PARALLEL COMPUTING

PROJECT DOCUMENTATION

Team 17

Name	Sec.	B.N	ID
Mohamed Yasser Mohamed	2	12	9211066
Mustafa Tarek Salah	2	22	9211178
Ahmed Bassem ELKady	1	7	9210048
Daniel Nabil Khalil	1	16	9210386

TABLE OF CONTENTS

PROJECT DESIGN

◆ DuckDBManager:

Handles DuckDB initialization, CSV loading, query analysis, execution plan traversal, and coordination of query processing.

Constructor

- DuckDBManager(...)
  - Initializes paths, database connection, and configuration flags.

Public Methods

- void InitializeDatabase()
  - Sets up DuckDB instance and creates a new connection.
- void LoadTablesFromCSV()
  - Scans the CSV directory and loads each file as a DuckDB table.
- void AnalyzeQuery(const std::string &query)
  - Parses the input SQL query and extracts the physical execution plan.
- ~DuckDBManager()
  - Cleans up resources and deallocates any dynamically allocated memory.

Private Methods

- std::vector<ColumnInfo> GetCSVHeaders(...)
  - Reads column names and types from a CSV file and identifies primary/foreign keys.
- std::string ConstructCreateTableQuery(...)
  - Builds a CREATE TABLE SQL statement from column metadata.
- void TopologicalSortTables(...)
  - Ensures tables are loaded in the correct order based on foreign key dependencies.
- void TraversePlan(duckdb::PhysicalOperator *op)
  - Recursively traverses the execution plan using DFS and triggers node execution.
- void ExecutePlan()
  - Executes each physical operator node based on its type (scan, join, etc.).
- void deleteLastTableScanned()
  - Frees memory from previously scanned tables to manage resources.

PROJECT DESIGN

Table:

Manages reading, batching, filtering, projecting, and joining data from CSV files. Also supports constructing new tables from joins.

Constructors

- Table(std::string FOLDER_PATH, const std::string &table_name, ...)
  - Loads the table from CSV, applies filter/projection, and creates batch files.
- Table(Table* table1, Table* table2, int* table1_indices, ...)
  - Used to construct a new table from the result of a join operation.
- Table(Table* table)
  - Copy constructor to duplicate an existing table.

Public Methods

- bool getTableBatch(const int &batch_idx)
  - Loads the data of a specific batch into memory.
- std::string getTableName(), int getNumColumns(), int getNumRows(), int getNumBatches(), void** getData(), char** getColumnNames(), void printData(), void writeDataToFile(std::string filename)
  - Getters functions.
- ~Table()
  - Destructor that cleans up all allocated memory.

Private Methods

- bool makeTableBatches(...)
  - Splits the data into batches, applies filters/projections, and stores them.
- bool makeBatches(...)
  - Handles reading the raw CSV and creating filtered batch files.
- bool makeBatchFile(...)
  - Writes a single batch of processed data to disk.
- bool checkConditions(...)
  - Evaluates whether a row satisfies the specified filter conditions.
- bool compareFloats(...), compareStrings(...), compareDateTime(...)
  - Utility methods for evaluating condition expressions based on data type.

PROJECT DESIGN

1) QUERY PARSING & GENERTAING EXECUTION PLAN:

In this project, we utilized DuckDB as the core engine for query processing. DuckDB provides a powerful in-process SQL database management system that enabled us to efficiently parse SQL queries and generate detailed execution plans. By leveraging DuckDB's capabilities, we were able to extract all necessary components involved in query execution, making it an ideal tool for analyzing and understanding query behavior in a lightweight and effective manner.

2) TRAVERSE THROUGH THE EXECUTION PLAN:

After retrieving the execution plan from DuckDB, we performed a Depth-First Search (DFS) traversal on the resulting plan tree. This traversal method allowed us to explore each branch of the execution plan in depth before backtracking, ensuring that all dependent operations were handled in the correct order. By visiting each node individually, we were able to isolate and execute each operation step-by-step—such as scans, filters, joins, and aggregations—according to the logical flow defined by the plan.

