Relational Database System Implementation

CS122 – Lecture 7

Winter Term, 2018-2019

Plan-Node Implementations

- Last time, began discussing how to implement all of our relational-algebra plan nodes
- Discussed selection, projection, and grouping/aggregation

Sorting Implementations

- Sorting is very straightforward to implement
- Biggest challenge is when input data-set doesn't fit entirely into memory
- In these cases, use external-memory sorting algorithm
  - Read in runs of tuples that use up to M blocks of buffer space
  - Sort each run in memory, and write it out to a run-file
  - Once all runs are sorted, perform an N-way merge-sort on the runs of data to generate the result

External Sort Algorithm

- Stage 1: Create N sorted runs from an input tuple file, using a max of M buffer pages

  i := 0

  while input file has more blocks:

    read up to M blocks of the input into memory

    sort the in-memory portion of the input

    write sorted results to run-file R_i

    i := i + 1

- *If entire input can be loaded in one shot, we're done!*

External Sort Algorithm (2)

- Stage 2: Merge the *N* sorted runs

  Open all *N* files and read the first block from each file

  do:

  choose the first tuple (in sort order) from all blocks, write it to the output, and advance past that tuple

  if that file's block has no more tuples, read the next block from that file (if more blocks exist)

  while a non-empty block remains for at least one file

External Sort Algorithm (3)

- If input relation is *extremely* large, may not be able to perform merge-sort step in one pass
  - e.g. if there aren't *N* buffer pages to open all *N* run-files
- Simply merge a subset of the run-files into a new larger run-file (and delete the merged run-files)
  - Repeat this process until all remaining run-files can be opened at the same time
  - Final merge-sort pass can produce the output of the sort operation by traversing these run-files

External Sort Algorithm (4)

- Can be other benefits from creating fewer sorted runs
- Example:
  - Could easily sort a file that requires 500 sorted runs…
  - Merging 500 run-files means jumping back and forth between all of these files…
  - Disk seeks can become costly when merging the data!
- Using fewer, larger runs can greatly reduce disk seeks
  - Load more than 1 block of each run-file into memory
  - Rely on read-ahead optimization to pull data from disk

Theta-Join Implementation

- Theta-join plan node is a bit more complicated
- Most simple implementation is *nested-loop join*

```
for t_r in r:
    for t_s in s:
        if pred(t_r, t_s):
            add join(t_r, t_s) to result
```

- Benefits: works for <u>arbitrary</u> predicates!
- Drawbacks: it's <u>very</u> slow

Nested-Loop Join (2)

- How do we extend this to compute $r ⟕_\theta s$?
  - Left outer join
  - t_r is included if it doesn't match any rows in s
- Original algorithm:

for t_r in r:
    for t_s in s:
        if pred(t_r, t_s):
            add join(t_r, t_s) to result

- Updated algorithm:

for t_r in r:
    matched = *false*
    for t_s in s:
        if pred(t_r, t_s):
            matched = *true*
            add join(t_r, t_s) to result
    if not *matched*:
        add padnulls(t_r) to result

Nested-Loop Join (3)

- What about $r ⟕_\theta s$?
  - Right outer join
  - t_s is included if it doesn't match any rows in r
- Original algorithm:

for t_r in r:
    for t_s in s:
        if pred(t_r, t_s):
            add join(t_r, t_s) to result

- Can't easily extend nested-loop algorithm to do right outer join
- But, $r ⟕_\theta s = \Pi_{R,S}(s ⟖_\theta r)$
  - (Must take care to adjust result schema properly)
- Unfortunately, $r ⟗_\theta s$ is similarly out of reach with nested-loop join

Nested-Loop Join (4)

- What about $r \ltimes_\theta s$?
  - Left semijoin
  - t_r is included once, if it matches any row in s
- Original algorithm:

for t_r in r:
    for t_s in s:
        if pred(t_r, t_s):
            add join(t_r, t_s) to result

- Updated algorithm:

for t_r in r:
    for t_s in s:
        if pred(t_r, t_s):
            add t_r to result
            break

- A very simple variant of inner join!

