Lecture

Database Implementation Techniques / Databases II

OvGU Magdeburg, WinSem 2009/2010

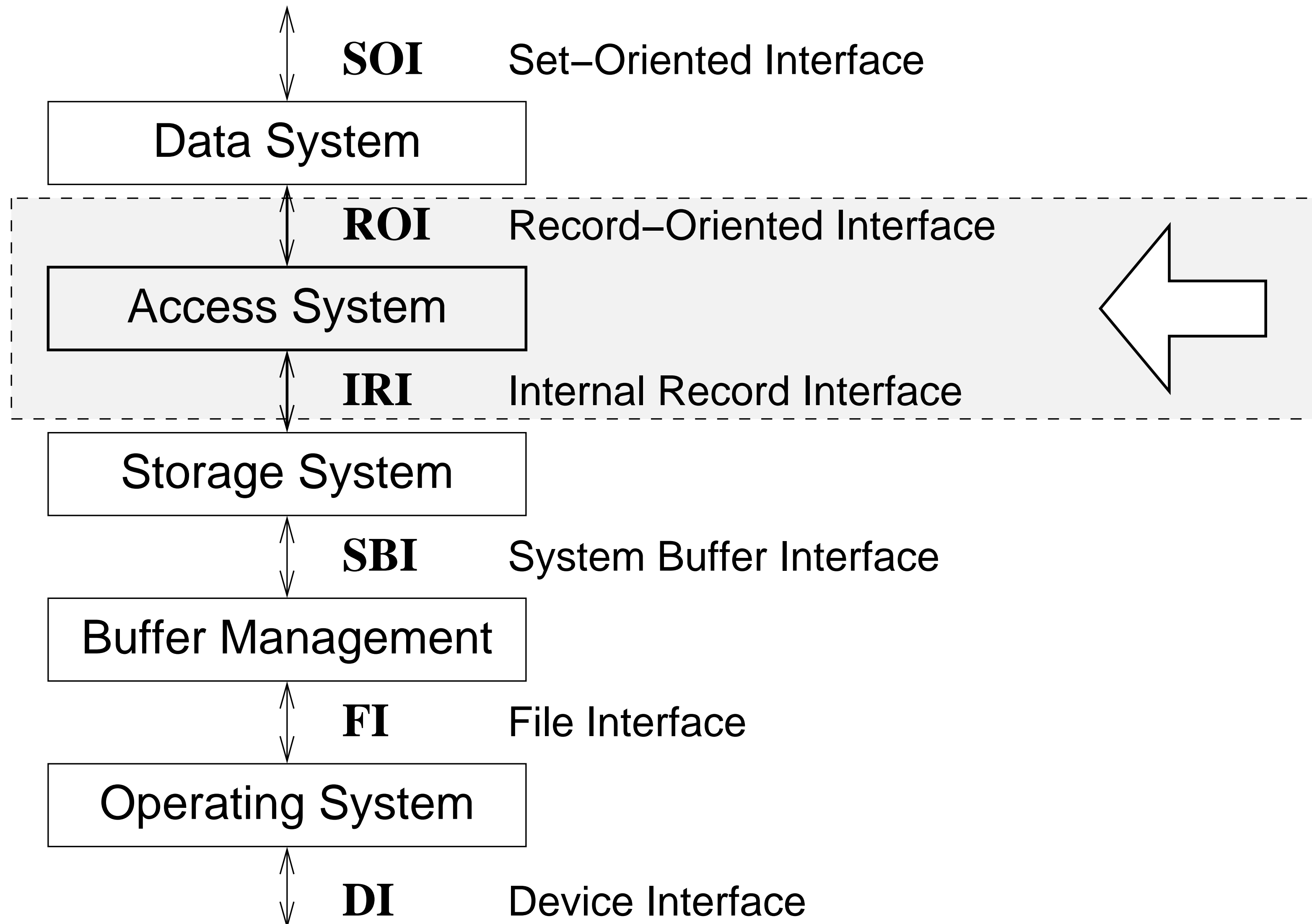
Sandro Schulze, Gunter Saake

{sanschul,saake}@iti.cs.uni-magdeburg.de.