3) EXECUTE EACH NODE OPERATION:

3.1) SEQ SCAN & FILTER & PROJECTION:

In the Sequential Scan (Seq Scan) node, we implemented a class called Table to handle data access and processing. This class contains multiple methods designed to facilitate efficient table operations. When a new instance of the Table class is created, the constructor receives the table name as input. It then searches the data directory for a corresponding CSV file with the same name. Once found, it creates a new folder—named after the table—inside a directory called Tables. The original dataset is then split into multiple CSV files, each representing a batch of rows based on a predefined BATCH_SIZE. This batching process allows for more manageable and efficient access to data during query execution.
Additionally, when a Seq Scan node is encountered in the execution plan, we inspect its immediate child node to determine if it is a Filter or Projection operation. If so, we apply these operations during the scan itself, rather than in separate steps. This optimization enables us to apply filtering and projection simultaneously while iterating through the data, which reduces unnecessary data processing and ensures we only retrieve the relevant records.

PROJECT DESIGN

3.2) HASH JOIN & NESTED JOIN & CROSS PRODUCT:

For Hash Join, Nested Loop Join, and Cross Product operations, we implemented a custom join kernel to handle the join logic efficiently. This kernel receives pointers to the two most recently scanned tables, along with an optional array of join conditions. It then spawns multiple threads, where each thread is responsible for joining a specific pair of rows—one from each table. Before producing a joined result, the thread evaluates all the provided conditions by iterating through the condition array.

DuckDB provides various representations for join operations in the execution plan—sometimes embedding filter conditions within the join node itself, and other times placing them in a separate node above the join. To ensure consistent and correct behavior, we accounted for all these cases by integrating filter evaluation directly into the join process. This means that, regardless of where the filter appears in the plan, we apply it at the same time as the join is performed, minimizing unnecessary data movement and ensuring efficient execution.

PROJECT DESIGN

3.3) AGGREGATES:

Min/Max on Numeric Values

- **Algorithm Design**
  - Multi-Stage Reduction:
    - GPU-side initial reduction on data chunks
    - CPU-side final reduction on intermediate results

  - Key Optimizations:
    - CUDA streams for concurrent execution
    - Warp-level shuffle operations for efficient comparisons
    - Shared memory to combine warp results
    - Pinned memory for faster host-device transfers

- **Kernel Implementation**

The `findWarpMin` and `findWarpMax` kernels perform parallel reduction in three phases:

  - Thread-level: Each thread loads one element
  - Warp-level: Uses `__shfl_down_sync` for fast reduction within warps
  - Block-level: First warp reduces results from all warps in the block

- **Data Flow**
  - Column data is divided into chunks processed by separate CUDA streams
  - Each stream runs a reduction kernel producing partial results
  - Results from all blocks are copied back to host memory
  - CPU performs final reduction to determine global min/max

Integration

  - Process one batch of data per call
  - Create and manage CUDA streams for parallel execution
  - Return the min/max value for the processed batch
  - Do CPU reduction on the values returned by each batch and return the overall min/max

PROJECT DESIGN

Min/Max on DateTime Values

- **Algorithm Design**
  - Convert string datetime representations to numeric form (long long)
  - Perform parallel reduction on numeric values
  - Final CPU reduction to find global min/max

  - Key Optimizations:
    - Pinned memory for input data and block sums
    - Process data in batches with CUDA streams

- **Kernel Implementation**
  The sumKernel performs parallel reduction in multiple stages:
  - Thread-level:
    - Each thread processes multiple elements with strided access
    - Double-precision accumulation within each thread
    - Skip NULL values during accumulation
  - Warp-level:
    - Use __shfl_down_sync for efficient parallel reduction
    - Avoids shared memory bank conflicts
  - Block-level:
    - First thread in each warp writes to shared memory
    - Thread 0 accumulates warp results
    - Atomic addition to global sum

