CSE462/562: Database Systems (Spring 23)

Lecture 17: Join algorithms

4/13/2023 & 4/18/2023
Joins

• Joins are very common
  • need to reconstruct complete rows due to schema normalization
  • collecting correlated data (e.g., sliding window on timestamps, spatial joins, etc.)

• Joins are very expensive!
  • join results can be as large as the cartesian product
  • but they are usually far from the full cartesian product
    • can we avoid evaluating the full cartesian product?

• Many approaches to reduce join cost
  • Nested-loop join (simple/block/indexed)
  • Sort-merge join
  • Hash join (basic hash partitioning vs hybrid hashing)
Running example

• A quick recap on our running example

• Notations: for relation $R$
  • $T_R$: number of records, $N_R$: number of pages in its heap file, $B_R$: (average) number of tuples per page
  • $h_I$: height of a B-tree index $I$ over the file
  • $M$: private workspace size in pages

• Running example
  • $t_S = 4 \text{ms}$, $t_T = 0.1 \text{ms}$, 4000-byte page
  • Student: $R(\text{sid}: \text{int}, \text{name}: \text{varchar(19)}, \text{login}: \text{varchar(19)}, \text{major}: \text{char(2)}, \text{adm_year}: \text{int})$
    • 50 bytes/tuple, $B_R = 80$, $T_R = 40,000$, $N_R = 500$
  • Enrollment: $E(\text{sid}: \text{int}, \text{semester}: \text{char(3)}, \text{cno}: \text{int}, \text{grade}: \text{double})$
    • 20 bytes/tuple, $B_E = 200$, $T_E = 200,000$, $N_E = 1000$
  • Consider the equi-join $R \bowtie_{R.\text{sid}=E.\text{sid}} E$ (denote the join predicate $R.\text{sid} = E.\text{sid}$ as $\theta$)
    • $R$ is called the outer relation, $E$ is called the inner relation
    • cost = $\#\text{seeks} \times t_S + \#\text{page\_transfers} \times t_T$
    • ignoring buffer effect; not counting the final output
Simple nested-loop join

- For each tuple in the outer relation $R$,
  - scan the entire inner relation $S$

```plaintext
foreach tuple $r$ in $R$ do
  foreach tuple $e$ in $E$ do
    if $(r, e)$ satisfies $\theta$ then
      emit $r \circ e$ as result
```

- Simple nested-loop join evaluates the full cartesian product
  - only keep those pairs that satisfy the predicate

- Cost? depends on the available memory
  - If $M = 2$, we’ll have to read every pages in the inner relation once for every tuple in the outer relation
    - number of pages to read: $N_R + T_R N_E$
    - number of seeks: $N_R + T_R$ (one seek for every page in $R$, and one seek for every scan of $E$)
    - cost = $t_T(N_R + T_R N_E) + t_S(N_R + T_R)$
    - running example: $\text{cost}(R \bowtie E) \approx 4162 s \approx 1.15 hr$ !
      - What about $\text{cost}(E \bowtie R)$ ?
        - $t_T(N_E + T_E N_R) + t_S(N_E + T_E) \approx 10804 s \approx 3 hr$

$\theta: R.sid = E.sid$
Simple nested-loop join

• For each tuple in the outer relation $R$,
  • scan the entire inner relation $E$

  ```
  foreach tuple r in R do
    foreach tuple e in E do
      if $(r, e)$ satisfies $\theta$ then
        emit $r \circ e$ as result
  ```

• Simple nested-loop join evaluates the full cartesian product
  • only keep those pairs that satisfy the predicate

• Cost? depends on the available memory
  • If $M = 2$, cost = $t_T(N_R + T_R N_E) + t_S(N_R + T_R)$
  • If $M \geq N_E + 2$, we can cache the inner relation $E$ in memory
    • number of pages to read: $N_R + N_E$
    • number of seeks: 2 (scanning $E$ in full, followed by scan of $R$)
    • cost = $t_T(N_R + N_E) + 2t_S = 0.158 s$

• How to fully utilize the memory if $3 \leq M < N_E + 2$?