6. Basic Algorithms for DB Operations

- Database parameter
- Complexity of basic algorithms
- Unary operations (Scan, Selection, Projection)
- Binary operations: Set operations
- Computation of Joins

Classification



Database Parameters

- Consideration of complexity ($O(n^2)$)
- Cost estimation/evaluation (concrete)
- Database parameters as basis
- Stored in the data dictionary of the DBS

Database Parameters (II)

- n_r: Number of tuples in relation r
- b_r: Number of blocks (pages), containing tuples from r
- s_r: (average) size of tuples from r
- f_r: *Blocking factor* (tuples from r per block)

$$f_r = \frac{bs}{s_r},$$

with bs block size

- tuples of one relation are packed tightly in blocks:

$$b_r = \left\lceil \frac{n_r}{f_r} \right\rceil$$

Database Parameters (III)

- $V(A, r)$: Number of different values for attribute A in relation r: $V(A, r) = |\pi_A(r)|$
- A is primary key: $V(A, r) = n_r$
- $SC(A, r)$: *selection cardinality*; average number of result tuples for $\sigma_{A=x}(r)$ with $x \in \pi_A(r)$
- Key attribute A: $SC(A, r) = 1$
- General:

$$SC(A, r) = \frac{n_r}{V(A, r)}$$

Further information: Branching factor of B-Tree indices, height of tree, number of leaf nodes

Complexity of Basic Algorithms

Assumptions

- Indices realized as B^+-Trees
- Dominant cost factor: Block access (on secondary storage)
- Access on secondary storage for intermediate relations as well
- Intermediate relation at first for each (basic) operation
- Intermediate relation (hopefully) to a large extent in the buffer
- Some operations (set operations) on set of addresses (TID lists)

Main Memory Algorithms

Important for throughput of the overall system, because they are frequently used

- *Comparison of tuples*
  (For duplicate detection, specify sorting order, . . .)
  realized iterative by comparison of single attributes, firstly attributes with high selectivity
- *TID access*
  TID within main memory: Usual approach for resolving indirect addresses

Access on Records

- *Relations*: internal identifier `RelID`
- *Indices*: internal identifier `IndexID`
  - *Primary index*, e.g., $I(\texttt{Persons(PANr)})$
    for $A = a$ at most one tuple per access is provided
  - *Secondary index*, e.g., $I(\texttt{Borrowing(PANr)})$
    Ex.: $\texttt{PANr} = 4711$ provides in general several tuples

Index access: Result usually as TID list(s)

Access on Records (II)

- **fetch-tuple**: direct access on tuple using TID value tuple is loaded to *tuple buffer*

$$\textbf{fetch-tuple}(\texttt{RelID}, \texttt{TID}) \rightarrow \texttt{tuple buffer}$$

- **fetch-TID**: Determine TID for (primary key) attribute value

$$\textbf{fetch-TID}(\texttt{IndexID}, \texttt{attribute value}) \rightarrow \texttt{TID}$$

