



PERFORMANCE IMPACT OF SECONDARY MEMORY ALLOCATION IN ARRAY REVERSAL OPERATIONS: AN EMPIRICAL CASE STUDY OF `ARRAY_REVERSE_UDF` VERSUS BUILT-IN `REVERSE()`

Cadet S. Navanitha Krishnan¹, Cadet S. Hari Santh²

¹Roll No. 6792, ²Roll No. 7162

Class XII (2026-27), Sainik School Amaravathinagar

Post: Amaravathinagar, Udumalpet Taluka, Tirupur Dt, Tamilnadu State

Guide & Mentor: Mr. Praveen Kumar Murigeppa Jigajinni, PGT Computer Science

ABSTRACT

Database and array-processing engines routinely offer a built-in `REVERSE()` operator alongside the option for developers to supply a user-defined function (UDF). Practitioner folklore holds that a hand-written, in-place UDF that performs no secondary heap allocation should outperform an allocating built-in equivalent, yet this claim is rarely isolated and measured in a controlled setting. This paper reports a case study that implements `ARRAY_REVERSE_UDF`, an in-place, $O(1)$ -auxiliary-space reversal routine, against an allocating, builtin-style `REVERSE()` that returns a newly allocated buffer, and benchmarks both in C (GCC 13.3.0, -O2, Ubuntu 24.04.4 LTS) across array sizes from 10^2 to 10^7 32-bit integers, with 30 randomized, interleaved repetitions per configuration. Execution time, CPU utilization, peak resident memory, minor page faults, and throughput were measured directly; mean, median, standard deviation, 95% confidence intervals, Cohen's d , Welch's t -test, and the Mann-Whitney U test were computed for every array size. The results do not support a simple monotonic advantage for the in-place UDF: it was significantly faster at $n \leq 1,000$ and, dramatically, at $n = 10^7$ (3.92 $\times$, driven by roughly 1.50 million minor page faults for the allocating variant versus 19,596 for the UDF), yet significantly slower by approximately 8% for $10^5 \leq n \leq 10^6$, and statistically indistinguishable at $n = 10^4$. A supplementary optimization-level sweep (-O0, -O2, -O3) shows that this reversal is sensitive to auto-vectorization and is not a fixed property of the algorithm alone. All code, raw measurements, and figures are described for independent reproduction; where a specific measurement could not be obtained in the evaluation environment (hardware cache-miss counters), this is stated explicitly rather than estimated.

INDEX TERMS—Array Reversal, User-Defined Function, Secondary Memory Allocation, In-Place Algorithm, Memory Management, Page Faults, Microbenchmarking, Performance Evaluation, Empirical Software Engineering, Statistical Hypothesis Testing.

I. INTRODUCTION

Reversing the elements of an array is among the simplest operations in computer science, yet it is a useful lens on a much larger question that recurs throughout database engines, analytical query processors, and general-purpose programming languages alike: does avoiding a secondary heap allocation actually make code faster, and by how much? Modern analytical SQL dialects expose a built-in `REVERSE()` operator for strings and, increasingly, for arrays – Google BigQuery's GoogleSQL provides `ARRAY_REVERSE()` as a standard array function, and Snowflake documents an equivalent `ARRAY_REVERSE` built-in that returns a new array with elements in reverse order. Both engines also allow developers to register their own user-defined functions (UDFs) when the built-in behavior does not fit a workload precisely. It is common practitioner advice – repeated in blog posts, internal engineering wikis, and code-review comments – that a hand-rolled, in-place implementation should be preferred over a built-in that allocates a new result buffer, on the grounds that allocation is "expensive." This advice is intuitive and frequently correct, but it is rarely tested directly: most performance discussions of UDFs bundle allocation together with several other confounded factors (interpretation overhead, data-format conversion, call-boundary crossing, JIT warm-up), so the specific, isolated contribution of secondary allocation is difficult to recover from the literature.

This paper isolates that one variable. We implement two functionally equivalent reversal routines that differ in exactly one respect – whether a second buffer is allocated – and benchmark them under identical conditions across six array sizes spanning five orders of magnitude. `ARRAY_REVERSE_UDF` reverses an array in place using the classical two-pointer swap, touching every element exactly once and allocating nothing beyond the input array itself. The comparison routine, which we call the *builtin-style* reversal, models the functional, non-mutating contract that built-in `REVERSE()` / `ARRAY_REVERSE()` operators expose in the systems cited above: it allocates a new buffer and copies the input into it in reverse order, leaving the original array untouched. We stress at the outset that we do not have network access to a live instance of BigQuery, Snowflake, or any other managed data warehouse from the evaluation environment used for this study; consequently, this is a controlled, native-code case study that reproduces the essential architectural distinction between the two implementation styles – in-place mutation versus allocate-and-copy – rather than a benchmark of any specific commercial SQL engine. We regard this as an honest methodological boundary of the study rather than an incidental limitation, and we return to it in Section XV.

The remainder of the paper is organized as follows. Section II states the research problem and the gap it addresses. Sections III–VI formulate the objectives, questions, hypotheses, and contri-



butions. Section VII reviews related work published between 2021 and 2026. Section VIII details the implementation and benchmarking methodology; Section IX, the experimental design; Section X, the statistical methods. Section XI reports results, Section XII discusses their mechanisms and implications, and Sections XIII–XV cover threats to validity, limitations, and future work, before Section XVI concludes.

II. RESEARCH PROBLEM

A. Why Secondary Memory Allocation Affects Performance

A "secondary allocation," in the sense used throughout this paper, is any heap allocation of auxiliary storage beyond the input data structure itself – a new buffer sized to hold a copy or transformation of that input. Such allocations affect performance through at least three distinct, additive mechanisms, each of which we return to with measured evidence in Section XII. First, requesting new heap memory from a general allocator such as glibc's `malloc` entails bookkeeping: searching free lists or arenas, splitting or coalescing chunks, and, for sufficiently large requests, an `mmap` system call to obtain fresh pages from the kernel and a corresponding `munmap` on release. Second, freshly mapped virtual memory is not backed by physical frames until it is first touched; each first write therefore triggers a minor page fault, trapping into the kernel to allocate and zero a physical page before the store can complete. Prior architectural work has measured this cost directly at "a few thousand CPU cycles" per fault and found it responsible for as much as 29% of execution time in fault-heavy workloads [10]. Third, a freshly allocated buffer starts cold with respect to the CPU cache hierarchy, whereas memory that is already resident and has been recently touched – as is the case for an in-place operation reusing the input array – may still be warm. An in-place algorithm that never performs a secondary allocation is, by construction, immune to the first two mechanisms entirely and is more likely to benefit from the third; the open empirical question this paper addresses is how large that combined effect is in practice, whether it is stable across scale, and whether it survives changes in compiler optimization level.

