A High-Performance Architecture for In-Memory Analytical Query Processing on Raw JSON Data

Part I: An Execution Plan for a High-Performance, In-Memory Analytical Query Engine

This document presents a comprehensive execution plan for the development of a high-performance, in-memory analytical query engine. The system is designed to address the challenge of processing millions of newline-delimited JSON (NDJSON) records, as specified in the project brief.[1] The plan prioritizes the key evaluation criteria: completion, performance optimization through parallelization and memory management, and resilience. It adheres to the primary constraints of prohibiting external data processing engines and implementing a custom file reading and parsing subsystem from scratch. The architecture detailed herein is engineered to achieve state-of-the-art performance on modern multi-core processors by adopting a cache-conscious philosophy, a columnar in-memory format, and a multi-layered parallelism strategy.

Section 1: Architectural Blueprint and Core Design Tenets

The foundation of any high-performance system lies in its architectural principles. These initial decisions regarding programming language, in-memory data representation, and interaction with the underlying hardware have profound and cascading effects on the system's ultimate performance, memory footprint, and implementation complexity. This section establishes the core tenets that will guide the engine's development.

1.1 Language Selection for Performance-Critical Data Systems: A Comparative Analysis

The project specification allows for the use of any programming language, making this the first and most critical decision point.[1] The choice directly dictates the ceiling for performance optimization, the available concurrency models, and the guarantees of memory and thread safety.

An analysis of leading contenders reveals a clear strategic choice. High-level languages such as Python, Java, or JavaScript, while dominant in data science and web development for their productivity and rich ecosystems [2], are fundamentally ill-suited for this task. Their reliance on interpreters, just-in-time (JIT) compilers, and automatic memory management (garbage collection) abstracts away the low-level control over memory layout and execution that is essential for building a truly high-performance engine from the ground up.[5]

The viable candidates are therefore systems-level languages that offer direct memory management and high-level abstractions without performance penalties.

- **C/C++:** The traditional languages for high-performance computing, C and C++ provide maximum control over system resources.[2] However, this control comes at the cost of manual memory management and a lack of built-in thread safety, which significantly increases development complexity and the likelihood of subtle, hard-to-debug errors like memory leaks, buffer overflows, and data races.[6]
- **Go:** A strong contender, Go is lauded for its simplicity and its highly effective concurrency model built on goroutines and channels.[7] Goroutines are lightweight, making it easy to spawn thousands for I/O-bound tasks. However, Go employs a garbage collector (GC). While highly optimized, the GC inevitably introduces non-deterministic pauses and overhead, which can become a significant bottleneck in applications with high allocation rates—a scenario guaranteed by the parsing of 20 million individual JSON objects.[10] For CPU-bound and memory-intensive analytical workloads, Go's performance, while good, is generally surpassed by languages that offer more granular control.[12]
- **Rust:** The prime candidate for this project. Rust provides performance competitive with C/C++ while guaranteeing memory and thread safety at compile time through its ownership and borrowing model.[8] This "fearless concurrency" eliminates entire classes of bugs common in parallel systems programming.[7] Crucially, Rust does not have a garbage collector, enabling predictable, low-latency performance and fine-grained control over memory layout. Its

"zero-cost abstractions" mean that high-level constructs like iterators compile down to machine code as efficient as hand-written C, and its support for low-level SIMD (Single Instruction, Multiple Data) intrinsics is robust and accessible.[15]

While the rapid development cycle of Go might seem attractive for a time-constrained challenge like a hackathon [12], the evaluation criteria explicitly prioritize "efficient use of parallelization," "appropriate memory management," and raw "execution time".[1] This shifts the strategic calculus away from development velocity and towards performance potential. The time invested in satisfying Rust's compiler is an upfront cost that pays dividends in performance and stability. It enables fearless, aggressive optimization and prevents subtle concurrency bugs that would be catastrophic under time pressure. For a challenge where performance is the primary metric, Rust's feature set aligns perfectly with the goal of achieving a top-tier score.

Decision: The query engine will be implemented in **Rust**. Its combination of bare-metal performance, memory safety, and powerful concurrency features provides the ideal foundation for meeting the project's stringent evaluation criteria.

