Announcements

• CIRC:
  – CIRC is down!!!
  – MongoDB and Spark (mini) projects are at stake. 😞

• Project 1 Milestone 4
  – is out
  – Due date: Last date of class
    • We will check your website after that date
    • But, finish early

• Due Dates:
  – Suggestions:
Due Dates

- 11/12 to 11/18
- 11/19 to 11/25 (Thanksgiving Week)
- 11/26 to 12/02
- 12/03 to 12/09:
  - Term Paper Due: 12/08
- 12/10 to 12/13 (Last Class):
  - Poster Session on: 12/11
  - Project 1 Milestone 4 is due on 12/13
- Final: December 18, 2017 at 7:15 pm

MongoDB
Spark
Term Paper
Poster Session
Topics for Today

• Query Processing (Chapter 18)
• Query Optimization (Chapter 19) on Wednesday
QUERY PROCESSING
Steps in Query Processing

• Scanning

• Parsing

• Validation

• Query Tree Creation

• Query Optimization (Query planning)

• Code generation (to execute the plan)

• Running the query code
Steps in Query Processing

Query in a high-level language

Scanning, parsing, and validating

Immediate form of query

Query optimizer

Execution plan

Query code generator

Code to execute the query

Runtime database processor

Result of query

Code can be:

- Executed directly (interpreted mode)
- Stored and executed later whenever needed (compiled mode)
SQL Queries

• SQL Queries are decomposed into **Query blocks**:  
  – Select...From...Where...Group By...Having

• Translate **Query blocks** into **Relational Algebraic expression**

• Remember, SQL includes aggregate operators:  
  – MIN, MAX, SUM, COUNT etc.
  – Part of the extended algebra
  – Let’s go back to Chapter 8 (Section 8.4.2)
Aggregate Functions and Grouping (Relational Algebra)

- Aggregate function: \( \exists \)

\[
< \text{grouping attributes} > \exists < \text{function list} > (R)
\]

\[
\begin{array}{|c|c|c|}
\hline
\text{Dno} & \text{COUNT Ssn} & \text{AVERAGE Salary} \\
\hline
5 & 4 & 33250 \\
4 & 3 & 31000 \\
1 & 1 & 55000 \\
\hline
\end{array}
\]
Semijoin (\(\bowtie\))

- \(R \bowtie S = \Pi_{A_1, \ldots, A_n} (R \bowtie S)\)
- Where \(A_1, \ldots, A_n\) are the attributes in \(R\)
- Example:
  - Employee \(\bowtie\) Dependents

**SQL:**

```sql
SELECT DISTINCT sid, sname, gpa
FROM Students, People
WHERE sname = pname;
```

**OR**

```sql
SELECT sid, sname, gpa
FROM Students
WHERE sname IN (SELECT pname FROM People);
```

**RA:**

Students \(\bowtie\) People
EXTERNAL SORTING
External Merge Sort
Why are Sort Algorithms Important?

• Data requested from DB in sorted order is extremely common
  – e.g., find students in increasing GPA order

• Why not just use quicksort in main memory??
  – What about if we need to sort 1TB of data with 1GB of RAM...

A classic problem in computer science!
So how do we sort big files?

1. Split into chunks small enough to sort in memory ("runs")

2. Merge pairs (or groups) of runs using the external merge algorithm

3. Keep merging the resulting runs (each time = a “pass”) until left with one sorted file!
2. EXTERNAL MERGE & SORT
How do we *efficiently* merge two sorted files when both are much larger than our main memory buffer?
External Merge Algorithm

- **Input:** 2 sorted lists of length M and N
- **Output:** 1 sorted list of length M + N
- **Required:** At least 3 Buffer Pages
- **IOs:** \(2(M+N)\)
Key (Simple) Idea

To find an element that is no larger than all elements in two lists, one only needs to compare minimum elements from each list.

If:

\[
\begin{align*}
A_1 &\leq A_2 \leq \cdots \leq A_N \\
B_1 &\leq B_2 \leq \cdots \leq B_M
\end{align*}
\]

Then:

\[
\begin{align*}
\text{Min}(A_1, B_1) &\leq A_i \\
\text{Min}(A_1, B_1) &\leq B_j
\end{align*}
\]

for i=1....N and j=1....M
External Merge Algorithm

Input:
Two sorted files

Output:
One merged sorted file

Disk

Main Memory

Buffer

$F_1$: 1,5 7,11 20,31

$F_2$: 2,22 23,24 25,30
External Merge Algorithm

- **Input:** Two sorted files
- **F₁:** 
  - 7,11
  - 20,31
- **F₂:** 
  - 23,24
  - 25,30
- **Output:** One merged sorted file