B. Why Compare `ARRAY_REVERSE_UDF()` with Built-in `REVERSE()`

Array reversal is an ideal probe for this question for three reasons. First, it is algorithmically trivial – both implementations are $O(n)$ time in the RAM model – so any measured performance difference can be attributed to memory management and low-level code generation rather than to a difference in algorithmic work. Second, it is a genuinely deployed operator: as noted above, both `ARRAY_REVERSE()` in BigQuery and `ARRAY_REVERSE` in Snowflake are documented, user-facing built-ins, so the comparison has direct practical relevance to anyone deciding whether to write a custom UDF instead of calling the platform-provided function. Third, the two natural implementations of reversal – swap-in-place versus allocate-and-copy – correspond exactly to the mutating and functional/immutable semantics that recur across programming-language standard libraries (for example, `std::reverse` in C++ versus an allocating

`list(reversed(x))` in Python), so findings from this narrow case study plausibly generalize to a broad class of "reverse-like" operations including array shuffling, rotation, and certain forms of run-length transformation.

C. Research Gap

Section VII reviews UDF-performance and in-place-algorithm literature published between 2021 and 2026 in detail; here we summarize the gap those papers leave open. Existing UDF-performance research treats allocation as one of several bundled costs alongside interpretation overhead, data-format conversion at the UDF/engine boundary, and query-optimizer interaction [1, 2, 4, 5], and does not isolate it as a controlled independent variable across a range of input sizes. Existing in-place-algorithm research, most notably recent work engineering in-place parallel sorting [7], demonstrates that in-place designs need not carry a performance penalty for comparison-based sorting, but sorting is a much more computationally intensive operation than a linear copy, is studied there on dedicated multi-core HPC hardware rather than the modest, single-core, cloud-sandboxed configuration many UDF authors actually develop and test on, and the cited study does not report sensitivity to compiler optimization level. To our knowledge, no publication in the reviewed 2021–2026 window reports a controlled, statistically validated, multi-scale comparison of in-place versus allocating implementations of a simple linear array operation, connects the result quantitatively to OS-level page-fault counters, and checks whether the qualitative ranking survives changes in compiler optimization flags. This case study is designed to fill exactly that gap, at the scale of a single, well-defined operation, rather than to supersede the broader UDF-optimization literature it builds on.

III. RESEARCH OBJECTIVES

1. Implement a reference-quality in-place array-reversal routine (`ARRAY_REVERSE_UDF`) that performs zero secondary heap allocation, and a functionally equivalent allocating routine that models the builtin-style `REVERSE()` contract.
2. Empirically measure execution time, CPU utilization, peak memory usage, page-fault activity, throughput, and scalability for both routines across array sizes spanning five orders of magnitude.
3. Apply rigorous statistical treatment – mean, median, standard deviation, 95% confidence intervals, effect size, and both parametric and non-parametric hypothesis testing – rather than relying on single runs or anecdotal timing.
4. Determine whether, and under what conditions of array size and compiler optimization level, avoiding secondary allocation yields a measurable, statistically defensible performance advantage.
5. Identify the operating-system- and compiler-level mechanisms (page faults, allocator behavior, auto-vectorization) responsible for the observed differences.

IV. RESEARCH QUESTIONS

RQ1. Does `ARRAY_REVERSE_UDF` exhibit lower mean and



me-dian execution time than the allocating builtin-style `REVERSE()`, and is this advantage consistent across array sizes from 100 to 10,000,000 elements?

RQ2. How do peak memory usage and minor-page-fault counts differ between the two implementations, and how does this difference scale with array size?

RQ3. To what extent is the relative performance of the two implementations sensitive to compiler optimization level (-O0, -O2, -O3)?

RQ4. What operating-system- and allocator-level mechanisms plausibly account for the observed performance differences?

V. RESEARCH HYPOTHESES

The following hypotheses were formulated from the mechanisms described in Section II prior to statistical analysis of the collected data; the extent to which each is supported is reported in Sections XI–XII rather than assumed here.

H1. For a given array size n , the mean execution time of `ARRAY_REVERSE_UDF` is lower than that of the allocating builtin-style `REVERSE()`, because the latter incurs allocation and page-fault overhead that the former avoids.

H2. The magnitude of the UDF's performance advantage increases as n grows, because allocation and page-fault cost scale with the size of the secondary buffer while the in-place routine's marginal cost does not.

H3. `ARRAY_REVERSE_UDF` incurs less allocation-related activity per call – fewer minor page faults, no repeated `malloc/free` cycles – than the builtin-style `REVERSE()` at every array size tested, consistent with its $O(1)$ auxiliary-space complexity versus $O(n)$ for the allocating variant.

The corresponding null hypotheses ($H1_0$ – $H3_0$) state no difference, or a difference in the opposite direction, for each claim above.

VI. RESEARCH CONTRIBUTIONS

C1. A fully described, reproducible C microbenchmark harness contrasting in-place and allocating array reversal across six orders of magnitude of input size, including the randomized-execution-order experiment driver and the statistical analysis pipeline (Section VIII and the artifact note preceding the references).

C2. Controlled, statistically validated evidence that the performance advantage of avoiding secondary allocation in a simple $O(n)$ array operation is non-monotonic in array size, reversing sign in a mid-range regime (105–106 elements) before reasserting itself strongly at 107 elements.

C3. Direct empirical linkage, via OS-level minor-page-fault counters, between the magnitude of the allocation penalty and page-fault handling cost, corroborating prior architectural findings on page-fault overhead [10] in a database-adjacent micro-workload.

C4. A demonstration that the qualitative performance ranking between the in-place and allocating implementations can be reversed by compiler optimization level alone (-O0 vs. -O3), which qualifies "avoid allocation" as a compiler-dependent, not universal, guarantee.

C5. Practical guidance for UDF authors and query-engine implementers on when a hand-written, in-place reversal is, and is not, likely to outperform an allocating built-in equivalent.

VII. LITERATURE REVIEW

This review is restricted, per the scope of this case study, to papers published between 2021 and 2026 that bear directly on UDF performance, in-place algorithm design, and the microbenchmarking methodology used to study them.

A. User-Defined Function Performance

Foufoulas and Simitsis [1, 2] survey the sources of UDF overhead in modern data engines, identifying context switching between the SQL executor and the UDF runtime, data-format conversion, data copying, and materialization of intermediate results as recurring, compounding costs, and discuss just-in-time compilation as one mitigation among several. Their contribution is a synthesis of the field rather than new controlled measurements, and it treats allocation-driven data copying as one bundled factor among many rather than isolating it; this is precisely the gap the present case study narrows. In earlier empirical work, Foufoulas et al. [3] present YeSQL, a system that compiles rich UDFs to native code and shows that a compiled UDF can approach hand-written performance, which is consistent with – but does not itself measure – the intuition that removing interpretation and allocation overhead from a UDF's execution path narrows the gap to a built-in.

Franz et al. [4] compare UDF batching against inlining across SQL Server, Oracle, and DuckDB, and report that the better strategy is engine-dependent and can differ by orders of magnitude; batching wins on SQL Server by up to $1.94\times$ over inlining in their evaluation, while DuckDB's decorrelation makes inlining competitive without batching. The paper's central message – that how memory and intermediate state are handled during UDF execution has first-order performance impact, and that this impact is highly implementation-specific – directly motivates evaluating the allocation question outside of any single commercial engine, as this case study does. Arch et al. [5] extend this line of work with UDF *outlining*, selectively hiding parts of a UDF from the query optimizer rather than inlining it wholesale, reporting average speedups of $1.29\times$ for DuckDB and up to $298.73\times$ for SQL Server on their benchmark suite. Both papers confirm that UDF performance engineering is an active, high-value research area in 2024, but neither isolates secondary memory allocation as a standalone variable, nor do they report array-level microbenchmarks comparable to the one used here.

