Lecture 11
Evaluation of Relational Operators (Joins)

Chapter 14.4: Ramakrishnan & Gehrke
Demetris Zeinalipour
http://www.cs.ucy.ac.cy/~dzeina/courses/epl446
Lecture Outline
Evaluation of Relational Operators

• 14.4) Algorithms for Evaluating **Joins** (Συνενώσεις)
  – **Simple Nested** Loops Join (SNLJ)
  – **Block-Nested** Loop Join (BNLJ)
  – **Index-Nested** Loops Join (INLJ)
  – **Sort-Merge** Join (SNLJ)
  – **Hash-Join** (Grace, Hybrid Hash)
  – **Comparisons:**
    • Hash-Join vs. Block-Nested Loops Join
    • Hash-Join vs. Sort-Merge Join

Omitted
Introduction to Join Evaluation
(Εισαγωγή στην Αποτίμηση του Τελεστή Συνένωσης)

• The JOIN operator (⊗) combines records from two tables in a database, creating a set that can be materialized (saved as an intermediate table) or used on-the-fly (we shall only consider the latter case).
• It is among the most common operators, thus must be optimized carefully.
• We know that $R \otimes S \Leftrightarrow \sigma_c(R \times S)$, yet $R$ and $S$ might be large so $R \times S$ followed by a selection is inefficient!
• Our objective is to implement the join without enumerating the underlying cross-product.
Cartesian Product vs. Join Example
(Παράδειγμα Καρτεσιανού Γινομένου vs. Συνένωσης)

**Cartesian Product**

<table>
<thead>
<tr>
<th>sid</th>
<th>bid</th>
<th>day</th>
<th>rname</th>
</tr>
</thead>
<tbody>
<tr>
<td>28</td>
<td>103</td>
<td>12/4/06</td>
<td>guppy</td>
</tr>
<tr>
<td>28</td>
<td>103</td>
<td>11/3/06</td>
<td>uppy</td>
</tr>
<tr>
<td>31</td>
<td>101</td>
<td>10/10/06</td>
<td>lubber</td>
</tr>
<tr>
<td>31</td>
<td>102</td>
<td>10/12/06</td>
<td>lubber</td>
</tr>
<tr>
<td>31</td>
<td>101</td>
<td>10/11/06</td>
<td>lubber</td>
</tr>
<tr>
<td>58</td>
<td>103</td>
<td>11/12/06</td>
<td>dustin</td>
</tr>
</tbody>
</table>

foreach tuple r in R do
  foreach tuple s in S do
    add <r, s> to result

**Join**

<table>
<thead>
<tr>
<th>sid</th>
<th>sname</th>
<th>rating</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>dustin</td>
<td>7</td>
<td>45.0</td>
</tr>
<tr>
<td>28</td>
<td>uppy</td>
<td>9</td>
<td>35.0</td>
</tr>
<tr>
<td>31</td>
<td>lubber</td>
<td>8</td>
<td>55.5</td>
</tr>
<tr>
<td>44</td>
<td>guppy</td>
<td>5</td>
<td>35.0</td>
</tr>
<tr>
<td>58</td>
<td>rusty</td>
<td>10</td>
<td>35.0</td>
</tr>
</tbody>
</table>

foreach tuple r in R do
  foreach tuple s in S do
    if R1.sid=S1.sid then
      add <r, s> to result

**Reserves**

```sql
SELECT * 
FROM Reserves R1, Sailors S1 
WHERE R1.sid=S1.sid
```

**Tuples Returned: M*N (i.e., 30 tuples with the above example)**

28, 103, 12/4/06, guppy, 22, dustin, 7, 45.0
28, 103, 11/3/06, uppy, 22, dustin, 7, 45.0
...
58, 103, 11/12/06, dustin, 22, dustin, 7, 45.0
28, 103, 12/4/06, guppy, 28, uppy, 9, 35.0
...
103, 11/12/06, dustin, 58, rusty, 10, 35.0

**Sailors**

```sql
SELECT sname, rating, age 
FROM Sailors 
WHERE sid=22 OR sid=28 OR sid=31 OR sid=44
```

**Tuples Returned:**

28, 103, 12/4/06, guppy, uppy, 9, 35.0
28, 103, 11/3/06, uppy, uppy, 9, 35.0
31, 101, 10/10/06, dustin, lubber, 8, 55.5
31, 102, 10/12/06, lubber, lubber, 8, 55.5
31, 101, 10/11/06, lubber, lubber, 8, 55.5
58, 103, 11/12/06, dustin, rusty, 10, 35.0