**Disk**

**Main Memory**

- **Buffer:**
  - 1,5
  - 2,22
**External Merge Algorithm**

**Input:**
Two sorted files

**Output:**
One *merged* sorted file

---

**Disk**

**F₁**

| 7,11 | 20,31 |

**F₂**

| 23,24 | 25,30 |

**Main Memory**

**Buffer**

| 5 | 22 | 1,2 |
External Merge Algorithm

Input:
Two sorted files

Output:
One *merged* sorted file

Disk

F₁

7,11 20,31

F₂

23,24 25,30

Main Memory

Buffer

5 22
External Merge Algorithm

Input:
Two sorted files

Output:
One *merged* sorted file

Disk

F₁

F₂

1, 2

7, 11  20, 31

23, 24  25, 30

Main Memory

Buffer

22  5

This is all the algorithm “sees”... Which file to load a page from next?
We know that $F_2$ only contains values $\geq 22$... so we should load from $F_1$!
External Merge Algorithm

Input:
Two sorted files

Output:
One merged sorted file

Disk

Main Memory

Buffer

1, 2
20, 31
23, 24
25, 30
7, 11
22
5

F1
F2
External Merge Algorithm

Input: Two sorted files
Output: One merged sorted file

Disk

Main Memory
Buffer

Input:
Two sorted files

Output:
One merged sorted file
External Merge Algorithm

Input:
Two sorted files

Output:
One merged sorted file

Disk

Main Memory
Buffer

1, 2  5, 7

20, 31

F1

F2

23, 24  25, 30

11  22
External Merge Algorithm

Input: Two sorted files
Output: One *merged* sorted file

And so on...
We can merge lists of arbitrary length with only 3 buffer pages.

If lists of size $M$ and $N$, then

**Cost:** $2(M+N)$ IOs

Each page is read once, written once
External Merge Sort Algorithm

Example:
• 3 Buffer pages
• 6-page file

Orange file = unsorted

1. Split into chunks small enough to sort in memory
EXTERNAL MERGE SORT (BEFORE MERGE)
External Merge Sort Algorithm

Example:
- 3 Buffer pages
- 6-page file

Orange file = unsorted

1. Split into chunks small enough to sort in memory
External Merge Sort Algorithm

Example:
- 3 Buffer pages
- 6-page file

Orange file = unsorted

1. Split into chunks small enough to sort in memory
External Merge Sort Algorithm

Example:
- 3 Buffer pages
- 6-page file

Orange file = unsorted

1. Split into chunks small enough to sort in memory
External Merge Sort Algorithm

Example:
- 3 Buffer pages
- 6-page file

Each sorted file is called a run

1. Split into chunks small enough to sort in memory
Example:
- 3 Buffer pages
- 6-page file

2. Now just run the external merge algorithm & we’re done!
Calculating IO Cost

For 3 buffer pages, 6 page file:

1. Split into two 3-page files and sort in memory
   = 1 R + 1 W for each file = 2*(3 + 3) = 12 IO operations

2. Merge each pair of sorted chunks *using the external merge algorithm*
   = 2*(3 + 3) = 12 IO operations

3. Total cost = 24 IO
Running External Merge Sort on Larger Files

Assume we still only have 3 buffer pages (Buffer not pictured)
Running External Merge Sort on Larger Files

1. Split into files small enough to sort in buffer...

Assume we still only have 3 buffer pages (Buffer not pictured)
Running External Merge Sort on Larger Files

1. Split into files small enough to sort in buffer... and sort

Assume we still only have 3 buffer pages (Buffer not pictured)

Call each of these sorted files a *run*
2. Now merge pairs of (sorted) files... the resulting files will be sorted!
Running External Merge Sort on Larger Files

Assume we still only have 3 buffer pages *(Buffer not pictured)*

3. And repeat...

Call each of these steps a *pass*
Running External Merge Sort on Larger Files

4. And repeat!
Assume for simplicity that we split an N-page file into N single-page runs and sort these; then:

- First pass: Merge N/2 pairs of runs each of length 1 page
- Second pass: Merge N/4 pairs of runs each of length 2 pages
- In general, for N pages, we do \( \lceil \log_2 N \rceil \) passes
  – +1 for the initial split & sort
- Each pass involves reading in & writing out all the pages = 2N IO

\[ \rightarrow 2N^* (\lceil \log_2 N \rceil + 1) \text{ total IO cost!} \]
Using B+1 buffer pages to reduce # of passes