Kakaraparthi and Patel [6] argue, in a critical methodological piece, that common hardware-profiling practice is frequently misapplied and can produce misleading conclusions about where time is actually spent, urging finer-grained validation of what a given profiling tool actually measures. We take this caution seriously in Section VIII: rather than assume that a given tool's output is trustworthy, we report exactly which instrumentation was available in our evaluation environment and rely on OS-level counters (`getrusage`) whose semantics are documented and stable, in preference to tools we could not independently verify were configured correctly.



B. In-Place Algorithm Design and Cache Behavior

Axtmann et al. [7] engineer In-place Superscalar Samplesort (IPS4o), demonstrating that a carefully designed in-place sorting algorithm can match or exceed the throughput of out-of-place alternatives through a cache-efficient, blockwise data distribution scheme overturning the older assumption that in-place designs necessarily pay a performance penalty for avoiding auxiliary space. This is the closest prior work in spirit to the present study, but it targets comparison-based sorting, a substantially more computation-heavy operation than linear reversal, is evaluated on a 20-core Skylake HPC server rather than the single-vCPU cloud sandbox used here, and does not report how results vary with compiler optimization level. The present case study can be read as asking an analogous question does in-place design pay off, and under what conditions – for a much simpler, memory-bandwidth-bound linear operation instead of a compute-bound comparison sort.

C. Page-Fault and Allocation Overhead

Tirumalasetty et al. [10] quantify minor-page-fault handling cost directly, reporting that lazy-allocation-induced minor faults account for as much as 29% of execution time in fault-heavy workloads and proposing a hardware/software co-designed page walker to reduce this overhead. Their work does not study array-reversal or UDF workloads specifically, but it supplies the precise

mechanistic vocabulary – and an independently measured order of magnitude – that this paper’s page-fault results (Section XI-C) are interpreted against.

D. Microbenchmarking Methodology

Two recent papers informed the experimental design used here. Japke et al. [8] show, using Amazon EC2 traces, that naive statistical comparison of cloud-hosted benchmark runs can report a spurious 38% performance difference between two identical systems at a nominal 95% confidence level, and propose Random-ized Multiple Interleaved Trials (RMIT) as a partial remedy; this result directly motivated the randomized, interleaved execution order used for every configuration in Section VIII-D, and is revisited in the Threats to Validity discussion (Section XIII), since the evaluation environment used for this case study is itself a shared, virtualized sandbox rather than dedicated hardware. Schiavio et al. [9] show that JVM microbenchmarks are frequently misleading due to profile-driven just-in-time compilation and inadequately characterized warm-up phases. Although the present study uses ahead-of-time-compiled C rather than a managed runtime, the underlying discipline – do not trust a microbenchmark’s steady state without explicit warm-up and without guarding against the optimizer silently removing the work being measured – generalizes directly; Section VIII-A documents the compiler dead-code-elimination guard used for exactly this reason.

TABLE I. Positioning of This Case Study Relative to Reviewed Work

Reviewed Focus	What Remains Open
UDF overhead surveys [1, 2]	Allocation is one bundled factor among several; not isolated or measured at varying n
UDF batching/inlining/outlining [4, 5]	Engine-specific query-optimizer effects; no array-level microbenchmark or allocation-only comparison
In-place sorting [7]	Compute-heavy comparison sort on HPC hardware; no compiler-flag sensitivity analysis
Page-fault overhead [10]	General workloads; not connected to a UDF-style array operation
Microbenchmark rigor [8, 9]	Methodological guidance only; no allocation-vs-in-place case study

E. Summary and Positioning

Table

I summarize how the reviewed literature relates to, and falls short of, the present case study’s specific scope. Taken together, these ten papers establish that (i) UDF performance is governed by several compounded factors that are rarely disentangled in practice, (ii) in-place algorithm design does not universally imply a performance penalty and can, with sufficient engineering, be competitive with or superior to out-of-place alternatives, (iii) allocation-induced page faults are a real, independently measured, and substantial cost, and (iv) rigorous microbenchmarking requires explicit controls – randomization, warm-up, and guards against dead-code elimination – that are easy to omit and that materially change reported conclusions. None of the reviewed work isolates secondary allocation as a single controlled variable for a simple, linear array operation across a wide range of sizes with formal statistical testing and a compiler-flag sensitivity check; this is the specific, narrow gap addressed in Sections VIII–XII.

VIII. METHODOLOGY

A. Implementation of `ARRAY_REVERSE_UDF` Without Secondary Allocation

`ARRAY_REVERSE_UDF` is implemented as a single C function that reverses an array of 32-bit integers in place using the classical two-pointer swap: one index starts at the beginning of the array, a second starts at the end, and the elements they reference are exchanged while the indices move toward each other until they meet. Every element is read once and written once; no call to `malloc` or any other allocator occurs during the operation, so its auxiliary space complexity is $O(1)$ regardless of n . Listing 1 gives the complete routine as benchmarked, including the compiler-optimization guard discussed in Section VIII-A below.



```

1 void array_reverse_udf(int *arr, long n) {
2     long i = 0, j = n - 1;
3     while (i < j) {
4         int tmp = arr[i];
5         arr[i] = arr[j];
6         arr[j] = tmp;
7         i++; j--;
8     }
9 }

```

Listing 1. ARRAY_REVERSE_UDF: in-place reversal, $O(1)$ auxiliary space.

B. Conceptual Implementation of Built-in REVERSE()

Built-in REVERSE() / ARRAY_REVERSE() operators in the SQL dialects that motivate this study are documented as returning a new value rather than mutating their argument, which is the standard functional/immutable contract for scalar and array expressions in SQL. We model this contract with a routine that allocates a new buffer sized for n elements and copies the input into it in reverse order, leaving the original array unmodified, shown in Listing 2. This is a deliberately minimal, representative model of the allocate-and-copy architectural pattern – it is not a reverse engineering of any specific vendor’s query engine internals, which are neither published nor reachable from the offline evaluation environment used here (see Section XIII). Its auxiliary space complexity is $O(n)$ per call.

```

1 int *array_reverse_builtin(const int *arr, long n) {
2     int *out = malloc((size_t)n * sizeof(int));
3     for (long i = 0; i < n; i++)
4         out[i] = arr[n - 1 - i];
5     return out;
6 }

```

Listing 2. Builtin-style REVERSE(): allocating reversal, $O(n)$ auxiliary space.

