L09: Join Algorithms

Dr. Kenneth LEUNG

Department of Computer Science and Engineering
The Hong Kong University of Science and Technology
Hong Kong SAR, China
Several different algorithms to implement joins

Choice based on cost estimate. We only take into account the I/O operations (reads and writes of pages)

Terminology:
- $r$, $s$ relations to be joined
- $n_r$, $n_s$ number of records in $r$, $s$
- $b_r$, $b_s$ number of pages in $r$, $s$
- $M$ available memory in pages

Examples assume equi-joins on the following tables
- Number of records of $Customer$: 10,000 $Depositor$: 5000
- Number of pages of $Customer$: 400 $Depositor$: 100
- The join attribute is the $customer-name$, which is the key of $Customer$
We wish to compute $r \Join s$

$r$ is called the **outer relation** and $s$ the **inner relation** of the join.

Block nested loop join requires no indices and can be used with any kind of join condition.

\[
\begin{align*}
\text{for each block } B_r \text{ of } r \\
\text{for each block } B_s \text{ of } s \\
\text{for each tuple } t_r \text{ in } B_r \\
\text{for each tuple } t_s \text{ in } B_s \\
\text{if } (t_r, t_s) \text{ satisfies the join condition} \\
\text{add } (t_r, t_s) \text{ to the result}
\end{align*}
\]
**Worst case estimate:** \( b_r \times b_s + b_r \) page accesses
- One read for each page in the outer relation \( r \)
- Each page in the inner relation \( s \) is read once for each page in the outer relation

**Best case:** \( b_r + b_s \) page accesses (the inner relation fits in memory)

**Improvements** to block nested loop:
- In block nested-loop, use \( M - 2 \) disk pages as blocking unit for the outer relation, where \( M = \) memory size in pages; use remaining two pages to buffer the inner relation and the output
  - Cost = \( \left\lceil \frac{b_r}{M - 2} \right\rceil \times b_s + b_r \)

**Optimizations**:
- If equi-join attribute forms a key in inner relation, stop inner loop on first match
- Scan inner loop forward and backward alternately, to make use of the pages remaining in buffer (with LRU replacement)
• Compute *Depositor* JOIN *Customer*, with *Depositor* as the outer relation and *Customer* as the inner relation
  - Number of pages of $b_{Depositor} = 100$, $b_{Customer} = 400$
• Worst case cost of block nested-loop join
  - $100 \times 400 + 100 = 40,100$ page accesses
  - How many main memory pages you need to apply block nested-loop?
• Best case cost of block nested-loop join
  - $100 + 400 = 500$ page accesses
  - How many main memory pages you need to achieve this cost?
• Worst case cost of block nested loops join with 52 main memory pages
  - $2 \times 400 + 100 = 900$ page accesses
Indexed Nested-Loop Join

- Index lookups can replace file scans if
  - the join is an equi-join or natural join and
  - an index is available on the inner relation’s join attribute
  - Can construct an index just to compute a join
- For each tuple $t_r$ in the outer relation $r$, use the index to look up tuples in $s$ that satisfy the join condition with tuple $t_r$
- **Cost:** $b_r + n_r * c$
  - where $c$ is the cost of traversing the index and fetching all matching $s$ tuples for one tuple of $r$
  - $c$ can be estimated as the cost of a single selection on $s$ using the join condition
- If indices are available on join attributes of both $r$ and $s$, use the relation with the fewest tuples as the outer relation
Example of Indexed Nested-Loop Join Costs

- Compute *Depositor JOIN Customer*, with *Depositor* as the outer relation and *Customer* as the inner relation
- Let *Customer* have a primary B+-tree index with 4 levels on the join attribute *customer-name* (which is the primary key of *Customer*)
- Number of pages $b_{Depositor} = 100$
- Number of records $n_{Depositor} = 5000$
- Cost of indexed nested loops join
  - $100 + 5000 \times 5 = \textbf{25,100}$ disk accesses.
- CPU cost likely to be less than that for block nested loops join
- Indexed Nested-Loop is the best algorithm if there are selective conditions on the outer relation
• **Sort** both relations on their join attribute (if not already sorted on the join attributes)