**Tuples Returned: Depends on selectivity (only 6 tuples in example)**
Schema for Examples
(Σχήμα για Παραδείγματα)

• **Notation:**

  - M tuples in **R (Reserves)**, \( p_R \) tuples per page,
    • \( M=1000 \) pages, \( p_R=100 \) tuples/page
  - N tuples in **S (Sailors)**, \( p_S \) tuples per page.
    • \( N=500 \) pages, \( p_S=80 \) tuples/page

  **Reserves** (\( sid: \) integer, \( bid: \) integer, \( day: \) dates, \( rname: \) string)
  **Sailors** (\( sid: \) integer, \( sname: \) string, \( rating: \) integer, \( age: \) real)

• **Query:** SELECT * FROM Reserves R1, Sailors S1
  WHERE R1.sid=S1.sid

• **Cost metric:** # of I/Os.

• We will *ignore output costs* (as always) as the results are sent to the user *on-the-fly*
Simple Nested Loops Join
(Απλή Συνένωση Εμφωλεμένων Βρόγχων)

foreach tuple \( r \) in \( R \) do  // Outer relation
    foreach tuple \( s \) in \( S \) do  // Inner relation
        if \( r_i = s_j \) then add \( <r, s> \) to result

• A) Tuple-at-a-time Nested Loops join: Scan outer relation \( R \), and for each tuple \( r \in R \), we scan the entire inner relation \( S \) a tuple-at-a-time.
   - Cost: \( M + p_R * M * N \) = 1000 + 100*1000*500 = 50,001,000 ~50M I/Os

• B) Page-at-a-time Nested Loops join: Scan outer relation \( R \), and for each page \( \in R \), scan the entire inner relation \( S \) a page-at-a-time
   - Cost: \( M + M * N \) = 1000 + 1000*500 = 501,000 I/Os
   - If smaller relation (\( S \)) is outer, cost = 500 + 500*1000 = 500,500 I/Os

**Rule:** The outer relation should be the smaller of the two relations (recall than \( R \otimes S \Leftrightarrow S \otimes R \), i.e., Commutative (Αντιμεταθετική))
Block Nested Loops Join

- **Problem:** SNLJ algorithm does **not** effectively **utilize buffer pages** (i.e., it uses 3 Buffer pages $B_R$, $B_S$ and $B_{out}$).
- **Idea:** Load the smaller relation in memory (if it fits, its ideal!)

- C) **Block-Nested Loops Join (Case I)**
  - Load the complete **smaller R** relation to memory (assuming it fits)
  - Use one page as an **output buffer**
  - Use **remaining pages** (even 1 page is adequate) to load the larger S in memory and perform the join.

**Cost:** $M+N$
Block Nested Loops Join

- **Problem:** BNLJ spends time to join the results in memory
- **Idea:** Build an In-Memory Hash Table for R (such that the in-memory matching is conducted in $O(1)$ time)

**C) Block-Nested Loops Join (Case II)**
- Load the complete smaller $R$ relation to memory and Build a Hashtable
- Use one page as an output buffer
- Use remaining pages (even 1 page is adequate) to load the larger $S$ in memory and perform the join (by using the in-memory Hashtable).

**Cost:** $M+N$

(But CPU cost is lower)
**Block Nested Loops Join (Συνένωση Εμφωλευμένων Βρόγχων με χρήση Μπλόκ)**

- **Problem**: What if smaller relation can’t fit in buffer?
- **Idea**: Use the previous idea but **break the relation R into blocks** (of size B-2) that can fit into the buffer.

- **C) Block-Nested Loops Join (Case III)**
  - Scan **B-2** pages of smaller R to memory (named R-block) (additionally, could build a hash table for this in-memory table)
  - Use 1 page as an **output buffer** and 1 page to scan S relation to memory a page-at-a-time (named S-page) and perform the join.
  - Need to repeat the above \(\lceil \frac{M}{(B-2)} \rceil \) times (i.e., Number of Rblocks)

```
foreach block of B-2 pages of R do
  foreach page of S do {
    for all matching in-memory tuples \(r \in R\)-block and \(s \in S\)-page,
    add \(\langle r, s \rangle\) to result
  }
```

**Cost**: \(M + N \times \lceil \frac{M}{(B-2)} \rceil\)
Examples of Block Nested Loops

• Let us consider an Example with BNLJ (case III), which has a cost of: \( M + N \times \left\lceil \frac{M}{B-2} \right\rceil \)