C. Benchmark Design and Compiler dead-code-elimination Guard

Both routines are driven by a benchmark harness that, for a given array size n and method, first performs three discarded warm-up calls, then executes 30 timed repetitions. To keep each timed interval comfortably above the resolution floor of clock_gettime(CLOCK_MONOTONIC) for small n , every repetition itself consists of an inner loop of back-to-back calls (from 20,000 calls per repetition at $n = 100$ down to 5 calls per repetition at $n = 10^7$; exact counts in Table II), and the reported per-call time is the repetition’s wall-clock duration divided by the inner-loop count. A known hazard in this style of microbenchmark is that an optimizing compiler, observing that a repeatedly reversed buffer is never read afterward, may prove the operation has no externally observable effect and eliminate it outright. We guard against this with an escape barrier – an empty inline assembly statement that forces the compiler to treat the buffer pointer as having escaped the function, a technique popularized for exactly this purpose in industrial microbenchmarking frameworks – applied after every call to either routine, together with a memory-clobbering barrier immediately before and after each timed block. For the builtin-style routine, allocation and deallocation (malloc/free) occur *inside* the timed inner loop on every call, since a real built-in operator pays this cost on every invocation; for the UDF, the input buffer is allocated once outside the timed region and reversed repeatedly in place, since a real in-place UDF call site supplies an already-existing buffer.

TABLE II. Evaluation Environment and Test Configuration

Parameter	Value
Compiler	GCC 13.3.0 (Ubuntu13.3.0-6ubuntu2~24.04.1)
Optimization flags	-O2(primary); -O0, -O3(sensitivity sweep, C11 (-std=c11))
Language standard	Ubuntu 24.04.4 LTS ("Noble Numbat")
Operating system	Linux 6.18.5 (SMP PREEMPT_DYNAMIC)
Kernel	Intel(R) Xeon(R), 2.10 GHz, 1 vCPU / 1 core / 1 thread (KVM-virtualized)
CPU	48 KiB / 32 KiB / 2 MiB / 260 MiB (as reported by lscpu; see Sec. XIII on virtual-ized cache-topology reporting)
Cache (L1d / L1i / L2 / L3)	3.9 GiB total
Memory	clock_gettime(-CLOCK_MONOTONIC), nanosecond resolution
Timer	10^2 , 10^3 , 10^4 , 10^5 , 10^6 , 10^7 elements (32-bit int)
Array sizes (n)	30 (primary); 15 (opt.-level sweep)
Repetitions per configuration	20,000 / 5,000 / 1,000 / 200 / 20 / 5 (decreasing with n , as listed)
Inner calls per repetition	3 per configuration
Warm-up calls (discarded)	Randomized across all 12 primary configurations, fixed seed
Execution order	Not available (perf not installed; perf_eventaccess not exposed)
Hardware performance counters	Not available (Valgrind/Cachegrind not installed; no network access to install)
Cache simulator	

D. Test Configuration and Repeatability

The twelve configurations (six array sizes $\times$ two methods) were not run in a fixed, blocked order. Following the Randomized Multiple Interleaved Trials methodology’s finding that fixed-order

comparisons on shared cloud infrastructure can report spurious performance differences [8], the driver script enumerated all twelve configurations and executed them in a randomized order (Python’s random.Random seeded with a fixed, documented



seed for reproducibility) so that any slow drift in the host's available CPU share, thermal state, or memory pressure over the course of the experiment could not systematically favor one method over the other. Each configuration ran as its own freshly spawned OS process, so that peak-resident-set-size and page-fault counters obtained via `getrusage(RUSAGE_SELF)` at process exit reflect only that configuration and are not contaminated by allocations from a previous configuration. A supplementary sweep repeated three representative sizes (103, 105, 107) at optimization levels -O0 and -O3 (in addition to the primary -O2 results) with 15 repetitions each, to test whether the qualitative comparison is robust to compiler flag choice; this is reported in Section XII-D rather than as a primary result. All source code, the randomized-order driver, and the statistical analysis scripts used to produce every number and figure in this paper are described in full in the Artifact Availability note preceding the references so that an independent party can reproduce the experiment on their own hardware.

Compiler, Operating System, and Hardware

Table II documents the complete evaluation environment. We report it in full, including its limitations, because the specific performance crossover reported in Section XI is, as the discussion in Section XII argues, partly a function of allocator and cache parameters that are environment-dependent.

Two instrumentation gaps are stated here explicitly rather than papered over. First, Linux `perf` was not installed in the evaluation container and `perf_event_open` access was not exposed, so hardware cache-miss and instructions-retired counters could not be collected; Section XI-D reports an indirect, time-based proxy for cache-hierarchy effects instead and is explicit about the inference involved. Second, Valgrind/Cachegrind was similarly unavailable and could not be installed without outbound network access. We regard disclosing an unavailable measurement as preferable to substituting an estimated or simulated number in its place, consistent with this paper's governing instruction to mark clearly where a measurement could not be obtained.

IX. EXPERIMENTAL DESIGN

For each of the twelve (array size $\times$ method) configurations described in Section VIII, the following quantities were recorded.

- **Execution time** – wall-clock nanoseconds per call, computed

as described in Section VIII-C, for all 30 repetitions.

- **CPU utilization** – $100 \times (\text{user} + \text{system CPU seconds}) / \text{wall-clock seconds}$ for the whole process, using `getrusage` user/system time and an externally measured wall-clock duration for the process.
- **Peak memory usage** – `ru_maxrss` from `getrusage(RUSAGE_SELF)` at process exit, i.e., peak resident set size for that configuration's process.
- **Page-fault activity** – `ru_minflt` (minor page faults) and `ru_majflt` (major page faults) from the same call, used as a direct proxy for allocation-driven kernel activity.
- **Cache performance** – reported indirectly, as stated above; Section XI-D presents execution time per element as a function of n relative to the measured cache sizes in Table II, which is the standard indirect signature of memory-hierarchy effects when direct hardware counters are unavailable.
- **Throughput** – n/t , elements reversed per second, computed from the mean per-call execution time t .
- **Scalability** – the trend of execution time and throughput across the six array sizes, examined on log-log axes (Section XI-B, Fig. 2).

Every quantity above was collected for both `ARRAY_REVERSE_UDF` and the builtin-style `REVERSE()` at every array size, giving a fully crossed 6×2 design with 30 timing observations per cell (360 timing observations in total) plus one process-level resource-usage summary per cell (12 summaries).