$\theta : R.sid = E.sid$
Block nested-loop join

- For each block for the outer relation $S$ and every block of the inner relation $E$,
  - first assume each block is a page
  - emit the pairs of records $(r, e)$ that satisfy the join predicate $\theta$

```plaintext
foreach block $B_R$ in R do
  foreach block $B_E$ in E do
    foreach tuple $r$ in $B_R$ do
      foreach tuple $e$ in $B_E$ do
        if $(r, e)$ satisfies $\theta$ then
          emit $r \circ e$ as result
```

- Block nested-loops only reads each page in the outer relation once
  - Cost = $t_T(N_R + N_R N_E) + 2 t_S N_R = 54.5$ s (block nested-loop) vs 1.15 hr (simple nested loop)
    - What about $E \bowtie S$?
      - cost = $58.1$ s -- use smaller relation as the outer relation
Block nested-loop join

• For each block for the outer relation $S$ and every block of the inner relation $E$,
  • first assume each block is a page
  • emit the pairs of records $(r, e)$ that satisfy the join predicate $\theta$

```c
foreach block $B_R$ in R do
  foreach block $B_E$ in E do
    foreach tuple $r$ in $B_R$ do
      foreach tuple $e$ in $B_E$ do
        if $(r, e)$ satisfies $\theta$ then
          emit $r \circ e$ as result
```

• Block nested-loops only reads each page in the outer relation once
  • Cost = $t_T(N_R + N_R N_E) + 2t_SN_R = 54.5$ s (block nested-loop) vs 1.15 hr (simple nested loop)
  • Only uses 3 buffer frames. What about $M > 3$ buffer frames?
    • Read every $M - 2$ pages at a time for the outer relation, i.e., $|B_S| = M - 2$
      • cost = $t_T\left( N_R + \left\lceil \frac{N_R}{M-2} \right\rceil N_E \right) + 2t_SN_R \left\lceil \frac{N_R}{M-2} \right\rceil$
      • $M = 12 \Rightarrow$ cost = 5.45 s, $M = 102 \Rightarrow$ cost = 0.59 s
    • caveat: CPU cost may not be negligible when I/O cost is low for NL/BNL

$\theta$: $R.sid = E.sid$
Index nested-loop join

- If there’s an index over the inner relation’s join attribute (e.g., $E.sid$)
  - only fetch records with matching values in the join attribute using the index

$$\text{cost} = N_R(t_S + t_T) + T_R \times c$$
  - where $c$ is the average time for scanning all the matching record for a tuple $r \in R$
  - $c$ depends on
    - selectivity $s_E$ or join degree $d = s_E T_E$
      - special case foreign-key join: $d = 1$ or $s_E = 1/T_E$
    - clustered vs unclustered index
    - data entry alternatives

```
foreach block $B_R$ in R do
  foreach tuple $r$ in $B_R$ do
    foreach tuple $e$ in $B_E$ s.t. $R.sid = E.sid$ do
      emit $r \circ e$ as result
```
Index nested-loop join

- \( R \bowtie_{R.sid=E.sid} E \)
  - BNL cost = 54.5 s

- Example 1: \( E \) as inner, B-Tree index over \( E(sid) \), alternative 2, clustered, height \( h = 3 \)
  - assuming uniformity, average join degree \( d = \frac{TE}{TR} = 5 \)
  - for each inner table scan, \( h \) random I/Os for tree search, 1 seek and \( \left\lfloor \frac{d}{BE} \right\rfloor = 1 \) heap pages read
    - \( c = h(t_S + t_T) + t_S + \left\lfloor \frac{d}{BE} \right\rfloor t_T = 16.1 \) ms
    - total = \( N_R(t_S + t_T) + TR \times c = 646.05 \) s

