Today:

- Join algorithms
Join Algorithms

A new schema

Sailors(sid, sname, rating, age)
Boats(bid, bname, color)
Reserves(sid, bid, day)

Consider this query

SELECT *
FROM Reserves R, Sailors S
WHERE R.sid = S.sid

• In relational algebra: Reserves $\bowtie_{sid=sid}$ Sailors

Assumptions:
• $M$ pages in $R$
• $P_R$ tuples per page in $R$
• $N$ pages in $S$
• $P_S$ tuples per page in $S$
Simple Nested-Loop Join

1. foreach tuple \( r \in R \)
2. foreach tuple \( s \in S \)
3. if \( r.sid == s.sid \)
4. add \(< r, s >\) to result

- We call \( R \) the “outer” relation
- We call \( S \) the “inner” relation

How does it work?

- For each tuple in the outer relation \( R \)
- We **scan** the entire inner relation \( S \) tuple-by-tuple

Q: What does it cost?

- Assume: \(^1\)
  - \( M = 1000 \) pages (in \( R \)), \( P_R = 100 \) tuples/page
  - \( N = 500 \) pages (in \( S \)), \( P_S = 80 \) tuples/page
  - 100 I/Os per second
- What is the cost of \( R \bowtie S \)?
  - Read all pages of \( R \) and for each \( R \) tuple, read in each page of \( S \)
  - \( M + (P_R \times M) \times N = 1000 + 100 \times 1000 \times 500 \) I/Os
  - \( 50,001,000 \) I/Os = 500,010 seconds \( \approx 5.8 \) days!

\(^1\)if a page is 1 Kb, then \( M \) is 1 Mb ... with 100 tuples per page though, this is only 10 bytes per tuple
Nested-loop Join in pictures ...

Table 1 on Disk

| 2, ... |
| 6, ... |
| 3, ... |

| 1, ... |
| 5, ... |
| 9, ... |

Memory Buffers:

Table 2 on Disk

| ... 2 |
| ... 7 |

| ... 6 |
| ... 9 |

| ... 1 |
| ... 5 |

Table 1 on Disk

| 2, ... |
| 6, ... |
| 3, ... |

| 1, ... |
| 5, ... |
| 9, ... |

Memory Buffers:

Table 2 on Disk

| ... 2 |
| ... 7 |

| ... 6 |
| ... 9 |

| ... 1 |
| ... 5 |

Query Answer:

MATCH!

2, ... ... 2
### Table 1 on Disk

<table>
<thead>
<tr>
<th>2, …</th>
<th>6, …</th>
<th>3, …</th>
</tr>
</thead>
<tbody>
<tr>
<td>1, …</td>
<td>5, …</td>
<td>9, …</td>
</tr>
</tbody>
</table>

### Memory Buffers:

- NO MATCH
- Discard!

### Table 2 on Disk

| …, 2 |
| …, 7 |

| …, 6 |
| …, 9 |

| …, 1 |
| …, 5 |

### Query Answer:

2, … 2

---

- And so on ...
Page-Oriented Nested-Loop Join

1. foreach page of tuples in \( R \)
2. foreach page of tuples in \( S \)
3. foreach record \( r \) and \( s \) in memory
4. if \( r.sid == s.sid \)
5. add \( < r, s > \) to result

Q: What does it cost?

- Again assume:
  - \( M = 1000 \) pages (in \( R \)), \( P_R = 100 \) tuples/page
  - \( N = 500 \) pages (in \( S \)), \( P_S = 80 \) tuples/page
  - 100 I/Os per second

- What is the cost of \( R \bowtie S \)?
  - \( M + M \times N = 1000 + 1000 \times 500 \) I/Os
  - \( 501,000 \) I/Os = 5,010 seconds \( \approx 1.4 \) hours!

- What is the cost of \( S \bowtie R \)?
  - \( N + N \times M = 500 + 500 \times 1000 = 500,500 \) I/Os
  - We typically will use smaller relation as the outer relation

- What does nested-loop join cost for \( S \bowtie R \)?
  - \( N + (P_S \times N) \times M = 500 + (80 \times 500) \times 1000 = 40,000,500 \) I/Os
  - So about 400,005 seconds (or about 4.6 days)
Page-oriented nested-loop join In pictures ...

• load a page from table 1 and a page from table 2
• try all combinations of records

• continue using another page from table 2
• and so on until all page combinations tried
“Block” nested-loop join: Use more buffers

1. assuming $B$ pages of memory in buffer
2. assign one page of buffer to output
3. load $B - 2$ pages of tuples from $R$
4. load 1 page of tuples from $S$
5. foreach record $r$ and $s$ in memory
   6. if $r.sid == s.sid$
   7. add $< r, s >$ to result

Q: What does it cost?

