads each page of R_1 and R_2 exactly **once** and writes **about the same** amount of pages out for the partitions
  - the probing phase reads each partition **once**
  - note that this cost estimation ignores **memory bottlenecks**

Grace Hash Join

Exercise: execution time of $\bowtie_p^{\textit{hash-join}}$

If the access time is 10 ms per page, how long will the join

$$\textit{Reserves} \bowtie_{sid=sid}^{\textit{hash-join}} \textit{Sailors}$$

take, assuming there are no memory bottlenecks?

Grace Hash Join

- For the probing phase, partitions should fit into memory
  - to minimize partition size, number of partitions has to be maximized
  - with B buffers, $B - 1$ partitions can be created in the partitioning phase
- Memory requirements for Grace hash join
  - assuming equal distribution, each R partition has size $\frac{\|R\|}{B-1}$
  - size of (in-memory) hash table built during the probing phase for an R partition is $\frac{f \cdot \|R\|}{B-1}$, where $f > 1$ is a **fudge factor** that captures the (small) increase in size between the partition and a hash table for the partition
  - probing phase needs to keep one such in-memory hash table plus an input and an output buffer in memory, i.e., $B > \frac{f \cdot \|R\|}{B-1} + 2$
  - algorithm needs **approximately** $B > \sqrt{f \cdot \|R\|}$ buffers to perform well
- Some cases requires multiple passes (recursive partitioning)
  - if input table R has more than $(B-1)^2$ pages
  - if values of R are not uniformly distributed over the partitions

Hybrid Hash Join

- If more than $B > \sqrt{f \cdot \|R\|}$ memory is available, **hybrid hash join** achieves better performance
- Suppose that $B > f \cdot (\|R\|/k)$, for some integer k
  - if R is divided into k partitions of size $\|R\|/k$, an in-memory hash table can be created for each partition
  - k output buffers and one input buffer are needed to partition R (and S), which leaves $B - (k + 1)$ extra buffer pages
- Suppose $B - (k + 1) > f \cdot (\|R\|/k)$, i.e., there is enough space to hold an in-memory hash table for a partition of R
  - **partitioning of R**: build hash table for first partition of R
  - **partitioning of S**: probe hash table with tuples of first partition of S
  - after the partitioning phase, first partitions of R and S are already joined
- Saves I/O cost for writing and reading first partitions of R and S

Hybrid Hash Join

Example: cost of $\bowtie_p^{\textit{hybrid-hash-join}}$

Consider the example with 500 pages in *Reserves*, 1000 pages in *Sailors*, and 300 buffer pages.

We can split *Reserves* (and *Sailors*) into 2 partitions and build an in-memory hash table for first partition of *Reserves*

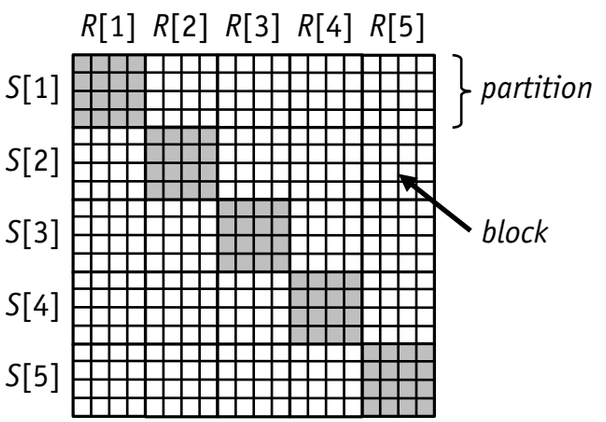
- **partitioning** of *Reserves*: scanning table and writing one partition (500 + 250 pages)
- **partitioning** of *Sailors*: scanning table and writing one partition (1000 + 500 pages)
- **probing phase**: read second partition of *Reserves* and *Sailors* (250 + 500 pages)
- **total cost**: 750 + 1500 + 750 = 3000 page I/O operations

Recall that the cost of the Grace hash join was $3 \cdot (1000 + 500) = 4500$ page I/O operations, but it only requires $\lceil \sqrt{f \cdot 500} \rceil \approx 25$ buffer pages.

Note that if there are $\mathrm{B} > f \cdot \|R\| + 2$ buffer pages available, i.e., the hash table for **all of** R fits into memory, the cost of the hybrid hash join is 500 + 1000 = 1500 page I/O operations.

Hash Join vs. Block Nested Loops Join

- Recall the variant of block nested loops joins that builds an in-memory hash table for the outer relation
- If a hash table for the entire smaller relation fits into memory, block nested loops join and hash join are the same
- If both relations are large relative to the available buffer size
  - block nested loops join needs several passes over one of the relations
  - hash join is a more effective application of hashing in this case



Hash Join vs. Sort-Merge Join

- $3 \cdot (N + M)$ page I/O operations
  - cost of **hash join**, if there are $B > \sqrt{\|M\|}$ buffer pages, where M is the **smaller** relation and we assume uniform partitioning
  - cost of **sort-merge join**, if there are $B > \sqrt{\|N\|}$ buffer pages, where N is the **larger** relation
- Choosing between hash join and sort-merge join
  - sort-merge join is less sensitive to **skew**, which would lead to partitions in hash join that are not uniformly sized
  - hash join costs less if the number of available buffer pages falls between $\sqrt{\|M\|}$ and $\sqrt{\|N\|}$ (the larger the difference in size between the two relations, the more important this factor becomes)
  - sort-merge join produces a **sorted result**

General Join Conditions

- Equalities over a combination of several attributes
  - **index nested loops join** can build a new index on combination of all attributes or use an existing index on combination of all attribute as well as of a subset of the attributes
  - **hash join** and **sort-merge join** partition over combination of attributes
- Inequalities
  - **index nested loops join** requires a B+ tree index
  - **hash join** and **sort-merge join** are not applicable
- Other join algorithms essentially remain unaffected

Summary of Join Algorithms

- No single join algorithm performs best under all circumstances
- Choice of algorithm affected by
  - size of relations joined
  - size of available buffer space
  - availability of indexes
  - form of join condition
  - selectivity of join predicate
  - available physical properties of inputs (e.g., sort order)
  - desirable physical properties of outputs (e.g., sort order)
- Performance differences between “good” and “bad” algorithm for any given join can be enormous
- Join algorithms have been subject to intensive research efforts

Selection

Simple selection query

```
SELECT *
FROM   Reserves R
WHERE  R.name = 'Joe'
```

$\sigma_{name='Joe'}(Reserves)$

- Selection can be implemented using **iteration** and **indexing**
- Choice of physical selection operator depends on
  - sortedness of the input file
  - presence or absence of indexes on the input file
- The following cases can be distinguished
  - no index, unsorted data
  - no index, sorted data
  - B+ tree index
  - hash index, equality selection