Table 1: Comparative Analysis of Programming Languages for High-Performance Data Systems

Feature	Rust	Go	C++
Memory Management	Compile-time ownership & borrowing, no GC. Predictable memory usage and no pause times. [9]	Garbage Collection (GC). Simplifies development but can introduce non-deterministic latency. [10]	Manual (RAII, smart pointers). Powerful but prone to memory leaks and use-after-free errors. [6]
Performance Ceiling	Extremely high, comparable to C++. Zero-cost abstractions ensure maximum efficiency. [10]	High, but limited by GC and runtime overhead. Excellent for I/O-bound tasks. [12]	Extremely high. The historical benchmark for performance-critical software. [2]
Concurrency Model	Ownership ensures data-race-free concurrency at	Lightweight goroutines and channels managed	Standard library threads, mutexes, atomics. Requires

	compile time. Supports OS threads and async/await. [7]	by the runtime. Simple and highly scalable for I/O. [7]	careful manual synchronization. [6]
Safety Guarantees	Strongest guarantees. Memory and thread safety are enforced by the compiler. [8]	Memory safe due to GC. Built-in race detector helps find concurrency bugs at runtime. [12]	Unsafe by default. Relies on developer discipline and tools like sanitizers to find errors. [6]
Low-Level Optimization (SIMD)	Excellent support for architecture-specific intrinsics via core::arch. [16]	Limited native support; often requires assembly or cgo for direct SIMD access. [19]	Excellent support for intrinsics, providing maximum control over vectorization. [20]
Ecosystem for Data Engineering	Growing rapidly with high-performance libraries like Polars and Arrow. Strong focus on performance. [5]	Mature ecosystem for cloud-native tools and networking (e.g., Kubernetes, Docker). [8]	Vast and mature, but can be fragmented. Many high-performance libraries available. [2]
Development Velocity	Steeper learning curve due to ownership model. Slower initial development but higher long-term robustness. [8]	Very fast development cycle. Simple syntax and tooling make it easy to learn and be productive. [11]	Complex language with a steep learning curve. Development can be slow and error-prone. [6]

1.2 The Case for Columnar Storage: Optimizing In-Memory Layout for Analytical Queries

Once parsed, the 20 million taxi trip records must be held in memory for querying. The layout of this in-memory data is the single most important architectural decision for achieving high analytical query performance. The choice lies between a traditional row-oriented layout and a column-oriented layout.

The queries specified in the project brief are classic Online Analytical Processing (OLAP) workloads.[1] They are characterized by scans over a large number of records

while accessing only a small subset of the available columns. For instance, Query 2 requires only

Trip_distance, Payment_type, Fare_amount, and Tip_amount out of the nine available fields.

A **row-oriented (or "row store")** layout, where all fields for a single record are stored contiguously in memory, is highly inefficient for such workloads.[21] To read the

Trip_distance for each record, the CPU would be forced to load the entire record—including unused columns like VendorID and tpep_dropoff_datetime—from main memory into the CPU caches. This phenomenon, known as cache pollution, wastes precious memory bandwidth and evicts potentially useful data from the cache, leading to severe performance degradation.[23]

A **column-oriented (or "columnar")** layout, by contrast, stores all values for a single column contiguously. This approach is vastly superior for OLAP workloads for three primary reasons [21]:

1. **Selective Data Retrieval:** Queries only need to read the memory corresponding to the columns they access. For Query 2, only the four required columns would be scanned, dramatically reducing the amount of data transferred from main memory to the CPU.[21]
2. **Enhanced Data Compression:** Data within a single column is homogeneous (all values are of the same type) and often exhibits lower cardinality than the dataset as a whole (e.g., Payment_type has only a few distinct values). This makes it highly amenable to compression techniques like dictionary encoding or run-length encoding, which can significantly reduce the in-memory footprint of the data.[22]
3. **Vectorized Processing (SIMD):** The contiguous, homogeneous layout of data in a column is the ideal structure for applying SIMD instructions. A single instruction can perform an operation (e.g., an addition or comparison) on a vector of values loaded from a column, yielding massive data-level parallelism and performance gains.[25]

Decision: The engine will implement a **columnar in-memory data store**. Each field from the JSON schema will be deserialized into its own dedicated, contiguous memory buffer (represented in Rust as a Vec<T>, where T is the appropriate primitive type like f64 or i32).

1.3 A Cache-Conscious Philosophy: Designing for Modern Multi-Core CPU Architectures

For any in-memory data processing system, the primary performance bottleneck is no longer disk I/O but the latency and bandwidth of the CPU-memory interface.[25] High performance is therefore synonymous with efficient utilization of the CPU's multi-level cache hierarchy (L1, L2, L3). The entire design of the query engine will be guided by a cache-conscious philosophy.