Nested-Loop Join (5)

- What about $r \rhd_\theta s$?
  - Left antijoin
  - t_r is included once, if it matches no rows in s
- Original algorithm:

for t_r in r:
    for t_s in s:
        if pred(t_r, t_s):
            add join(t_r, t_s) to result

- Updated algorithm:

for t_r in r:
    matched = *false*
    for t_s in s:
        if pred(t_r, t_s):
            matched = *true*
            break
    if not *matched*:
        add t_r to result

- Again, very similar to left-outer join

Nested-Loop Join IO Cost

- Nested-loop join:

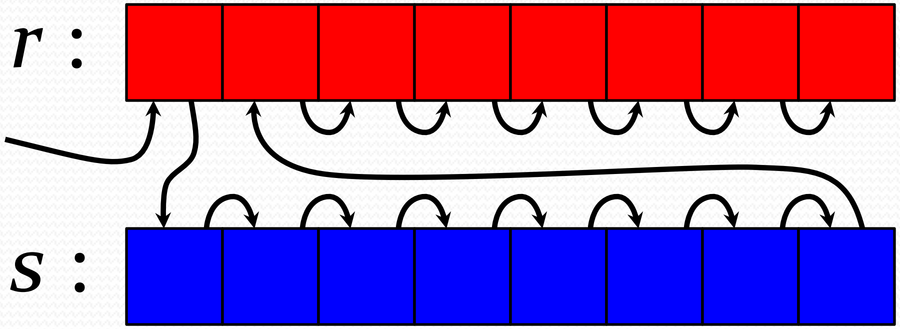
```
for t_r in r:
    for t_s in s:
        if pred(t_r, t_s):
            add join(t_r, t_s) to result
```

- Assume that both *r* and *s* fit entirely within memory
  - b_r is number of blocks in *r*, b_s is number of blocks in *s*
- How many “large” disk seeks are required?
- How many block-reads will this operation perform?

Nested-Loop Join IO Cost (2)

- Nested-loop join:

```
for t_r in r:
    for t_s in s:
        if pred(t_r, t_s):
            add join(t_r, t_s) to result
```



1. (Probably) one large seek to read first tuple in *r*
2. Another large seek when first tuple in *s* is read
3. All of *s* is scanned the first time through the inner loop, and the entire table *s* is cached in the Buffer Manager
4. A third large seek when second block of *r* is read
5. After this, all seeks will be small as *r* is scanned. (Inner loop always reads *s* out of the Buffer Manager.)

- Performs $b_r + b_s$ reads, and 2-3 large seeks total

Nested-Loop Join IO Cost (3)

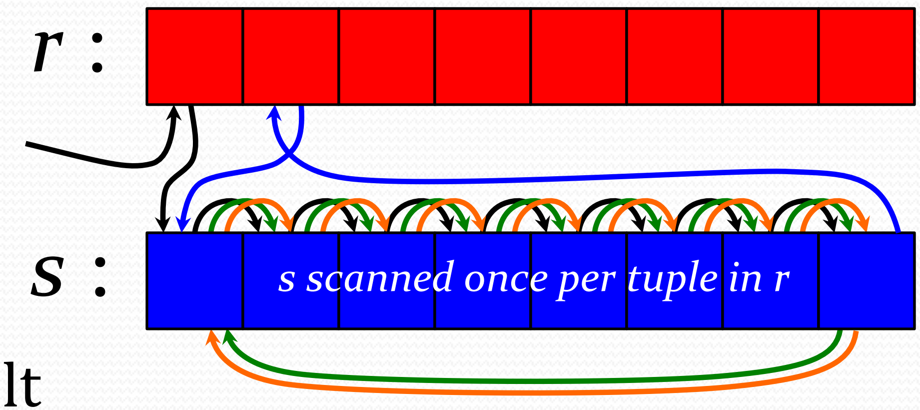
- Nested-loop join:

  for t_r in r:

  for t_s in s:

  if pred(t_r, t_s):

  add join(t_r, t_s) to result

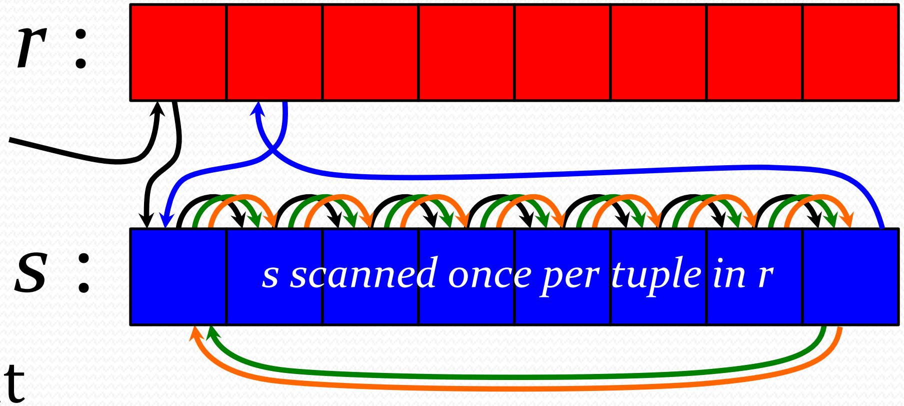


- Worst case: Database can only hold one block of each table in memory. How many block reads are required?
  - Outer loop performs b_r block-reads
  - Inner loop traverses s once *per tuple* in r: $n_r \times b_s$
- Performs $b_r + n_r \times b_s$ block reads

Nested-Loop Join IO Cost (4)

- Nested-loop join:

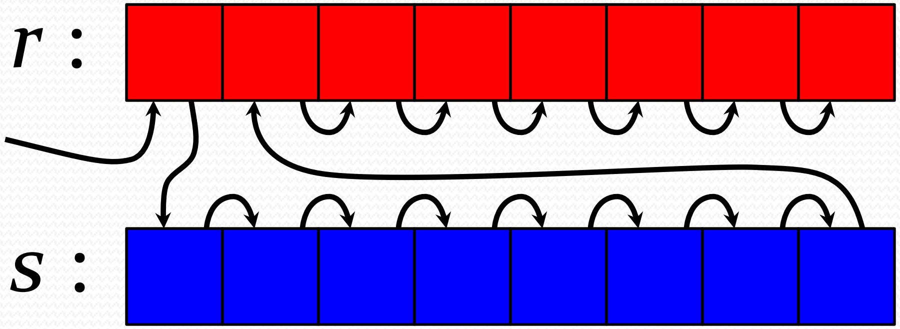
  for t_r in r:
    for t_s in s:
      if pred(t_r, t_s):
        add join(t_r, t_s) to result



- Worst case: Database can only hold one block of each table in memory. How many large seeks are required?
  - Inner loop traverses s sequentially: once per loop = n_r
  - Outer loop traverses r in b_r blocks: b_r total seeks
- Performs $b_r + n_r$ large seeks

Nested-Loop Join IO Cost (5)

- Nested-loop join:
  for t_r in r:
      for t_s in s:
          if pred(t_r, t_s):
              add join(t_r, t_s) to result



- How many reads and seeks if only s fits in memory?
  - s is loaded once, in sequence: 1 seek, b_s reads
  - Outer loop traverses r in b_r blocks: 1-2 seeks, b_r reads
- Performs $b_r + b_s$ reads, and 2-3 seeks total
  - …just like optimal case when both tables fit in memory!
- **If smaller table fits in memory, put it on inner loop.**

Improving Nested-Loop?

