Relational Database Systems 2
5. Query Processing

Christoph Lofi
Benjamin Köhncke
Institut für Informationssysteme
Technische Universität Braunschweig
http://www.ifis.cs.tu-bs.de
• Tree data structures are good index structures
  – $O(\log n)$ performance in average
  – But they may degenerate $\Rightarrow O(n)$
    • Balancing necessary!
  – Block structure of hard discs must be considered
    • $\Rightarrow$ Block trees
  – B-Tree
    • Self-balancing block tree with fill-guarantees
  – B+-Tree
    • Special inner nodes without data pointers
    • Leaf nodes optimized for linear traversal
5 Query Processing

5.1 Introduction: the Query Processor
5.2 How do DBMS actually answer queries?
5.3 Query Parsing/Translation
5.4 Query Optimization
5.5 Query Execution
5.6 Implementation of Joins
5.1 Introduction

- What is a **query processor**?
  - Remember: Simple View of a DBMS
5.1 Introduction

• Queries are posed to the DBMS and processed before the actual evaluation
A query is usually stated in a high-level declarative DB language (e.g., SQL)

- For relational databases: DB language can be mapped to relational algebra for further processing

To be evaluated it has to be translated into a low level execution plan

- Expressions that can be used at physical level of the file system
- For relational databases: physical relational algebra
  - Extends relational algebra with primitives to search through internal data structures
5.2 Query Processing

Parser & Translator

Query

Relational Algebra Expression

Query Optimizer

Evaluation Engine

Query Result

Execution Plan

Access Paths

Statistics

Data

Query

Parser & Translator

Relational Algebra Expression

Query Optimizer

Evaluation Engine

Query Result

Execution Plan

Access Paths

Statistics

Data
5.3 Parser and Translator

• Queries need to be translated to an internal form
  – Queries posed in a **declarative** DB language
    • “what should be returned”, not “where is it found”
  – Queries can be evaluated in different ways

• Scanner **tokenizes** the query
  – DB language keywords, table names, attribute names, etc.

• Parser **checks** syntax and **verifies** relations, attributes, data types, etc.
5.3 Parser and Translator

• Result of the scanning/parsing process
  – Either query is executable, or error message is returned (e.g., SQLCODE, SQLSTATE, …)
5.3 Parser and Translator

• But often also like this…

```sql
4795 data DB2 UDB relation data serv sqlrr_rds_common_post fn c (3.3.18.161.0.510)
pid 333360 tid 1 cpid -1 node 0 probe 510
bytes 16
    Consistency Token: 2AxFKHEW

4796 data DB2 UDB relation data serv sqlrr_rds_common_post fn c (3.3.18.161.0.511)
pid 333360 tid 1 cpid -1 node 0 probe 511
bytes 144
    sqlca.sqlcaid : SQLCA
    sqlca.sqlcabc : 136
    sqlca.sqlcode : -818
    sqlca.sqlerrm : 0
    sqlca.sqlerrmc :
    sqlca.sqlerrp : SQLRALDP
    sqlca.sqlerrd[1] : 0x8012006D
    sqlca.sqlerrd[2] : 0x00000000
    sqlca.sqlerrd[3] : 0x00000000
    sqlca.sqlerrd[4] : 0x00000000
    sqlca.sqlerrd[5] : 0x00000000
    sqlca.sqlerrd[6] : 0x00000000
    sqlca.sqlwarn :
    sqlca.sqlstate : 51003
```
• **Translation** into relational algebra is necessary for actually evaluating the query
  – Internal exchange format between DBMS components
  – Algebra allows for symbolic calculations
    • Important for query optimization
  – Individual operators can be annotated with execution algorithms
    • Evaluation primitives
5.3 Parser and Translator

- Evaluation primitives refer to a single operator
  - Tuple scan operators
  - Tuple selection operators
  - Index scan operators
  - Various join operators
  - Sort operator
  - Duplicate elimination operator
  - ...
• A crash course relational algebra and SQL
  – Basic operations
  – Translation
Relational Algebra

• Made popular by E.F. Codd 1970
• Theoretical foundation of relational databases
  – Describes how to retrieve interesting parts of available relations
  – Lead to the development of SQL
  – Relational algebra is mandatory to understand the query optimization process
    • Topic of the next lecture
• Defines six base operations
  – Selection
  – Projection
  – Cartesian Product
  – Set Union
  – Set Difference
  – Rename
### Example Relations

