Relational Database System Implementation

CS122 – Lecture 8

Winter Term, 2018-2019

Last Time: Other Join Algorithms

- Started looking at 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…)

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 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

Sort-Merge Join Costs

- Assume that input relations are already sorted… ☺
- Also, assume join-attributes are a primary key in both input relations
  - Each row on left will join with at most one row on right (i.e. no marking or resetting required on right table)
  - For $r \bowtie s$, results in $b_r + b_s$ blocks read
- How many disk seeks, if buffer manager can only hold one block from each of r and s?
  - Would generally expect $b_r + b_s$ disk seeks as well. SLOW.

Sort-Merge Join Costs (2)

- Sort-merge join really *requires* buffering for input relations, to avoid disk seek issues
  - Allocate b_b blocks of buffering for each input relation
  - Use read-ahead on input tables (always read b_b blocks!)
  - Reduces seeks to ceiling(b_r/b_b) + ceiling(b_s/b_b)
- What if all rows in *r* and *s* have the same join value?
  - Algorithm will mark first tuple in *s*, then scan through *s* for each row in *r*
  - If buffer manager can only hold one page from each file:
    - Blocks read will be $b_r + n_r \times b_s$
    - Disk seeks will be $b_r + n_r$
    - Worst case, sort-merge join behaves just like nested-loops join

Sort-Merge Join Costs (3)

- Apply same strategies to sort-merge join as with nested-loops join
  - Table on right side of join should fit within memory, if possible
  - If not, allocate plenty of buffer space for processing join
  - If right subplan is more complex than a table scan, use a materialize node to allow results to be traversed multiple times
- Our cost estimates assumed that the inputs are sorted
  - Usually not the case
  - Need to include cost of sorting in costing estimates too

Outer Joins with Sort-Merge?

- Can we modify this algorithm to produce left/right/full outer joins?

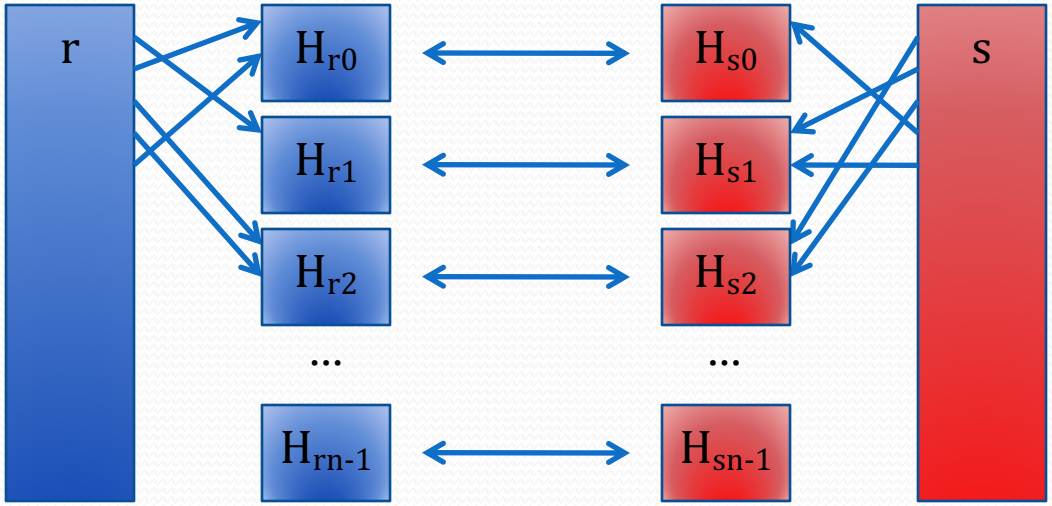
```
SortMergeJoin {
  leftTup = initial left tuple
  rightTup = initial right tuple
  while (true) {
    while (leftTup != rightTup) {
      if (leftTup < rightTup)
        advance left subplan
      else
        advance right subplan
    }

    // 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
    }
  }
}
```

Can generate outer-join results here!