Selection: No Index, Unsorted Data

Selection

```
function σ(p, R_in, R_out)
  in ← openScan(R_in) ;
  out ← createFile(R_out) ;
  while (r ← nextRecord(in)) ≠ ⟨EOF⟩ do
    if p(r) then appendRecord(out, r) ;
  closeFile(out) ;
end
```

- Selection (σ_p) reads an input file R_{in} of records and writes those records that satisfy predicate p into the output file R_{out}
- Remarks
  - special "record" ⟨**EOF**⟩ indicates the end of the input file
  - simple algorithm **does not require** input file to have special physical properties (algorithm is defined exclusively in terms of heap files)
  - in particular, predicate p may be **arbitrary**

Selection: No Index, Unsorted Data

Cost of $\sigma_p^{scan}(R_{in})$ using a sequential scan

access path	file scan (`openScan`) of R_{in}
prerequisites	none (p arbitrary, R_{in} may be a heap file)
I/O cost	$\underbrace{\|\|R_{in}\|\|}_{\text{input cost}} + \underbrace{sel(p) \cdot \|\|R_{in}\|\|}_{\text{output cost}}$

- Notation
  - $\|\|R_{in}\|\|$ denotes the **number of pages** in file R_{in}
  - $|R_{in}|$ denotes the **number of records** in file R_{in}
  - if b records fit on one page, we have $\|\|R_{in}\|\| = \lceil |R_{in}|/b \rceil$
  - $sel(p)$ denotes the **selectivity** (or **reduction factor**) of predicate p
- Note that in a fully **pipelined** plan, selection can be applied on-the-fly without incurring any cost!

Selectivity

Definition

The **selectivity** (or **reduction factor**) or a predicate p, denoted by $sel(p)$, is the faction of records in a relation R that satisfy the predicate p.

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

Examples

What can you say about the following selectivities?

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

Selection: No Index, Sorted Data

- If input file R_{in} is **sorted** with respect to a sort key k
  - **binary search** could be used to find the first record
  - to find more hits, scan the **sorted** file
- Sort key k must **match** selection predicate p

When does a predicate match a (sort) key?

Assume R_{in} is sorted (or indexed) on attribute A in ascending order. Which of the selections below can benefit from the sortedness of (or index on) R_{in}?

1. $\sigma_{A=27}(R_{in})$
2. $\sigma_{A>27}(R_{in})$
3. $\sigma_{A>27 \wedge A<100}(R_{in})$
4. $\sigma_{A>27 \vee A>100}(R_{in})$
5. $\sigma_{A>39 \wedge A<27}(R_{in})$
6. $\sigma_{A>27 \wedge B=10}(R_{in})$
7. $\sigma_{A>27 \vee B=10}(R_{in})$

Selection: No Index, Sorted Data

Cost of $\sigma_p(R_{in})$ using binary search

access path	binary search, then sorted file scan of R_{in}
prerequisites	R_{in} sorted on sort key k that matches p
I/O cost	$\underbrace{\log_2 \|\|R_{in}\|\|}_{\text{input cost}} + \underbrace{sel(p) \cdot \|\|R_{in}\|\|}_{\text{sorted scan}} + \underbrace{sel(p) \cdot \|\|R_{in}\|\|}_{\text{output cost}}$

- Remarks
  - **disjunctive predicates** (e.g., A < 27 ∨ A > 100) are examined later
  - even if input is sorted, binary search **cannot be used** without incurring I/O cost in a fully pipelined plan!

The Real World

It is unlikely that a relation will be kept sorted if the DBMS supports variant ❶ for index entries. Most systems therefore do not implement binary search for value lookups.

Selection: B+ Tree Index

- The cost of using a B+ tree index on R_{in} whose sort key **matches** the selection predicate p to evaluate $\sigma_p(R_{in})$ depends on
  - number of qualifying tuples
  - whether the index is clustered or unclustered

Implementing $\sigma_p(R_{in})$ using a B+ tree

- Descend the B+ tree to retrieve the first index entry that satisfies p
- If the index is **clustered**, access that record on its page in R_{in} and continue to scan inside R_{in}
- If the index is **unclustered** and $sel(p)$ indicates a **large number of qualifying records**, it pays off to
  1. read all matching index entries $k^* = \langle k, rid \rangle$ in the sequence set
  2. sort those entries on the *rid* field
  3. access the pages of R_{in} in sorted *rid* order

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

Selection: B+ Tree Index

Cost of $\sigma_p(R_{in})$ using a clustered B+ tree index

access path	access of B+ tree on R_{in}, then sequence set scan
prerequisites	clustered B+ tree on R_{in} with key k that matches p
I/O cost	$\underbrace{\approx 3}_{\text{B+ tree access}} + \underbrace{sel(p) \cdot \|\|R_{in}\|\|}_{\text{sorted scan}} + \underbrace{sel(p) \cdot \|\|R_{in}\|\|}_{\text{output cost}}$

IBM DB2

IBM DB2 uses the physical operator quadruple **`IXSCAN`**/**`SORT`**/**`RIDSCN`**/**`FETCH`** to implement $\sigma_p(R_{in})$ using a B+ tree index.

Selection: Hash Index, Equality Selection

- A selection predicate p **matches a hash index** on $R_{in}.A$ only if it contains a term of the form $A = \mathrm{c}$, where c is a constant
- Application of hash function $h(\mathrm{c})$ returns the address of the bucket containing qualifying records
  - additional cost may incur due to presence of **overflow chains**
  - **selectivity** $sel(p)$ is likely to be close to 0 for equality predicates

Cost of $\sigma_p(R_{in})$ using a hash index

access path	hash index on R_{in}
prerequisites	R_{in} hashed on key k, p has a term $k = \mathrm{c}$
I/O cost	$\underbrace{\approx 1.2}_{\text{B+ tree access}} + \underbrace{sel(p) \cdot \|\|R_{in}\|\|}_{\text{output cost}}$

General Selection Conditions

- Selection operations with simple predicates like $\sigma_{A\,\theta\,c}(R_{in})$ are only a special case
  - in practice, selection operators need to support **complex predicates**
  - complex predicates use **Boolean connectives** $\wedge$ (**AND**) and $\vee$ (**OR**) to combine **simple comparisons** of the form $A\ \theta\ c$, where $\theta \in \{=, <, >, \leq, \geq\}$
- So far, the **conjunctive normal form (CNF)** has been defined as a number of **conjuncts** of the form $A\ \theta\ c$, connected by $\wedge$ (**AND**)

$$\underbrace{A_1\ \theta_1\ c_1}_{\textit{conjunct}} \wedge A_2\ \theta_2\ c_2 \wedge \ldots \wedge A_n\ \theta_n\ c_n$$

- Recall that a conjunctive predicate p **matches** a (multi-attribute)
  - hash index, if p **covers** the key
  - B+ tree index, if p is a **prefix** of the key

General Selection Conditions

- In general, each conjunct consist of one or more **terms** of the form $A\ \theta\ c$ that are connected by $\vee$ (**OR**)
- Conjuncts that contain $\vee$ (**OR**) are said to be **disjunctive** or to **contain disjunction**

Exercise: converting selection predicates

The selection predicate

$$(day < 12/15/2013 \wedge rname = \text{'Joe'}) \vee bid = 5 \vee sid = 3$$

is **not** in conjunctive normal form. Actually, it is in **disjunctive normal form (DNF)**. Can you convert it from DNF to CNF?