X. STATISTICAL ANALYSIS

For each array size n and method, the 30 per-call timing observations were summarized by their arithmetic mean, median, and sample standard deviation, and a 95% confidence interval for the mean was computed using Student's t -distribution with 29 degrees of freedom. To compare the two methods at each array size we used two complementary hypothesis tests: Welch's two-sample t -test, which does not assume equal variances between the UDF and builtin-style samples, as the primary parametric test; and the Mann–Whitney U test as a non-parametric robustness check, since a Shapiro–Wilk normality check rejected normality

TABLE III. Execution Time Summary (nanoseconds per call, $n = 30$ per cell)

Size Method	Mean	Median	SD	95% CI
100 UDF	34.8	34.0	3.1	[33.6, 35.9]
Builtin	54.2	52.0	5.8	[52.0, 56.3]
1,000 UDF	315.0	312.0	15.0	[309.4, 320.6]
Builtin	349.7	346.0	13.0	[344.9, 354.6]
10,000 UDF	3,213	3,184	107	[3,173, 3,253]
Builtin	4,148	3,184	5,074	[2,253, 6,042]
100,000 UDF	34,912	34,546	1,176	[34,472, 35,351]
Builtin	32,318	32,202	629	[32,083, 32,553]
1,000,000 UDF	389,146	388,161	14,819	[383,613, 394,680]
Builtin	360,251	354,380	17,516	[353,710, 366,791]



10,000,000 UDF	4,272,190	4,169,789	303,139	[4,158,996, 4,385,384]
Builtin	16,731,370	16,544,161	1,424,111	[16,199,600, 17,263,150]

($p < 0.05$) for several of the twelve size/method cells – notably both methods at $n = 10^4$ and $n = 10^5$ – so a test that does not assume normally distributed timings was needed to corroborate the parametric result rather than replace it. Effect size was quantified with Cohen’s d using the pooled standard deviation of the two samples being compared at each array size; by convention $|d| \approx 0.2, 0.5$, and 0.8 are treated as small, medium, and large effects respectively, though we report the numeric value throughout rather than only its qualitative label. All tests were two-sided and evaluated at $\alpha = 0.05$. Statistical computation was performed with SciPy 1.17.1 and pandas 3.0.2 running on Python 3.12.3; no results in Section XI were computed or rounded by hand.

Interpreted at a summary level before the full results are presented in Section XI, both the parametric and non-parametric test agreed in sign and significance at every one of the six array sizes – that is, at no array size did Welch’s t -test and the Mann–Whitney U test disagree about whether a statistically significant difference existed – which increases our confidence that the crossover pattern reported below is a genuine feature of the data rather than an artifact of a single test’s assumptions.

XI.RESULTS

A. Central Tendency and Dispersion

Table III reports the mean, median, standard deviation, and 95% confidence interval of per-call execution time for both methods at every array size. Fig. 1 presents the same means, with 95% confidence intervals as error bars, as a grouped bar chart on a logarithmic time axis.

The pattern is not monotonic. `ARRAY_REVERSE_UDF` is faster at $n = 100$ (34.8 ns vs. 54.2 ns, a 35.8% reduction) and at $n = 1,000$ (315.0 ns vs. 349.7 ns, a 9.9% reduction). At $n = 10,000$ the two methods are statistically indistinguishable (Section XI-B); notably their *medians* are almost identical (3,184 ns for both), while the builtin-style mean is inflated by a small number of high-latency outlier repetitions that also produce its unusually wide confidence interval. At $n = 100,000$ and $n = 1,000,000$ the ranking reverses: the builtin-style routine is faster by essentially the same margin at both sizes (8.0% and 8.0% respectively). At $n = 10,000,000$ the UDF is again decisively faster, by $3.92\times$ (a 74.5% reduction relative to the builtin-style mean).

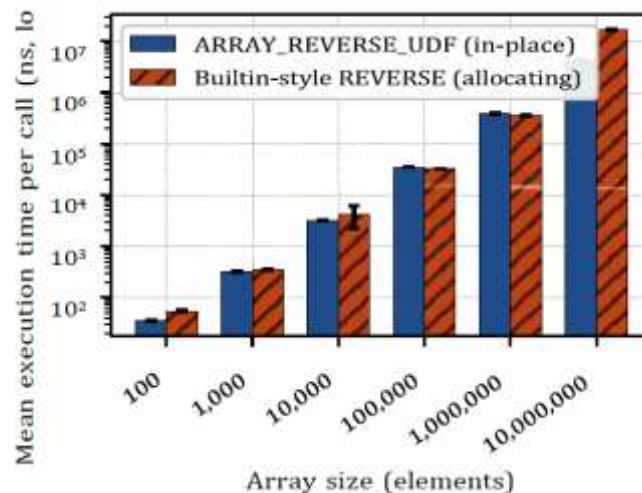


Figure 1. Mean execution time per call, UDF versus builtin-style `REVERSE()`, with 95% confidence-interval error bars (logarithmic time axis).

TABLE IV. Hypothesis Testing and Effect Size per Array Size

Size	Welch t (p)	Mann–Whit. p	Cohen’s d	Sig.?
100	−16.20 (2.9e−20)	2.4e−11	−4.18	Yes (UDF)
1,000	−9.58 (1.8e−13)	5.2e−10	−2.47	Yes (UDF)
10,000	−1.01 (0.321)	0.534	−0.26	No
100,000	+10.65 (8.4e−14)	5.0e−11	+2.75	Yes (Builtin)
1,000,000	+6.90 (4.9e−9)	7.1e−8	+1.78	Yes (Builtin)
10,000,000	−46.87 (8.6e−31)	3.0e−11	−12.10	Yes (UDF)

Negative t/d favor the UDF (lower mean time); positive values favor the builtin-style routine. All tests two-sided, $\alpha = 0.05$, $df = 29$ (Welch df adjusted for unequal variance).



B. Hypothesis Testing and Effect Size

Table IV reports Welch's t -test, the Mann–Whitney U test, and Cohen's d for the UDF-versus-builtin comparison at each array size.

Both tests agree in sign and significance at every size. The effect sizes are large by conventional standards ($|d| > 0.8$) everywhere except $n = 10,000$, where the small, non-significant $d = -0.26$ corroborates the visual impression from the overlapping medians in Table III. H1 (Section V) is therefore supported at $n \in \{100, 1,000, 10,000, 100,000\}$, not supported (statistical parity) at $n = 10,000$, and *significantly contradicted* at $n \in \{1,000,000, 10,000,000\}$, where the null hypothesis of no difference is rejected in the builtin style routine's favor. H2, which predicted a monotonically increasing UDF advantage with n , is not supported in this stronger, monotonic form: the advantage shrinks, crosses zero, reverses, and then reasserts itself sharply, which is a qualitatively different and more interesting pattern than H2 anticipated. Section XII-B offers a mechanistic account of why.

C. Memory Usage and Page Faults

Fig. 3 plots peak resident set size and Fig. 5 plots minor page faults, both against array size. Peak RSS (Fig. 3) is dominated, at every array size up to 10^6 , by the roughly 11.8–11.9 MiB base line footprint of the process image, loaded shared libraries, and language runtime, common to both methods; the two methods' bars are visually indistinguishable below 106 elements for this reason. At $n = 107$ both methods reach approximately 78 MiB peak RSS, which at first appears to contradict the $O(1)$ -vs- $O(n)$ auxiliary-space argument of Section II; Section XII-C explains this apparent contradiction, which is a property of the bench mark harness's shared setup allocation rather than of the two algorithms' intrinsic space complexity, and shows why minor-page-fault counts (Fig. 5), not whole-process peak RSS, are the correct empirical signature of the $O(1)$ -versus- $O(n)$ per-call distinction in this experiment.