From PostgreSQL: nodeMergejoin.c

Hash Join

- Can also use hashing to perform equijoins efficiently
- For $r \bowtie s$, performing equijoin on JoinAttrs
  - Apply a hash function h_p(JoinAttrs) to partition tuples in r and s into n partitions
  - Tuples in partition H_{ri} will only join with tuples in H_{si}

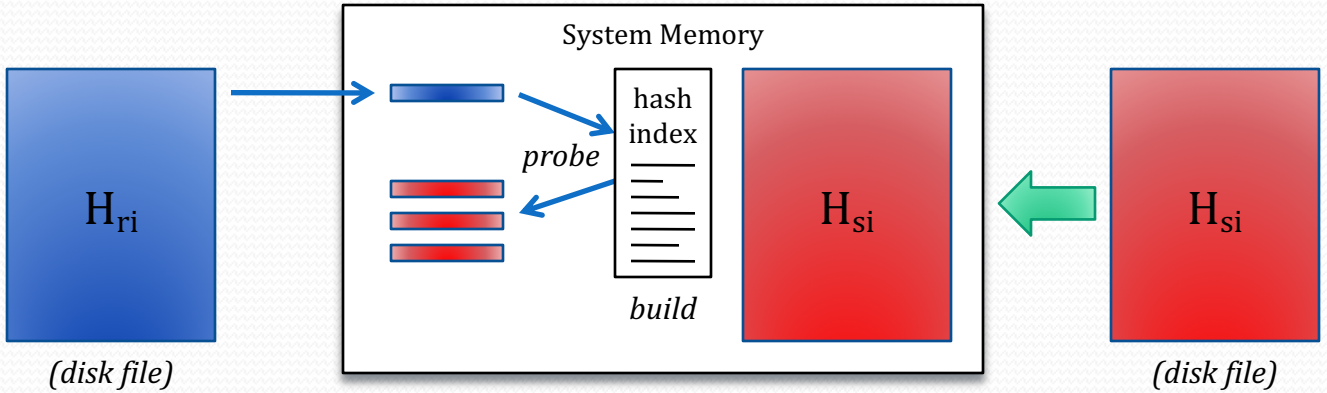


Hash Join (2)

- Once input relations are partitioned, join each pair of partitions H_{ri} and H_{si} in sequence:
  - Load H_{si} into memory, and build a hash index against it
    - Use a different hash function $h_i()$ for this hash-index
    - Just reusing previous hash function $h_p()$ won't provide a uniform random distribution of input tuples
  - For each tuple t_r in H_{ri}, probe the hash index to find all tuples in H_{si} that join with t_r
- Only require that entirety of H_{si} fits into memory (plus its corresponding hash-index)
  - Partitions are stored on disk until they are needed

Hash Join (3)

- s is called the *build relation* (a.k.a. the *build input*)
  - The hash index is built against partitions of s
  - Partitions of the build relation must fit in memory
- r is called the *probe relation* (a.k.a. the *probe input*)
  - The join algorithm probes the hash index using tuples from partitions of r
  - Partitions of probe relation don't need to fit in memory
- Generally, smaller relation should be the build relation



Hash Join Costing

- Partitioning the relations requires a complete pass over both *r* and *s*, and the partitions are written to disk
  - Requires $2(b_r + b_s)$ disk transfers
  - Could also result in partially full blocks, since a partition won't necessarily be completely full
    - Adds a small overhead based on the number of partitions
- The join process itself must read each partition once
  - Requires $b_r + b_s$ disk transfers
- Total disk access cost is approximately $3(b_r + b_s)$

Hash Join Issues

- Biggest issue is if a partition H_{si} doesn't fit into memory
  - e.g. perhaps distribution of join-attribute values isn't friendly to hash function
- *Overflow resolution*:
  - If a hash overflow is detected, apply a second, different hash-function to large partition
- *Overflow avoidance*:
  - Partition input relations into many smaller partitions, then combine partitions into units that fit into memory
- If data distribution isn't suitable to hash join, may simply need to use a different join algorithm!
  - Good statistics (e.g. histograms) essential to determine this

Hash Join Issues (2)

- Another issue with large tables is if number of partitions required by table size is too large to fit in memory
  - e.g. since partitions are written to disk, database must be able to hold at least one disk block per partition in its buffers
- Requires *recursive partitioning*:
  - On first pass, split table into as many partitions as possible
  - Repeat this process on previously generated partitions (using a different hash-function) until all partitions of build relation fit in memory
- Generally not required until tables are many TBs in size