- Nested-loop join:

  for t_r in r:

      for t_s in s:

          if pred(t_r, t_s):

              add join(t_r, t_s) to result
- If DB can only hold one block of each table in memory:
  - Inner loop traverses *s* once *per tuple* in *r*: $n_r \times b_s$ reads
- What if the outer loop traverses *r* by *blocks*, not tuples?
  - Try to join all tuples from a block in *r* against a block in *s*

Block Nested-Loop Join

- Traversing *r* and *s* by blocks instead of tuples:

```
for B_r in r:
    for B_s in s:
        for t_r in B_r:
            for t_s in B_s:
                if pred(t_r, t_s):
                    add join(t_r, t_s) to result
```

- Improves worst-case read-behavior of nested-loop join
  - Outer loop performs b_r block-reads
  - Inner loop traverses *s* once *per block* in *r*: $b_r \times b_s$ reads
- Performs $b_r \times (b_s + 1)$ block reads

Block Nested-Loop Join (2)

- Traversing *r* and *s* by blocks instead of tuples:

  for B_r in *r*:
      for B_s in *s*:
          for t_r in B_r:
              for t_s in B_s:
                  if pred(t_r, t_s):
                      add join(t_r, t_s) to result
- Worst-case performance – large disk seeks:
  - Inner loop still traverses *s* sequentially: once per loop = b_r
  - Outer loop traverses *r* in b_r blocks: b_r total seeks
- Performs $2b_r$ large seeks

Block Nested-Loops Join (3)

- Best-case scenario: at least one table fits in memory
  - Performs $b_r + b_s$ reads, and 2-3 seeks total
  - Put smaller table on inner loop of join
- Worst-case scenario: only two blocks fit in memory
  - Performs $b_r \times (b_s + 1)$ block reads, and $2b_r$ large seeks
  - Put smaller table on outer loop of join (minimize seeks)
- Similarly, if neither table fits entirely in memory, put smaller table on outer loop of join

Block Nested-Loop Optimizations

- Several other optimizations to block nested-loop join, most notably:
- Instead of reading outer table in blocks, read as much as will fit into memory
  - For M total blocks, read in $M - 1$ blocks from r, 1 from s
  - Reduces total number of large disk seeks to $b_r / (M - 1)$
- For inner loop, scan table forward and then backward
  - Alternate direction of file-scan on subsequent iterations
  - Data pages from previous iteration will still be in the buffer manager's memory

Other Join Algorithms

- Nested-loops join is generally useful, but slow
  - Compares every tuple in *r* with every tuple in *s*
  - Performs $n_r \times n_s$ iterations through loops
- Most joins involve equality tests against attributes
  - Such joins are called *equijoins*
- Two other join algorithms for evaluating equijoins
  - Are often <u>much</u> faster than nested-loops join
  - Can only be used in specific situations (but these situations are extremely common…)

Sort-Merge Join

- If relations being joined are ordered on join-attributes, can use *sort-merge join* to compute the result
- Maintain two positions into the input relations
- If left relation's values for join-attributes are smaller, move left pointer forward
- If right relation's values for join-attributes are smaller, move right pointer forward
- If join-attribute values are identical then join the runs of tuples with equal values

r:

	A	B
	9	cat
➡	11	dog
	11	horse
	15	pig
	15	frog
	19	cow

s:

	A	C
	7	green
	9	yellow
➡	11	pink
	14	orange
	15	blue
	15	red
	19	mauve
	23	puce

Sort-Merge Join (2)

- Most difficult part of sort-merge join implementation is handling runs of tuples with the same value
- Example: given *r* and *s* contents, should end up with:
  - four rows with A = 15
  - (15, pig, blue)
  - (15, pig, red)
  - (15, frog, blue)
  - (15, frog, red)
- Clearly need a way to go back in the tuple-stream

r:

A	B
9	cat
11	dog
11	horse
➡ 15	pig
15	frog
19	cow

s:

A	C
7	green
9	yellow
11	pink
14	orange
➡ 15	blue
15	red
19	mauve
23	puce

Sort-Merge Join (3)

- In some cases, a plan-node might need to go back to an earlier point in its child's tuple-stream
  - e.g. when *r*'s pointer moves forward, if join-attributes don't change then need to go back to start of the corresponding values in *s*
- Plan nodes can support marking, and resetting to last marked position
- Alternative:
  - Store all rows in *s* with same values in memory…
  - But, can't always guarantee they'll fit!

r:

A	B
9	cat
11	dog
11	horse
15	pig
15	frog
19	cow

s:

A	C
7	green
9	yellow
11	pink
14	orange
15	blue
15	red
19	mauve
23	puce

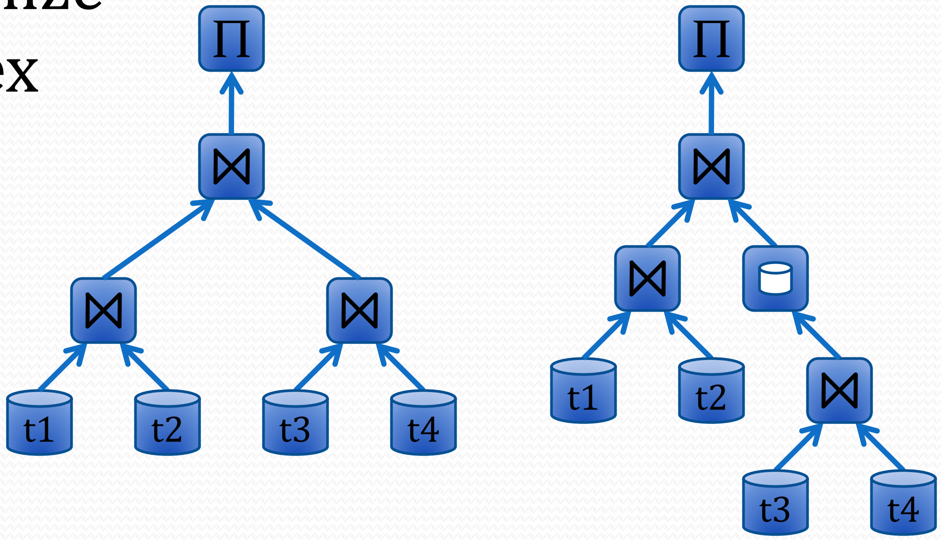
marked

Materialized Results

- Not every kind of plan-node can provide marking
  - (nor should it, necessarily…)
  - Similarly, not every kind of plan-node can be reset to the beginning of its tuple-stream
- In cases where a plan-node requires marking from one of its children, but the child doesn't support marking:
  - Insert a *materialize* plan-node above the child
  - The materialize plan-node buffers every row the child plan-node produces, allowing marking and resetting
  - If the materialize node's memory usage grows beyond a set limit, it can use a temporary file to store the results

Nested-Loops and Materialize

- Nested-loop joins evaluate right subplan once for each tuple (or block) produced by left subplan
  - Anything more complex than a simple file-scan on right of nested-loops join will be very expensive to evaluate
- Instead, insert a materialize plan-node above complex sub-plans on right side



Sort-Merge Join with Marking

- Implement sort-merge join to only require marking on right subplan

```
SortMergeJoin {
  leftTup = initial left tuple
  rightTup = initial right tuple
  while (true) {
    while (leftTup != rightTup) {
      if (leftTup < rightTup)
        advance left subplan
      else
        advance right subplan
    }

    // Now left and right tuples
    // have the same values.
    mark right subplan position
    markedValue = rightTup
    while (true) {
      while (leftTup == rightTup) {
        add joined tuples to result
        advance right subplan
      }
      advance left subplan
      if (leftTup == markedValue)
        reset right subplan to mark
      else
        // return to top of outer loop
        break
    }
  }
}
```

From PostgreSQL: nodeMergejoin.c