- Further operation on relations and indices: *Scans*

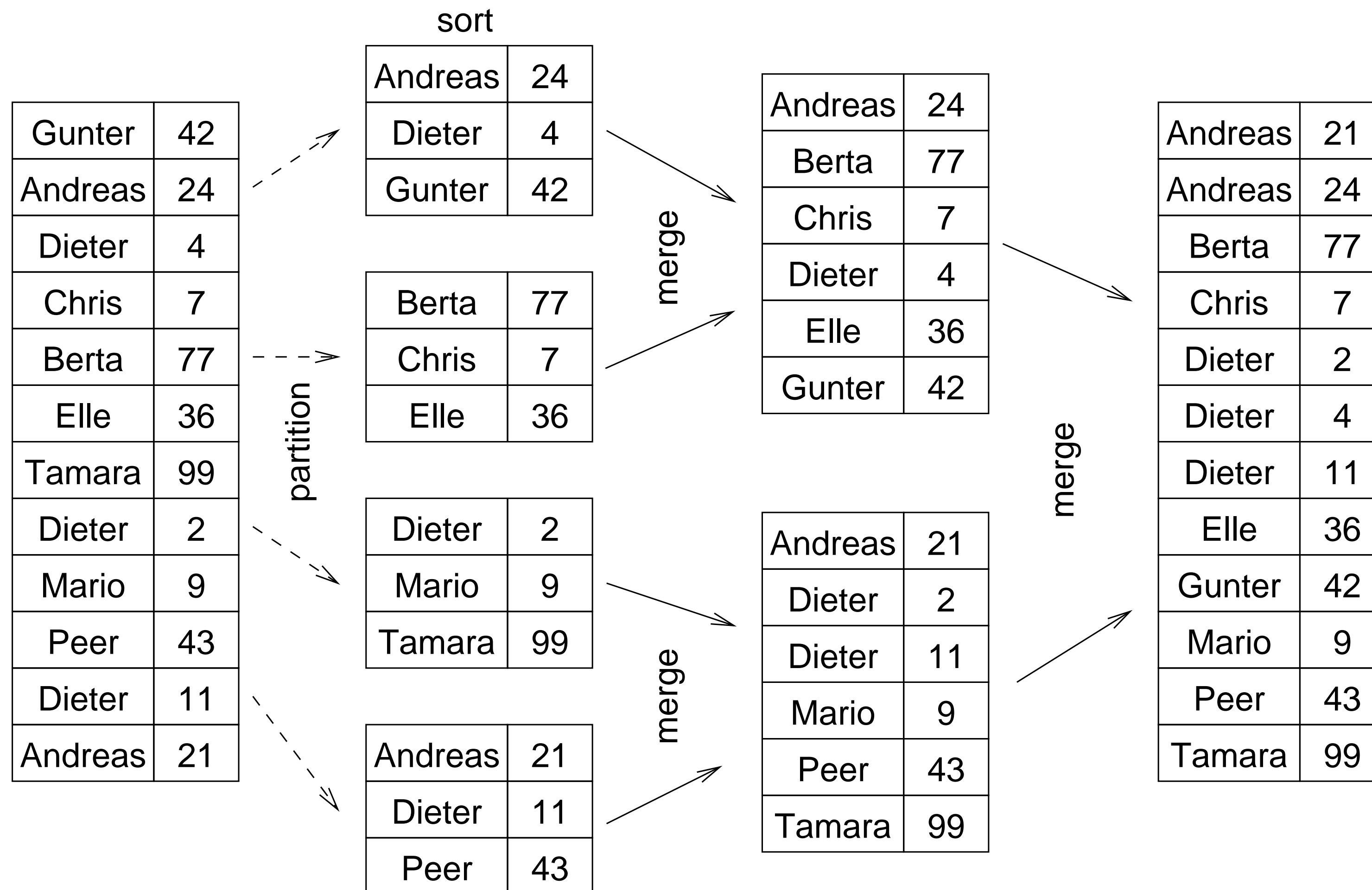
Example in SQL

```
select *
from CUSTOMER
where CName ='Brown'
```

- Query for equality on a key
- **put**: used here for displaying the result

```
currentTID :=
   fetch-TID(CUSTOMER-CName-Index, ‘Brown’);
currentBuffer:=
   fetch-tuple(CUSTOMER-RelationID, currentTID);
put(currentBuffer);
```

External Sorting Algorithms



External sorting by merging; Complexity: $O(n \log n)$ operations for ordering (during merge process)

Unary Operations

Scan traverses tuples of a relation

- *Full table scan* traverses all tuples of a relation in arbitrary order
  Costs: b_r
- *Index Scan* uses index to select tuples in sorting order
  Costs: Number of tuples plus height of index

Comparison

- Full table scan performs better by exploiting the blocking
- Index scan performs better in the case that only few data are needed (but worse for reading many tuples)

Operations on Scans

- Open full table scan
  open-rel-scan$(\texttt{RelationID}) \rightarrow \texttt{ScanID}$
  returns `ScanID` that can be used for identification for the following operation
- Initialize index scan
  open-index-scan$(\texttt{IndexID}, \texttt{Min}, \texttt{Max}) \rightarrow \texttt{ScanID}$
  returns `ScanID`; `Min` and `Max` determines range of range query
- **next-TID** provides next TID; scan cursor is increased
- **end-of-scan** returns **true**, if no TID is left for the scan
- **close-scan**