• Let us consider various scenarios:
  - Reserves (R) as outer and \( B=102 \)
    • Cost = \( 1000 + 500 \times \left\lceil \frac{1000}{100} \right\rceil = 1000 + 500 \times 10 = 6000 \) I/Os
  - Reserves (R) as outer and \( B=92 \)
    • Cost = \( 1000 + 500 \times \left\lceil \frac{1000}{90} \right\rceil = 1000 + 500 \times 12 = 7000 \) I/Os
  - Sailors (S) as outer and \( B=102 \)
    • Cost = \( 500 + 1000 \times \left\lceil \frac{500}{100} \right\rceil = 500 + 1000 \times 5 = 5500 \) I/Os
  - Sailors (S) as outer and \( B=92 \)
    • Cost = \( 500 + 1000 \times \left\lceil \frac{500}{90} \right\rceil = 500 + 1000 \times 6 = 6500 \) I/Os

• It might be best to divide buffers evenly between R and S (instead of allocating \( B-2 \) to one of the two relations)
  - Seek time can be reduced (data can be transferred sequentially to memory instead of 1 page-at-a-time for the S-page)

SNLJ
BNLJ
INLJ
SMJ

Less Buffers => More I/O
Larger Outer => More I/O
Index Nested Loops Join

- **Problem**: Previous approaches essentially enumerate the $R \times S$ set and do not exploit any existing indexes.

- **Idea**: If there is an index on the join column of one relation (say $S$), why not make it the inner and exploit the index.

- **d) Index-Nested Loops Join**
  - Scan outer relation $R$ (page-at-a-time), for each tuple $r \in R$, we use the available index to retrieve the matching tuples of $S$.
  
  - **Cost**: $M + (p_R \times M \times \text{Index\_Cost})$

- **Index\_Cost = Probing\_Cost + Retrieval\_Cost**
  - **Probing\_Cost**: Depends on Index Type
    - Hash Index: ~1.2 I/Os   B+Tree Index: 2-3 I/Os
  - **Retrieval\_Cost**: Depends on Clustering
    - Clustered (Altern. 2): 1 I/O (typical)   Clustered (Altern. 1): 0 I/Os
    - Unclustered (Altern. 2): upto 1 I/O per matching $S$ tuple.
Examples of Index Nested Loops

Let us consider an Example with INLJ which has a cost:

\[ M + (p_R \times M \times \text{Index Cost}) \]

**Hash-index (Alt. 2) on sid of Sailors (as inner):**
- **Cost** = 1000 + 100 * 1000 * (1.2 + 1.0) = **220,000 I/Os**
- **Retrieval Cost**: 1.2 I/Os to get data entry in index, plus 1.0 I/O to get (the exactly one, as sid is sailor’s key) matching Sailors tuple.
- **Note**: Better than Simple (Page-at-a-time) Nested Loops join: M + M* N, which was **500,500 I/Os**!
- Not comparing with BNLJ as the performance of the latter depends on the buffer size (shall compare BNLJ with SMJ later).

**Hash-index (Alt. 1) on sid of Sailors (as inner):**
- **Cost** = 1000 + 100 * 1000 * (1.2 + 0.0) = **120,000 I/Os**
Sort-Merge Join
(Σύζευξη με Ταξινόμηση και Συγχώνευση)

- Another method, like Index-Nested Loop Join, that avoids enumerating the \( R \times S \) set.
- **Sort-Merge Join** utilizes a **partition-based approach** to join two relations (works only for equality joins).

**e) Sort Merge Join Algorithm:**
- **Sort Phase:** Sort both relations \( R \) and \( S \) on the **join attribute** using an **external sort** algorithm.
- **Merge Phase:** Look for **qualifying tuples** \( r \in R \) and \( s \in S \) by **merging** the two relations.