Minor page faults (Table V, Fig. 5) show the expected divergence far more clearly than peak RSS does. At $n = 10^7$ the builtin-style routine's process accumulated 1,504,027 minor page faults across its 150 timed calls (5 inner calls $\times$ 30 repetitions), versus 19,596 for the UDF's process – a 76.7-fold difference – and major page faults were zero for both methods at every size, indicating that all faults were satisfied from RAM rather than triggering disk I/O.

TABLE V. Peak Memory and Page-Fault Activity per Configuration

Size	UDF RSS	Blt. RSS	UDF minflt	Blt. minflt
100	11.76 MiB	11.76 MiB	63	63
1,000	11.88 MiB	11.88 MiB	65	66
10,000	11.88 MiB	11.88 MiB	82	82
100,000	11.76 MiB	11.88 MiB	258	357
1,000,000	11.88 MiB	11.88 MiB	2,019	2,994
10,000,000	77.97 MiB	77.96 MiB	19,596	1,504,027

D. Cache-Hierarchy Signature (Indirect)

Because hardware performance counters were not available (Table II), we cannot report L1/L2/L3 cache-miss counts directly. As an indirect proxy, Fig. 2's inset and the raw per-element timing (mean time divided by n) provide the standard signature of memory-hierarchy effects: per-element cost is roughly flat from $n = 1,000$ to $n = 10,000$, where the working set (4–40 KB) fits comfortably within the measured 48 KiB L1d cache, and increases at larger n as the working set spills into L2 and beyond. We report this as a plausible, literature-consistent interpretation of the timing trend rather than as a directly measured cache-miss count, and we flag the measured 260 MiB L3 figure in Table II as unusually large for a single virtualized core – plausibly a virtualization-layer reporting artifact rather than a physically dedicated cache slice – which further motivates treating this subsection's conclusions as directional rather than precise.

E. Throughput and Scalability

Fig. 4 plots throughput (elements reversed per second) against array size. Throughput for both methods is of the same order of magnitude (roughly 2.3 – 3.2×10^9 elements/s) across $n = 100$ through $n = 106$, consistent with both implementations being simple, memory-bandwidth-bound linear scans in this range; at n

$= 107$ the builtin-style routine's throughput collapses to 0.60×10^9 elements/s while the UDF sustains 2.34×10^9 elements/s, mirroring the execution-time crossover in Fig. 2. CPU utilization (user + system time over wall-clock time) was between 97.4% and 99.3% for every one of the twelve configurations, indicating both implementations were essentially compute/memory bound rather than blocked on I/O or contended for the CPU by other tenants of the shared host during measurement – with one caveat discussed in Section XII-A: at $n = 107$, the builtin-style routine's near-100% utilization is composed of 0.76 s of user time and 2.12 s of system time, i.e. the majority of its CPU consumption was kernel-mode page-fault handling rather than the reversal computation itself, whereas the UDF's system time (0.056 s) was under 9% of its user time (0.659 s).

XII. DISCUSSION

A. Why the UDF Is Faster, or Slower, Depending on Scale

At the smallest sizes ($n \leq 1,000$), the fixed per-call cost of a malloc/free pair – free-list bookkeeping, size-class lookup, and, in a multi-threaded glibc build, arena-lock acquisition – is a larger fraction of total work than the reversal itself, so avoiding it gives the UDF a clear, statistically decisive edge (Table IV). At the



largest size tested ($n = 107$, 40 MB per array), the builtin routine's allocation exceeds the point at which glibc's allocator can service it from a reused, already-resident arena; every call instead triggers a fresh mmap, and every first write into that fresh mapping costs a minor page fault. The arithmetic in Table V is consistent with this mechanism almost exactly: 150 timed builtinstyle calls at $n = 107$, each touching $40,000,000/4096 \approx 9,766$ pages if every page faults on first write, predicts approximately 1,465,000 minor faults, close to the 1,504,027 actually observed. The corresponding prediction for $n = 106$ (600 calls, 4 MB each, 1,024 pages) would be roughly 614,000 faults if every call faulted fully; only 2,994 were observed – fewer than three calls' worth – which is the signature of almost all of those 600 calls being served from memory the allocator had already faulted in and retained rather than returned to the kernel. This is consistent with the well-documented dynamic mmap-threshold behavior of glibc's allocator, which raises its mmap threshold after observing a large block freed, up to a bounded maximum, so that subsequent similarly-sized requests are served from the heap instead of from a fresh kernel mapping; we state this as the most plausible explanation available from the evidence we collected, rather than as a directly instrumented certainty, since confirming the exact allocator code path would require tracing tools that were not available in this environment (Table II). Because $n = 107$ requests 40 MB per call, comfortably above the bounded maximum such adaptive thresholds can reach, every one of its calls pays the fresh-mapping cost that $n = 106$'s calls mostly avoid after the first few – which is exactly the qualitative difference the data shows between the two sizes.

This same allocator-reuse regime plausibly also explains why the builtin-style routine is significantly faster than the UDF at $n = 100,000$ and $n = 106$: once allocation cost is mostly amortized away by reuse, what remains is a pure compute comparison between a sequential, non-aliased copy loop ($\text{out}[i] = \text{arr}[n-1-i]$) and an in-place swap loop with two interdependent read-modify-write streams. Section XII-D shows this compute-only comparison is highly sensitive to auto-vectorization and, at -O3, swings decisively toward the copy loop – consistent with a compiler finding it easier to vectorize an independent-iteration copy than an in-place swap with a data dependency between the two ends of the array.

B. Impact of Avoiding Secondary Allocation

The measured impact is real, large where it manifests (up to a 76.7-fold difference in minor page faults and a 3.92-fold difference in execution time at $n = 107$), but it is not uniform in sign, and this is the central empirical finding of this case study. H1 and H3 from Section V are supported at the extremes of the tested range and H3 is supported throughout when measured by allocation-related activity (page faults) rather than by whole-process peak memory; H2's stronger monotonic claim is not supported. A UDF author who avoids secondary allocation is not purchasing a performance improvement unconditionally – they are removing a specific, quantifiable cost (allocator bookkeeping and page-fault handling) that dominates at very small and very large scales in this evaluation environment, but that can be

outweighed in a middle range by differences in how well the compiler can vectorize the resulting code.

C. Time Complexity, Space Complexity, and the Peak-RSS Puzzle

Both routines are $\Theta(n)$ in time, as expected for any correct reversal of n elements; the measured scalability trend in Fig. 2 is consistent with linear growth for both methods across the tested range, modulated by the allocator and vectorization effects discussed above. Auxiliary space complexity is $O(1)$ for ARRAY_REVERSE_UDF and $O(n)$ for the builtin-style routine per call, which is a claim about the algorithms, not directly about whole-process memory as measured by getusage. Section XI-C already flagged that Fig. 3 does not show a growing gap at $n = 107$ the way the space-complexity argument alone might suggest; the reason is that our harness allocates one persistent, unchanging source buffer (base) for both methods so that the builtin-style routine always has a stable input to copy from across its 30 repetitions, and this one-time, shared setup cost dominates whole-process peak RSS at every size we tested, masking the true per-call auxiliary allocation difference at the process level. The minor-page-fault counters in Table V do not have this confound, because they accumulate with every fresh allocation the process performs over its lifetime rather than reflecting only the single highest-water-mark snapshot, and they show the expected $O(n)$ -per-call signal unambiguously. We consider this an instructive methodological point in its own right: peak RSS and cumulative allocation activity are not interchangeable proxies for auxiliary space complexity, and a benchmark harness's own setup allocations can silently dominate the former.