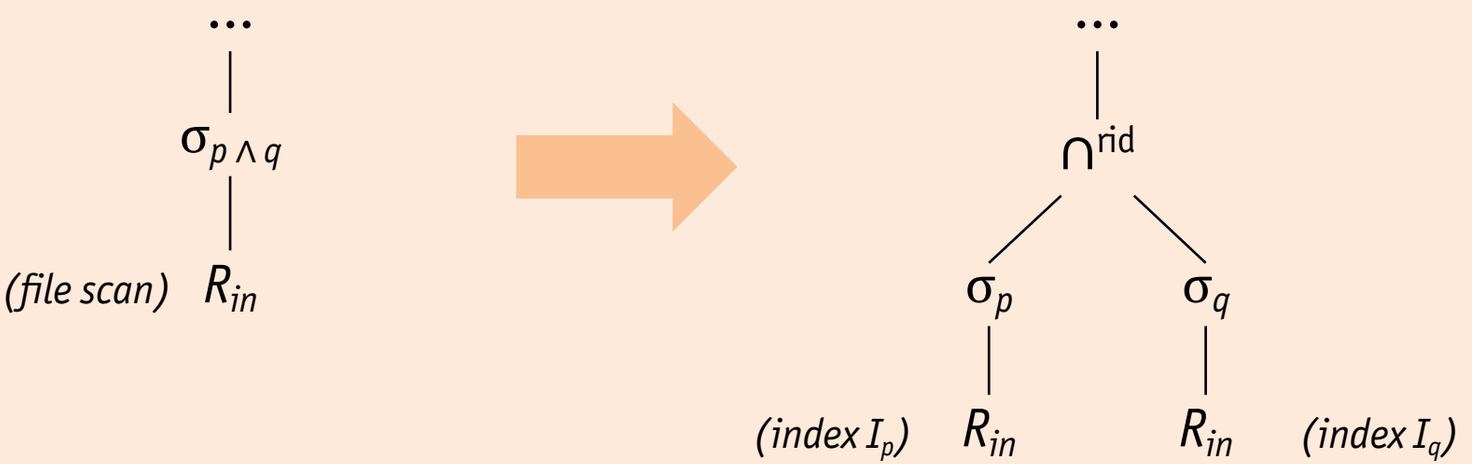
Selections without Disjunction

- If the selection predicate is a **conjunction of terms**, i.e., does not contain disjunction, there are three evaluation options
  - **single file scan**
  - **single index** that matches a subset of (primary) conjuncts and apply all non-primary conjuncts to each retrieved tuple (on-the-fly)
  - **multiple indexes** that each match a subset of conjuncts
- Recall that we have already discussed the first two options and will now focus on the third

Selection without Disjunction

Intersecting *rid* sets

The conjunctive predicate in $\sigma_{p \wedge q}(R_{in})$ does **not** match a single index, but both conjuncts p and q match indexes I_p and I_q, respectively. Assume that I_p and I_q contain index entries using variant ❷ or ❸. A typical query optimizer might decide to transform the query as follows.



Here, $\cap^{\text{rid}}$ denotes a **set intersection operator defined by** *rid* **equality**.

Selection without Disjunction

The Real World

- **IBM DB2**
  - physical operator **`IXAND`** does intersection of *rids* sets from indexes
  - implemented using Bloom filters
- **Oracle**
  - uses several techniques to do *rid* set intersection
  - bitwise and of bitmaps
  - hash join of indexes
- **Microsoft SQL Server**
  - implements *rid* set intersection through index joins

Selections with Disjunction

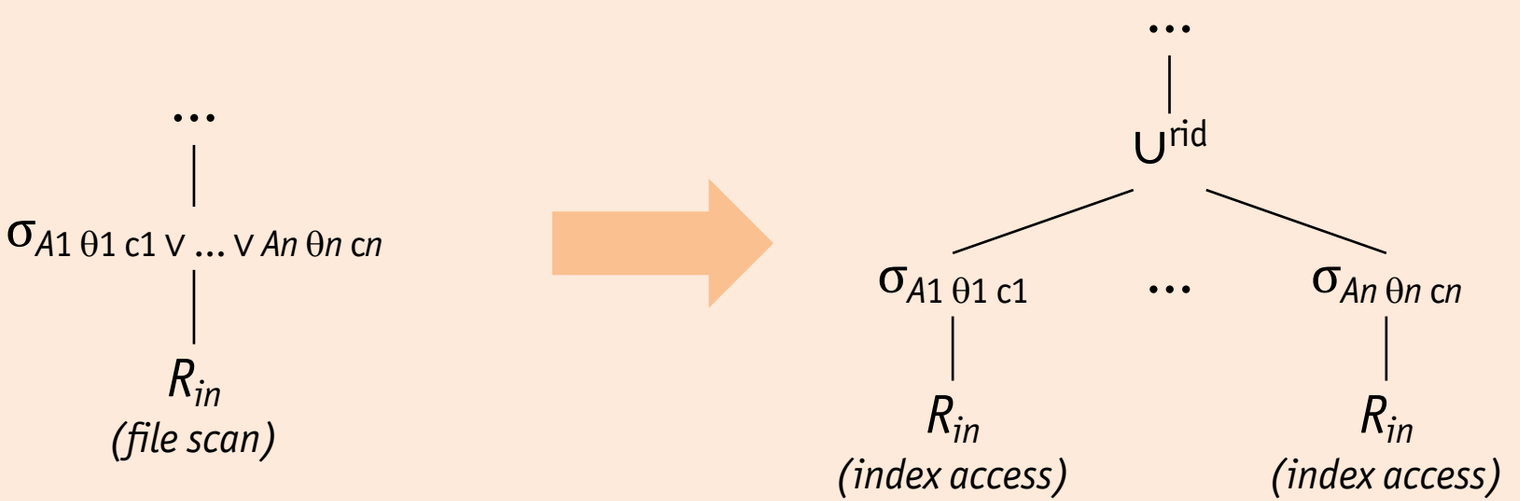
- Choosing a reasonable execution plan for **disjunctive selection predicates** of the following general form is **much** harder

$$A_1\ \theta_1\ c_1 \vee A_2\ \theta_2\ c_2 \vee \ldots \vee A_n\ \theta_n\ c_n$$

  - if only **one single** term does **not** match any index, evaluation is forced to use a naïve file scan
  - if **all** terms are supported by indexes, *rid*-**based set union** $\cup^{\text{rid}}$ can be exploited

Selections with Disjunction

Unioning *rid* sets



Exercise: selectivity of disjunctive predicates

What can you say about the selectivity of the disjunctive predicate $p \vee q$, $sel(p \vee q)$?