Hash Join Algorithm

- Hash join algorithm:

```
# Partition s
for each tuple t_s in s:
    i = h(t_s[JoinAttrs]);
    Add t_s to partition H_si;

# Partition r
for each tuple t_r in r:
    i = h(t_r[JoinAttrs]);
    Add t_r to partition H_ri;
```

```
/* Perform hash-join */
for i = 0 to n_h:
    read H_si and build
        in-memory hash index
    for each tuple t_r in H_ri:
        probe hash-index to find all
            tuples t_s that join with t_r
        for each matching tuple t_s:
            add join(t_r, t_s) to result
```

Hash Join Algorithm (2)

- Hash join algorithm:

```
# Partition s
for each tuple t_s in s:
    i = h(t_s[JoinAttrs]);
    Add t_s to partition H_si;

# Partition r
for each tuple t_r in r:
    i = h(t_r[JoinAttrs]);
    Add t_r to partition H_ri;
```

- s is partitioned before r to allow an optimization:
- If enough memory is available, partition H_{s0} is kept in memory from the "partition s" phase
  - A hash index also built on H_{s0}
- During partitioning of r, tuples that hash into H_{r0} are tested against in-memory H_{s0} index
- Reduces disk IOs by a small but significant amount
- This is called *hybrid hash-join*

Outer Joins with Hash Join? (1)

- Can we alter this to perform left-outer joins?

```
# Partition s
for each tuple t_s in s:
    i = h(t_s[JoinAttrs]);
    Add t_s to partition H_si;

# Partition r
for each tuple t_r in r:
    i = h(t_r[JoinAttrs]);
    Add t_r to partition H_ri;
```

```
/* Perform hash-join */
for i = 0 to n_h:
    read H_si and build
        in-memory hash index
    for each tuple t_r in H_ri:
        probe hash-index to find all
            tuples t_s that join with t_r
        for each matching tuple t_s:
            add join(t_r, t_s) to result
```

Outer Joins with Hash Join? (2)

- Change probe logic to perform left-outer joins

```
# Partition s
for each tuple t_s in s:
    i = h(t_s[JoinAttrs]);
    Add t_s to partition Hsi;

# Partition r
for each tuple t_r in r:
    i = h(t_r[JoinAttrs]);
    Add t_r to partition H_ri;
```

```
/* Perform hash-join */
for i = 0 to n_h:
    read H_si and build
        in-memory hash index
    for each tuple t_r in H_ri:
        probe hash-index to find all
            tuples t_s that join with t_r
        if t_r has matching tuples:
            for each matching tuple t_s:
                add join(t_r, t_s) to result
        else:
            add join(t_r, null_s) to result
```

Outer Joins with Hash Join? (3)

- What about full-outer joins?

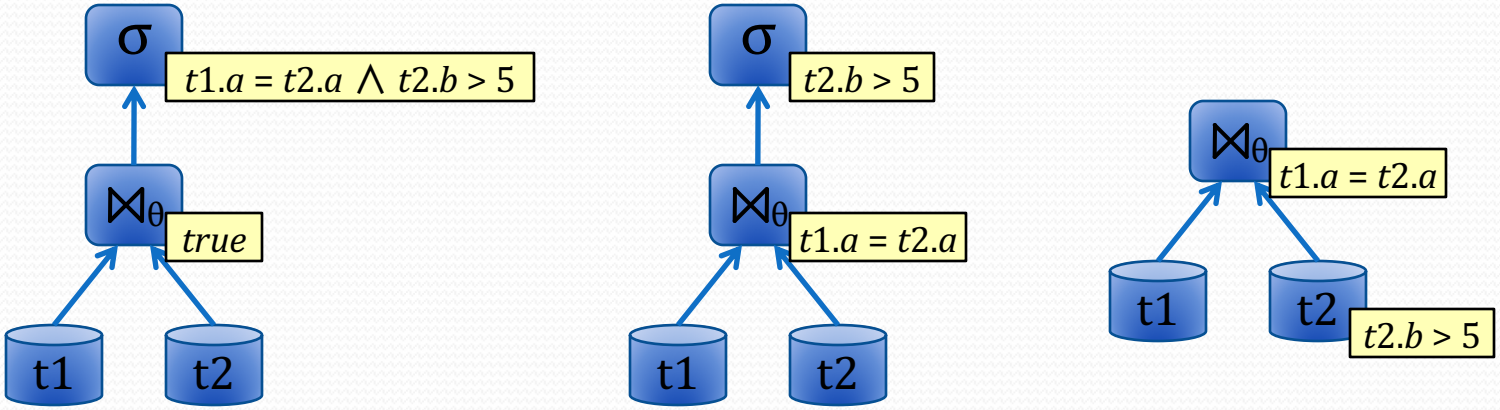
```
# Partition s
for each tuple t_s in s:
    i = h(t_s[JoinAttrs]);
    Add t_s to partition Hsi;

# Partition r
for each tuple t_r in r:
    i = h(t_r[JoinAttrs]);
    Add t_r to partition H_ri;
```

```
/* Perform hash-join */
for i = 0 to n_h:
    read H_si and build
        in-memory hash index
    for each tuple t_r in H_ri:
        probe hash-index to find all
            tuples t_s that join with t_r
        for each matching tuple t_s:
            add join(t_r, t_s) to result
```