- **Maximizing Data Locality:** The columnar storage layout is the cornerstone of this philosophy. By iterating through data column by column, the engine ensures high spatial locality. When the CPU fetches a 64-byte cache line from memory, it will contain multiple useful data points from the same column, maximizing the work done per memory access.[25] This stands in stark contrast to the random-access memory patterns often induced by row-based object stores.
- **Minimizing Pointer Chasing:** The use of simple, contiguous arrays (Vec<T>) for each column avoids the pointer indirection inherent in storing complex objects. Pointer chasing is a primary cause of cache misses, as it forces the CPU to jump to disparate memory locations to assemble a single logical record.
- **NUMA (Non-Uniform Memory Access) Awareness:** On modern multi-socket servers, a CPU core can access its local memory bank faster than a remote bank attached to another CPU.[25] The parallelism strategy, detailed in Section 4, will incorporate this by partitioning data and scheduling computational tasks on cores with affinity to the memory they need to access, minimizing costly cross-socket data transfers.[27]
- **Optimizing Instruction Locality:** Performance is also affected by the instruction cache. Complex control flow with deep, recursive function calls inside tight processing loops can lead to instruction cache misses. The implementation will favor iterative, batch-oriented processing loops to keep the critical code path small and hot in the instruction cache.[25]

Section 2: The Ingestion Subsystem: High-Throughput Parsing of Newline-Delimited JSON

The project's most significant constraint—and therefore its greatest opportunity for innovation and competitive advantage—is the requirement to build the JSON file reader and parser from scratch.[1] A naive, single-threaded, character-by-character parser would be exceptionally slow and would fail to meet the performance goals. This plan outlines a multi-stage, parallel, and low-level ingestion pipeline designed to process the 20 million record dataset at gigabytes per second, drawing inspiration from state-of-the-art parsing techniques.

The performance of this subsystem is paramount. The evaluation criteria for the challenge place a heavy emphasis on execution time, and for a data-intensive task of this scale, the initial data loading and parsing phase is often the dominant contributor to total runtime.[1] Standard library parsers are often not optimized for maximum throughput, typically involving excessive memory allocations and inefficient, single-threaded processing logic.[29] Research into specialized parsers like

simdjson has demonstrated that by leveraging modern CPU features, it is possible to achieve a greater than 10x speedup over conventional methods.[30] Therefore, implementing a custom parser based on these advanced principles is not merely an academic exercise; it is the most direct and impactful strategy for achieving a winning performance in the challenge.

2.1 Stage 1 - Parallel I/O: Chunking and Reading the Source File

The first bottleneck to overcome is reading the large data file from storage into memory. A single-threaded approach would be limited by the speed of a single CPU core managing I/O operations.

- **Strategy:**
    1. **Memory-Mapped Files:** Instead of issuing explicit read() system calls, the engine will use a memory-mapped file (mmap). This instructs the operating system to map the file's contents directly into the application's virtual address space. The OS then handles paging the data into physical RAM on-demand as it is accessed. This is a highly efficient mechanism for large files, as it avoids extra data copies between kernel and user space and leverages the OS's sophisticated page cache.[32]
    2. **Parallel Chunking and Re-alignment:** To parallelize the workload, the memory-mapped region will be divided into large, roughly equal-sized

chunks, with one chunk assigned to each available CPU core. A naive division is problematic, as it could split a JSON record in the middle. To solve this, a "chunk-and-realign" strategy will be used: each worker thread receives its assigned chunk and scans forward from the very first byte to find the next newline character (\n). This marks the true beginning of its workload. The data before this first newline belongs to the previous worker's chunk. This simple and robust technique ensures that every worker operates on a set of complete, independent JSON lines, enabling flawless parallel processing from the outset.[32] Both Go and Rust provide the necessary concurrency primitives to implement this pattern effectively.[34]

2.2 Stage 2 - Lexical Analysis with SIMD: A "From-Scratch" simdjson-inspired Approach

With file reading parallelized, the next bottleneck is the CPU-intensive task of parsing the text. This is where the custom parser's design will provide a decisive performance advantage. The approach will be a simplified version of the revolutionary two-stage architecture pioneered by the simdjson library.[28]

- **Strategy (The Two-Stage Parser):**
  1. **Stage 1 (Structural Indexing):** This stage rapidly identifies the location of all structurally important characters. Each worker thread processes its data chunk in large blocks (e.g., 64 bytes at a time). Using SIMD intrinsics available in Rust's core::arch module (specifically AVX2 instructions), the thread can perform parallel byte-level comparisons. In a single CPU instruction, a 64-byte vector of input data can be compared against a vector filled with {, another filled with }, another with ", and so on. The results of these parallel comparisons are combined into bitmasks that serve as an index of all structural characters ({}:,) and special characters (", \) within the chunk.[30] This SIMD-based structural identification is orders of magnitude faster than a traditional scalar loop that inspects one character at a time.
  2. **Stage 2 (Hierarchical Parsing):** A second-stage parser then operates on this structural index, not the raw byte stream. This parser is a more conventional state machine, but its task is simplified and accelerated. It navigates the JSON structure by jumping between the pre-identified character locations, validating the hierarchy (e.g., ensuring braces and brackets are matched) and identifying the boundaries of keys and values.