#### students

<table>
<thead>
<tr>
<th>matNr</th>
<th>firstName</th>
<th>lastName</th>
<th>sex</th>
</tr>
</thead>
<tbody>
<tr>
<td>1005</td>
<td>Clark</td>
<td>Kent</td>
<td>male</td>
</tr>
<tr>
<td>2832</td>
<td>Lois</td>
<td>Lane</td>
<td>female</td>
</tr>
<tr>
<td>4512</td>
<td>Lex</td>
<td>Luther</td>
<td>male</td>
</tr>
<tr>
<td>5119</td>
<td>Charles</td>
<td>Xavier</td>
<td>male</td>
</tr>
<tr>
<td>6676</td>
<td>Erik</td>
<td>Magnus</td>
<td>male</td>
</tr>
<tr>
<td>8024</td>
<td>Jean</td>
<td>Gray</td>
<td>female</td>
</tr>
<tr>
<td>9876</td>
<td>Logan</td>
<td></td>
<td>male</td>
</tr>
</tbody>
</table>

#### courses

<table>
<thead>
<tr>
<th>crsNr</th>
<th>title</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>Intro. to being a Superhero</td>
</tr>
<tr>
<td>101</td>
<td>Secret Identities 2</td>
</tr>
<tr>
<td>102</td>
<td>How to take over the world</td>
</tr>
</tbody>
</table>

#### exams

<table>
<thead>
<tr>
<th>student</th>
<th>course</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>9876</td>
<td>100</td>
<td>3.7</td>
</tr>
<tr>
<td>2832</td>
<td>102</td>
<td>5.0</td>
</tr>
<tr>
<td>1005</td>
<td>101</td>
<td>4.0</td>
</tr>
<tr>
<td>1005</td>
<td>100</td>
<td>1.3</td>
</tr>
<tr>
<td>6676</td>
<td>102</td>
<td>1.3</td>
</tr>
<tr>
<td>5119</td>
<td>101</td>
<td>1.7</td>
</tr>
</tbody>
</table>
• **Selection σ**
  
  – Selects all tuples (rows) fulfilling a given Boolean expression from a relation
  
  – $\sigma_{\text{condition}}(R)$
  
  – **Condition clauses:**
    
    • $\langle \text{attribute} \rangle \theta <value>$
    • $\langle \text{attribute} \rangle \theta <\text{attribute}>$
    • $\theta \in \{=, <, \leq, \geq, >, \neq\}$
  
  – Clauses may be connected by $\land, \lor$ and $\lnot$
• Selection Examples

\[ \sigma_{sex=\text{female}} \text{students} \]

<table>
<thead>
<tr>
<th>matNr</th>
<th>firstName</th>
<th>lastName</th>
<th>sex</th>
</tr>
</thead>
<tbody>
<tr>
<td>2832</td>
<td>Lois</td>
<td>Lane</td>
<td>female</td>
</tr>
<tr>
<td>8024</td>
<td>Jean</td>
<td>Gray</td>
<td>female</td>
</tr>
</tbody>
</table>

\[ \sigma_{\text{course}=100 \land \text{result} \geq 3.0} \text{exams} \]

<table>
<thead>
<tr>
<th>student</th>
<th>course</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>9876</td>
<td>100</td>
<td>3.7</td>
</tr>
</tbody>
</table>
• **Projection** $\pi$
  
  – Retrieves only attributes (columns) with given names
  
  \[ \pi_{<\text{attributeList}>}(R) \]

\[
\begin{array}{|l|}
\hline
\pi_{\text{title}} \text{courses} \\
\hline
\text{title} \\
\hline
\text{Intro. to being a Superhero} \\
\hline
\text{Secret Identities 2} \\
\hline
\text{How to take over the world} \\
\hline
\end{array}
\]

\[
\begin{array}{|l|l|}
\hline
\pi_{\text{firstName, lastName}} \sigma_{\text{sex=female}} \text{students} \\
\hline
\text{firstName} & \text{lastName} \\
\hline
\text{Lois} & \text{Lane} \\
\hline
\text{Jean} & \text{Gray} \\
\hline
\end{array}
\]
• **Rename operator** \( \rho \)

  – Renames a relation \( S \) and/or its attributes

  • Also denoted by \( \leftarrow \)

\[
\rho_{S(B_1, B_2, \ldots, B_n)}(R) \quad \text{or} \quad \rho_S(R) \quad \text{or} \quad \rho_{(B_1, B_2, \ldots, B_n)}(R)
\]

\[\text{lectures} \leftarrow \pi_{\text{crsNr}} \text{courses}\]

\[\rho_{\text{results}(\text{matNo, crsNo, grade})} \sigma_{\text{course}=100} \text{exams}\]
• **Union $\cup$, Intersection $\cap$ and Set Difference** –
  – Operators work as already known from set theory
    • Operands have to be union-compatible (i.e. have to have same attributes)
  – $R \cup S$ or $R \cap S$ or $R-S$ (alt.: $R\setminus S$)

\[
\sigma_{\text{course}=100} \text{exams} \cup \sigma_{\text{course}=102} \text{exams}
\]