- Example 2: \( E \) as inner, B-Tree index over \( E(sid) \), alternative 2, unclustered, height \( h = 3 \)
  - still \( d = 5 \)
  - for each inner table scan, \( h \) random I/Os for tree search, 5 random I/Os for reading 5 heap records
    - \( c = h(t_S + t_T) + d(t_S + t_T) = 32.8 \) ms
    - total = \( N_R(t_S + t_T) + TR \times c = 1314.05 \) s
Index nested-loop join

- Now consider $\sigma_{adm\_year=2021} R \bowtie_{R.sid=E.sid} E$, assuming selectivity of $adm\_year = 2021$ is $s = 0.001$
  - suppose we have an unclustered B-Tree index over $R(adm\_year)$, $h_1 = 2$
    - can use the index to find all the $[sT_R] = 40$ records
    - Using nested loop for join, need to scan the inner for every $s \in \sigma_{adm\_year=2021}$
      - cost = $(h_1 + [sT_R])(t_s + t_T) + [sT_R](t_s + t_T N_E) \approx 4.33s$

- Example 3: $E$ as inner, B-Tree index over $E(sid)$, alternative 2, clustered, height $h = 3$, $d = 5$
  - for each inner table scan, $h$ random I/Os for tree search, 1 seek and $\left\lfloor \frac{d}{B_E} \right\rfloor = 1$ heap pages read
    - $c = h(t_s + t_T) + t_s + \left\lfloor \frac{d}{B_E} \right\rfloor t_T = 16.1 ms$
    - total = $(h_1 + [sT_R])(t_s + t_T) + [sT_R] \times c \approx 0.82s$

- Example 4: $E$ as inner, B-Tree index over $E(sid)$, alternative 2, unclustered, height $h = 3$, $d = 5$
  - for each inner table scan, $h$ random I/Os for tree search, 5 random I/Os for reading 5 heap records
    - $c = h(t_s + t_T) + d(t_s + t_T) = 32.8 ms$
    - total = $(h + [sT_R])(t_s + t_T) + [sT_R] \times c \approx 1.48s$
Sort-merge join

- Idea: sort $R$ on $R.sid$ and sort $E$ on $E.sid$
  “merge” them and emit the pairs with matching values on the join columns
- Useful if
  - One or both relations are already sorted on the join attributes
    - If not, sort them using external sorting algorithms – this may still be cheaper than BNL
  - Output should be sorted on the join attributes
    - e.g., SELECT * from $R$, $E$ WHERE $R.sid = E.sid$ ORDER BY $R.sid$
- Algorithm sketch:
  - Naïve version:
    ```
    pr = address of first tuple in R
    pe = address of first tuple in E
    done = false
    while (not done && pe != end && pr != end) do
      if (pe->sid != pr->sid)
        if pe->sid < pr->sid then ++pe else ++pr
        continue
      pr2 = first address after pr such that pr2 == end || pr2->sid != pr->sid
      pe2 = first address after pe such that pe2 == end || pe2->sid != pe->sid
      emit all pairs between [pr, pr2) and [pe, pe2)
      pe = pe2; pr = pr2;
    ```
Sort-merge join

- Sort-merge join: naïve version
  - Problem?

\[ R \bowtie_{R.sid=E.sid} E \]

<table>
<thead>
<tr>
<th>sid</th>
<th>name</th>
<th>login</th>
<th>major</th>
<th>adm_year</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>Alice</td>
<td>alicer34</td>
<td>CS</td>
<td>2021</td>
</tr>
<tr>
<td>101</td>
<td>Bob</td>
<td>bob5</td>
<td>CE</td>
<td>2020</td>
</tr>
<tr>
<td>102</td>
<td>Charlie</td>
<td>charlie7</td>
<td>CS</td>
<td>2021</td>
</tr>
<tr>
<td>103</td>
<td>David</td>
<td>davel</td>
<td>CS</td>
<td>2020</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>sid</th>
<th>semester</th>
<th>cno</th>
<th>grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>s22</td>
<td>562</td>
<td>2.0</td>
</tr>
<tr>
<td>100</td>
<td>f21</td>
<td>560</td>
<td>3.7</td>
</tr>
<tr>
<td>101</td>
<td>s21</td>
<td>560</td>
<td>3.3</td>
</tr>
<tr>
<td>101</td>
<td>f21</td>
<td>560</td>
<td>3.3</td>
</tr>
<tr>
<td>102</td>
<td>s22</td>
<td>562</td>
<td>2.3</td>
</tr>
<tr>
<td>102</td>
<td>f21</td>
<td>560</td>
<td>4.0</td>
</tr>
<tr>
<td>103</td>
<td>s22</td>
<td>460</td>
<td>2.7</td>
</tr>
<tr>
<td>103</td>
<td>f21</td>
<td>250</td>
<td>4.0</td>
</tr>
</tbody>
</table>
Sort-merge join

