Relational Database System Implementation

CS122 – Lecture 7

Winter Term, 2017-2018

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:

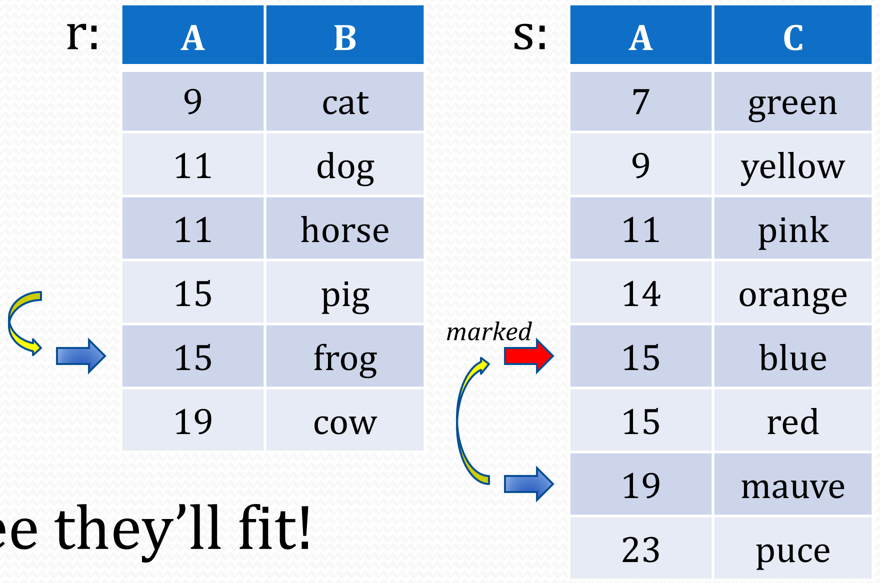
	A	B
	9	cat
	11	dog
	11	horse
➡	15	pig
	15	frog
	19	cow

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:

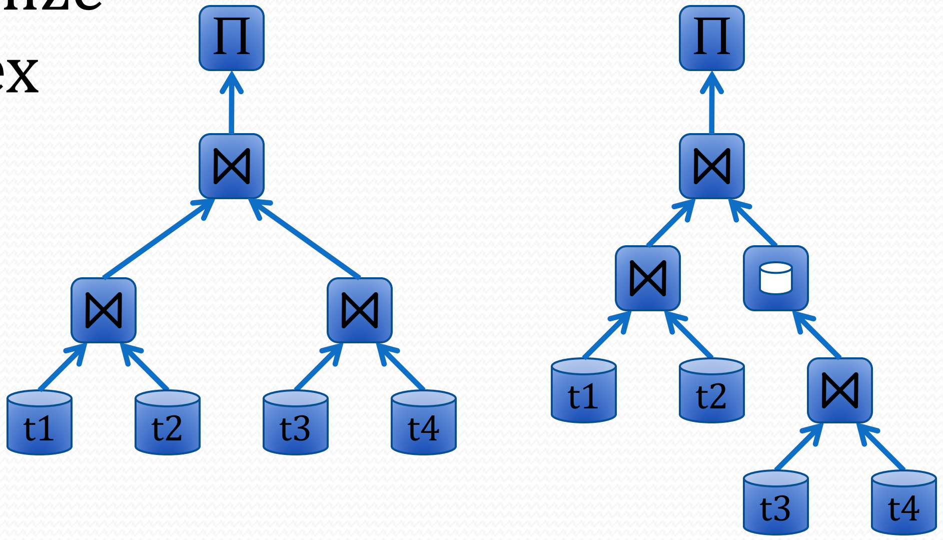
A	C
7	green
9	yellow
11	pink
14	orange
15	blue
15	red
19	mauve
23	puce

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

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?