Suppose we have B+1 buffer pages now; we can:

1. **Increase length of initial runs.** Sort B+1 at a time!
   At the beginning, we can split the N pages into runs of length B+1 and sort these in memory

**IO Cost:**

<table>
<thead>
<tr>
<th>Starting with runs of length 1</th>
<th>Starting with runs of length B+1</th>
</tr>
</thead>
<tbody>
<tr>
<td>$2N(\lfloor \log_2 N \rfloor + 1)$</td>
<td>$2N(\left\lfloor \log_2 \frac{N}{B+1} \right\rfloor + 1)$</td>
</tr>
</tbody>
</table>
Suppose we have $B+1$ buffer pages now; we can:

2. **Perform a $B$-way merge.**

On each pass, we can merge groups of $B$ runs at a time (vs. merging pairs of runs)!

**IO Cost:**

- Starting with runs of length 1: $2N([\log_2 N] + 1)$
- Starting with runs of length $B+1$: $2N(\left\lceil \frac{N}{B+1} \right\rceil + 1)$
- Performing $B$-way merges: $2N(\left\lfloor \log_B \frac{N}{B+1} \right\rfloor + 1)$
Algorithm for Select Operation

• Read Section 18.3 (18.3.1, 18.3.2, 18.3.3, 18.3.4)
• Mostly covers searching:
  
  • 1. Linear Search
  • 2. Binary Search
  • 3. Indexing
  • 4. Hashing
  • 5. B+ Tree

• (Skip bitmap index and functional index)
Algorithm for Join Operation

- The most time consuming operation
What you will learn about in this section

1. Nested Loop Join (NLJ)
2. Block Nested Loop Join (BNLJ)
3. Index Nested Loop Join (INLJ)
4. Sorted-Merge Join
5. Hash Join
RECAP: Joins
Joins: Example

Example: Returns all pairs of tuples \( r \in R, s \in S \) such that \( r.A = s.A \)

SELECT \( R.A, B, C, D \)
FROM \( R, S \)
WHERE \( R.A = S.A \)
Joins: Example

Example: Returns all pairs of tuples $r \in R, s \in S$ such that $r.A = s.A$.
Example: Returns all pairs of tuples \( r \in R, s \in S \) such that \( r.A = s.A \)

\[
\begin{align*}
\text{SELECT } & \quad R.A, B, C, D \\
\text{FROM } & \quad R, S \\
\text{WHERE } & \quad R.A = S.A
\end{align*}
\]
Joins: Example

\[ R \Join S \]

Example: Returns all pairs of tuples \( r \in R, s \in S \) such that \( r.A = s.A \)

\[
\begin{array}{c|c|c|c}
A & B & C \\
\hline
1 & 0 & 1 \\
2 & 3 & 4 \\
2 & 5 & 2 \\
3 & 1 & 1 \\
\end{array}
\quad \begin{array}{c|c}
A & D \\
\hline
3 & 7 \\
2 & 2 \\
2 & 3 \\
\end{array}
\]

\[
\begin{array}{c|c|c|c}
A & B & C & D \\
\hline
2 & 3 & 4 & 2 \\
2 & 3 & 4 & 3 \\
2 & 5 & 2 & 2 \\
2 & 5 & 2 & 3 \\
\end{array}
\]

SELECT R.A, B, C, D
FROM R, S
WHERE R.A = S.A
Joins: Example

Example: Returns all pairs of tuples \( r \in R, s \in S \) such that \( r.A = s.A \)

\[ R \bowtie S \]

**Select**

\[ \text{SELECT} \quad R.A, B, C, D \]

**From**

\[ \text{FROM} \quad R, S \]

**Where**

\[ \text{WHERE} \quad R.A = S.A \]

**Example:**

\[
\begin{array}{cccc}
R & & S \\
A & B & C & A & D \\
1 & 0 & 1 & 3 & 7 \\
2 & 3 & 4 & 2 & 2 \\
2 & 5 & 2 & 2 & 3 \\
3 & 1 & 1 & 2 & 7 \\
\end{array}
\]
Semantically: A Subset of the Cross Product

\[ R \bowtie S \]

**Example:** Returns all pairs of tuples \( r \in R, s \in S \) such that \( r.A = s.A \)

Can we actually implement a join in this way?
• We write $R \bowtie S$ to mean *join R and S by returning all tuple pairs where all shared attributes are equal*

• We write $R \bowtie S$ on $A$ to mean *join R and S by returning all tuple pairs where attribute(s) $A$ are equal*

• For simplicity, we’ll consider joins on two tables and with equality constraints (“equijoins”)

However joins *can* merge > 2 tables, and some algorithms do support non-equality constraints!
Nested Loop Joins
• We are again considering “IO aware” algorithms: \textit{care about disk IO}

• Given a relation \( R \), let:
  \begin{itemize}
  \item \( T(R) = \# \) of tuples in \( R \)
  \item \( P(R) = \# \) of pages in \( R \)
  \end{itemize}

• Note also that we omit ceilings in calculations… good exercise to put back in!