Need to alter hash-index to record which tuples in H_{si} were joined. Then we can compute full-outer joins.

Alternative Plans

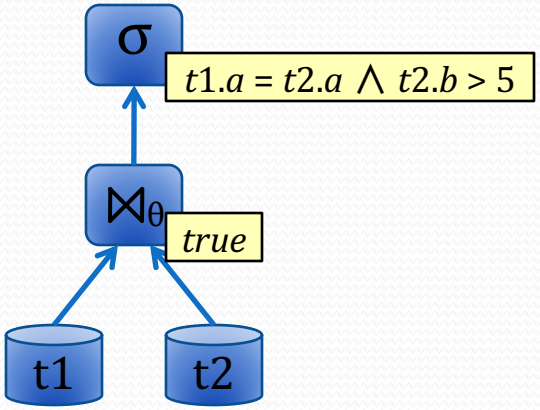
- Earlier, saw three plans for a query:
  - SELECT * FROM t1, t2 WHERE t1.a = t2.a AND t2.b > 5;

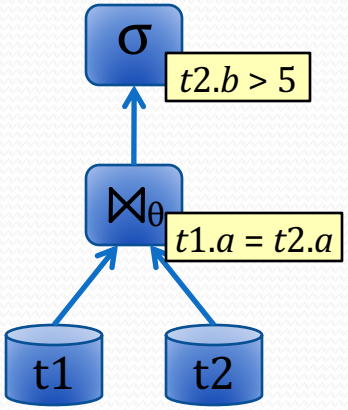


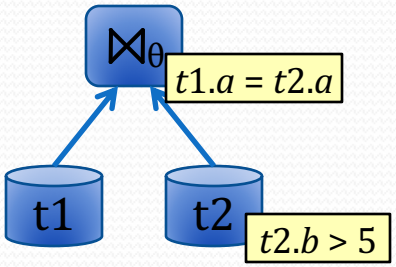
- Two questions:
  - How do we know which plan is best?
  - How do we know the plans are actually equivalent?

Plan Costing

- Can devise ways of measuring costs of different plans
- Basic measurements:
  - Number of rows generated by each plan-node
  - Number of disk-accesses performed by each plan-node
- More advanced measures:
  - CPU/memory usage, avg size of each row in bytes, etc.

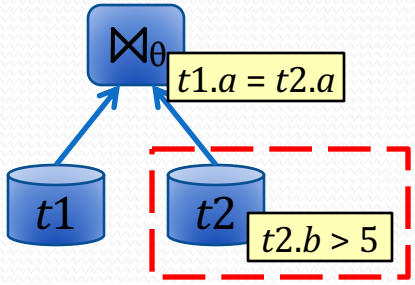






Plan Costing (2)