- **Data Flow**
  - Date strings are copied to contiguous pinned memory in chunks.
  - Conversion kernel transforms strings to int64 values
  - Reduction kernels find min/max values within each block
  - Block results are transferred back to host
  - CPU performs final reduction to determine global min/max

Integration
  - Process one batch of data per call to ExecuteMinMaxDate
  - Final CPU reduction handles special cases (invalid dates)
  - Returns the min/max value per batch as a 64-bit integer
  - Do CPU reduction on the values returned by each batch and return the overall min/max datetime

PROJECT DESIGN

SUM on Numeric Values

- **Algorithm Design**
  - Parallel reduction on GPU for each data batch
  - Double-precision accumulation for numerical stability
  - Final CPU reduction of batch results

  - Key Optimizations:
    - Pinned memory for input data and block sums
    - Process data in batches with CUDA streams

- **Kernel Implementation**
  The sumKernel performs parallel reduction in multiple stages:
  - Thread-level:
    - Each thread processes multiple elements with strided access
    - Skip NULL values during accumulation
  - Warp-level:
    - Use __shfl_down_sync for efficient parallel reduction
    - Avoids shared memory bank conflicts
  - Block-level:
    - First thread in each warp writes to shared memory
    - Thread 0 accumulates warp results
    - Atomic addition to global sum

- **Data Flow**
  - Float data is copied to pinned host memory in chunks.
  - Kernels calculate partial sums for each chunk
  - Block sums are transferred back to host
  - CPU performs final reduction across all blocks
  - Results from multiple batches are combined to get the Overall Sum

- **Integration**
  - Process one batch of data per call to ExecuteSumFloat
  - Final CPU reduction combines results from all batches
  - Return sum as double-precision value

PROJECT DESIGN

Count

- **Algorithm Design**
  - Process data in parallel to count non-NULL values
  - Each thread independently counts elements in its assigned range
  - Atomic operations to combine thread-level counts
  - Different implementations for various data types (string, float)

- Key Optimizations:
  - Type-Specific Processing:
    - String: Check for NULL, empty or whitespace-only strings
    - Float: Check for NaN values using standard functions
  - Memory Management:
    - Contiguous memory layout for string data
    - Pinned memory for float data
    - Stream-based processing for large datasets

- **Kernel Implementation**
  Two specialized kernels handle different data types:
  - countNonNullStringsKernel:
    - Each thread checks multiple strings using strided access
    - Uses isStringNull device function to properly validate strings
    - Thread-local accumulation before atomic update
  - countNonNullFloatsKernel:
    - Each thread processes multiple float values
    - Uses built-in isnan function for NULL detection
    - Atomic addition to global counter for thread-safe updates

- **Integration**
  - Process one batch of data per call to ExecuteCountString or ExecuteCountFloat
  - Batched processing for large datasets
  - Return unified count value for query result
  - Support for different column types with type-specific implementations

PROJECT DESIGN

3.4) SORTING:

Overall Approach

- Multi-stage merge sort algorithm
- Type-specific implementations (float, datetime)
- Stream-based parallel processing of large datasets
- Final merging of sorted chunks on GPU

Sorting Algorithm

- GPU Merge Sort Implementation:
  - Bottom-up merge sort approach
  - Sort chunks independently in parallel
  - Kernels that implement width-doubling strategy
- Per-Chunk Processing:
  a. Transfer chunk data to GPU asynchronously
  b. Execute merge sort on the GPU for each chunk
  c. Copy sorted indices and keys back to host
- Multi-Chunk Integration:
  - Merge separately sorted chunks using GPU
  - Maintain sort order across all chunks

Data Reordering

- Indirect Sorting Strategy:
  - Sort indices rather than directly moving data
  - Apply sorted order to all columns in table
  - Preserve relationships between columns
- Final Reorganization:
  a. Create backups of all columns in pinned memory
  b. Reorder all columns using sorted indices
  c. Handle different data types appropriately
  d. Use CPU for reordering