Recall that we read / write entire pages with disk IO
Nested Loop Join (NLJ)

Compute $R \bowtie S$ on $A$:

for $r$ in $R$:
   for $s$ in $S$:
      if $r[A] == s[A]$:
         yield ($r, s$)
Nested Loop Join (NLJ)

Compute $R \bowtie S$ on $A$:

```python
for r in R:
    for s in S:
        if r[A] == s[A]:
            yield (r, s)
```

Cost:

1. Loop over the tuples in $R$

Note that our IO cost is based on the number of
\textit{pages} loaded, not the number of tuples!
Nested Loop Join (NLJ)

**Compute** $R \bowtie S$ on $A$:

```python
for r in R:
    for s in S:
        if r[A] == s[A]:
            yield (r, s)
```

**Cost:**

$P(R) + T(R) \times P(S)$

1. Loop over the tuples in $R$
2. For every tuple in $R$, loop over all the tuples in $S$

Have to read *all of $S$* from disk for *every tuple in $R$*!
**Nested Loop Join (NLJ)**

Compute \( R \bowtie S \) on \( A \):

for \( r \) in \( R \):
    for \( s \) in \( S \):
        if \( r[A] = s[A] \):
            yield \((r, s)\)

Note that NLJ can handle things other than equality constraints... just check in the *if* statement!

Cost:

\[ P(R) + T(R) \times P(S) \]

1. Loop over the tuples in \( R \)
2. For every tuple in \( R \), loop over all the tuples in \( S \)
3. Check against join conditions
Nested Loop Join (NLJ)

Compute $R \bowtie S$ on $A$:

for $r$ in $R$:
    for $s$ in $S$:
        if $r[A] == s[A]$:
            yield $(r, s)$

Cost:

$P(R) + T(R)*P(S) + OUT$

1. Loop over the tuples in $R$
2. For every tuple in $R$, loop over all the tuples in $S$
3. Check against join conditions
4. Write out (to page, then when page full, to disk)
Nested Loop Join (NLJ)

Compute $R \bowtie S$ on $A$:
for $r$ in $R$:
  for $s$ in $S$:
    if $r[A] == s[A]$:
      yield $(r,s)$

Cost:

$P(R) + T(R) \times P(S) + OUT$

What if $R$ (“outer”) and $S$ (“inner”) switched?

$P(S) + T(S) \times P(R) + OUT$

Outer vs. inner selection makes a huge difference - DBMS needs to know which relation is smaller!
Block Nested Loop Join (BNLJ)
Block Nested Loop Join (BNLJ)

**Compute R \( \bowtie \) S on \( A \):**

1. **for each page \( pr \) of R:**
   - **for each page \( ps \) of S:**
     - **for each tuple \( r \) in \( pr \):**
       - **for each tuple \( s \) in \( ps \):**
         - if \( r[A] == s[A] \):
           - yield \((r,s)\)

**P(R):**

1. Load in 1 page of R at a time (leaving 1 page each free for S & output)

Given 3 pages of memory

Cost:

Note: There could be some speedup here due to the fact that we’re reading in multiple pages sequentially however we’ll ignore this here!
Block Nested Loop Join (BNLJ)

Compute $R \bowtie S$ on $A$:

for each page $pr$ of $R$:
    for page $ps$ of $S$:
        for each tuple $r$ in $pr$:
            for each tuple $s$ in $ps$:
                if $r[A] == s[A]$:
                    yield $(r,s)$

Given 3 pages of memory

Cost:

$P(R) + P(R).P(S)$

1. Load in 1 page of $R$ at a time (leaving 1 page each free for $S$ & output)
2. For each page segment of $R$, load each page of $S$

Note: Faster to iterate over the smaller relation first!
Block Nested Loop Join (BNLJ)

Compute $R \bowtie S$ on $A$:
for each page $pr$ of $R$:
  for page $ps$ of $S$:
    for each tuple $r$ in $pr$:
      for each tuple $s$ in $ps$:
        if $r[A] = s[A]$:
          yield $(r,s)$