Bypass Selections

- Parts of a selection predicate may be **expensive** to check or be **very unselective**
  - so far, we assumed that checking a predicate was cheap (low CPU cost)
  - it is a good strategy to evaluate cheap and selective predicates first
- Boolean laws used for this optimization include

$$\text{true} \vee P \equiv \text{true} \qquad \text{(evaluating } P \text{ is not necessary)}$$

$$\text{false} \vee P \equiv P \qquad \text{(only evaluate P now)}$$

Example

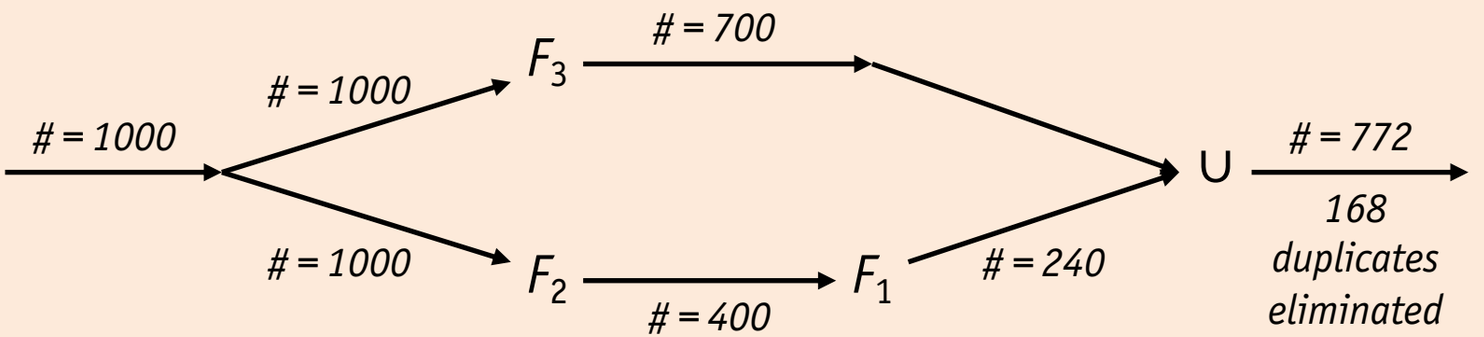
$Q := \sigma_{(F1 \wedge F2) \vee F3}(R)$, where the selectivities and cost of each part of the selection condition are given in the table on the right.

formula	selectivity	cost
F_1	$s_1 = 0.6$	$C_1 = 18$
F_2	$s_2 = 0.4$	$C_2 = 3$
F_3	$s_3 = 0.7$	$C_3 = 40$

Bypass Selections

- First evaluation alternative
  - convert selection condition to **disjunctive normal form** (example is already in DNF)
  - push each tuple from input through each disjunct **in parallel**
  - **collect** matching tuples from each disjunct (duplicate elimination)

Example



Mean cost per tuple (ignoring duplicate elimination): $C_3 + C_2 + s_2 \cdot C_1 = 50.2$

Bypass Selections

- Second evaluation alternative
  - convert selection condition to **conjunctive normal form**

$$\mathrm{CNF}[(F_1 \wedge F_2) \vee F_3] = (F_1 \vee F_3) \wedge (F_2 \vee F_3)$$

  - push each tuple from input through each conjunct **sequentially**
  - matching tuples "survive" conjuncts (**no** duplicate elimination)

Example

$$\xrightarrow[\#\,=\,1000]{} F_2 \vee F_3 \xrightarrow[\#\,=\,820]{} F_1 \vee F_3 \xrightarrow[\#\,=\,772]{}$$

Mean cost per tuple:

$$C_2 + (1 - s_2) \cdot C_3 + (s_2 + (1 - s_2) \cdot s_3) \cdot C_1 + (1 - s_1) \cdot (s_2 + (1 - s_2) \cdot s_3) \cdot C_3 = 54.88$$

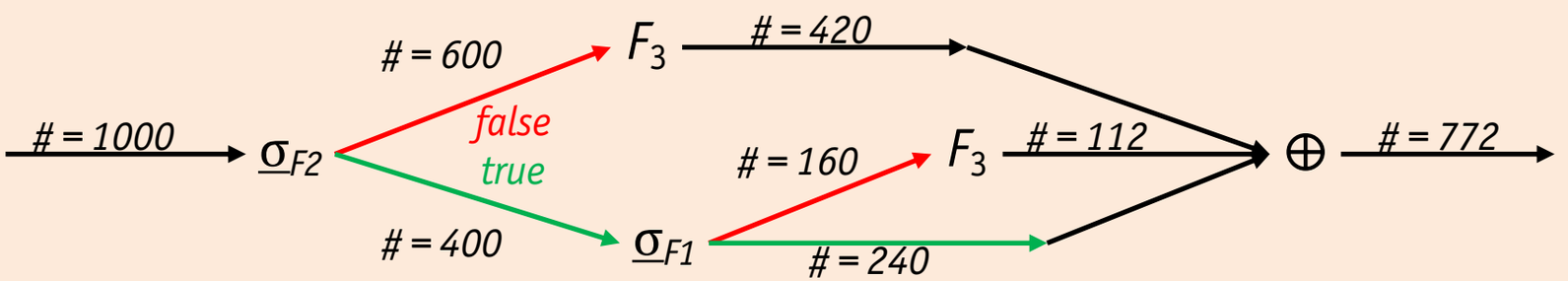
Problem: $F3$ is evaluated multiple times, result could be cached

Mean cost per tuple with caching: $C_2 + C_3 + s_2 \cdot (1 - s_3) \cdot C_1 = 45.16$

Bypass Selections

- **Goal:** eliminate tuples early, avoid duplicate
- Bypass selection operator $\underline{\sigma}_F$
  - produces two disjoint outputs: true and false results
  - bypass plans are derived from the conjunctive normal form
  - Boolean factors and disjuncts in factors are sored by cost

Example



Mean cost per tuple ($\oplus$ disjoint union): $C_2 + (1 - s_2) \cdot C_3 + s_2 \cdot (C_1 + (1 - s_1) \cdot C_3) = 40.6$

Note that many variations are possible, e.g., for tuning in parallel environments

Selection with Disjunction

The Real World

Most systems do not handle selection conditions with disjunction efficiently and concentrate on optimizing selections without disjunction.