Memory Management

- Efficient Resource Handling:
  - Allocate device memory only for active chunks
  - Use pinned memory for host arrays

PROJECT DESIGN

4) STREAMS AND PINNED MEMORY USAGE:

4.1) STREAMS

- Concurrent Execution:
  - Created multiple CUDA streams (NUM_STREAMS)
  - Enabled overlapping of memory transfers and kernel executions
  - Allowed processing different data chunks simultaneously
- Data Partitioning:
  - Each stream handled an independent portion of the data
  - Balanced workload distribution across streams
- Execution Flow:
  - Divided work into stream-sized chunks
  - Performed asynchronous transfers to device memory
  - Launched kernels on appropriate streams
  - Asynchronously copied results of each stream back to host
  - Synchronized streams before final CPU processing for stream results

4.1) PINNED MEMORY

- Faster Memory Transfers:
  - Allocated host memory with cudaMallocHost instead of standard malloc
  - Enabled direct memory access (DMA) by GPU
  - Eliminated need for extra staging buffers in pageable memory
- Bandwidth Optimization:
  - Achieved higher bandwidth for host-device transfers
  - Reduced overhead from page faults and memory copies
- Asynchronous Operation Support:
  - Enabled true concurrency between CPU and GPU
  - Essential for stream-based execution model

EXPERIMENTS & RESULTS

	CPU	GPU	Speedup
Sort (10M)	37.265 s	28.016 s	1.33x
Join (10000*10000 =100M)	2.006 s	0.25 s	8.024x
MAX (N) (10M)	12.01 s	8.902 s	1.34x
MAX (D) (10M)	12.235 s	8.416 s	1.45x
SUM (N) (10M)	11.604 s	9.636 s	1.34x
COUNT(N) (10M)	10.651 s	9.407 s	1.212x
AVG(N) (10M)	11.675 s	10 s	1.18x

PERFORMANCE ANALYSIS

1) What are the CPU benchmarks ?

- Query Execution Time (Latency):
  - Time to complete a single query.
  - Often measured for different types of queries.

- Scalability with Dataset Size:
  - Performance with increasing number of rows (e.g., 1M, 10M, 100M, 1B records).

- CPU Utilization (%):
  - How much of the CPU is being used during queries.
  - Higher utilization may mean better use of resources, or CPU saturation.

- Throughput (Queries per Second - QPS):
  - Number of queries a system can handle per second under a given load.

2) What about their GPU counterparts ?

Query Execution Time (Latency): Measures kernel execution time plus memory transfer overhead between CPU and GPU memory, often separated to identify bottlenecks.

Scalability with Dataset Size: Performance curve across different data volumes, with special attention to the threshold where data exceeds GPU memory capacity.

PERFORMANCE ANALYSIS

3) How much is the speedup of the GPU over the CPU?

	CPU	GPU	Speedup
Sort (10M)	37.265 s	28.016 s	1.33x
Join (10000*10000 =100M)	2.006 s	0.25 s	8.024x
MAX (N) (10M)	12.01 s	8.902 s	1.34x
MAX (D) (10M)	12.235 s	8.416 s	1.45x
SUM (N) (10M)	11.604 s	9.636 s	1.34x
COUNT(N) (10M)	10.651 s	9.407 s	1.212x
AVG(N) (10M)	11.675 s	10 s	1.18x

PERFORMANCE ANALYSIS

4) How does this compare to the theoretical speedup ?