- Sounds similar to **external sorting**. In fact the Sorting phase of the sort alg. can be combined with the sorting phase of SMJ (we will see this next).
Sort-Merge Join
(Σύζευξη με Ταξινόμηση και Συγχώνευση)

• Sort-Merge Join I/O Cost

\[ \text{I/O Cost} = \text{ExternalSort}(R) + \text{ExternalSort}(S) + M + N \]

\[ = 2M \times \text{#passes} + 2N \times \text{#passes} + M + N \]

\[ = 2M \left( 1 + \left\lceil \log_{B^{-1}} \left\lceil \frac{M}{B} \right\rceil \right) \right) + 2N \left( 1 + \left\lceil \log_{B^{-1}} \left\lceil \frac{N}{B} \right\rceil \right) \right) + M + N \]

• Asymptotically, the I/O cost for SMJ is:

\[ = O(M \log M) + O(N \log N) + O(M + N) \in O(M \log M + N \log N) \]

(However we will utilize the real cost in our equations)

• See next slide for examples…
Sort-Merge Join
(Σύζευξη με Ταξινόμηση και Συγχώνευση)

- Let us consider an Example with SMJ, which has a cost of:
  \[ 2M \left( 1 + \left\lceil \log_{B-1} \left\lceil \frac{M}{B} \right\rceil \right\rceil \right) + 2N \left( 1 + \left\lceil \log_{B-1} \left\lceil \frac{N}{B} \right\rceil \right\rceil \right) + M + N \]

- Let us consider various scenarios:
  - **Buffer B=35, M=1000, N=500**
    - Cost = \[2 \times 1000 \times 2 + 2 \times 500 \times 2 + 1000 + 500 = 7500 \text{ I/Os}\]
      - Note: \[1 + \left\lceil \log_{B-1} \left\lceil \frac{M}{B} \right\rceil \right\rceil = 1 + [\log_{34} 1000/35] = 1 + [0.73] = 2\]
    - Block-Nested Loops Join: \[N + M \times [N/(B-2)] = 500 + 1000 \times [500/33] = 16,500 \text{ I/Os}\]
  - **Buffer B=100, M=1000, N=500**
    - Cost = \[2 \times 1000 \times 2 + 2 \times 500 \times 2 + 1000 + 500 = 7500 \text{ I/Os}\]
    - Similar to the Block-Nested Loops Join: \[N + M \times [N/(B-2)] = 6500 \text{ I/Os}\]
  - **Buffer B=300, M=1000, N=500**
    - Cost = \[2 \times 1000 \times 2 + 2 \times 500 \times 2 + 1000 + 500 = 7500 \text{ I/Os}\]
    - Block-Nested Loops Join: \[M + N \times [M/(B-2)] = 500 + 1000 \times [500/300] = 2,500 \text{ I/Os}\]

* The number of passes during sorting remains at 2 in the above examples.
• We can combine the **merging phases of sorting** with the **merging phase of the join**.
  
  - **Step 1 (Sort):** Generate runs of size $B$ for both $R$ and $S$ (using Phase 1 of the External Sort Algorithm)

  ![Diagram]

  - Each Relation is read/written once, thus the **Cost = 2(M+N)**
Step 2 (Merge/Join): Load ALL runs of R and S and join merge/join them.

- How big should the Buffer B be to accomplish this task?
- Answer: $B > 2\sqrt{M}$ i.e.,
  - We need to fit all Runs in memory (i.e., $#R\_runs + #S\_runs$)
  - $B > \#R\_runs + \#S\_runs$
    - $#R\_runs$: $B > \lceil M/(B-1) \rceil \Rightarrow B(B-1) > M$, approximately $B > \sqrt{M}$
    - $#S\_runs$: $B > \lceil N/(B-1) \rceil \Rightarrow B(B-1) > N$, approximately $B > \sqrt{N} \Rightarrow B > \sqrt{M}$
      » To simplify the notation let is assume that a larger buffer is available, i.e., $B > \sqrt{M} > \sqrt{N}$ (where R is the larger relation).
  - Consequently, $B > 2\sqrt{M}$
  - Each Relation is read once (results outputted on-the-fly), consequently the cost of this phase is Cost = $M+N$)

- Total Cost = 3 ($M+N$)
  (Refined SMJ)

In example, cost goes down from 7500 to 4500 I/Os.
Join Alg. in Commercial DBMS
(Αλγόριθμοι Συνένωσης σε Εμπορικές Β.Δ.)

- **Oracle 8** supports page-oriented nested loop, sort-merge join and a variant of hybrid hash join.
- **IBM DB2** supports block-nested loop, sort-merge and hybrid hash join.
- **Microsoft SQL Server** supports block nested loops, index nested loops, sort merge, hash join.
- **Informix**: supports block-nested loops, index-nested loops and hybrid hash join.
- **Sybase ASE**: support index nested loop and Sort-Merge Join.
- **Sybase ASIQ**: page-oriented nested loop, index-nested loop, simple hash join and sort-merge join.