Example: Scan

```
select *
from Persons
where surname between 'Heuer' and
      'Jagellowsk'
```

Example: Full table scan

```
currentScanID := open-rel-scan(Persons-RelationID);
currentTID := next-TID(currentScanID);
while not end-of-scan(currentScanID) do
begin
   currentBuffer :=
      fetch-tuple(Persons-RelationID,currentTID);
   if currentBuffer.Surname >= 'Heuer'
      and currentBuffer.Surname <= 'Jagellowsk'
   then put (currentBuffer);
   endif;
   currentTID := next-TID(currentScanID);
end;
close (currentScanID);
```

Example: Index Scan

```
currentScanID :=
   open-index-scan(Persons-Surname-IndexID,
         'Heuer','Jagellowsk');
currentTID := next-TID(currentScanID);
while not end-of-scan(currentScanID) do
begin
   currentBuffer :=
      fetch-tuple(Persons-RelationID,currentTID);
   put(currentBuffer);
   currentTID := next-TID(currentScanID);
end;
close (currentScanID);
```

Selection

- *exact search*, *range selections*, complex, composed selection criteria
- Composite predicate φ composed from atomic predicates (exact search, range query) with **and**, **or**, **not**

Tuplewise approach

- Given $\sigma_\varphi(r)$
- Relation scan: For all $t \in r$ compute $\varphi(t)$
- Complexity $O(n_r)$, more precisely b_r

Selection: Conjunctive Normal Form

- Apply access path to complex predicates $\Rightarrow$ analyze and transform (suitable) φ
- For instance, transform φ to conjunctive normal form CNF; consists of *conjunctives*
- Select conjunctive (heuristically), which can be good analyzed by an index(e.g., $A = c$ in the case that index on A exists)
- Analyze selected conjunctive; for resulting TID list, all other parts of CNF are analyzed tuplewise
- Alternative: Analyze several suitable conjunctives and intersect the resulting TID lists

Selection: Filter Methods

- For *filter method* all conditions are set to **true**, which are not supported by an access method
- Resulting predicate: φ'.
- $r' = \sigma_{\varphi'}(r)$ can now be analyzed using indices
- $\sigma_{\varphi}(r')$ on the (hopefully much smaller) intermediate result r' can now be analyzed using the tuplewise approach
- Filter methods are only suitable, if φ' actually reduces the data amount (attention with disjunctions)

Projection

- Relational algebra: with duplicate elimination
- SQL: no duplicate elimination, if not required with **distinct** (modified scan)
- With duplicate elimination:
  - Sorted output of an index is helpful for duplicate elimination
  - Projection on indexed attributes without access on stored tuples

Projection (II)

- Projection $\pi_X(r)$:
  1. r have to be sorted by X
  2. $t \in r$ are dded to the result, in the case that $t(X) \neq$ **previous**$(t(X))$
- Time complexity: $O(n_r \log n_r)$
- If r is already sorted by X: $O(n_r)$
- Key $K \subseteq X$: $O(n_r)$

Scan Semantics

- Scan-based (positional) change operations: Definition of scan semantics $\leadsto$ Effectiveness on subsequent scan operations
- Example: Deletion of current record
- States: before first record, point to a record, between two records, behind last record, in empty set
- Furthermore: Transition rules for states

Scan Semantics (II)

Halloween problem (System R):

- SQL statement:

```
update employee e
   set salary = salary * 1.05
```

- Record-oriented analysis using index scan on $I_{employee}$(`salary`) and immediate index update
- Without specific precautions: infinite number of salary increases