2.3 Stage 3 - Zero-Allocation Value Parsing

After the structure of a JSON object is identified, its literal values (strings, numbers, booleans) must be parsed and converted into their native types. This stage must be executed with a strict zero-allocation policy to maintain high performance. Standard library conversion functions often allocate temporary memory, which would introduce significant overhead when repeated 20 million times.[28]

- **Strategy:**
  - **String Parsing:** Strings will be handled as "zero-copy" slices (&str in Rust). The parser will identify the start and end quotes of a string value within the original memory-mapped buffer and create a view (a slice) of that memory region. No new memory is allocated for the string content itself. The only potential modification is handling escaped characters (e.g., \", \\), which can be done by writing to a small, reusable buffer only when escapes are detected.
  - **Number Parsing:** Custom, high-performance integer and floating-point parsing routines will be implemented. These routines will operate directly on the byte slice representing the number, using simple iterative arithmetic (multiplication and addition) to construct the numeric value. This avoids the function call overhead and potential allocations of generic library functions like strconv.ParseFloat or strtod.[28]
  - **Minimizing Allocations:** The guiding principle for this stage is the aggressive reduction of heap allocations. By avoiding allocations, the engine reduces pressure on the memory management subsystem and improves data locality, leading to better cache performance and lower overall runtime.[29]

2.4 Stage 4 - Data Transformation: Transposing to Columnar Buffers

As each worker thread parses its JSON objects, the extracted values must be efficiently placed into the final columnar data store.

- **Strategy:**
  1. **Pre-allocation:** Before parsing begins, the main thread will pre-allocate the

global columnar buffers (e.g., Vec<f64> for trip_distance) with a capacity sufficient for all 20 million records. This single, large allocation at the start prevents a cascade of smaller, inefficient re-allocations and memory copies as the vectors grow during ingestion.

2. **Thread-Local Buffers and Scatter-Gather:** To avoid synchronization bottlenecks where multiple threads try to write to the same global buffers concurrently, each worker thread will write its parsed data to a set of *thread-local* columnar buffers. This "scatter" phase proceeds with no locking or contention.
3. **Final Concatenation:** Once all worker threads have completed parsing their chunks, a final, single-threaded "gather" step will iterate through the thread-local buffers and append their contents to the corresponding global columnar buffers. This approach confines contention to a short, final merge step, maximizing parallelism during the expensive parsing phase.

Section 3: The Query Execution Subsystem: A Parallelized, Volcano-Style Model

With the data ingested and organized in a columnar format, the execution subsystem is responsible for running the four predefined queries.[1] The design will be based on a parallelized and vectorized "Volcano-style" iterator model, a standard and highly composable architecture for database query engines.[40]

3.1 Physical Operator Design

Execution is modeled as a tree of physical operators, where each operator implements an next() method that produces a batch of results. This allows for complex queries to be composed from simple, reusable components.

- **Core Operators:**
  - **ColumnScan:** The leaf operator in a plan. Its next() method reads a batch of values from a specified columnar buffer.
  - **Filter:** Consumes batches from an input operator. It applies a boolean predicate (e.g., trip_distance > 5.0) to each value and produces an output batch containing either the surviving values themselves or, more efficiently, a

bitmask or a vector of indices corresponding to the surviving rows.
    - **Projection:** Takes a batch of row indices from a Filter or Scan operator and gathers the corresponding values from one or more columns to produce output tuples.
    - **HashAggregate:** The workhorse for queries involving GROUP BY. It consumes input tuples and builds a hash table. The keys of the hash table are the grouping columns (e.g., Payment_type or VendorID), and the values are structs containing the intermediate state for each aggregate (e.g., {count, sum_fare, sum_tip}).
    - **SimpleAggregate:** A specialized operator for aggregations without a GROUP BY clause, such as counting the total number of records for Query 1.

3.2 Vectorized Execution within Operators: Leveraging SIMD for Data-Level Parallelism

To achieve maximum performance, operators will not process data one tuple at a time. Instead, they will be "vectorized" or "batch-oriented," operating on arrays of data within their next() method. This design improves cache performance by amortizing function call overhead and, most importantly, enables the use of SIMD instructions for data-level parallelism.[25]

- **Strategy:**
    - All operators will process data in fixed-size batches (e.g., 1024 or 4096 elements).
    - **SIMD-Accelerated Filtering:** The Filter operator will load a vector of data from a column (e.g., eight f64 trip distances into a 512-bit AVX-512 register) and compare them against a threshold value in a single instruction. This produces a bitmask indicating which elements passed the filter, which can then be used to compact the output or generate a list of indices.[20] This is vastly more efficient than a scalar
    if statement inside a loop.
    - **SIMD-Accelerated Aggregation:** Simple aggregations like SUM are prime candidates for SIMD. For example, a 512-bit register can hold eight f64 values. The operator can maintain eight partial sums in parallel within a single register, accumulating values from the input vector. These partial sums are then horizontally added together at the end of the batch to produce a final sum.[20]

3.3 High-Performance Group-By and Aggregation

Queries 2, 3, and 4 all depend on efficient GROUP BY and aggregation, which is often the most computationally intensive stage of an OLAP query.[1] The strategy will focus on parallelizing the hash table construction and vectorizing the aggregation updates.