<table>
<thead>
<tr>
<th>crsNr</th>
<th>title</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>Intro. to being a Superhero</td>
</tr>
<tr>
<td>102</td>
<td>How to take over the world</td>
</tr>
</tbody>
</table>
• **Cartesian Product** \( \times \)
  
  – Also called cross product
  
  – Creates a new relation combining two relations in a combinatorial fashion
  
  – \( R \times S \)
  
  – Will create a new relation with all attributes of \( R \) and all attributes of \( S \)
  
  – Each entry of \( R \) will be combined with each entry of \( S \)
  
  • Result will have \(|R| \times |S|\) rows
### Relational Algebra

#### badGrades

<table>
<thead>
<tr>
<th>student</th>
<th>course</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>9876</td>
<td>100</td>
<td>3.7</td>
</tr>
<tr>
<td>2832</td>
<td>102</td>
<td>5.0</td>
</tr>
<tr>
<td>1005</td>
<td>101</td>
<td>4.0</td>
</tr>
</tbody>
</table>

#### females

<table>
<thead>
<tr>
<th>matNo</th>
<th>lastName</th>
</tr>
</thead>
<tbody>
<tr>
<td>2832</td>
<td>Lane</td>
</tr>
<tr>
<td>8024</td>
<td>Gray</td>
</tr>
</tbody>
</table>

#### cross

\[
\text{cross} \leftarrow \text{badGrades} \times \text{females}
\]

\[
\text{badGrades} \leftarrow \sigma_{\text{result} \geq 3.0} \text{exams}
\]

\[
\text{females} \leftarrow \pi_{\text{matNo}, \text{lastName}} \sigma_{\text{sex} = \text{female}} \text{students}
\]

<table>
<thead>
<tr>
<th>student</th>
<th>course</th>
<th>result</th>
<th>matNo</th>
<th>lastName</th>
</tr>
</thead>
<tbody>
<tr>
<td>9876</td>
<td>100</td>
<td>3.7</td>
<td>2832</td>
<td>Lane</td>
</tr>
<tr>
<td>2832</td>
<td>102</td>
<td>5.0</td>
<td>2832</td>
<td>Lane</td>
</tr>
<tr>
<td>1005</td>
<td>101</td>
<td>4.0</td>
<td>2832</td>
<td>Lane</td>
</tr>
<tr>
<td>9876</td>
<td>100</td>
<td>3.7</td>
<td>8024</td>
<td>Gray</td>
</tr>
<tr>
<td>2832</td>
<td>102</td>
<td>5.0</td>
<td>8024</td>
<td>Gray</td>
</tr>
<tr>
<td>1005</td>
<td>101</td>
<td>4.0</td>
<td>8024</td>
<td>Gray</td>
</tr>
</tbody>
</table>
The combination of Projection, Selection and Cartesian Product is very important for DB queries.

This kind of query is called “join.”

\[
\pi_{lastName, title, result} \sigma_{matNo=student \land course=crsNo} females \times badGrades \times courses
\]

<table>
<thead>
<tr>
<th>lastName</th>
<th>course</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lane</td>
<td>How to take over the world?</td>
<td>5.0</td>
</tr>
</tbody>
</table>
Theta Join

- Creates a new relation combining two relations by joining related tuples
- $R \bowtie_{\sigma(\text{condition})} S$
- Theta joins can have similar conditions to selections

\[
\pi_{\text{lastName}, \text{title}, \text{result}} \text{females} \bowtie_{\sigma(\text{matNo}=\text{student})} \text{badGrades} \bowtie_{\sigma(\text{course}=\text{crsNo})} \text{courses}
\]

\[
\equiv
\]

\[
\pi_{\text{lastName}, \text{title}, \text{result}} \sigma_{\text{matNo}=\text{student} \land \text{course}=\text{crsNo}} \text{females} \times \text{badGrades} \times \text{courses}
\]

<table>
<thead>
<tr>
<th>lastName</th>
<th>title</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lane</td>
<td>How to take over the world?</td>
<td>5.0</td>
</tr>
</tbody>
</table>
• **EquiJoin**
  
  – Joins two relations only using equivalence conditions
  
  – \( R \bowtie_{(condition)} S \)
  