Binary Operations: Set Operations

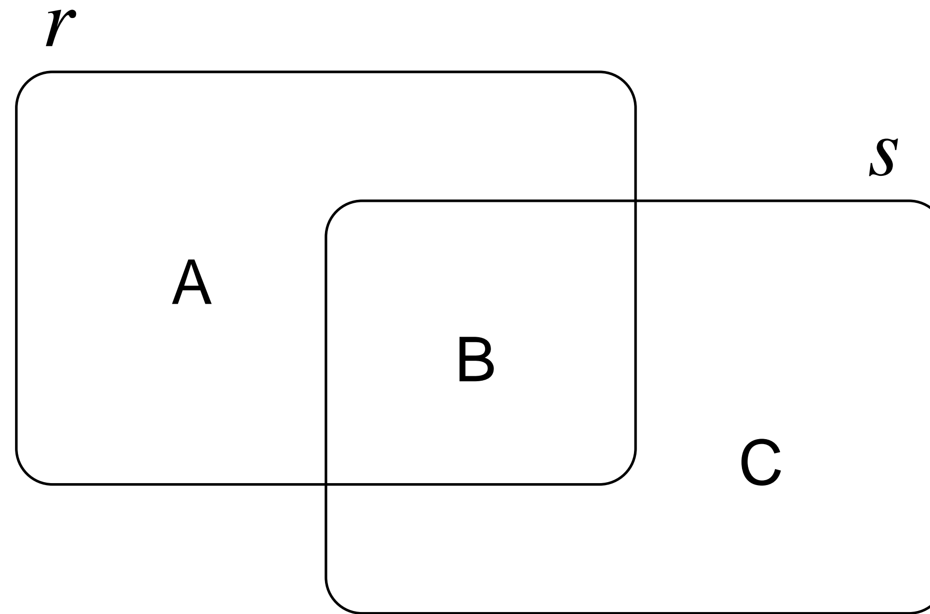
Binary operations mostly realized on the basis of tuplewise comparison of particular sets of tuples

- *Nested-Loops technique* or *loop iteration*
  - The inner relation r is completely traversed for every tuple of an outer relation s
  - Complexity: $O(n_s * n_r)$
- *Merge technique* or *merge method*
  - r and s (sorted) are traversed stepwise in the given tuple order
  - Complexity: $O(n_s + n_r)$
  - If sorting has to be carried out: *Sort-Merge technique*
  - Complexity: $n_r \log n_r$ and/or $n_s \log n_s$

Set Operations (II)

- *Hash methods*
  - Smaller relation in hash table
  - Tuples of second relation are assigned to comparison fellow using a hash function
  - Ideally complexity $O(n_s + n_r)$

Classes of Binary Operations



Classes of Binary Operations (II)

Result extensions	Compliance for all attributes	Compliance for some attributes
A	Difference $r - s$	Anti-Semi Join
B	Intersection $r \cap s$	Join, Semi Join
C	Difference $s - r$	Anti-Semi Join
$A \cup B$		Left Outer Join
$A \cup C$	Symmetric Difference $(r - s) \cup (s - r)$	Anti Join
$B \cup C$		Right Outer Join
$A \cup B \cup C$	Union $r \cup s$	Full Outer Join

Union with Duplicate Elimination

Union by insertion

- Variant of Nested-Loops techniques
- Create copy of one relation r_2 with name r'_2, subsequently insert tuples $t_1 \in r_1$ in r'_2 (time complexity depends on organisation form of copy)

Specific techniques for the union

- Concatenate r and s
- Projection on all attributes of the concatenated relation

Time complexity: $O((n_r + n_s) \times \log(n_r + n_s))$ (like projection)

Union (II)

Union through merge techniques (**merge-union**)

1. Sorting of r and s (if unsorted)
2. Merging of r and s
   - $t_r \in r$ smaller than $t_s \in s$: add t_r to the result, read next $t_r \in r$
   - $t_r \in r$ bigger than $t_s \in s$: add t_s to the result, read next $t_s \in s$
   - $t_s = t_r$: add t_r to the result, read next $t_r \in r$ and $t_s \in s$ respectively

- Time complexity: $O(n_r \times \log n_r + n_s \times \log n_s)$ with sorting, $O(n_r + n_s)$ withot sorting

Computation of Joins

Variants

- Nested-Loops Join
- Block-Nested-Loops Join
- Merge Join
- Hash Join
- . . .

