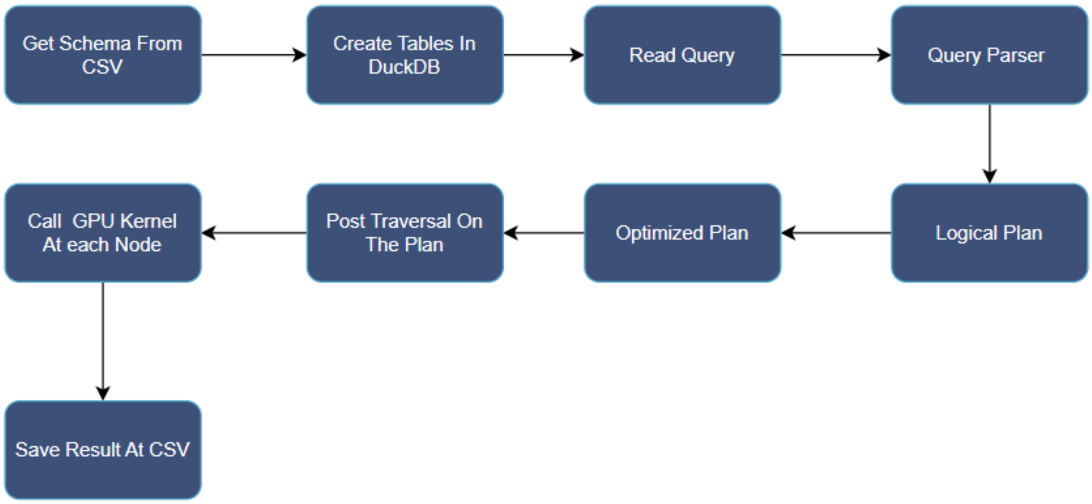
Parallel Computing Project Report

Team 14

Name	Sec	B.N.	Code
Nesma Abdelkader	2	28	9211292
Sara Gamal	1	20	9210455
Eman Ibrahim	1	14	9210265
Yousef Osama	2	32	9211386

Pipeline



Design

When the data size exceeds a specified threshold, it is divided into smaller, manageable batches. Each batch is then passed to a stream, which is responsible for invoking the kernel to process the data. This approach ensures efficient handling of large datasets by breaking them down into chunks that can be processed independently. Once all batches have been processed, the CPU collects and merges the results from each batch, producing the final aggregated output.

Kernels

For all kernels, the block size is a multiple of 32 to align with warp size for optimal performance and efficient thread scheduling.

1. **Filter Kernel**
   a. **Steps**
      i. **Data Access:** Each GPU thread accesses a specific row from the input data, using a unique index.
      ii. **Condition Evaluation:** The kernel calls `eval_condition_tokens` to check whether the current row meets the specified conditions.

    iii. **Row Selection:** If the row satisfies the conditions, it is copied to the output buffer. An atomic operation is used to increment the output counter to avoid race conditions among threads.

   b. **Optimization Techniques**

      i. **Condition Evaluation Optimization:** The conditions are evaluated using the `eval_condition_tokens` function, which efficiently handles complex condition combinations (AND/OR) using a stack-based approach.

      ii. **Efficient NULL Handling:** NULL values are excluded from calculations, ensuring they do not affect the results.

2. **Aggregate Kernel**

   a. **Steps**

      i. **Data Loading:** Each thread loads two elements into shared memory for efficient reduction.

      ii. **Reduction within the Block:** Each thread compares its two loaded values and reduces them using a loop with synchronized threads.

      iii. **Result Writing:** The final result of the block reduction is stored in global memory by the first thread in the block.

   b. **Optimization Techniques**

      i. **Shared Memory Utilization:** Partial results are stored in shared memory to minimize global memory access, reducing latency.

      ii. **Thread Synchronization:** Threads within the block are synchronized using `__syncthreads()` to avoid data hazards during reduction.

      iii. **Efficient NULL Handling:** NULL values are excluded from calculations, ensuring they do not affect the results.

3. **Project Kernel**

   a. **Steps**

      i. **Data Access:** Each GPU thread is assigned a specific row from the input dataset using its unique thread index (`idx`), and computes a pointer to the start of that row.

      ii. **Column Projection:** For each selected column, the kernel uses the `acc_sum` array to determine the byte offset within the row and the `size` array to determine the number of bytes to copy.

      iii. **Output Construction:** The selected fields are copied into a contiguous output buffer at calculated offsets. This forms the projected row containing only the required columns.

   b. **Optimization Techniques**

i. **Contiguous Memory Access:** Output memory is accessed in a coalesced manner, improving memory throughput by reducing unaligned and scattered memory accesses.

ii. **Precomputed Offsets:** Using the `acc_sum` and `size` arrays avoids recalculating field positions at GPU, reducing computation overhead inside the kernel.

iii. **Minimal Branching:** The loop and memory copy logic avoid branching, leading to efficient warp execution without divergence.