- **Oracle**
  - convert the query into a **union query without `OR`**
  - if the conditions involve the same attribute (e.g., $sal < 5 \vee sal > 30$), use a **nested query with an `IN` list** and an index on the attribute to retrieve tuples matching a value in the list
  - use **bitmap operations**, e.g., evaluate $sal < 5 \vee sal > 30$ by generating bitmaps for the value 5 and 30 and then bitwise or the bitmaps to find tuples that satisfy one of the conditions
  - simply apply the disjunctive condition as a **filter** on the set of retrieved tuples
- **Microsoft SQL Server**
  - considers the use of **union queries** and **bitmaps** to deal with disjunctive conditions

Projection

Simple projection query

```
SELECT DISTINCT R.sid, R.bid
FROM   Reserves R
```

$\pi_{\{sid,bid\}}(Reserves)$

- **Projection** (π_l) modifies each record in its input file
  1. **removes** unwanted attributes, i.e., those not specified in attribute list *l*
  2. **eliminate** any duplicate tuples produced (SQL keyword **`DISTINCT`**)
- In general, the size of the resulting file will therefore only be a fraction of the original input file size
- Projection is implemented using **iteration** and **partitioning**
- Projection
  - **without** duplicate elimination can be fully **pipelined**
  - **with** duplicate elimination has to **materialize** intermediate results

Projection

Example

$\pi_{\{sid,bid\}}$

sid	bid	day	rname
1	1	07/04/2013	George
1	2	07/05/2013	Thomas
1	1	08/01/2013	George
1	2	08/02/2013	Thomas
2	1	07/05/2013	Joe
2	1	08/02/2013	Joe

=
①

sid	bid
1	1
1	2
1	1
1	2
2	1
2	1

=
②

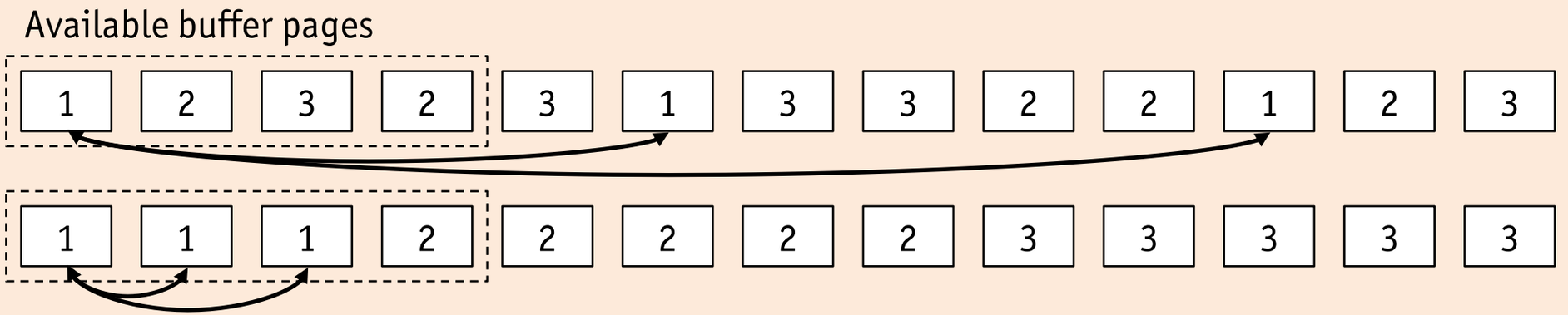
sid	bid
1	1
1	2
2	1

- While **step** ① calls for a rather straightforward file scan (cannot really use indexes), it is **step** ② which makes projection costly
- Partitioning is used to implement duplicate elimination
  - sorting
  - hashing

Partitioning

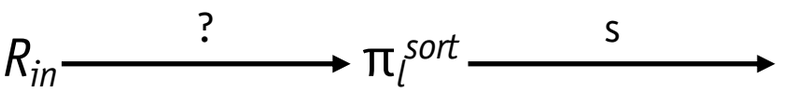
- Duplicate elimination
  - compare each tuple to **all other tuples** to check whether it is a duplicate
  - in general, it is **not possible** to fit all other tuples in memory
- Partitioning
  - given *B* buffer pages, group "similar" tuples into partitions
  - load partitions into memory one after another
  - only compare a tuple to other tuples in that partition

Illustration



Projection Based on Sorting

- Records with all fields equal will be adjacent after sorting
  1. scan R_{in} and output set of tuples that contain **only** the desired attributes
  2. sort this set of tuples using **all** of its attributes as sort key
  3. scan the sorted result, comparing adjacent tuples, and discard duplicates
- A benefit of sort-based projection is that the π_l^{sort} operator outputs a sorted result



Exercise: sort ordering

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

Projection Based on Sorting

Projection based on sorting

```
function π^sort (l, R_in, R_out)
   in ← openScan(R_in) ;
   out ← createFile(R_tmp) ;
   while (r ← nextRecord(in)) ≠ ⟨EOF⟩ do
      r′ ← r with any field not listed in l cut off ;
      appendRecord(out, r) ;
   closeFile(out) ;
   external-merge-sort(R_tmp, □R_tmp□, θ) ;
   in ← openScan("run␣*␣0") ;
   out ← createFile(R_out) ;
   lastr ← ⟨⟩ ;
   while (r ← nextRecord(in)) ≠ ⟨EOF⟩ do
      if r ≠ lastr then
         appendRecord(out, r) ;
         lastr ← r ;
   closeFile(out) ;
end
```

Projection Based on Sorting

- Approach can be improved by modifying the sorting algorithm to do projection with duplicate elimination

Marriage of sorting and projection with duplicate elimination

Step ① (**projection**) and Step ② (**duplicate elimination**) can be **integrated** into the passes performed by the external merge sort algorithm.