- **Strategy:**
    1. **Parallel Hash Table Construction:** The most significant optimization is to parallelize the creation of the aggregation hash table. The input data will be partitioned based on the hash of the grouping key. Each worker thread will be assigned a partition of the data and will build its own, independent, thread-local hash table. This design completely avoids the massive synchronization overhead and contention that would arise from having all threads attempt to update a single, shared hash table.[25]
    2. **Vectorized Updates:** Within each thread, as it processes its partition, the aggregation logic will be vectorized where possible. For each row that maps to a hash table entry, the operator will update all necessary aggregate states simultaneously (e.g., for Query 2, it will increment the count, add to sum_fare, and add to sum_tip).
    3. **Final Merge Phase:** After all threads have completed building their local hash tables, a final, fast, single-threaded step is required. This step iterates through all the thread-local hash tables and merges them into the final global result set. Since the number of unique groups is typically small (e.g., a handful of payment types or vendors), this merge phase is very fast compared to the initial parallel build phase.
    4. **Deferred Average Calculation:** The AVG aggregate is never calculated during the main aggregation loop. The engine only maintains the SUM and COUNT. The final division (SUM / COUNT) is performed only once per group during the final output formatting stage, preventing redundant floating-point divisions.

Section 4: A Multi-Layered Parallelism Strategy

A coherent parallelism strategy is essential to fully exploit the capabilities of modern

multi-core hardware. The engine will employ multiple forms of parallelism, primarily focusing on data parallelism, which offers the highest return on investment for the specified workloads.

4.1 Intra-Operator Parallelism (Data Parallelism)

This is the most fundamental and impactful form of parallelism for this engine. It involves executing a single operator, such as a filter or aggregation, across multiple threads, with each thread working on a different partition of the data.[40]

- **Implementation:** The columnar data store is inherently partitionable. The full set of 20 million rows can be divided into contiguous ranges of indices. A worker thread is then assigned a specific range to process. In Rust, the rayon library provides a high-level, data-parallelism framework that makes this straightforward. A call to .par_iter() on a range of row indices can automatically distribute the workload across a managed thread pool, abstracting away the complexities of thread creation and scheduling.[32] This strategy will be the default for all major data-processing operators, including
  ColumnScan, Filter, and the initial build phase of the HashAggregate.

4.2 Inter-Operator Parallelism (Pipeline Parallelism)

This more complex form of parallelism involves executing different operators in a query plan concurrently, creating a "pipeline" where data flows from one operator to the next as it is produced.[40] For example, a

Filter operator could begin processing the first batch of data produced by a Scan operator while the Scan is already working on the second batch.

- **Implementation:** While true pipeline parallelism can improve performance in very complex queries with multiple independent branches, it introduces significant implementation complexity, requiring sophisticated mechanisms for synchronization and backpressure management. The queries specified for this challenge are linear and relatively simple (e.g., Scan -> Filter -> Aggregate).[1] In such cases, the performance gains from pipelining are marginal compared to the

massive gains from robust intra-operator parallelism. The engineering effort required to build a full pipeline scheduler is not justified for this specific problem. Therefore, the engine will adopt a simpler and more robust **"materialization barrier"** model. Each operator will be executed to completion using intra-operator parallelism, and its full set of results will be "materialized" into an intermediate buffer (e.g., a Vec<u32> of filtered indices). The subsequent operator in the plan will then consume this materialized result, again executing in parallel. This approach sacrifices the latency benefits of true pipelining but is far simpler to implement correctly and robustly, while still achieving a very high degree of parallelism.

4.3 Thread and Resource Management for Optimal Scalability

Simply increasing the number of threads does not guarantee a linear performance improvement. Beyond a certain point, contention for shared hardware resources, such as the memory bus and CPU caches, as well as software synchronization overhead, can lead to diminishing or even negative returns, a principle described by Amdahl's Law.[26]

- **Strategy:**
  - **Thread Pool Sizing:** A global thread pool will be used, with its size configured to match the number of *physical* CPU cores on the host machine. This avoids the overhead of OS context switching that comes from oversubscribing the cores with more threads than they can run simultaneously.
  - **NUMA-Aware Data Partitioning:** On systems with multiple CPU sockets, the initial data will be partitioned across the NUMA nodes if possible. The thread pool will be configured to ensure that threads have affinity to the CPU core and memory node where their assigned data partition resides. This minimizes slow, remote memory accesses.[25]
  - **Work Granularity:** The chunking strategies for both file reading (Section 2.1) and query processing will be designed to ensure that each thread receives a substantial block of work. This amortizes the fixed overhead of thread scheduling and coordination, ensuring that threads spend the vast majority of their time performing useful computation rather than waiting.