Nested-Loops Join

Double loop iterates over all $t_1 \in r$ and all $t_2 \in s$ for an operation $r \bowtie s$

$r \bowtie_\varphi s$:

```
for each t_r ∈ r do
begin
    for each t_s ∈ s do
    begin
        if φ(t_r, t_s) then put(t_r · t_s) endif
    end
end
```

Nested-Loops Join with Scan

```
R1ScanID := open-rel-scan(R1ID);
R1TID := next-TID(R1ScanID);
while not end-of-scan(R1ScanID) do
begin
   R1Buffer := fetch-tuple(R1ID,R1TID);
   R2ScanID := open-rel-scan(R2ID);
   R2TID := next-TID(R2ScanID);
   while not end-of-scan(R2ScanID) do
   begin
      .../* Scan on inner relation */
   end;
   close (R2ScanID);
   R1TID := next-TID(R1ScanID);
end;
close (R1ScanID);
```

Nested-Loops Join with Scan (II)

```
/* Scan on inner relation */
R2Buffer := fetch-tuple(R2ID,R2TID);
if R1Buffer.X = R2Buffer.Y
then insert into RES
   (R1.Buffer.A1, ..., R1.Buffer.An, R1.Buffer.X,
   R2.Buffer.B1, ..., R1.Buffer.Bm);
endif;
R2TID := next-TID(R2ScanID);
```

Improvement: Nested-Loops Join connects all $t_1 \in r$ with the result $\sigma_{X=t_1(X)}(s)$ (good for index on X in r_2)

Block-Nested-Loops Join

Iteration takes place on blocks instead of tuples

```
for each Block B_r of r do
begin
    for each Block B_s of s do
    begin
        for each Tuple t_r ∈ B_r do
        begin
            for each Tuple t_s ∈ B_s do
            begin
                if φ(t_r, t_s) then put(t_r · t_s) endif
            end
        end
    end
end
```

Costs: $b_r * b_s$

Merge Techniques

$X := R \cap S$; if unsorted, first of all sorting of r and s according to X

1. $t_r(X) < t_s(X)$, read next $t_r \in r$
2. $t_r(X) > t_s(X)$, read next $t_s \in s$
3. $t_r(X) = t_s(X)$, connect t_r with t_s and all successors of t_s, which are identical with t_s according to X
4. For the first $t'_s \in s$ with $t'_s(X) \neq t_s(X)$ starting with original t_s the same procedure is repeated with the successors of t'_r von t_r, as long as $t_r(X) = t'_r(X)$

Merge Techniques: Complexity

- All tuples with same X-value: $O(n_r \times n_s)$
- X is key of R or S: $O(n_r \log n_r + n_s \log n_s)$
- In the case of (pre)sorted relations even: $O(n_r + n_s)$

Merge Join with Scan

- Join attributes of both relations have key characteristic
- **min**(X) and **max**(X): minimal and maximal stored value for X respectively

Merge Join with Scan (II)

```
R1ScanID := open-index-scan(R1XIndexID,
   min(X), max(X));
R1TID := next-TID(R1ScanID);
R1Buffer := fetch-tuple(R1ID,R1TID);
R2ScanID := open-index-scan(R2YIndexID,
   min(Y), max(Y));
R2TID := next-TID(R2ScanID);
R2Buffer := fetch-tuple(R2ID,R2TID);
while not end-of-scan(R1ScanID)
   and not end-of-scan(R2ScanID) do
begin
      .../* merge */
end;
close (R1ScanID);
close (R2ScanID);
```

Merge Join with Scan (III)

```
/* merge */
if R1Buffer.X < R2Buffer.Y
then R1TID := next-TID(R1ScanID);
   R1Buffer := fetch-tuple(R1ID,R1TID);
else if R1Buffer.X > R2Buffer.y
   then R2TID := next-TID(R2ScanID);
      R2Buffer := fetch-tuple(R2ID,R2TID);
   else insert into RES
      (R1.Buffer.A1, ..., R1.Buffer.An, R1.Buffer.X,
      R2.Buffer.B1, ..., R1.Buffer.Bm);
      R1TID := next-TID(R1ScanID);
      R1Buffer := fetch-tuple(R1ID,R1TID);
      R2TID := next-TID(R2ScanID);
      R2Buffer := fetch-tuple(R2ID,R2TID);
   endif;
endif;
```