Pass 0

1. read *B* pages at a time, **projecting unwanted attributes out**
2. use in-memory sort to sort the records of these *B* pages
3. write a sorted run of *B* internally sorted pages out to disk

Pass 1, ..., *n*

1. select *B* – 1 runs from previous pass, read a page from each run
2. perform a (*B* – 1)-way merge, **eliminating duplicates**
3. use the *B*-th page as an output buffer

Projection Based on Sorting

Cost of $\pi_l^{sort}(R_{in})$ using sorting for duplicate elimination

access path	file scan (`openScan`) of R_{in}
prerequisites	none (B available buffer pages)
I/O cost	$\underbrace{\|R_{in}\| + \|R_{tmp}\|}_{\text{projection}} + \underbrace{2 \cdot \|R_{tmp}\| \cdot \left(\left\lceil \log_{B-1} \left\lceil {}^{\|R_{tmp}\|}/_{B} \right\rceil \right\rceil\right)}_{\text{duplicate elimination}}$

- Remarks
  - depending on the fraction of duplicate tuples, $\|R_{tmp}\|$ will be smaller in the duplicate elimination phase

Projection Based on Hashing

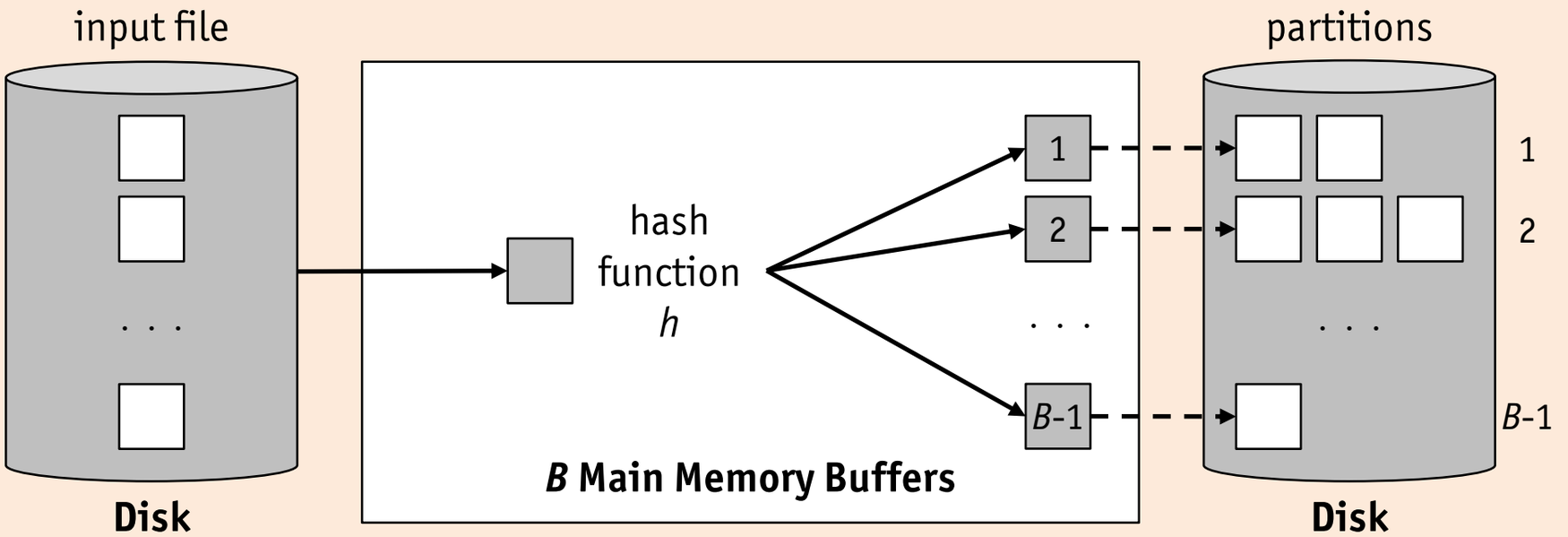
- Hash-based projection $\pi_l^{hash}(R_{in})$
  - worth considering if there is a fairly large number of buffer pages (say, B) relative to the total number of pages of R_{in}
  - two phases: **partitioning** and **duplicate elimination**

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_{in} page by page: for each record r, **project out** the attributes not listed in list l.
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 $h_1(r)$.
4. If the hash bucket is **full**, write it to disk, i.e., the overflow chain of a bucket resides on disk

Projection Based on Hashing

Partitioning phase



- Two **identical** records r, r' will be mapped to the **same partition**
  - $h_1(r) = h_1(r') \Leftarrow r = r'$
  - duplicate elimination is an intra-partition problem only
  - **but**, due to hash collisions, the records in a partition are not guaranteed to be all equal, i.e., $h_1(r) = h_1(r') \not\Leftrightarrow r = r'$

Projection Based on Hashing

Duplicate elimination phase

1. For each partition, read each partition page by page (possibly in parallel), using the same buffer layout as before.
2. To each record, apply a hash function h_2 ($h_2 \neq h_1$) to all record fields.
3. Only if two records collide with respect to h_2, check if $r = r'$ and, 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.

Exercise: large partitions

Projection based on hashing only works efficiently, if duplicate elimination can be **performed in the buffer** (main memory). What can be done if the partition size exceeds the buffer size?

Projection Based on Hashing

Cost of $\pi_l^{hash}(R_{in})$ using hashing for duplicate elimination

access path	file scan (`openScan`) of R_{in}
prerequisites	none (B available buffer pages)
I/O cost	$\underbrace{\|R_{in}\| + \|R_{tmp}\|}_{\text{projection}} + \underbrace{\|R_{tmp}\| + \|R_{tmp}\|}_{\text{duplicate elimination}}$

- Remarks
  - depending on the fraction of duplicate tuples, $\|R_{tmp}\|$ will be smaller in the duplicate elimination phase

Sorting Versus Hashing

- Sorting is the standard approach for duplicate elimination
  - superior to hashing if there are many duplicates or if the distribution of (hash) values is very non-uniform
  - sorting is required for a number of other reasons and therefore already implemented in most systems
- If there are $B > \sqrt{\|R_{tmp}\|}$ buffer pages, both approaches have the same cost
  - sorting takes two passes: $\|R_{in}\| + \|R_{tmp}\| + \|R_{tmp}\| + \|R_{tmp}\|$
  - hashing is the same: $\|R_{in}\| + \|R_{tmp}\| + \|R_{tmp}\| + \|R_{tmp}\|$
- Choice of projection algorithm therefore depends on **CPU cost**, desirability of **sorted order**, **skew** in value distribution, etc.

Using Indexes for Projection

- Index key contains **all attributes** retained in the projection
  - use an index-only plan to retrieve all values from index without accessing actual data records
  - apply hashing or sorting to eliminate duplicates from this (much smaller) set of pages
- Index key contains projected attributes as a **prefix** and is **sorted**
  - use an index only plan both to retrieve projected attribute values **and** to eliminate duplicates

The Real World

- **IBM DB2** and **Oracle**
  - sorting is used for duplicate elimination
- **Microsoft SQL Server**
  - implements both hash-based and sort-based algorithms for duplicate elimination

Set Operations

- **Intersection** and **cross-product** are implemented as special cases of join
  - **equality on all attributes** as join condition computes intersection
  - **true** as join condition computes the cross-product
- **Union** and **difference** can be thought of as a selection with a complex selection condition
  - main point to address in **union** implementation is duplicate elimination
  - **difference** can be implemented using a variation of duplicate elimination
  - again, both **sorting** and **hashing** can be used for duplicate elimination

Set Operations

Sorting for union and difference

Implementation of $R \cup S$

1. sort both R and S using the combination of **all** fields
2. scan the sorted R and S in parallel and merge them, eliminating duplicates

As with projection, the implementation of difference can be integrated with the external sort operator.