Section 5: Ensuring Correctness and Resilience

A production-quality engine must be resilient to the inconsistencies and imperfections of real-world data. The evaluation criteria explicitly require correct handling of null values, inconsistent formats, and other edge cases.[1]

5.1 Strategies for Handling Data Heterogeneity

- **Null Values:** The columnar store will natively support nullability. In Rust, this is achieved elegantly using the Option<T> enum, which makes nullability an explicit, compile-time-checked part of the type system. A column of nullable integers would be stored as a Vec<Option<i32>>. All query operators, particularly aggregation functions, will be implemented to handle None values correctly according to standard SQL semantics (e.g., SUM and AVG ignore nulls, while COUNT(*) includes them).
- **Inconsistent Date Formats:** The custom datetime parser will be designed for robustness. It will first attempt to parse the primary expected format (e.g., YYYY-MM-DDTHH:MM:SS). If this fails, it can have a series of fallbacks for other common ISO 8601 variants. If all parsing attempts fail, the value will be treated as null for that record, and an error can be logged.
- **Type Mismatches:** The custom JSON parser must be strict. If it expects a numeric value for a field like passenger_count but encounters a string (e.g., "one"), it will not attempt a complex conversion. It will treat the value as invalid, record it as null in the columnar store, and potentially increment an error counter.
- **Duplicates:** The specified queries do not require any special handling for duplicate records. The engine will process them as they appear in the dataset.

5.2 Maintaining Numerical Precision and Managing Edge Cases

- **Numerical Precision:** The project brief requires floating-point output to at least two decimal places.[1] All monetary values (fare_amount, tip_amount) and distance calculations will use 64-bit floating-point numbers (f64) throughout the query pipeline to minimize precision loss. A more

robust, albeit more complex, approach for financial data is to parse monetary values into a fixed-point decimal type or directly into an integer representing the number of cents. All intermediate calculations would then be performed using integer arithmetic, which is exact. The final conversion to a floating-point dollar amount would only occur at the very end for display purposes.

- **Aggregation Edge Cases:** The implementation of aggregation functions must be mindful of edge cases. For example, the AVG function, calculated as SUM / COUNT, must handle the case where a group has a COUNT of zero. A check for division-by-zero is mandatory to prevent a program crash. In such a case, the result for AVG should be defined as 0 or null, consistent with standard database behavior.

Part II: A Blueprint for Academic Contribution: Structuring the Research Paper

The successful implementation of the high-performance query engine described in Part I provides the foundation for a significant academic contribution. This section outlines the structure and content of a research paper suitable for submission to a top-tier computer systems or database conference, such as ACM SIGMOD, VLDB, or a journal like Proceedings of the ACM on Measurement and Analysis of Computing Systems (POMACS).[47] The goal is to reframe the engineering effort as a rigorous scientific investigation.

Section 1: Introduction and Motivation

The introduction must establish the context, identify a critical problem, and clearly articulate the paper's novel contributions to solving that problem.

1.1 The Problem: The Performance Gap in Ad-Hoc Analytics on Raw Data Formats

The paper will begin by highlighting the growing industry trend of performing analytical queries directly on semi-structured data formats like JSON and Parquet. This trend is driven by the desire to reduce the latency and complexity of traditional ETL (Extract, Transform, Load) pipelines that require moving data into a rigid, structured data warehouse before it can be analyzed.[48] The core problem is identified as a performance gap: existing general-purpose data processing tools are often too slow or resource-intensive for interactive, ad-hoc querying, while building custom, high-performance solutions from scratch is a formidable engineering challenge. This paper explores an architectural sweet spot that combines bespoke, high-performance components with modern systems programming paradigms to bridge this gap.

1.2 Related Work: A Critical Review of Data Processing Systems

To establish novelty, the proposed engine must be positioned within the landscape of existing research and commercial systems.