- Sort-merge join: naïve version
  - Problem? each matched group is scanned for an additional pass

### student R

<table>
<thead>
<tr>
<th>sid</th>
<th>name</th>
<th>login</th>
<th>major</th>
<th>adm_year</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>Alice</td>
<td>alicer34</td>
<td>CS</td>
<td>2021</td>
</tr>
<tr>
<td>101</td>
<td>Bob</td>
<td>bob5</td>
<td>CE</td>
<td>2020</td>
</tr>
<tr>
<td>102</td>
<td>Charlie</td>
<td>charlie7</td>
<td>CS</td>
<td>2021</td>
</tr>
<tr>
<td>103</td>
<td>David</td>
<td>davel</td>
<td>CS</td>
<td>2020</td>
</tr>
</tbody>
</table>

### enrollment E

<table>
<thead>
<tr>
<th>sid</th>
<th>semester</th>
<th>cno</th>
<th>grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>s22</td>
<td>562</td>
<td>2.0</td>
</tr>
<tr>
<td>100</td>
<td>f21</td>
<td>560</td>
<td>3.7</td>
</tr>
<tr>
<td>101</td>
<td>s21</td>
<td>560</td>
<td>3.3</td>
</tr>
<tr>
<td>101</td>
<td>f21</td>
<td>560</td>
<td>3.3</td>
</tr>
<tr>
<td>102</td>
<td>s22</td>
<td>562</td>
<td>2.3</td>
</tr>
<tr>
<td>102</td>
<td>f21</td>
<td>560</td>
<td>4.0</td>
</tr>
<tr>
<td>103</td>
<td>s22</td>
<td>460</td>
<td>2.7</td>
</tr>
<tr>
<td>103</td>
<td>f21</td>
<td>250</td>
<td>4.0</td>
</tr>
</tbody>
</table>
Sort-merge join

- Idea: sort $R$ on $R.sid$ and sort $E$ on $E.sid$
  “merge” them and emit the pairs with matching values on the join columns

- Algorithm sketch:
  - How to ensure $R$ is scanned once, each $S$ group is scanned once per matching $r \in R$?

```c
pr = address of first tuple in R
pe = address of first tuple in E
done = false
while (not done && pe != end && pr != end) do
  if (*pe != *pr)
    if *pe < *pr then ++pe else ++pr
    continue
  key = pr->sid
  pe0 = pe
  while pr != end && pr->sid == key
    pe = pe0
    while pe != end && pe->sid == key
      emit *pr *pe; ++pe
      pe2 = pe; ++pr
    pe = pe2
```

A few caveats in actual implementation:
1. Need to restructure the algorithm to fit into volcano model (project 6)
2. Rewinding (setting to a previously saved pointer) on iterator may be expensive!
3. Handling NULLs (NULLs never compare equal)
Sort-merge join

• Cost analysis: sorting cost + merge cost, let $M = 110, B = 10, \frac{M}{B} = 11$
  