  – Condition may only contain equivalences between attributes (\( a_1 = a_2 \))
  
  – Specialization of Theta Join

\[
\pi_{lastName, title, result} \text{females} \bowtie_{matNo=student} \text{badGrades} \bowtie_{course=crsNo} \text{courses} \\
\equiv \\
\pi_{lastName, title, result} \sigma_{matNo=student \land course=crsNo} \text{females} \times \text{badGrades} \times \text{courses}
\]

<table>
<thead>
<tr>
<th>lastName</th>
<th>title</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lane</td>
<td>How to take over the world?</td>
<td>5.0</td>
</tr>
</tbody>
</table>
• **Natural Join** $\bowtie$
  
  – Specialization of EquiJoin
  
  – $R \bowtie_{(\text{attributeList})} S$

  – Implicit join condition
  
  • Join attributes in list need to have equal names in both relations
  
  • If no attributes are explicitly stated, all attributes with equal names are implicitly used
## Relational Algebra

### badGrades

<table>
<thead>
<tr>
<th>matNo</th>
<th>crsNo</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>9876</td>
<td>100</td>
<td>3.7</td>
</tr>
<tr>
<td>2832</td>
<td>102</td>
<td>5.0</td>
</tr>
<tr>
<td>1005</td>
<td>101</td>
<td>4.0</td>
</tr>
</tbody>
</table>

### females

<table>
<thead>
<tr>
<th>matNo</th>
<th>lastName</th>
</tr>
</thead>
<tbody>
<tr>
<td>2832</td>
<td>Lane</td>
</tr>
<tr>
<td>8024</td>
<td>Gray</td>
</tr>
</tbody>
</table>

### courses

<table>
<thead>
<tr>
<th>crsNo</th>
<th>title</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>Intro. to being a Superhero</td>
</tr>
<tr>
<td>101</td>
<td>Secret Identities 2</td>
</tr>
<tr>
<td>102</td>
<td>How to take over the world</td>
</tr>
</tbody>
</table>

\[ \pi_{lastName, title, result} (\text{females} \bowtie_{matNo} \text{badGrades} \bowtie_{matNo} \text{courses}) \]

<table>
<thead>
<tr>
<th>lastName</th>
<th>title</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lane</td>
<td>How to take over the world?</td>
<td>5.0</td>
</tr>
</tbody>
</table>
• SQL (Structured Query Language)
  – Most renowned implementation of relational algebra
  – Originally invented 1970 by IBM for System R (SEQUEL)
    • Donald D. Chamberlin and Raymond F. Boyce
  – Standardized multiple times
    • 1986 by ANSI (ANSI SQL, SQL-86)
      – Accredited by ISO in 1987 (SQL-87)
    • 1992 by ISO (SQL2, SQL-92)
      – Added additional types, alterations, functions, more joins, security features, etc.
      – Supported by most major databases
• 1999 by ISO (SQL 3, SQL:1999)
  – Added procedural and recursive queries, regular expression matching, triggers, OOP features, etc.

• 2003 by ISO (SQL:2003)
  – Added basic XML support, auto-generated keys and sequences, etc.

• 2006 by ISO (SQL:2006)
  – Deeper integration with XML, support for mixed relational and XML databases, XQuery integration

• 2008 by ISO (SQL:2008)
  – Truncate table statement, enhanced merge and diagnostic statements, instead of triggers, …

– However, most database vendors use proprietary forks of the standards

• SQL developed for one DBMS often needs adoption to be ported
• Basic Select-Query Structure
  – SELECT <Attributes>
  FROM <Relation>
  WHERE <condition>

• Map relational algebra to SQL
  – $\pi_{(attributeList)} R : \text{Select } attributeList \text{ from } R$
  – $\sigma_{(condition)} R : \ldots \text{ where (condition)}$
SQL

\[ \sigma_{\text{course}=100 \land \text{result} \geq 3.0} \text{exams} \]

\[ \text{select} \ * \ \text{from} \ \text{exams} \ \text{where} \ \text{course}=100 \ \text{and} \ \text{result} \geq 3 \]

\[ \pi_{\text{title}} \text{courses} \]

\[ \text{select} \ \text{title} \ \text{from} \ \text{courses} \]

\[ \pi_{\text{firstName}, \text{lastName}} \sigma_{\text{sex}=\text{female}} \text{students} \]

\[ \text{select} \ \text{firstname}, \ \text{lastName} \ \text{from} \ \text{students} \ \text{where} \ \text{sex}='\text{female}' \]
Joins are explicitly indicated by the `join` keyword