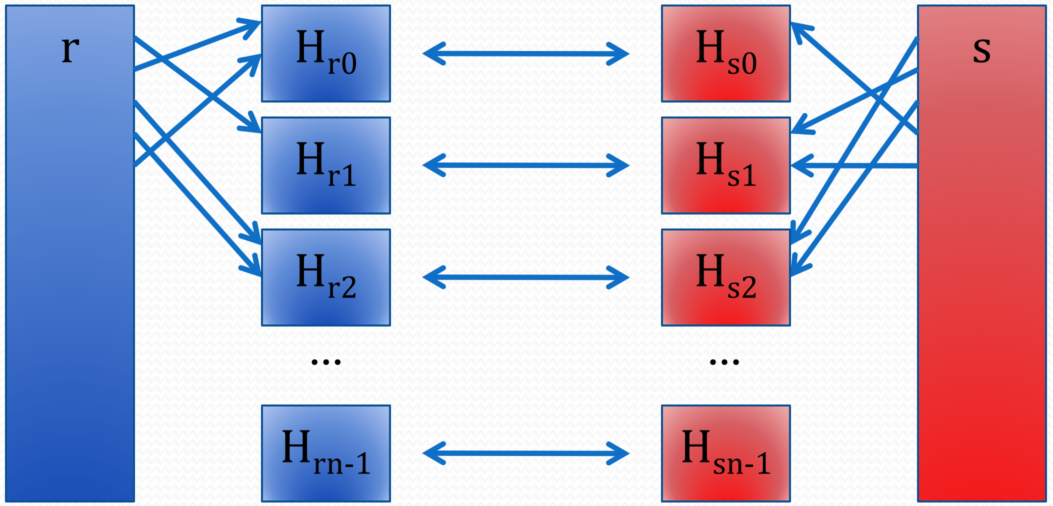
```
SortMergeJoin {
  leftTup = initial left tuple
  rightTup = initial right tuple
  while (true) {
    while (leftTup != rightTup) {
      if (leftTup < rightTup)
        advance left subplan
      else
        advance right subplan
    }

    // 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}

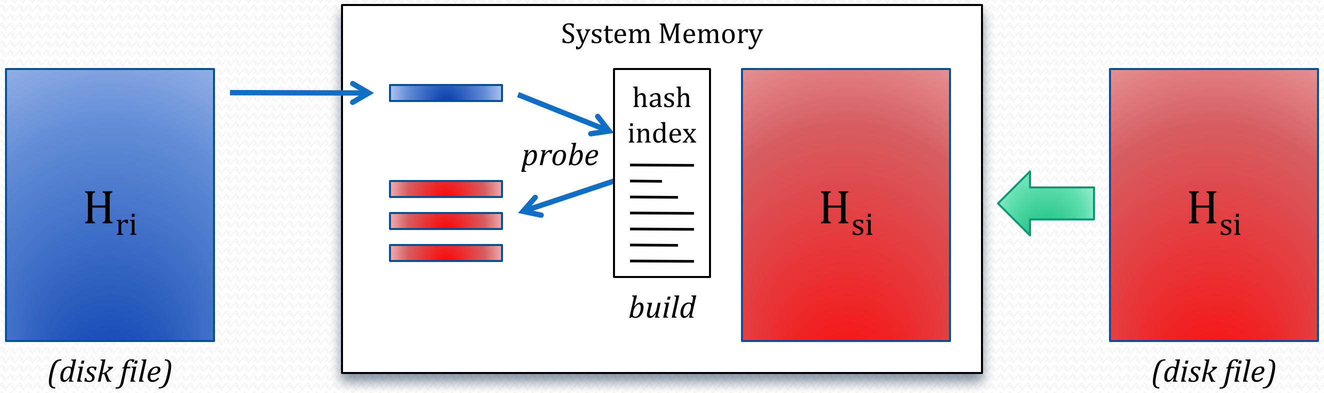


Hash Join (2)

- 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 <u>different</u> 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 <u>must</u> 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 $3(b_r + b_s)$
  - (Plus change…)

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 GBs 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.