Implementation of $R - S$ is similar. During the merging pass tuples of R are only written to the result after checking that they **do not appear** in S.

Set Operations

Hashing for union and difference

Implementation of $R \cup S$

1. partition both R and S using a hash function $h(\cdot)$ over the combination of **all** fields
2. process each partition i as follows
   - build an in-memory hash table, using hash function $h2(\cdot) \neq h(\cdot)$, for $S[i]$
   - scan $R[i]$; for each tuple probe the hash table for $S[i]$; if the tuple is in the hash table, discard it; otherwise, add it to the table
   - write out the hash table; clear it to prepare for the next partition

Implementation of $R - S$ is similar, but processes the partition differently. After building an in-memory hash table for $S[i]$, $R[i]$ is scanned and $S[i]$ is probed for every tuple in $R[i]$. If the tuple is **not in the table**, it is written to the result.

Aggregate Operations

- Several techniques exist to implement the aggregate operations (**`SUM`**, **`AVG`**, **`COUNT`**, **`MIN`**, and **`MAX`**) supported since SQL-92
- **Basic algorithm**
  - scan whole relation (possibly on-the-fly) and maintain some **running information** during that scan
  - upon completion of the scan, compute aggregate value from running information

Aggregate	Running Information
`SUM`	**Total** of values read
`AVG`	⟨**Total**, **Count**⟩ of values read
`COUNT`	**Count** of values read
`MIN`	**Smallest** value read
`MAX`	**Largest** value read

Aggregate Operations

- For **aggregation combined with grouping**, there are two evaluation algorithms based on sorting or hashing, respectively
- **Sorting approach** (cost equal to I/O cost of sorting)
  - sort relation on grouping attribute(s)
  - scan again to compute the result of the aggregate operation (using the basic algorithm) for each group
  - as a refinement, aggregation can be done as part of the sorting step
- **Hashing approach** (if hash table fits into memory, cost is $\|R\|$)
  - build a (in-memory) hash table on the grouping attribute(s) with entries ⟨grouping value, running information⟩
  - for each tuple of the relation, probe hash table to find entry of the group to which the tuple belongs and update running information
  - when hash table is complete, its entries for grouping value can be used to compute the corresponding result tuples in a straightforward way

Aggregate Operations

- Using an **index** to select a subset of tuples is not applicable
- Under certain conditions, index entries instead of data records can be used to evaluate aggregate operations efficiently
  - if index search key includes **all attributes** needed for the aggregation query, fetching data records can be avoided by working with index entries
  - if **`GROUP BY`** clause attribute list forms a prefix of index search key and index is a tree index, sorting can be avoided by retrieving index entries (and data records, if necessary) in order required by grouping operation
- An index may support one or both of these techniques
- Both techniques are examples of **index-only** plans

Impact of Buffering

- Effective use of the buffer pool is very important
  - crucial for efficient implementation of a relation query engine
  - several operators use size of available buffer space as parameter
- Points to note
  1. operators that execute concurrently have to **share** the buffer pool
  2. if tuples are accessed using an index, the likelihood of finding a page in the buffer pool becomes (unpredictably) dependent on its **size** and **replacement policy**
  3. if tuple are accessed using an unclustered index, the buffer pool **fills up quickly** as each retrieved tuple is likely to require a new page to be brought into the buffer pool
  4. if an operation has a **repeated pattern of page accesses**, a clever replacement policy and/or sufficient number of buffer pages can speed up an operation significantly (*see next slide*)

Impact of Buffering

Examples of patterns of repeated access

Simple Nested Loops Join: *"for each tuple of the outer relation, scan all pages of the inner relation"*

If there is enough buffer space to hold entire inner relation, the replacement policy is irrelevant, otherwise it is critical

- LRU will **never find** a requested page in the buffer pool ("sequential flooding")
- MRU achieves best buffer utilization: the first $B - 2$ pages will **always stay** in the buffer pool

Block Nested Loops Join: *"for each block of the outer relation, scan all pages of the inner relation"*

Since only one unpinned page is available for the scan of the inner relation, the replacement policy make no difference

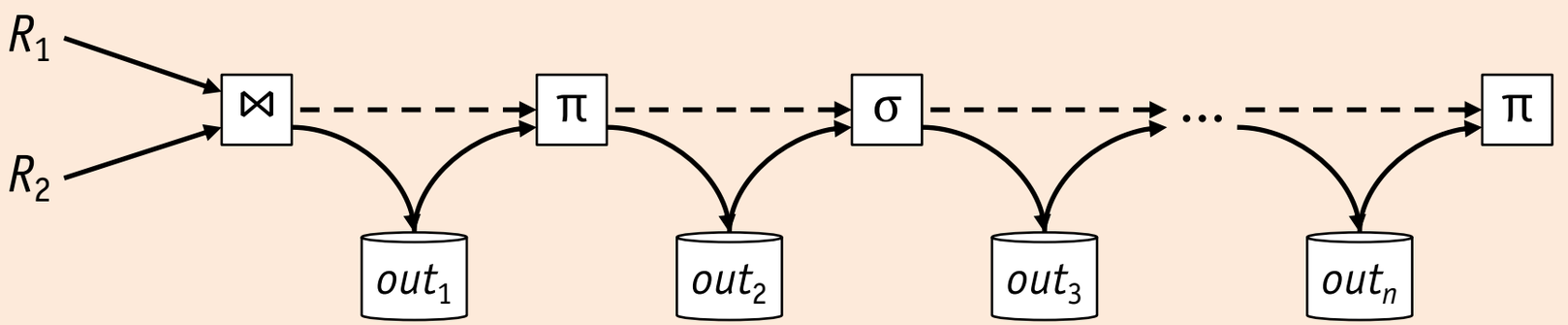
Index Nested Loops Join: *"for each tuple of the outer relation, probe the index to find matching tuples of the inner relation"*

For duplicate values in the join attributes of the outer relation, there is a repeated access pattern on the inner relation, which can be maximized by sorting the outer relation on the join attributes

Materialized vs. Pipelined Evaluation

- Pseudo-code implementation of all relational operators discussed so far assumes **materialized evaluation**
  - operators communicate through secondary storage by **reading** (R_{in}) and **writing** (R_{out}) files, which causes **a lot of disk I/O**
  - operator **cannot start** processing until all its inputs are fully materialized
  - all operators are executed **in sequence**, first result tuple is only available after the last operator has executed