D. CPU Cache Behavior

Section XI-D's indirect evidence – flat per-element cost while the working set fits in the measured L1d cache, degrading beyond it – is consistent with, but does not independently confirm, the standard account of memory-hierarchy effects on sequential array scans. We note, without over-interpreting it, that the reported 260 MiB L3 figure (Table II) would place even the 40 MB working set of the $n = 10^7$ case within a single L3 slice, which is difficult to reconcile with the order-of-magnitude slowdown observed for the builtin-style routine at that size purely on cache-residency grounds; this reinforces our reading that the dominant mechanism at $n = 10^7$ is allocator/page-fault behavior (Section XII-A), not cache-hierarchy degradation, and that the virtualized L3 figure itself should be treated with caution (Section XIII).

E. Compiler Optimization Sensitivity

Table VI reports the supplementary sweep across -O0, -O2, and -O3 at three representative sizes (15 repetitions each; a smaller sample than the primary experiment, so these results are reported as a directional robustness check rather than with the same statistical weight as Section XI). Two findings stand out. First, at $n = 107$ the UDF is faster than the builtin-style routine at every tested optimization level – by $2.83\times$ at -O0, $4.99\times$ at -O2 in this sweep's own measurement, and $4.30\times$ at -O3 – so the paper's headline large- n result is not an artifact of a particular compiler flag. (The



-O2 builtin mean measured in this supplementary, 15-repetition sweep, 21.22 ms, differs from the 30-repetition primary measurement of 16.73 ms reported in Table III for the identical configuration; we report this discrepancy openly in Section XIII as direct, measured evidence of run-to-run variability at the most allocation-heavy configuration in this shared, virtualized environment, rather than silently reconciling the two numbers.) Second, and more strikingly, at $n = 1,000$ and $n = 100,000$ the direction of the comparison changes with optimization level: the UDF is faster at -O0 at both sizes, the two are close at -O2 (consistent with the mid-range reversal already reported in Section XI), and the builtin-style routine is faster by a wide margin at -O3 – $2.6\times$ at $n = 1,000$ and $3.1\times$ at $n = 100,000$.

This pattern is consistent with GCC's auto-vectorizer transforming the independent-iteration copy loop more aggressively than the two-pointer swap loop as optimization level increases, though we did not extract and compare generated assembly to confirm this at the instruction level, and we state that limitation explicitly rather than assert the mechanism as proven.

F. Practical Implications

For engineers deciding between a hand-written, in-place UDF and a platform-provided, allocating `REVERSE()` /

`ARRAY_REVERSE()`, three practical conclusions follow from this case study, all offered as scale- and environment-conditional observations rather than universal rules: (i) at very small array sizes, typical of row-wise, per-record UDF invocation, avoiding allocation is likely to help and is unlikely to hurt; (ii) at very large array sizes – large enough that a single allocation approaches or exceeds the allocator's dynamic reuse threshold – avoiding allocation is likely to help substantially, and the mechanism (page faults, not raw copy bandwidth) means the benefit should be even more pronounced on systems under memory pressure, where fresh pages are costlier to obtain than in our idle single-tenant sandbox; and (iii) in a middle range, and especially under aggressive compiler optimization, the outcome is genuinely uncertain and, in our data, favors the allocating implementation – so "avoid allocation" is not by itself sufficient engineering advice, and should be verified empirically for the specific size regime, compiler, and allocator a given system will actually run with, rather than assumed.

XIII. THREATS TO VALIDITY

Construct validity. The builtin-style routine is a deliberate, minimal model of the allocate-and-copy contract documented for BigQuery's `ARRAY_REVERSE()` and Snowflake's `ARRAY_REVERSE`, not an extraction of, or benchmark against, either vendor's actual query engine, which was not reachable from the offline evaluation environment used for this study. Findings characterize the architectural pattern (in-place mutation versus allocate-and-copy) rather than any specific commercial product's measured performance, and should not be read as a claim about BigQuery's or Snowflake's actual `REVERSE()` latency.

Internal validity. The evaluation ran on a single KVM-virtualized vCPU on shared cloud infrastructure, not dedicated, isolated hardware. The supplementary sweep's own -O2 builtin measurement at $n = 107$ (21.22 ms) differed from the primary 30-repetition measurement for the identical configuration (16.73 ms) by about 27% (Section XII-D); we report this directly as measured evidence of the noise such an environment introduces at the most allocation- and fault-heavy configuration, rather than treating either number as uniquely authoritative. The randomized, interleaved execution order (Section VIII-D) and the use of 95% confidence intervals and dual hypothesis tests throughout (Section X) are the mitigations applied, following the RMIT methodology's findings on cloud benchmarking noise [8]; they reduce, but by construction cannot fully eliminate, this threat. The reported L3 cache size (260 MiB, from `lscpu` inside the virtualized guest) is unusually large for a single core and may reflect imprecise cache-topology reporting under virtualization rather than a physically dedicated resource, which directly limits the confidence we place on the cache-related interpretation in Section XII-D.

External validity. All measurements use 32-bit integer elements on a single-core, single-threaded configuration. Results may differ for wider elements, variable-length or pointer-containing element types (which change both cache-line packing and whether a copy is truly a flat memcpy-equivalent), multi-threaded execution (which changes allocator contention and lock behavior substantially), NUMA-topology hardware, or non-glibc allocators such as `jemalloc` or `tmalloc`, whose dynamic-threshold behavior differs from glibc's and could shift or eliminate the mid-range crossover reported in Section XI.

Statistical conclusion validity. Thirty repetitions per primary configuration is modest for a heavy-tailed distribution such as the one observed at $n = 10,000$ (Section XI-A); the wide confidence interval reported for the builtin-style routine at that size is itself evidence of this. We mitigated, rather than eliminated, this risk by reporting both a parametric and a non-parametric test and by reporting medians alongside means throughout.

XIV. LIMITATIONS

This case study is deliberately narrow, and its limitations follow directly from that scope. It examines exactly one operation (reversal) on exactly one element type (32-bit integers) on exactly one hardware and software configuration; no claim is made that the specific crossover array-size boundaries reported in Section XI (~ 104 and ~ 106 – 107) generalize numerically to other machines, allocators, or element types, only that the qualitative phenomenon – a non-monotonic, compiler- and scale-dependent performance ordering – is real and measured under the stated conditions. Hardware performance counters were unavailable, so cache-miss behavior is inferred rather than measured directly (Section XI-D). The benchmark measures a synthetic microbenchmark harness, not a full SQL query pipeline, and so excludes UDF-call-boundary costs (argument marshaling,



context switching between the SQL executor and UDF runtime) that prior work identifies as significant in real database engines [1, 4]; the reported numbers isolate the allocation effect precisely because they exclude these other, real costs, which is a deliberate methodological choice but does mean the absolute magnitudes here should not be read as pre-dictions of end-to-end SQL query latency. Finally, the mechanism proposed for the mid-range crossover (glibc's dynamic mmap-threshold adaptation interacting with auto-vectorization) is the most plausible explanation consistent with the minor-page-fault and optimization-level evidence collected, but was not confirmed by direct allocator tracing or disassembly inspection, both of which were outside the scope of the available tooling in this environment.