Given 3 pages of memory

Cost:
$P(R) + P(R) \cdot P(S)$

1. Load in 1 page of $R$ at a time
   (leaving 1 page each free for $S$ & output)

2. For each page segment of $R$,
   load each page of $S$

3. Check against the join conditions

BNLJ can also handle non-equality constraints
Compute $R \bowtie S$ on $A$:
   for each page $pr$ of $R$:
     for page $ps$ of $S$:
       for each tuple $r$ in $pr$:
         for each tuple $s$ in $ps$:
           if $r[A] == s[A]$:
             yield $(r, s)$

Block Nested Loop Join (BNLJ)

Given 3 pages of memory

Cost:
$P(R) + P(R).P(S)$

1. Load 1 page of $R$ at a time
   (leaving 1 page each free for $S$ & output)
2. For each page segment of $R$,
   load each page of $S$
3. Check against the join conditions
4. Write out
Compute \( R \bowtie S \) on \( A \):

- for each \( B-1 \) pages \( pr \) of \( R \):
  - for page \( ps \) of \( S \):
    - for each tuple \( r \) in \( pr \):
      - for each tuple \( s \) in \( ps \):
        - if \( r[A] == s[A] \):
          - yield \((r,s)\)

Given \( B+1 \) pages of memory

**Cost:**

- 1. Load in \( B-1 \) pages of \( R \) at a time (leaving 1 page each free for \( S \) & output)

**Note:** There could be some speedup here due to the fact that we’re reading in multiple pages sequentially however we’ll ignore this here!
Block Nested Loop Join (BNLJ)

Compute $R \bowtie S$ on $A$:

for each $B-1$ pages $pr$ of $R$:
  for page $ps$ of $S$:
    for each tuple $r$ in $pr$:
      for each tuple $s$ in $ps$:
        if $r[A] == s[A]$:
          yield $(r,s)$

Given $B+1$ pages of memory

Cost:

$$P(R) + \frac{P(R)}{B-1} P(S)$$

1. Load in $B-1$ pages of $R$ at a time (leaving 1 page each free for $S$ & output)

2. For each $(B-1)$-page segment of $R$, load each page of $S$

Note: Faster to iterate over the smaller relation first!
Compute $R \bowtie S$ on $A$:
for each $B-1$ pages $pr$ of $R$:
  for page $ps$ of $S$:
    for each tuple $r$ in $pr$:
      for each tuple $s$ in $ps$:
        if $r[A] = s[A]$:
          yield $(r,s)$

Given $B+1$ pages of memory

Cost:
$P(R) + \frac{P(R)}{B-1} P(S)$

1. Load in $B-1$ pages of $R$ at a time (leaving 1 page each free for $S$ & output)
2. For each ($B-1$)-page segment of $R$, load each page of $S$
3. Check against the join conditions

BNLJ can also handle non-equality constraints
Block Nested Loop Join (BNLJ)

Compute $R \bowtie S$ on $A$:

- for each $B-1$ pages $pr$ of $R$:
  - for each page $ps$ of $S$:
    - for each tuple $r$ in $pr$:
      - for each tuple $s$ in $ps$:
        - if $r[A] == s[A]$:
          - yield $(r,s)$

Given $B+1$ pages of memory

Cost:

$P(R) + \frac{P(R)}{B-1} P(S) + \text{OUT}$

1. Load in $B-1$ pages of $R$ at a time (leaving 1 page each free for $S$ & output)
2. For each $(B-1)$-page segment of $R$, load each page of $S$
3. Check against the join conditions
4. Write out
In BNLJ, by loading larger chunks of R, we minimize the number of full disk reads of S
- We only read all of S from disk for every \((B-1)\)-page segment of R!
- Still the full cross-product, but more done only in memory

\[
P(R) + T(R) \times P(S) + \text{OUT}
\]

\[
\text{BNLJ is faster by roughly } \frac{(B-1)T(R)}{P(R)}
\]
BNLJ vs. NLJ: Benefits of IO Aware

• Example:
  - R: 500 pages
  - S: 1000 pages
  - 100 tuples / page
  - We have 12 pages of memory \((B = 11)\)

• NLJ: Cost = 500 + 50,000*1000 = 50 Million IOs \(\sim 140\) hours

• BNLJ: Cost = 500 + \(\frac{500*1000}{10}\) = 50 Thousand IOs \(\sim 0.14\) hours

A very real difference from a small change in the algorithm!
Acknowledgement

• Some of the slides in this presentation are taken from the slides provided by the authors.
• Many of these slides are taken from cs145 course offered by Stanford University.