Today:

- Hash join, including parallel version.
- Algorithms for other operations.

Soon:

- Recovery from crashes.
- Supporting transactions = atomic operations.

Hash Join

- Pick a hash function h that maps B-values to buckets.
    - If "B" is really a combination of attributes, then the hash function involves them all.
- Send R and S tuples to separate hash tables, each based on h.
- Examine the ith buckets of both hash tables to find joining tuples.

Example

Two hash tables of 50 buckets, used for our example R and S.

- As we read R, we hash to buckets, which may fill up. If so, we move the current block for that bucket to disk and regard its buffer as a new block, to which the old block is chained.
    - Total disk I/O's = 1000 reads and a little more than 1000 writes (because some blocks may not be completely full).

- Similarly, hashing S requires about 1000 disk I/O's.
- Average bucket for R has 20–21 blocks; those for S average 10–11.
    - ❖ Thus, unless there is a lot of data skew, we should be able to bring an entire R-bucket and an entire S bucket into memory at the same time.
    - ❖ Additional disk I/O's for reading buckets: about 1500.

Comparison

For our example:

1. Nested-loop: 5500 (not 6000: interchange R and S).
2. Sort-Join, naive: 7500.
3. Sort-Join, combine phase 2 of 2PMMS with join step: 4500
4. Hash-Join: 4500+.

But note:

- Nested-loop is essentially quadratic, the other approaches are linear.
- We might take advantage of one argument being sorted already.
    - Example: $AB \bowtie AC \bowtie AD$.
- In hash-join, we can hash only recordID-joinAttribute pairs, using fewer blocks for the buckets.
    - Downside: we may need to read a lot of blocks for the tuples themselves if a lot of pairs join.

Parallel Hash Join

If there are many processors, hash join allows all processors to be useful at the same time (even in a "shared nothing" architecture).

- Assume R and S are distributed across the processors.

Distribution Phase

- If there are p processors, all use a hash function h from the values of the join attribute(s) to $[0..p-1]$.
- Each processor hashes its R and S tuples, sending tuple t to the processor numbered $h(t[B])$, where B represents the join attribute(s).
  - In a message-passing architecture, we should bundle a number of tuples in one message.

Local Join Phase

- Once distribution is complete, each processor takes a look at what it has received, and hashes it to new, local hash tables.
  - One table for each of R and S, just like uniprocessor hash-join.
  - But make sure you use a hash function other than h, or you will get a nasty surprise.

Analysis

Once the data is distributed, the elapsed time is similar to hash-join on relations each $1/p$th as large.

- But we incur whatever communication costs there are.

Using Indexes

What if we want $R(A, B) \bowtie S(B, C)$ and we have an index on $R.B$?

- Keep index buffered in memory (will it fit?).

```
for each S-tuple s do
    consult index for matching
            R-tuples;
    output all matches;
end
```

Analysis

For our running example:

- 500 disk I/O's to read S.
- If each S-tuple matches k R-tuples, we make $5000k$ disk reads for R.
  - Could be best or worst method for this example.
  - If index is clustered, then disk reads are divided by records-per-block, but at least one disk I/O per S-tuple.

Other Things to Do With an Index

- If the index is based on a sorted R, use it to read R sorted without paying the sorting price in sort-join.
- If R is not clustered, then all the other analyses are bogus (count of disk I/O's to read R is wrong).
  - Gives an advantage to index-join.

Problem

Suppose there are indexes on both $R.B$ and $S.B$. How could we take advantage?

Computing Full-Relation Operations

- $R \cup S$, $R \cap S$, $R - S$, $\delta(R)$.
- Bag versions of $\cup_B$, etc.
- Approaches:
  1. Sort-based.
  2. Hash-based.
  3. Index-based.

Sort-Based Algorithms

Use 2PMMS (entire tuple is the sort key). But in second phase, we need a different approach to producing output.

- $R \cup S$ or $R \cup_B S$. Move each tuple to the output as in a merge.
  - Only one copy for set-union.
  - But there is a special algorithm for $R \cup_B S$ only: just copy R and S to the output.
- $R \cap S$ or $R \cap_B S$.
  - If leading tuples are equal, move one to the output and skip over both.
  - Otherwise, skip over smaller.
  - Example $1, 2, 2, 4$ and $2, 3, 4, 4$.