- 4 types of “normal” join: `inner`, `outer`, `left`, `right`

```sql
students ⋈_{students.matNo=exams.student} exams
```

```
select * from students inner join exams on students.matNo=exams.student
```

Joins are often also specified implicitly

- May lead to performance lacks as Cartesian product may be computed

```
select * from students, exams where students.matNo=exams.student
```
• Cartesian Product (implicit & explicit)

students \times \text{exams}

\text{select * from students, exams}

\text{select * from students cross join exams}

• Natural Join

students \bowtie \text{exams}

\text{select * from students natural join exams}
Several relational algebra expressions might lead to the **same results**
- Each statement can be used for query evaluation
- But... **different statements might also result in vastly different performance!**

This is the area of **query optimization**, the heart of every database kernel
- Avoid crappy operator orders by all means
- Next lecture...
5.4 Query Optimization

\[
\pi_{\text{firstName}, \text{lastName}, \text{result}}\sigma_{\text{student}=\text{matNo} \land \text{course}=100} \text{students} \times \text{exams}
\]

\[\equiv\]

\[
\pi_{\text{firstName}, \text{lastName}, \text{result}}\sigma_{\text{student}=\text{matNo}} \left(\pi_{\text{firstName}, \text{lastName}, \text{matNo}} \text{students} \times \sigma_{\text{course}=100} \text{exams}\right)
\]

\[\equiv\]

\[
\pi_{\text{firstName}, \text{lastName}, \text{result}} \left(\text{students} \bowtie \text{student}=\text{matNo} \left(\sigma_{\text{course}=100} \text{exams}\right)\right)
\]
• The query optimization determines the specific order of the relational algebra operators
  – Operator tree
• Still each \textbf{single} relational algebra operator can be evaluated using one of several different algorithms
• The \textbf{evaluation plan} is an annotated expression that specifies a detailed evaluation strategy
5.5 Query Execution

- Annotated Operator Tree
  - Simplified

\[ \pi_{\text{attributes}}(r_1 \bowtie r_2 \bowtie \ldots \bowtie r_k; \sigma_{\text{condition}}) \]

- Use Linear Scan
- Use Merge Join Algorithm
- Use Hash Join Algorithm
- Use Index \(_1\)
- Buffer Completely
5.5 Query Execution

• Cost of each operator is usually measured as total elapsed time for executing the operator.

• The time cost is given by
  – Number of disk accesses
    • Simply scanning relations vs. using index structures
  – CPU time
  – Network communication

• Usually disk accesses are the predominant factor.
5.5 Query Execution

• Disk accesses can be measured by
  – (Number of seeks * average seek costs)
  – (Number of block reads * average block read costs)
  – (Number of blocks writes * average block write costs)

  • Costs for writing a block are higher than costs for reading it, because data is usually also read after writing for verification

• Since CPU time is mostly negligible, it is often ignored for simplicity
  – But remember In-Memory-Databases…
5.5 Query Execution

• The select operator evaluation primitives
  – Relation scan
  – Index lookup
  – Relation scan with comparison
  – Complex selections
5.5 Relation Scan

• Used to locate and retrieve records that fulfill a **selection** condition

• **Linear Search** over relation R
  – Fetch pages from database files on disk that contain records from R
  – Scan each record and retrieve all rows fulfilling the condition

• Cost estimate:
  \#pages containing records of relation R
  – 0.5* \#pages on average, if selection is on a primary key attribute
  • Scanning can be stopped after record is found
5.5 Relation Scan

• **Binary Search** over ordered relation R
  
  • only applicable, if selection is equality on ordered attribute
  
  – Assume that relation R is stored contiguously
  
  – Fetch median page from database files on disk that contain records from R
  
  – Decide whether tuple is in previous or later segment and repeat the median search until record has been found

• Cost estimate:
  \[ \lceil \log_2(\text{#pages containing records of relation R}) \rceil \]
  
  – Actually: cost for locating the first tuple
  
  – Plus: number of additional pages that contain records satisfying the selection condition (overflow lists)
5.5 Index Scan

- **Index Scan** over relation R
  - Selection condition is on search key of an index
  - If there is an index – use it!
  - Equality selection on **primary index** for **key** attributes
    - Cost estimate: Height of the tree or 1 for hash index
  - Equality selection on **primary index** for **non-key** attributes (**cluster index**)
    - Records will be on contiguous pages
    - Cost estimate: (Height of the tree or 1 for hash index) plus #pages that contain overflow lists
5.5 Index Scan