For Max Operation on GPU:

- Dataset Specifications
  - Size: 10 million float elements
  - Memory footprint: 10,000,000 × 4 bytes = 40 MB input data
- Memory Access Analysis
  - Global Memory Reads:
    - Each thread reads one float: 40,000,000 bytes total
  - Global Memory Writes:
    - Assuming BLOCK_SIZE = 1024:
    - Number of blocks: $\lceil 10{,}000{,}000 \div 1024 \rceil$ = 9,766 blocks
    - Block results written: 9,766 × 4 bytes = 39,064 bytes
  - Total Memory Traffic:
    - Read + Write: 40,039,064 bytes (~40.04 MB)
- Computation Analysis
  - Warp-Level Reductions:
    - Total warps: 10,000,000 ÷ 32 = 312,500 warps
    - Each warp performs $\log_2(32)$ = 5 reduction steps
    - Each step requires 2 operations (shuffle + max)
    - Warp reduction operations: 312,500 × 5 × 2 = 3,125,000 ops
  - Block-Level Processing:
    - One warp per block does final reduction: ~97,660 additional ops
    - Shared memory operations: ~9,766 writes

Op/Byte = Total operations / Total memory traffic
Op/Byte ≈ 3,222,660 / 40,039,064 3Op/Byte ≈ 0.08

PERFORMANCE ANALYSIS

4) How does this compare to the theoretical speedup ?

For Max Operation on CPU:

- Memory Access Analysis
  - Data reads: 10,000,000 × 4 bytes = 40 MB
  - No significant intermediate memory writes
- Computation Analysis
  - For each element:
    - One NaN check (std::isnan)
    - One comparison operation (std::min or std::max)
    - Total operations: 10,000,000 × 2 = 20,000,000 ops

Op/Byte = 20,000,000 / 40,000,000 = 0.5

CPU vs GPU Specifications

CPU

Bandwidth = Memory Clock × Memory Width × Channels × 2 (for DDR) / 8 (bytes)
Bandwidth = 3200 MT/s × 64 bits × 2 channels × 2 / 8
Bandwidth = 3200 × 64 × 2 × 2 / 8
Bandwidth = 3200 × 64 × 4 / 8
Bandwidth = 3200 × 64 × 0.5
Bandwidth = 3200 × 32
Bandwidth = 102,400 MB/s
Bandwidth ≈ 51.2 GB/s

GPU

GTX 1650 Ti Bandwidth: ~160 GB/s

PERFORMANCE ANALYSIS

4) How does this compare to the theoretical speedup ?

For a 10 million float element dataset (40 MB):

GPU Time:

- Time = 40 MB / 160 GB/s ≈ 0.25 ms

CPU Time:

- Time = 40 MB / 51.2 GB/s ≈ 0.78 ms

Speedup = 0.78 ms / 0.25 ms ≈ 3.1x
Our Speedup = 1.15x

PERFORMANCE ANALYSIS

5) If your speedup is below the theoretical one (this is mostly the case), how do you explain this, what could be changed to achieve a better one ?

The significant gap between theoretical (3.1x) and observed (1.34x) speedup can be attributed to several factors:

- **Additional Overheads in the GPU Implementation**
  - Multiple Kernel Launches: Our implementation uses multiple streams and kernels
  - Multi-phase Reduction: Two-phase reduction (per-block then final) adds overhead
  - Shared Memory Operations: Extra steps for shared memory usage

- **Data Transfer Overhead**
  - Transferring 40MB of data to and from the GPU can consume a significant portion of execution time
  - Memory Copy Operations: cudaMemcpy calls add substantial latency

- **GPU Execution Overheads**
  - Kernel Launch Latency: 5-10μs per kernel launch
  - Synchronization Costs: cudaDeviceSynchronize() and stream synchronization
  - Context Switching: Switching between CPU and GPU execution contexts

- **CPU Advantages Not Considered**
  - Branch Prediction: Modern CPUs efficiently predict the flow in simple loops

- **Real vs. Theoretical Bandwidth**
  - Memory Controller Efficiency: Both CPU and GPU achieve 60-70% of peak bandwidth in practice
  - Memory Latency: Raw bandwidth calculations don't account for access latency

To improve our speedup, we'll minimize data transfers by processing more operations per kernel, and consolidate to a single-phase reduction kernel.