Join by Hashing

- Idea:
  - Exploiting available main memory for minimizing the accesses on secondary storage
  - Locating join fellows by hashing
  - Querys such as $r \bowtie_{r.A=s.B} s$

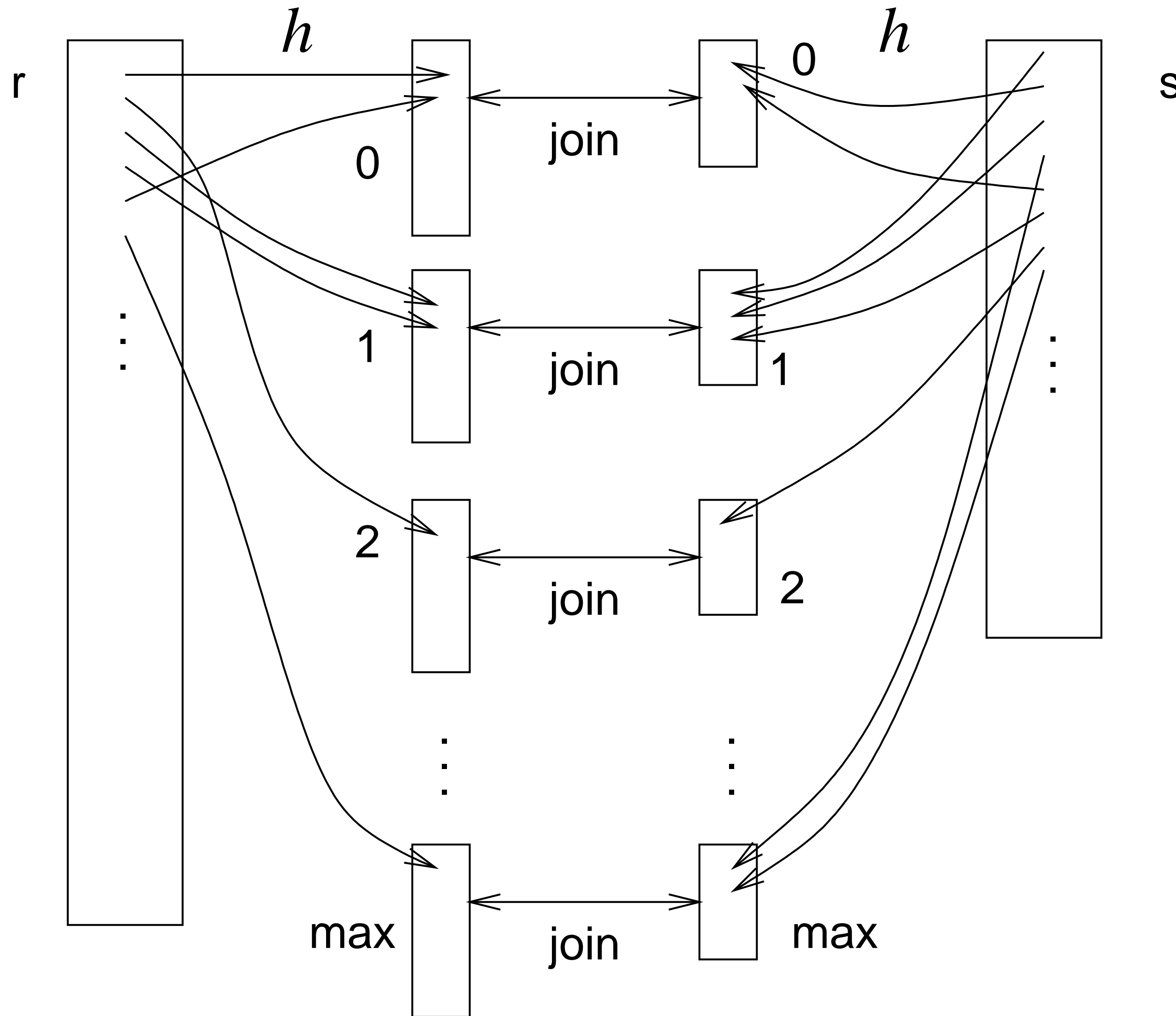
Classical Hashing

- Preparation: Smaller relation becomes r
- Procedure
  1. Tuples of r are read into main memory using scan and inserted into hash table H using hash function $h(r.A)$
  2. If H is full (or r is read completely):
     Scan on S and lookup for join fellow with $h(s.B)$
  3. If scan on r is not finished:
     H is restructured and a scan is executed on S again
- Complexity: $O(b_r + p * b_s)$ with p as number of scans on s

