Parallel and Distributed Databases

Parallel Query Processing

In order to achieve **Intraoperation Parallelism**, need to develop parallel versions of relational algebra operations.

Relational algebra operations:

- Set operations: $\cup$ (union), $\cap$ (intersection), $-$ (difference)
- Sort (τ)
- Selection (σ)
- Projection (π)
- Cartesian product ($\times$)
- Join ($\bowtie$)
- Grouping/aggregation (γ)
- Duplicate elimination (δ)

Parallel Sort

Let $R = (D_0, \ldots, D_{n-1})$ be a relation, partitioned into n disks and $\tau_A(R)$ be the sort operation to be performed. Let $P_0, \ldots, P_{n-1}$ be n CPUs associated with disks $D_0, \ldots, D_{n-1}$ respectively.

Two approaches:

- **Range-partition sort**: partition the table into ranges, sort each range.
- **Merge-sort**: partition table in any way, run two-step merge-sort.

Range-partition Sort.

- **Step 0:** Separate the range of attribute A into $m < n$ partitions. Pick CPUs $P_0, \ldots, P_{m-1}$ to process and store partitions 1 through m.
- **Step 1:** In parallel, redistribute R into m partitions according to the ranges. The ith partition will be handled by CPU P_{i-1} and stored on its local disk:
  - On each CPU P, perform a *full table scan*.
  - For each tuple, determine its range i and send it to CPU P_{i-1}.
  - Additionally, on CPUs $P_0, \ldots, P_{m-1}$, wait for tuples to arrive from other CPUs, store received tuples locally.

  I/O cost: $2 * B(R)$, where $B(R)$ is the total number of disk blocks in R on all disks.

  Communication overhead: significant (depends on specific locations of tuples, but it is expected that the majority of the tuples will be set to a different CPU/disk).
- **Step 2:** In parallel, sort each partition $1, \ldots m$ on its respective CPU. Use traditional DBMS sorting algorithms (e.g., merge-sort or partition-sort).

  Output partitions in order $1, \ldots, m$.

  I/O Cost: combined cost of m DBMS sort operations - depends on the size of each partition.

  Communication overhead: none.

Merge-partition Sort.

- **Step 1: Sort.** Each CPU $P_0, \ldots, P_{n-1}$ sorts its partition $D_0, \ldots, D_{n-1}$ of R.

  I/O Cost: combined cost of n DBMS sort operations - depends on the sizes of $D_0, \ldots, D_{n-1}$.

  Communication overhead: none.
- **Step 2: Merge.** Sorted partitions are merged in parallel. This is done as follows:
  - Each sorted partition D_i of R is range-partitioned into m ranges. The tuples in these ranges are sent concurrently to CPUs $P_0, \ldots P_{m-1}$.
  - $P_0, \ldots P_{m-1}$ merge-sort the received n streams.
  - The result is output in the order $P_0, \ldots, P_{m-1}$.

  I/O Cost: $2 * B(R)$ (assuming m blocks can fit main memory on each CPU).

  Commmunication cost: almost each tuple needs to be shipped to a different CPU.

Parallel Join

For **equijoins**, it is possible to partition the join operation - i.e., make each processor perform a traditional RDBMS join on the "slices" of the two joined tables, *without loss of correctness.*

For more general **Θ-joins**, a **fragment-and-replicate** technique is used. Here, one relation is fragmented and partitioned across n CPUs. The other relation is *replicated* on each CPU.

Partitioned Join. For join queries of the sort $R \bowtie S$ or $R \bowtie_{R.A=S.B} S$, **partitioned join** works as follows:

- **Step 1:** Select a partitioning technique, and partition R and S into n fragments based on the values of the join attributes.
- **Step 2:** Perform local RDBMS join of fragments R_i and S_i on each CPU P_i. Return the union of all results from all partitions.

R and S can be partitioned using:

- Range partitioning
- Hash partitioning

Fragment-and-replicate Join. For join operations of the sort $R \bowtie_{R.A<S.B} S$, partitionaing does not work.

The **fragment-and-replicate join** works as follows:

- **Step 1: Fragment.** Partition R into n fragments in any way; distribute them to n CPUs[1].
- **Step 2: Replicate.** Replicate S on every CPU.
- **Step 3: Join.** Run local join operations $R_i \bowtie_{Theta} S$; on each CPU, combine results.

Parallel Selection.

Let R be partitioned into fragments $R_1, \ldots R_n$.

$\sigma_C(R)$ can proceed in parallel on each fragment R_i, the results are combined.

Special case 1. R is range- or hash- partitioned on attribute A, selection is $\sigma_{A=x}(R)$. In this case, only one CPU, corresponding to location of x in A's value is used.

Special case 2. R is range-partitioned on A and selection is $\sigma_{A>x}(R)$. Then selection proceeds only on those CPUs that correspond to the ranges that contain x and all higher values.

(same for $A < x$ and $A < x \& A > y$ conditions)

[1] If R is already partitioned, do nothing.

Parallel Duplicate Elimination.

Solution 1: Use parallel sort, remove duplicates from streams.

Solution 2: Range- or hash-partition R. Run local RDBMS duplicate elimination procedures.

Projection.

Without duplicate elimination. Round-robin partitioning.

With duplicate elimination. Use parallel duplicate elimination, remove attributes before eliminating duplicates.

Grouping and Aggregation

Solution 1. Range- or hash- partition R on grouping attribute(s). Perform local RDBMS grouping/aggregation operation on each CPU.

Solution 2. (essentially MapReduce) Perform grouping and partitioning at each CPU. (MAP) Partition the grouping attributes into n fragments. Send aggregate information to appropriate CPUs, finalize aggregate computations (REDUCE).