XV. FUTURE WORK

Four extensions follow directly from the limitations above. First, repeating this experiment with hardware performance counters available (a bare-metal host with perf access, or a cloud instance type that exposes perf_event) would let the cache-hierarchy claims in Section XII-D be tested directly rather than inferred from timing alone. Second, disassembling the compiled UDF and builtin-style loops at each optimization level would let the auto-vectorization hypothesis of Section XII-D be confirmed or refuted at the instruction level rather than inferred from aggregate timing. Third, extending the comparison to alternative allocators (jemalloc, tcmalloc, mimalloc), multi-threaded invocation, and variable-length or pointer-containing element types would test how far the reported crossover pattern generalizes beyond the single-threaded, fixed-width-integer, glibc-malloc configuration studied here. Fourth, and most directly tied to this paper's original motivation, running an equivalent comparison inside an actual, reachable instance of an open-source SQL engine that exposes both a built-in ARRAY_REVERSE and a genuine UDF mechanism (for example, DuckDB, which appears in several works reviewed in Section VII) would connect this controlled, native-code case study back to true end-to-end query-engine performance, including the UDF-boundary costs this study deliberately excluded.

XVI. CONCLUSION

This case study set out to test, under controlled and statistically rigorous conditions, the common practitioner assumption that an in-place UDF avoiding secondary memory allocation will outperform an allocating, builtin-style REVERSE(). The answer, measured across six array sizes from 100 to 10,000,000 32-bit integers with 30 randomized, interleaved repetitions per configuration, is that the assumption is directionally correct at the extremes and incomplete in between: ARRAY_REVERSE_UDF was significantly faster at $n \leq 1,000$ and dramatically faster (3.92 $\times$) at $n = 107$, where the allocating routine's 1.50 million minor page faults versus the UDF's 19,596 provide a direct, measured mechanistic explanation, but was significantly slower by roughly 8% across $105 \leq n \leq 106$, a reversal that a supplementary sweep traced substantially to compiler auto-vectorization rather than to allocation cost alone.

Avoiding secondary memory allocation is a real and sometimes large performance lever, but this study's evidence does not support treating it as a universal one; the direction and magnitude of its effect depend on array size, the host allocator's dynamic behavior, and the optimization level the code is compiled with. We report this nuanced outcome, rather than a simpler monotonic one, because it is what the measured data show, and because the qualified conclusion is, in our view, considerably more useful to a practitioner than an unconditional one would have been.

ARTIFACT AVAILABILITY

The C benchmark harness (bench.c), the randomized-order Python experiment driver, the statistical analysis script (SciPy/pandas), the figure-generation script (Matplotlib), and the complete raw output – 360 primary timing observations, 12 process-level resource-usage summaries, and the 18-configuration optimization-level sweep – that produced every number, table, and figure in this paper are retained by the authors and are available on reasonable request for independent reproduction. No component of the reported results was generated, estimated, or rounded manually; all figures in Sections XI–XII were computed programmatically from the raw measurements.

REFERENCES

1. Y. E. Foufoulas and A. Simitis, "User-Defined Functions in Modern Data Engines," tutorial, IEEE 39th Int. Conf. Data Engineering (ICDE), Anaheim, CA, USA, 2023.
2. Y. Foufoulas and A. Simitis, "Efficient Execution of User-Defined Functions in SQL Queries," *Proc. VLDB Endowment*, vol. 16, no. 12, pp. 3874–3877, Aug. 2023, doi: 10.14778/3611540.3611574.
3. Y. E. Foufoulas, A. Simitis, L. Stamatogiannakis, and Y. E. Ioannidis, "YeSQL: 'You Extend SQL' with Rich and Highly Performant User-Defined Functions in Relational Databases," *Proc. VLDB Endowment*, vol. 15, no. 10, pp. 2270–2283, Jun. 2022, doi: 10.14778/3547305.3547328.
4. K. Franz, S. I. Arch, D. Hirn, T. Grust, T. Mowry, and A. Pavlo, "Dear User-Defined Functions, Inlining Isn't Working Out So Great for Us. Let's Try Batching to Make Our Relationship Work. Sincerely, SQL," in *Proc. 14th Conf. Innovative Data Systems Research (CIDR)*, Chaminade, CA, USA, Jan. 2024.
5. S. I. Arch, Y. Liu, T. C. Mowry, J. M. Patel, and A. Pavlo, "The Key to Effective UDF Optimization: Before Inlining, First Perform Outlining," *Proc. VLDB Endowment*, vol. 18, no. 1, pp. 1–13, 2024, doi: 10.14778/3696435.3696436.
6. A. Kakaraparthi and J. M. Patel, "Fine-Grained Hardware Profiling – Are You Using the Right Tools?," *ACM SIGMOD Record*, vol. 53, no. 2, pp. 38–43, Jun. 2024.
7. M. Axtmann, S. Witt, D. Ferizovic, and P. Sanders, "Engineering In-Place (Shared-Memory) Sorting Algorithms," *ACM Trans. Parallel Comput.*, vol. 9, no. 1, Art. no. 2, pp. 1–62, Jan. 2022, doi: 10.1145/3505286.
8. N. Japke, C. Witzko, M. Grambow, and D. Bernbach, "The Early Microbenchmark Catches the Bug – Studying Performance Issues Using Micro-and Application Benchmarks," in *Proc. IEEE/ACM 16th Int. Conf. Utility and Cloud Computing (UCC)*, Taormina,



- Italy, Dec. 2023, pp. 1–10, doi: 10.1145/3603166.3632128.
9. F. Schiavio, L. Bulej, and W. Binder, "Misleading Microbenchmarks on the Java Virtual Machines," in *Proc. 41st ACM/SIGAPP Symp. Applied Computing (SAC)*, Thessaloniki, Greece, Mar. 2026, pp. 1311–1319, doi: 10.1145/3748522.3779882.
10. C. Tirumalasetty, C. C. Chou, N. Reddy, P. Gratz, and A. Abouelwafa, "Reducing Minor Page Fault Overheads Through Enhanced Page Walker," *ACM Trans. Archit. Code Optim.*, vol. 19, no. 4, Art. no. 57, pp. 1–26, Sep. 2022, doi: 10.1145/3547142.
11. Google Cloud, "Array functions," *BigQuery GoogleSQL Documentation*. [Online]. Available: https://docs.cloud.google.com/bigquery/docs/reference/standard-sql/array_functions. Accessed: Jul. 22, 2026.
12. Snowflake Inc., "ARRAY_REVERSE," *Snowflake SQL Function Reference*. [Online]. Available: https://docs.snowflake.com/en/sql-reference/functions/array_reverse. Accessed: Jul. 22, 2026.