  • Sorting cost: $2t_T N_R \left( \left\lceil \log_{\frac{M}{B}} \left( \frac{N_R}{M} \right) \right\rceil + 1 \right) + 2t_S \left( \left\lceil \frac{N_R}{M} \right\rceil + \left\lfloor \frac{N_R}{B} \right\rfloor \left\lceil \log_{\frac{M}{B}} \left( \frac{N_R}{M} \right) \right\rceil \right) + 2t_T N_E \left( \left\lceil \log_{\frac{M}{B}} \left( \frac{N_E}{M} \right) \right\rceil + 1 \right) + 2t_S \left( \left\lceil \frac{N_E}{M} \right\rceil + \left\lfloor \frac{N_E}{B} \right\rfloor \left\lceil \log_{\frac{M}{B}} \left( \frac{N_E}{M} \right) \right\rceil \right)$
    
    • includes the cost of writing the sort results to two temporary files
    • running example: sorting cost = 0.64 + 1.28 $s$ = 1.92 $s$
  
  • Merge cost: two scans over the temporary files
    • number of pages read: $N_R + N_E = 1500$ (assuming all 5 matching tuples of $S$ are on the same page)
      • This could be up to $N_R + N_R N_E$ in extreme case (why?)
    • number of seeks? (depending on the block size)
      • If we fetch one page from $R$ and $E$ at a time, then $N_R + N_E = 1500$
      • If we fetch $b = \left\lceil \frac{M}{2} - 1 \right\rceil = 54$ pages at a time for both, then $\left\lceil \frac{N_R}{b} \right\rceil + \left\lceil \frac{N_E}{b} \right\rceil = 10 + 19 = 29$
      • running example: cost = 6 $s$ (one page at a time) or 0.112 $s$ (54 pages at a time)
    • Total cost: $\approx 7.92$ $s$ (one page at a time) or 2.03 $s$ (54 pages at a time)
Sort-merge join

• In practice, the cost of sort-merge join for an equi-join is usually linear to the relation sizes
  • assuming we have a large enough buffer for sorting everything in two passes
  • can even combine the merge phase of external sorting with the merge phase in sort-merge join (i.e., pipelining)

• Question: how large the tables can be in order to complete the sort-merge join in two passes? (minimal needed for sort-merge joins)
  • For simplicity, let \( B = 1 \)
  • Let \( N = \max(N_R, N_E) \), we need \( \log_{M-1} \left\lceil \frac{N}{M} \right\rceil \leq 1 \) => roughly \( N \leq M^2 - M \)
  • In other words, to perform a sort-merge join in two passes
    • the buffer size \( M \geq 0.5 + \sqrt{N} + 0.25 = O(\sqrt{N}) \)
      • good enough to use \( \sqrt{N} + c \) for some small constant \( c \) in practice

• Exercise: \( B > 1 \)?
Hash join

- Idea: build a hash table on outer relation $R$ over its join attribute
  - Scan the outer relation and probe the hash table

  Build a hash table over $R$ with hash function $h_r$
  foreach tuple $e$ in $E$ do
    probe the hash table for all the matching $r$ in $R$
    and emit the join results

- However, the hash table might be too large to fit in memory.
- Extendible hashing/linear hashing have overhead for dynamic updates
  - not suitable for QP purpose

- Solution: partitioning using a hash function $h_p$
Hash join

- Two phases
  - Partitioning
    - Partitioning both outer $R$ and inner $E$ using the same hash function $h_p$
  - Rehashing and probing
    - load a partition for the outer $R$, rehash using a different hash function $h_r$ and build a hash table
    - scan the partition of the outer $E$ with the same hash value for $h_p$ and probe the in-memory hash table
Hash join

• What if a partition won’t fit into memory in the rehashing phase?
  • *Recursive partitioning!*
  • In the rehash and probe phase, if both partitions with the same hash value are larger than \( M - 2 \)
    • recursively partition them as if they were the original relations to be joined
    • use a different partitioning hash function \( h'_p \)