- **Large-Scale Distributed Systems:** The paper will briefly review the architectures of scale-out systems like Apache Spark and Hadoop. It will acknowledge their strengths in fault tolerance and processing petabyte-scale, disk-resident data but argue that their architectural overhead (distributed schedulers, network communication, JVM overhead) makes them unsuitable for the specific problem of high-performance, single-node, in-memory analytics.[49]
- **High-Performance In-Memory DataFrame Libraries:** The work will be compared to modern, high-performance libraries such as Polars, DuckDB, and Cylon.[50] These systems will be acknowledged as the state-of-the-art in performance. The paper's contribution will be framed not as necessarily outperforming them, but as providing a "glass box" view by building core components from first principles. By implementing the parser and execution engine from scratch, this work can dissect and quantify the performance impact of specific architectural choices (e.g., SIMD parsing, columnar layout) in a way that is not possible when using monolithic, black-box dependencies.[1]
- **Academic Research in In-Memory Databases:** The paper will cite foundational surveys on in-memory data management to ground its design in established principles.[25] It will reference the importance of cache-conscious algorithm design, the different forms of parallelism (intra-operator, inter-operator), and the challenges of concurrency control on multi-core systems.[25] This demonstrates an understanding of the academic lineage of the problem.

1.3 Our Contribution

The introduction will conclude with a clear, concise, and defensible list of the paper's contributions:

1. The design, implementation, and evaluation of a novel in-memory analytical query engine, built from the ground up in Rust, demonstrating a vertically integrated approach to high-performance data processing.
2. A high-throughput, from-scratch NDJSON parsing subsystem that leverages a simdjson-inspired, two-stage architecture using SIMD instructions for lexical analysis and employs zero-copy techniques for value parsing.
3. A comprehensive performance evaluation on a multi-core machine that quantifies the end-to-end performance and scalability of the engine, isolating the benefits of the custom parser and the parallel execution strategy.
4. A micro-architectural analysis that provides deep insights into the system's interaction with the underlying hardware, measuring metrics like cache utilization and instructions per cycle to explain the sources of performance.

Section 2: System Design and Implementation

This section provides the core technical details of the system, presented with sufficient rigor for an expert audience to understand and potentially replicate the work.

2.1 A Formal Description of the Engine's Architecture

A high-level architectural diagram will be presented, illustrating the primary components: the Parallel I/O Manager, the SIMD-Accelerated Parsing Pipeline, the Columnar Buffer Manager, and the Parallel Query Executor. The diagram will trace the flow of data from the raw NDJSON file on disk, through the parsing and transposition stages, into the in-memory columnar store, and finally through the query operator

tree to produce a result.

2.2 Justification of Key Design Decisions

This subsection will formalize the architectural arguments made in Part I. It will present a reasoned defense for the choice of Rust (zero-cost abstractions, safety, performance), the columnar in-memory layout (its natural fit for OLAP workloads and modern CPUs), and the materialization-barrier parallelism model (as a pragmatic trade-off between implementation simplicity and high performance for the target queries).

2.3 Novel Techniques and Implementation Highlights

This is the technical heart of the paper and will focus on the most innovative aspects of the implementation.

- **The SIMD-Accelerated Parser:** This will be the centerpiece. The two-stage (structural indexing and hierarchical parsing) process will be detailed with pseudo-code and diagrams. The specific SIMD instructions used (e.g., AVX2's _mm256_cmpeq_epi8 for parallel byte comparison) and how their results are combined into bitmasks will be explained.
- **Parallel Hash Aggregation:** The implementation of the HashAggregate operator will be described in detail, focusing on the parallel build phase using thread-local hash tables and the final merge phase. The data structures used for the hash table and the aggregate state will be specified.
- **Parallel File Ingestion:** The "chunk-and-realign" strategy for parallel reading from a memory-mapped file will be formally described.

Section 3: Experimental Methodology and Evaluation

This section presents the empirical evidence to support the paper's performance claims. The methodology must be scientifically rigorous and the results presented

clearly and honestly.

3.1 Experimental Setup

- **Hardware:** The exact specifications of the test machine will be provided: CPU model, number of cores and threads, base and boost clock speeds, L1/L2/L3 cache sizes, total RAM capacity and speed, and storage device type (e.g., NVMe SSD).
- **Dataset:** The 20-million-row taxi trip dataset will be described, including its on-disk size in GB and the full schema of the JSON records.[1]
- **Systems for Comparison (Baselines):** To contextualize the engine's performance, it will be benchmarked against well-understood baselines:
    1. **High-Level Scripting Baseline:** A Python script using the highly optimized polars library to perform the same queries. This represents the state-of-the-art for ease-of-use and high-level data manipulation.
    2. **Standard Parser Baseline:** A version of the engine where the custom SIMD parser is replaced by Rust's standard, high-quality serde_json library. This allows for a direct, apples-to-apples measurement of the performance gain attributable *specifically* to the novel parsing subsystem.

3.2 Performance Analysis: End-to-End Query Latency and Throughput

The primary results will be presented in a comprehensive table showing the end-to-end wall-clock execution time for each of the four queries. The table will compare the proposed engine against both baselines. Throughput will be reported in both rows per second and megabytes of raw input data processed per second.