• **Merge** the sorted relations to join them
  - Join step is similar to the merge stage of the sort-merge algorithm
  - Main difference is handling of duplicate values in join attribute — every pair with same value on join attribute must be matched
• Can be used only for equi-joins and natural joins

• Each block needs to be read only once (assuming all tuples for any given value of the join attributes fit in memory)

• Thus, number of page accesses for merge-join is 
  \( b_r + b_s + \text{the cost of sorting} \) (if relations are unsorted)
• Applicable for equi-joins and natural joins

• A hash function $h$ is used to partition tuples of both relations into $n$ buckets (i.e., a hash file organization)

• $h$ maps $JoinAttrs$ values to $\{0, 1, \ldots, n-1\}$, where $JoinAttrs$ denotes the common attributes of $r$ and $s$ used in the natural join

  $r_0, r_1, \ldots, r_{n-1}$ denote partitions of $r$ tuples
  • Each tuple $t_r \in r$ is in partition $r_i$ where $i = h(t_r[JoinAttrs])$

  $s_0, s_1, \ldots, s_{n-1}$ denote partitions of $s$ tuples
  • Each tuple $t_s \in s$ is in partition $s_i$ where $i = h(t_s[JoinAttrs])$
Hash-Join (cont.)
• \( r \) tuples in bucket/partition \( r_i \) need only to be compared with \( s \) tuples in \( s_i \)

• Need not be compared with \( s \) tuples in any other partition, since:
  - an \( r \) tuple and an \( s \) tuple that satisfy the join condition will have the same value for the join attributes
  - if that value is hashed to some value \( i \), the \( r \) tuple has to be in \( r_i \) and the \( s \) tuple in \( s_i \)
1. Partition the relation $r$ using hashing function $h$. When partitioning a relation, one page of memory is reserved as the output buffer for each bucket, plus one for the input.

2. Partition $s$ similarly.

3. For each $i$:
   - Load bucket $r_i$ into memory and build an in-memory hash index on it using the join attribute. This hash index uses a different hash function than the earlier one $h$. Relation $r$ is called the **build input**.
   - Read the tuples in bucket $s_i$ from the disk page by page. For each tuple $t_s$ locate each matching tuple $t_r$ in $r_i$ using the in-memory hash index. Relation $s$ is called the **probe input**.
During the partitioning phase, we have \( M \geq n+1 \), because we need one output page for each partition, plus one input buffer.

Also, the number of buckets \( n \) is such that each bucket of the build input \( r \) should fit in the available main memory pages \( M \). Assuming each bucket has the same size:

\[
M \geq \lceil \frac{b_r}{n} \rceil + 2 \quad \text{(one input and one output buffer page)}
\]

In order to satisfy these conditions: approx. \( M > \sqrt{b_r} \)
- The probe relation partitions need not fit in memory

Recursive partitioning required if number of partitions \( n \) is greater than number of pages \( M \) of memory
- Rarely necessary: e.g., recursive partitioning not needed for relations of 1GB or less with memory size 2MB and page size 4KB
Example of Hash-Join Costs

- Assume that memory size is $M = 25$ pages, $b_{Depositor} = 100$ and $b_{Customer} = 400$
- $Depositor$ is the build input
  - Partition $Depositor$ into 5 buckets, each of size 20 pages. This partitioning can be done in one pass
- $Customer$ is the probe input
  - Partition $Customer$ into 5 buckets, each of size 80 pages. This is also done in one pass
- Read each bucket in turn of the build input in memory, and probe against records of the corresponding probe bucket
- Therefore, total cost: $3 \times (100 + 400) = 1500$ page transfers
  - ignores cost of writing partially filled pages
If the memory is large enough, we can keep one or more buckets of one file in memory at all times

- Let’s say that we have 10 buckets and that each bucket is 90 pages. If we have 100 main memory pages, when we partition the build input $r$ we keep the entire first bucket in memory and allocate 9 pages for the remaining buckets and 1 for reading the file page by page.

- When we read the probe input $s$, we use again 10 buckets and the same hash function. If a record falls in the first bucket, we produce immediately results since we have the first bucket of $r$ (90 pages) in memory.

- In this way, we avoid writing and reading back the first buckets of both $r$ and $s$.

- If we have more memory, we can keep more buckets.

- It is better to partition the smallest file first (i.e., make it the build input) since it has smaller buckets and we may be able to keep more in memory.