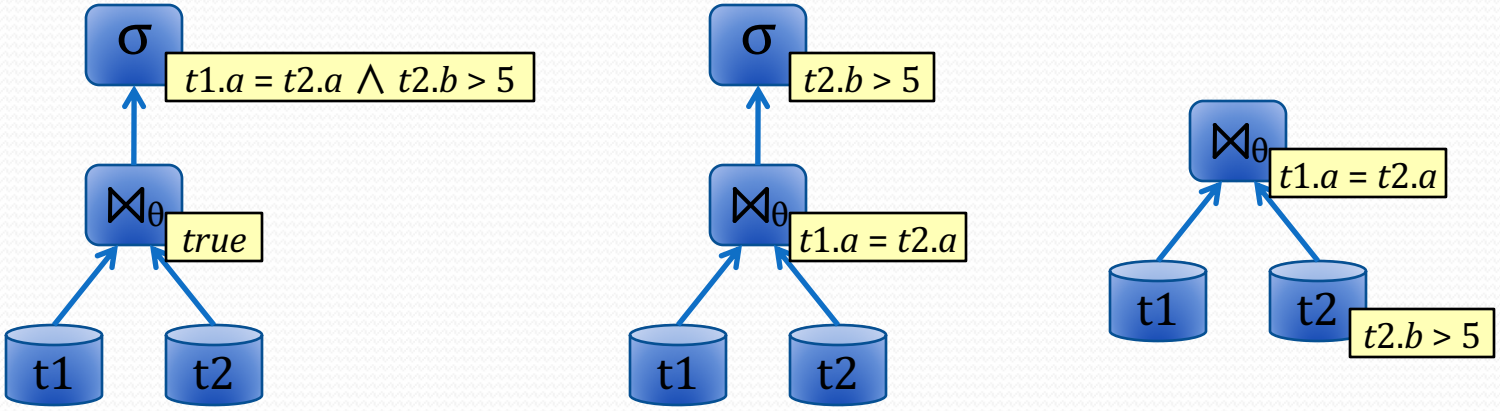
- Example: $\sigma_{b>5}(t2)$
  - Given: $t2$ is a heap file, with no indexes on b
- How many disk blocks are accessed?
  - Every disk block in $t2$
- How many rows will be produced?
  - *???*
- If we knew the minimum and maximum values for $t2.b$:
  - Assume: b is uniformly distributed
  - Guess: # rows in $t2 \times (b_{max} - 5) / (b_{max} - b_{min})$



- If we had a histogram for $t2.b$'s values, could make a *much* better guess!

Plan Costing Goals (Ideal)

- Estimates should be as accurate as possible
- Estimates should be easy to compute
- Estimates are logically consistent
  - Estimated statistics for a query shouldn't vary in abnormal ways, based on how the query is computed
  - SELECT * FROM t1, t2 WHERE t1.a = t2.a AND t2.b > 5;
    - Ideally, estimates of how many tuples are produced by each plan will be roughly the same



Plan Costing Goals (Reality)

- Goals of plan costing:
  - Estimates should be as accurate as possible
  - Estimates should be easy to compute
  - Estimates are logically consistent
- Unfortunately, very hard to achieve in practice

- All we *really* require:
  - **Faster plans end up with lower cost than slower ones**

Plan Costing and Statistics

- To make effective cost estimates, the database must keep statistics on values that appear in each table
- Generally, statistics are very expensive to compute…
  - Databases generally don't keep these stats up to date
  - Some update stats when # of rows in a table changes substantially; others require manual updating of stats
- The statistics don't need to be perfect!
  - Just need to be good enough to guide optimization phase
- But, if stats are very different from actual table data, generated plans are likely to be horrible.

Table Statistics

- Some useful statistics to keep per table:
  - n_r – the number of tuples in table r
  - b_r – the number of blocks containing tuples in r
    - For heap files, will be very close to total # of blocks in file
    - For sequential and hashing files, may be very different
  - l_r – the average size of a tuple in r, in bytes
  - f_r – the blocking factor of table r
    - The average number of tuples in r that fit in one block
    - Generally, $b_r \approx \text{ceiling}(n_r / f_r)$

Table Statistics (2)

- More useful statistics:
  - V(A, r) – the number of distinct values of attribute A that appear in table r
  - min(A, r) – the minimum value of attribute A in table r
  - max(A, r) – the maximum value of attribute A in table r

- Provide an operation to compute/update these stats for a given table
  - Expose it as a command, and/or update automatically
  - e.g. **`ANALYZE TABLE t;`**