- **Index Scan** over relation R
  - Equality selection on secondary index
    - Remember: records will usually not be on contiguous pages, i.e. cost for page accesses will be much higher
    - **Cost estimate:**
      - for secondary keys: Height of the tree or 1 for hash index
      - For non-keys: (Height of the tree or 1 for hash index) plus #records retrieved
• For a **comparative selection** of the form $(R.a \ \theta \ \text{value})$ a linear scan/binary search can be used
  – Simple range query

• If a **primary** index is available (sorted on $a$)
  – For $\theta \in \{>, \geq\}$ find first value and scan rest of relation
  – For $\theta \in \{<, \leq\}$ do not use index, but scan relation until value is found
5.5 Scan with Comparison

- **If a secondary index is available**
  - For $\theta \in \{>, \geq\}$ find first value and scan rest of index to find pointers to records
  - For $\theta \in \{<, \leq\}$ scan index and collect record pointers until value is found

- **In any case for the cost estimation:**
  - *Every record needs to be fetched* (plus costs of locating first record)
  - Linear file scan *maybe cheaper*, if many records have to be retrieved
5.5 Complex Selection Conditions

• Selection conditions can be joined with logical junctors *and*, *or* and *not*
  – Depending on the existence of indexes in some cases a combination of the above algorithms can be used
    • For **conjunctions** always choose the operation with **smallest selectivity** for the relation scan and then test all other conditions for records in main memory
    • For **disjunctions** make sure that each block is only fetched once
    • For **negations** usually a linear scan is needed
  – Often a complete linear index scan is cheaper anyway
5.5 Sorting Operations

• Sometimes an operator needs to sort records in a relation that is not already stored in sorted order
  – “ORDER BY”-clause in SQL for non-primary key
  – Sorting may increase performance for more complex algorithms

• With a secondary index on the sort key the relation can be read in-order
  – Inefficient, since relation is only sorted logically
5.5 Sorting Operations

• Sorting is **no problem**, if entire relation fits into main memory
  – Quicksort & Co.

• If only some pages of a relation can be fetched into the DB buffer, **external sorting** is needed
  – most commonly used: **Merge-Sort** (n-way merge)
    • Divide relation file into **runs**, sort each run separately in main memory, and write all sorted runs back to disk
    • Merge results by reading records from each run and integrating them into the completely sorted result relation
5.5 Sorting Operations

- Merge Sort by first name (assume 4 records fit into buffer)

<table>
<thead>
<tr>
<th>matNr</th>
<th>firstName</th>
</tr>
</thead>
<tbody>
<tr>
<td>1005</td>
<td>Clark</td>
</tr>
<tr>
<td>2832</td>
<td>Lois</td>
</tr>
<tr>
<td>4512</td>
<td>Lex</td>
</tr>
<tr>
<td>5119</td>
<td>Charles</td>
</tr>
<tr>
<td>6676</td>
<td>Erik</td>
</tr>
<tr>
<td>8024</td>
<td>Jean</td>
</tr>
<tr>
<td>9875</td>
<td>Bruce</td>
</tr>
<tr>
<td>9967</td>
<td>Peter</td>
</tr>
</tbody>
</table>

Relation on Disk

Run 1

<table>
<thead>
<tr>
<th>matNr</th>
<th>firstName</th>
</tr>
</thead>
<tbody>
<tr>
<td>5119</td>
<td>Charles</td>
</tr>
<tr>
<td>1005</td>
<td>Clark</td>
</tr>
<tr>
<td>4512</td>
<td>Lex</td>
</tr>
<tr>
<td>2832</td>
<td>Lois</td>
</tr>
</tbody>
</table>

Run 2

<table>
<thead>
<tr>
<th>matNr</th>
<th>firstName</th>
</tr>
</thead>
<tbody>
<tr>
<td>9875</td>
<td>Bruce</td>
</tr>
<tr>
<td>5119</td>
<td>Charles</td>
</tr>
<tr>
<td>1005</td>
<td>Clark</td>
</tr>
<tr>
<td>6676</td>
<td>Erik</td>
</tr>
<tr>
<td>8024</td>
<td>Jean</td>
</tr>
<tr>
<td>4512</td>
<td>Lex</td>
</tr>
<tr>
<td>2832</td>
<td>Lois</td>
</tr>
</tbody>
</table>

Relation on Disk
5.5 Projection

- **Projection** can be implemented by performing a projection on each tuple
  - Generally needs complete relation scan, since relations are split horizontally over blocks
  - If only non-key attributes are involved, **duplicates** have to be removed
    - Duplicate elimination can be easily implemented using sorting such that identical records appear adjacent
    - If merge sort is used, duplicates can already be deleted in each run, before merging
5.5 Set Operations