- $R - S$ or $R -_B S$.
  - If R-tuple precedes S-tuple, move it to the output and skip over it.
  - If tuples are equal, no output, skip over both.
  - If S tuple precedes R-tuple, no output, skip over it.
  - Example: $1, 2, 2, 4$ and $2, 3, 4, 4$.
- $\delta(R)$: when sorting R, output a tuple only if it is different from following tuple, or at end.

Hash-Based Algorithms

1. Hash R; hash S to separate tables with same hash function.

    - Tuple-and-count, rather than repeated tuples, OK for bags.

2. Read corresponding buckets from each hash table and perform the appropriate operation. Examples:

    - $\cap$: output tuples appearing in both buckets.

    - $-_B$: Let r be the number of times a tuple t appears in R and s the number of times it appears in S. If $r > s$ output the tuple $r - s$ times; else do not output the tuple.

- For $\delta(R)$: hash R and then output (once) tuples that appear in each bucket.

“Edge-Effect” Trick for Hash Schemes

If there is enough room to keep some buckets in memory at all times, you can save the cost of writing and then reading them.

- When hashing tuples of the second relation S, if they go in a main-memory bucket, do not bucketize them at all, but deal with them immediately using the corresponding bucket of R.
- Extreme case: when the whole relations fit in memory.
- Generalizes *hybrid-hash-join* mentioned in SKS.

Indexed-Based Algorithms

- Most useful for set-based operations and δ.
- Assume there is an index on key attribute(s) of S. For $R \cap S$, read R, lookup each tuple in S, output if there.
  - ❖ Wins if S is small or nonclustered.
- Similar for $R - S$.
- If there is an index on any attribute A of R, then can compute $\delta(R)$ by scanning the index, bringing tuples with a fixed A-value to memory and finding unique tuples.

Analysis

All sort- and hash-based algorithms essentially read and write every tuple twice.

- But if relations fit in main memory, then only once.
- At the other extreme, the hash-based algorithms may face the problem similar to 2PMMS failing because the data is too large.
  - ❖ If the number of buffers B available (which equals the number of buckets we can use) is so small that we cannot then fit a bucket into main memory (i.e., B^2 less than number of blocks of relation), we can:

  1. Rehash each bucket, and handle a subbucket at a time.
  2. Use a larger number of buckets, but only store tuples from the first B buckets on one pass. Make repeated passes to handle tuples that belong in other buckets.

Group-By

Example:

```
SELECT a, b, SUM(c)
...
GROUP BY a, b;
```

If no `HAVING`, then we can:

1. Sort by the grouping attributes, accumulate a value for each group and output the result.
2. Hash based on grouping attributes. Then examine buckets and aggregate for each group.
   - Remember fixups if $B^2 \quad <$ blocks of relation.
3. Index-based: use index on grouping attribute(s) to retrieve one group at a time.

Group-By with Having

- Sometimes, a `HAVING` clause is a mistaken where-condition.
    - ❖ Replace this by a σ.
- If the `HAVING` is real, like

```
SELECT a, b, SUM(c)
...
GROUP BY a, b
HAVING AVG(d) > 10;
```

  then accumulate the necessary aggregates for each group.
    - ❖ Hint: `AVG` can be computed from `SUM` and `COUNT`.

Selection

Key idea: take advantage of indexes. Example;
$\sigma_{A=1\ AND\ B=2\ AND\ C<10}(R)$.

- Use a (clustering?) index on A (or B) to retrieve matching tuples, along with some that don't match.
  - Clustering index: estimate of number of disk I/O's = $Blocks(R)/V(A,R)$ or $Blocks(R)/V(B,R)$, whichever is smaller.
  - Nonclustering index: estimate $N(R)/V(A,R)$ or $N(R)/V(B,R)$, whichever is smaller.
  - But may have to account for disk I/O's to read some index blocks.
  - In either case, examine retrieved tuples to see that they match the other parts of the selection.
- Use several indexes, intersect pointers in memory before accessing tuples.
  - Can use fewer I/O's — already discussed.