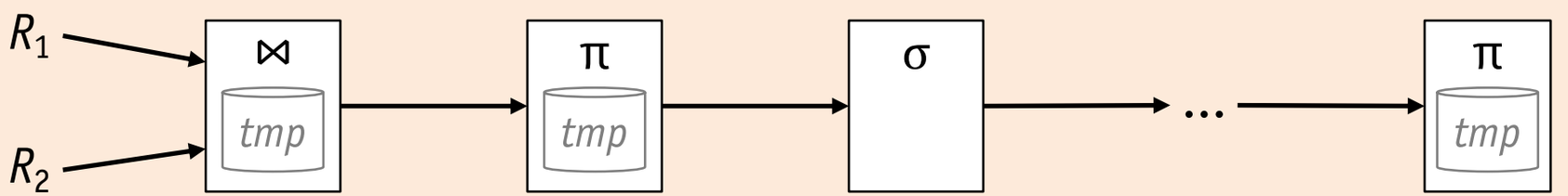
Materialized evaluation



Materialized vs. Pipelined Evaluation

- Alternatively, **pipelined evaluation** can be used to avoid writing temporary files to disk whenever possible
- Each operator passes its results **directly** to the next operator
  - as soon as results are available, propagate output **immediately**
  - as soon as input data is available, start computing **as early as possible**
  - all operators can execute in **parallel**

Pipelined evaluation



Pipelined Evaluation

- To support pipelined evaluation, the current implementation of the relational operators discussed so far has to be adapted
- The **granularity** at which data is passed is an important design decision that may influence performance
  - **communication:** fine granularity reduces response time as results are produced as early as possible
  - **scheduling/control:** coarse granularity may improve the effectiveness of (instruction) caches and buffer pool as there are fewer context switches

The Real World

Actual systems typically operate at one **tuple at a time**.

Pipelined Evaluation

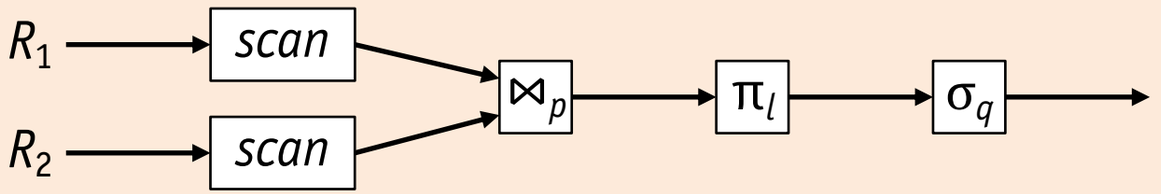
- Pipelined evaluation is typically implemented based on the **open-next-close interface** or **Volcano iterator model**
- Each relational operator implements the functions

`open()`	**initialize** the operator's internal state
`next()`	produce and return the **next result tuple** or ⟨`EOF`⟩
`close()`	**free** any operator-internal resources, typically after all tuples have been processed

- All **state** is kept within each operator instance
  1. upon call to `next()`, operator produces a tuple, updates its internal state and then **pauses**
  2. upon subsequent call to `next()`, operator uses its internal state to **resume** and produce another tuple

Pipelined Evaluation

Example



- Query plans like the one shown above are evaluated by the query processor as follows
  1. reset query plan by calling **`open()`** on root operator, i.e., σ_q.**`open()`**
  2. operators forward **`open()`** call through the query plan
  3. control returns to query processor
  4. next (first) tuple is produced by calling **`next()`** on the root operator, i.e., σ_q.**`next()`**
  5. operators forward **`next()`** call through the query plan as needed
  6. as soon as the next (first) record is produced, control returns to query processor again

Pipelined Evaluation

Volcano-style query evaluator

```
function eval(q)
  q.open();
  r ← q.next();
  while r ≠ ⟨EOF⟩ do
    emit(r);                    (deliver record r, i.e., print or send to DB client)
    r ← q.next();
  q.close();                    (deallocate all resources)
end
```

- Iterator interface above implements a **demand-driven** query processing infrastructure
  - **consumers** (later operators) request input on-demand from **producers** (earlier operators) whenever they are ready to process it
  - demand-driven pipelining **minimizes** resource requirements and "wasted" effort in case a user/client does not request the whole result

Pipelined Evaluation

Volcano-style implementation of $\sigma_p(R_{in})$

```
function open()
  R_in.open();
end
```

```
function next()
  while (r ← R_in.next()) ≠ ⟨EOF⟩ do
    if p(r) then return r;
  return ⟨EOF⟩;
end
```

```
function close()
  R_in.close();
end
```

- Remarks
  - R_{in} is the input operator (sub-plan root) of $\sigma_p(R_{in})$
  - $p(\cdot)$ is the predicate of $\sigma_p(R_{in})$

Pipelined Evaluation

Exercise: Volcano-style implementation of $\bowtie_p^{nl}(R_1, R_2)$

Pipelined Evaluation

- Pipelining **avoids materialization** and **reduces response time** because data is processed one tuple (chunk of data) at a time
- So-called **blocking operators** cannot be implemented in this way
  - entire input needs to be consumed before output can be produced
  - operator has to internally buffer (materialize) the data on disk

Exercise: examples of blocking operators

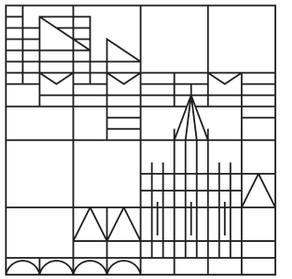
Which operators discussed so far do not permit pipelining without internal materialization?

Pipelined Evaluation

Iterator	open()	next()	close()	State
print	open input	call **next()** on input, format item on screen	close input	
scan	open file	read next item	close file	open file descriptor
select	open input	call **next()** on input, until an item qualified	close input	
hash join without overflow resolution	allocate hash directory, open left *build* input, build hash table calling **next()** on build input, close build input, open right *probe* input	call **next()** in *probe* input until a match is found	close *probe* input, deallocate hash directory	hash directory
merge join without duplicates	open both inputs	get **next()** item from input with smaller key until a match is found	close both inputs	
sort	open input, build all initial run files calling **next()** on input, close input, merge run files until only one step is left	determine next output item, read new item from the correct run file	destroy remaining run files	merge heap, open file descriptors for run files

Pipelined Evaluation

- An alternative query processing infrastructure to the demand-driven pipelining of the iterator model is **data-driven** pipelining
  - producer and consumer operators are connected by a **queue**
  - all operators execute **asynchronously** and in **parallel**
  - operators process all their input data available from incoming queue(s), produce results as fast as possible, and enqueue them to outgoing queue
- This model relies on queues for communication and scheduling
  - queues **buffer** data that is pipelined between operators
  - queues **suspend** operators using blocking enqueue/dequeue calls
- Data-driven pipelining is able **exploit more parallelism** than demand-driven pipelining
  - operators only need to wait, if input queue is empty or output queue is full
  - trades off higher resources requirements for more parallelism



TO BE CONTINUED...