• All **union, intersection, and set-difference** operations can be implemented by
  
  – First sorting both relations
  
  – Then scanning through each relation
  
  – Producing the result relation by
    
    • **Union**: Returning just one record for all records appearing in both relations
    
    • **Intersection**: selecting only records that occur in both relations
    
    • **Difference**: retaining only records that are absent in the other relation
5.5 Set Operations

• Cost estimate:
  – #pages in first relation + #pages in second relation
  – Plus: sorting costs, if relations are not sorted

• Alternative implementation
  – Partition both relations using the same hash function
  – Build an in-memory hash index for each partition
  – Scan one partition and use the hash index of the other relation’s respective partition to determine, which records to choose for result of set operation
• **Joins** are a special type of the Cartesian product
• Joins usually have to access **two different** relations
  – Only records having a counterpart in the second relation are in the result table
    • Size of join results can be estimated via the **selectivity** of the join condition and the number and overlap of distinct values in both relations
  – Sequences of joins spanning multiple relations are also possible
Possibilities to implement joins include

- Nested loop join
- Block nested loop join
- Index nested loop join
- Merge join
- Hash join
5.6 Implementing Joins

- **Nested Loop Join** \( (T1 \bowtie T2) \)
  - Simplest Join
- **Algorithm**
  - For each record \( r1 \) in \( T1 \) (outer loop)
    - For each record \( r2 \) in \( T2 \) (inner loop)
      - Compare \( r1 \) to \( r2 \) and add to result if they match
- **Example Effort** (block accesses; assuming block buffer size of 1)
  - \( 6 \times (3+1) = 24 \)

<table>
<thead>
<tr>
<th>matNr</th>
<th>firstName</th>
</tr>
</thead>
<tbody>
<tr>
<td>1005</td>
<td>Clark</td>
</tr>
<tr>
<td>2832</td>
<td>Lois</td>
</tr>
<tr>
<td>4512</td>
<td>Lex</td>
</tr>
<tr>
<td>5119</td>
<td>Charles</td>
</tr>
<tr>
<td>6676</td>
<td>Erik</td>
</tr>
<tr>
<td>8024</td>
<td>Jean</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>matNr</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>9876</td>
<td>3.7</td>
</tr>
<tr>
<td>2832</td>
<td>5.0</td>
</tr>
<tr>
<td>1005</td>
<td>4.0</td>
</tr>
<tr>
<td>1005</td>
<td>1.3</td>
</tr>
<tr>
<td>6676</td>
<td>1.3</td>
</tr>
<tr>
<td>5119</td>
<td>1.7</td>
</tr>
</tbody>
</table>

Directly compare all students to all exams
5.6 Implementing Joins

• **Block Nested Loop Join** \((T_1 \bowtie T_2)\)
  – Idea: Reduce block read overhead by prefetching multiple records of \(T_1\)

• **Algorithm**
  – For every \(w_{\text{size}}\) records in \(T_1\)
    • Prefetch \(w_{\text{size}}\) records into \textit{window}
    • For each record \(r_1\) in \textit{window}
      – For each record in \(r_2\) on \(T_2\)
        » Compare \(r_1\) to \(r_2\) and add to result if they match
5.6 Implementing Joins

- Example Effort (block accesses)
  - assuming block buffer size of 1 and \( w_{\text{size}} = 2 \)
  - \( 3*(3+1) = 12 \) block accesses

<table>
<thead>
<tr>
<th>matNr</th>
<th>firstName</th>
</tr>
</thead>
<tbody>
<tr>
<td>1005</td>
<td>Clark</td>
</tr>
<tr>
<td>2832</td>
<td>Lois</td>
</tr>
<tr>
<td>4512</td>
<td>Lex</td>
</tr>
<tr>
<td>5119</td>
<td>Charles</td>
</tr>
<tr>
<td>6676</td>
<td>Erik</td>
</tr>
<tr>
<td>8024</td>
<td>Jean</td>
</tr>
</tbody>
</table>

Compare window to T2

<table>
<thead>
<tr>
<th>matNr</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>9876</td>
<td>3.7</td>
</tr>
<tr>
<td>2832</td>
<td>5.0</td>
</tr>
<tr>
<td>1005</td>
<td>4.0</td>
</tr>
<tr>
<td>1005</td>
<td>1.3</td>
</tr>
<tr>
<td>6676</td>
<td>1.3</td>
</tr>
<tr>
<td>5119</td>
<td>1.7</td>
</tr>
</tbody>
</table>
• **Index Nested Loop Join** \((T_1 \bowtie T_2)\)
  