• Assuming there’s no recursive partitioning
  • Cost of partitioning on \( R \) and \( E \):
    \[
    2t_TN_R + 2t_SN_R + 2t_TN_E + 2t_SN_E
    \]
  • can also use larger blocks \( B \) to reduce the number of seeks to \( \frac{2N_R}{B} + \frac{2N_E}{B} \)
  • Cost of rehashing and probing:
    \[
    t_TN_R + t_TN_E + 2t_S\left\lfloor \frac{M}{B} - 1 \right\rfloor \Rightarrow \text{linear to relation sizes}
    \]
  • total cost is roughly the cost of scanning both relations for three times
    • running example: \( M = 100, B = 10 \Rightarrow \text{cost} \approx 1.72 \text{s} \);
    • \( M = 1000, B = 10 \Rightarrow \text{cost} \approx 13.2 \text{s} \) (!)

• How big the outer table can be such that we can finish join in two passes (one partitioning pass)? assuming \( B = 1 \)
  • \( M - 1 \) partitions in Phase 1
  • Each should be no more than \( M-2 \) page large
  • Answer: \((M-2)(M-1)\) – assuming uniformity among the keys
    • i.e., we can do hash join in one pass in about \( O\left(\sqrt{N_R}\right) \) space
  • Much like sorting, but only dependent on the *outer relation size (usually the smaller one)*
    • Do need to use \( c\sqrt{N_R} \) in practice in case of key skews
    • Exercise: \( B > 1 \)?
Hybrid Hashing

- Can we do it better when both relations fit in memory?
  - In-memory hash join can finish in 1 scan instead of 3!
- Hybrid hashing
  - Idea: keep a small 1st partition (of size k) in memory in the partitioning phase
  - directly scan and probe the keys in the 1st partition after partitioning of the inner relation finishes
Hybrid hashing

- Assume we have the hash-partition function \( h_p : X \rightarrow [M - k - 1] \) (\( X \) is the domain of the key, i.e., the join column)

- Define \( h_h \) as follows: (technically, it is determined by the sequence of the keys)
  - \( h_h(x) = 1 \) if in-memory hash table is not yet full
  - \( h_h(x) = 1 \) if \( x \) is already in the hash table
  - \( h_h(x) = h_p(x) + 1 \) otherwise

- This ensures that:
  - Bucket 1 fits in \( k \) pages of memory
  - If the entire set of distinct hash table entries is smaller than \( k \), there is not spilling!

- During partitioning of the outer \( R \)
  - If \( h_h(r.\text{sid}) = 1 \)
    - insert \( r \) into in-mem hash table
  - Otherwise,
    - write \( r \) to its partition

- During partitioning of inner \( E \)
  - If \( h_h(e.\text{sid}) = 1 \)
    - probe in-mem hash table
  - Otherwise,
    - write \( e \) to its partition

- Only enter the rehashing and probing phase if there is any spill
Hybrid hashing

• Assume we have the hash-partition function $h_p: X \rightarrow [M - k - 1]$ (X is the domain of the key, i.e., the join column)

• Define $h_h$ as follows: (technically, it is determined by the sequence of the keys)
  • $h_h(x) = 1$ if in-memory hash table is not yet full
  • $h_h(x) = 1$ if x is already in the hash table
  • $h_h(x) = h_p(x) + 1$ otherwise

• This ensures that:
  • Bucket 1 fits in k pages of memory
  • If the entire set of distinct hash table entries is smaller than k, there is not spilling!

• Running example
  • $M = 1000, k = 900$
  • Cost = $2t_S + t_T(N_R + N_E) \approx 0.15s$
Hashing for single-table ops

- Recursive hashing and hybrid hashing can also be applied to aggregation and deduplication operators
  - Instead of rehashing and probing
  - We only rehash each partition and maintain aggregates/distinct values
  - Cost analysis is similar to hash joins
Summary

• This lecture
  • Join algorithms
    • Nested loop (simple/block/index)
    • Sort-merge join
    • Hash join

• Next lecture
  • Query optimization

• Reminder: Project 6 is released on Apr 18
  • Write-up is due on May 14, 2023, 1:00 AM EDT