Relational Database System Implementation

CS122 – Lecture 6

Winter Term, 2017-2018

Plan-Node Implementations

- Previously: To evaluate SQL queries, we must…

1. Implement relational algebra operations in some way
2. Translate the SQL abstract syntax tree (AST) into a corresponding relational algebra plan
   - Covered a variety of naïve translations that will work
3. Figure out how to evaluate plan and generate results
   - We will use a pull-based, pipelined evaluation of query plans

- Still need implementations of our plan nodes

Select Implementations

- Select σ plan-nodes are easy to implement
  - Retrieve tuples from child plan-node (or from a table) until predicate is true, then pass the tuple to parent
    - An unspecified predicate is treated as *true*
- Several different kinds, based on source of tuples
  - File-scan through a table – no children; reads from table
  - Simple filter plan-node – one child plan-node
  - *(Also index-scans – will discuss in a later lecture…)*
- If select predicate has an equality condition on a key, it can stop once it returns its first row
  - Halves the expected cost of the select operation

Project Implementations

- Project Π plan-nodes are also easy to implement
  - Retrieve next tuple from child plan-node, and compute an output tuple based on the project criteria
- Project expressions are evaluated in the context of the child node's schema and tuple data
  - Child schema specifies variable names; tuples specify values
- Both selects and projects can have a "hidden" cost:
  - If planner/optimizer is not able to rewrite subqueries in the SELECT clause, or in a WHERE/HAVING clause, either of these plan-nodes could end up doing correlated evaluation

Group/Aggr. Implementations

- Implementing grouping and aggregation is similarly straightforward
- If input tuples are sorted on grouping attributes, can implement a sort-based grouping/aggregation node
- For each input tuple:
  - If grouping-attribute values changed from previous input (or child plan-node finishes producing tuples) then the current group is completed
  - Output a tuple containing grouping-attribute values, and also aggregate function values
  - Reset aggregates, store new group-attribute values, and begin calculating the new group's aggregates

Group/Aggr. Implementations (2)

- Sketch of sort-based implementation: $_{g1,g2,\ldots}\mathcal{G}_{e1,e2,\ldots}(E)$

```
current_group = []
current_aggregates = []
do:
   t := next tuple from E
   group := compute g1, g2, ... using t                  // Skip if t is null
   if t == null or group != current_group:               // Current group is done
      add join(current_group, current_aggregates) to result
      current_group := group
      reset current_aggregates
   update current_aggregates using t
while t != null
```

Group/Aggr. Implementations (3)

- Aggregate functions work differently from simple scalar functions
  - Simple functions take inputs and return an output
- Aggregate functions are fed a sequence of input values, and update their aggregate state with each input
- Example: MIN(x) aggregate function
  - As a group of input tuples is being consumed, MIN(x) function is handed each input value in sequence
  - When group of input tuples is completed, MIN(x) function can be queried for its aggregate result

Group/Aggr. Implementations (4)

- If input tuples aren't sorted on grouping attributes then a hash-based implementation must be used
- Plan-node maintains a hash-table that maps distinct values of $\langle g1, g2, \ldots \rangle$ to aggregate functions $\langle e1, e2, \ldots \rangle$
- No way of knowing when all tuples for a given group have been seen…
  - Hash-based implementation can't output any results until all input tuples have been seen
- This can have serious memory implications for large data sets with large numbers of distinct groups
  - Must use external memory if internal memory overflows

Group/Aggr. Implementations (5)

- Sketch of hash-based implementation: $_{g1,g2,\ldots}\mathcal{G}_{e1,e2,\ldots}(E)$

```
// Compute all groups, and their corresponding aggregates
group_aggregates = {}
while E has more tuples:
    t := next tuple from E
    group := compute g1, g2, ... using t
    aggregates := group_aggregates[group]          // Add entry if missing
    update aggregates using t

// Output all of our computed groups and aggregates as tuples
for group, aggregates in group_aggregates:
    add join(group, aggregates) to result
```

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 arbitrary predicates!
- Drawbacks: it's very slow

Nested-Loop Join (2)

- How do we extend this to compute $r \ltimes_\theta s$?
  - 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 $r \ltimes_\theta s$?
  - 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, $r \ltimes_\theta s = \Pi_{R,S}(s \rtimes_\theta r)$
  - (Must take care to adjust result schema properly)
- Unfortunately, $r \rtimes_\theta s$ is similarly out of reach with nested-loop join
  - *(This is why MySQL can't do full-outer joins.)*

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 $r \triangleright_\theta s$?
  - 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)