  – Use *indexes* for inner loop to avoid scanning
    
    • Index may be existing or temporary and just be created for that single join
    
    • May also be used within a block nested algorithm (with window prefetching)

• **Algorithm**
  
  – For each record \(r_1\) in \(T_1\) *(outer loop)*
    
    • Add to result all records matching to \(r_1\) using index lookups
5.6 Implementing Joins

- Example Effort (block accesses)
  - assuming block buffer size of 1, \( w_{\text{size}} = 2 \) and index in main memory
  - 7 block accesses

<table>
<thead>
<tr>
<th>matNr</th>
<th>firstName</th>
</tr>
</thead>
<tbody>
<tr>
<td>1005</td>
<td>Clark</td>
</tr>
<tr>
<td>2832</td>
<td>Lois</td>
</tr>
<tr>
<td>4512</td>
<td>Lex</td>
</tr>
<tr>
<td>5119</td>
<td>Charles</td>
</tr>
<tr>
<td>6676</td>
<td>Erik</td>
</tr>
<tr>
<td>8024</td>
<td>Jean</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>matNr</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>9876</td>
<td>3.7</td>
</tr>
<tr>
<td>2832</td>
<td>5.0</td>
</tr>
<tr>
<td>1005</td>
<td>4.0</td>
</tr>
<tr>
<td>1005</td>
<td>1.3</td>
</tr>
<tr>
<td>6676</td>
<td>1.3</td>
</tr>
<tr>
<td>5119</td>
<td>1.7</td>
</tr>
</tbody>
</table>

Use index to find matches
• **Merge Join** \((T_1 \bowtie T_2)\)
  – Only useable for EquiJoins and NaturalJoins
  – Adapts techniques from Merge Sort

• **Algorithm**
  – If \(T_1\) and \(T_2\) are not sorted by join attributes
    • Sort them into temporary tables
  – Scan through both relations linearly (as in merge sort)
    • Find matching tuples and add them to result
5.6 Implementing Joins

- Example Effort (block accesses)
  - assuming block buffer size of 2
  - 6 block accesses when sorted before
- But sorting also needs effort

<table>
<thead>
<tr>
<th>matNr</th>
<th>firstName</th>
</tr>
</thead>
<tbody>
<tr>
<td>1005</td>
<td>Clark</td>
</tr>
<tr>
<td>2832</td>
<td>Lois</td>
</tr>
<tr>
<td>4512</td>
<td>Lex</td>
</tr>
<tr>
<td>5119</td>
<td>Charles</td>
</tr>
<tr>
<td>6676</td>
<td>Erik</td>
</tr>
<tr>
<td>8024</td>
<td>Jean</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>matNr</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>B4</td>
<td>1005</td>
</tr>
<tr>
<td></td>
<td>1005</td>
</tr>
<tr>
<td>B5</td>
<td>2832</td>
</tr>
<tr>
<td></td>
<td>5119</td>
</tr>
<tr>
<td>B6</td>
<td>6676</td>
</tr>
<tr>
<td></td>
<td>9876</td>
</tr>
</tbody>
</table>

“Merge” relations linearly
• **Hash Join** \((T_1 \bowtie T_2)\)
  – Only useable for EquiJoins and NaturalJoins

• **Algorithm**
  – Hash all tuples in \(T_1\) and \(T_2\) by their join attributes
    • (Tuples with equal values will be in same bucket)
    • For each bucket \(b\)
      – Compare all tuples from \(T_1\) to all tuples from \(T_2\) and add to result if they match
5.6 Implementing Joins

- Example Effort (block accesses)
  - assuming block buffer size of 1; hash table in memory
  - 6 block accesses
• Having a suitable evaluation plan annotated with evaluation primitives for each operator, the query can be **executed**
  
  – For the result of each operator, a **temporary file** has to be created
  
  – Temporary files can be input for other operators
    • e.g., results of a selection on a relation may be input for some Cartesian product
  
  – Storing the temporary files on disk is expensive, but necessary if DB buffer is small
    • Creating a temporary file **for each operator** is generally too expensive
• Often algorithms for **sequences of operations** are built
  – Joins are a prominent example
  – The code for such algorithms is usually generated dynamically
  – We will discuss this in detail during the next lecture
• The Query Processor
  – How do DBMS actually answer queries?
  – Query Parsing/Translation
    • Relational Algebra
  – Query Optimization
  – Query Execution
  – Implementation of Joins
6 Outlook: Query Optimization

- Introduction into query optimization
- Algebraic query rewriting
- Execution cost estimation
- The SQL EXPLAIN statement
- Choosing a plan