4. **Order By Kernel**

   a. **Steps**

      i. **Iterative Merge Sort (Batch Merge):** The sorting is performed iteratively using a batch-based merge sort approach:

         - For each merge width (starting from 1), the dataset is divided into multiple merge batches.
         - Each CUDA block handles the merging of two sorted segments (`A` and `B`) using multiple threads via co-ranking.

      ii. **Co-Rank Calculation:** The `co_rank` function is used by each thread to determine its merge range from the two sorted segments.

      iii. **Data Comparison and Merge:** Each thread compares corresponding elements from both segments based on the sort column and writes the smaller (or larger, if descending) value into the output buffer.

      iv. **Buffer Swap:** After each pass, the input and output buffers are swapped to prepare for the next merge iteration.

   b. **Optimization Techniques**

      i. **Co-Ranking for Parallel Merging:** The `co_rank` function enables efficient partitioning of merge work across threads, avoiding overlapping memory access and ensuring even workload distribution.

      ii. **Double Buffering:** By alternating between two device buffers (`d_input` and `d_temp`), the sort avoids unnecessary memory allocation and copying between iterations.

      iii. **Efficient NULL Handling:** NULL values in the sort column are treated consistently by mapping them to sentinel values (`-DBL_MAX`), allowing comparison logic to remain streamlined.

5. **Join Kernel**

   a. **Steps**

      i. **Kernel Launch:** The `nested_loop_join` kernel is launched with one thread per row of table A. Each thread iterates over all rows in table B, forming pairs for evaluation.

ii. **Condition Evaluation:** For each (A, B) row pair, the `eval_condition_tokens` function is called. This function interprets a stack-based postfix representation of the join condition logic, supporting compound conditions with AND/OR operators.

iii. **Join Condition Matching:** The `eval_join_condition` function checks individual condition matches:

- **Numerical Conditions:** Operands are cast to doubles and compared.
- **Textual Conditions:** Operands are compared using a device-side `strcmp` function.

iv. **Result Storage:** If a pair satisfies the condition, it is copied into the result buffer using an atomic counter to safely increment the output index across threads.

b. **Optimization Techniques**

i. **Stack-Based Expression Evaluation:** The join condition is compiled into postfix notation and evaluated via a stack, enabling support for arbitrarily complex expressions with minimal branching.

ii. **Modular Condition Evaluation:**

The use of `JoinConditionToken` enables modular, scalable evaluation logic that can easily be extended to support more complex join types or operators.

iii. **Shared Memory Utilization:** Using input tile technique to partition `Table B` in shared memory to minimize global memory access, reducing latency.

Performance Analysis

Query Type	GPU Time (s)	CPU Time (s)	Table 1 Size	Table 2 Size	Table 3 Size
Get All Rows	4.9	1.62	264,821 Rows	————	————
Projection (Numeric)	7.44	1.77	————	330,609 Rows	————
Projection + Filter (Numeric)	5.70	1.79	264,821 Rows	————	————
Filter (String)	5.61	1.85	264,821 Rows	————	————
Projection + Complex Filter (String and Datetime)	5.65	2.01	264,821 Rows	————	————
Max Aggregate	7.65	1.89	————	330,609 Rows	————
Min Aggregate	7.55	2.92	————	330,609 Rows	————

Query Type	GPU Time (s)	CPU Time (s)	Table 1 Size	Table 2 Size	Table 3 Size
Sum Aggregate	7.51	1.78	———	330,609 Rows	———
Sorting descendingly (Datetime)	8.23	1.69	———	330,609 Rows	———
Sorting ascendingly (Numeric)	17.14	2.02	264,821 Rows	———	———
Basic Join (one condition)	32.52	2.02	264,821 Rows	330,609 Rows	———
Complex Join (two different conditions)	45.3	2.13	264,821 Rows	330,609 Rows	———
Projection + Filter (Numeric)	13.52	7.06	———	———	764,821 Rows
Projection + Complex Filter	12.26	3.34	———	———	764,821 Rows
Basic Join	62.5	4.15	264,821 Rows	———	764,821 Rows

📈 Observations

1. For **simple queries** or **small datasets**, the CPU can sometimes outperform the GPU due to **kernel launch overhead**.
2. For **Complex queries** or **big datasets**, **GPU** Speed up rate is higher than **CPU** rate .