Partitioning using Hash Function

- Tuples from r and s over X are „hashed“ in common file with k blocks (*buckets*)
- Tuples in same bucket are connected using a join algorithm

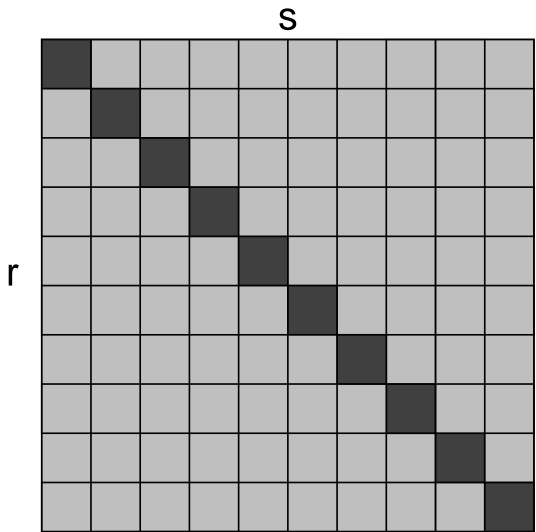
Partitioning using Hash Function (II)



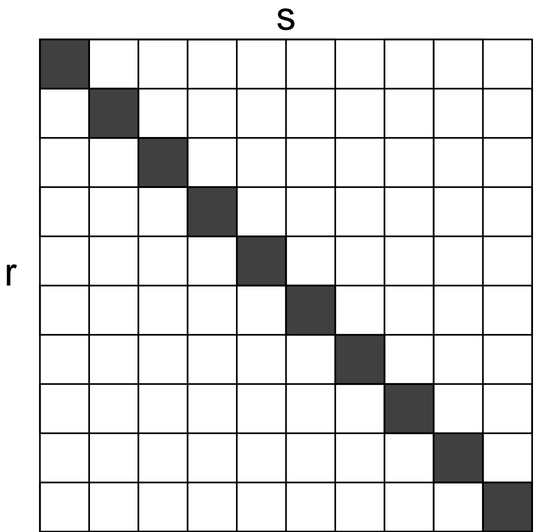
Partitioning using Hash Function (III)

```
for each t_r in r do
    begin
        i := h(t_r(X));
        H_i^r := H_i^r ∪ t_r(X);
    end;
for each t_s in s do
    begin
        i := h(t_s(X));
        H_i^s := H_i^s ∪ t_s(X);
    end;
for each k in 0...max do
    H_k^r ⋈ H_k^s;
```

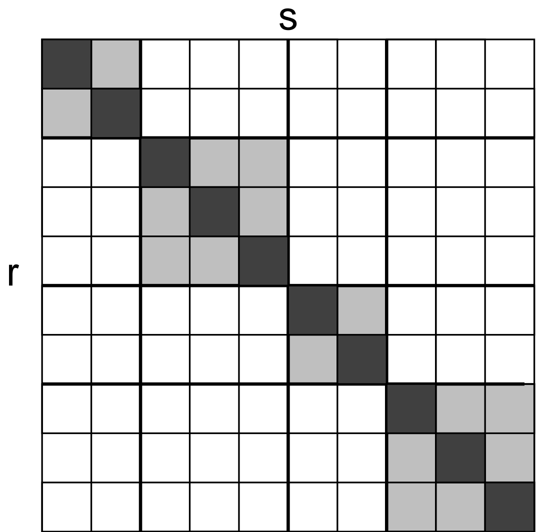
Comparison of Techniques



Nested–Loops Join



Merge Join



Hash Join

Aggregation & Grouping

- Queries:

```
select A, count(*)
from T
group by A
```

- Algebra operator: $\gamma_{\text{count(*)},A}(r(t))$
- Implementation variants:
  - Nested Loops
  - Sorting
  - Hashing