3.3 Scalability Analysis: Measuring Performance as a Function of Core Count

To evaluate the effectiveness of the parallel architecture, the engine will be benchmarked while varying the number of active worker threads (e.g., 1, 2, 4, 8, 16, up to the total number of physical cores). The results will be presented as a speedup plot,

where speedup is defined as T1/TN (time on 1 core / time on N cores). This plot will be compared against the ideal linear speedup line to visually assess the parallel efficiency of the system and identify the onset of contention or other scalability bottlenecks.[26]

Table 2: End-to-End Performance and Scalability Results

Query	System	1 Core	4 Cores	8 Cores	16 Cores
		Time (s)	**Time (s)**	**Time (s)**	**Time (s)**
Query 1	**Our Engine**	*[data]*	*[data]*	*[data]*	*[data]*
(Record Count)	Baseline (serde_json)	*[data]*	*[data]*	*[data]*	*[data]*
	Baseline (polars)	*[data]*	*[data]*	*[data]*	*[data]*
Query 2	**Our Engine**	*[data]*	*[data]*	*[data]*	*[data]*
(Distance Filter)	Baseline (serde_json)	*[data]*	*[data]*	*[data]*	*[data]*
	Baseline (polars)	*[data]*	*[data]*	*[data]*	*[data]*
Query 3	**Our Engine**	*[data]*	*[data]*	*[data]*	*[data]*
(Date Filter)	Baseline (serde_json)	*[data]*	*[data]*	*[data]*	*[data]*
	Baseline (polars)	*[data]*	*[data]*	*[data]*	*[data]*
Query 4	**Our Engine**	*[data]*	*[data]*	*[data]*	*[data]*
(Daily Stats)	Baseline (serde_json)	*[data]*	*[data]*	*[data]*	*[data]*

	Baseline (polars)	*[data]*	*[data]*	*[data]*	*[data]*

Note: The table would also include columns for Memory Usage (GB) and Throughput (rows/s) for each configuration to provide a complete picture.

3.4 Micro-architectural Analysis

To move beyond simple timings and explain *why* the system performs as it does, hardware performance counters will be collected using a profiling tool like perf on Linux. This analysis will report key metrics during the execution of a representative query (e.g., Query 2):

- **Instructions Per Cycle (IPC):** A measure of CPU efficiency. A higher IPC indicates the CPU is performing more useful work per clock cycle.
- **Cache Miss Rates (L1, L2, L3):** The percentage of memory accesses that are not satisfied by a given level of the cache. Low miss rates are a direct validation of the cache-conscious design philosophy.
- **Memory Bandwidth Utilization:** The rate at which data is being read from main memory. This can help identify if the system is bottlenecked by the memory bus.

These low-level metrics provide the "ground truth" that connects the architectural design choices (like the columnar layout) to the observed performance results.[25]

Section 4: Discussion

This section interprets the experimental results, connecting them back to the system's design and acknowledging the work's limitations.

4.1 Interpretation of Results

The discussion will analyze the data presented in Section 3. For example, it might

state: "The results in Table 2 show that our engine outperforms the serde_json baseline by an average of 8x across all queries. Micro-profiling reveals that this gain is almost entirely due to the SIMD-accelerated parser, which reduced the data ingestion phase from 25 seconds to under 3 seconds. The scalability plot shows near-linear speedup up to 8 cores, after which performance begins to plateau. Micro-architectural analysis suggests this is due to saturation of the available memory bandwidth, indicating that for this workload on this hardware, the system becomes memory-bound rather than CPU-bound beyond 8 cores." This level of analysis demonstrates a deep understanding of the system's behavior.[46]

4.2 Limitations and Avenues for Future Work

No research is without limitations. The paper will honestly acknowledge the scope of the work. For example, the engine only supports a small, fixed set of queries and has no query optimizer. It is a single-node system and lacks the fault tolerance of production distributed systems.

This leads naturally to proposing avenues for future research, such as:

- Developing a cost-based query optimizer that can generate and choose between different parallel execution plans.[40]
- Extending the engine to a distributed, shared-nothing architecture to scale beyond a single machine.[52]
- Investigating the use of more advanced compression schemes within the columnar store.
- Expanding the query capabilities to support a larger subset of the SQL language, including complex joins and subqueries.

Section 5: Conclusion

The paper will conclude with a concise summary of its findings and a restatement of its core contributions.

5.1 A Summary of the Research Findings and Reiteration of Contributions

This final section will briefly summarize the problem of ad-hoc analytics on raw data, the design of the proposed high-performance query engine, and the key experimental findings. It will then crisply reiterate the main contributions—the design of the vertically integrated engine, the novel SIMD parser, and the comprehensive performance and scalability analysis—to leave the reader with a clear understanding of the paper's impact and significance in the field of high-performance data systems.