- Again assume:
  - $M = 1000$ pages (in $R$), $P_R = 100$ tuples/page
  - $N = 500$ pages (in $S$), $P_S = 80$ tuples/page
  - 100 I/Os per second

- Also assume:
  - $B = 52$

- What is the cost of $R \bowtie S$?
  - $M + (M/(B - 2)) \times N = 1000 + (1000/50) \times 500$ I/Os
  - 11,000 I/Os = 110 seconds $\approx 1.8$ minutes!

- What is the cost of $S \bowtie R$?
  - $N + (N/(B - 2)) \times M = 500 + (500/50) \times 1000$ I/Os
  - 10,500 I/Os = 105 seconds $\approx 1.75$ minutes!
Indexed nested-loop join

1. assuming there is an index on $S.sid$
2. foreach record $r \in R$
3. find tuples $s \in S$ with matching search key
4. foreach matching $s$
5. add $< r, s >$ to result

Q: What does it cost?

- Again assume:
  - $M = 1000$ pages (in $R$), $P_R = 100$ tuples/page
  - $N = 500$ pages (in $S$), $P_S = 80$ tuples/page
  - 100 I/Os per second

- Also assume:
  - it costs 3 I/Os to find matching tuples in the index

- What is the cost of $R \bowtie S$?
  - $M + (M \times P_R) \times 3 = 1000 + (1000 \times 100) \times 3 = 3,010$ I/Os
  - 301,000 I/Os $= 3,010$ seconds $\approx 50$ minutes!

- What is the cost of $S \bowtie R$?
  - $N + (N \times P_S) \times 3 = 500 + (500 \times 80) \times 3 = 1205$ I/Os
  - 120,500 I/Os $= 1205$ seconds $\approx 20$ minutes!

- again, pick the smaller outer relation (fewest records)
Merge Join

If each relation is sorted on the join attributes:

- Cost of joining $R$ and $S$ can be reduced to $M + N!$ ... best case
  1. compare 1st record in $R$ with 1st record in $S$
  2. if they match, output $<r,s>$
  3. otherwise discard smallest and repeat

What if the relations aren’t sorted on the join attributes?

- **Challenge**: The tables do not fit into memory!
- **Solution**: Use external sorting!

Q: How else can we sort? ... using a (clustered) B+ Tree
Q: What if the B+ Tree is unclustered? ... can incur many I/Os to sort
  – in fact, up to 1 I/O per record!

- We’ll see other operator implementations that also use sorting
N-Way External Sorting

Basic idea
- Modify the mergesort algorithm to work over file pages
- Create a “sorted run” (set of small sorted files)
- Do an “N-way merge” of the sorted sub files

Creating a sorted run
1. Read in \( B \) pages into memory
2. Sort data in buffer pages on join attribute (search key)
3. Write result back out to disk

Doing an N-way merge
- In mergesort, we merge two sub-lists at a time
- Here we merge \( N = B - 1 \) pages (sub-lists) at a time
- One buffer page used for output
Example

Assume $B = 4$ (4 buffer pages)

- First sort pass ... create sorted runs
  - load all pages, sort, write out as sorted sub-files
  - each sorted sub-file is at most $B$ pages in size

New File of 3 sorted sub-files
• First **merge** pass ... merge $B - 1$ of the sorted sub files

---

**Buffer**

Load Buffer

In this case 3-way merge

---

**Buffer**

Output 1st Page

In this case 3-way merge

---
In this case 3-way merge

Buffer

Load Buffer Again

In this case 3-way merge

Output 2\textsuperscript{nd} Page

In this case 3-way merge

- And so on until $N$ subfiles are sorted
Additional notes on N-Way External Sorting

- merge may require multiple passes (like in plain-old mergesort)
- each merge pass reduces the number of sub-files by \( B - 1 \)

How much does it cost?

- On each pass:
  - cost is \( 2 \times M \) I/Os (for \( M \) pages in table)
  - that is, must read and write entire file (all pages) in each pass
  - so the question is .... how many passes are needed?
  - note: sometimes \( 2 \times M - 1 \) on a pass (but ignore this case)

- Number of merge passes depends on \( B \) (buffer space available)
  - \( \text{Passes} = \lceil \log_{B-1}(M/B) \rceil \) \( \ldots \) why \( M/B \) ?
  - Can sort 100 million pages in 4 passes w/ 129 pages of memory
  - Can sort \( M \) pages using \( B \) buffer pages in 2 passes if \( \sqrt{M} < B \)
Example

- Assume we want to sort $R$ where:
  - $M = 1000$
  - 100 I/Os per second
  - $B = 52$
  - Total cost: $2 \times 1000 + 2 \times 1000 \times \lceil \log_{51}(1000/52) \rceil = 4000$ I/Os
  - Note we can simplify the formula: $2 \times M \times (\lceil \log_{B-1}(M/B) \rceil + 1)$
Merge Join (cont.)