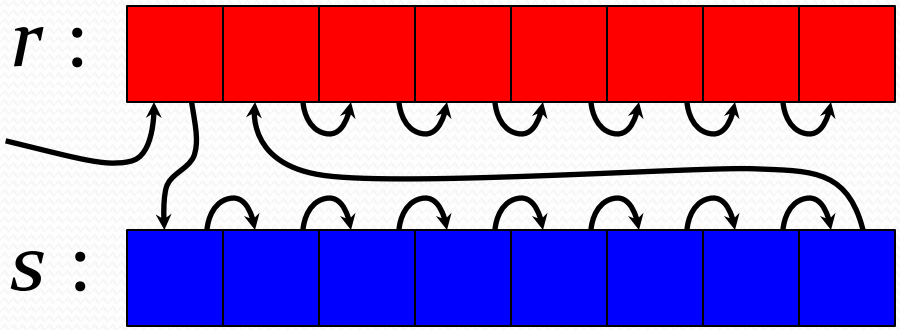
- Nested-loop join:

  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)

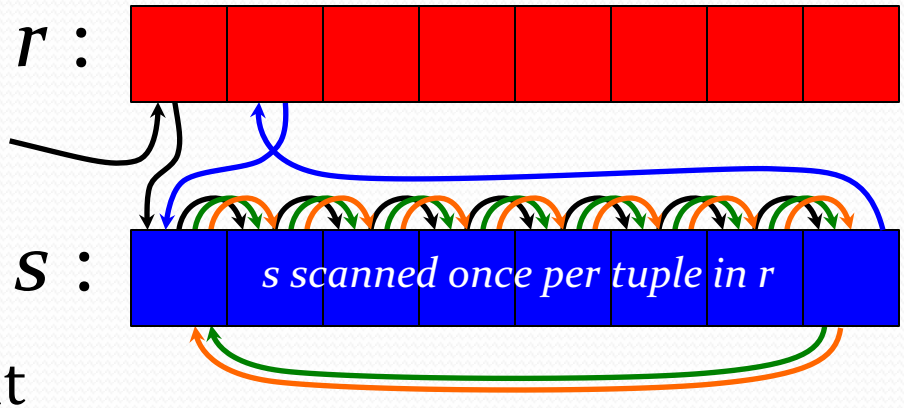
- Nested-loop join:

  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)

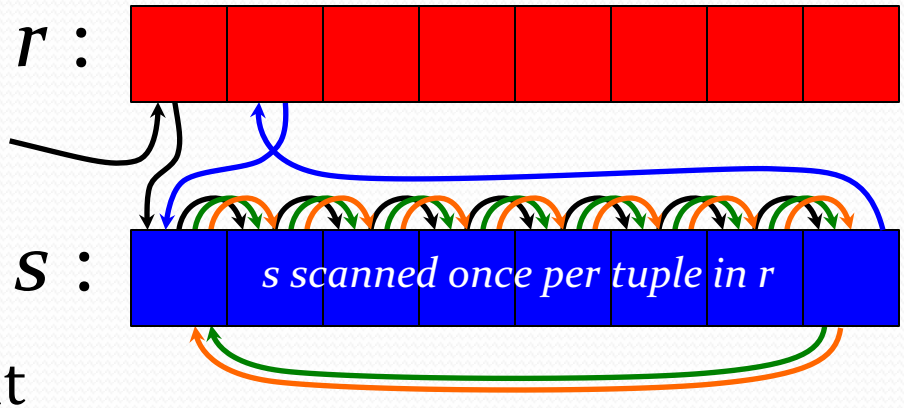
- Nested-loop join:

  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)

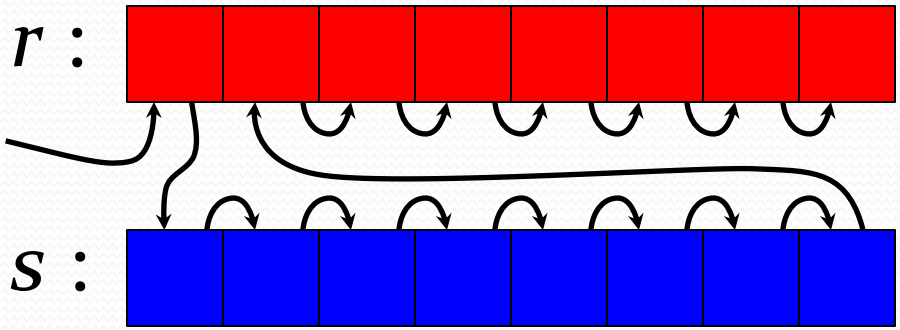
- Nested-loop join:

  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 much faster than nested-loops join
  - Can only be used in specific situations (but these situations are extremely common…)