Basic idea:

1. Sort $R$ on join attribute (if not already sorted)
2. Sort $S$ on join attribute (if not already sorted)
3. Merge $R$ and $S$ (to perform join)
4. scan each $R$ until an $r$-tuple $\geq$ current $s$-tuple
5. if $r$-tuple == $s$-tuple, output match
6. then scan $S$ until an $s$-tuple $\geq$ current $r$-tuple
7. if $s$-tuple == $r$-tuple, output match
8. Continue until run out of tuples in $R$ or $S$

Outer relation $R$ is scanned once

- each time an $r$-tuple matches first $s$-tuple
- we form a "group" of $s$-tuples that match $r$
- each such group is scanned once per matching $r$-tuple
- either:
  - this group fits into memory (the scan is “free”)
  - or we have extra page I/Os (to reread the group)
Cost of Merge Join

Best-case cost (all matches in memory)

- (cost to sort $R$) + (cost to sort $S$) + ($M + N$)

Worst-case cost (all $R$ and $S$ have same value)

- matching group is the entire $S$ relation
- (cost to sort $R$) + (cost to sort $S$) + $M + M \times N$

- which is worse than page-oriented nested-loop (must sort $R$ and $S$)

Example

- Assume:
  - $M = 1000$ pages (in $R$), $P_R = 100$ tuples/page, unsorted
  - $N = 500$ pages (in $S$), $P_S = 80$ tuples/page, unsorted
  - 100 I/Os per second
  - $B = 52$

- What is the cost of $R \bowtie S$ (best case)
  - Sorting $R$: $4000$ I/Os
  - Sorting $S$: $2 \times 500 \times ([\log_{51}(500/52)] + 1) = 2000$
  - Total cost: $4000 + 2000 + 1000 + 500 = 7500$ page I/Os
  - **1.25 minutes**!
Hash Join

Hash indexes: Basic idea

- A hash function $h$ maps keys $k$ to pages
- So, data entry $k^*$ is found on page $h(k)$
- Number of primary pages (buckets) fixed ... in static hashing
  - allocated sequentially
  - not de-allocated (don’t grow and shrink)
  - overflow pages (buckets) are used as needed
- Large overflow buckets can degrade performance
  - based on search key values, hash function, number of primary buckets
Hash Join: Simple case

- Entire $S$ table fits into memory
- Build an in-memory hash index for $S$ ("build" phase)
- Scan $R$ and find matching $S$-records ("probe" phase)
  - identical to indexed nested-loop join

Q: What is the cost?
  - Cost to read $R$ (outer relation)
  - Cost to read $S$ (build phase of inner relation)
  - Each time we read a page in $R$ we find all matches with $S$ (in memory)
  - So total cost is only $M + N$ ... no overflow buckets

This is too easy!

- What do we do if $S$ does not fit in memory?
  - We "partition" $S$ into smaller sets of records that fit in memory
  - Do simple case (indexed nested-loop join) on partitions
Hash Join: Harder case

- Define a hash function \( h \) that can be used to partition \( R \) and \( S \)
- Each \( S \) partition should be small enough to fit into main memory
- Apply \( h \) to \( R \) and \( S \) and store each resulting partition in a file
- Do the simple case on each pair of matching partitions (files)
Q: What is the cost?

- $2 \times M$ to partition $R$ (read and write)
- $2 \times N$ to partition $S$ (read and write)
- Cost to join partitions: $M + N$
- Total cost: $3 \times (M + N)$

• In our example:
  - $3 \times (1000 + 500) = 4500$ I/Os $\approx 45$ seconds
Merge Join vs. Hash Join

Merge Join

- less sensitive to data “skew” (clusters of similar values)
- result is sorted (more later)

Hash Join

- highly parallelizable (join partitions concurrently)

For inequality conditions (e.g., \( r1.age < r2.age \))

- neither hash or merge join applicable
- block nested-loop likely best approach
- index nested-loop also possible (B+ tree)
Summary of Join Algorithms

And the winner is ... (for $R \bowtie S$)

<table>
<thead>
<tr>
<th>Join Algorithm</th>
<th>I/Os</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simple nested-loop</td>
<td>50,001,000</td>
<td>5.8 days</td>
</tr>
<tr>
<td>Page-oriented nested-loop</td>
<td>501,000</td>
<td>1.4 hours</td>
</tr>
<tr>
<td>Block nested-loop</td>
<td>11,000</td>
<td>1.8 minutes</td>
</tr>
<tr>
<td>Index nested-loop</td>
<td>301,000</td>
<td>50 minutes</td>
</tr>
<tr>
<td>Merge join</td>
<td>7,500</td>
<td>1.25 minutes</td>
</tr>
<tr>
<td>Hash join</td>
<td>4,500</td>
<td>45 seconds</td>
</tr>
</tbody>
</table>